
python-tools-for-students Documentation

Release 1be72ec

Sebastian Weigand

2020-08-22

Learning:

1	python-tools-for-students	2
1.1	Why python?	2
1.2	Where are we with this project right now?	3
1.3	What do you need?	4
1.4	Who are we?	4
1.5	How can YOU help?	4
1.6	Why in english?	4
1.7	Contributors	5
2	Getting started	5
2.1	Run it locally	5
2.2	Run it in the cloud	7
3	Chapters	7
3.1	Using JuPyteR-Lab	7
4	Examples	16
5	Cheat Sheets	16
5.1	Python	16
5.2	Numpy	16
5.3	Pandas	16
5.4	Datascience	17
6	Tutorials	17
6.1	Python	17
7	Credits	17
7.1	Development Lead	17
7.2	Contributors	17
8	Contributing	17
8.1	Types of Contributions	17
8.2	Structure of the project	18
8.3	Getting Started!	19
8.4	Testing	20
8.5	Add your changes to the docs	20
8.6	Style guide	21

8.7 Pull Request Guidelines	21
9 Indices and tables	22

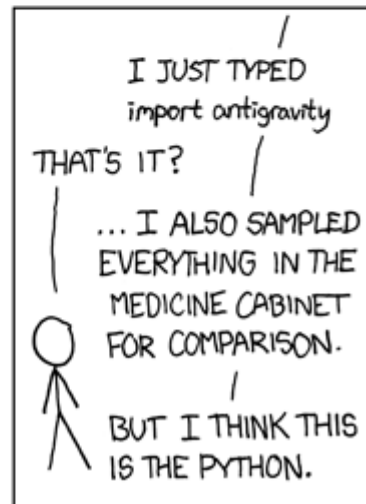
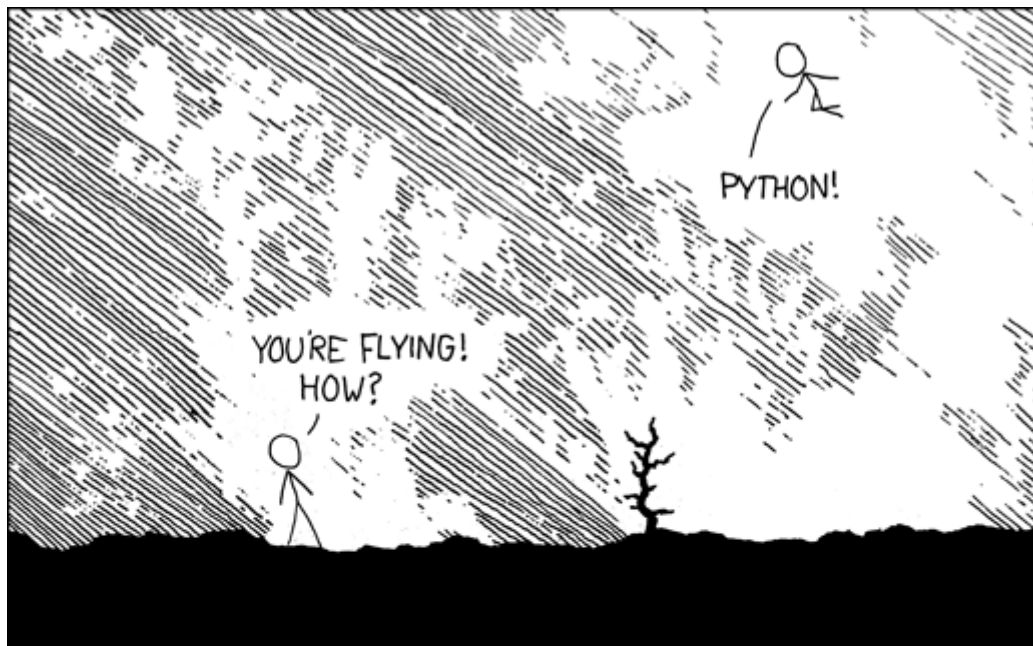
1 python-tools-for-students

This projects aims to teach students the basics of how to use **powerful Python Tools** for the day to day life at university:

- Solving equations
 - Analytically
 - Numerically
- Handling data
 - Reading differently structured plaintext files
 - Doing calculations on data
 - Fitting of data
 - Visualization (plotting)

1.1 Why python?

Python is one the **fastest growing programming languages**, it is easy to learn (i.e. compared to C), is multi purpose (there is pretty much no task that can't be done in python) and it has a rich eco system of libraries for math, physics, engineering, data science and data evaluation in general.



1.2 Where are we with this project right now?

At the moment we are writing teaching material and gather ideas what to teach and build a useful structure to teach the materials. Which is why **this is your chance** to tell us what you want to learn. Just open an issue, write about what you want to learn (examples help with explanations) and we can discuss if this should be part of this course or is too much of an edge case. And of course we are also always happy if you want to contribute and teach others the skills you acquired.

