
SoftPython

Introductory guide to data cleaning and analysis with Python 3

David Leoni, Alessio Zamboni, Marco Caresia

Sep 28, 2020

Copyright © 2020 by David Leoni, Alessio Zamboni, Marco Caresia.

SoftPython is available under the Creative Commons Attribution 4.0 International License, granting you the right to copy, redistribute, modify, and sell it, so long as you attribute the original to David Leoni, Alessio Zamboni, Marco Caresia and identify any changes that you have made. Full terms of the license are available at:

<http://creativecommons.org/licenses/by/4.0/>

The complete book can be found online for free at:

<https://en.softpython.org>

CONTENTS

	Prefazione	1
	News	1
1	Panoramica	3
1.1	Intended audience	3
1.2	Contents	3
1.3	Authors	4
1.4	License	4
1.5	Acknowledgments	5
2	Overview	7
2.1	Chapters	7
2.2	Why Python?	8
2.3	References	9
2.4	Approach and goals	10
2.5	Doesn't work, what should I do?	10
2.6	Installation and tools	11
2.7	Let's start !	11
3	Installation	13
3.1	Installing Python	13
3.2	Installing packages	17
3.3	Jupyter Notebook	17
3.4	Projects with virtual environments	21
3.5	Further readings	23
4	A - Foundations	25
4.1	Tools and scripts	25
4.2	Python basics	43
4.3	Strings 1 - introduction	82
4.4	Strings 2 - operators	99
4.5	Strings 3 - methods	118
4.6	Error handling and testing solutions	135
4.7	Commandments	153
5		159
6	Index	161

Prefazione

Introductory 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.org¹

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](https://www.python.org)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>

PANORAMICA

1.1 Intended audience

This book can be useful for both novices who never really programmed before, and for students with more technical background, who have a desire to know 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. *Installation*
2. *Tools and scripts*
3. *Basics* 4.

Strings

1. *introduction*
2. *operators*
3. *methods*
5. *Error handling and testing*

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.it³

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

Alessio Zamboni (2018 March Edition assistant @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 technilologies 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.

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.



UNIVERSITY
OF TRENTO
Department of Information
Engineering and Computer Science



All the material in this website is distributed with license CC-BY 4.0 International Attribution <https://creativecommons.org/licenses/by/4.0/deed.en>

Basically, you can freely redistribute and modify the content, just remember to cite University of Trento and [the authors](#)⁷

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

³ <https://davidleoni.it>

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>

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.it¹⁰) for the numerous datasets provided.



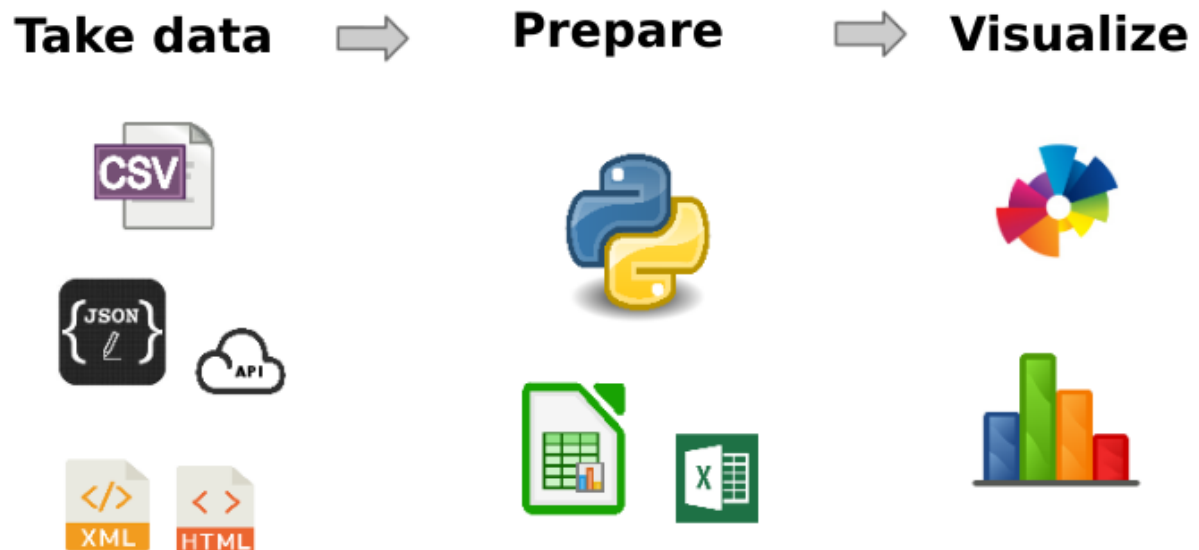
Other numerous intitutions and companies that in over time contributed material and ideas are cited [in this page](#)

¹⁰ <https://dati.trentino.it>

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.

WHAT ARE WE GOING TO DO?



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
 - [Tiobe¹¹](https://www.tiobe.com/tiobe-index/) Index
 - [popularity on Github¹²](https://madnight.github.io/github/#/pull_requests/2020/1)
- **Licence** open source & business friendly¹³
 - 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/github/#/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 mysterious 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 Loop³⁵
 - Infinite Recursion³⁶
 - Flow of Execution³⁷

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#sec235>

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 exception³⁸
 - I added so many print statements I get inundated with output³⁹
- Semantic errors⁴⁰
 - My program doesn't work⁴¹
 - we got a big hairy expression and it doesn't do what I expect.⁴²
 - we 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>

INSTALLATION

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

3.1 Installing Python

There are various 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](https://repl.it)⁴⁹

Another useful resource is [Python Tutor](http://pythontutor.com/visualize.html#py=3)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](http://try.jupyter.org)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 already many programs, it might be you already have installed Python without even knowing it, maybe in version 2, but we need the 3! Overlaying 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 3.8⁵²](https://www.anaconda.com/download/#windows) 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 see 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](https://brew.sh/)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*. From 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](https://diveintopython3.problemsolving.io/installing-python.html)⁵⁴ in particular by going to the subsection [installing in Ubuntu Linux](https://diveintopython3.problemsolving.io/installing-python.html#ubuntu)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](https://diveintopython3.problemsolving.io/installing-python.html#other)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 insert 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 Terminal and 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 directly 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 Jupyter⁵⁷:

- 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 interpreter 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, preferably 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 yours 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 than 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 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:

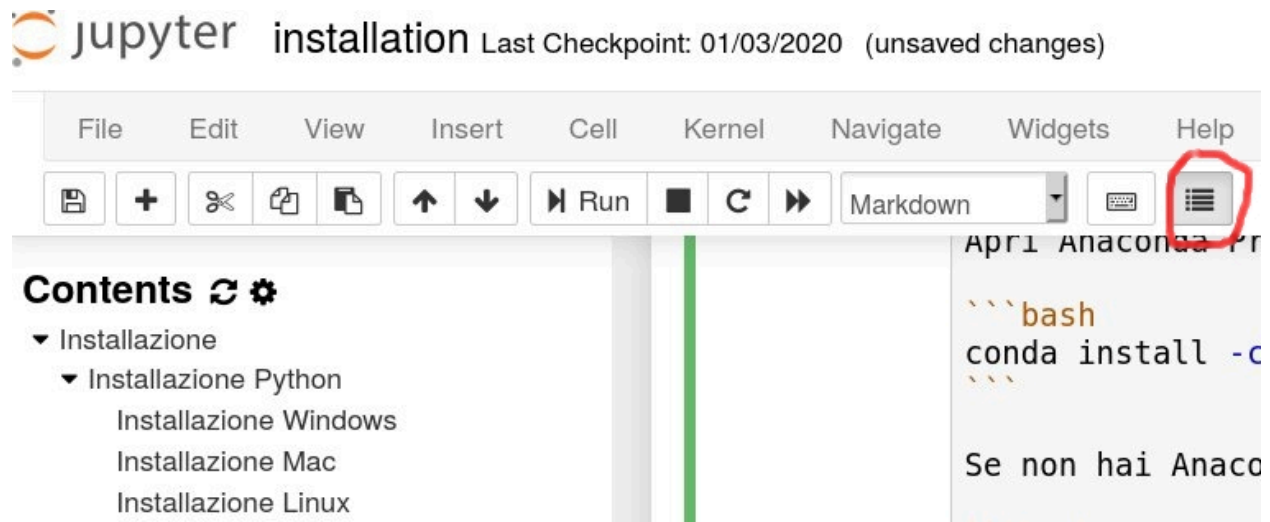
```
jupyter contrib nbextension install --user
```

3. Enable extensions:

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

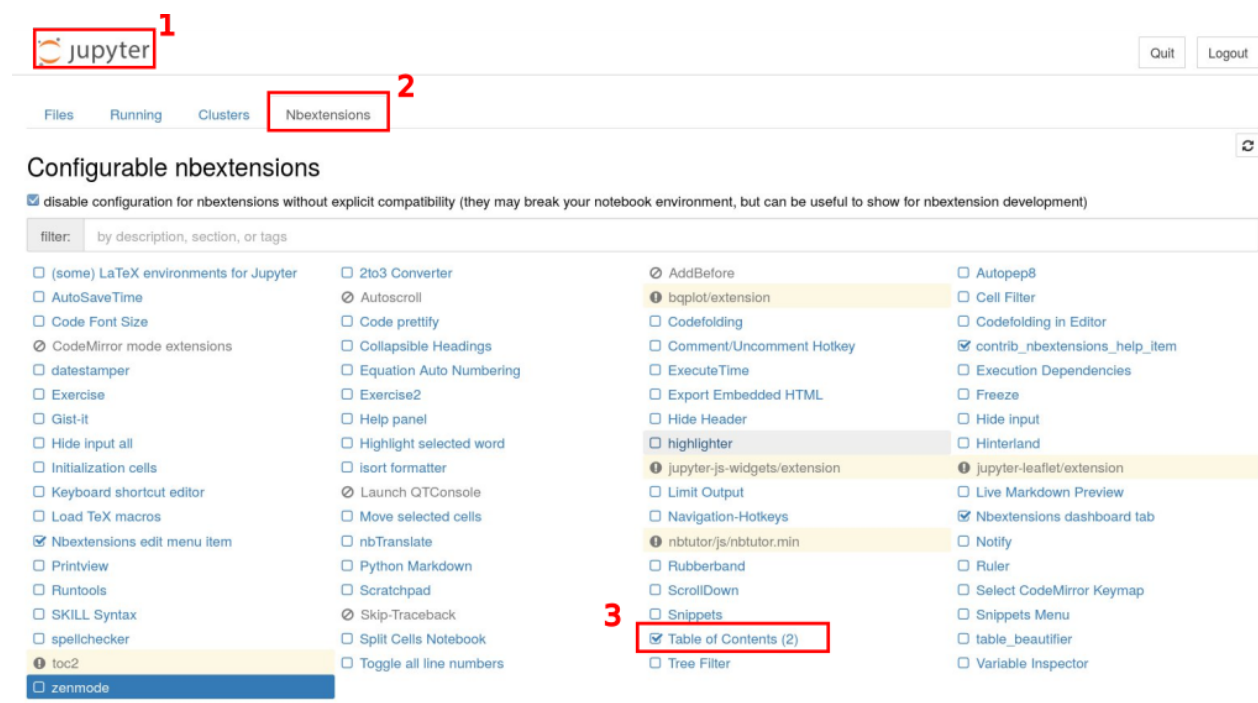
⁵⁸ https://github.com/ipython-contrib/jupyter_contrib_nbextensions

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 typing ``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 Python 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 commands 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 visible 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 [activate 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 *Jupyter 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](https://en.softpython.org/tools/tools-sol.html)⁶¹ to learn how to use other editors and Python architecture.

⁶¹ <https://en.softpython.org/tools/tools-sol.html>

A - FOUNDATIONS

4.1 Tools and scripts

4.1.1 Download exercises zip

Browse files online⁶²

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
MYFOLDER              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
MYFOLDER              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 Python 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 command `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
MYFOLDER               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
MYFOLDER                              java0.log                             _
↪RegistroDocenteStandard(1).pdf      java1.log                             _
backupsys                             ↪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 (also 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 MYFOLDER:

- **WINDOWS:** if you use *Notepad*, in the save window you have 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. Make 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: in this 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 `MYFOLDER` there is our file `my_script.py`

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

```
C:\Users\David\MYFOLDER>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 [PyCharm 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)
```

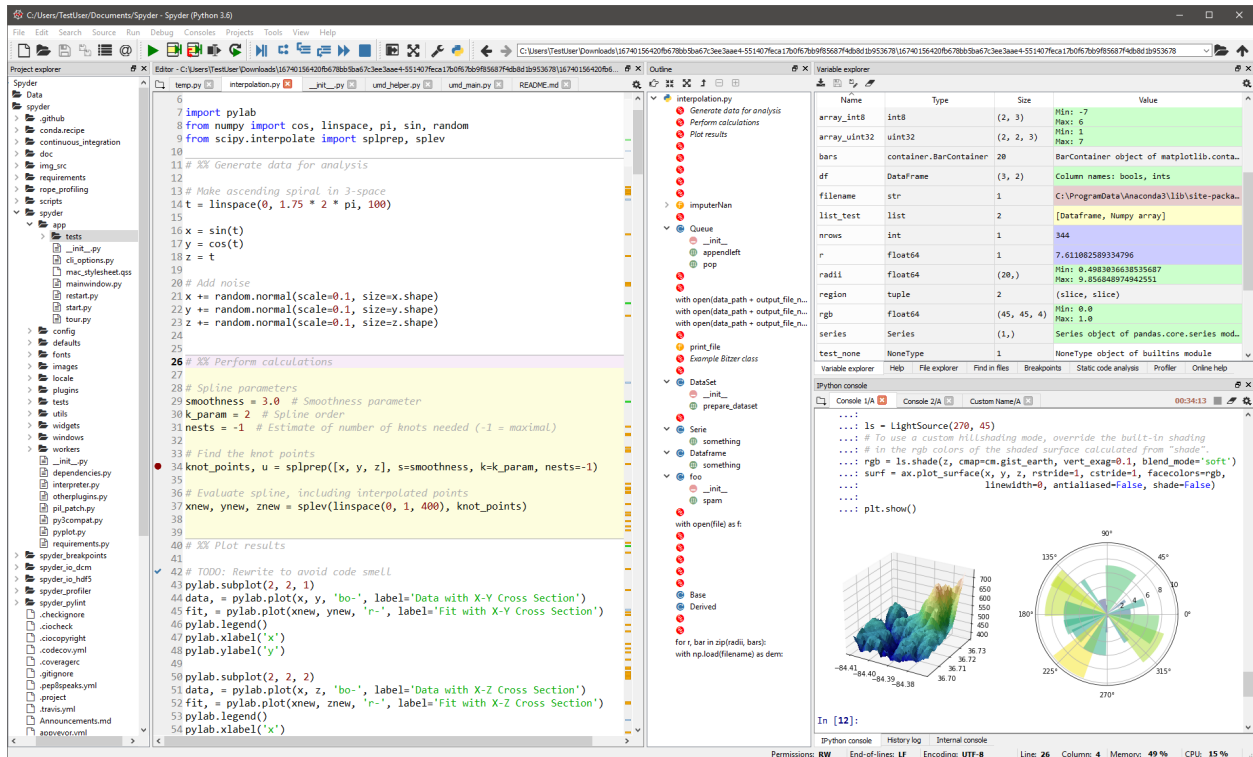
- once the code is in the script, hit F5
- select only `print(x+y)` and hit F9
- select only `x=3` and hit F9
- click on the gear the right of the console panel, and select *Restart kernel*, then select only `print(x+y)` and hit F9. What happens?

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 executed 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. Two things should appear, first a console and then a browser. In the browser navigate the files to reach the unzipped folder, and open the notebook `tools.ipynb`

WARNING: DO NOT click Upload button in Jupyter

Just navigate until you reach the file.

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

Seeing now the solutions is too easy ;-)

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`

If you don't find Jupyter / something doesn't work: have a look at [installation](#)⁶⁷

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 afterwards, 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.

⁶⁷ <https://en.softpython.org/installation.html#Jupyter-Notebook>

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 explicitly 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 writing 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 verifying 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.

