Installing Python Packages from a Jupyter Notebook

Tue 05 December 2017

In software, it's said that all abstractions are leaky (https://www.joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions/), and this is true for the Jupyter notebook as it is for any other software. I most often see this manifest itself with the following issue:

> I installed *package X* and now I can't import it in the notebook. Help!

This issue is a perrennial source of StackOverflow questions (e.g. this (https://stackoverflow.com/questions/39007571/running-jupyter-with-multiple-python-and-ipython-paths/), that (https://stackoverflow.com/questions/42500142/importerror-no-module-named-jwt-in-jupyter), here (https://stackoverflow.com/questions/32777807/importerror-no-module-named-cv2-using-jupyter), there (https://stackoverflow.com/questions/42500649/failed-to-import-numpy-as-np-when-i-worked-with-jupyter-notebook), another (https://stackoverflow.com/questions/46634660/jupyter-notebook-wrong-sys-path-and-sys-executable), this one (https://stackoverflow.com/questions/44222513/cannot-import-datashader-installed-using-miniconda), that one (https://stackoverflow.com/questions/42178070/jupyter-notebook-importerror-no-module-named-sklearn), and this (https://stackoverflow.com/questions/42034508/fail-pandas-in-python3-jupyter-notebook)... etc.).

Fundamentally the problem is usually rooted in the fact that the **Jupyter kernels are disconnected from Jupyter's shell**; in other words, the installer points to a different Python version than is being used in the notebook. In the simplest contexts this issue does not arise, but when it does, debugging the problem requires knowledge of the intricacies of the operating system, the intricacies of Python package installation, and the intricacies of Jupyter itself. In other words, the Jupyter notebook, like all abstractions, is leaky.

In the wake of several discussions on this topic with colleagues, some online (exhibit A (https://twitter.com/amuellerml/status/932637063748444160), exhibit B (https://twitter.com/jakevdp/status/922846245848150016)) and some off, I decided to treat this issue in depth here. This post will address a couple things:

- **First**, I'll provide a quick, bare-bones answer to the general question, *how can I install a Python package so it works with my jupyter notebook, using pip and/or conda?*.

- **Second**, I'll dive into some of the background of exactly *what* the Jupyter notebook abstraction is doing, how it interacts with the complexities of the operating system, and how you can think about where the "leaks" are, and thus better understand what's happening when things stop working.

- **Third**, I'll talk about some ideas the community might consider to help smooth-over these issues, including some changes that the Jupyter, Pip, and Conda developers might consider to ease the cognitive load on users.

This post will focus on two approaches to installing Python packages: pip (https://pip.pypa.io/en/stable/) and conda (https://conda.io/docs/). Other package managers exist (including platform-specific tools like yum (http://yum.baseurl.org/), apt (https://help.ubuntu.com/community/AptGet/Howto#Package_management_with_APT), homebrew (https://brew.sh/), etc., as well as cross-platform tools like enstaller (http://enstaller.readthedocs.io/en/latest/)), but I'm less familiar with them and won't be remarking on them further.

Quick Fix: How To Install Packages from the Jupyter Notebook

If you're just looking for a quick answer to the question, *how do I install packages so they work with the notebook*, then look no further.

pip vs. conda

First, a few words on `pip` vs. `conda`. For many users, the choice between pip and conda can be a confusing one. I wrote way more than you ever want to know (https://jakevdp.github.io/blog/2016/08/25/conda-myths-and-misconceptions/) about these in a post last year, but the essential difference between the two is this:

- *pip* installs **python** packages in **any environment**.
- *conda* installs **any** package in **conda environments**.

If you already have a Python installation that you're using, then the choice of which to use is easy:

- If you installed Python using Anaconda or Miniconda, then use `conda` to install Python packages. If conda tells you the package you want doesn't exist, then use pip (or try conda-forge (https://conda-forge.org/), which has more packages available than the default conda

channel).

- If you installed Python any other way (from source, using pyenv, virtualenv, etc.), then use `pip` to install Python packages

Finally, because it often comes up, I should mention that you should **never** use `sudo pip install`.

