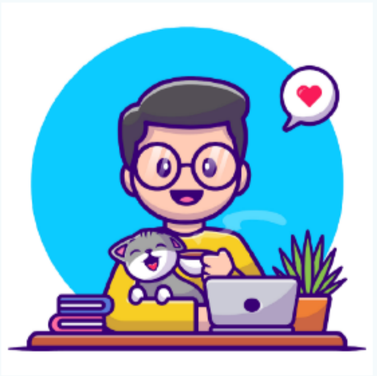
Jupyter Notebook:
Tips & Tricks



Pawel Kubiak

pawel.kubiak@pm.me

Jupyter Notebook?

Interactive notebooks which allows us to mix code and markdown text to create beautiful reports and analysis.

It was spun off from the IPython project in 2014 and has since become one of the most popular environments for data analysis.

Jupyter Notebook?

Interactive notebooks which allows us to mix code and markdown text to create beautiful reports and analysis.

It was spun off from the IPython project in 2014 and has since become one of the most popular environments for data analysis.

Starting Jupyter notebook

```
> pip install notebook
> jupyter notebook
```

Popular implementations

- Official `Jupyter notebook` / `Jupyter lab`
- Google Colab
- Integrations with IDE:
  - Visual Studio Code
  - PyCharm
  - DataSpell
- and more :)

Tip #1: Other kernels

A as name states `Jupyter` started with a support for `Julia`, `Python` and `R` programming languages, but nowadays it support a lot more of them.

Among the most important ones are kernels for `JavaScript`, `Ruby`, `Go`, `Java`, but also for the more unexpected like `C++`, `SQL` or `Assembly`.

ref: https://github.com/jupyter/jupyter/wiki/Jupyter-kernels

Tip #2: Connecting to your `virtualenv`

Using Jupyter, we can not only use kernels built for different programming languages, but also for different versions of python and even for different virtual environments. To make your `virtualenv` visible for the Jupyter server you must:

Tip #2: Connecting to your `virtualenv`

Using Jupyter, we can not only use kernels built for different programming languages, but also for different versions of python and even for different virtual environments. To make your `virtualenv` visible for the Jupyter server you must:

- install `jupyter` package (in your virtualenv)

```
> pip install jupyter
```

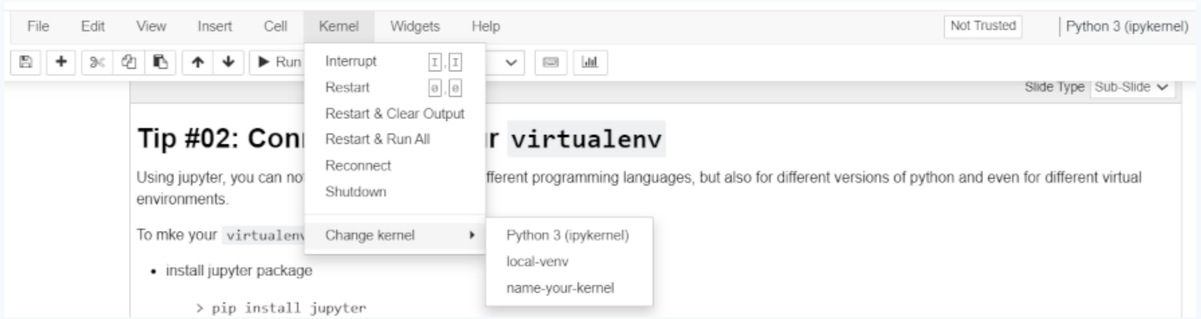
- then register your kernel (from inside of your virtualenv):

```
> ipython kernel install --name "name-your-kernel" --user
```

- then register your kernel (from inside of your virtualenv):

```
> ipython kernel install --name "name-your-kernel" --user
```

- finally your kernel should be selectable from Kernel selection list in your jupyter notebook:



Tip #3: Magic Commands

🔗 https://ipython.readthedocs.io/en/stable/interactive/magics.html

Notebook enable different kinds of magic :D

- `line magic` - every code line started with `%`
- `cell magic` - first line of code started with `%%`

Magic commands perform additional 'magic' on cell content.

