Ipython Notebook

Mathurin MASSIAS

February 7, 2017

Contents

1 Notebook readability

1.1 Structure

The first cell of your notebook should contain all of your imports, so that the dependencies are clear to the reader. It also makes correction of partial pieces of your code easier.

Your cells should be as atomic as possible, meaning that one cell corresponds to an independent piece of code (for example, one question or one subquestion). Independent cells make your code easier to read. They also avoir running code uselessly: when you need to rerun a cell (to find a bug, to modify a plot, because you corrected a typo, etc.), if you have some heavy computation at the beginning, you have to execute it again which is a waste of time. If this piece of code is isolated in its own cell, you can run it only once and debug/plot in the following one.

1.2 Presenting results

You can add a cell in Markdown format, to interpret your results or explain your algorithms. Avoid using `print()` as much as possible (if the cell is long, it is difficult to track its orgin). Also, Markdown format supports LaTex, so you can include formulas in your text.

When you use `print()`, do not print too much stuff (large arrays, digits with 20 decimal).

If you are timing code, you should use the `%timeit` magic, which does all the work for you and format results nicely (is it really useful to have precision up to 10^{-18} s?). Which one of the following alternatives do you prefer:

```
In [1]: import time
        import numpy as np
        np.random.seed(2406)
        n, p = 1000, 5000
        X = np.random.randn(n, p)
        alpha = 1e-5
```

```
In [2]: %timeit -n 1 -r 1 np.linalg.inv(np.dot(X.T, X) + alpha * np.eye(p))
```

```
1 loop, best of 1: 4.82 s per loop
```

```
In [3]: start = time.time()
        test = np.linalg.inv(np.dot(X.T, X) + alpha * np.eye(p))
        end = time.time()
        print(end - start)
        print("total time: %f s" % (end - start))
```

```
4.829550504684448
total time: 4.829551 s
```

`%timeit` avoids polluting your namespace with `start` and `end` variables. You can use the n parameter to modify the numbers of times the code is run in a loop, and r to change the number of loops performed. You can use `%%timeit` to time the execution of a whole cell rather than a single line of code. Read the doc for more info.

1.3 import * and %pylab inline

Unless you know exactly what you are doing, I advise you against using `%pylab inline` or `from myModule import *`, because you (and the reader) end up forgetting where the variables are imported from. For example, there are differences between `numpy.linalg.norm` and `scipy.linalg.norm`, or between `numpy.linalg.eig` and `scipy.linalg.eig`; if they are just called `norm` and `eig` with no explicit import to trace them, it's confusing.

Let's see what `%pylab inline` imports:

```
In [1]: print(len(dir()))  # dir() returns the list of names in the current local scope
```

```
23
```

```
In [2]: %pylab inline
```

```
Populating the interactive namespace from numpy and matplotlib
```

```
In [3]: print(len(dir()))
```

```
1001
```

So there are nearly a thousand additional variables polluting your scope now (autocompletion now suggests lots of choices), and if you don't do this in the first cell, you might override some variables: common variable names such as `test`, `clf`... are now bound to variables imported by pylab.

1.4 Prototyping

Bcause of their interactive properties, notebooks are a very powerful tool to prototype. Everyone tends to rerun cells multiple times, change their order, change variable names... Because deleting a cell does not delete variables defined in it (which is a good thing), some variables may continue to exist even though you don't declare them anymore, giving you the false impression that your code runs fine whereas another person won't be able to run it.
→ before handing over your notebook, restart your kernel and run all cells to check that everything is fine, even if it takes a few minutes.

2 Useful shortcuts

2.1 In command mode

Command mode is when you are not editting the content of a cell. If you are editting a cell, you have to press Esc to enable command mode.

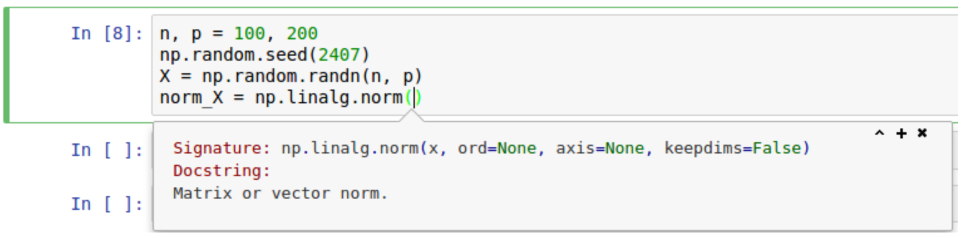
- The most important one: h to show the shortcuts
- ctrl + ↵ to execute a cell a keep this cell selected
- ⇧ + ↵ to execute this cell and select the next
- alt + ↵ to execute the current cell and create an empty one below
- I, I (pressed twice) to interrupt the kernel
- x to cut the cell
- c to copy the cell
- v to paste the copied cell (below currently selected cell)
- ⇧ + v to paste the copied cell (above currently selected cell)
- d, d (pressed twice) to **d**elete cell. Can be undone with ctrl + z.
- a to insert cell **a**bove
- b to insert cell **b**elow

2.2 In edit mode

Edit mode when you are editting the content of a cell (your cursor is blinking inside the cell).