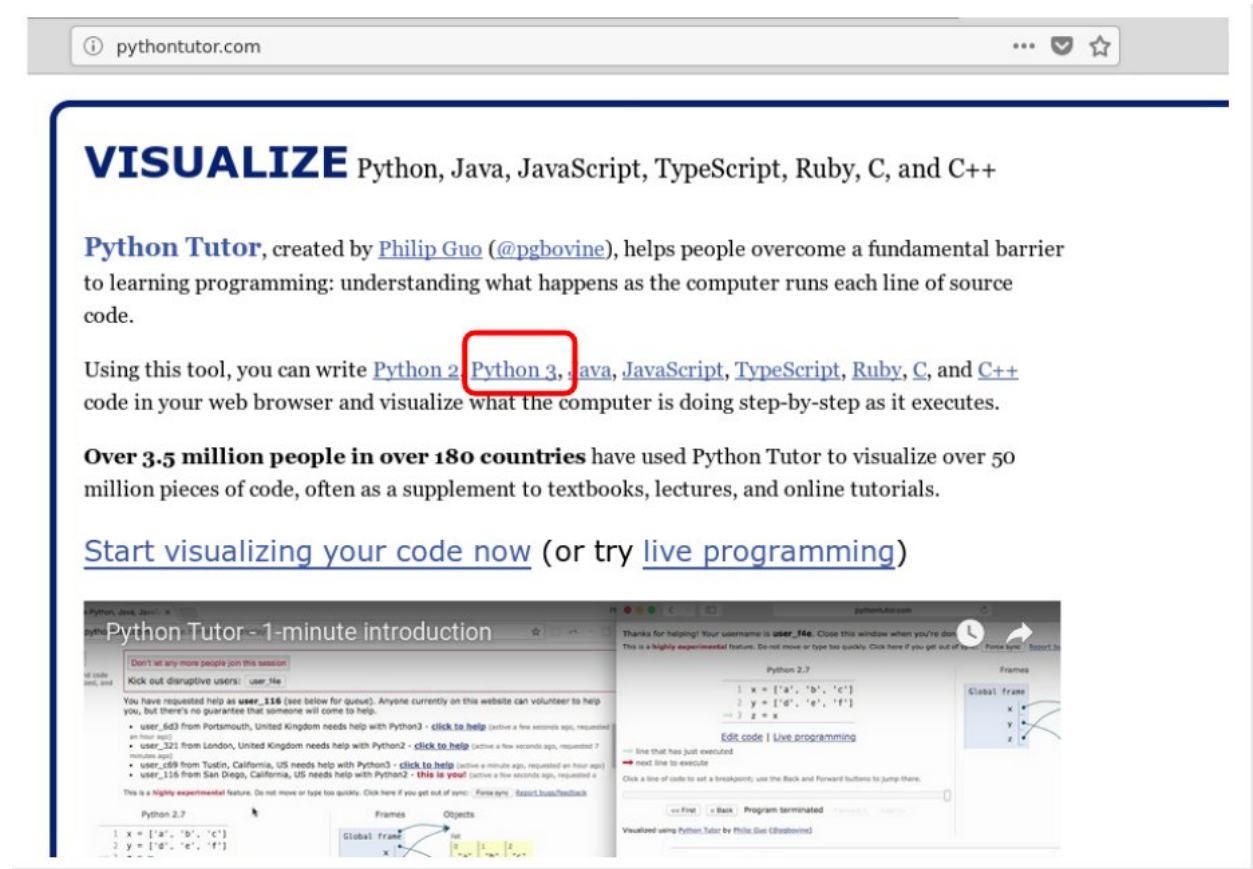
[Python tutor](http://pythontutor.com/)⁶⁸ is a very good website to visualize online Python code execution, allowing to step forth and *back* in code flow. Exploit it as much as you can, it should work with many of the examples we shall see in the book. Let's try an example

Python tutor 1/4

Go to pythontutor.com⁶⁹ and select *Python 3*

⁶⁸ <http://pythontutor.com/>

⁶⁹ <http://pythontutor.com/>



VISUALIZE Python, Java, JavaScript, TypeScript, Ruby, C, and C++

Python Tutor, created by [Philip Guo \(@pgbovine\)](#), helps people overcome a fundamental barrier to learning programming: understanding what happens as the computer runs each line of source code.

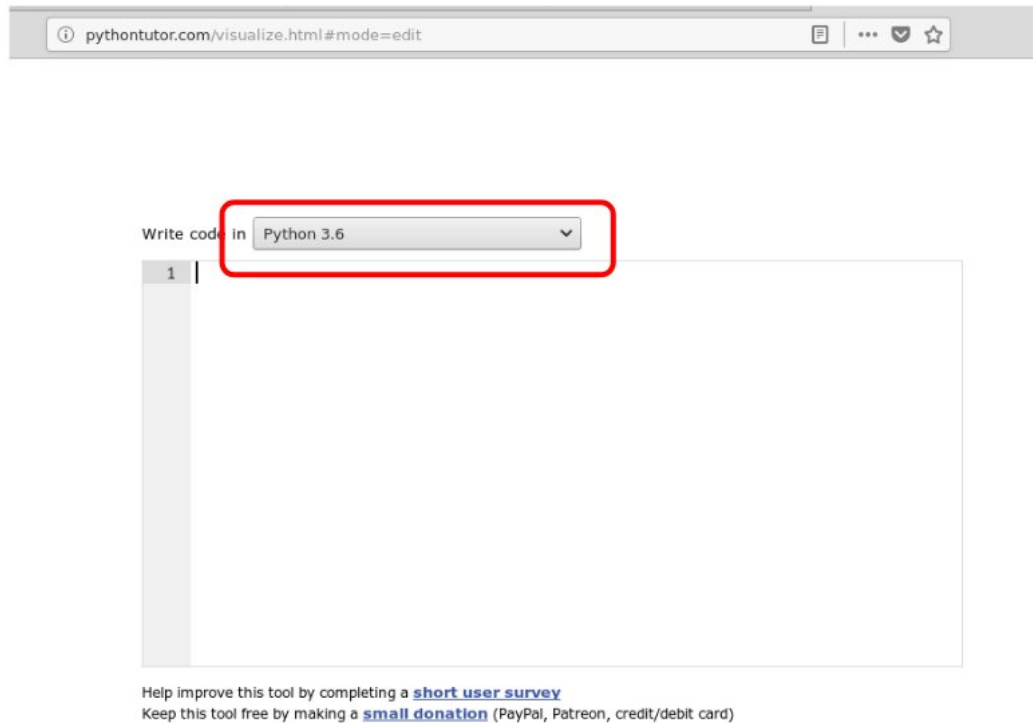
Using this tool, you can write [Python 2](#), **Python 3**, [Java](#), [JavaScript](#), [TypeScript](#), [Ruby](#), [C](#), and [C++](#) code in your web browser and visualize what the computer is doing step-by-step as it executes.

Over 3.5 million people in over 180 countries have used Python Tutor to visualize over 50 million pieces of code, often as a supplement to textbooks, lectures, and online tutorials.

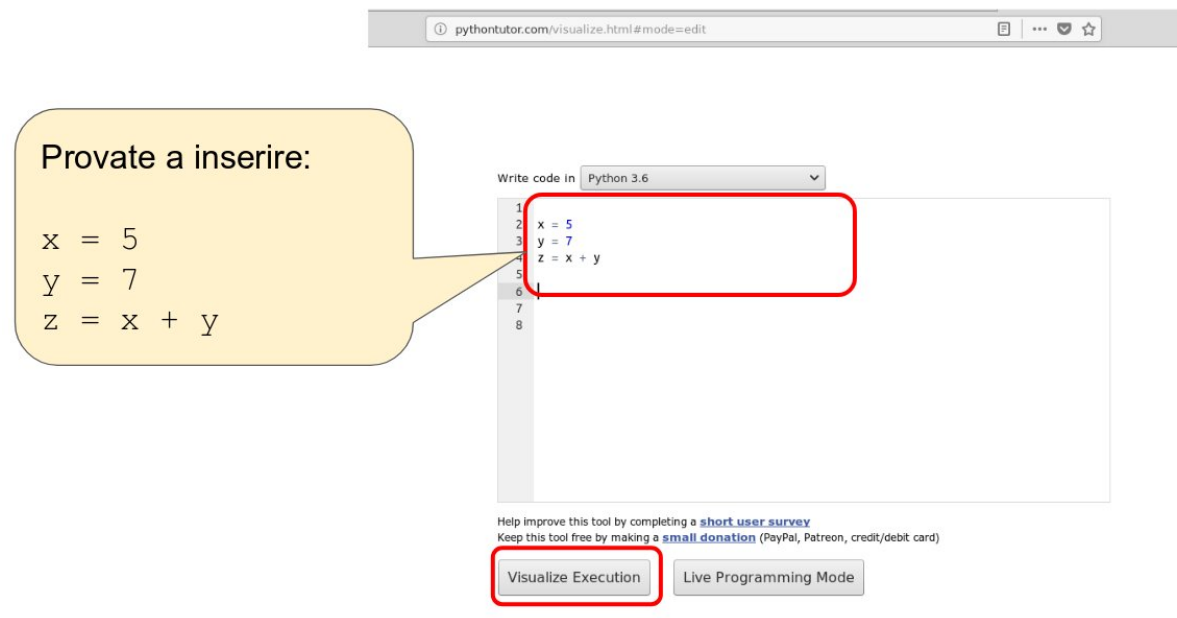
[Start visualizing your code now](#) (or try [live programming](#))

The screenshot shows the Python Tutor interface with a code editor containing Python 2.7 code: `1 x = ['a', 'b', 'c']`, `2 y = ['d', 'e', 'f']`, and `3 z = x`. The 'Frames' panel on the right shows the 'Global frame' with variables `x`, `y`, and `z`. The 'Objects' panel shows the memory representation of these variables.

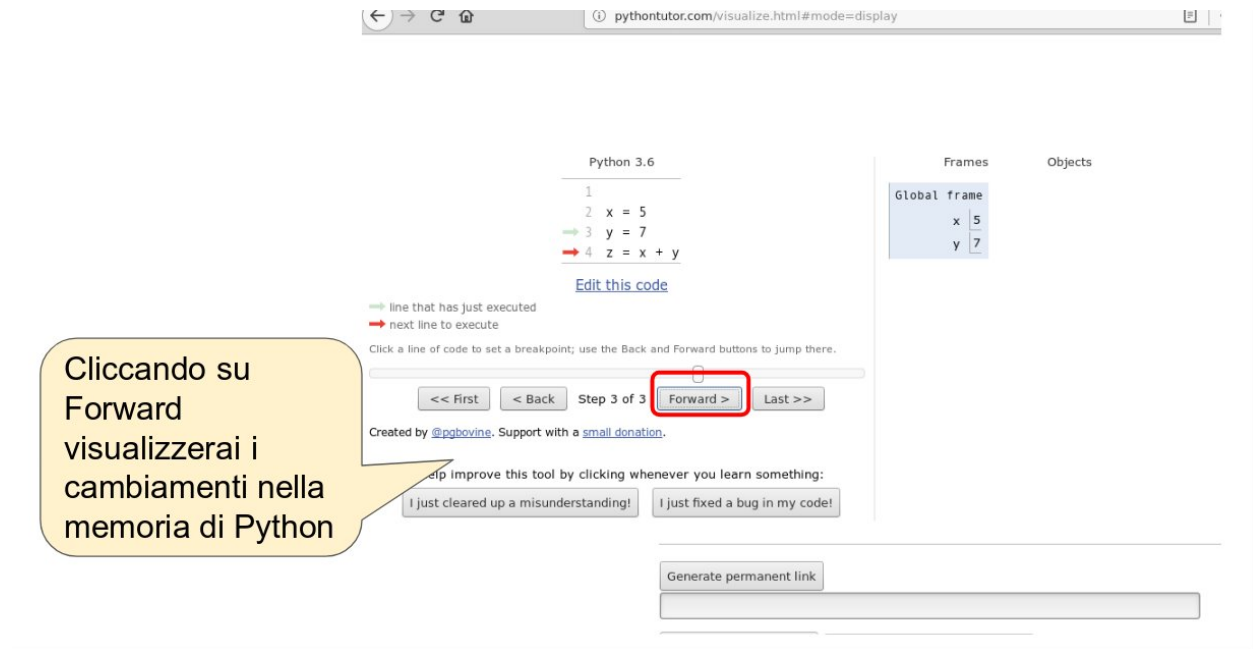
Python tutor 2/4



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 to 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 be 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>
```

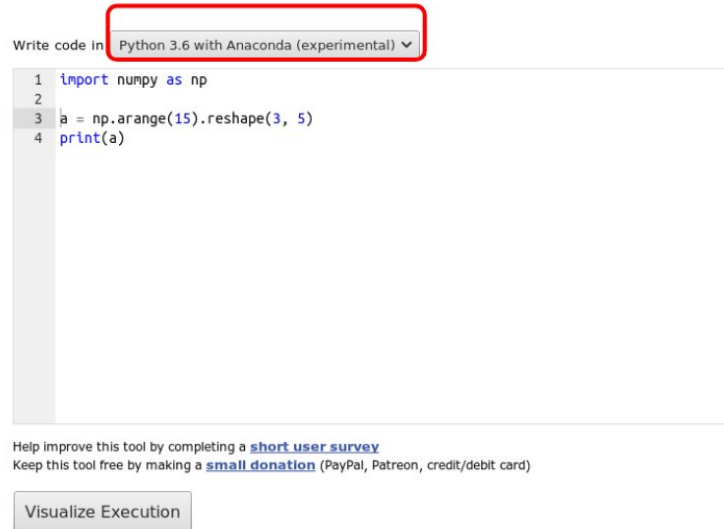
Python Tutor : Limitation 2

WARNING: Python Tutor only uses functions from standard 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`

Se vuoi usare librerie particolari come numpy, Prova a selezionare Python 3.6 with Anaconda (sperimentale !)



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:

```
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
```

- **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 intelligible 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	<code>.ipynb</code>	
Python Compiler	Python source code	<code>.py</code>	<code>x = 3 y = 5 print(x + y)</code>
Python Interpreter	Python bytecode	<code>.pyc</code>	<code>0 LOAD_CONST 1 (3) 3 STORE_FAST 0 (x)</code>
Processor (CPU)	Machine code	<code>.asm</code>	<code>cmp edi, 0 je mult _end</code>

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 typically 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 files⁷²

PREREQUISITES:

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

4.2.2 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 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. Two things should appear, first a console and then a browser. In the browser navigate the files to reach the unzipped folder, and open the notebook `basics.ipynb`

WARNING: DO NOT click Upload button in Jupyter

Just navigate until you reach the file.

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: open the notebook WITHOUT the `-sol` at the end!

Seeing now the solutions is too easy ;-)

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`

If you don't find Jupyter / something doesn't work: have a look at [installation](#)⁷⁵

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 afterwards, 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, number 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?
<code>bool</code>	Condition	True, False	no
<code>int</code>	Integer	\mathbb{Z}	no
<code>long</code>	Integer	\mathbb{Z}	no
<code>float</code>	Rational	\mathbb{Q} (more or less)	no
<code>str</code>	Text	Text	no
<code>list</code>	Sequence	Collezione di oggetti	yes
<code>tuple</code>	Sequence	Collezione di oggetti	no
<code>set</code>	Set	Collezione di oggetti	yes
<code>dict</code>	Mapping	Mapping between objects	yes

⁷⁵ <https://en.softpython.org/installation.html#Jupyter-Notebook>

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

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 keep 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 created in the environment and assigned to that object

Detect the type of a variable

When you see a variable or constant 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 happened 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`

Reusing 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 arithmetic 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)
```

Exercise - cycling

⊗ Write a program that given three variables with numbers 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 during 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. Names can only contain upper/lower case digits (A-Z, a-z), numbers (0-9) or underscores _;
2. Names cannot start with a number;
3. Variable names should start with a lowercase letter
4. 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 COMMANDMENT⁷⁶: 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 !!!
15. asino
16. As

⁷⁶ <https://en.softpython.org/commandments.html#V-COMMANDMENT>

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
<code>**</code>	power
<code>+ -</code>	Unary plus and minus
<code>* / // %</code>	Multiplication, division, integer division, module
<code>+ -</code>	Addition and subtraction

There are also several predefined functions:

Function	Description
<code>min(x, y, ...)</code>	the minimum among given numbers
<code>max(x, y, ...)</code>	the maximum among given numbers
<code>abs(x)</code>	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
<code>math.floor(x)</code>	round x to inferior integer
<code>math.ceil(x)</code>	round x to superior integer
<code>math.sqrt(x)</code>	the square root
<code>math.log(x)</code>	the natural logarithm of n
<code>math.log(x, b)</code>	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 remainder 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 COMMANDMENT⁷⁸: 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 through bridge b5

The company wants to know which are the maximum tons it can drive to destination in a single 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 seatback 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 producing 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
<code>x or y</code>	<code>not(not x and not y)</code>
<code>x and y</code>	<code>not(not x or not y)</code>

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 integers 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` independently 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 might 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 constants 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 program 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 from 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 even 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
<code>a == b</code>	True if and only if $a = b$
<code>a != b</code>	True if and only if $a \neq b$
<code>a < b</code>	True if and only if $a < b$
<code>a > b</code>	True if and only if $a > b$
<code>a <= b</code>	True if and only if $a \leq b$
<code>a >= b</code>	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 recursion⁷⁹, in particular Sections 5.2 and 5.3, Boolean expressions⁸⁰ 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, exponent 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!

Scientific notation

Whenever numbers are very big or very small, to avoid having to write too many zeros it is convenient to use scientific 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 an 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](https://en.softpython.org/matrices-numpy/matrices-numpy-sol.html#NaN-e-infinities)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 they 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]: 20.533333333333333
```

Reals - conversion

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

Function	Description	Mathematical symbol	Result
<code>math.floor(x)</code>	round x to inferior integer	$\lfloor 8.7 \rfloor$	8
<code>int(x)</code>	round x to inferior integer	$\lfloor 8.7 \rfloor$	8
<code>math.ceil(x)</code>	round x to superior integer	$\lceil 5.3 \rceil$	6
<code>round(x)</code>	round x to closest integer	$\lfloor 2.5 \rfloor$	2
		$\lceil 2.51 \rceil$	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 karité 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 transferring, 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

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

```
[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 might 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 floats 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 another, 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 represent 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 during 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 which is most similar to others. You then conduct a rigorous statistical survey among tourists obsessed by selfie sticks with the goal to find the most frequent brands of 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% tourists each. Write some code which prints this result:

```

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 % )

```

- **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 conscious of their limits of representation and equality. If you really need more precision and/or predictability, 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 constant, 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('4.0999999999999996447286321199499070644378662109375')
```

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('1.414213562373095048801688724')
```

Remember: computer memory is still finite!