Trick #1: `lsmagic`

List all available magic commands

```
%lsmagic
```

```
Available line magics:
%alias  %alias_magic  %autoawait  %autocall  %automagic  %autosave  %bookmark  %cat  %cd  %clear  %col
ors  %conda  %config  %connect_info  %cp  %debug  %dhist  %dirs  %doctest_mode  %ed  %edit  %env  %gui
%hist  %history  %killbgscripts  %ldir  %less  %lf  %lk  %ll  %load  %load_ext  %loadpy  %logoff  %log
on  %logstart  %logstate  %logstop  %ls  %lsmagic  %lx  %macro  %magic  %man  %matplotlib  %mkdir  %mo
re  %mv  %notebook  %page  %pastebin  %pdb  %pdef  %pdoc  %pfile  %pinfo  %pinfo2  %pip  %popd  %pprin
t  %precision  %prun  %psearch  %psource  %pushd  %pwd  %pycat  %pylab  %qtconsole  %quickref  %recall
%rehashx  %reload_ext  %rep  %rerun  %reset  %reset_selective  %rm  %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  %%debug  %%file  %%html  %%javascript  %%js  %%latex  %%markdow
n  %%perl  %%prun  %%pypy  %%python  %%python2  %%python3  %%ruby  %%script  %%sh  %%svg  %%sx  %%syst
em  %%time  %%timeit  %%writefile

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

Trick #2: `prun`

Time Profile execution of given cell

```
%%prun
for i in range(10):
    a = np.random.normal(size=(10, 10))
    b = np.random.normal(size=(10, 10))
    c = a @ b
    time.sleep(0.1)
```

Trick #3: `time`

Measure execution time

```
%%time
for i in range(10):
    time.sleep(0.1)
```

```
CPU times: user 854 µs, sys: 1.05 ms, total: 1.9 ms
Wall time: 1 s
```

Trick #3: `time`

Measure execution time

```
%%time
for i in range(10):
    time.sleep(0.1)
```

```
CPU times: user 854 µs, sys: 1.05 ms, total: 1.9 ms
Wall time: 1 s
```

```
%time x = sum(range(1_000_000))

%time y = sum(range(10_000_000))
```

```
CPU times: user 8.44 ms, sys: 0 ns, total: 8.44 ms
Wall time: 8.27 ms
CPU times: user 70.9 ms, sys: 0 ns, total: 70.9 ms
Wall time: 70.6 ms
```

Trick #4: `timeit`

Precisely measure execution time of given statement

```
#%%timeit

%timeit 2**100

%timeit 3**100
```

```
202 ns ± 1.81 ns per loop (mean ± std. dev. of 7 runs, 10,000,000 loops each)
220 ns ± 3.92 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
```

Trick #5: `capture`

Capture output (`stdout` / `stderr`) of the cell and store it in given variable

```
%%capture output
for i in range(5):
    print(i, i**2)
```

```
print(output)
```

```
0 0
1 1
2 4
3 9
4 16
```

Trick #6: `autoreload`

🔗 https://ipython.org/ipython-doc/3/config/extensions/autoreload.html

When editing source code of modules already imported in our notebook, we must restart whole kernel and restart all cells to apply changes, it is very anoyying!

Autoreload module for the rescue!

```
%load_ext autoreload
%autoreload 2
```

and now all imported modules are reloaded before executing any cell **when source code change is detected**!

Tip #4: `sh` in notebook

🔗 https://jakevdp.github.io/PythonDataScienceHandbook/01.05-ipython-and-shell-commands.html

Notebook setup often require executing some system commands (installing packages, copying file, ...).

You can do it with python, but you can do it better with Jupyter :P

Executing commands

Prepend your `sh` command with `!` (exclemation mark).