- ctrl + ↑ to go to start of cell
- ctrl + ↓ to go to end of cell
- ctrl + ⇧ + - to split cell (very useful but does not work on every keyboard...)
- ctrl + ← to go one word left. Also works with right. When used while holding ⇧, this allows you to quickly select portions of text.
- ctrl + ↵, ⇧ + ↵ and alt + ↵ also work in edit mode.

When typing the name of a function, hitting ⇧ + ⇥ (tab, not right arrow) after the function name (or inside the parenthesis of the function call) will display a short version of this function's help. This is very useful to remember the order of the parameters, their names or their default values.



```
In [8]: n, p = 100, 200
        np.random.seed(2407)
        X = np.random.randn(n, p)
        norm_X = np.linalg.norm()

In [ ]:  Signature: np.linalg.norm(x, ord=None, axis=None, keepdims=False)
         Docstring:
In [ ]:  Matrix or vector norm.
```

2.3 Commenting and uncommenting blocks of code (more advanced)

There is no default shortcut, but you can add a custom one (in this case, ctrl + ,). Add these lines to your `custom.js` file:

```
define([
    'base/js/namespace',
    'base/js/events'
    ],
    function(IPython, events) {
        events.on("app_initialized.NotebookApp",
            function () {
                IPython.Cell.options_default.cm_config.extraKeys =
                    {"\ctrl-," : "toggleComment"};
            }
        );
    }
);
```

If you are using anaconda 3 as recommended, your `custom.js` should be located in

```
~/anaconda3/envs/myCondaEnvName/lib/python2.7/site-packages/notebook/
    static/custom/custom.js
```

if you're using a conda env (python2.7 is the version of python inside the conda env, change its name accordingly), or in

```
~/anaconda3/lib/python3.5/site-packages/notebook/static/custom/custom.js
```

if you use plain conda (replace `python3.5` with your python version).

Once this is done, restart jupyter. You can comment and uncomment blocks selecting them, then hitting ctrl + , .

3 bash

It is possible to type bash commands inside a notebook; just type `!` before the line of code. This is useful to avoid having to switch between your navigator and your file explorer, opening files to see the first lines, etc. You can do all this from inside your notebook:

```
In [1]: !pwd

/home/mathurin/workspace/teaching/notebooks


In [2]: !ls
        !echo # blank line for readability
        !head -n 5 defraconsumption.csv # useful to see the separator used in a csv file

bash.ipynb  contre_exemples.ipynb  defraconsumption.csv  time.tex
bash.tex    contre_exemples.tex    time.ipynb

;England;Wales;Scotland;N Ireland
Cheese;105;103;103;66
Carcass meat;245;227;242;267
Other meat;685;803;750;586
Fish;147;160;122;93
```

```
In [3]: !jupyter nbconvert --to latex bash.ipynb # exporting this ipynb to LaTeX format

[NbConvertApp] Converting notebook bash.ipynb to latex
[NbConvertApp] Writing 15898 bytes to bash.tex
```

4 Frequent issues

4.1 My plots are invisible

Did you use `%matplotlib inline` after `import matplotlib.pyplot as plt` ?

4.2 Source filed has changed, but nothing changes in the notebook

When you modify a source file you import in your notebook, it's safer to restart your kernel rather than just rerunning `from my_source import my_function`. `import` in Python means "if it has not already been imported, import", so rerunning `import` does not actually reload the function. If restarting your kernel is too expensive, have a look at the `autoreload` module.

4.3 Python version issue

If you have trouble with the version of Python you're using (python 2 and python 3 installed, virtualenvs, conda envs...) and you want to check which one is used, type:

```
import sys
print(sys.executable)
```

Be aware that silent issues can happen if you start jupyter notebook from inside a conda env, while ipython is not installed inside this env: the executable used is conda root instead of the one of your conda env; in that case, printing `sys.executable` will show you that you are using the wrong executable of python (which explains why you have the wrong version or you cannot import your modules).

4.4 Module and module version

If you have troubles importing a module, this StackOverflow question shows how to print a list all installed modules. `myModule.__path__` and `myModule.__version__` can also be useful.