NEVER.

It will always lead to problems in the long term, even if it seems to solve them in the short-term. For example, if `pip install` gives you a permission error, it likely means you're trying to install/update packages in a system python, such as `/usr/bin/python`. Doing this can have bad consequences, as often the operating system itself depends on particular versions of packages within that Python installation. For day-to-day Python usage, you should isolate your packages from the system Python, using either virtual environments (https://virtualenv.pypa.io/en/stable/) or Anaconda/Miniconda (https://conda.io/docs/user-guide/install/download.html) — I personally prefer conda for this, but I know many colleagues who prefer virtualenv.

How to use Conda from the Jupyter Notebook

If you're in the jupyter notebook and you want to install a package with conda, you might be tempted to use the `!` notation to run conda directly as a shell command from the notebook:

In [1]:
```
# DON'T DO THIS!
!conda install --yes numpy
```

```
Fetching package metadata ...........
Solving package specifications: .

# All requested packages already installed.
# packages in environment at /Users/jakevdp/anaconda/envs/python3.6:
#
numpy                     1.13.3           py36h2cdce51_0
```

(Note that we use `--yes` to automatically answer `y` if and when conda asks for user confirmation)

For various reasons that I'll outline more fully below, this **will not generally work** if you want to use these installed packages from the current notebook, though it may work in the simplest cases.

Here is a short snippet that should work in general:

In [2]:
```
# Install a conda package in the current Jupyter kernel
import sys
!conda install --yes --prefix {sys.prefix} numpy
```

```
Fetching package metadata ..........
Solving package specifications: .

# All requested packages already installed.
# packages in environment at /Users/jakevdp/anaconda:
#
numpy                     1.13.3           py36h2cdce51_0
```

That bit of extra boiler-plate makes certain that conda installs the package in the currently-running Jupyter kernel (thanks to Min Ragan-Kelley (https://twitter.com/minrk/status/842067777150169088) for suggesting this approach). I'll discuss why this is needed momentarily.

How to use Pip from the Jupyter Notebook

If you're using the Jupyter notebook and want to install a package with `pip`, you similarly might be inclined to run pip directly in the shell:

In [3]:
```
# DON'T DO THIS
!pip install numpy
```

```
Requirement already satisfied: numpy in /Users/jakevdp/anaconda/envs/
```

For various reasons that I'll outline more fully below, this **will not generally work** if you want to use these installed packages from the current notebook, though it may work in the simplest cases.

Here is a short snippet that should generally work:

In [4]:
```
# Install a pip package in the current Jupyter kernel
import sys
!{sys.executable} -m pip install numpy
```

```
Requirement already satisfied: numpy in /Users/jakevdp/anaconda/lib/p
```

That bit of extra boiler-plate makes certain that you are running the `pip` version associated with the current Python kernel, so that the installed packages can be used in the current notebook. This is related to the fact that, even setting Jupyter notebooks aside, it's better to install packages using

```
$ python -m pip install <package>
```

rather than

```
$ pip install <package>
```

because the former is more explicit about where the package will be installed (more on this below).

The Details: Why is Installation from Jupyter so Messy?

Those above solutions should work in all cases... but why is that additional boilerplate necessary? In short, it's because in Jupyter, **the shell environment and the Python executable are disconnected**. Understanding why that matters depends on a basic understanding of a few different concepts:

1. how your operating system locates executable programs,
2. how Python installs and locates packages
3. how Jupyter decides which Python executable to use.

For completeness, I'm going to delve briefly into each of these topics (this discussion is partly drawn from This StackOverflow answer (https://stackoverflow.com/questions/39007571/running-jupyter-with-multiple-python-and-ipython-paths/39022003#39022003) that I wrote last year).

Note: the following discussion assumes Linux, Unix, MacOSX and similar operating systems. Windows has a slightly different architecture, and so some details will differ.

How your operating system locates executables

When you're using the terminal and type a command like `python`, `jupyter`, `ipython`, `pip`, `conda`, etc., your operating system contains a well-defined mechanism to find the executable file the name refers to.

