

# Python\_Env\_Conda\_Pip\_Jupyter

April 17, 2020

## 1 Python - Environments, Conda, Pip, Jupyter, aaaaah!

Dennis Bakhuis - 17th April 2020 <https://linkedin.com/in/dennisbakhuis/>

<https://github.com/dennisbakhuis>

### 1.0.1 Contents

1. Python is great, but...
2. One way to organize Python, Environments, and packages
  1. Drop the gui and use a shell
  2. Installing the conda package manager
3. What a typical workflow can be
  1. Creating Python environments for project and tasks
  2. Installing packages with pip
  3. Removing environments and other commands.
4. Round up

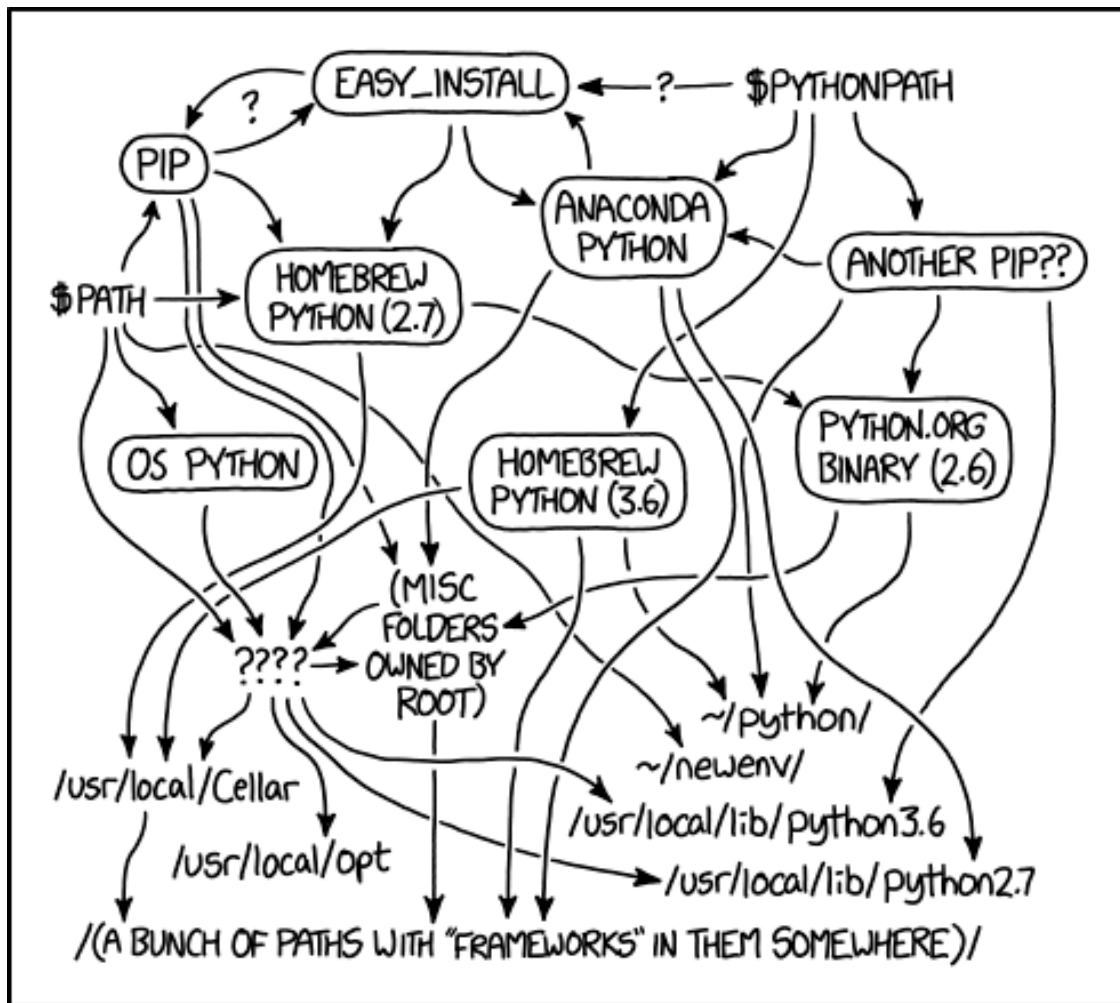
### 1.1 1) Python is great, but...

As many others, I love Python as you very rapidly translate your ideas in very readable code solutions. One reason why it is so successful is the very active community in which amazing people share their awesome solutions. This is why you do not have to write Data structures from scratch, but you simply import Pandas. Write the data to an hdf5 file format? Import h5py! Plot some figure, xkcd style? Import matplotlib! Even better, there are multiple *flavors* so if you prefer to Plot your data in a different way, import one of the various other plotting system, e.g. Plotly, Bokeh, ggplot, to name a few. All this sharing goodness makes that Python is quite popular in many rapid evolving fields, such as Machine Learning.

