**[A non-magical introduction to Pip and Virtualenv for Python beginners](https://www.dabapps.com/blog/introduction-to-pip-and-virtualenv-python/)**

Tagged:

[python](https://www.dabapps.com/blog/tag/python)

[technical](https://www.dabapps.com/blog/tag/technical)

One of the hurdles that new Python developers have to get over is understanding the Python packaging ecosystem. This blog post is based on material covered in our [Python for Programmers training course](https://www.dabapps.com/services/training/python-for-programmers/), which attempts to explain [pip](http://pip-installer.org/) and [virtualenv](http://www.virtualenv.org/) for new Python users.

**Prerequisites**

Python for Programmers is aimed at developers who are already familiar with one or more programming languages, and so we assume a certain amount of technical knowledge. It will help if you're reasonably comfortable with a command line. The examples below use bash, which is the default shell on Macs and most Linux systems. But the commands are simple enough that the concepts should be transferrable to any terminal, such as PowerShell for Windows.

**pip**

Let's dive in. [pip](http://pip-installer.org/) is a tool for installing Python packages from the [Python Package Index](http://pypi.python.org/).

PyPI (which you'll occasionally see referred to as [The Cheeseshop](http://stackoverflow.com/questions/5393986/why-is-pypi-called-the-cheese-shop)) is a repository for open-source third-party Python packages. It's similar to RubyGems in the Ruby world, PHP's Packagist, CPAN for Perl, and NPM for Node.js.

Python actually has another, more primitive, package manager called easy\_install, which is installed automatically when you install Python itself. pip is vastly superior to easy\_install for lots of reasons, and so should generally be used instead. You can use easy\_install to install pip as follows:

$ sudo easy\_install pip

You can then install packages with pip as follows (in this example, we're installing [Django](https://www.djangoproject.com/)):

# DON'T DO THIS

$ sudo pip install django

Here, we're installing Django *globally* on the system. But in most cases, you **shouldn't install packages globally**. Read on to find out why.

**virtualenv**

[virtualenv](http://www.virtualenv.org/) solves a very specific problem: it allows multiple Python projects that have *different* (and often *conflicting*) requirements, to coexist on the same computer.

**What problem does it solve?**

To illustrate this, let's start by pretending virtualenv doesn't exist. Imagine we're going to write a Python program that needs to make HTTP requests to a remote web server. We're going to use the [Requests](http://python-requests.org) library, which is brilliant for that sort of thing. As we saw above, we can use pip to install Requests.

But where on your computer does pip install the packages to? Here's what happens if I try to run pip install requests:

$ pip install requests

Downloading/unpacking requests

Downloading requests-1.1.0.tar.gz (337Kb): 337Kb downloaded

Running setup.py egg\_info for package requests

Installing collected packages: requests

Running setup.py install for requests

error: could not create '/Library/Python/2.7/site-packages/requests': Permission denied

Oops! It looks like pip is trying to install the package into /Library/Python/2.7/site-packages/requests. This is a special directory that Python knows about. Anything that's installed in site-packages can be imported by your programs.

We're seeing the error because /Library/ (on a Mac) is not usually writeable by "ordinary" users. To fix the error, we can run sudo pip install requests (sudo means "run this command as a superuser"). Then everything will work fine:

$ sudo pip install requests

Password:

Downloading/unpacking requests

Running setup.py egg\_info for package requests

Installing collected packages: requests

Running setup.py install for requests

Successfully installed requests

Cleaning up...

This time it worked. We can now type python and try importing our new library:

>>> import requests

>>> requests.get('http://dabapps.com')

<Response [200]>

So, we now know that we can import requests and use it in our program. We go ahead and work feverishly on our new program, using requests (and probably lots of other libraries from PyPI too). The software works brilliantly, we make loads of money, and our clients are so impressed that they ask us to write another program to do something slightly different.

But this time, we find a brand new feature that's been added to requests since we wrote our first program that we *really* need to use in our second program. So we decide to upgrade the requests library to get the new feature:

sudo pip install --upgrade requests

Everything seems fine, but we've unknowingly created a disaster!

Next time we try to run it, we discover that our original program (the one that made us loads of money) has completely stopped working and is raising errors when we try to run it. Why? Because something in the API of the requests library has *changed* between the previous version and the one we just upgraded to. It might only be a small change, but it means our code no longer uses the library correctly. Everything is broken!

Sure, we could fix the code in our first program to use the new version of the requests API, but that takes time and distracts us from our new project. And, of course, a seasoned Python programmer won't just have two projects but *dozens* - and each project might have dozens of dependencies! Keeping them all up-to-date and working with the same versions of *every* library would be a complete nightmare.