On Linux & Mac systems, the system will first check for an alias (http://tldp.org/LDP/abs/html/aliases.html) matching the command; if this fails it references the `$PATH` environment variable:

In [5]:
```
!echo $PATH
```

```
/Users/jakevdp/anaconda/envs/python3.6/bin:/Users/jakevdp/anaconda/envs
```

`$PATH` lists the directories, in order, that will be searched for any executable: for example, if I type `python` on my system with the above `$PATH`, it will first look for `/Users/jakevdp/anaconda/envs/python3.6/bin/python`, and if that doesn't exist it will look for `/Users/jakevdp/anaconda/bin/python`, and so on.

(Parenthetical note: why is the first entry of `$PATH` repeated twice here? Because every time you launch `jupyter notebook`, Jupyter prepends the location of the `jupyter` executable to the beginning of the `$PATH`. In this case, the location was already at the beginning of the path, and the result is that the entry is duplicated. Duplicate entries add clutter, but cause no harm).

If you want to know what is actually executed when you type `python`, you can use the `type` shell command:

In [6]:
```
!type python
```

```
python is /Users/jakevdp/anaconda/envs/python3.6/bin/python
```

Note that this is true of *any* command you use from the terminal:

In [7]:
```
!type ls
```

```
ls is /bin/ls
```

Even built-in commands like `type` itself:

In [8]:
```
!type type
```

```
type is a shell builtin
```

You can optionally add the `-a` tag to see *all available* versions of the command in your current shell environment; for example:

In [9]:
```
!type -a python
```

```
python is /Users/jakevdp/anaconda/envs/python3.6/bin/python
python is /Users/jakevdp/anaconda/envs/python3.6/bin/python
python is /Users/jakevdp/anaconda/bin/python
python is /usr/bin/python
```

In [10]:
```
!type -a conda
```

```
conda is /Users/jakevdp/anaconda/envs/python3.6/bin/conda
conda is /Users/jakevdp/anaconda/envs/python3.6/bin/conda
conda is /Users/jakevdp/anaconda/bin/conda
```

In [11]:
```
!type -a pip
```

```
pip is /Users/jakevdp/anaconda/envs/python3.6/bin/pip
pip is /Users/jakevdp/anaconda/envs/python3.6/bin/pip
pip is /Users/jakevdp/anaconda/bin/pip
```

When you have multiple available versions of any command, it is important to keep in mind the role of `$PATH` in choosing which will be used.

How Python locates packages

Python uses a similar mechanism to locate imported packages. The list of paths searched by Python on import is found in `sys.path`:

In [12]:
```
import sys
sys.path
```

Out[12]:
```
['',
 '/Users/jakevdp/anaconda/lib/python36.zip',
 '/Users/jakevdp/anaconda/lib/python3.6',
 '/Users/jakevdp/anaconda/lib/python3.6/lib-dynload',
 '/Users/jakevdp/anaconda/lib/python3.6/site-packages',
 '/Users/jakevdp/anaconda/lib/python3.6/site-packages/schemapi-0.3.0.
 '/Users/jakevdp/anaconda/lib/python3.6/site-packages/setuptools-27.2
 '/Users/jakevdp/anaconda/lib/python3.6/site-packages/IPython/extensi
 '/Users/jakevdp/.ipython']
```

By default, the first place Python looks for a module is an empty path, meaning the current working directory. If the module is not found there, it goes down the list of locations until the module is found. You can find out which location has been used using the `__path__` attribute of an imported module:

In [13]:
```
import numpy
numpy.__path__
```

Out[13]:
```
['/Users/jakevdp/anaconda/lib/python3.6/site-packages/numpy']
```

In most cases, a Python package you install with `pip` or with `conda` will be put in a directory called `site-packages`. The important thing to realize is that each Python executable has **its own `site-packages`**: what this means is that when you install a package, it is **associated with particular**

python executable and by default can only be used with that Python installation!

