PLURALSIGHT

Managing Python Environments

By John Walk - January 10, 2020
21 minutes - 4312 words

Python is, in general, an exceptionally flexible programming language - and this extends to its environment management. Unfortunately, that can mean that it's exceptionally easy for your Python install(s) to become a messy, convoluted trash fire. A whole host of environment management tools exist to tame this mess... but this can end up being even more convoluted, especially for a new Python developer!

In this post, we'll go through the pros and cons of available tools so you can make an informed decision about your setup. Realistically, we just need to agree on a few principles:

- **virtualization is your friend:** isolating your Python environment per-project makes your life infinitely easier by avoiding dependency clashes between projects
- **projects should be reproducible:** the more tightly you can specify dependencies, the easier it is to exactly reproduce the running environment for your code on the fly, for yourself or another dev
- **self-contained = deployable**: the easier it is to pack up and ship an environment with all the trimmings, the easier it is to

get projects running on radically different systems (like moving from a dev environment to deployment)

Any of our Python virtualization solutions can satisfy these - though like any other tool, you'll find *very* strong opinions as to which is best! Our goal here is for you to be able to make an informed decision about your Python environment and save yourself a few headaches on the way. We'll begin with the simplest low-level standard tooling, and work our way through newer, more powerful (though sometimes more esoteric and restrictive) options, discussing pros and cons along the way - I encourage you to consider all the options that could suit your project's needs.

whose environment is it, anyway?

First off, newer Python coders might be wondering - what are we talking about when we say *environment* control? Broadly, there's three levels of control we want to be cognizant of:

- installed python packages: what can we `import` into our running Python instance?
- the installed python binary itself: what version and flavor of Python are we actually running?
- non-python, system-level dependencies: what about things like the C/C++ toolchain underlying our numeric packages?

Being able to explicitly control any (or all) of these things for a project makes it that much easier to pack up the project and get it up and running in a fresh environment. At the simplest level, most virtualization tools in Python will create a *virtual environment* that looks something like below:

```
venv/
|-- bin/
|   |-- python
|   |-- pip
|   |-- activate
|   |-- <other binaries/CLI tools>
|-- lib/
|   |-- python3.7/
|   |   |-- site-packages/
|   |   |   |-- <pip-installed packages>
|-- include/
|   |-- <underlying C header files and such>
|-- <config files>
```

A directory like this contains everything a Python instance needs to function - binaries and callable scripts (like an environment-specific `python` executable) in `bin`, any installed Python packages in `lib`, and any additional non-python headers and configurations needed. By setting system search paths to target this directory (usually accomplished with an included activation script), we can run a completely self-contained version of Python, with its executables, installed packages, etc. never referencing Python tooling outside of this directory. In addition to enabling project independence, this is a great solution for situations like running a Python environment on a system without `root` privileges, as we can just create the virtualenv in a location with permissions available to the user.

Since the virtual environment includes its own version of the `pip` package manager, installing packages to the environment is trivial - we simply `pip install` like normal with the environment active, and the packages will end up in the right place. This also lets us

specify dependencies for the environment - we can recreate an environment using only a known Python version (since we'd want to use the same executable) and a `requirements.txt` file listing the required packages and their versions, which can be passed directly to `pip install`.

Since environment activation is controlled by system path variables, we can technically put this directory anywhere - it's not necessary to have environment packages co-located with the project as it is with Node.js projects for example. It can be beneficial to keep the virtual environment with its associated project code (provided it's excluded from things like `git` tracking, as we only need to actually keep track of the Python version and requirements file), but it's equally workable to use a centralized directory for all of your Python projects.

So, without further ado, let's see what our options are for environment management!

built-in: `venv`

Since version 3.3, the Python standard library has shipped with a simple built-in tool, `venv`, for creating virtual environments. Simply invoking

```
$ python -m venv $VENV_PATH
```

(substituting your venv path as desired) will create a virtual environment like the above in the specified directory, along with a launch script - the environment can be activated or deactivated by calling

```
$ source $VENV_PATH/bin/activate
$ deactivate
```

Once activated, `pip install` (individual packages or from a requirements file) will work as expected. To pack up a virtual environment to be reproduced elsewhere, you just need to generate a requirements file with the environment's contents:

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

with the environment active will generate a requirements file that can be installed into a fresh virtual environment on another system.