1.3 What do you need?

- **A base knowledge in Python**

We won't teach you python from the ground up, so you need to already have some basic knowledge about python. For this purpose we recommend that you complete i.e. the [SoloLearn Python 3 Tutorial](#), which is also available as a free mobile app and/or watch the tutorials made by [sentdex](#)

- **A working python installation**

The best way to get Python up and running for your OS is to download [Anaconda](#), which is crossplatform and comes with most tools needed prebundled (batteries included).

- **A github account**

The github account is needed so you can write issues and tell us what **YOU** want to learn, what we should improve or if we did something wrong.

1.4 Who are we?

We are a group of master students from TU-Berlin (Germany), who want to share their experience with python tools, which make the day to day life at university (homework, writing reports and evaluating data) easier.

1.5 How can YOU help?

Since all contributors are working on this in their spare time, we could use your help and encourage you to contribute to this project as well. This will also give you inside on how to work on a project on github.

What we need:

- **Correction reader:** Since we all are only humans, it might happen that there are spelling errors, so if you see them feel free to correct them.
- **Creators:** Since we are lacking manpower, we would highly appreciate if you would share your knowledge as well and help us giving people the knowledge they need.
- **Summary writer:** Since writing the material itself is very time consuming, we would love for the community to write a summary in the style of a **TL;DR**, for the material we provide.
- **Example writer:** If you have an example/use case on how to use the knowledge gained here, feel free to share it with everyone and we will review if it fits in the context of this project.

1.6 Why in english?

Even so we are a group of german students, the community of english speaking programmers is much bigger than the one of programmers which only speak german. And since most problems you might have, other people already had and solved them, it is of benefit to know the terms we introduce in english, since this gives you access to a bigger knowledge base (google/duckduckgo and stackoverflow are your friends).

1.7 Contributors

Thanks goes to these wonderful people ([emoji key](#)):

This project follows the [all-contributors](#) specification. Contributions of any kind welcome!

2 Getting started

2.1 Run it locally

This is the ideal way, since you will have everything you need in your day to day student life installed on your computer and all your data will persist.

Install python

Anaconda

Anaconda is our recommendation to install python on all operating systems, since it comes with most needed packages prebundled (batteries included). Another bonus is that `conda` isn't just a python package manager like `pip`, but a package manager for multiple resources (i.e. [node.js](#) or [latex](#)) and also an [environment](#) manager. Yet another bonus of `conda` is that it has a build pipeline in place, which allows to install all packages from binary (no compiling needed from your side, which at times can be pretty time consuming). Download the installer from the official website of [Anaconda](#), follow the instructions and you will be good to go.

Note: If you are using a Posix system (Linux/OsX) you don't want to mess with the system python, since many system tools rely on it, and in a worse case scenario you could break your Os.

Using Anaconda only if needed

Depending on other software you run on your computer, which depends on the installed system python version (i.e. QtiPlot), you may not want to use Anaconda as your default python or add it to the `PATH` variable, since this might cause conflicts and/or break that software.

Posix like Shells:

Users of a Posix like terminal (i.e. `bash`), can simply add the following function to their shell configuration file (i.e. `~/.bashrc`/`~/.bash_profile`)

Linux/MacOs:

```
ANACONDA_BIN_DIR=<anaconda-install-folder>/bin
use_conda(){
    export PATH="$ANACONDA_BIN_DIR:$PATH"
}
```

Windows (git-bash, MinGw):

```
ANACONDA_BIN_DIR=<anaconda-install-folder>/bin
use_conda(){
    CONDA_PATHS="$ANACONDA_INSTALL_DIR"
    CONDA_PATHS="$ANACONDA_INSTALL_DIR\Library\mingw-w64\bin;$CONDA_PATHS"
    CONDA_PATHS="$ANACONDA_INSTALL_DIR\Library\usr\bin;$CONDA_PATHS"
```

(continues on next page)

(continued from previous page)

```
CONDA_PATHS="$ANACONDA_INSTALL_DIR\Library\bin;$CONDA_PATHS"
CONDA_PATHS="$ANACONDA_INSTALL_DIR\Scripts;$CONDA_PATHS"
export PATH="$CONDA_PATHS:$PATH"
}
```

CMD on Windows:

If you are working on Windows and for some reason want to use CMD as your terminal, you can create a batch script `use_conda.bat` in a folder which is part of the `PATH` variable (i.e. `C:\Windows`, this needs Admin rights).

```
@echo off
SET ANACONDA_INSTALL_DIR=<anaconda-install-folder>
SET CONDA_PATHS=%ANACONDA_INSTALL_DIR%
SET CONDA_PATHS=%ANACONDA_INSTALL_DIR%\Library\mingw-w64\bin;%CONDA_PATHS%
SET CONDA_PATHS=%ANACONDA_INSTALL_DIR%\Library\usr\bin;%CONDA_PATHS%
SET CONDA_PATHS=%ANACONDA_INSTALL_DIR%\Library\bin;%CONDA_PATHS%
SET CONDA_PATHS=%ANACONDA_INSTALL_DIR%\Scripts;%CONDA_PATHS%
SET PATH=%CONDA_PATHS%;%PATH%
```

This will temporarily add the conda paths to the open terminal and allows you to simply call `use_conda/use_conda.bat` (which in both cases autocompletes), when you want to use conda. When you open a new terminal, it won't know about conda and work as it normally does.