```
!whoami
!ls
!cal 2022 | head
#!curl http://example.com
```

```
solmyr
Jupyter.Tips.And.Tricks.ipynb  generated_report.ipynb  sparkline.ipynb
LightningTalk                  initialization.ipynb    your_report.ipynb
README.md                      requirements.txt
__pycache__                    rise.css
                                2022
      January                   February                  March
Su Mo Tu We Th Fr Sa  Su Mo Tu We Th Fr Sa  Su Mo Tu We Th Fr Sa
                   1            1  2  3  4  5            1  2  3  4  5
 2  3  4  5  6  7  8   6  7  8  9 10 11 12   6  7  8  9 10 11 12
 9 10 11 12 13 14 15  13 14 15 16 17 18 19  13 14 15 16 17 18 19
16 17 18 19 20 21 22  20 21 22 23 24 25 26  20 21 22 23 24 25 26
23 24 25 26 27 28 29  27 28                 27 28 29 30 31
30 31
```

Passing Values to and from the Shell

```
contents = !ls

print(contents)
```

```
['Jupyter.Tips.And.Tricks.ipynb', 'LightningTalk', 'README.md', '__pycache__', 'generated_report.ipyn
b', 'initialization.ipynb', 'requirements.txt', 'rise.css', 'sparkline.ipynb', 'your_report.ipynb']
```

Passing Values to and from the Shell

```
contents = !ls

print(contents)
```

```
['Jupyter.Tips.And.Tricks.ipynb', 'LightningTalk', 'README.md', '__pycache__', 'generated_report.ipyn
b', 'initialization.ipynb', 'requirements.txt', 'rise.css', 'sparkline.ipynb', 'your_report.ipynb']
```

```
pattern = "*.ipynb"

! ls {pattern}
```

```
Jupyter.Tips.And.Tricks.ipynb  initialization.ipynb  your_report.ipynb
generated_report.ipynb         sparkline.ipynb
```

Tip #5: Ipywidgets

🔗 https://ipywidgets.readthedocs.io/

You can make your notebook even more interactive using `ipywidgets`!

Basic interactions

```
from ipywidgets import interact

def f(x, y):
    return str(x**2 + 5) + y

interact(f, x=(-10, 10), y=['a', 'b', 'c']);
```

Interactive `matplotlib`

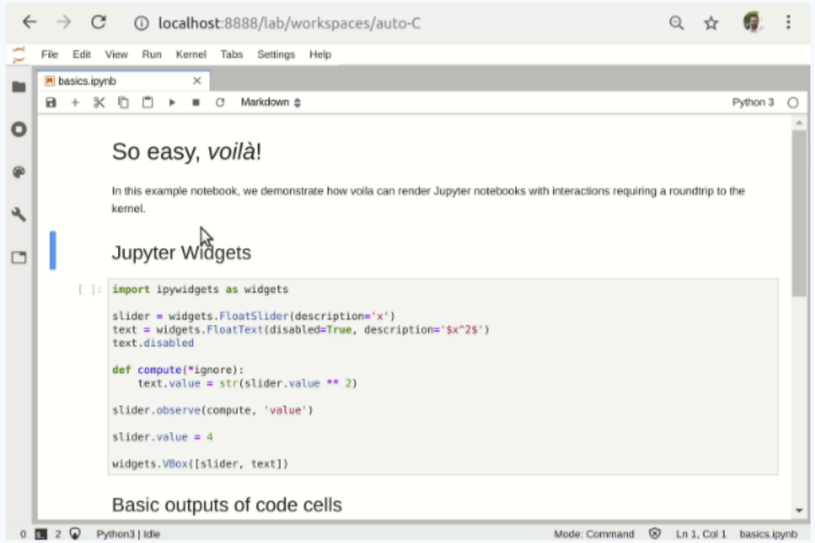
Simple widget for manual Linear regression:

```
%matplotlib inline
from ipywidgets import interactive

Xs = np.random.uniform(-10, 10, 30)
Ys = 0.3*Xs + 1.0 + np.random.normal(0, 0.2, 30)

def f(m, b):
    plt.figure(figsize=(8,4))
    x =  np.linspace(-10, 10, num=1000)
    plt.scatter(Xs, Ys, marker='.')
    plt.plot(x, m * x + b, color='red')
    plt.ylim(-5, 5)

    Yp = m*Xs + b
    mse = ((Ys-Yp)**2).mean()
    plt.title(f"Linear Regression (MSE: {mse:.2f})")
```