We can see this by printing the `sys.path` variables for each of the available `python` executables in my path, using Jupyter's delightful ability to mix Python and bash commands in a single code block:

In [14]:
```
paths = !type -a python
for path in set(paths):
    path = path.split()[-1]
    print(path)
    !{path} -c "import sys; print(sys.path)"
    print()
```

```
/Users/jakevdp/anaconda/envs/python3.6/bin/python
['', '/Users/jakevdp/anaconda/envs/python3.6/lib/python36.zip', '/Use

/usr/bin/python
['', '/System/Library/Frameworks/Python.framework/Versions/2.7/lib/py

/Users/jakevdp/anaconda/bin/python
['', '/Users/jakevdp/anaconda/lib/python36.zip', '/Users/jakevdp/anac
```

The full details here are not particularly important, but it is important to emphasize that *each Python executable has its own distinct paths*, and unless you modify `sys.path` (which should only be done with great care) you cannot import packages installed in a different Python environment.

When you run `pip install` or `conda install`, these commands are associated with a particular Python version:

- `pip` installs packages in the Python in its same path
- `conda` installs packages in the current active conda environment

So, for example we see that `pip install` will install to the conda environment named `python3.6`:

In [15]:
```
!type pip
```

```
pip is /Users/jakevdp/anaconda/envs/python3.6/bin/pip
```

And `conda install` will do the same, because `python3.6` is the current active environment (notice the `*` indicating the active environment):

In [16]:
```
!conda env list
```

```
# conda environments:
#
python2.7                /Users/jakevdp/anaconda/envs/python2.7
python3.5                /Users/jakevdp/anaconda/envs/python3.5
python3.6             *  /Users/jakevdp/anaconda/envs/python3.6
rstats                   /Users/jakevdp/anaconda/envs/rstats
root                     /Users/jakevdp/anaconda
```

The reason both `pip` and `conda` default to the conda `python3.6` environment is that this is the Python environment I used to launch the notebook.

I'll say this again for emphasis: **the shell environment in Jupyter notebook matches the Python version used to *launch* the notebook.**

How Jupyter executes code: Jupyter Kernels

The next relevant question is how Jupyter chooses to execute Python code, and this brings us to the concept of a *Jupyter Kernel*.

A Jupyter kernel is a set of files that point Jupyter to some means of executing code within the notebook. For Python kernels, this will point to a particular Python version, but Jupyter is designed to be much more general than this: Jupyter has dozens of available kernels (https://github.com/jupyter/jupyter/wiki/Jupyter-kernels) for languages including Python 2, Python 3, Julia, R, Ruby, Haskell, and even C++ and Fortran!

If you're using the Jupyter notebook, you can change your kernel at any time using the *Kernel* → *Choose Kernel* menu item.

To see the kernels you have available on your system, you can run the following command in the shell:

In [17]:
```
!jupyter kernelspec list
```

```
Available kernels:
  python3       /Users/jakevdp/anaconda/envs/python3.6/lib/python3.6/
  conda-root    /Users/jakevdp/Library/Jupyter/kernels/conda-root
  python2.7     /Users/jakevdp/Library/Jupyter/kernels/python2.7
  python3.5     /Users/jakevdp/Library/Jupyter/kernels/python3.5
  python3.6     /Users/jakevdp/Library/Jupyter/kernels/python3.6
```

Each of these listed kernels is a directory that contains a file called `kernel.json` which specifies,

among other things, which language and executable the kernel should use. For example:

```
In [18]: !cat /Users/jakevdp/Library/Jupyter/kernels/conda-root/kernel.json
```

```
{
 "argv": [
  "/Users/jakevdp/anaconda/bin/python",
  "-m",
  "ipykernel_launcher",
  "-f",
  "{connection_file}"
 ],
 "display_name": "python (conda-root)",
 "language": "python"
}
```

If you'd like to create a new kernel, you can do so using the jupyter ipykernel command (http://ipython.readthedocs.io/en/stable/install/kernel_install.html#kernels-for-different-environments); for example, I created the above kernels for my primary conda environments using the following as a template:

```
$ source activate myenv
$ python -m ipykernel install --user --name myenv --display-name "Python (myenv)"
```

The Root of the Issue

Now we have the full background to answer our question: *Why don't `!pip install` or `!conda install` always work from the notebook*?