Note: For this to work you need to replace `<anaconda-install-folder>`, with the actual path you installed Anaconda to.

Note: If you use the other software more sparsely than the conda python, you could of course, just turn this approach around and prepend the path to the system python to the `PATH` variable, when you don't want to use conda.

Pure CPython

If you don't want to install conda, this [Python installation guide](#) can guide you through the process of getting the pure CPython Interpreter.

Get the project

The sources for `python-tools-for-students` can be downloaded from the [Github repo](#).

You can either clone the public repository:

```
$ git clone git://github.com/students-teach-students/python-tools-for-students
```

Note: This should be the preferred way since you can easily update the files by running `$ git pull` and won't clutter your download folder with incremental tarballs.

Or download the [tarball](#):

```
$ curl -OJL https://github.com/students-teach-students/python-tools-for-students/tarball/
↪ master
```

Once you have a copy of the source, you need to install the dependencies install it with:

```
$ pip install -r requirements.txt
```

Start jupyter lab

Once you have everything up and running you just need to open a terminal in the project folder (or its material subfolder) and run the following command:

```
$ jupyter lab
```

After that jupyter lab will open a new tab in you default browser and you can start exploring.

Note: For Windows users we recommend to use [Git bash](#) to start jupyter lab, since CMD and Powershell might not support all system calls we showcase.

Trouble shooting

If a new notebook isn't working, it might be that added new requirements, just try installing them by running this command in your terminal:

```
$ pip install -r requirements.txt
```

2.2 Run it in the cloud

If you don't want to install python and just play around a bit with the notebooks, you can always just run in our online demo at mybinder.org.

Warning: The binder session expires after 10 minutes inactivity and you will loose all your progress if you didn't download the files you edited/created.

3 Chapters

3.1 Using JuPyteR-Lab

What is jupyter-lab?

jupyter-lab is a browser based editor, with its main focus on editing jupyter-notebooks (which we will also focus on). It is part of the jupyter project, which is an extension of the iPython project. The iPython project and especially the iPython-Notebooks, were developed as an effort to have a free and open source alternative to Mathematica, with a similar interface. This is why it also uses an Input-Output-Cell-Structure and allows you to have your displayed data (i.e. tables, plots or formulas) right next to the code that produced them.

The name JuPyteR is derived from the programming languages which its development initially was focused on Julia, Python and R. But nowadays it can run a lot more programming languages (i.e. bash, C++ or Fortran), as long as you have the needed interpreter or Compiler installed. For a list of available kernels and how to install them, have a look [here](#).

Why jupyter-notebooks?

jupyter-notebooks are perfect for data exploration and a fast interactive working with code. That you can directly see the output of your code, is helpful if you are new to some specific library or programming in general, because you can see the effect of your changes directly after you applied them. Also since this class is for students and aimed at their day to day university needs (i.e. homework), it is also worth mentioning that jupyter-notebooks can be easily exported to PDF (File->Export Notebook as...->Export Notebook to PDF).

Note:

If you want develop a library, GUI or similar, you should choose a proper IDE to do so.

How to start jupyter?

To start jupyter-lab simply open a shell/terminal/command line in the folder you want as the base directory for your project and run:

```
jupyter lab
```

This will start the jupyter lab server locally and open its url in your default browser.

Note:

You can browse all the subdirectories of that folder in the File Browser but can't see higher level folders.

Different input modes

Jupyter-lab has two different modes to work with cells the command mode and the insert mode, are used for different interactions with cells.

Command mode

The command mode is used to control the behavior of a whole cell or multiple cells, such as creating, deleting or converting the type of cells. It can also be used to faster scroll your notebook, since the navigation with the arrow keys will be on a cell level. To activate the command mode press Esc. Being in command mode is recognizable by the grayed out interior of the cell, the missing blue border around it and the missing cursor.



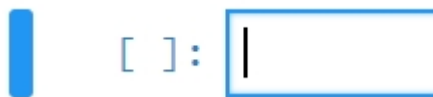
Most commonly used shortcuts

- DD - Deletes currently marked cells (use with caution, this can't be undone by Ctrl+Z)
- A - Creates a cell Above the current cell
- B - Creates a cell Below the current cell
- M - Convert current cell to a Markdown cell
- Y - Convert current cell to a Code cell
- R - Convert current cell to a Raw cell
- Shift+Arrow Keys - Mark multiple cells

Insert mode

As command mode allows you to interact with cells as a whole, insert mode allows you to change its content.