```
interactive(f, m=(-2.0, 2.0, 0.1), b=(-3, 3, 0.1))
```

et `voila`!

If you would like to share your interactive analysis with the world, `voila` is the perfect tool for that! It allows you to easily turn Jupyter Notebook into a full-featured web application with single terminal command!



Tip #6: HTML in notebooks

Using `markdown` is nice, but often it is not enought.

You probably would like to use HTML in you notebook?

HTML in markdown

No problem! You can use it in markdown, just surround it with some whitespaces

```
Hello world

<font color="red">Warning</font>
```

HTML in markdown

No problem! You can use it in markdown, just surround it with some whitespaces

```
Hello world

<font color="red">Warning</font>
```

Hello world

Warning

Generating HTML in python

To display HTML from python code, use `display_html` method. You can use it at the same time displaying graphs or Dataframes.

Generating HTML in python

To display HTML from python code, use `display_html` method. You can use it at the same time displaying graphs or Dataframes.

```
from IPython.display import display_html

display_html(f"""
<h2>Header</h2>
<table>
    <tr>
        <td><b>Hello</b></td>
        <td>World</td>
        <td>{np.random.uniform(5)}
    </tr>
</table>""", raw=True)
```

Header

Hello	World	1.5226218812454029

Pretty printing using HTML

You may have wondered how it's possible that `pandas` DataFrames in Notebooks, display as beautifully styled tables? It's simple, they just use the `_repr_html_` method.

Pretty printing using HTML

You may have wondered how it's possible that `pandas` DataFrames in Notebooks, display as beautifully styled tables? It's simple, they just use the `_repr_html_` method.

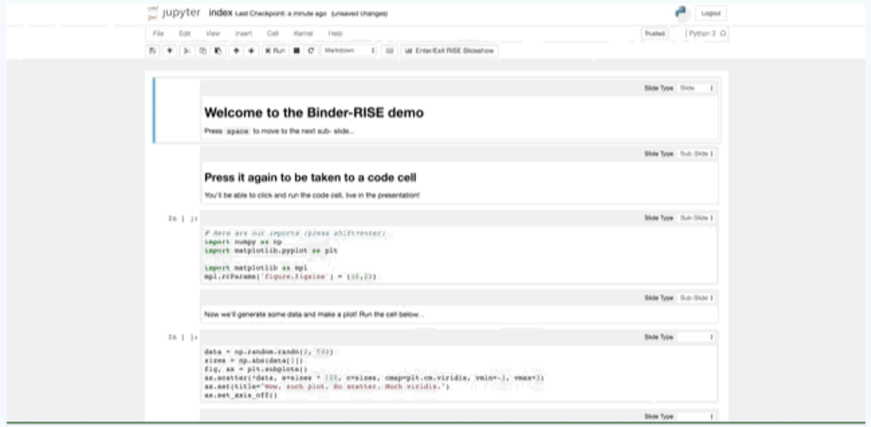
```
class ColorBox:
    def __init__(self, text, color):
        self.text = text
        self.color = color

    def _repr_html_(self):
        return f"<span style='background:{self.color}; padding:10px'>{self.text}</span>"

ColorBox("Hello World", "lightgreen")
```

Hello World

Tips #7: RISE presentations

🔗 https://rise.readthedocs.io/

RISE extension simply convert Jupyter Notebook into interactive `reveal.js` presentation. Slides can be build using HTML and CSS. This whole presentation is created using RISE ;)



Start with installing RISE extension:

```
$ pip install RISE
```

Then enable RISE toolbar: `Menu View > Cell Toolbar > Slideshow`. Now for each cell you can set `Slide Type` to control how RISE should treat it:

- `Empty`: cell works together with previous cell,
- `Slide`: create main-line slide (vertical order),
- `Sub-Slide`: create sub-slide (horizonal order),
- `Fragment`: cell appears on slide after click,
- `Skip`: not visible on slideshow ,

Pros #1: Our presentation is very interactive

Pros #2: You can modify slides on-line (by clicking on any cell).

Pros #3: By pressing character `t` you can open speaker notes view.

Pitfall #1: However, if you needed to export such a presentation to PDF, it might be a bit problematic, take then a look at decktape package.

Tips #8: Notebooks imports

🔗 https://ipynb.readthedocs.io/

Code written in notebooks generaly is not very reusable and often it will be copied between multiple notebooks / places. It can't be DRY.

Tips #8: Notebooks imports

🔗 https://ipynb.readthedocs.io/

Code written in notebooks generaly is not very reusable and often it will be copied between multiple notebooks / places. It can't be DRY.

... But wait, you can use `ipynb` package :D