The root of the issue is this: the shell environment is determined when the Jupyter notebook is launched, while the Python executable is determined by the kernel, and the two do not necessarily match. In other words, there is no guarantee that the `python`, `pip`, and `conda` in your `$PATH` will be compatible with the `python` executable used by the notebook.

Recall that the `python` in your path can be determined using

```
In [19]: !type python
```

```
python is /Users/jakevdp/anaconda/envs/python3.6/bin/python
```

The Python executable being used in the notebook can be determined using

```
In [20]: sys.executable
```

```
Out[20]: '/Users/jakevdp/anaconda/bin/python'
```

In my current notebook environment, the two differ. This is why a simple `!pip install` or `!conda install` does not work: the commands install packages in the `site-packages` of the wrong Python installation.

As noted above, we can get around this by explicitly identifying where we want packages to be installed.

For **conda**, you can set the prefix manually in the shell command:

```
$ conda install --yes --prefix /Users/jakevdp/anaconda numpy
```

or, to automatically use the correct prefix (using syntax available in the notebook)

```
!conda install --yes --prefix {sys.prefix} numpy
```

For **pip**, you can specify the Python executable explicitly:

```
$ /Users/jakevdp/anaconda/bin/python -m pip install numpy
```

or, to automatically use the correct executable (again using notebook shell syntax)

```
!{sys.executable} -m pip install numpy
```

Remember: you need your *installation command* to match the *current python kernel* if you want installed packages to be available in the notebook.

Some Modest Proposals

So, in summary, the reason that installation of packages in the Jupyter notebook is fraught with difficulty is fundamentally that **Jupyter's shell environment and Python kernel are mismatched**, and that means that you have to do more than simply `pip install` or `conda install` to make things work. The exception is the special case where you run `jupyter notebook` from the same Python environment to which your kernel points; in that case the simple installation approach should work.

But that leaves us in an undesireable place, as it increases the learning curve for novice users who may want to do something they (rightly) presume should be simple: install a package and then use it. So what can we as a community do to smooth-out this issue?

I have a few ideas, some of which might even be useful:

Potential Changes to Jupyter

As I mentioned, the fundamental issue is a mismatch between Jupyter's shell environment and compute kernel. So, could we massage kernel specifications such that they force the two to match?

Perhaps: for example, this github issue (https://github.com/jupyterhub/jupyterhub/issues/847) shows an approach to modifying shell variables as part of kernel startup.

Basically, in your kernel directory, you can add a script `kernel-startup.sh` that looks something like this (and make sure you change the permissions so that it's executable):

```
#!/usr/bin/env bash

# activate anaconda env
source activate myenv

# this is the critical part, and should be at the end of your script:
exec python -m ipykernel $@
```

Then in your `kernel.json` file, modify the `argv` field to look like this:

```
"argv": [
   "/path/to/kernel-startup.sh",
   "-f",
   "{connection_file}"
 ]
```

Once you do this, switching to the `myenv` kernel will automatically activate the `myenv` conda environment, which changes your `$CONDA_PREFIX`, `$PATH` and other system variables such that `!conda install XXX` and `!pip install XXX` will work correctly. A similar approach could work for virtualenvs or other Python environments.

There is one tricky issue here: this approach will fail if your `myenv` environment does not have the `ipykernel` package installed, and probably also requires it to have a jupyter version compatible with that used to launch the notebook. So it's not a full solution to the problem by any means, but if

Python kernels could be designed to do this sort of shell initialization by default, it would be far less confusing to users: `!pip install` and `!conda install` would simply work.