To activate the insert mode press Enter. Being in insert mode is recognizable by the white interior of the cell, the blue border around it and the blinking cursor.



Most commonly used shortcuts

- Shift+Enter - Runs/Evaluates the cell and selects the cell below (also works in command mode)
- Ctrl+Enter - Runs/Evaluates cell inplace (also works in command mode)
- Tab - Autocomplete

Different input cell types

By default cell can have one of three types code cell, markdown cell or raw cell, each serve a different purpose.

Code cells (Ctrl+Y)

The purpose of code cells should be quite self explanatory, these cells are used to write the code you want to execute. Which kind of code you want to execute depends on what you told the cell its content is (see Jupyter-magic), the default being Python or which ever kernel you started the notebook with.

[]:

Markdown cells

Markdown cells are used to document your code in a structured and RichText fashion. It allows you to insert headings, lists, tables, images and even videos (for more information). The special extended kind of Markdown used by jupyter notebooks also allows you to write formulas in LaTeX which are then rendered accordingly.

The expression: $\int_{-\infty}^{\infty} f(x) dx$,

Evaluates to: $\int_{-\infty}^{\infty} f(x) dx$

Note:

For the Markdown to be rendered the cell needs to be executed. If a Markdown cell is empty and was executed it will show the boilerplate text:

'Type Markdown and LaTeX: α^2 '

Raw cells

The purpose of raw cells is to be ignore in the normal notebook and if you convert the notebook i.e. to a LaTeX document, it keeps its content intact and allows you to insert code in your converted document, without having manually temper with the file after conversion over and over again.

Note:

There also are slide cells for presentations, which we might cover in a later chapter.

Jupyter-magic

The so called jupyter-magic is a collection of special commands, which allow you to influence behavior jupyter-lab and/or to run with different kernels. Jupyter-magic is grouped in two kinds, the line-magic and cell magic. An other kind of magic is OSMagic (system calls), which depend on the terminal you opened jupyter-lab with and return the same result as calling that command from the terminal directly.

Note:

On Windows we recommended to run it in Git-Bash or a similar Posix based terminal, since those terminals are much more powerful than the default Windows ones (CMD, Powershell).

Line magic

Commands listed as line magic only apply to a single line and start with a %.

Most commonly used line magic

- `%lsmagic` - Lists the available magic commands.
- `%timeit` - This is used to gain information about the execution time of a single line of code, by running it multiple times and measuring the time it took to finish.
- `%ls` - Lists all files and directories (OS independent)
- `%pycat <filename>` - Displays the content of a file with Python syntax highlighting
- `%matplotlib inline` - This is normally used at the start of a notebook and includes figures generated with matplotlib, which are returned by your code, directly in the corresponding output cell.
- `%matplotlib notebook` - More or less the same as `%matplotlib inline` but the plots are more interactive.

```
[1]: %lsmagic
[1]: Available line magics:
%alias %alias_magic %autoawait %autocall %automagic %autosave %bookmark %cd %clear
→ %cls %colors %conda %config %connect_info %copy %ddir %debug %dhist %dirs
→ %doctest_mode %echo %ed %edit %env %gui %hist %history %killbgscripts %ldir
→ %less %load %load_ext %loadpy %logoff %logon %logstart %logstate %logstop %ls
→ %lsmagic %macro %magic %matplotlib %mkdir %more %notebook %page %pastebin %pdb
→ %pdef %pdoc %pfile %pinfo %pinfo2 %pip %popd %pprint %precision %prun %psearch
→ %psource %pushd %pwd %pycat %pylab %qtconsole %quickref %recall %rehashx
→ %reload_ext %ren %rep %rerun %reset %reset_selective %rmdir %run %save %sc
→ %set_env %store %sx %system %tb %time %timeit %unalias %unload_ext %who %who_
→ %ls %whos %xdel %xmode

Available cell magics:
%%! %%HTML %%SVG %%bash %%capture %%cmd %%debug %%file %%html %%javascript %%js
→ %%latex %%markdown %%perl %%prun %%pypy %%python %%python2 %%python3 %%ruby %
→ %script %%sh %%svg %%sx %%system %%time %%timeit %%writefile

Automagic is ON, % prefix IS NOT needed for line magics.
```

Cell magic

Commands listed as cell magic apply to a whole cell and start with `%%`.