```
pip install ipynb
```

Full Imports

If you have a notebook file named `server.ipynb`, you can import it via:

```
import ipynb.fs.full.server
```

You can use the `from ... import ..` too.

```
from ipynb.fs.full.server import X, Y, X
```

Definitions only import

Sometimes your notebook has been used as a way to run an analysis or other computation, and you only want to import the functions / classes defined in it - and not the extra statements you have in there. This can be accomplished via `ipynb.fs.defs`:

```
import ipynb.fs.defs.server
```

or using `from ... import ..` syntax:

```
from ipynb.fs.defs.server import fun1
```

Example

Import function `show_sparkline` from `sparkline.ipynb`:

```
from ipynb.fs.defs.sparkline import show_sparkline

show_sparkline("Price")
```

```
Price: ▆▁ ▁▄▆▁▆▃▅
```

Testing Notebooks

If you haven't heard of this functionality until now, you may find it ideal for testing the code contained in the notebook. There is an idea, but there is a better tool for this :D

Testing Notebooks

If you haven't heard of this functionality until now, you may find it ideal for testing the code contained in the notebook. There is an idea, but there is a better tool for this :D

Package is called `testbook` and allows us to easily extract and execute functions or entire cells from inside the notebook. In addition, it integrates very well with `pytest`.

```
from testbook import testbook

@testbook('/path/to/example_notebook.ipynb', execute=True)
def test_func(tb):
    func = tb.get("func")
    assert func(1, 2) == 3
```

Include complete notebooks

If the previous method turned out to be too hackneyed for you, or you are forced to copy sizable fractions of code (e.g. initialization) between notebooks, you can try to organize your notebooks in a different way.

We can use another magic command `%run`, which, placed in the code, will result in the execution of the entire indicated notebook at this time:

Include complete notebooks

If the previous method turned out to be too hackneyed for you, or you are forced to copy sizable fractions of code (e.g. initialization) between notebooks, you can try to organize your notebooks in a different way.

We can use another magic command `%run`, which, placed in the code, will result in the execution of the entire indicated notebook at this time:

```
%run initialization.ipynb
print(f"> CONSTANT = {CONSTANT}\n")
%run sparkline.ipynb
```

```
Initializing repo
> CONSTANT = 42

Sparkline: ▁▂▂▃▃▄▄ ▆▆ ▇
```

Using this method we can build an entire notebook from separate "bricks".

Tips #9: Installing packages

When you need to install additional package for your Notebook, you may quickly run:

```
!pip install numpy
```

in a code cell, but be awere it may fail sometimes (eg. when using kernel from virtualenv).

Tips #9: Installing packages

When you need to install additional package for your Notebook, you may quickly run:

```
!pip install numpy
```

in a code cell, but be awere it may fail sometimes (eg. when using kernel from virtualenv).

The better option is to use `pip` package directly:

```
import pip
pip.main(["install", "numpy"])
```

Tip #10: Clean & Tidy

A frequently pointed problem with Jupyter Notebooks is the difficulty of maintaining their quality, leading many to believe that they are mainly suitable for prototyping rather than production applications.

One possible prescription for this problem, is to use the `nbQA` package!

It allows us to run code quality-checking tools (such as `black`, `isort`, `flake8`, ...) on our notebooks, and often even makes corrections by itself.

It's really simple to use:

```
> nbqa flake8 my_notebook.ipynb
```

Tips #11: Ins & Outs

If you have forgotten to assign results of your compution to variable and you're afraid you've lost your working hours, don't worry, be happy :P

```
np.random.uniform(10)
```

```
9.806893240688936
```

You can access output of every cell with variable `_<N>` or via `Out[]` array. In addition result of most recent computation is available as `_` variable.

