# 10 tips on using Jupyter Notebook 使用Jupyter笔记本的10个技巧

原文链接：  
原文链接：



img  
img公司

(a.k.a iPython Notebook) is brilliant coding tool. It is ideal for doing reproducible research. Here is my list of 10 tips on structuring Jupyter notebooks, I worked out over the time.  
（也就是iPython笔记本）是一个出色的编码工具。它是进行可重复性研究的理想选择。这是我列出的10个关于构建Jupyter笔记本的技巧，我一直在练习。

#### 1. Use virtualenv to create self-contained environment 一。使用virtualenv创建自包含环境

You might be tempted to install all research libraries within your operating system and share them among all your projects. Soon you will discover that when you add some additional library it may update ones installed previously. Some of the other libraries will no longer work with newer versions. So when you go back to a previous project you will waste a lot of time trying to figure out what changed and how to fix it.  
您可能会尝试在操作系统中安装所有研究库，并在所有项目中共享它们。很快你就会发现，当你添加一些额外的库时，它可能会更新以前安装的库。其他一些库将不再使用较新版本。因此，当你回到以前的项目时，你会浪费很多时间去弄清楚什么改变了以及如何修复它。

The solution is to use separate virtual environment for each of your projects. I recommend using via . To avoid problems with resolving paths to the virtual environment you should install Jupyter in each environment separately.  
解决方案是为每个项目使用单独的虚拟环境。我建议使用via。为了避免解析到虚拟环境的路径时出现问题，应该分别在每个环境中安装Jupyter。

#### 2. Go with Python 3 2。使用Python 3

It’s better. Really.  
好多了。真正地。

#### 3. Include requirements.txt 三。包含requirements.txt

When you have a separate environment for your project, it is a good idea to save the list of dependencies. It will save you a lot of time in the future. For example when you will want to recreate the environment.  
当您的项目有单独的环境时，最好保存依赖项列表。它会在将来为你节省很多时间。例如，何时需要重新创建环境。

$ pip freeze > requirements.txt  
$pip freeze>需求.txt

#### 4. Do all imports in the first cell 四。在第一个单元格中执行所有导入

Do all your imports in the first cell of your notebook. It has two benefits. The dependencies and tools used are obvious at the first glance. When you restart the notebook server, you can have all your imports restored with a single re-run. It is especially useful when you don’t want to re-execute the entire notebook.  
在笔记本的第一个单元格中完成所有导入操作。它有两个好处。所使用的依赖项和工具乍一看是显而易见的。重新启动笔记本服务器时，可以通过一次重新运行来还原所有导入。当您不想重新执行整个笔记本时，它特别有用。

I also use this cell to define any filesystem paths used in the notebook.  
我还使用此单元格定义笔记本中使用的任何文件系统路径。

#### 5. Start dirty and keep your draft 5个开始脏兮兮的保持你的草稿

Start quick and dirty. The fastest you get to what you want to do, the better. The inspiration is perishable [, by Jason Fried]. But when you notice that you start stepping on your own toes, that you are no longer effective and the development become clumsy, it is time to organize the notebook. Start over, copy the good code, rewrite and generalize bad one, but whatever you do: KEEP THE DRAFT NOTEBOOK!  
快点下手。你做你想做的事越快越好。灵感是易腐的。但当你注意到你开始自力更生，你不再有效，发展变得笨拙时，是时候整理笔记本了。从头开始，复制好的代码，重写和概括坏的代码，但无论你做什么：保留草稿笔记本！

#### 6. Wrap cell content in a function 6。在函数中包装单元格内容

Many of the notebook cells will look like this:  
许多笔记本电池的外观如下：

parameter1 = 1.0  
parameter2 = 100  
step1 = X \* parameter1  
step1 \* parameter2  
参数1=1.0参数2=100步骤1=X\*参数1步骤1\*参数2

There are parameters at the beginning of the cell. You change them and re-execute the cell or you even copy the entire cell and modify parameters. There are some intermediate computations and at the end, there is a line to display the results.  
单元格开头有参数。您可以更改它们并重新执行单元，甚至可以复制整个单元并修改参数。有一些中间计算，在最后，有一行显示结果。

It’s ok in the draft. But after a while it becomes unmanageable. You got plenty of intermediary variables trashing a global namespace. You lose the steps that led you to the current parameter choices.  
草案里没问题。但过了一段时间它就变得无法控制了。有很多中间变量破坏了全局命名空间。您将丢失导致您选择当前参数的步骤。