Most commonly used cell magic

- `%%timeit` - This is used to gain information about the execution time of a cell of code, by running it multiple times and measuring the time it took to finish.
- `%%bash` - Makes the code written in a code cell be executed if it was a bash script (this also works for other languages, depending on which kernels you have installed).
- `%%writefile <filename>` - Writes the content of a cell, to a given file.

```
[2]: %%bash
ls -la

total 32
drwxrwxrwx 1 dafuq dafuq 4096 Jul 22 15:54 .
drwxrwxrwx 1 dafuq dafuq 4096 Jun 15 00:15 ..
```

(continues on next page)

(continued from previous page)

```
drwxrwxrwx 1 dafuq dafuq 4096 Feb  7 13:54 .ipynb_checkpoints
-rwxrwxrwx 1 dafuq dafuq 29504 Jul 22 15:55 1_Using-Jupyter-Lab.ipynb
drwxrwxrwx 1 dafuq dafuq 4096 Apr 14 16:19 images
```

System calls (OSMagics)

System calls start with a `!` and execute a command as if it was run in the terminal jupyter lab was started with.

Most commonly used system calls (posix like system)

- `!ls -la` - Lists all files and directories (also hidden ones), as well as their size and read, write, execution rights.
- `!head <filename>` - Displays the first lines of a file. This is especially helpful if you want to read text files, which contain a table like structure and you aren't sure which is the column separating character is.
- `!tail <filename>` - Displays the last lines of a file. This is useful if you want to see the progress of a log file (what were the last written entries).
- `!pwd` - Displays the path of the current working directory.

```
[3]: !ls -la
total 32
drwxr-xr-x 1 dafuq 197121      0 Jul 22 15:54 .
drwxr-xr-x 1 dafuq 197121      0 Jun 15 00:15 ..
drwxr-xr-x 1 dafuq 197121      0 Feb  7 13:54 .ipynb_checkpoints
-rw-r--r-- 1 dafuq 197121 29504 Jul 22 15:55 1_Using-Jupyter-Lab.ipynb
drwxr-xr-x 1 dafuq 197121      0 Apr 14 16:19 images
```

Getting familiar with shortcuts (Ctrl+Shift+C)

The easiest way to get familiar with the commands you can run and their keyboard shortcuts, is to use

the commands panel (Ctrl+Shift+C or click the Icon  on the left panel selection).

Using commands panel you can also execute commands, which don't have a shortcut like Restart Kernel and Run All Cells.... Since the search is a [fuzzy search](#) it also helps to find commands, which command name you don't know exactly or to simply look up which commands you can use.

Getting help information

In Python proper written functions/classes/methods contain information about what they are supposed to do and sometimes even usage examples. This is called the `docstring` and gives the first help in how to use something. In jupyter lab this `docstring` can be accessed in multiple ways.

Print help (?)

In jupyter lab the ? is a special symbol. If it's written before or after an expression, the docstring of this expression will be written to the output cell, when the cell is executed.

Note:

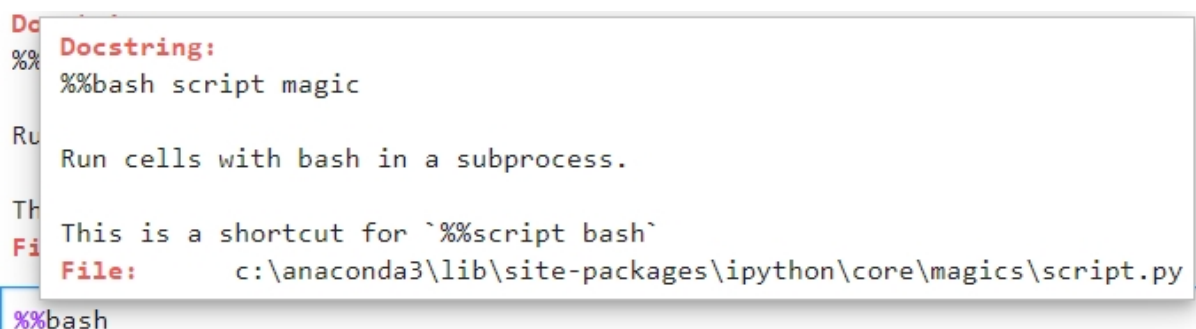
If a function/method is being called (has () at the end), this doesn't work and will give you an error.

```
[4]: %%bash?  
  
Docstring:  
%%bash script magic  
  
Run cells with bash in a subprocess.  
  
This is a shortcut for `%%script bash`  
File:      c:\anaconda3\lib\site-packages\ipython\core\magics\script.py
```

```
[5]: ?%%bash  
  
Docstring:  
%%bash script magic  
  
Run cells with bash in a subprocess.  
  
This is a shortcut for `%%script bash`  
File:      c:\anaconda3\lib\site-packages\ipython\core\magics\script.py
```

Quick help (Ctrl+Shift)

Another method to get a fast glimpse of the docstring is the quick help function, which is triggered by pressing Ctrl+Shift and will open a small popup window above your cursor. This is especially useful if you just want to look up the function-/method-signature, because you forgot which argument was at which place or you aren't sure what a keyword argument (kwarg) was called.