Decimals can't 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 financial 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 combination 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
```

Challenge - 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 experiments, `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 (m_j) equal to 30 units, minor base (m_n) 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

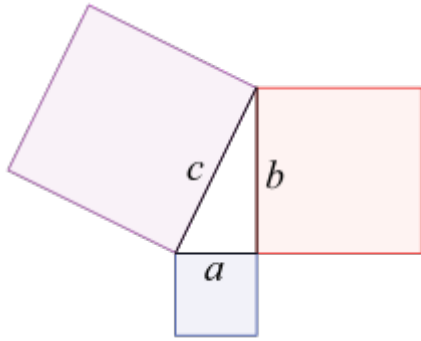
It should show Sum of first 1200 integers: 720600.0

```
[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 go to the airport, expecting to spend some time in several 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 examined, 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 shortest 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

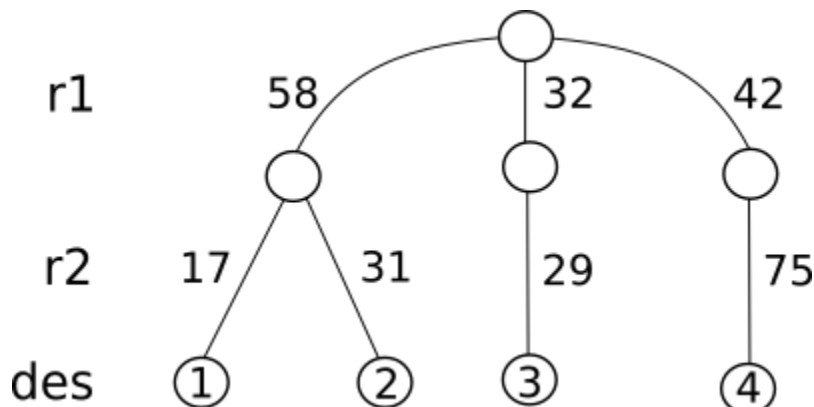
# write here
```

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

        # write here
```

4.2.11 References

- Think Python - Chapter 1⁸⁵: The way of the program
- Think Python - Chapter 2⁸⁶: Variables, expressions and statements

```
[ ]:
```

4.3 Strings 1 - introduction

4.3.1 Download exercises zip

Browse files online⁸⁷

Strings are *immutable* character sequences, and one of the basic Python types. In this notebook we will see how to manipulate them.

⁸⁵ <http://greenteapress.com/thinkpython2/html/thinkpython2002.html>

⁸⁶ <http://greenteapress.com/thinkpython2/html/thinkpython2003.html>

⁸⁷ <https://github.com/DavidLeoni/softpython-it/tree/master/strings>

4.3.2 What to do

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

```
strings
  strings1.ipynb
  strings1-sol.ipynb
  strings2.ipynb
  strings2-sol.ipynb
  strings3.ipynb
  strings3-sol.ipynb
  strings4.ipynb
  strings4-sol.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 a browser. The browser should show a file list: navigate the list and open the notebook `strings1.ipynb`
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 ☆☆☆

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 create a new cell afterwards, press `Alt + Enter`
- If the notebooks look stuck, try to select `Kernel -> Restart`

4.3.3 Creating strings

There are several ways to define a string.

Double quotes, in one line

```
[2]: a = "my first string, in double quotes"
```

```
[3]: print(a)
```

```
my first string, in double quotes
```

Single quotes, in one line

This way is equivalent to previous one.

```
[4]: b = 'my second string, in single quotes'
```

```
[5]: print(b)
```

```
my second string, in single quotes
```

Between double quotes, on many lines

```
[6]: c = """my third string
      in triple double quotes
      so I can put it

      on many rows"""
```

```
[7]: print(c)

my third string
in triple double quotes
so I can put it

on many rows
```

Three single quotes, many lines

```
[8]: d = '''my fourth string,
      in triple single quotes
      also can be put

      on many lines
      '''
```

```
[9]: print(d)

my fourth string,
in triple single quotes
also can be put

on many lines
```

4.3.4 Printing - the cells

To print a string we can use the function `print`:

```
[10]: print('hello')

hello
```

Note that apices are *not* reported in printed output.

If we write the string without the `print`, we will see the apices indeed:

```
[11]: 'hello'
[11]: 'hello'
```

What happens if we write the string with double quotes?

```
[12]: "hello"
[12]: 'hello'
```

Notice that by default Jupyter shows single apices.

The same applies if we assign a string to a variable:


```
[13]: x = 'hello'
```

```
[14]: print(x)
```

```
hello
```

```
[15]: x
```

```
[15]: 'hello'
```

```
[16]: y = "hello"
```

```
[17]: print(y)
```

```
hello
```

```
[18]: y
```

```
[18]: 'hello'
```

4.3.5 The empty string

The string of zero length is represented with two double quotes "" or two single apices ''

Note that even if we write two double quotes, Jupyter shows a string beginning and ending with single apices:

```
[19]: ""
```

```
[19]: ''
```

The same applies if we associate an empty string to a variable:

```
[20]: x = ""
```

```
[21]: x
```

```
[21]: ''
```

Note that even if we ask Jupyter to use `print`, we won't see anything:

```
[22]: print("")
```

```
[23]: print('')
```

4.3.6 Printing many strings

For printing many strings on a single line there are different ways, let's start from the most simple with `print`:

```
[24]: x = "hello"
      y = "Python"

      print(x,y)    # note that in the printed characters Python inserted a space:
hello Python
```

We can add to `print` as many parameters we want, which can also be mixed with other types like numbers:

```
[25]: x = "hello"
      y = "Python"
      z = 3

      print(x,y,z)
hello Python 3
```

4.3.7 Length of a string

To obtain the length of a string (or any sequence in general), we can use the function `len`:

```
[26]: len("ciao")
[26]: 4

[27]: len("")    # empty string
[27]: 0

[28]: len('')    # empty string
[28]: 0
```

QUESTION: Can we write something like this?

```
"len"("hello")
```

ANSWER: no, `"len"` between quotes will be interpreted as a string, not as a function, so Python will complain telling us we cannot apply a string to another string. Try to see which error appears by rewriting the expression below:

```
[29]: # write here

      #"len"("hello")
```

QUESTION: can we write something like this? What does it produce? an error? a number? which one?

```
len("len('hello')")
```

ANSWER: it returns the number 12: by putting the Python code `len('hello')` among double quotes, it became a string like any other. So by writing `len("len('hello')")` we count how long the string `"len('hello')"` is.

QUESTION: What do we obtain if we write like this?

```
len((((("ciao")))))
```

1. an error
2. the length of the string
3. something else

ANSWER: The second: "ciao" is an expression, as such we can enclose it in as many parenthesis as we want.

Counting escape sequences: Note that some particular sequences called *escape sequences* like for example `\t` occupy less space of what it seems (with `len` they count as 1), but if we print them they will occupy even more than 2 !!

Let's see an example (in the next paragraph we will delve into the details):

```
[30]: len('a\tb')
```

```
[30]: 3
```

```
[31]: print('a\tb')
```

```
a      b
```

4.3.8 Printing - escape sequences

Some characters sequences called *escape sequences* are special because instead of showing characters, they force the printing to do particular things like line feed or inserting extra spaces. These sequences are always preceded by the *backslash* character `\`:

Description	Escape sequence
Linefeed	<code>\n</code>
Tabulation (<i>ASCII tab</i>)	<code>\t</code>

Esempio - line feed

```
[32]: print("hello\nworld")
```

```
hello
world
```

Note the line feed happens only when we use `print`, if instead we directly put the string into the cell we will see it verbatim:

```
[33]: "ciao\nmondo"
```

```
[33]: 'ciao\nmondo'
```

In a string you can put as many escape sequences as you like:

```
[34]: print("Today is\na great day\nisn't it?")
```

```
Today is
a great day
isn't it?
```

Example - tabulation

```
[35]: print("hello\tworld")
```

```
hello    world
```

```
[36]: print("hello\tworld\twith\tmany\t\ttabs")
```

```
hello    world    with    many    tabs
```

EXERCISE: Since *escape sequences* are special, we might ask ourselves how long they are. Use the function `len` to print the string length. Do you notice anything strange?

- `'ab\ncd'`
- `'ab\tcd'`

```
[37]: # write here
```

EXERCISE: Try selecting the character sequence printed in the previous cell with the mouse. What do you obtain? A space sequence, or a single tabulation character? Note this can vary according to the program that actually printed the string.

EXERCISE: find a SINGLE string which printed with `print` is shown as follows:

```
This    is
an
apparently  simple      challenge
```

- USE ONLY combinations of `\t` and `\n`
- DON'T use spaces
- start and end the string with a single apex

```
[38]: # write here
```

```
print('This\tis\nan\nnapparently\tsimple\t\tchallenge')
```

```
This    is
an
apparently    simple      challenge
```

EXERCISE: try to find a string which printed with `print` is shown as follows:

```
At  te
n
    t  ion
    please!
```

- USE ONLY combinations of `\t` and `\n`
- DON'T use any space
- DON'T use triple quotes

```
[39]: # write here
print("At\tte\nn\n\tt\t\tion\n\ttplease!")
```

```
At      te
n

      t      ion
      please!
```

Special character: if we want special characters like the single apex ' or double quotes " inside a string, we must create a so-called *escape sequence*, that is, we must first write the *backslash* character \ and then follow it with the special character we're interested in:

Description	Escape sequence	Printed result
Single apex	\ '	'
Double quote	\ "	"
Backslash	\\	\

Example:

Let's print a string containing a single apex ' and a double quote ":

```
[40]: my_string = "This way I put \'apices\' e \"double quotes\" in strings"
```

```
[41]: print(my_string)

This way I put 'apices' e "double quotes" in strings
```

If a string begins with double quotes, inside we can freely use single apices, even without *backslash* \:

```
[42]: print("There's no problem")

There's no problem
```

If the string begins with single apices, we can freely use double quotes even without the *backslash* \:

```
[43]: print('It Is So "If You Think So"')

It Is So "If You Think So"
```

EXERCISE: Find a string to print with print which shows the following sequence:

- the string **MUST** start and finish with single apices '

```
This "genius" of strings wants to /\//\ trick me \//\ with atrocious exercises O_o'
```

```
[44]: # write here

print('This "genius" of strings wants to /\//\ trick me \//\ with atrocious_
↪exercises O_o\'')
```

```
This "genius" of strings wants to /\//\ trick me \//\ with atrocious exercises O_o'
```

4.3.9 Encodings

ASCII characters

When using strings in your daily programs you typically don't need to care much how characters are physically represented as bits in memory, but sometimes it does matter. The representation is called *encoding* and must be taken into account in particular when you read stuff from external sources such as files and websites.

The most famous and used character encoding is [ASCII](https://en.wikipedia.org/wiki/ASCII)⁸⁸ (American Standard Code for Information Interchange), which offers 127 slots made by basic printable characters from English alphabet (a-z, A-Z, punctuation like . ; , ! and characters like (, @ ...) and control sequences (like \t, \n)

- See [Printable characters](https://en.wikipedia.org/wiki/Printable_characters)⁸⁹ (Wikipedia)
- [ASCII Control codes](https://en.wikipedia.org/wiki/ASCII_control_codes)⁹⁰ (Wikipedia)

Original ASCII table lacks support for non-English languages (for example, it lacks Italian accented letters like è, à, ...), so many extensions were made to support other languages, for examples see [Extended ASCII](https://en.wikipedia.org/wiki/Extended_ASCII)⁹¹ page on Wikipedia.

Unicode characters

Whenever we need particular characters like ☼ which are not available on the keyboard, we can look at Unicode characters. There are a lot⁹², and we can often use them in Python 3 by simple copy-pasting. For example, if you go to [this page](https://en.wikipedia.org/wiki/Unicode)⁹³ we can copy-paste the character ☼. In other cases it might be so special it can't even be correctly visualized, so in these cases you can use a more complex sequence in the format \uxxxx like this:

Description	Escape sequence	Printed result
Example star in a circle in format \uxxxx	\u272A	☼

EXERCISE: Search Google for *Unicode heart* and try to print a hear in Python, both by directly copy-pasting the character and by using the notation \uxxxx

```
[45]: # write here

print("I ♥ Python, with copy-paste")
print("I \u2665 Python, also in format \\\uxxxx")

I ♥ Python, with copy-paste
I ♥ Python, also in format \uxxxx
```

Unicode references: Unicode can be a complex topic we just mentioned, if you ever need to deal with complex character sets like japanese or heterogenous text encodings here a couple of references you should read:

- first part on Unicode encoding [from Strings chapter from book Dive into Python 3](https://en.wikipedia.org/wiki/Unicode)⁹⁴
- [Python 3 Unicode](https://docs.python.org/3/howto/unicode.html)⁹⁵ documentation

⁸⁸ <https://en.wikipedia.org/wiki/ASCII>

⁸⁹ https://en.wikipedia.org/wiki/ASCII#Printable_characters

⁹⁰ https://en.wikipedia.org/wiki/C0_and_C1_control_codes#Basic_ASCII_control_codes

⁹¹ https://en.wikipedia.org/wiki/Extended_ASCII

⁹² <http://www.fileformat.info/info/unicode/char/a.htm>

⁹³ <https://www.fileformat.info/info/unicode/char/272a/index.htm>

⁹⁴ <https://diveintopython3.net/strings.html>

⁹⁵ <https://docs.python.org/3/howto/unicode.html>

4.3.10 Strings are immutable

Strings are *immutable* objects, so once they are created you cannot change them anymore. This might appear restrictive, but it's not so tragic, because we still have available these alternatives:

- generate a new string composed from other strings
- if we have a variable to which we assigned a string, we can assign another string to that variable

Let's generate a new string starting from previous ones, for example by joining two of them with the operator +

```
[46]: x = 'hello'
```

```
[47]: y = x + 'world'
```

```
[48]: x
```

```
[48]: 'hello'
```

```
[49]: y
```

```
[49]: 'helloworld'
```

The + operation, when executed among strings, it joins them by creating a NEW string. This means that the association to x it didn't change at all, the only modification we can observe will be the variable y which is now associated to the string 'helloworld'. Try making sure of this in Python Tutor by repeatedly clicking on *Next* button:

```
[50]: # WARNING: before using the function jupman.pytut() which follows,
# it is necessary to first execute this cell with Shit+Enter

# it's sufficient to execute it only once, you find it also in all other notebooks in...
↳ the first cell

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

```
[51]: x = 'hello'
y = x + 'world'
```

```
print(x)
print(y)
```

```
jupman.pytut()
```

```
hello
helloworld
```

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

Reassign variables

Other variations to memory state can be obtained by reassigning the variables, for example:

```
[52]: x = 'hello'
```

```
[53]: y = 'world'
```

```
[54]: x = y          # we assign to x the same string contained in y
```

```
[55]: x
```

```
[55]: 'world'
```

```
[56]: y
```

```
[56]: 'world'
```

If a string is created and at some point no variables point to it, Python automatically takes care to eliminate it from the memory. In the case above, the string `hello` is never actually changed: at some point no variable is associated with it anymore and so Python eliminates the string from the memory. Have a look at what happens in Python Tutor:

```
[57]: x = 'hello'
```

```
y = 'world'
```

```
x = y
```

```
jupman.pytut()
```

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

Reassign a variable to itself

We may ask ourselves what happens when we write something like this:

```
[58]: x = 'hello'
```

```
x = x
```

```
[59]: print(x)
```

```
hello
```

No big changes, the assignment of `x` remained the same without alterations.

But what happens if to the right of the `=` we put a more complex formula?

```
[60]: x = 'hello'
```

```
x = x + 'world'
```

```
print(x)
```

```
helloworld
```

Let's try to carefully understand what happened.

In the first line, Python generated the string `'hello'` and assigned it to the variable `x`. So far, nothing extraordinary.

Then, in the second line, Python did two things:

1. it calculated the result of the expression `x + 'world'`, by generating a NEW string `helloworld`
2. it assigned the generated string `helloworld` to the variable `x`

It is fundamental to understand that whenever a reassignment is performed both passages occurs, so it's worth repeating them:

- FIRST the result of the expression to the right of `=` is calculated (so when the old value of `x` is still available)
- THEN the result is associated to the variable to the left of `=` symbol

If we check out what happens in Python Tutor, this double passage is executed in a single shot:

```
[61]: x = 'hello'
      x = x + 'world'

      jupman.pytut()

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

EXERCISE: Write some code that changes memory state in such a way so that in the end the following is printed:

```
z = This
w = was
x = a problem
y = was
s = This was a problem
```

- to write the code, USE ONLY the symbols `=,+,z,w,x,y,s` AND NOTHING ELSE
- you can freely use as many lines of code as you deem necessary
- you can freely use any symbol as many times you deem necessary

```
[62]: # these variables are given

      z = "This"
      w = 'is'
      x = 'a problem'
      y = 'was'
      s = ' '

      # write here the code

      w = y

      s = z + s + y + s + x
```

```
[63]: print("z = ", z)
      print("w = ", w)
      print("x = ", x)
      print("y = ", y)
      print("s = ", s)

      z = This
      w = was
      x = a problem
      y = was
      s = This was a problem
```

4.3.11 Strings and numbers

Python strings have the type `str`:

```
[64]: type("hello world")
```

```
[64]: str
```

In strings we can insert characters which represent digits:

```
[65]: print("The character 5 represents the digit five, the character 3 represents the_
↪digit three")
```

```
The character 5 represents the digit five, the character 3 represents the digit three
```

Obviously, we can also substitute a sequence of digits, to obtain something which looks like a number:

```
[66]: print("The sequence of characters 7583 represents the number seven thousand five_
↪hundred eighty-three")
```

```
The sequence of characters 7583 represents the number seven thousand five hundred_
↪eighty-three
```

Having said that, we can ask ourselves how Python behaves when we have a *string* which contains *only* a sequence of characters which represents a number, like for example `'254'`

Can we use `254` (which we wrote like it were a string) also as if it were a number? For example, can we sum 3 to it?