Unfortunately, all effort from the large community comes at a price. The packages you used get updated, restructured, improved, or just rewritten, because the authors came up with a better way to solve their problem. These changes can be *breaking* changes for the code you have written. Popular packages, such as Numpy or Matplotlib are very reliable, and chances that you get breaking changes are slim. However, using packages that are not as popular, breaking changing can happen, especially when upgrading the package or the version of Python itself.

A way, the Python community solved this problem is with the use of virtual environments. These create isolated Python installations with their own set of packages. It is good practice to have a unique environment for each project or task. This ensures that dependencies of one project will not create breaking changes for another. This solution works great, but also creates some bookkeeping

as you have different Python installations, for which each might come with their own package manager Pip. All of these references pointing Python and/or Pip can create a mess quite quickly and we end up in the famous diagram van XKCD(<https://xkcd.com/1987/>) below:



MY PYTHON ENVIRONMENT HAS BECOME SO DEGRADED  
THAT MY LAPTOP HAS BEEN DECLARED A SUPERFUND SITE.

While all XKCDs are funny they generally contain some truth and indeed, if you do not have some sort of system to do the bookkeeping, your Python installations can become a mess. As with all things in Python, there are many different ways to organize this, including some great tools such as Poetry, Pipenv, and many more. In the next section I will describe the system I am using. This works for me and this might also work for you. These are just my two cents, and you should do whatever suits you.

[ ]:

[ ]:

## 1.2 2) One way to organize Python, Environments, and packages

While there are many ways to organize your Python versions, virtual environments, and packages I use the following: 1. conda for Python and virtual environments 2. pip for package management inside the virtual environments

### 1.2.1 2a) Drop the gui and use a shell

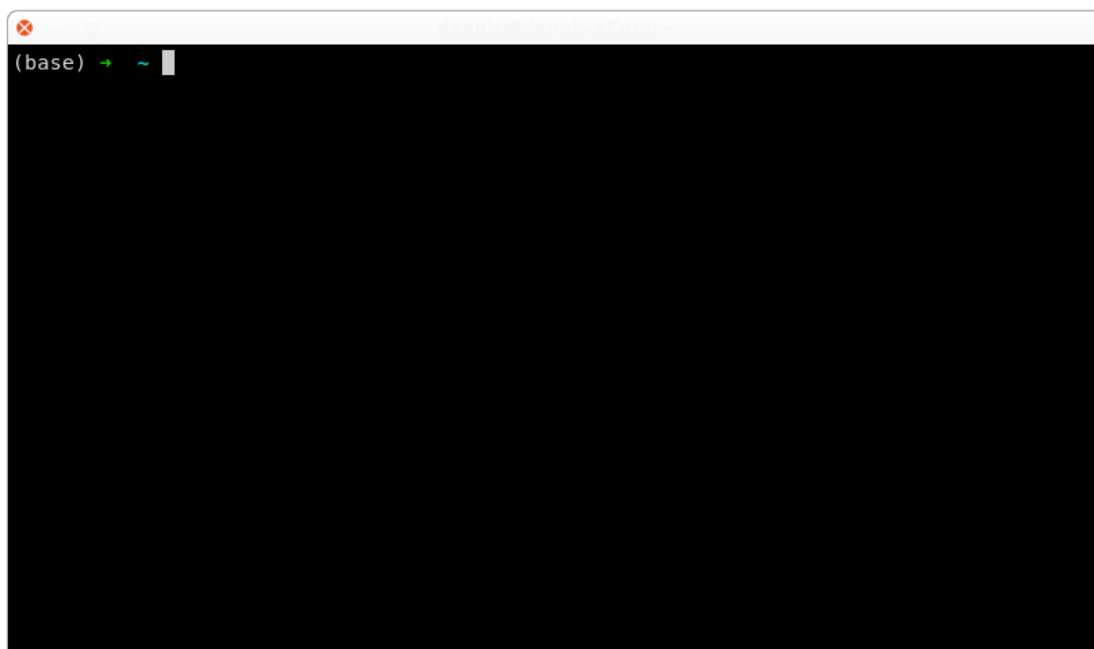
There are probably very nice and flexible graphical user interfaces (GUIs) out there, however I prefer the command line interace (CLI) over them. They give me the impression that I am more in control. I am not sure if that is true, but at least I get confirmation over each step that I am doing. There are many different shells to choose from and it really does not matter that much. Use the one you have available is generally the easiest. When you spend more time in the CLI, you might get pickier one day and choose a different *flavor*.