The screenshot shows a Jupyter Lab cell with the input `%%bash`. A quick help popup window is displayed above the cursor, showing the docstring for `%%bash`. The popup content is as follows:

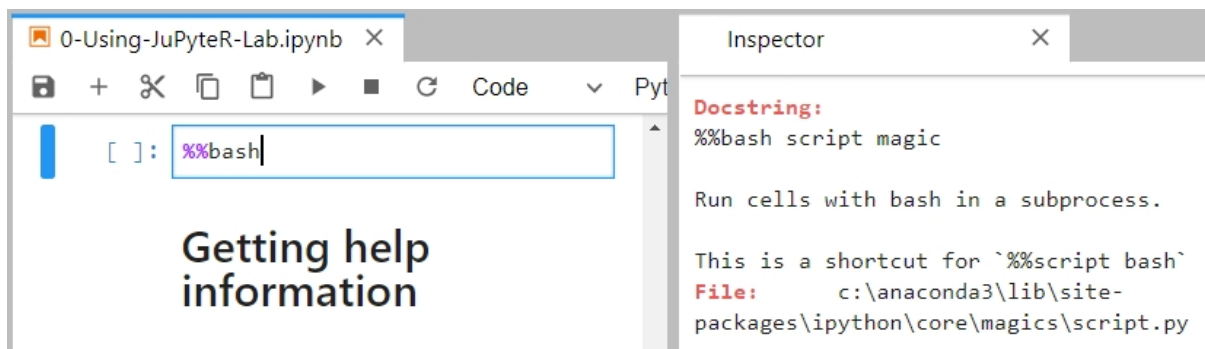
```
Docstring:  
%%bash script magic  
  
Run cells with bash in a subprocess.  
  
This is a shortcut for `%%script bash`  
File:      c:\anaconda3\lib\site-packages\ipython\core\magics\script.py
```

Inspector/Contextual Help window(Ctrl+I)

Last but not least, you can also open the Inspector/Contextual Help window by pressing `Ctrl+I`. The inspector shows the docstring of the code while you are typing it, in separate tab which you can i.e. arrange right next to your code. This is especially useful if you are learning to program or are starting to work with a new package/library and aren't familiar with all its functionality yet.

Note:

The Inspector was renamed to Contextual Help in jupyterlab 1.0 and also changed usage behavior. While the Inspector listened to keyboard input and changed its content while typing, with the Contextual Help you select elements by clicking at them and they stay the same while you type.



Cutomizing jupyter-lab

Jupyter lab has a lot of extensions (plugins), which can give you a better experience writing your code and change your editors capabilities how you need them. A list of great extensions can be found at [awesome-jupyterlab](https://awesome-jupyterlab.com/).

Our recommended extensions are:

- [Jupyter Widgets](#) Adds sliders and other dynamic inputs you can use to make your notebooks more interactive.
- [Spellcheckeing](#) Adds spellchecking for english language in markdown cells.
- [Variableinspector](#) Adds a tab showing the variables which exist in the current session.
- [Code Formatter](#) Formats your code in the selected style.
- [Jupyterlab Celltags](#) Allows to add Tags (meta information) to cells. This addon is part of the jupyter lab core since the 2.0 release.
- [Jupyterlab TOC](#) Adds a TOC (Table of Content) navigation panel.
- [Jupyterlab Shortcutui](#) Adds a graphical shortcut editor.

Common Pitfalls/Missconceptions

Execution order matters/Interpreter runtime memory

Since normally all cells in jupyter-notebook are executed with the same kernel, the order of execution matters. If you for example define a variable `a=2` at the start of a notebook, run some code depending on it (i.e. `[In] a+5`, `[Out] 7`) in a different cell, redefine `a=3` at the end of your notebook and run the cell with code depending on `a` again, it will give a different output (`[Out] 8`) since `a` has changed. This allows you to run parts of your code without having to rerun everything again (i.e. very useful if you have a long running simulation and want to style the plot of the results). But it can also lead to problems, when we don't take this behavior into account when writing the code.

Common Problems:

- **Broken Notebook** - The most severe problem that can occur is that when you open your notebook again and want to run your code (Run All Cells), it fails at a specific cell telling you that something (i.e. function, variable) isn't defined. The most common reason for that is that you deleted/renamed that 'something'/its cell in the last session or defined 'something' in a cell below the cell where it is first needed. Since the kernel still remembered/already knew it (deleting from the notebook \neq deleting from the kernel), everything worked fine, but when you started a new session this 'something', wasn't there when it was needed, since the cells are evaluated in order from top to bottom.

Note:

To solve ordering problems you can move cells in the notebook.

- **It doesn't change, even so it should** - This is an error, which will lead to a **Broken Notebook** and often wastes quite some time to find. This mostly happens if you rename (refactor) a function/variable in one part of your code, but forgot to change it somewhere else, since the kernel remembers the old function/variable, changing the new one won't affect parts where the old name is still used.