```
'254' + 3

-----
TypeError                                 Traceback (most recent call last)
<ipython-input-29-d39aa62a7e3d> in <module>
----> 1 "254" + 3

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

As you see, Python immediately complains, because we are trying to mix different types.

SO:

- by writing `'254'` between apices we create a *string* of type `str`
- by writing `254` we create a *number* of type `int`

```
[67]: type('254')
```

```
[67]: str
```

```
[68]: type(254)
```

```
[68]: int
```

BEWARE OF `print` !!

If you try to print a string which only contains digits, Python will show it without apices, and this might mislead you about its true nature !!

```
[69]: print('254')
```

```
254
```

```
[70]: print(254)
```

```
254
```

Only in Jupyter, to show constants, variables or results of calculations, as `print` alternative you can directly insert a formula in the cell. In this case we are simply showing a constant, and whenever it is a string you will see apices:

```
[71]: '254'
```

```
[71]: '254'
```

```
[72]: 254
```

```
[72]: 254
```

The same reasoning applies also to variables:

```
[73]: x = '254'
```

```
[74]: x
```

```
[74]: '254'
```

```
[75]: y = 254
```

```
[76]: y
```

```
[76]: 254
```

So, *only in Jupyter*, when you need to show a constant, a variable or a calculation often it's more convenient to directly write it in the cell without using `print`.

4.3.12 Conversions - from string to number

Let's go back to the problem of summing `'254' + 3`. The first one is a string, the second a number. If they were both numbers the sum would surely work:

```
[77]: 254 + 3
```

```
[77]: 257
```

So we can try to convert the string `'254'` into an authentic integer. To do it, we can use `int` as if it were a function, and pass as argument the string to be converted:

```
[78]: int('254') + 3
```

```
[78]: 257
```

WARNING: strings and numbers are immutable !!

This means that by writing `int('254')` a *new* number is generated without minimally affecting the string `'254'` from where we started from. Let's see an example:

```
[79]: x = '254'      # assign to variable x the string '254'

[80]: y = int(x)     # assign to variable y the number obtained by converting '254' in int

[81]: x              # variable x is now assigned to string '254'
[81]: '254'

[82]: y              # in y now there is a number instead (note we don't have apices here)
[82]: 254
```

It might be useful to see again the example in Python Tutor:

```
[83]: x = "254"

      y = int(x)

      print(y + 3)

      jupman.pytut()

257

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

EXERCISE: Try to convert a string which represents an ill-formed number (for example a number with inside a character: '43K12') into an `int`. What happens?

```
[84]: # write here
```

4.3.13 Conversions - from number to string

Any object can be converted to string by using `str` as if it were a function and by passing the object to convert. Let's try then to convert a number into a string.

```
[85]: str(5)

[85]: '5'
```

note the apices in the result, which show we actually obtained a string.

If by chance we want to obtain a string which is the concatenation of objects of different types we need to be careful:

```
x = 5
s = 'Workdays in a week are ' + x
print(s)

-----
TypeError                                 Traceback (most recent call last)
<ipython-input-154-5951bd3aa528> in <module>
      1 x = 5
----> 2 s = 'Workdays in a week are ' + x
```

(continues on next page)

(continued from previous page)

```
3 print(s)

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

A way to circumvent the problem (even if not the most convenient) is to convert into string each of the objects we're using in the concatenation:

```
[86]: x = 3
      y = 1.6
      s = "This week I've been jogging " + str(x) + " times running at an average speed of
      ↪ " + str(y) + " km/h"
      print(s)

This week I've been jogging 3 times running at an average speed of 1.6 km/h
```

QUESTION: Having said that, after executing the code in previous cell, variable `x` is going to be associated to a *number* or a *string* ?

If you have doubts, use Python Tutor.

ANSWER: numbers, like strings, are immutable. So by calling the function `str(x)` it is impossible for the number 5 associated to `x` to be changed in any way. `str(x)` will simply generate a NEW string '5' which will then be used in the concatenation.

4.3.14 Formatting strings

Concatenating strings with plus sign like above is cumbersome and error prone. There are several better solutions, for a thorough review we refer to [Real Python](https://realpython.com/python-formatted-output/)⁹⁶ website. In particular, check out the most handy which is *f-strings*⁹⁷ and available for Python >= 3.6

Formatting with %

Here we now see how to format strings with the `%` operator. This solution is not the best one, but it's widely used and supported in all Python versions, so we adopted it throughout the book:

```
[87]: x = 3
      "I jumped %s times" % x

[87]: 'I jumped 3 times'
```

Notice we put a so-called *place-holder* `%s` inside the string, which tells Python to replace it with a variable. To feed Python the variable, *after* the string we have to put a `%` symbol followed by the variable, in this case `x`.

If we want to place more than one variable, we just add more `%s` place-holders and after the external `%` we place the required variables in round parenthesis, separating them with commas:

```
[88]: x = 3
      y = 5
      "I jumped %s times and did %s sprints" % (x, y)

[88]: 'I jumped 3 times and did 5 sprints'
```

We can put as many variables as we want, also non-numerical ones:

⁹⁶ <https://realpython.com/python-formatted-output/>

⁹⁷ <https://realpython.com/python-formatted-output/#the-python-formatted-string-literal-f-string>

```
[89]: x = 3
      y = 5
      prize = 'Best Athlet in Town'
      "I jumped %s times, did %s sprints and won the prize '%s'" % (x,y,prize)
[89]: "I jumped 3 times, did 5 sprints and won the prize 'Best Athlet in Town'"
```

Exercise - supercars

You've got some money, so you decide to buy two models of supercars. Since you already know accidents are on the way, for each model you will buy as many cars as there are characters in each model name.

Write some code which stores in the string `s` the number of cars you will buy.

Example - given:

```
car1 = 'Jaguar'
car2 = 'Ferrari'
```

After your code, it should show:

```
>>> s
'I will buy 6 Jaguar and 7 Ferrari supercars'
```

- **USE %s placeholders**

```
[90]: car1, car2 = 'Jaguar','Ferrari'      # I will buy 6 Jaguar and 7 Ferrari supercars
      #car1, car2 = 'Porsche','Lamborghini' # I will buy 7 Porsche and 11 Lamborghini
      ↪supercars

      # write here

      s = 'I will buy %s %s and %s %s supercars' % (len(car1), car1, len(car2), car2)
      print(s)

      I will buy 6 Jaguar and 7 Ferrari supercars
```

4.3.15 References

- Think Python, Chapter 8, Strings⁹⁸
- Think Python, Chapter 9, Word play⁹⁹
- Some extra for people wanting to do *text mining*;, have a look at **NLTK library**¹⁰⁰

⁹⁸ <http://greenteapress.com/thinkpython2/html/thinkpython2009.html>

⁹⁹ <http://greenteapress.com/thinkpython2/html/thinkpython2010.html>

¹⁰⁰ <https://www.nltk.org/>

4.3.16 Continue

Go on reading notebook [Strings 2 - operators](#)¹⁰¹

[]:

4.4 Strings 2 - operators

4.4.1 Download exercises zip

Browse files online¹⁰²

Python offers several operators to work with strings:

Operator	Use	Result	Meaning
<code>len</code>	<code>len(str)</code>	<code>int</code>	Returns the length of the string
<code>concatenation</code>	<code>str + str</code>	<code>str</code>	Concatenate two strings
<i>inclusion</i> <code>_</code>	<code>str in str</code>	<code>bool</code>	Controls whether the string is contained inside another string
<i>indexing</i>	<code>str[int]</code>	<code>str</code>	Reads the character at the specified index
<i>slice</i>	<code>str[int : int]</code>	<code>str</code>	Extracts a sub-string
<i>equality</i>	<code>==, !=</code>	<code>bool</code>	Checks whether strings are equal or different
<i>comparisons</i>	<code><, <=, >, >=</code>	<code>bool</code>	Performs lexicographic comparison
<i>replication</i>	<code>str * int</code>	<code>str</code>	Replicate the string

4.4.2 What to do

1. Unzip [exercises zip](#) in a folder, you should obtain something like this:

```
strings
strings1.ipynb
strings1-sol.ipynb
strings2.ipynb
strings2-sol.ipynb
strings3.ipynb
strings3-sol.ipynb
strings4.ipynb
strings4-sol.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 `strings2.ipynb`
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 ☆☆☆

Shortcut keys:

¹⁰¹ <https://en.softpython.org/strings/strings2-sol.html>

¹⁰² <https://github.com/DavidLeoni/softpython-en/tree/master/strings>

- 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 afterwards, press `Alt + Enter`
- If the notebooks look stuck, try to select `Kernel -> Restart`

4.4.3 Reading characters

A string is a sequence of characters, and often we might want to access a single character by specifying the position of the character we are interested in.

It's important to remember that the position of characters in strings start from 0. For reading a character in a certain position, we need to write the string followed by square parenthesis and specify the position inside. Examples:

```
[2]: 'park'[0]
```

```
[2]: 'p'
```

```
[3]: 'park'[1]
```

```
[3]: 'a'
```

```
[4]: #0123  
     'park'[2]
```

```
[4]: 'r'
```

```
[5]: #0123  
     'park'[3]
```

```
[5]: 'k'
```

If we try to go beyond the last character, we will get an error:

```
#0123  
'park'[4]  
-----  
IndexError                                Traceback (most recent call last)  
<ipython-input-106-b8f1f689f0c7> in <module>  
      1 #0123  
----> 2 'park'[4]  
  
IndexError: string index out of range
```

Before we used a string by specifying it as a literal, but we can also use variables:

```
[6]: #01234  
     x = 'cloud'
```

```
[7]: x[0]
```

```
[7]: 'c'
```

```
[8]: x[2]
```

```
[8]: 'o'
```


How is represented the character we've just read? If you noticed, it is between quotes like if it were a string. Let's check:

```
[9]: type(x[0])
[9]: str
```

It's really a string. To somebody this might come as a surprise, also from a philosophical standpoint: Python strings are made of... strings! Other programming languages may use a specific type for the single character, but Python uses strings to be able to better manage complex alphabets as, for example, japanese.

QUESTION: Let's suppose `x` is *any* string. If we try to execute this code:

```
x[0]
```

we will get:

1. always a character
2. always an error
3. sometimes a character, sometimes an error according to the string

ANSWER: 3: we might obtain an error with the empty string (try it)

QUESTION: Let's suppose `x` is an empty string. If we try to execute this code:

```
x[len(x)]
```

we will get:

1. always a character
2. always an error
3. sometimes a character, sometimes an error according to the string at hand

ANSWER: 2: since indexing starts from 0, `len` always gives us a number which is the biggest usable index plus one.

Exercise - alternate

Given two strings both of length 3, print a string which alternates characters from both strings. Your code must work with any string of this length

Example - given:

```
x="say"
y="hi!"
```

it should print:

```
shaiy!
```

```
[10]: # write here

x="say"
y="hi!"
print(x[0] + y[0] + x[1] + y[1] + x[2] + y[2])

shaiy!
```

Negative indexes

In Python we can also use negative indexes, which instead to start from *the beginning* they start *from the end*:

```
[11]: #4321
      "park"[-1]
```

```
[11]: 'k'
```

```
[12]: #4321
      "park"[-2]
```

```
[12]: 'r'
```

```
[13]: #4321
      "park"[-3]
```

```
[13]: 'a'
```

```
[14]: #4321
      "park"[-4]
```

```
[14]: 'p'
```

If we go one step beyond, we get an error:

```
#4321
"park"[-5]

-----
IndexError                                Traceback (most recent call last)
<ipython-input-126-668d8a13a324> in <module>
----> 1 "park"[-5]

IndexError: string index out of range
```

QUESTION: Suppose `x` is a NON-empty string. What do we get with the following expression?

```
x[-len(x)]
```

1. always a character
2. always an error
3. sometimes a character, sometime an error according to the string

ANSWER: 1. (we supposed the string is never empty)

QUESTION: Suppose `x` is a some string (possibly empty), the expressions

```
x[len(x) - 1]
```

and

```
x[-len(x)]
```

are equivalent ? What do they do ?

ANSWER: the expressions are equivalent: if the string is empty both produce an error, if it is full both give the last character

QUESTION: If `x` is a non-empty string, what does the following expression produce? Can we simplify it to a shorter one?

```
(x + x) [len(x)]
```

ANSWER: it's the same as `x[0]`

QUESTION: If `x` is a non-empty string, what does the following expression produce? An error? Something else? Can we simplify it?

```
'park' [0] [0]
```

ANSWER: We know that `'park' [0]` produces a character, but we also know that in Python characters extracted from strings are also strings of length 1. So, if after the expression `"park" [0]` which produces the string `'p'` we add another `[0]` it's like we were writing `'p' [0]`, which returns the zeroth character found in the string in the string `'p'`, that is `'p'` itself.

QUESTION: If `x` is a non-empty string, what does the following expression produce? An error? Something else? Can we simplify it?

```
(x[0]) [0]
```

ANSWER: `x[0]` is an expression which produces the first character of the string `x`. In Python, we can place expressions among parenthesis whenever we want. So in this case the parenthesis don't produce any effect, and the expression becomes equivalent to `x[0] [0]` which as we've seen before it's the same as writing `x[0]`

4.4.4 Substitute characters

We said strings in Python are immutable. Suppose we have a string like this:

```
[15]: #01234
x = 'port'
```

and, for example, we want to change the character at position 2 (in this case, the `r`) into an `s`. What do we do?

We might be tempted to write like the following, but Python would punish us with an error:

```
x[2] = 's'

-----
TypeError                                 Traceback (most recent call last)
<ipython-input-113-e5847c6fa4bf> in <module>
----> 1 x[2] = 's'

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

The correct solution is assigning a completely new string to `x`, obtained by taking pieces from the previous one:

```
[16]: x = x[0] + x[1] + 's' + x[3]
```

```
[17]: x
```

```
[17]: 'post'
```

If seeing `x` to the right of equal sign baffles you, we can decompose the code like this and it will work the same way:

```
[18]: x = "port"
      y = x
      x = y[0] + y[1] + 's' + y[3]
```

Try it in Python Tutor:

```
[19]: x = "port"
      y = x
      x = y[0] + y[1] + 's' + y[3]

      jupman.pytut()

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

4.4.5 Slice

We might want to read only a subsequence which starts from a position and ends up in another one. For example, suppose we have:

```
[20]:      #0123456789
      x = 'mercantile'
```

and we want to extract the string 'canti', which starts at index 3 **included**. We might extract the single characters and concatenate them with + sign, but we would write a lot of code. A better option is to use the so-called [slices](#)¹⁰³: simply write the string followed by square parenthesis containing only start index (**included**), a colon, and finally end index (**excluded**):

```
[21]:      #0123456789
      x = 'mercantile'

      x[3:8]      # note the : inside start and end indexes

[21]: 'canti'
```

WARNING: Extracting with slices DOES NOT modify the original string !!

Let's see an example:

```
[22]:      #0123456789
      x = 'mercantile'

      print('          x is', x)
      print('The slice x[3:8] is', x[3:8])
      print('          x is', x)      # note x continues to point to old string!

          x is mercantile
The slice x[3:8] is canti
          x is mercantile
```

QUESTION: if `x` is any string of length at least 5, what does this code produce? An error? It works? Can we shorten it?

¹⁰³ <http://greenteapress.com/thinkpython2/html/thinkpython2009.html#sec95>

```
x[3:4]
```

ANSWER: If the string has length at least 5, can might have a situation like this:

```
#01234
x = 'abcde'
```

The slice `x[3:4]` will extract from position 3 **included** until position 4 **excluded**, so as a matter of fact it will extract only one character from position 3. So the code is equivalent to `x[3]`

Da fare - garalampog

Write some code to extract and print `alam` from the string `"garalampog"`. Try guessing the correct indexes.

```
[23]: x = "garalampog"

# write here

#      0123456789
print(x[3:7])

alam
```

Exercise - ifEweEfav lkSD lkWe

Write some code to extract and print `kD` from the string `"ifE\te\nfav lkD lkWe"`. Be careful of spaces and special characters (before you might want to print `x`). Try guessing correct indexes.