Instead, you can wrap it all in one a function:  
相反，您可以将其全部打包到一个a函数中：

def computation(parameter1=1.0, parameter2=100):  
 step1 = X \* parameter1  
 return step1 \* parameter2  
computation()  
...  
computation(parameter1=10.0)  
def计算（parameter1=1.0，parameter2=100）：步骤1=X\*parameter1返回步骤1\*parameter2 computation（）…计算（parameter1=10.0）

You can modify the parameters and re-execute in a separate cell, keeping the history of changes. The intermediary steps will no longer trash the global namespace and consume memory.  
您可以修改参数并在单独的单元格中重新执行，保留更改的历史记录。中间步骤将不再丢弃全局命名空间并消耗内存。

#### 7. Use joblib for caching output 7号。使用joblib缓存输出

You thought your neural network for three days and now you are ready to build on top of it. But you forgot to plug your laptop to a power source and it runs out of batteries. So you scream: Why didn’t I pickle!? The answer is: because it is pain in the back. Managing file names, checking if the file exists, saving, loading… What to do instead? Use .  
你花了三天的时间思考你的神经网络，现在你已经准备好在上面建立它了。但你忘了给笔记本电脑插上电源，它就没电了。所以你尖叫：我为什么不泡菜！？答案是：因为背部疼痛。管理文件名，检查文件是否存在，保存，加载……做什么？使用。

from sklearn.externals.joblib import Memory  
memory = Memory(cachedir='/tmp', verbose=0)  
@memory.cache  
def computation(p1, p2):  
 ...  
从sklearn.externals.joblib import Memory Memory=Memory（cachedir='/tmp'，verbose=0）@Memory.cachedef计算（p1，p2）：。。。

With three lines of code, you get caching of the output of any function. Joblib traces parameters passed to a function, and if the function has been called with the same parameters it returns the return value cached on a disk.  
通过三行代码，可以缓存任何函数的输出。Joblib跟踪传递给函数的参数，如果使用相同的参数调用函数，则返回缓存在磁盘上的返回值。

#### 8. Make sections of the notebook self-contained 8个。使笔记本的各个部分独立

Make sections of your notebook loosely bound. Use as little global variables as possible. If you wrap your cells in functions and you use joblib for caching, it is really inexpensive to call same code within each section. It’s better than making code reliable on the variables created several cells above.  
把笔记本的各个部分装订得松散些。尽可能少地使用全局变量。如果将单元格包装在函数中并使用joblib进行缓存，则在每个部分中调用相同的代码非常便宜。这比在上面几个单元格中创建的变量上使代码可靠要好得多。

In general, try to limit the number of cells you have to re-run after the restart to continue on your work.  
通常，请尝试限制重新启动后必须重新运行的单元格数，以便继续工作。

#### 9. Reuse variable names.

Don’t use long variable names. When you get a chance re-use existing ones. It is contrary to the advice I would give when developing other kinds of software, but in case of a notebook this approach works better.

Let me illustrate it with an example. Let’s assume that your algorithms need a list of clusters. You try various versions of clustering and algorithms. Your code can look like this:

clusters\_kmeans\_k10 = KMeans(k=10).fit\_predict(X)  
clusters\_kmean\_k5 = KMeans(k=5).fit\_predict(X)  
# many cells further  
algorithm1(clusters\_kmeans\_k10)  
algorithm2(clusters\_kmeans\_k10)  
algorithm1(clusters\_kmeans\_k5)  
algorithm2(clusters\_kmeans\_k5)

But instead you can use joblib cached function and re-use variables:

@memory.cache  
def kmeans(X, k):  
 return KMeans(k=k).fit\_predict(X)  
# many cells further  
clusters = kmeans(X, k=10)  
algorithm1(clusters)  
algorithm2(clusters)  
clusters = kmeans(X, k=5)  
algorithm1(clusters)  
algorithm2(clusters)

#### 10. Use assertions to test utility functions

When you create some utility function, create short tests using *assert* keyword. For example:

def norm\_scale(X, axis=0):  
 mx = np.max(X, axis=axis)  
 mi = np.min(X, axis=axis)  
 epsilon = 10\*\*-32  
 return (X — mi) / (np.abs(mi) + mx + epsilon)  
norm = norm\_scale(X)  
assert np.min(norm) >= 0  
assert np.max(norm) <= 1

Here are my tips? What are yours? How do you organize your notebooks?



img