Pros:

- comes stock in with Python, no additional tooling required
- creates a standard virtual environment that plays with pretty much any tooling: `requirements.txt` works for any environment manager using `pip`

Cons:

- Only aware of installed packages: creates an environment with whatever Python was invoked to create it, so you're still stuck managing the Python version manually
- no bells and whistles apart from what's `pip`-installable into the environment

`venv` with more: `virtualenv`

There's actually an older (dating back to Python 2.x) tool, `virtualenv`, for creating these environments. In fact, `venv` was

created by bringing a subset of `virtualenv` functionality into the Python 3.3+ standard library. This is still supported and installable via `pip` - though users should take note that this will install for the currently active Python only (defaulting to the system install). To avoid clashing with system packages (and for cases where the user lacks the privileges needed to install to the system Python), this can be installed on a per-user basis with `pip install --user virtualenv`. Once installed, invocation is similar to `venv`:

```
% virtualenv -p $PYTHON_CALLABLE $VENV_PATH
```

creating a virtual environment directory at the specified location, that can be activated/deactivated just like one from `venv`. Notably, we have the option of supplying a Python callable - whereas `venv` creates an environment for the Python used to call it, `virtualenv` can create an environment for any Python install available on the system, meaning we can just run one tool from the system Python to create environments for separately managed Python installs (if the `-p` option is omitted, it will default to using the current active Python version). Once created, `pip install` by name or by requirements file works as expected.

Pros:

- creates the same standard virtual enviroment that, like `venv`, plays nicely with most tooling
- can create environments for any installed Python with the same call
- includes some advanced functionality, like the ability to create bootstrap scripts for the environment

Cons:

- installed Python versions still need to be managed manually
- requires managing a package install to the system Python

extending to install management: `pyenv` and `pyenv-virtualenv`

Both of the above solutions only address package management - in either case, the user is left managing the installed Python version(s) manually. Fortunately, there is the excellent `pyenv` utility to address this, installable either through `homebrew` on OSX or by direct `git` checkout and build.

Once set up, new Python versions can be installed easily by

```
$ pyenv install $PYTHON_VERSION_OR_DEFINITION_FILE
```

The currently active version or a list of all installs can be shown by

```
$ pyenv version
$ pyenv versions
```

User-level defaults or project-directory specific Python versions can be set by

```
$ pyenv global <desired default version>
$ pyenv local <desired version for current working directory>
```

On its own, `pyenv` only controls installed Python versions, not virtual environments. Of course, within a `pyenv`-controlled Python version we could easily use `venv` or `virtualenv` to build the virtual environment - but the `pyenv` developers have also rolled out a

plugin, `pyenv-virtualenv`, for managing environments with `pyenv`-installed Python versions. After installing via `homebrew` or `git` checkout+build, running

```
$ pyenv virtualenv $PYTHON_VERSION $VENV_NAME
```

will create a virtual environment using the specified `pyenv`-managed Python version with the given name (alternately, the Python version can be omitted to use the current default). The virtual environment can be activated or deactivated by

```
$ pyenv activate $VENV_NAME
$ pyenv deactivate
```

The install location for virtual environments is managed by `pyenv-virtualenv`, so we don't need to worry about specifying a directory (in our project, or in a central location) for the environment. It is, however, a normal virtual environment, so `pip` installs, requirements files, etc. all work like we expect. Notably, `pyenv` tracks a Python version for each virtual environment, so we can actually use `pyenv local` to set a specific virtual environment for a project directory, not just a Python version.

Pros:

- no Python dependencies, only depends on shell commands
- one-stop management of all installed Python versions
- quickly set per-user and per-project default Python versions and virtual environments, including auto-switching for project directories
- easily create virtualenvs tied to specific Python installs

Cons:

- OSX/linux only (though a Windows port does exist)
- somewhat convoluted install/setup process

all-in-one: `pipenv`

Up to now, we've been working with multiple tools for Python & environment management, and package installation into those environments - what if we could roll all of that into a single tool? `Pipenv` aims to do just that, by bundling together Python and strong package version control in the style of `npm` or `yarn` for Javascript. After installing via package manager (`homebrew`, `apt`, `dnf`, etc.) or `pip` installing into an existing Python environment (recommended to install as a user-level utility, as with `virtualenv`), we can create a new `pipenv` project in our project directory with

```
$ pipenv --python $PYTHON_VERSION
```

which will initialize the project using the specified Python version (if `pyenv` is installed, it can even install Python versions on-demand). To start with, this creates:

- a `Pipfile` config file at the project home specifying Python version, sources, and any installed packages
- a fresh virtual environment housed in the `pipenv` working directory

We no longer have to manage installs with `pip` and virtual environments separately - `pipenv` takes care of both! To install a package, simply running

```
$ pipenv install $PACKAGE_NAME
```

will both install the package into the virtual environment, and write the package as a dependency into the Pipfile. This Pipfile is then all we need to rebuild the project elsewhere, rather than the `requirements.txt` used by other managers - simply running `pipenv install` on a directory with a Pipfile will recreate the environment. To activate the environment,

```
$ pipenv shell
```

will launch a new shell process using the project's virtual environment.