**How does virtualenv help?**

virtualenv solves this problem by creating a completely isolated *virtual environment* for each of your programs. An environment is simply a directory that contains a complete copy of everything needed to run a Python program, including a copy of the python binary itself, a copy of the entire Python standard library, a copy of the pip installer, and (crucially) a copy of the site-packages directory mentioned above. When you install a package from PyPI using the *copy* of pip that's created by the virtualenv tool, it will install the package into the site-packages directory *inside* the virtualenv directory. You can then use it in your program just as before.

**How can I install virtualenv?**

If you already have pip, the easiest way is to install it globally sudo pip install virtualenv. Usually pip and virtualenv are the only two packages you *ever* need to install globally, because once you've got both of these you can do all your work inside virtual environments.

In fact, virtualenv *comes with* a copy of pip which gets copied into every new environment you create, so virtualenv is really all you need. You can even install it as a separate standalone package (rather than from PyPI). This might be easier for Windows users. See [virtualenv.org](http://www.virtualenv.org) for instructions.

**How do I create a new virtual environment?**

You only need the virtualenv tool itself when you want to *create* a new environment. This is really simple. Start by changing directory into the root of your project directory, and then use the virtualenv command-line tool to create a new environment:

$ cd ~/code/myproject/

$ virtualenv env

New python executable in env/bin/python

Installing setuptools............done.

Installing pip...............done.

Here, env is just the name of the directory you want to create your virtual environment inside. It's a common convention to call this directory env, and to put it inside your project directory (so, say you keep your code at ~/code/projectname/, the environment will be at ~/code/projectname/env/ - each project gets its own env). But you can call it whatever you like and put it wherever you like!

**Note: if you're using a version control system like git, you shouldn't commit the env directory. Add it to your .gitignore file (or similar).**

**How do I use my shiny new virtual environment?**

If you look inside the env directory you just created, you'll see a few subdirectories:

$ ls env

bin include lib

The one you care about the most is bin. This is where the local copy of the python binary and the pip installer exists. Let's start by using the copy of pip to install requests *into* the virtualenv (rather than globally):

$ env/bin/pip install requests

Downloading/unpacking requests

Downloading requests-1.1.0.tar.gz (337kB): 337kB downloaded

Running setup.py egg\_info for package requests

Installing collected packages: requests

Running setup.py install for requests

Successfully installed requests

Cleaning up...

It worked! Notice that we *didn't* need to use sudo this time, because we're not installing requests globally, we're just installing it inside our home directory.

Now, instead of typing python to get a Python shell, we type env/bin/python, and then...

>>> import requests

>>> requests.get('http://dabapps.com')

<Response [200]>

**But that's a lot of typing!**

virtualenv has one more trick up its sleeve. Instead of typing env/bin/python and env/bin/pip every time, we can run a script to *activate* the environment. This script, which can be executed with source env/bin/activate, simply adjusts a few variables in your shell (temporarily) so that when you type python, you actually get the Python binary *inside* the virtualenv instead of the global one:

$ which python

/usr/bin/python

$ source env/bin/activate

$ which python

/Users/jamie/code/myproject/env/bin/python

So now we can just run pip install requests (instead of env/bin/pip install requests) and pip will install the library into the environment, instead of globally. The adjustments to your shell only last for as long as the terminal is open, so you'll need to remember to rerun source env/bin/activate each time you close and open your terminal window. If you switch to work on a different project (with its own environment) you can run deactivate to stop using one environment, and then source env/bin/activate to activate the other.

Activating and deactivating environments does save a little typing, but it's a bit "magical" and can be confusing. Make your own decision about whether you want to use it.

**Requirements files**

virtualenv and pip make great companions, especially when you use the requirements feature of pip. Each project you work on has its own requirements.txt file, and you can use this to install the dependencies for that project into its virtual environment:

env/bin/pip install -r requirements.txt

See [the pip documentation](http://www.pip-installer.org/en/latest/cookbook.html#requirements-files) for more details.

**Recap**

* pip is a tool for installing packages from the Python Package Index.
* virtualenv is a tool for creating isolated Python environments containing their own copy of python, pip, and their own place to keep libraries installed from PyPI.
* It's designed to allow you to work on multiple projects with different dependencies at the same time on the same machine.
* You can see instructions for installing it at [virtualenv.org](http://www.virtualenv.org).
* After installing it, run virtualenv env to create a new environment inside a directory called env.
* You'll need one of these environments for each of your projects. Make sure you exclude these directories from your version control system.
* To use the versions of python and pip inside the environment, type env/bin/python and env/bin/pip respectively.
* You can "activate" an environment with source env/bin/activate and deactivate one with deactivate. This is entirely optional but might make life a little easier.

pip and virtualenv are *indispensible* tools if you're a regular Python user. Both are fairly simple to understand, and we highly recommend getting to grips with them.