```
[24]: x = "ifE\te\nfav lkD lkWe"

# write here

#      0123 45 67890123456789
#x = "ifE\te\nfav lkD lkWe"

print(x[12:14])

kD
```

Slices - limits

Whenever we use slice we must be careful with index limits. Let's see how they behave:

```
[25]: #012345
"chair"[0:3] # from index 0 *included* to 3 *excluded*

[25]: 'cha'
```

```
[26]: #012345
"chair"[0:4] # from index 0 *included* to 4 *excluded*

[26]: 'chai'
```

```
[27]: #012345
      "chair"[0:5]  # from index 0 *included* to 5 *excluded*

[27]: 'chair'
```

```
[28]: #012345
      "sedia"[0:6]  # if we go beyond string length Python doesn't complain

[28]: 'sedia'
```

QUESTION: if `x` is any string (also empty), what does this expression do? Can it give an error? Does it return something useful?

```
x[0:len(x)]
```

ANSWER: It always returns a NEW copy of the whole string, because it starts from index 0 *included* and ends at index `len(x)` *excluded*. It also works with the empty string, as `' '[0:len(' ')]` is equivalent to `' '[0:0]` that is a substring from 0 *included* to 0 *excluded*, so we don't take any character and we do not go beyond string limits. Actually, even if we went beyond, we wouldn't upset Python (try writing `' '[0:100]`)

Slice - Omitting limits

If we want, it's possible to omit the starting index, in this case Python will suppose it's a 0:

```
[29]: #0123456789
      "catamaran"[:3]

[29]: 'cat'
```

It's also possible to omit the ending index, in that case Python will extract until the end of the string:

```
[30]: #0123456789
      "catamaran"[3:]

[30]: 'amaran'
```

By omitting both indexes we obtain the full string:

```
[31]: "catamaran"[:]

[31]: 'catamaran'
```

Exercise - ysterymyster

Write some code that given a string `x` prints the string composed with all the characters of `x` except the first one, followed by all characters of `x` except the last one.

- your code must work with any string

Example 1 - given:

```
x = "mystery"
```

must print:

```
ysterymyster
```

Example 2 - given:

```
x = "talking"
```

must print:

alkingtalkin

```
[32]: x = "mystery"
      #x = "talking"

      # write here

      print(x[1:] + x[0:len(x)-1])

      ysterymyster
```

Slice - negative limits

If we want, it's also possible to set negative limits, although it's not always intuitive:

```
[33]: #0123456

      "vegetal"[3:0]    # from index 3 to positive indexes <= 3 doesn't produce anything
```

```
[33]: ''
```

```
[34]: #0123456

      "vegetal"[3:1]    # from index 3 to positive indexes <= 3 doesn't produce anything
```

```
[34]: ''
```

```
[35]: #0123456

      "vegetal"[3:2]    # from index 3 to positive indexes <= 3 doesn't produce anything
```

```
[35]: ''
```

```
[36]: #0123456

      "vegetal"[3:3]    # from index 3 to positive indexes <= 3 doesn't produce anything
```

```
[36]: ''
```

Let's see what happens with negative indexes:

```
[37]: #0123456    positive indexes
      #7654321    negative indexes
      "vegetal"[3:-1]
```

```
[37]: 'eta'
```

```
[38]: #0123456    positive indexes
      #7654321    negative indexes
      "vegetal"[3:-2]
```

```
[38]: 'et '
```

```
[39]: #0123456    positive indexes
      #7654321    negative indexes
      "vegetal"[3:-3]
```

```
[39]: 'e'
```

```
[40]: #0123456   positive indexes  
      #7654321   negative indexes  
      "vegetal"[3:-4]
```

```
[40]: ''
```

```
[41]: #0123456   positive indexes  
      #7654321   negative indexes  
      "vegetal"[3:-5]
```

```
[41]: ''
```

Exercise - javarnanda

Given a string `x`, write some code to extract and print its last 3 characters joined to the first 3.

- Your code should work for any string of length equal or greater than 3

Example 1 - given:

```
x = "javarnanda"
```

it should print:

```
javnda
```

Example 2 - given:

```
x = "abcd"
```

it should print:

```
abcbcd
```

```
[42]: x = "javarnanda"  
      #x = "abcd"  
  
      # write here  
  
      print(x[:3] + x[-3:])  
  
      javnda
```

Slice - modifying

Suppose to have the string

```
[43]: #0123456789  
      s = "the table is placed in the center of the room"
```

and we want to change `s` assignment so it becomes associated to the string:


```
#0123456789
"the chair is placed in the center of the room"
```

Since both strings are similar, we might be tempted to only redefine the character sequence which corresponds to the word "table", which goes from index 4 included to index 9 excluded:

```
s[4:9] = "chair"    # WARNING! WRONG!

-----
TypeError                                 Traceback (most recent call last)
<ipython-input-57-0de7363c6882> in <module>
----> 1 s[4:9] = "chair"    # WARNING! WRONG!

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

Sadly, we would receive an error, because as repeated many times strings are IMMUTABLE, so we cannot select a chunk of a particular string and try to change the original string. What we can do instead is to build a NEW string from pieces of the original string, concatenates the desired characters and associates the result to the variable of which we want to modify the assignment:

```
[44]: #0123456789
s = "the table is placed in the center of the room"
s = s[0:4] + "chair" + s[9:]
print(s)

the chair is placed in the center of the room
```

When Python finds the line

```
s = s[0:4] + "chair" + s[9:]
```

FIRST it calculates the result on the right of the =, and THEN associates the result to the variable on the left. In the expression on the right only NEW strings are generated, which once built can be assigned to variable s

Exercise - the run

Write some code such that when given the string s

```
s = 'The Gold Rush has begun.'
```

and some variables

```
what = 'Atom'
happened = 'is over'
```

substitutes the substring 'Gold' with the string in the variable what and substitutes the substring 'has begun' with the string in the variable happened.

After executing your code, the string associated to s should be

```
>>> print(s)
"The Atom Rush is over."
```

- **DON'T** use constant characters in your code, i.e. dots '.' aren't allowed !

```
[45]: #01234567890123456789012345678
s = 'The Gold Rush has begun.'
what = 'Atom'
happened = 'is over'

# write here

s = s[0:4] + what + s[8:14] + happened + s[23:]
print(s)

The Atom Rush is over.
```

4.4.6 in operator

To check if a string is contained in another one, we use the `in` operator.

Note the result of this expression is a boolean:

```
[46]: 'the' in 'Singing in the rain'
```

```
[46]: True
```

```
[47]: 'si' in 'Singing in the rain' # in operator is case-sensitive
```

```
[47]: False
```

```
[48]: 'Si' in 'Singing in the rain'
```

```
[48]: True
```

Exercise - contained 1

You are given two strings `x` and `y`, and a third `z`. Write some code which prints `True` if `x` and `y` are both contained in `z`.

Example 1 - given:

```
x = 'cad'
y = 'ra'
z = 'abracadabra'
```

it should print:

```
True
```

Example 2 - given:

```
x = 'zam'
y = 'ra'
z = 'abracadabra'
```

it should print:

```
False
```

```
[49]: x,y,z = 'cad','ra','abracadabra'    # True
      #x,y,z = 'zam','ra','abracadabra'  # False

      # write here

      print((x in z) and (y in z))

True
```

Exercise - contained 2

Given three strings x, y, z, write some code which prints True if the string x is contained in at least one of the strings y or z, otherwise prints False

- your code should work with any set of strings

Example 1 - given:

```
x = "ope"
y = "honesty makes for long friendships"
z = "I hope it's clear enough"
```

it should print:

True

Example 2 - given:

```
x = "nope"
y = "honesty makes for long friendships"
z = "I hope it's clear enough"
```

it should print:

False

Example 3 - given:

```
x = "cle"
y = "honesty makes for long friendships"
z = "I hope it's clear enough"
```

it should show:

True

```
[50]: x,y,z = "ope","honesty makes for long friendships","I hope it's clear enough"    # True
      #x,y,z = "nope","honesty makes for long friendships","I hope it's clear enough"    #_
      ↪False
      #x,y,z = "cle","honesty makes for long friendships","I hope it's clear enough"    # True

      # write here

      print((x in y) or (x in z))

True
```

4.4.7 Comparisons

Python offers us the possibility to perform a *lexicographic comparison* among strings, like we would when placing names in an address book. Although sorting names is something intuitive we often do, we must be careful about special cases.

First, let's determine when two strings are equal.

Equality operators

To check whether two strings are equal, you can use the operator `==` which as result produces the boolean `True` or `False`

WARNING: `==` is written with TWO equal signs !!!

```
[51]: "dog" == "dog"
```

```
[51]: True
```

```
[52]: "dog" == "wolf"
```

```
[52]: False
```

Equality operator is case-sensitive:

```
[53]: "dog" == "DOG"
```

```
[53]: False
```

To check whether two strings are NOT equal, we can use the operator `!=`, which we can expect to behave exactly as the opposite of `==`:

```
[54]: "dog" != "dog"
```

```
[54]: False
```

```
[55]: "dog" != "wolf"
```

```
[55]: True
```

```
[56]: "dog" != "DOG"
```

```
[56]: True
```

As an alternative, we might use the operator `not`:

```
[57]: not "dog" == "dog"
```

```
[57]: False
```

```
[58]: not "wolf" == "dog"
```

```
[58]: True
```

```
[59]: not "dog" == "DOG"
```

```
[59]: True
```

QUESTION: what does the following code print?

```
x = "river" == "river"
print(x)
```

ANSWER: When Python encounters `x = "river" == "river"` it sees an assignment, and associates the result of the expression `"river" == "river"` to the variable `x`. So **FIRST** it calculates the expression `"river" == "river"` which produces the boolean `True`, and **THEN** associates the value `True` to the variable `x`. Finally `True` is printed.

QUESTION: for each of the following expressions, try to guess whether it produces `True` or `False`

1. `'hat' != 'Hat'`
2. `'hat' == 'HAT'`
3. `'choralism'[2:5] == 'contemporary'[7:10]`
4. `'AlAbAmA'[4:] == 'aLaBaMa'`
5. `'bright'[9:20] == 'dark'[10:15]`
6. `'optical'[-1] == 'crystal'[-1]`
7. `('hat' != 'jacket') == ('trousers' != 'bow')`
8. `('stra' in 'stradivarius') == ('div' in 'digital divide')`
9. `len('note') in '5436'`
10. `str(len('note')) in '5436'`
11. `len('posters') in '5436'`
12. `str(len('posters')) in '5436'`

Exercise - statist

Write some code which prints `True` if a word begins with the same two characters it ends with.

- Your code should work for any word

```
[60]: word = 'statist'      # True
      #word = 'baobab'     # False
      #word = 'maxima'     # True
      #word = 'karma'      # False

      # write here

      print(word[:2] == word[-2:len(word)])

True
```

Comparing characters

Characters have an inherent order we can exploit. Let's see an example:

```
[61]: 'a' < 'g'
```

```
[61]: True
```

another one:

```
[62]: 'm' > 'c'
```

```
[62]: True
```

They sound reasonable comparisons! But what about this (notice capital 'Z')?

```
[63]: 'a' < 'Z'
```

```
[63]: False
```

Maybe this doesn't look so obvious. And what if we get creative and compare with symbols such as square bracket or Unicode¹⁰⁴ hearts ??

```
[64]: 'a' > '♥'
```

```
[64]: False
```

To determine how to deal with this special cases, we must remember ASCII¹⁰⁵ assigns a position number to each character, defining as a matter of fact *an ordering* between all its characters.

If we want to know the corresponding number of a character, we can use the function `ord`:

```
[65]: ord('a')
```

```
[65]: 97
```

```
[66]: ord('b')
```

```
[66]: 98
```

```
[67]: ord('z')
```

```
[67]: 122
```

If we want to go the other way, given a position number we can obtain the corresponding character with `chr` function:

```
[68]: chr(97)
```

```
[68]: 'a'
```

Uppercase characters have different positions:

```
[69]: ord('A')
```

```
[69]: 65
```

```
[70]: ord('Z')
```

¹⁰⁴ <https://en.softpython.org/strings/strings1-sol.html#Unicode-characters>

¹⁰⁵ <https://en.softpython.org/strings/strings1-sol.html#ASCII-characters>

```
[70]: 90
```

EXERCISE: Using the functions above, try to find which characters are *between* capital Z and lowercase a

```
[71]: # write here
      #print(chr(91),chr(92), chr(93),chr(94), chr(95),chr(96))
```

The ordering allows us to perform *lexicographic comparisons* between single characters:

```
[72]: 'a' < 'b'
```

```
[72]: True
```

```
[73]: 'g' >= 'm'
```

```
[73]: False
```

Also, since Unicode character set *includes* ASCII, the ordering of ASCII characters can be used to safely compare them against unicode characters, so comparing characters or their `ord` should be always equivalent:

```
[74]: ord('a')    # ascii
```

```
[74]: 97
```

```
[75]: ord('♥')    # unicode
```

```
[75]: 9829
```

```
[76]: 'a' > '♥'
```

```
[76]: False
```

```
[77]: ord('a') > ord('♥')
```

```
[77]: False
```

Python also offers lexicographic comparisons on strings with more than one character. To understand what the expected result should be, we must distinguish among several cases, though:

- strings of equal / different length
- strings with same / mixed case

Let's begin with same length strings:

```
[78]: 'mario' > 'luigi'
```

```
[78]: True
```

```
[79]: 'mario' > 'wario'
```

```
[79]: False
```

```
[80]: 'Mario' > 'Wario'
```

```
[80]: False
```

```
[81]: 'mario' > 'Wario'  # capital case is *before* lowercase in ASCII
```

```
[81]: True
```

Comparing different lengths

Short strings which are included in longer ones come first in the ordering:

```
[82]: 'troll' < 'trolley'
```

```
[82]: True
```

If they only share a prefix with a longer string, Python compares characters after the common prefix, in this case it detects that `s` is greater than corresponding `e`:

```
[83]: 'trolls' < 'trolley'
```

```
[83]: False
```

Exercise - Character intervals

You are given a couple of strings `i1` and `i2` of two characters each.

We suppose they represent character intervals: the first character of an interval always has order number lower or equal than the second.

There are five possibilities: either the first interval ‘is contained in’, or ‘contains’, or ‘overlaps’, or ‘is before’ or ‘is after’ the second interval. Write some code which tells which containment relation we have.

Example 1 - given:

```
i1 = 'gm'
i2 = 'cp'
```

Your program should print:

```
gm is contained in cp
```

To see why, you can look at this little representation (you **don't** need to print this!):

```
  c   g       m   p
a b c d e f g h i j k l m n o p q r s t u v w x y z
```

Example 2 - given:

```
i1 = 'mr'
i2 = 'pt'
```

Your program should print:

```
mr overlaps pt
```

because `mr` is not contained nor contains nor completely precedes nor completely follows `pt` (you **don't** need to print this!):

```
      m   p r t
a b c d e f g h i j k l m n o p q r s t u v w x y z
```


- if `i1` interval coincides with `i2`, it is considered as containing `i2`
- **DO NOT** use cycles nor `if`
- **HINT:** to satisfy above constraint, think about [booleans evaluation order](#)¹⁰⁶, for example the expression

```
'g' >= 'c' and 'm' <= 'p' and 'is contained in'
```

produces as result the string `'is contained in'`

```
[84]: i1,i2 = 'gm','cp'      # gm is contained in cp
      #i1,i2 = 'dh','dh'    # gm is contained in cp #(special case)
      #i1,i2 = 'bw','dq'    # bw contains dq
      #i1,i2 = 'ac','bd'    # ac overlaps bd
      #i1,i2 = 'mr','pt'    # mr overlaps pt
      #i1,i2 = 'fm','su'    # fm is before su
      #i1,i2 = 'xz','pq'    # xz is after pq

      # write here
      res = (i1[0] >= i2[0] and i1[1] <= i2[1] and 'is contained in') \
            or (i1[0] < i2[0] and i1[1] > i2[1] and 'contains') \
            or (i1[0] >= i2[0] and i1[0] <= i2[1] and 'overlaps') \
            or (i1[1] >= i2[0] and i1[1] <= i2[1] and 'overlaps') \
            or (i1[1] < i2[0] and 'is before') \
            or (i1[0] > i2[1] and 'is after')

      #print(i1, res, i2)
```

4.4.8 Replication operator

With the operator `*` you can replicate a string `n` times, for example:

```
[85]: 'beer' * 4
[85]: 'beerbeerbeerbeer'
```

Note a NEW string is created, without tarnishing the original:

```
[86]: drink = "beer"

[87]: print(drink * 4)
      beerbeerbeerbeer

[88]: drink
[88]: 'beer'
```

¹⁰⁶ <https://en.softpython.org/basics/basics-sol.html#Booleans---Evaluation-order>

Exercise - za za za

Given a `syllable` and a `phrase` which terminates with a character `n` as a digit, write some code which prints a string with the `syllable` repeated `n` times, separated by spaces.

- Your code must work with any string assigned to `syllable` and `phrase`

Example - given:

```
phrase = 'the number 7'
syllable = 'za'
```

after you code, it should print:

```
za za za za za za za
```

```
[89]: phrase = 'the number 7'
      syllable = 'za'          # za za za za za za za
      #phrase = 'Give me 5'    # za za za za za

      # write here

      print((syllable + ' ') * (int(phrase[-1])))

      za za za za za za za
```

4.4.9 Continue

Go on reading notebook [Strings 3 - methods](#)¹⁰⁷

```
[ ]:
```

4.5 Strings 3 - methods

4.5.1 Download exercises zip

[Browse files online](#)¹⁰⁸

Every data type has associated particular methods for that type, let's see those associated to type string (`str`)

WARNING: ALL string methods ALWAYS generate a NEW string

The original string object is NEVER changed (because strings are immutable).