Solution

To make sure that the notebook runs properly even if you run it for the first time, you should **always** confirm that everything is runnable with a fresh kernel **after refactoring** code or **before stopping to work on it**. You can confirm this by running `Restart Kernel` and `Run All Cells`....

Not rendered Plots

By default plots aren't rendered in the output cells. To activate this for `matplotlib` (which is also the default backend for `pandas`) you need to add the magic command `%matplotlib inline` or `%matplotlib notebook`.

For other libraries like `bokhe` or `plotly` there are special plugins needed to work in jupyter-lab, since they rely on extra javascript code, those plugins are called [jupyter extensions](#).

ModuleNotFoundError

During this course we will introduce different libraries and how to use them. Depending on your Python installation, those might not be installed, which will cause the `ModuleNotFoundError` when you try to run the examples provided. We recommend to use Anaconda as your installation of choice, since the installers are available for all major platforms (Linux, Windows and MacOS) and most modules come as binaries (called wheels in Python), which means you don't need to compile them yourself. Also the default Anaconda installer comes with most libraries we present you included. If any library is still missing you can install it by running:

```
conda install missing_library_name.
```

If you don't want to use Anaconda or this library is only on PyPi (Python Packaging Index) and not on Anaconda, you can always install it by running:

```
pip install missing_library_name.
```

```
[6]: import not_a_valid_module

-----
ModuleNotFoundError                                Traceback (most recent call last)
<ipython-input-6-b58700dcf11b> in <module>
----> 1 import not_a_valid_module

ModuleNotFoundError: No module named 'not_a_valid_module'
```

4 Examples

5 Cheat Sheets

Collection of cheat sheets for a quick lookup.

5.1 Python

- [python-cheatsheet](#)

5.2 Numpy

- [3rd party](#)

5.3 Pandas

- [official](#)

5.4 Datascience

- [collection](#)

6 Tutorials

6.1 Python

- WebSites
 - [pythonprogramming.net](#)
 - [realpython.com](#)
- Videos
 - [Learning to program with Python 3 \(py 3.7\)](#)
- Mobile Apps
 - [SoloLearn](#)
 - [ProgrammingHub](#)

7 Credits

7.1 Development Lead

- Sebastian Weigand <s.weigand.phy@gmail.com>
- Deniz Sharideh

7.2 Contributors

None yet. Why not be the first?

8 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

8.1 Types of Contributions

Report Errors

Report errors as an [issue](#) at [github](#).

If you are reporting an error, please include:

- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

Fix Bugs

Look through the [GitHub issues](#) for bugs. Anything tagged with “bug” and “help wanted” is open to whoever wants to implement it. You might also encounter typos, spelling and grammar errors, we appreciate all help we can get to make this the best learning experience possible, so don’t be shy and contribute. :)

Implement Topics

Look through the [GitHub issues](#) for features. Anything tagged with “enhancement” and “help wanted” is open to whoever wants to implement it. Tell us that you are working on this topic, so the same work won’t be done by two people at the same time. Of course if someone is already working on a topic you can always offer your help.

Write TL;DR’s or Examples

We will leave the writing of TL;DR’s and examples mostly to the community, since this is the perfect opportunity for **you** to get involved. Not only is it a great start to work with git on an open source project, it will also help you to amplify your understanding of the tools we are teaching you. If you are writing examples make sure that they are documented (markdown cells) and explain what/why it is being done. Also make sure that the example you are using isn’t so specific to your field of studying, that others will have problems understanding (i.e. no detailed knowledge of quantum mechanics should be needed to understand your example.)

Submit Feedback

The best way to send feedback, is to file an [issue at Github](#).

If you are proposing a topic:

- Explain in detail what you would want to learn and why it should be included.
- Keep the scope as narrow as possible and add an example, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

8.2 Structure of the project

If you don’t know which number the chapter you want to work on should have, have a look at issue [#3](#), where the structuring of the course is discussed.

To make navigating through the material consistent and also give new (and old) contributors a good starting point on how to organize and name files and folders, the following structure was proposed in issue [#9](#):

```
Repository-root
|-- material
    |-- <chapter_nr>_<chapter_name>
        |-- data
        |-- images
        |-- <chapter_nr>_<chapter_name>.ipynb
        |-- TL_DR.md
        |-- additional_materials.md
        |-- code_snippets.md
        |-- Examples
            |-- data
```

(continues on next page)

```

        |-- <data_description>-example1.txt
        |-- ...
    |-- example1.ipynb
    |-- ...
|-- cheat_sheets.md
|-- tutorials.md
|-- docs
    |-- material
        |-- <chapter_nr>_<chapter_name>
        |-- <chapter_nr>_<chapter_name>.nblink
    |-- examples
        |-- <chapter_nr>_<chapter_name>
        |-- example1.nblink

```

8.3 Getting Started!

Ready to contribute? Here's how to set up python-tools-for-students for local development.