Next, `pipenv` can do something fairly unique - it fully determines and specifies dependencies for the project. At the minimum, `pip install` just needs a package name to install, e.g. `pip install numpy`. We can, of course, specify version limits, e.g. `numpy==1.18.1`, in `pip install` or requirements files. However, beyond this, `pip` doesn't really do much validation - while pulling required dependencies of the packages we want to install, `pip` can potentially end up pulling clashing versions, so unless we've actually checked that everything installs (like by actually installing everything and then generating the requirements file directly from `pip freeze`) we can run into issues trying to rebuild the environment from the package requirements. Instead, `pipenv` exhaustively builds out the dependency graph, flagging any issues and generating a validated `Pipfile.lock` for fully specifying every dependency in the project. We can trigger this manually for the requirements in our Pipfile with

```
$ pipenv lock
```

to pull the specifically requested packages from the Pipfile and generate the dependency graph for `Pipfile.lock`. While this does produce environments that can be deterministically reproduced, the dependency resolution can be quite complex, so `pipenv` environments are slower to write than using bare `pip`.

Pros:

- officially supported by Python Packaging Authority
- single tool for project, virtual environment, and package management
- plays well with `pyenv` and `conda` for Python & environment types
- validated, deterministic dependencies for every project

Cons:

- incompatible with other management tools, so requires consistent use across projects and users
- dependency resolution is quite slow

`poetry` in motion

Similarly, `poetry` includes environment control and dependency resolution, but is geared more specifically towards Python package development rather than general project control. After installing with the custom installer, we can create a new project with

```
$ poetry init
```

which will run through a series of interactive prompts to fill out a `pyproject.toml` config file specifying your project's dependencies. Alternately,

```
$ poetry new $PACKAGE_NAME
```

will create a directory structure like

```
package-name/
|-- pyproject.toml
|-- README.rst
|-- package_name/
|   |-- __init__.py
|-- tests/
|   |-- __init__.py
|   |-- test_package_name.py
```

Essentially, this has created a skeleton of exactly the structure we'd want for building a Python package, albeit with the configuration TOML file taking the place of the `setup.py` file used by the standard library's packaging tools. We can add project dependencies via

```
$ poetry add $PACKAGE_NAME
```

after which running `poetry install` will install all the specified packages into a fresh virtual environment if needed (if `poetry` is already running in a virtual environment, it will use that instead) while ensuring fully validated dependencies in a `poetry.lock` file. For specifying Python versions for the project, `poetry` can integrate with `pyenv` (or you can specify the environment manually with `poetry env use <path>`). We can use this directory for any code really, but where it shines is in Python package building - `poetry build` and `poetry publish` will assemble Python `sdist` and `wheel` package distributions and publish them to your repository of choice.

Pros:

- single tool for project, virtual environment, and package management
- validated, deterministic dependencies for every project
- integrated build/publish tooling for Python packages

Cons:

- dependency resolution is quite slow
- requires tool-specific install & updates, rather than using stock package managers
- tooling geared more towards package projects, rather than things like app development (but can still be used for it!)
- can currently only build pure Python wheels, so we can't include things like C/C++ dependencies or `Cython` integration code
- does not integrate with other environment/package managers

a challenger appears: `anaconda`

Historically, Python package management has faced one major issue - while Python packages can require non-python dependencies (e.g., compiled C/C++ underlying nearly all numerical tooling in Python), packages could not meaningfully track these dependencies in a controlled way. Older `sdist` ("source distributions") package distributions could only share source code, and as such required a compatible compiler on the host machine to build the package at all - vagaries between compiler toolchains could introduce errors into package installs. While this was greatly improved by the migration to `wheel` distributions, which can include compiled dependencies like shared-object `.so` files, Python's package trees do not version track non-Python dependencies (so,

for example, they are not really aware of things like version changes in compiled dependencies).