¹⁰⁷ <https://en.softpython.org/strings/strings3-sol.html>

¹⁰⁸ <https://github.com/DavidLeoni/softpython-en/tree/master/strings>

Result	Method	Meaning
str	<i>str.upper()</i>	Return the string with all characters uppercase
str	<i>str.lower()</i>	Return the string with all characters lowercase
str	<i>str.capitalize()</i>	Return the string with the first uppercase character
str	<i>str.strip(str)</i>	Remove strings from the sides
str	<i>str.lstrip(str)</i>	Remove strings from left side
str	<i>str.rstrip(str)</i>	Remove strings from right side
str	<i>str.replace(str, str)</i>	Substitute substrings
bool	<i>str.startswith(str)</i>	Checks if the string begins with another one
bool	<i>str.endswith(str)</i>	Checks whether the string ends with another one
int	<i>str.find(str)</i>	Return the first position of a substring starting from the left
int	<i>str.rfind(str)</i>	Return the first position of a substring starting from the right
int	<i>str.count(str)</i>	Counts the number of occurrences of a substring
bool	<i>str.isalpha(str)</i>	Checks if all characters in the string are alphabetic
bool	<i>str.isdigit(str)</i>	Checks if all characters in the string are digits

Note: the list is not exhaustive, here we report only the ones we use in the book. For the full list see [Python documentation](#)¹⁰⁹

4.5.2 What to do

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

```
strings
  strings1.ipynb
  strings1-sol.ipynb
  strings2.ipynb
  strings2-sol.ipynb
  strings3.ipynb
  strings3-sol.ipynb
  strings4.ipynb
  strings4-sol.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 `strings3.ipynb`
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 ☆☆☆

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 create a new cell afterwards, press `Alt + Enter`
- If the notebooks look stuck, try to select `Kernel -> Restart`

¹⁰⁹ <https://docs.python.org/3/library/stdtypes.html#string-methods>

4.5.3 Example - upper

A method is a function of an object that takes as input the object to which it is applied and does some calculation.

The type of the string (`str`) has predefined methods like `str.upper()` which can be applied to other string objects (i.e.: `'hello'` is a string object)

The method `str.upper()` takes the string to which it is applied, and creates a NEW string in which all the characters are in uppercase. To apply a method like `str.upper()` to the particular string object `'hello'`, we must write:

```
'hello'.upper()
```

First we write the object on which apply the method (`'hello'`), then a dot `.`, and afterwards the method name followed by round parenthesis. The brackets can also contain further parameters according to the method.

Examples:

```
[2]: 'hello'.upper()
```

```
[2]: 'HELLO'
```

```
[3]: "I'm important".upper()
```

```
[3]: "I'M IMPORTANT"
```

WARNING: like ALL string methods, the original string object on which the method is called does NOT get modified.

Example:

```
[4]: x = "hello"
     y = x.upper()    # generates a NEW string and associates it to the variables y
```

```
[5]: x                # x variable is still associated to the old string
```

```
[5]: 'hello'
```

```
[6]: y                # y variable is associated to the new string
```

```
[6]: 'HELLO'
```

Have a look now at the same example in Python Tutor:

```
[7]: x = "hello"
     y = x.upper()
     print(x)
     print(y)
```

```
jupman.pytut()
```

```
hello
HELLO
```

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

4.5.4 Exercise - walking

Write some code which given a string `x` (i.e.: `x='walking'`) prints twice the row:

```
walking WALKING walking WALKING
walking WALKING walking WALKING
```

- **DO NOT** create new variables
- your code must work with any string

```
[8]: x = 'walking'

print(x, x.upper(), x, x.upper())
print(x, x.upper(), x, x.upper())

walking WALKING walking WALKING
walking WALKING walking WALKING
```

Help: If you are not sure about a method (for example, `strip`), you can ask Python for help this way:

WARNING: when using help, DON'T put parenthesis after the method name !!

```
[9]: help("hello".strip)

Help on built-in function strip:

strip(...) method of builtins.str instance
    S.strip([chars]) -> str

    Return a copy of the string S with leading and trailing
    whitespace removed.
    If chars is given and not None, remove characters in chars instead.
```

4.5.5 lower method

Return the string with all lowercase characters

```
[10]: my_string = "HEllo WorLd"

another_string = my_string.lower()

print(another_string)

hello world
```

```
[11]: print(my_string)  # didn't change

HEllo WorLd
```

Exercise - lowermid

Write some code that given any string `x` of odd length, prints a new string like `x` having the mid-character as lowercase.

- your code must work with any string !
- **HINT:** to calculate the position of the mid-character, use integer division with the operator `//`

Example 1 - given:

```
x = 'ADORATION'
```

it should print:

ADORaTION

Example 2 - given:

```
x = 'LEADINg'
```

it should print:

LEADINg

```
[12]:      #012345678
x = 'ADORATION'
#x = 'LEADINg'
k = len(x) // 2

print(x[:k] + x[k].lower() + x[k+1:])

ADORaTION
```

4.5.6 capitalize method

`capitalize()` creates a NEW string having only the FIRST character as uppercase:

```
[13]: "artisan".capitalize()
```

```
[13]: 'Artisan'
```

```
[14]: "premium".capitalize()
```

```
[14]: 'Premium'
```

```
[15]: x = 'goat'
      y = 'goat'.capitalize()
```

```
[16]: x      # x remains associate to the old value
```

```
[16]: 'goat'
```

```
[17]: y      # y is associated to the new string
```

```
[17]: 'Goat'
```

Exercise - Your Excellence

Write some code which given two strings `x` and `y` returns the two strings concatenated, separating them with a space and both as lowercase except the first two characters which must be uppercase

Example 1 - given:

```
x = 'yoUR'
y = 'exCelLenCE'
```

it must print:

```
Your Excellence
```

Example 2 - given:

```
x = 'hEr'
y = 'maJEsty'
```

it must print:

```
Her Majesty
```

```
[18]: x,y = 'yoUR','exCelLenCE'
      #x,y = 'hEr','maJEsty'

      # write here

      print(x.capitalize() + " " + y.capitalize())

      Your Excellence
```

4.5.7 strip method

Eliminates white spaces, tabs and linefeeds from the *sides* of the string. In general, this set of characters is called *blanks*.

NOTE: it does NOT removes *blanks* inside string words! It only looks on the sides.

```
[19]: x = ' \t\n\n\t carpe diem \t ' # we put white space, tab and line feeds at the_
      ↪sides
```

```
[20]: x
```

```
[20]: ' \t\n\n\t carpe diem \t '
```

```
[21]: print(x)
```

```
carpe diem
```

```
[22]: len(x) # remember that special characters like \t and \n occupy 1 character
```

```
[22]: 20
```

```
[23]: y = x.strip()
```

```
[24]: y
[24]: 'carpe diem'

[25]: print(y)
      carpe diem

[26]: len(y)
[26]: 10

[27]: x      # IMPORTANT: x is still associated to the old string !
[27]: ' \t\n\n\t carpe diem \t '
```

Specifying character to strip

If you only want Python to remove some specific character, you can specify them in parenthesis. Let's try to specify only one:

```
[28]: 'salsa'.strip('s')      # not internal `s` is not stripped
[28]: 'alsa'
```

If we specify two or more, Python removes all the characters it can find from the sides

Note the order in which you specify the characters does **not** matter:

```
[29]: 'caustic'.strip('aci')
[29]: 'ust'
```

WARNING: If you specify characters, Python doesn't try anymore to remove blanks!

```
[30]: 'bouquet '.strip('b')    # it won't strip right spaces !
[30]: 'ouquet '

[31]: '\tbouquet '.strip('b')  # ... nor strip left blanks such as tab
[31]: '\tbouquet '
```

According to the same principle, if you specify a space ' ', then Python will **only** remove spaces and won't look for other blanks!!

```
[32]: ' careful! \t'.strip(' ') # strips only on the left!
[32]: 'careful! \t'
```

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

1. `'\ttumultuous\n'.strip()`


```
2. ' a b c '.strip()
```

```
3. '\ta\tb\t'.strip()
```

```
4. '\\tMmm'.strip()
```

```
5. 'sky diving'.strip('sky')
```

```
6. 'anacondas'.strip('sad')
```

```
7. '\nno way '.strip(' ')
```

```
8. '\nno way '.strip('\n')
```

```
9. '\nno way '.strip('\n')
```

```
10. 'salsa'.strip('as')
```

```
11. '\t ACE '.strip('\t')
```

```
12. ' so what? '.strip("")
```

```
13. str(-3+1).strip("+ "+"-")
```

4.5.8 lstrip method

Eliminates white spaces, tab and line feeds from *left side* of the string.

NOTE: does NOT remove *blanks* between words of the string! Only those on left side.

```
[33]: x = '\n \t the street \t '
```

```
[34]: x
```

```
[34]: '\n \t the street \t '
```

```
[35]: len(x)
```

```
[35]: 17
```

```
[36]: y = x.lstrip()
```

```
[37]: y
```

```
[37]: 'the street \t '
```

```
[38]: len(y)
```

```
[38]: 13
```

```
[39]: x          # IMPORTANT: x is still associated to the old string !
[39]: '\n \t the street \t '
```

4.5.9 `rstrip` method

Eliminates white spaces, tab and line feeds from *left side* of the string.

NOTE: does NOT remove *blanks* between words of the string! Only those on right side.

```
[40]: x = '\n \t the lighthouse \t '
```

```
[41]: x
[41]: '\n \t the lighthouse \t '
```

```
[42]: len(x)
[42]: 21
```

```
[43]: y = x.rstrip()
```

```
[44]: y
[44]: '\n \t the lighthouse'
```

```
[45]: len(y)
[45]: 18
```

```
[46]: x          # IMPORTANT: x is still associated to the old string !
[46]: '\n \t the lighthouse \t '
```

Exercise - hatespace

Given a string `x` which may contain some *blanks* (spaces, control characters like `\t` and `\n`, ...) from begin to end, write some code which prints the string without *blanks* and the strings 'START' and 'END' at the sides

Example - given:

```
x = ' \t \n \n hatespace\n \t \n'
```

prints

```
STARThatespaceEND
```

```
[47]: # write here

x = ' \t \n \n hatespace\n \t \n'

print('START' + x.strip() + 'END')

STARThatespaceEND
```

Exercise - Bad to the bone

You have an uppercase string `s` which contains at the sides some stuff you want to remove: punctuation, a lowercase char and some blanks. Write some code to perform the removal

Example - given:

```
char = 'b'
punctuation = '!?.,;'
s = ' \t\n...bbbbbbBAD TO THE BONE\n!'
```

your code should show:

```
'BAD TO THE BONE'
```

- use only `strip` (or `lstrip` and `rstrip`) methods (if necessary, you can do repeated calls)

```
[48]: char = 'b'
      punctuation = '!?.,;'
      s = ' \t\n...bbbbbbBAD TO THE BONE\n!'

      # write here
      s.strip().strip(char + punctuation).strip()

[48]: 'BAD TO THE BONE'
```

4.5.10 replace method

`str.replace` takes two strings and looks in the string on which the method is called for occurrences of the first string parameter, which are substituted with the second parameter. Note it gives back a NEW string with all substitutions performed.

Example:

```
[49]: "the train runs off the tracks".replace('tra', 'ra')
[49]: 'the rain runs off the racks'

[50]: "little beetle".replace('tle', '')
[50]: 'lit bee'

[51]: "talking and joking".replace('ING', 'ed') # it's case sensitive
[51]: 'talking and joking'

[52]: "TALKING AND JOKING".replace('ING', 'ED') # here they are
[52]: 'TALKED AND JOKED'
```

As always with strings, `replace` DOES NOT modify the string on which it is called:

```
[53]: x = "On the bench"

[54]: y = x.replace('bench', 'bench the goat is alive')
```

```
[55]: y
```

```
[55]: 'On the bench the goat is alive'
```

```
[56]: x # IMPORTANT: x is still associated to the old string !
```

```
[56]: 'On the bench'
```

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

```
1. '$£ciao£$'.replace('£','').replace('$','')
```

```
2. '$£ciao£$'.strip('£').strip('$')
```

Exercise - substitute

Given a string `x`, write some code to print a string like `x` but with all occurrences of `bab` substituted by `dada`

Example - given:

```
x = 'kljsfsdbabòkkrbabfej'
```

it should print

```
kljsfsddadaòkkrdadaej
```

```
[57]: # write here
```

```
x = 'kljsfsdbabòkkrbabfej'
print(x.replace('bab', 'dada'))
```

```
kljsfsddadaòkkrdadaej
```

4.5.11 startswith method

`str.startswith` takes as parameter a string and returns `True` if the string before the dot begins with the string passed as parameter. Example:

```
[58]: "the dog is barking in the road".startswith('the dog')
```

```
[58]: True
```

```
[59]: "the dog is barking in the road".startswith('is barking')
```

```
[59]: False
```

```
[60]: "the dog is barking in the road".startswith('THE DOG') # uppercase is different from
↳ lowercase
```

```
[60]: False
```

```
[61]: "THE DOG BARKS IN THE ROAD".startswith('THE DOG') # uppercase is different from
↳ lowercase
```

```
[61]: True
```

Exercise - by Jove

Write some code which given any three strings `x`, `y` and `z`, prints `True` if both `x` and `y` start with string `z`, otherwise prints `False`

Example 1 - given:

```
x = 'by Jove'
y = 'by Zeus'
z = 'by'
```

it should print:

```
True
```

Example 2 - give:

```
x = 'by Jove'
y = 'by Zeus'
z = 'from'
```

it should print:

```
False
```

Example 3 - given:

```
x = 'from Jove'
y = 'by Zeus'
z = 'by'
```

it should print:

```
False
```

```
[62]: x,y,z = 'by Jove','by Zeus','by'      # True
      #x,y,z = 'by Jove','by Zeus','from'  # False
      #x,y,z = 'from Jove','by Zeus','by'  # False

      # write here

      print(x.startswith(z) and y.startswith(z))

      True
```

4.5.12 endswith method

`str.endswith` takes as parameter a string and returns `True` if the string before the dot ends with the string passed as parameter. Example:

```
[63]: "My best wishes".endswith('st wishes')
```

```
[63]: True
```

```
[64]: "My best wishes".endswith('best')
```

```
[64]: False
```

```
[65]: "My best wishes".endswith('WISHES')    # uppercase is different from lowercase
```

```
[65]: False
```

```
[66]: "MY BEST WISHES".endswith('WISHES')    # uppercase is different from lowercase
```

```
[66]: True
```

Exercise - Snobbonis

Given couple names husband and wife, write some code which prints `True` if they share the surname, `False` otherwise.

- assume the surname is always at position 9
- your code must work for any couple husband and wife

```
[67]:          #0123456789          #0123456789
husband, wife = 'Antonio  Snobbonis', 'Carolina Snobbonis' # True
#husband, wife = 'Camillo  De Spaparanzi', 'Matilda  Degli Agi' # False

# write here

print(wife.endswith(husband[9:]))

True
```

4.5.13 count method

The method `count` takes a substring and counts how many occurrences are there in the string before the dot.

```
[68]: "astral stars".count('a')
```

```
[68]: 3
```

```
[69]: "astral stars".count('A')    # it's case sensitive
```

```
[69]: 0
```

```
[70]: "astral stars".count('st')
```

```
[70]: 2
```

Optionally, you can pass a two other parameters to indicate an index to start counting from (included) and where to end (excluded):

```
[71]: #012345678901
      "astral stars".count('a',4)
```

```
[71]: 2
```

```
[72]: #012345678901
      "astral stars".count('a',4,9)
```

```
[72]: 1
```

Exercise - astro money

During 2020 lockdown while looking at the stars above you started feeling... waves. After some thinking, you decided *THEY* want to communicate with you so you set up a dish antenna on your roof to receive messages from aliens. After months of apparent irrelevant noise, one day you finally receive a message you think you can translate. Aliens are *obviously* trying to tell you the winning numbers of lottery!

A message is a sequence of exactly 3 *different* character repetitions, the number of characters in each repetition is a number you will try at the lottery. You frantically start developing the translator to show these lucky numbers on the terminal.

Example - given:

```
s = '$$$$€€€€€!!'
```

it should print:

```
$ € !
4 5 2
```

- **IMPORTANT:** you can assume all sequences have **different** characters
- **DO NOT** use cycles nor comprehensions
- for simplicity assume each character sequence has at most 9 repetitions

```
[73]: #01234567890      # $ € !
      s = '$$$$€€€€€!!' # 4 5 2
                        # a b c
      #s = 'aaabbbbbbbccc' # 3 6 3
                        # H A L
      #s = 'HAL'          # 1 1 1
```

```
# write here
p1 = 0
d1 = s.count(s[p1])
p2 = p1 + d1
d2 = s.count(s[p2])
p3 = p2 + d2
d3 = s.count(s[p3])

print(s[p1],s[p2],s[p3])
print(d1,d2,d3)
```

```
$ € !
4 5 2
```

4.5.14 find method

`find` returns the index of the *first* occurrence of some given substring:

```
[74]: #0123456789012345
      'bingo bongo bong'.find('ong')
```

```
[74]: 7
```

If no occurrence is found, it returns `-1`:

```
[75]: #0123456789012345
      'bingo bongo bong'.find('bang')
```

```
[75]: -1
```

```
[76]: #0123456789012345
      'bingo bongo bong'.find('Bong')    # case-sensitive
```

```
[76]: -1
```

Optionally, you can specify an index from where to start searching (included):

```
[77]: #0123456789012345
      'bingo bongo bong'.find('ong', 10)
```

```
[77]: 13
```

And also where to end (excluded):

```
[78]: #0123456789012345
      'bingo bongo bong'.find('g', 4, 9)
```

```
[78]: -1
```

Exercise - bananas

While exploring a remote tropical region, an ethologist discovers a population of monkeys which appear to have some concept of numbers. They collect bananas in the hundreds which are then traded with coconuts collected by another group. To communicate the quantities of up to 999 bananas, they use a series of exactly three guttural sounds. The ethologist writes down the sequencies and formulates the following theory: each sound is comprised by a sequence of the same character, repeated a number of times. The number of characters in the first sequence is the first digit (the hundreds), the number of characters in the second sequence is the second digit (the decines), while the last sequence represents units.

Write some code which puts in variable `bananas` **an integer** representing the number.

For example - given:

```
s = 'bb bbbbb aaaa'
```

your code should print:

```
>>> bananas
254
>>> type(bananas)
int
```

- **IMPORTANT 1:** different sequences may use the **same** character!

- **IMPORTANT 2: you cannot assume which characters monkeys will use:** you just know each digit is represented by a repetition of the same character
- **DO NOT** use cycles nor comprehensions
- the monkeys have no concept of zero

```
[79]: #0123456789012
s = 'bb bbbbbb aaaa'      # 254
#s = 'ccc cc ccc'        # 323
#s = 'vvv rrrr ww'       # 342
#s = 'cccc h jjj'        # 413
#s = '??? ?????? ?????' # 364 (you could get *any* weird character, also unicode ...)