1. Fork the python-tools-for-students repo on GitHub.

2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/python-tools-for-students.git
```

3. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

Now you can make your changes locally.

4. Install all required libraries:

```
$ pip install -r requirements_dev.txt
```

5. Start jupyter lab in the folder of your local copy and write the changes you want.

6. Make sure all tests pass:

```
$ tox
```

7. Commit your changes and push your branch to GitHub:

```
$ git add .
```

```
$ git commit -m "Your detailed description of your changes."
```

```
$ git push origin name-of-your-bugfix-or-feature
```

8. Submit a pull request through the GitHub website.

Note: You might need to [install git](#) if you haven't done so before. Especially for beginners we recommend [GitKraken](#), which is a graphical user interface for git. But you should definitely learn how to work with git in a terminal, since you might have to work in an environment where you won't have a graphical user interface (i.e. ssh connection to a server/cluster where you want to do your calculation on) or something doesn't work as expected and you need to fix it.

Note for Windows users:

If you want the care free package of a 'properly' configured Posix like shell (more powerful and feature rich command line), just install [cmder full](#) with [get-cmder](#).

8.4 Testing

To make sure that all our notebooks are working properly and have a uniform code style, we test them with:

- `tox`
- `pytest`
- `nbval`
- `flake8-nb`

Where `tox`, `pytest` and `nbval` ensure that the provided notebooks reproducibly work with all supported python versions and `flake8-nb` ensures the [code style](#).

`nbval`

In some cases the output might depend on the operating system/current time or you want to showcase an Exception, in those cases you can use Tags (meta information) to mark a cell for `nbval` to change its testing behavior. For more information have a look at `nbval`'s documentation [Avoid output comparison for specific cells](#) and [Using tags instead of comments](#).

`flake8-nb`

As for `nbval` we also encourage to use [cell tags](#) to configure the reported code style violations of `flake8-nb`. Please only use this scarcely and when absolutely needed, i.e. if you have a cell with different language code or if you want to showcase bad code.

8.5 Add your changes to the docs

To make the provided information more accessible (i.e. on mobile when you are on your way to university), we also generate documentation as an html page, PDF and epub, which is published at [Read The Docs](#).

Adding notebooks

Notebooks are included in the docs using `nbsphinx` and `nbsphinx-link`. In order to add a notebook to the docs, you need to create a `*.nblink` file in the appropriate folder in the docs and add its path to `docs/material.rst` / `docs/examples.rst`. If your notebooks contain extra media like images, you need to add them as extra-media entry in the `*.nblink` file.

Adding markdown files

Markdown files are included in the docs using `myst-parser`. Sadly sphinx does not recognize files outside of the docs root folder (`docs`). So in order not to copy files and maintain two versions, the best solution is to create a new file inside the `docs` folder with the following code, pointing to the appropriate file.

```
```{include} <relative_path_to_the_file_to_be_included>
```
```

After that you can include it in any `*.rst` file [as you would normally](#).

Building the docs locally

To build the documentation, open a terminal, navigate to the docs folder and run `make html` (Posix like) / `make.bat html` (Windows). This will create the documentation inside the folder `docs/_build/html`.

Note: For the docs to be build it is mandatory that you use a conda installation of python or at least have conda installed. This is due to the fact that the notebook inclusion in the docs utilizes the tool [pandoc](#). Even so pandoc is present in many package manager repository indices, this is mostly a too old version, which is why we recommend to use the [version provided by conda](#).

```
$ conda install -c conda-forge pandoc
```

This also requires that the terminal you execute the make command with knows about the conda binary path/s (see [Using Anaconda only if needed](#)).

Note for Windows users:

If you are on Windows and want to use [git bash for Windows](#), you might not have the make command installed. To install make into git bash you can follow [this guide](#) or use `install_make_git_bash_standalone.bat` from [get-cmder](#).

8.6 Style guide

To make the learning and reading experience as pleasant and uniform as possible, as well as giving you pointers to possible pitfalls, we added this style guide. Before you write new content you should check back here, see if something has changed and also refresh your memory on what the style rules for this project are.

- Always capitalize keys for keyboard shortcuts (i.e. Shift+Ctrl)
- Always write commands which can be executed in lower case (i.e. Jupiter-lab should be jupyter-lab)
- No starting or tailing whitespaces in inline equations markdown cells (i.e. `$ \int $` should be `$\int$`), since this will break in the docs ([see](#))

8.7 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

- Respect our folder structure and style guide, since this guarantees a consistent and easy to navigate experience for everyone.
- Make sure that the notebooks work, when running `Restart Kernel` and `Run All ...` and the tests pass.
- If your code needs a 3rd party library to work and it is not yet present in the `requirements.txt`, please add it with a minimum version (i.e.: `package_name>=1.0.0`).
- Add your changes to the docs and make sure that they render properly.

9 Indices and tables

- `genindex`
- `modindex`
- `search`