This, and other issues (like the lack of a rigorous dependency resolver in `pip`) motivated the development of Anaconda - an all-in-one Python distribution (though it will behave identically to the usual Python callables you're used to), package & environment manager called `conda`, and a new package distribution format. The whole deal is shipped with a graphical installer, which will also inject instructions into your startup scripts such that the default Python will come from the Anaconda distribution. We can then create new virtual environments with the `conda` manager using

```
$ conda create --name $ENV_NAME
```

and activate/deactivate the environment with

```
$ conda activate $ENV_NAME
$ conda deactivate
```

Within the conda environment, simply running

```
$ conda install $PACKAGE_NAME
```

will pull a package from the conda repository and install it into the environment. To export a `conda` environment or recreate one from the exported file (equivalent to installing from a `requirements.txt` file):

```
$ conda list --export > $REQUIREMENTS_FILE
$ conda create --name $ENV_NAME --file $REQUIREMENTS_FILE
```

This is the biggest distinction between Anaconda/`conda` and the other managers we've discussed here - while everything else is building on `pip` and using the standard `wheel` format for Python packages, `conda` redesigns from the ground up how packaging really works in its environments, and takes a rather different philosophy. In short: `pip` installs *python* dependencies in *any* environment, while `conda` installs *any* dependencies in *conda* environments. Within a `conda` environment, you've got fine-grained control over your dependencies, at the cost of only being functional within the `conda` environment framework - the package manager is inextricably linked to the environment, and relies on a packaging structure and environment specification that is incompatible with other Python tools. In contrast, `pip` is thoroughly general in terms of environment for handling Python dependencies (it's used under the hood even in the more detailed environment managers like `pipenv` and `poetry`), and can install into essentially any environment that runs Python, including `conda` environments.

All this means that Anaconda and its associated tools can be really powerful for getting your local environment up and running, and for fairly painless management of your own projects. As such, it's a common solution for (and is directly marketed to) Data Scientists, who more commonly are running code in their own bespoke environments and have particular needs for clean handling of compiled dependencies in numerical packages. However, it is significantly more difficult to integrate with other systems (unless they are also all running `conda`), particularly for dealing with production deployments.

Pros:

- integrated Python distribution, environment, and package management
- meaningfully handle non-Python dependencies
- includes rigorous dependency resolution similar to `pipenv` and `poetry`
- ships with a ready-made Data Science stack available
- works cross-platform

Cons:

- package management does not integrate with standard package repositories, meaning falling back to `pip` installs may still be necessary
- does not integrate with any other environment manager, no cross-compatibility
- mix-and-match with `conda` and `pip` installs can be hard to replicate
- entirely new toolchain for Python package development

if it works on your machine, we'll ship your machine: `docker`

This is a bit of an oddity for the purposes of this article, but it's so critical for environment management that it bears including. Docker, unlike the other tools here, is not a Python environment manager at all - rather, it is a *container* manager. Each Docker container runs a lightweight environment including all code, runtimes, system tools, and libraries on top of isolated resources. From the developer's standpoint, the container appears to be an entirely independent machine running a Linux environment, without the resource overhead of a full virtual machine - it's

entirely feasible to run a number of containers in parallel on one machine (for example, while building a full-stack app, the developer might simultaneously run separate containers for the frontend, backend, and database instance). This gives us complete control over *everything* in our code environment, right down to low-level system dependencies, and lets create a portable container that can exactly reproduce that environment anywhere running Docker.

Really, learning Docker warrants a post in its own right, but let's quickly run through an example for running some Python code. First, our project directory would look something like this:

```
my-project/
|-- Dockerfile
|-- docker-compose.yml
|-- requirements.txt
|-- project_code/
|   |-- <your Python code goes here>
```

in which we have:

- `Dockerfile`: the instructions for building the Docker container. These assembled instructions constitute a *Docker image*, which can be spun up for any number of independent *container* instances.
- `docker-compose.yml`: instructions and settings for running containers. This is optional, as everything in the YAML can be done with `docker` command-line calls, but it makes our life a lot easier.
- `requirements.txt` and Python code: this is just like your Python setup in any other environment. In fact, we can substitute `requirements.txt` for the environment spec in any other manager,

e.g. we could easily run `pipenv` and include a Pipfile for the container instead.