**Windows** has two different CLIs installed by default, the Command Line Prompt (CMD) and Windows Powershell. Both are fine, but the power shell gives more of an shell feeling. If you have never used a CLI, it can be useful to watch a tutorial of the power shell on youtube. While I have not used Windows for a while, there are other options including running the famous linux bash shell on windows.

**MacOs** has by default bash (MacOs Catalina has Zsh) which is great. You can access the shell using the Terminal application which is a way to interact with bash or zsh. I personally prefer Zsh as a shell and iTerm2 to interact with it. Both can be installed using home-brew. If you are not familiar with bash, I highly advice to watch a tutorial on it, as it is incredibly useful.

**Linux** users are probably already familiar with a shell. Which shell and terminal application is installed, depends on the distribution you have installed. Generally all are fine, use what you have available.

While your shell will most likely look quite different, here is an image of my shell:



### 1.2.2 2B) Installing the conda package manager

Now that we are familiar with a shell, let's get the next requirement: a conda distribution. Conda is a popular package manager for Python (and many other languages) and gives you access practically all Python versions and packages. It includes a easy system to manage virtual environments. While conda can be used to install packages, I only use it for virtual environments and Python versions. Conda has a feature called dependency checking, which works quite well, but sometimes can be a bit slow. Also, some packages are newer using Pip and therefore I chose to only use pip. Mixing both will probably work but it is probably better to just use a single tool.

Often, conda is installed using Anaconda, which is a full fledged distribution, including many packages, tools and a GUI. It installs many packages you will probably never use and I find the GUI slow to work with. The other option to install conda is to use miniconda, another distribution which is much smaller as the name suggests. Miniconda is a base installation with a Python system, Pip, Conda, and some other tools. While it is straight forward to install, here are some simple guide lines: - Install in your *home directory* **if and only if** you do *not* have spaces in the full path. For example, if you have a username using a space, e.g. "dennis bakhuis", your home directory path will also contain a space (/home/dennis bakhuis/). This can cause problems with some packages as not all imports use quotes around paths, which is required for spaces folder names. If you happen to have a space in the folder, install miniconda in a different location. For example in Windows, just use the root directory: 'C:/miniconda3' - After installing miniconda, you should have the conda command available in your shell. To test this, open your shell and type "conda -version". If the command is not found, the path to miniconda has to be added to the global path. - For powershell users, there is an additional step which can be typed in the powershell: "conda init powershell"

This is all there is in installing the tools required to work with Python in virtual environments. In the next section I will explain a typical workflow.

[ ]:

[ ]:

## 1.3 3) What a typical workflow can be

### 1.3.1 3A) Creating Python environments for project and tasks

With the previous steps, you are all set up. If you now open a shell, you have your new Python, Conda, and Pip ready for your bidding. You could install packages straight into your 'base' environment, however I highly advise against this. If you, for whatever reason made a mess of your 'base' environment, there is no way to delete it. Options are a reinstall (which actually is not all that bad) or removing packages by hand. Maybe there are tricks, but much easier is to just create virtual environments.

I would create an environment for each project or task, just to keep things separate. As mentioned before, I create an environment using conda:

```
conda create --name tutorial python=3.7
```

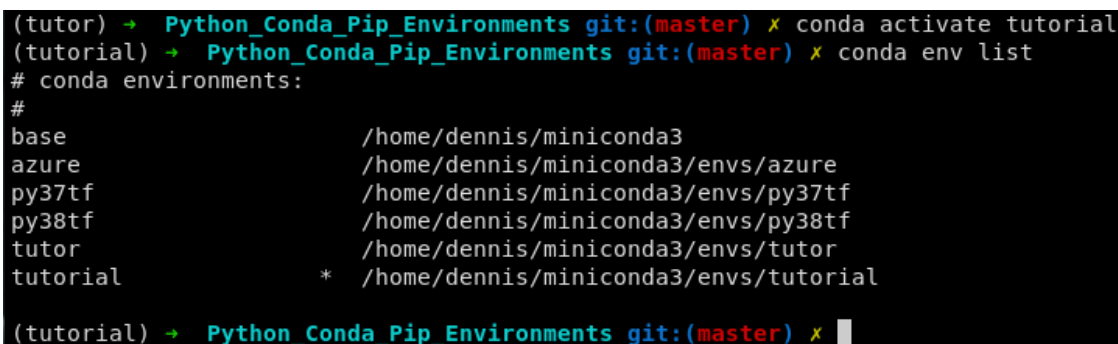
This will create a new environment with the name 'tutorial' with Python version 3.7.x. Because we used a single '=', we tell conda to use the latest version in the Python 3.7 tree. At the moment this is version 3.7.7. If we would have used two equal signs '==' we would tell conda to give exactly version 3.7, so there is a subtle difference.