# write here
p1 = s.find(' ')
bananas = len(s[:p1])*100
p2 = s.find(' ', p1+1)
bananas += len(s[p1+1:p2])*10
bananas += len(s[p2+1:])*1

#print('The bananas are',bananas)
#type(bananas)
```

4.5.15 rfind method

Like *find method*, but search starts from the right.

4.5.16 isalpha method

The method `isalpha` returns `True` if all characters in the string are alphabetic:

```
[80]: 'CoralReel'.isalpha()
```

```
[80]: True
```

Numbers are not considered alphabetic:

```
[81]: 'Route 666'.isalpha()
```

```
[81]: False
```

Also, blanks are *not* alphabetic:

```
[82]: 'Coral Reel'.isalpha()
```

```
[82]: False
```

... nor punctuation:

```
[83]: '!'.isalpha()
```

```
[83]: False
```

... nor weird Unicode stuff:

```
[84]: '♥'.isalpha()
```

```
[84]: False
```

```
[85]: ''.isalpha()
```

```
[85]: False
```

4.5.17 isdigit method

isdigit method returns True if a string is only composed of digits:

```
[86]: '391'.isdigit()
```

```
[86]: True
```

```
[87]: '400m'.isdigit()
```

```
[87]: False
```

Floating point and scientific notations are not recognized:

```
[88]: '3.14'.isdigit()
```

```
[88]: False
```

```
[89]: '4e29'.isdigit()
```

```
[89]: False
```

4.5.18 Other exercises

QUESTION: For each following expression, try to find the result.

1. `'gUrP'.lower() == 'GuRp'.lower()`

2. `'NaNo'.lower() != 'nAnO'.upper()`

3. `'O' + 'ortaggio'.replace('o', '\t \n').strip() + 'O'`

4. `'DaDo'.replace('D', 'b') in 'barbados'`

4.5.19 Continue

Go on reading notebook [Strings 4 - other exercises](#)¹¹⁰

```
[ ]:
```

¹¹⁰ <https://en.softpython.org/strings/strings4-sol.html>

4.6 Error handling and testing solutions

4.6.1 Download exercises zip

Browse files online¹¹¹

4.6.2 Introduction

In this notebook we will try to understand what our program should do when it encounters unforeseen situations, and how to test the code we write.

For some strange reason, many people believe that computer programs do not need much error handling nor testing. Just to make a simple comparison, would you ever drive a car that did not undergo scrupolous checks? We wouldn't.

What to do

- unzip exercises in a folder, you should get something like this:

```
errors-and-testing
  errors-and-testing.ipynb
  errors-and-testing-sol.ipynb
  jupman.py
```

WARNING 1: to correctly visualize the notebook, it MUST be in an unzipped folder !

- 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 `strings/strings.ipynb`

WARNING 2: DO NOT use the *Upload* button in Jupyter, instead navigate to the unzipped folder while in Jupyter browser!

- Go on reading that notebook, and follow instnctions inside.

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 afterwards, press `Alt + Enter`
- If the notebooks look stuck, try to select `Kernel -> Restart`

¹¹¹ <https://github.com/DavidLeoni/sciprog-ds/tree/master/errors-and-testing>

4.6.3 Unforeseen situations

It is evening, there is to party for a birthday and they asked you to make a pie. You need the following steps:

1. take milk
2. take sugar
3. take flour
4. mix
5. heat in the oven

You take the milk, the sugar, but then you discover there is no flour. It is evening, and there aren't open shops. Obviously, it makes no sense to proceed to point 4 with the mixture, and you have to give up on the pie, telling the guest of honor the problem. You can only hope she/he decides for some alternative.

Translating everything in Python terms, we can ask ourselves if during the function execution, when we find an unforeseen situation, is it possible to:

1. **interrupt** the execution flow of the program
2. **signal** to whoever called the function that a problem has occurred
3. **allow to manage** the problem to whoever called the function

The answer is yes, you can do it with the mechanism of **exceptions** (`Exception`)

`make_problematic_pie`

Let's see how we can represent the above problem in Python. A basic version might be the following:

```
[2]: def make_problematic_pie(milk, sugar, flour):  
    """ Suppose you need 1.3 kg for the milk, 0.2kg for the sugar and 1.0kg for the_  
    ↪flour  
  
    - takes as parameters the quantities we have in the sideboard  
    """  
  
    if milk > 1.3:  
        print("take milk")  
    else:  
        print("Don't have enough milk !")  
  
    if sugar > 0.2:  
        print("take sugar")  
    else:  
        print("Don't have enough sugar!")  
  
    if flour > 1.0:  
        print("take flour")  
    else:  
        print("Don't have enough flour !")  
  
    print("Mix")  
    print("Heat")  
    print("I made the pie!")
```

(continues on next page)

(continued from previous page)

```

make_problematic_pie(5,1,0.3)  # not enough flour ...

print("Party")

take milk
take sugar
Don't have enough flour !
Mix
Heat
I made the pie!
Party

```

QUESTION: this above version has a serious problem. Can you spot it ??

ANSWER: the program above is partying even when we do not have enough ingredients !

4.6.4 Check with the return

EXERCISE: We could correct the problems of the above pie by adding `return` commands. Implement the following function.

WARNING: DO NOT move the `print("Party")` **inside the function**

The exercise goal is keeping it outside, so to use the value returned by `make_pie` for deciding whether to party or not.

If you have any doubts on functions with return values, check [Chapter 6 of Think Python](#)¹¹²

```

[3]: def make_pie(milk, sugar, flour):
      """ - suppose we need 1.3 kg for milk, 0.2kg for sugar and 1.0kg for flour

          - takes as parameters the quantities we have in the sideboard
          IMPROVE WITH return COMMAND: RETURN True if the pie is doable,
                                          False otherwise

          *OUTSIDE* USE THE VALUE RETURNED TO PARTY OR NOT

          """
      # implement here the function
      #jupman-strip
      if milk > 1.3:
          print("take milk")
          # return True # NO, it would finish right here
      else:
          print("Don't have enough milk !")
          return False

      if sugar > 0.2:
          print("take sugar")
      else:
          print("Don't have enough sugar !")
          return False

```

(continues on next page)

¹¹² <http://greenteapress.com/thinkpython2/html/thinkpython2007.html>

(continued from previous page)

```

if flour > 1.0:
    print("take flour")
else:
    print("Don't have enough flour !")
    return False

print("Mix")
print("Heat")
print("I made the pie !")
return True
#jupman-strip

# now write here the function call, make_pie(5,1,0.3)
# using the result to declare whether it is possible or not to party :-(

#jupman-strip
made_pie = make_pie(5,1,0.3)

if made_pie == True:
    print("Party")
else:
    print("No party !")
#jupman-strip

take milk
take sugar
Don't have enough flour !
No party !

```

4.6.5 Exceptions

Real Python - Python Exceptions: an Introduction¹¹³

Using `return` we improved the previous function, but remains a problem: the responsibility to understand whether or not the pie is properly made is given to the caller of the function, who has to take the returned value and decide upon that whether to party or not. A careless programmer might forget to do the check and party even with an ill-formed pie.

So we ask ourselves: is it possible to stop the execution not just of the function, but of the whole program when we find an unforeseen situation?

To improve on our previous attempt, we can use the *exceptions*. To tell Python to **interrupt** the program execution in a given point, we can insert the instruction `raise` like this:

```
raise Exception()
```

If we want, we can also write a message to help programmers (who could be ourselves ...) to understand the problem origin. In our case it could be a message like this:

```
raise Exception("Don't have enough flour !")
```

Note: in professional programs, the exception messages are intended for programmers, verbose, and typically end up hidden in system logs. To final users you should only show short messages which are understandable by a non-technical public. At most, you can add an error code which the user might give to the technician for diagnosing the problem.

¹¹³ <https://realpython.com/python-exceptions/>

EXERCISE: Try to rewrite the function above by substituting the rows containing `return` with `raise Exception()`:

```
[4]: def make_exceptional_pie(milk, sugar, flour):
    """ - suppose we need 1.3 kg for milk, 0.2kg for sugar and 1.0kg for flour

    - takes as parameters the quantities we have in the sideboard

    - if there are missing ingredients, raises Exception

    """
    # implement function
    #jupman-strip

    if milk > 1.3:
        print("take milk")
    else:
        raise Exception("Don't have enough milk !")
    if sugar > 0.2:
        print("take sugar")
    else:
        raise Exception("Don't have enough sugar!")
    if flour > 1.0:
        print("take flour")
    else:
        raise Exception("Don't have enough flour!")
    print("Mix")
    print("Heat")
    print("I made the pie !")
    #/jupman-strip
```

Once implemented, by writing

```
make_exceptional_pie(5,1,0.3)
print("Party")
```

you should see the following (note how “Party” is *not* printed):

```
take milk
take sugar

-----
Exception                                 Traceback (most recent call last)
<ipython-input-10-02c123f44f31> in <module>()
----> 1 make_exceptional_pie(5,1,0.3)
      2
      3 print("Party")

<ipython-input-9-030239f08ca5> in make_exceptional_pie(milk, sugar, flour)
     18     print("take flour")
     19     else:
--> 20         raise Exception("Don't have enough flour !")
     21     print("Mix")
     22     print("Heat")

Exception: Don't have enough flour !
```

We see the program got interrupted before arriving to mix step (inside the function), and it didn't even arrived to party

(which is outside the function). Let's try now to call the function with enough ingredients in the sidebar:

```
[5]: make_exceptional_pie(5,1,20)
      print("Party")
```

```
take milk
take sugar
take flour
Mix
Heat
I made the pie !
Party
```

Manage exceptions

Instead of brutally interrupting the program when problems are spotted, we might want to try some alternative (like go buying some ice cream). We could use some `try except` blocks like this:

```
[6]: try:
      make_exceptional_pie(5,1,0.3)
      print("Party")
    except:
      print("Can't make the pie, what about going out for an ice cream?")
```

```
take milk
take sugar
Can't make the pie, what about going out for an ice cream?
```

If you note, the execution jumped the `print("Party")` but no exception has been printed, and the execution passed to the row right after the `except`

Particular exceptions

Until now we used a generic `Exception`, but, if you will, you can use more specific exceptions to better signal the nature of the error. For example, when you implement a function, since checking the input values for correctness is very frequent, Python gives you an exception called `ValueError`. If you use it instead of `Exception`, you allow the function caller to intercept only that particular error type.

If the function raises an error which is not intercepted in the catch, the program will halt.

```
[7]: def make_exceptional_pie_2(milk, sugar, flour):
      """ - suppose we need 1.3 kg for milk, 0.2kg for sugar and 1.0kg for flour

          - takes as parameters the quantities we have in the sidebar

          - if there are missing ingredients, raises Exception
      """

      if milk > 1.3:
          print("take milk")
      else:
          raise ValueError("Don't have enough milk !")
      if sugar > 0.2:
          print("take sugar")
      else:
```

(continues on next page)

(continued from previous page)

```

        raise ValueError("Don't have enough sugar!")
    if flour > 1.0:
        print("take flour")
    else:
        raise ValueError("Don't have enough flour!")
    print("Mix")
    print("Heat")
    print("I made the pie !")

try:
    make_exceptional_pie_2(5,1,0.3)
    print("Party")
except ValueError:
    print()
    print("There must be a problem with the ingredients!")
    print("Let's try asking neighbors !")
    print("We're lucky, they gave us some flour, let's try again!")
    print("")
    make_exceptional_pie_2(5,1,4)
    print("Party")
except: # manages all exceptions
    print("Guys, something bad happened, don't know what to do. Better to go out and_
    ↪take an ice-cream !")

```

```

take milk
take sugar

```

```

There must be a problem with the ingredients!
Let's try asking neighbors !
We're lucky, they gave us some flour, let's try again!

```

```

take milk
take sugar
take flour
Mix
Heat
I made the pie !
Party

```

For more explanations about `try catch`, you can see [Real Python - Python Exceptions: an Introduction](https://realpython.com/python-exceptions/)¹¹⁴

4.6.6 assert

They asked you to develop a program to control a nuclear reactor. The reactor produces a lot of energy, but requires at least 20 meters of water to cool down, and your program needs to regulate the water level. Without enough water, you risk a meltdown. You do not feel exactly up to the job, and start sweating.

Nervously, you write the code. You check and recheck the code - everything looks fine.

On inauguration day, the reactor is turned on. Unexpectedly, the water level goes down to 5 meters, and an uncontrolled chain reaction occurs. Plutonium fireworks follow.

Could we have avoided all of this? We often believe everything is good but then for some reason we find variables with unexpected values. The wrong program described above might have been written like so:

¹¹⁴ <https://realpython.com/python-exceptions/>

```
[8]: # we need water to cool our reactor

water_level = 40 # seems ok

print("water level: ", water_level)

# a lot of code

# a lot of code

# a lot of code

# a lot of code

water_level = 5 # forgot somewhere this bad row !

print("WARNING: water level low! ", water_level)

# a lot of code

# a lot of code

# a lot of code

# a lot of code

# after a lot of code we might not know if there are the proper conditions so that
↪ everything works allright

print("turn on nuclear reactor")

water level: 40
WARNING: water level low! 5
turn on nuclear reactor
```

How could we improve it? Let's look at the `assert` command, which must be written by following it with a boolean condition.

`assert True` does absolutely nothing:

```
[9]: print("before")
      assert True
      print("after")

before
after
```

Instead, `assert False` completely blocks program execution, by launching an exception of type `AssertionError` (Note how "after" is not printed):

```
print("before")
assert False
print("after")
```

```
before
-----
AssertionError                                Traceback (most recent call last)
```

(continues on next page)

(continued from previous page)

```
<ipython-input-7-a871fdc9ebee> in <module>()
----> 1 assert False

AssertionError:
```

To improve the previous program, we might use `assert` like this:

```
# we need water to cool our reactor

water_level = 40    # seems ok

print("water level: ", water_level)

# a lot of code

# a lot of code

# a lot of code

# a lot of code

water_level = 5    # forgot somewhere this bad row !

print("WARNING: water level low! ", water_level)

# a lot of code

# a lot of code

# a lot of code

# a lot of code

# after a lot of code we might not know if there are the proper conditions so that
# everything works alright so before doing critical things, it is always a good idea
# to perform a check ! if asserts fail (that is, the boolean expression is False),
# the execution suddenly stops

assert water_level >= 20

print("turn on nuclear reactor")
```

```
water level:  40
WARNING: water level low!  5

-----
AssertionError                                Traceback (most recent call last)
<ipython-input-3-d553a90d4f64> in <module>
      31 # the execution suddenly stops
      32
----> 33 assert water_level >= 20
      34
      35 print("turn on nuclear reactor")

AssertionError:
```

When to use assert?

The case above is willingly exaggerated, but shows how a check more sometimes prevents disasters.

Asserts are a quick way to do checks, so much so that Python even allows to ignore them during execution to improve the performance (calling python with the `-O` parameter like in `python -O my_file.py`).

But if performance are not a problem (like in the reactor above), it's more convenient to rewrite the program using an `if` and explicitly raising an `Exception`:

```
# we need water to cool our reactor

water_level = 40    # seems ok

print("water level: ", water_level)

# a lot of code

# a lot of code

# a lot of code

# a lot of code

water_level = 5    # forgot somewhere this bad row !

print("WARNING: water level low! ", water_level)

# a lot of code

# a lot of code

# a lot of code

# a lot of code

# after a lot of code we might not know if there are the proper conditions so
# that everything works all right. So before doing critical things, it is always
# a good idea to perform a check !