The `Dockerfile` will look something like

```
FROM python:3.7

WORKDIR /opt/app

COPY ./requirements.txt ./requirements.txt

RUN pip install -r requirements.txt
```

in which we:

- declare a “base image” starting point: in this case, an official Python 3.7 image running on Debian linux
- set the current working directory for subsequent instructions
- copy files into the container so they’re available for subsequent commands
- execute commands for setup: we can follow `RUN` calls with anything that can run in the command line for the operating system of the container

This declares a container with the desired Python version installed, then brings in and installs our desired package dependencies - we could also use `RUN` calls to install any necessary system dependencies (C compilers, `git` calls to bring in source code, etc.). To build and run the container, we could use `docker` command line calls, but it’s generally much simpler to use the `docker-compose` wrapper tool. We specify settings for that in YAML format:

```
version: '3.7'

services:
  python-app:
    build:
      context: ./
    volumes:
      - ./python_code:/opt/app/python_code
    command: python
```

This lets us specify things all sorts of useful information for the container: shorthand build instructions (so `docker-compose` is aware of how to build an image from the `Dockerfile`), volume mounts to share code & data between the container and host machine (so we can live edit our code and rerun it without needing to rebuild the container), port mappings and environment variables for system control, and the command we actually want it to run. To build our container then, we just need to run

```
$ docker-compose build
```

which will assemble an image for any service(s) specified in the compose file. Then running

```
$ docker-compose run
```

will launch a Python shell in our command line, running inside the container (or, for background services, we can use `docker-compose up`).

Finally, we should take a moment to consider which base image to use, as there are a number of options. Technically, we could pull a base image for our linux distro of choice and install Python on it (e.g., with `pyenv`) - but if we don't have specific needs for an OS, we can easily just use the stock `python` images. This gives us a Debian environment with the Python version listed in the tag as its system Python (e.g., `python:3.7.5` will default to using Python 3.7.5) with no clashes, so we can go ahead and run that like normal. Alternately, we can use the "slim" images (e.g., `python:3.7.5-slim`), which strip out some potentially unnecessary system dependencies. We may need to add these back in - for example, certain packages with compiled dependencies will need `gcc` for a C compiler - but it generally results in a significantly smaller image compared to the "full-fat" version, and dependencies can easily be re-installed in `Dockerfile` instructions.

Devs coming from other languages will likely be familiar with the `alpine` image type - this is an extremely stripped-down distro designed for security and size constraints. Alpine python images do exist, but I would strongly caution against their use. Alpine uses the `musl` toolchain for all its C dependencies (again, for size and security), whereas Python packages generally have their dependencies built assuming the `glibc` toolchain for performance reasons. Getting these packages running on alpine requires either rebuilding the `glibc` toolchain, or rebuilding those packages for `musl` - either way, it's a headache for devs and can result in an image of comparable size to the `slim` version, eliminating the advantage of using Alpine in the first place. Unless you have specific deployment needs for Alpine, I'd recommend defaulting to the slim Python. Alternately, if you're already familiar with and prefer to use Anaconda tooling, there are well-supported `anaconda` and `miniconda` (a stripped-down, smaller distro) Docker base images.

Pros:

- complete control over all our dependencies, down to the system level - can even use any of the environment managers in this article
- explicitly specify all instructions to replicate the code environment
- containers can easily be packed up and shipped to run on production environments
- common instructions can be built into new base images and reused between projects
- can programmatically define interactions between multiple services running independently in a single `docker-compose` spec

Cons:

- whole new API to learn
- weird vagaries with build tooling on `alpine` images

wrapping up

Of course, I said that any of these tools are workable for environment management - that's true, but that doesn't mean I don't have my own opinions! Out of all of these, I prefer to use `pyenv`/`pyenv-virtualenv` for control on my local machine, although for most projects I'll go ahead and use Docker (possibly with `venv` if I don't want to run as root) to have the maximum level of control and easiest possible deployment for my code. In nearly all cases, I'll default to using the slim Python images, as this strikes a good balance between image size & removing unnecessary dependencies vs. ensuring I can easily get the tooling that I *do* need.

I've generally found dependency resolution to be less of an issue (particularly for highly isolated projects), and thus found tools like `pipenv` or `poetry` to be less necessary - although it's a great idea for larger projects, and better dependency resolution is a high-level priority for the Python Software Foundation. In the case of a project with significant dependency issues, running `pipenv` on a Docker container is an excellent solution.

The difficulties of dealing with production environments, coupled with the gains for dealing with compiled dependencies provided by the newer `wheel` format for Python packages (which was introduced since the development of Anaconda) means I prefer to avoid `conda` as an environment manager entirely. While it's marketed for data scientists, I find that a code setup that looks more like production, supplemented by consistent build environments provided by Docker, pays off in the long run.

Categories: practices
Tags: python, docker, getting started