Potential Changes to pip

One source of installation confusion, even outside of Jupyter, is the fact that, depending on the nature of your system's aliases and `$PATH` variable, `pip` and `python` might point to different paths. In this case `pip install` will install packages to a path inaccessible to the `python` executable. For this reason, it is safer to use `python -m pip install`, which explicitly specifies the desired Python version (explicit is better than implicit (https://www.python.org/dev/peps/pep-0020/), after all).

This is one reason that `pip install` no longer appears in Python's docs (https://docs.python.org/3/installing/index.html#basic-usage), and experienced Python educators like David Beazley never teach bare pip (https://twitter.com/dabeaz/status/922859605247643649). CPython developer Nick Coghlan has even indicated (https://twitter.com/ncoghlan_dev/status/922979220711661568) that the `pip` executable may someday be deprecated in favor of `python -m pip`. Even though it's more verbose, I think forcing users to be explicit would be a useful change, particularly as the use of virtualenvs and conda envs becomes more common.

Changes to Conda

I can think of a couple modifications to conda's API that may be helpful to users

Explicit invocation

For symmetry with `pip`, it would be nice if `python -m conda install` could be expected to work in the same way the `pip` counterpart does. You can call `conda` this way in the root environment, but the conda Python package (as opposed to the conda executable) cannot currently be installed anywhere but the root environment:

```
(myenv) jakevdp$ conda install conda
Fetching package metadata ...........

InstallError: Error: 'conda' can only be installed into the root environment
```

I suspect that allowing `python -m conda install` in all conda environments would require a fairly significant redesign of conda's installation model, so it may not be worth the change just for symmetry with `pip` 's API. That said, such a symmetry would certainly be a help to users.

A pip channel for conda?

Another useful change conda could make would be to add a channel that essentially mirrors the Python Package Index (https://pypi.python.org/pypi), so that when you do `conda install some-package` it will automatically draw from packages available to `pip` as well.

I don't have a deep enough knowledge of conda's architecture to know how easy such a feature would be to implement, but I *do* have loads of experiences helping newcomers to Python and/or conda: I can say with certainty that such a feature would go a long way toward softening their learning curve.

New Jupyter Magic Functions

Even if the above changes to the stack are not possible or desirable, we could simplify the user experience somewhat by introducing `%pip` and `%conda` magic functions within the Jupyter notebook that detect the current kernel and make certain packages are installed in the correct location.

pip magic

For example, here's how you can define a `%pip` magic function that works in the current kernel:

In [21]:
```
from IPython.core.magic import register_line_magic

@register_line_magic
def pip(args):
    """Use pip from the current kernel"""
    from pip import main
    main(args.split())
```

Running it as follows will install packages in the expected location

In [22]:
```
%pip install numpy
```

```
Requirement already satisfied: numpy in /Users/jakevdp/anaconda/lib/p
```

Note that Jupyter developer Matthias Bussonnier has published essentially this in his pip_magic

(https://github.com/Carreau/pip_magic) repository, so you can do

```
$ python -m pip install pip_magic
```

and use this right now (that is, assuming you install `pip_magic` in the right place!)

conda magic

Similarly, we can define a conda magic that will do the right thing if you type `%conda install XXX`. This is a bit more involved than the `pip` magic, because it must first confirm that the environment is conda-compatible, and then (related to the lack of `python -m conda install`) must call a subprocess to execute the appropriate shell command:

In [23]:
```
from IPython.core.magic import register_line_magic
import sys
import os
from subprocess import Popen, PIPE


def is_conda_environment():
    """Return True if the current Python executable is in a conda env"
    # TODO: make this work with Conda.exe in Windows
    conda_exec = os.path.join(os.path.dirname(sys.executable), 'conda'
    conda_history = os.path.join(sys.prefix, 'conda-meta', 'history')
    return os.path.exists(conda_exec) and os.path.exists(conda_history


@register_line_magic
def conda(args):
    """Use conda from the current kernel"""
    # TODO: make this work with Conda.exe in Windows
    # TODO: fix string encoding to work with Python 2
    if not is_conda_environment():
        raise ValueError("The python kernel does not appear to be a co
                         "Please use ``%pip install`` instead.")

    conda_executable = os.path.join(os.path.dirname(sys.executable), '
    args = [conda_executable] + args.split()

    # Add --prefix to point conda installation to the current environm
    if args[1] in ['install', 'update', 'upgrade', 'remove', 'uninstal
        if '-p' not in args and '--prefix' not in args:
            args.insert(2, '--prefix')
            args.insert(3, sys.prefix)

    # Because the notebook does not allow us to respond "yes" during t
    # installation, we need to insert --yes in the argument list for s
    if args[1] in ['install', 'update', 'upgrade', 'remove', 'uninstal
        if '-y' not in args and '--yes' not in args:
            args.insert(2, '--yes')

    # Call conda from command line with subprocess & send results to s
    with Popen(args, stdout=PIPE, stderr=PIPE) as process:
        # Read stdout character by character, as it includes real-time
        for c in iter(lambda: process.stdout.read(1), b''):
            sys.stdout.write(c.decode(sys.stdout.encoding))
        # Read stderr line by line, because real-time does not matter
        for line in iter(process.stderr.readline, b''):
            sys.stderr.write(line.decode(sys.stderr.encoding))
```

You can now use `%conda install` and it will install packages to the correct environment:

```
In [24]: %conda install numpy
```

```
Fetching package metadata ..........
Solving package specifications: .

# All requested packages already installed.
# packages in environment at /Users/jakevdp/anaconda:
#
numpy                     1.13.3           py36h2cdce51_0
```

This conda magic still needs some work to be a general solution (cf. the TODO comments in the code), but I think this is a useful start.

If a pip magic and conda magic similar to the above were added to Jupyter's default set of magic commands, I think it could go a long way toward solving the common problems that users have when trying to install Python packages for use with Jupyter notebooks. This approach is not without its own dangers, though: these magics are yet another layer of abstraction that, like all abstractions, will inevitably leak. But if they are implemented carefully, I think it would lead to a much nicer overall user experience.

Summary

In this post, I tried to answer once and for all the perennial question, *how do I install Python packages in the Jupyter notebook*.

After proposing some simple solutions that can be used today, I went into a detailed explanation of *why* these solutions are necessary: it comes down to the fact that in Jupyter, the kernel is disconnected from the shell. The kernel environment can be changed at runtime, while the shell environment is determined when the notebook is launched. The fact that a full explanation took so many words and touched so many concepts, I think, indicates a real usability issue for the Jupyter ecosystem, and so I proposed a few possible avenues that the community might adopt to try to streamline the experience for users.

One final addendum: I have a huge amount of respect and appreciation for the developers of Jupyter, conda, pip, and related tools that form the foundations of the Python data science ecosystem. I'm fairly certain those developers have already considered these issues and weighed some of these potential fixes – if any of you are reading this, please feel free to comment and set me straight on anything I've overlooked! And, finally, thanks for all that you do for the open source community.

Thanks to Andy Mueller, Craig Citro, and Matthias Bussonnier for helpful comments on an early draft

of this post.

This post was written within a Jupyter notebook; you can view a static version here (http://nbviewer.jupyter.org/url/jakevdp.github.com/downloads/notebooks/JupyterInstallation.ipynb) or download the full notebook here (http://jakevdp.github.com/downloads/notebooks/JupyterInstallation.ipynb).

jupyter (http://jakevdp.github.io/tag/jupyter.html) conda (http://jakevdp.github.io/tag/conda.html)

pip (http://jakevdp.github.io/tag/pip.html)

Comments

ALSO ON **PYTHONIC PERAMBULATIONS**

Frequentism and Bayesianism V: ...

5 years ago • 26 comments

Model Fitting vs Model Selection¶ The difference between model fitting and …

The Waiting Time Paradox, or, Why Is ...

2 years ago • 14 comments

Image Source: Wikipedia License CC-BY-SA 3.0 If you, like me, frequently …

On Frequentism and Fried Chicken

6 years ago • 13 comments

My recent series of posts on Frequentism and Bayesianism have drawn …

Expl... in Py...

3 year...

I foun... norma... tweet...