if water_level < 20:
    raise Exception("Water level too low !")    # execution stops here

print("turn on nuclear reactor")
```

```
water level:  40
WARNING: water level low!  5

-----
Exception                                Traceback (most recent call last)
<ipython-input-30-4840536c3388> in <module>
    30
    31 if water_level < 20:
--> 32     raise Exception("Water level too low !")    # execution stops here
    33
    34 print("turn on nuclear reactor")
```

(continues on next page)

(continued from previous page)

```
Exception: Water level too low !
```

Note how the reactor was *not* turned on.

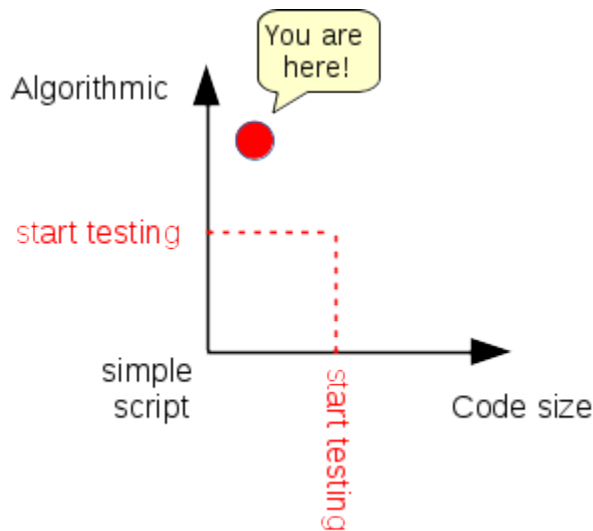
4.6.7 Testing

- If it seems to work, then it actually works? *Probably not.*
- The devil is in the details, especially for complex algorithms.
- We will do a crash course on testing in Python

WARNING: Bad software can cause losses of million \$/€ or even kill people. Suggested reading: [Software Horror Stories](#)¹¹⁵

Where Is Your Software?

As a data scientist, you might likely end up with code which is algorithmically complex, but maybe not too big in size. Either way, when red line is crossed you should start testing properly:



In a typical scenario, you are a junior programmer and your senior colleague ask you to write a function to perform some task, giving only an informal description:

```
[10]: def my_sum(x, y):
      """ RETURN the sum of x and y
      """
      raise Exception("TODO IMPLEMENT ME!")
```

Even better, your colleague might provide you with some automated tests you might run to check your function meets his/her expectations. If you are smart, you will even write tests for your own functions to make sure every little piece you add to your software is a solid block you can build upon.

¹¹⁵ <https://www.cs.tau.ac.il/~nachumd/horror.html>

4.6.8 Testing with asserts

NOTE: in this book we test with `assert`, but there are much better frameworks for testing!

If you get serious about software development, please consider using something like [PyTest](https://docs.pytest.org/en/stable/)¹¹⁶ (recent and clean) or [Unittest](https://docs.python.org/3/library/unittest.html)¹¹⁷ (Python default testing suite, has more traditional approach)

We can use `assert` to quickly test functions, and verify they behave like they should.

For example, from this function:

```
[11]: def my_sum(x, y):  
      s = x + y  
      return s
```

We expect that `my_sum(2, 3)` gives 5. We can write in Python this expectation by using an `assert`:

```
[12]: assert my_sum(2, 3) == 5
```

Se `my_sum` is correctly implemented:

1. `my_sum(2, 3)` will give 5
2. the boolean expression `my_sum(2, 3) == 5` will give `True`
3. `assert True` will be executed without producing any result, and the program execution will continue.

Otherwise, if `my_sum` is NOT correctly implemented like in this case:

```
def my_sum(x, y):  
    return 666
```

1. `my_sum(2, 3)` will produce the number 666
2. the boolean expression `my_sum(2, 3) == 5` will give `False`
3. `assert False` will interrupt the program execution, raising an exception of type `AssertionError`

Exercise structure

Exercises are often structured in the following format:

```
def my_sum(x, y):  
    """ RETURN the sum of numbers x and y  
    """  
    raise Exception("TODO IMPLEMENT ME!")  
  
assert my_sum(2, 3) == 5  
assert my_sum(3, 1) == 4  
assert my_sum(-2, 5) == 3
```

If you attempt to execute the cell, you will see this error:

¹¹⁶ <https://docs.pytest.org/en/stable/>

¹¹⁷ <https://docs.python.org/3/library/unittest.html>

```

-----
Exception                                Traceback (most recent call last)
<ipython-input-16-5f5c8512d42a> in <module>()
      6
      7
----> 8 assert my_sum(2,3) == 5
      9 assert my_sum(3,1) == 4
     10 assert my_sum(-2,5) == 3

<ipython-input-16-5f5c8512d42a> in somma(x, y)
      3     """ RETURN the sum of numbers x and y
      4     """
----> 5     raise Exception("TODO IMPLEMENT ME!")
      6
      7

Exception: TODO IMPLEMENT ME!

```

To fix them, you will need to:

1. substitute the row `raise Exception("IMPLEMENTAMI")` with the body of the function
2. execute the cell

If cell execution doesn't result in raised exceptions, perfect ! It means your function does what it is expected to do (the `assert` which succeed do not produce any output)

Otherwise, if you see some `AssertionError`, probably you did something wrong.

NOTE: The `raise Exception("TODO IMPLEMENT ME")` is put there to remind you that the function has a big problem, that is, it doesn't have any code !!! In long programs, it might happen you know you need a function, but in that moment you don't know what code put in the function body. So, instead of putting in the body commands that do nothing like `print()` or `pass` or `return None`, it is WAY BETTER to raise exceptions so that if by chance the program reaches the function, the execution is suddenly stopped and the user is signalled with the nature and position of the problem. Many editors for programmers, when automatically generating code, put inside function skeletons to implement some `Exception` like this.

Let's try to willingly write a wrong function body, which always return 5, independently from `x` and `y` given in input:

```

def my_sum(x,y):
    """ RETURN the sum of numbers x and y
    """
    return 5

assert my_sum(2,3) == 5
assert my_sum(3,1) == 4
assert my_sum(-2,5) == 3

```

In this case the first assertion succeeds and so the execution simply passes to the next row, which contains another `assert`. We expect that `my_sum(3,1)` gives 4, but our ill-written function returns 5 so this `assert` fails. Note how the execution is interrupted at the *second* `assert`:

```

-----
AssertionError                            Traceback (most recent call last)
<ipython-input-19-e5091c194d3c> in <module>()
      6
      7 assert my_sum(2,3) == 5
----> 8 assert my_sum(3,1) == 4
      9 assert my_sum(-2,5) == 3

```

(continues on next page)

(continued from previous page)

AssertionError:

If we implement well the function and execute the cell we will see no output: this means the function successfully passed the tests and we can conclude that it is *correct with reference to the tests*:

ATTENTION: always remember that these kind of tests are *never* exhaustive ! If tests pass it is only an indication the function *might* be correct, but it is never a certainty !

```
[13]: def my_sum(x, y):
      """ RITORNA the sum of numbers x and y
      """
      return x + y

      assert my_sum(2, 3) == 5
      assert my_sum(3, 1) == 4
      assert my_sum(-2, 5) == 3
```

EXERCISE: Try to write the body of the function multiply:

- substitute `raise Exception("TODO IMPLEMENT ME")` with `return x * y` and execute the cell. If you have written correctly, nothing should happen. In this case, congratulations! The code you have written is *correct with reference to the tests* !
- Try to substitute instead with `return 10` and see what happens.

```
[14]: def my_mul(x, y):
      """ RETURN the multiplication of numbers x and y
      """
      #jupman-raise
      return x * y
      #/jupman-raise

      assert my_mul(2, 5) == 10
      assert my_mul(0, 2) == 0
      assert my_mul(3, 2) == 6
```

even_numbers example

Let's see a slightly more complex function:

```
[15]: def even_numbers(n):
      """
      Return a list of the first n even numbers

      Zero is considered to be the first even number.

      >>> even_numbers(5)
```

(continues on next page)

(continued from previous page)

```
[0, 2, 4, 6, 8]
"""
raise Exception("TODO IMPLEMENT ME!")
```

In this case, if you run the function as it is, you are reminded to implement it:

```
>>> even_numbers(5)
```

```
-----
Exception                                Traceback (most recent call last)
<ipython-input-2-d2cbc915c576> in <module>()
----> 1 even_numbers(5)

<ipython-input-1-a20a4ea4b42a> in even_numbers(n)
      8     [0, 2, 4, 6, 8]
      9     """
----> 10     raise Exception("TODO IMPLEMENT ME!")

Exception: TODO IMPLEMENT ME!
```

Why? The instruction

```
raise Exception("TODO IMPLEMENT ME!")
```

tells Python to immediately stop execution, and signal an error to the caller of the function `even_number`. If there were commands right after `raise Exception("TODO IMPLEMENT ME")`, they would not be executed. Here, we are directly calling the function from the prompt, and we didn't tell Python how to handle the `Exception`, so Python just stopped and showed the error message given as parameter to the `Exception`

Spend time reading well the function text!

Always read very well function text and ask yourself questions! What is the supposed input? What should be the output? Is there any output to return at all, or should you instead modify *in-place* a passed parameter (i.e. for example, when you sort a list)? Are there any edge cases, es what happens for $n=0$? What about $n < 0$?

Let's code a possible solution. As it often happens, first version may be buggy, in this case for example purposes we intentionally introduce a bug:

```
[16]: def even_numbers(n):
      """
      Return a list of the first n even numbers

      Zero is considered to be the first even number.

      >>> even_numbers(5)
      [0, 2, 4, 6, 8]
      """
      r = [2 * x for x in range(n)]
      r[n // 2] = 3    # <-- evil bug, puts number '3' in the middle, and 3 is not even .
      ↪
      return r
```

Typically the first test we do is printing the output and do some 'visual inspection' of the result, in this case we find many numbers are correct but we might miss errors such as the wrong 3 in the middle:

```
[17]: print(even_numbers(5))  
[0, 2, 3, 6, 8]
```

Furthermore, if we enter commands at the prompt, each time we fix something in the code, we need to enter commands again to check everything is ok. This is inefficient, boring, and prone to errors.

Let's add assertions

To go beyond the dumb “visual inspection” testing, it's better to write some extra code to allow Python checking for us if the function actually returns what we expect, and throws an error otherwise. We can do so with `assert` command, which verifies if its argument is `True`. If it is not, it raises an `AssertionError` immediately stopping execution.

Here we check the result of `even_numbers(5)` is actually the list of even numbers `[0, 2, 4, 6, 8]` we expect:

```
assert even_numbers(5) == [0, 2, 4, 6, 8]
```

Since our code is faulty, `even_numbers` returns the wrong list `[0, 2, 3, 6, 8]` which is different from `[0, 2, 4, 6, 8]` so assertion fails showing `AssertionError`:

```
-----  
AssertionError                                Traceback (most recent call last)  
<ipython-input-21-d4198f229404> in <module>()  
----> 1 assert even_numbers(5) != [0, 2, 4, 6, 8]  
  
AssertionError:
```

We got some output, but we would like to have it more informative. To do so, we may add a message, separated by a comma:

```
assert even_numbers(5) == [0, 2, 4, 6, 8], "even_numbers is not working !!"
```

```
-----  
AssertionError                                Traceback (most recent call last)  
<ipython-input-18-8544fcd1b7c8> in <module>()  
----> 1 assert even_numbers(5) == [0, 2, 4, 6, 8], "even_numbers is not working !!"  
  
AssertionError: even_numbers is not working !!
```

So if we modify code to fix bugs we can just launch the `assert` commands and have a quick feedback about possible errors.

Error kinds

As a fact of life, errors happen. Sometimes, your program may have inconsistent data, like wrong parameter type passed to a function (i.e. string instead of integer). A good principle to follow in these cases is to try have the program detect weird situations, and stop as early as such a situation is found (i.e. in the Therac 25 case, if you detect excessive radiation, showing a warning sign is not enough, it's better to stop). Note stopping might not always be the desirable solution (if one pigeon enters one airplane engine, you don't want to stop all the other engines). If you want to check function parameters are correct, you do the so called *precondition checking*.

There are roughly two cases for errors, external user misusing your program, and just plain wrong code. Let's analyze both:

Error kind a) An external user misuses your program.

You can assume whoever uses your software, final users or other programmers, they will try their very best to wreck your precious code by passing all sort of non-sense to functions. Everything can come in, strings instead of numbers, empty arrays, None objects ... In this case you should signal the user he made some mistake. The most crude signal you can have is raising an Exception with `raise Exception("Some error occurred")`, which will stop the program and print the stacktrace in the console. Maybe final users won't understand a stacktrace, but at least programmers hopefully will get a clue about what is happening.

In these case you can raise an appropriate Exception, like `TypeError`¹¹⁸ for wrong types and `ValueError`¹¹⁹ for more generic errors. Other basic exceptions can be found in [Python documentation](#)¹²⁰. Notice you can also define your own, if needed (we won't consider custom exceptions in this course).

NOTE: Many times, you can consider yourself the 'careless external user' to guard against.

Let's enrich the function with some appropriate type checking:

Note that for checking input types, you can use the function `type()`:

```
[18]: type(3)
[18]: int

[19]: type("ciao")
[19]: str
```

Let's add the code for checking the *even_numbers* example:

```
[20]: def even_numbers(n):
      """
      Return a list of the first n even numbers

      Zero is considered to be the first even number.

      >>> even_numbers(5)
      [0, 2, 4, 6, 8]
      """
      if type(n) is not int:
          raise TypeError("Passed a non integer number: " + str(n))

      if n < 0:
          raise ValueError("Passed a negative number: " + str(n))

      r = [2 * x for x in range(n)]
      return r
```

Let's pass a wrong type and see what happens:

```
>>> even_numbers("ciao")

-----
TypeError                                Traceback (most recent call last)
<ipython-input-14-a908b20f00c4> in <module>()
```

(continues on next page)

¹¹⁸ <https://docs.python.org/3/library/exceptions.html#TypeError>

¹¹⁹ <https://docs.python.org/3/library/exceptions.html#ValueError>

¹²⁰ <https://docs.python.org/3/library/exceptions.html#built-in-exceptions>

(continued from previous page)

```

----> 1 even_numbers("ciao")

<ipython-input-13-b0b3a85f2b2a> in even_numbers(n)
      9     """
     10     if type(n) is not int:
----> 11         raise TypeError("Passed a non integer number: " + str(n))
     12
     13     if n < 0:

TypeError: Passed a non integer number: ciao

```

Now let's try to pass a negative number - it should suddenly stop with a meaningful message:

```

>>> even_numbers(-5)

-----
ValueError                                Traceback (most recent call last)
<ipython-input-15-3f648fdf6de7> in <module>()
----> 1 even_numbers(-5)

<ipython-input-13-b0b3a85f2b2a> in even_numbers(n)
     12
     13     if n < 0:
----> 14         raise ValueError("Passed a negative number: " + str(n))
     15
     16     r = [2 * x for x in range(n)]

ValueError: Passed a negative number: -5

```

Now, even if you ship your code to careless users, and as soon as they commit a mistake, they will get properly notified.

Error kind b): Your code is just plain wrong

In this case, it's 100% your fault, and these sort of bugs should never pop up in production. For example your code passes internally wrong stuff, like strings instead of integers, or wrong ranges (typically integer outside array bounds). So if you have an internal function nobody else should directly call, and you suspect it is being passed wrong parameters or at some point it has inconsistent data, to quickly spot the error you could add an assertion:

```

[21]: def even_numbers(n):
      """
      Return a list of the first n even numbers

      Zero is considered to be the first even number.

      >>> even_numbers(5)
      [0, 2, 4, 6, 8]
      """
      assert type(n) is int, "type of n is not correct: " + str(type(n))
      assert n >= 0, "Found negative n: " + str(n)

      r = [2 * x for x in range(n)]

      return r

```

As before, the function will stop as soon we call it we wrong parameters. The big difference is, this time we are assuming `even_numbers` is just for personal use and nobody else except us should directly call it.

Since assertion consume CPU time, IF we care about performances AND once we are confident our program behaves correctly, we can even remove them from compiled code by using the `-O` compiler flag. For more info, see [Python wiki](#)¹²¹

EXERCISE: try to call latest definition of `even_numbers` with wrong parameters, and see what happens.

NOTE: here we are using the correct definition of `even_numbers`, not the buggy one with the 3 in the middle of returned list !

```
[ ]:
```

4.7 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* !

4.7.1 I COMANDAMENT

You shall write Python code

Who does not writes Python code, does not learn Python

4.7.2 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 !!

0
1
2
2
```

¹²¹ <https://wiki.python.org/moin/UsingAssertionsEffectively>

```
[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
```

4.7.3 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
```

4.7.4 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 variable.

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).

4.7.5 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`

4.7.6 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.

4.7.7 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 of the program. Tables, nodes, arrows, all can help figuring out a solution for the problem.

4.7.8 VIII COMANDAMENT

You shall never ever reassign `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
```

4.7.9 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](#)¹²²

4.7.10 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 everything, 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](#)¹²³ or make first a copy of the original sequence.

¹²² <https://en.softpython.org/errors-and-testing/errors-and-testing-sol.ipynb>

¹²³ <https://en.softpython.org/control-flow/flow3-while-sol.html>

CHAPTER
FIVE

**CHAPTER
SIX**

INDEX