```
_22
```

```
9.806893240688936
```

```
Out[22]
```

```
9.806893240688936
```

You can also access source code of every cell with `_i<N>` or `In[]` array.

```
_i22
```

```
'np.random.uniform(10)'
```

```
In[22]
```

```
'np.random.uniform(10)'
```

Tips #12: papermill

🔗 https://papermill.readthedocs.io/

Have you ever had to perform the same analysis for different sets of parameters (date ranges, users segments, marketplaces)?

Running the same notebook multiple times for different parameter sets can be tiring and uncomfortable :'(

Tips #12: papermill

🔗 https://papermill.readthedocs.io/

Have you ever had to perform the same analysis for different sets of parameters (date ranges, users segments, marketplaces)?

Running the same notebook multiple times for different parameter sets can be tiring and uncomfortable :'(

... don't worry, `Papermill` is a tool for parameterizing and executing Jupyter Notebooks :P

1. Define parameters cell by taggin it as `parameters`. If you can't see cell tags, enable them from `View > Cell Toolbar` menu.

1. Define parameters cell by taggin it as `parameters`. If you can't see cell tags, enable them from `View > Cell Toolbar` menu.

2. Inspect parameters detected by `papermill`:

```
> papermill --help-notebook your_report.ipynb

Parameters inferred for notebook 'your_report.ipynb':
  CULTURE: Unknown type (default "fr-FR")
  DEVICE: Unknown type (default "mobile")
```

3. Generate notebook with overwritten parameters:

```
> papermill -p CULTURE en-US -p DEVICE tv \
    your_report.ipynb generated_report.ipynb
```

3. Generate notebook with overwritten parameters:

```
> papermill -p CULTURE en-US -p DEVICE tv \
    your_report.ipynb generated_report.ipynb
```

4. Your generated notebook is `generated_report.ipynb`:

Tips #13: Sharing Notebooks

If you would like to publish your notebook in the internet, then there are numerous ways to do it.

You can put it on your GitHub, or export to HTML using `nbconvert`, you can also present it with `nbviewer` or deploy it as a live Jupyter Notebook instance using `Binder`. And probably in many more ways ;)

Tips #13: Sharing Notebooks

If you would like to publish your notebook in the internet, then there are numerous ways to do it.

You can put it on your GitHub, or export to HTML using `nbconvert`, you can also present it with `nbviewer` or deploy it as a live Jupyter Notebook instance using `Binder`. And probably in many more ways ;)

Unfortunately if you would like to publish it on your own website (blog or Confluence page), it may be a little more problematic :(

Tips #13: Sharing Notebooks

If you would like to publish your notebook in the internet, then there are numerous ways to do it.

You can put it on your GitHub, or export to HTML using `nbconvert`, you can also present it with `nbviewer` or deploy it as a live Jupyter Notebook instance using `Binder`. And probably in many more ways ;)

Unfortunately if you would like to publish it on your own website (blog or Confluence page), it may be a little more problematic :(

... don't worry `nbembed.js` is here to help you ;)

Warning: Self-promotion !!!

🔗 https://github.com/pkubiak/nbembed.js

1. Upload your notebook (in `*.ipynb` format) as attachement to confluence page,
2. Copy link to it. It should looks like: `https://<confluence-url>/download/attachments/<number>/<filename>`
3. Insert HTML widget on your page, with following code:

```
<script src="https://pawelkubiak.me/nbembed.js/dist/nbembed.js"></script>
<notebook src="<LINK_TO_YOUR_NOTEBOOK>"
          config="hide-prompts,hide-stderr"></notebook>
```

Benefits

- We can use notebooks from public GitHub repos or uploaded as Confluence page attachment
- Rendered notebook is sandboxed from the outside
- Interactive content is allowed
- Display properties can be configured

Usage Examples

- Sample notebook

Thank you for your attention :)

Presentation repo: github.com/pkubiak/jupyter-notebook-tips-and-tricks

My other projects: pawelkubiak.me / github.com/pkubiak