Applied Math 205: Configuring Python

Michael S. Emanuel

08-Sep-2021

The Challenge of Software Dependencies

One of the joys of using Python is the ease of importing libraries. Advanced linear algebra and numeric functions are available after just one short statement: `import numpy as np`. The dream can quickly become a nightmare if you are working on multiple projects that require different versions of software packages. This problem is so widespread it has been dubbed dependency hell. Wikpedia says "Dependency hell is a colloquial term for the frustration of some software users who have installed software packages which have dependencies on specific versions of other software packages. The dependency issue arises when several packages have dependencies on the same *shared* packages or libraries, but they depend on different or incompatible versiosn of the shared packages."

Clashing dependencies between different libraries can be a particularly acute problem for Python development. The sprawling community of Python users who have created packages to solve diverse problems is one of the most attractive features of the language and has fueled its growth. It has also led to a proliferation of popular packages from small, independent software creators. Many of these packages have long dependency chains. The problem becomes most severe if you are working with machine learning libraries. Many of these depend on Nvidia Cuda to get close to the latest GPU hardware. This can lead to a situation where installing a new Nvidia video driver borks your entire Python installation![1]

Understanding Python Distributions

Python is first and foremost a language standard. If you want to run a Python program, you need to install a Python interpreter on your computer. The bare interpreter itself is of limited use without libraries. A **Python distribution** includes

- A Python interpreter that meets the language standard (e.g. Python 3.9)
- A set of core libraries such as `numpy`
- A package management system that allows users to download additional packages

There are a number of high quality Python distributions available including the "official" CPython distribution from the Python Software Foundation (PSF), the Anaconda distribution from Continuum Analytics, and the ActiveState ActivePython distribution.

The course staff for AM 205 recommends that students new to Python start with the Anaconda distribution. This is in no way a requirement for the course. In fact, it is not required to use Python at all for the course. We recommend Anaconda because it includes a high quality and popular package management system with facilities to quickly create environments (more on this below). This combination of features makes it easy to get up and running quickly and isolate dependencies across packages that can create nasty bugs. Anaconda has also become the most popular Python distribution in data science / machine learning community. The large number of users maximizes the probability that someone has made sure that the package you want to install will "just work" when you run `conda install` on it in a clean Python environment.

The Anaconda Python distribution is supported on all major computing platforms including Windows, Mac and Linux. Anaconda offers a free individual license. Harvard students using Python for their coursework qualify under the terms of this license. The download page can be found at Anaconda Individual Edition.

[1] Yes, this has happened to me before.

Understanding Python Virtual Environments

A Python **virtual environment** is a facility for isolating a set of dependencies that control the behavior of the Python interpreter. These dependencies include

- The executable Python interpreter that runs, e.g. `/usr/bin/python`
- The directories used to store and retrieve site packages
- The settings of other environment variables such as the path

A virtual environment keeps a record of all of these settings and makes it easy to create, activate, deactivate, and manage your environments.

It's easiest to learn about Python environments by example. Assuming you've installed the Anaconda Python distribution on your computer successfully, you should be able to run the program `conda` from a terminal (command line). All Anaconda distributions need one special environment called `base`. Anaconda uses this environment to update itself, so I highly recommend that you install the absolute minimum number of packages into it. If it gets corrupted you will probably end up needing to reinstall it completely.[2]

You can test this environment by issuing the terminal command `conda activate base`. After I run this command, my terminal prompt changes from `michael@Michael-PC$` to `(base) michael@Michael-PC$`. More interestingly, when I run `which python`, the result changes; it now shows `/home/michael/anaconda3/bin/python`. If I type `python` at the terminal, I go into an interactive session for the Anaconda version. Two of the only packages I suggest adding to your base environment facilitate exposing the kernels from different Python environments to Jupyter notebooks. You can install these packages by running two commands (in the base environment) `conda install ipykernel` and `conda install nb_conda_kernels`.

Once you have set up your base environment, you can set up one conda environment for each suitable project you work on. Determining how to map work projects to virtual environments is more art than science. A good starting point for students is to create one environment for each school course that involves programming. For instance, I have an environment just for AM-205. I created it by running: `conda create -n am205` Whenever I use Python for AM-205, I start my session by typing `conda activate am205`. If I try to use a library that is not installed, I install it just in that environment. If I run `python` in my new, empty environment, I get an interactive session. I can try to run `>>> import numpy as np`. Oops-I get an error message: `ModuleNotFoundError: No module named 'numpy'`. OK, I know how to fix this. Get out of the interactive session with `exit()`. Then at the terminal, in the `am205` environment, run the command `(am205) $ conda install numpy`. After the installation, the `import numpy` command works successfully and I'm ready to program.

Closing Thoughts

Configuring a Python installation and managing virtual environments is a large and complex task. There are people who do professional systems administartion and spend a lot of their careers getting it right. As students and scholars, we don't need to function at that level. We can benefit by adopting a few best practices, including researching our choice of Python distribution and using virtual environments to manage dependencies. Students who are interested in learning more of these skills at Harvard can consider taking CS 107 (formerly CS 207).

[2] Yes, this has happened to me, more than once. Please learn from my pain, which was acute.