After the environment is created, we have to switch to the newly created environment. For this conda has the activate command:

```
conda activate tutorial
```

Now you are in the isolated python environment called ‘tutorial’, which has its own version of Python, Conda, and pip. It is possible that you are not completely sure how the environment was named. To check the available environments, you can use:

```
conda env list
```

A terminal window with a dark background and light-colored text. The prompt is '(tutor) → Python\_Conda\_Pip\_Environments git:(master) ×'. The user enters 'conda activate tutorial' and 'conda env list'. The output shows a list of conda environments: base, azure, py37tf, py38tf, tutor, and tutorial. The 'tutorial' environment is marked with an asterisk, indicating it is the active environment. The paths for each environment are listed in the second column.

```
(tutor) → Python_Conda_Pip_Environments git:(master) × conda activate tutorial
(tutor) → Python_Conda_Pip_Environments git:(master) × conda env list
# conda environments:
#
base                /home/dennis/miniconda3
azure                /home/dennis/miniconda3/envs/azure
py37tf              /home/dennis/miniconda3/envs/py37tf
py38tf              /home/dennis/miniconda3/envs/py38tf
tutor               /home/dennis/miniconda3/envs/tutor
tutorial            * /home/dennis/miniconda3/envs/tutorial
(tutor) → Python_Conda_Pip_Environments git:(master) ×
```

The previous picture shows all my available environments. These are just subdirectories in your miniconda folder. The currently active environment is shown with an asterix. Let’s now install some packages in our fleshly created environment in the next section.

### 1.3.2 3B) Installing packages with pip

In your activated environment, it is dead easy to install packages using the command ‘pip install’. For this example we will install the packages numpy, pandas, jupyterlab, matplotlib. While the dependency checking of pip is not as sophisticated as conda, it does however know that Pandas depends on numpy, and will install the dependency if it is missing. To install the packages type:

```
pip install pandas matplotlib jupyterlab
```

After the install, the packages are ready to go. For example to start jupyter type:  
jupyter lab

Sometimes it happens when you work in a notebook, bot forgot to install that one package. For example, want to use tqdm to have some progress bars. To install that package, open another shell next to the one you are already using, activate the environment, and use pip to install tqdm. The package is immediately ready for use in the notebook you are working in. Or install it directly in you notebook using:

```
[3]: !pip install tqdm
```

```
Collecting tqdm
```

```
  Downloading tqdm-4.45.0-py2.py3-none-any.whl (60 kB)
```

```
|                                     | 60 kB 3.6 MB/s eta 0:00:01
```

```
Installing collected packages: tqdm
```

```
Successfully installed tqdm-4.45.0
```

The ‘!’ is used to execute shell commands. For example to do an *list directory* (ls) we do:

```
[1]: !ls
```

```
assets  Python_Env_Conda_Pip_Jupyter.ipynb
```

So using ‘!pip install tqdm’ we can directly install the package in the current environment. The shell method is in my opinion a bit more explicit as you are absolutely sure in which environment you are installing the package, but both are great!.

Another great way of installing packages is using the requirements.txt file. This is a list generated using the ‘pip freeze’ command and gives exact versions of packages used and is a great way to reproduce environments of other people. To create a requirements.txt yourself, type the following in a shell:

```
pip freeze > requirements.txt
```

Of course you can also invoke this command from jupyter itself:

```
[2]: !pip freeze > requirements.txt
```

If you open the file, or run it without the ‘> requirements.txt’ you will see a list of all packages in your environment and the exact version behind the double equal sign. If you provide this file in the root folder of your project or git-repo, others can install all required packages with a single command:

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

As you can see, requirements.txt is a nice way to reproduce environments.

### 1.3.3 3C) Removing environments and other commands.

After a while, you will be collecting quite some environments, which can introduce some clutter to your system. To remove environments that are not needed anymore, we can simply delete them. If you would require it again, using the requirements.txt it is dead easy to recreate it.

Before we can remove an environment, we have to deactivate it. To do that type:

```
conda deactivate
```

Now we can delete an environment by typing:

```
conda env remove --name tutorial
```

To verify, the environment is indeed gone:

```
conda env list
```

#### **Some commands that might be useful:**

Clone an existing environment:

```
conda create --clone tutorial --name tutorial2
```

Search for available packages:

```
pip search tensorflow
```

While I never use it, some might like an ide like Spyder. To use it for a project, you need to install it in the environment you want to use it in, just as jupyterlab. For example, if you had the environment tutorial:

```
pip install spyder
```

```
spyder
```

[ ]:

[ ]:

#### 1.4 4) Round up

This was it for managing Pythons, ENVs, PIPs, and all the snakes. As I mentioned before, this is one way to do it and there are many others. The other ways might work better, but this works for me. Feel free to comment on how this process can be done better or what works for you.

So long and thanks for all the fish!

[ ]: