 MAR 28TH, 2016 | COMMENTS

Python Virtual Environments - a Primer

In this article, we'll show how to use virtual environments (<https://docs.python.org/3/library/venv.html#venv-def>) to create and manage separate environments for your Python projects, each using different versions of Python for execution, as well as how Python dependencies are stored and resolved.



Why the need for virtual environments?

Python, like most other modern programming languages, has its own unique way of downloading, storing, and resolving packages (or modules (https://en.wikipedia.org/wiki/Modular_programming)). While this has its advantages, there were some *interesting* decisions made about package storage and resolution, which has lead to some problems – namely how and where packages are stored.

There are a few different locations where these packages can be installed on your system. For example, most system packages are stored in a child directory of the path stored in `sys.prefix` (<https://docs.python.org/3/library/sys.html#sys.prefix>).

On Mac OS X, you can easily find where `sys.prefix` points to using the Python shell:

```
1 >>> import sys
2 >>> sys.prefix
3 '/System/Library/Frameworks/Python.framework/Versions/3.5'
```

More relevant to the topic of this article, third party packages installed using `easy_install` (https://pythonhosted.org/setuptools/easy_install.html) or `pip` ([https://en.wikipedia.org/wiki/Pip_\(package_manager\)](https://en.wikipedia.org/wiki/Pip_(package_manager))) are typically placed in one of the directories pointed to by `site.getsitepackages` (<https://docs.python.org/3/library/site.html#site.getsitepackages>):

```
1 >>> import site
2 >>> site.getsitepackages()
3 [
4     '/System/Library/Frameworks/Python.framework/Versions/3.5/Extras/lib/python',
5     '/Library/Python/3.5/site-packages'
6 ]
```

So, why do all of these little details matter?

It's important to know this because, by default, every project on your system will use these same directories to store and retrieve *site* packages (3rd party libraries). At first glance this may not seem like a big deal, and it isn't really for *system* packages – packages part of the standard Python library – but it does matter for *site* packages.

Consider the following scenario where you have two projects – *ProjectA* and *ProjectB*, both of which have a dependency on the same library, *ProjectC*. The problem becomes apparent when we start requiring different versions of *ProjectC*. Maybe *ProjectA* needs v1.0.0, while *ProjectB* requires the newer v2.0.0, for example.

This is a real problem for Python since it can't differentiate between versions in the “site-packages” directory. So both v1.0.0 and v2.0.0 would reside in the same directory with the same name:

```
1 /System/Library/Frameworks/Python.framework/Versions/3.5/Extras/lib/python/ProjectC
```

And since projects are stored according to just their name there is no differentiation between versions. Thus, both projects, *ProjectA* and *ProjectB*, would be required to use the same version, which is unacceptable in many cases.

This is where the concept of virtual environments (and the `virtualenv` (<https://virtualenv.readthedocs.org/en/latest/>)/`pyenv` (<https://docs.python.org/3/library/venv.html>) tools) comes into play...

What is a virtual environment?

At its core, the main purpose of Python virtual environments is to create an isolated environment for Python projects. This means that each project can have its own dependencies, regardless of what dependencies every other project has.

So, in our little example above, we'd just need to create a separate virtual environment for both *ProjectA* and *ProjectB* and we'd be good to go. Each environment, in turn, would be able to depend on whatever version of *ProjectC* they choose, independent of the other.

The great thing about this is that there are no limits to the number of environments you can have since they're just directories containing a few scripts. Plus, they're easily created using the `virtualenv` or `pyenv` command line tools.

Using virtual environments

To get started, if you're not using Python 3, you'll want to install the `virtualenv` tool with `pip` :

```
1 $ pip install virtualenv
```

If you are using Python 3 then you should already have `pyenv` installed. This is a file that uses the `venv` (<https://docs.python.org/3/library/venv.html>) library underneath.

From here on out we'll assume you're using the newer `pyenv` tool, since there are few differences between it and `virtualenv` with regard to the actual commands. In reality, though, they are very different (<http://stackoverflow.com/questions/29950300/what-is-the-relationship-between-virtualenv-and-pyenv>) tools.

Start by making a new directory to work with:

```
1 $ mkdir python-virtual-environments && cd python-virtual-environments
```

Create a new virtual environment inside the directory:

```
1 $ pyenv env
```

By default this will NOT include any of your existing site packages.

In the above example, this command creates a directory called "env", which contains a directory structure similar to this:

```

1  |— bin
2  |   |— activate
3  |   |— activate.csh
4  |   |— activate.fish
5  |   |— easy_install
6  |   |— easy_install-3.5
7  |   |— pip
8  |   |— pip3
9  |   |— pip3.5
10 |   |— python -> python3.5
11 |   |— python3 -> python3.5
12 |   |— python3.5 -> /Library/Frameworks/Python.framework/Versions/3.5/bin
13 |— /python3.5
14 |— include
15 |— lib
16 |   |— python3.5
17 |       |— site-packages
    |— pyvenv.cfg

```

What does each folder contain?

- “bin” – files that interact with the virtual environment
- “include” – C headers that compile the Python packages
- “lib” – a copy of the Python version along with a “site-packages” folder where each dependency is installed

Further, there are copies of, or symlinks (https://en.wikipedia.org/wiki/Symbolic_link) to, a few different Python tools and to the Python executables themselves. These files are used to ensure all Python code and commands are executed within the context of the current environment, which is how the isolation from the global environment is achieved. We'll explain this in more detail in the next section.

More interestingly are the *activate* scripts in the “bin” directory. These scripts are used to set up your shell to use the environment's Python executable and its site-packages by default.

In order to use this environment's packages/resources in isolation, you need to “activate” it. To do this, just run:

```

1  $ source env/bin/activate
2  (env) $

```

Notice how your prompt is now prefixed with the name of your environment (`env` , in our case). This is the indicator that `env` is currently active, which means the `python` executable will *only* use this environment's packages and settings.

To show the package isolation in action, we can use the `bcrypt` (<https://github.com/pyca/bcrypt/>) module as an example. Let's say we have `bcrypt` installed system-wide, but not in our virtual environment.

Before we test this, we need to go back to the “system” context by executing `deactivate` :

```
1 (env) $ deactivate
2 $
```

Now your shell session is back to normal, and the `python` command refers to the global Python install. Remember to do this whenever you're done using a specific virtual environment.

Now, install `bcrypt` and use it to hash a password:

```
1 $ pip -q install bcrypt
2 $ python -c "import bcrypt; print(bcrypt.hashpw('password'.encode('utf-8'),
3     bcrypt.gensalt()))"
$2b$12$vWa/VsvxxyQ9d.WGgVTdre11515Ctux36LCga8nM5QTW0.4w8TXXi
```

What happens if we try the same command when the virtual environment is activated?

```
1 $ source env/bin/activate
2 (env) $ python -c "import bcrypt; print(bcrypt.hashpw('password'.encode('utf-8'),
3     bcrypt.gensalt()))"
4 Traceback (most recent call last):
5   File "<string>", line 1, in <module>
  ImportError: No module named 'bcrypt'
```

As you can see, the behavior of the `python -c "import bcrypt..."` command changes after the `source env/bin/activate` call.

In one instance we have `bcrypt` available to us, and in the next we don't. This is the kind of separation we're looking to achieve with virtual environments, which is now easily achieved.

How does a virtual environment work?

So what exactly does it mean to “activate” an environment? Knowing what's going on under the hood can be pretty important for a developer, especially when you need to understand execution environments, dependency resolution, etc.

To explain how this works, let's first check out the locations of the different `python` executables. With the environment “deactivated”, run:

```
1 $ which python
2 /usr/bin/python
```

Now activate it and run the command again:

```
1 $ source env/bin/activate
2 (env) $ which python
3 /Users/michaelherman/python-virtual-environments/env/bin/python
```

After activating the environment we're now getting a different path for the `python` executable because

in an active environment the `$PATH` environment variable is slightly modified.

Notice the difference between the first path in `$PATH` before and after the activation:

```
1 $ echo $PATH
2 /usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:
3
4 $ source env/bin/activate
5 (env) $ echo $PATH
6 /Users/michaelherman/python-virtual-environments/env/bin:/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:
```

In the latter example, our virtual environment's "bin" directory is now at the beginning of the path. That means it's the *first* directory searched when running an executable on the command line. Thus, the shell uses our virtual environment's instance of Python instead of the system-wide version.

Other packages that bundle Python, like Anaconda (https://en.wikipedia.org/wiki/Anaconda_%28Python_distribution%29), also tend to manipulate your path when you activate them. Just be aware of this in case you run into problems with your other environments. This can become a problem if you start activating multiple environments at once.

This begs the questions:

- What's the difference between these two executables anyway?
- How is the virtual environment's Python executable able to use something other than the system's site-packages?

This can be explained by how Python starts up and where it is located on the system. There actually isn't any difference between these two Python executables. *It's their directory locations that matter.*

When Python is starting up, it looks at the path of its binary (which, in a virtual environment, is actually just a copy of, or symlink to, your system's Python binary). It then sets the location of `sys.prefix` and `sys.exec_prefix` based on this location, omitting the "bin" portion of the path.

The path located in `sys.prefix` is then used for locating the "site-packages" directory by searching the relative path `lib/pythonX.X/site-packages/`, where `X.X` is the version of Python you're using.

In our example, the binary is located at `/Users/michaelherman/python-virtual-environments/env/bin`, which means `sys.prefix` would be `/Users/michaelherman/python-virtual-environments/env`, and therefore the "site-packages" directory used would be `/Users/michaelherman/python-virtual-environments/env/bin/lib/pythonX.X/site-packages`. Finally, this path is stored in the `sys.path` array, which contains all of the locations that a package can reside.

Managing virtual environments with

virtualenvwrapper

While virtual environments certainly solve some big problems with package management, they're not perfect. After creating a few environments, you'll start to see that they create some problems of their own, most of which revolve around managing the environments themselves. To help with this, the `virtualenvwrapper` (<https://virtualenvwrapper.readthedocs.org/en/latest/>) tool was created, which is just some wrapper scripts around the main `virtualenv` tool.

A few of the more useful features of `virtualenvwrapper` are that it:

- Organizes all of your virtual environments in one location;
- Provides methods to help you easily create, delete, and copy environments; and,
- Provides a single command to switch between environments

While some of these features may seem small or insignificant, you'll soon learn that they're important tools to add to your workflow.

To get started, you can download the wrapper with `pip`:

```
1 $ pip install virtualenvwrapper
```

For Windows, you should use `virtualenvwrapper-win` (<https://pypi.python.org/pypi/virtualenvwrapper-win>) instead.

Once installed, we'll need to activate its shell functions, which can be done by running `source` on the installed `virtualenvwrapper.sh` script. When you first install it with `pip`, the output of the installation will tell you the exact location of `virtualenvwrapper.sh`. Or you can simply run:

```
1 $ which virtualenvwrapper.sh
2 /usr/local/bin/virtualenvwrapper.sh
```

Using that path, add the following three lines to your shell's startup file. If you're using the Bash shell, you would place these lines in either the `~/.bashrc` file or `~/.profile` file. For other shells, like `zsh`, `csh`, or `fish`, you would need to use the startup files specific to that shell. All that matters is these commands are executed when you log in or open a new shell.

```
1 export WORKON_HOME=$HOME/.virtualenvs # optional
2 export PROJECT_HOME=$HOME/projects    # optional
3 source /usr/local/bin/virtualenvwrapper.sh
```

It's not required to define the `WORKON_HOME` and `PROJECT_HOME` environment variables. `virtualenvwrapper` has default values for those, but you can override them by defining values.

Finally, reload the startup file:

```
1 $ source ~/.bashrc
```

There should now be a directory located at `$WORKON_HOME` that contains all of the virtualenvwrapper data/files:

```
1 $ echo $WORKON_HOME
2 /Users/michaelherman/.virtualenvs
```

You'll also now have the shell commands available to you to help you manage the environments. Here are just a few available:

- `workon` (http://virtualenvwrapper.readthedocs.org/en/latest/command_ref.html#workon)
- `deactivate` (http://virtualenvwrapper.readthedocs.org/en/latest/command_ref.html#deactivate)
- `mkvirtualenv` (http://virtualenvwrapper.readthedocs.org/en/latest/command_ref.html#mkvirtualenv)
- `cdvirtualenv` (https://virtualenvwrapper.readthedocs.org/en/latest/command_ref.html#cdvirtualenv)
- `rmvirtualenv` (https://virtualenvwrapper.readthedocs.org/en/latest/command_ref.html#rmvirtualenv)

For more info on commands, installation, and configuring virtualenvwrapper, check out their documentation (<http://virtualenvwrapper.readthedocs.org/en/latest/install.html>).

Now anytime you want to start a new project, all you have to do is:

```
1 $ mkvirtualenv my-new-project
2 (my-new-project) $
```

This will create and activate a new environment in the directory located at `$WORKON_HOME`, where all virtualenvwrapper environments are stored.

To stop using that environment, you just need to deactivate it like before:

```
1 (my-new-project) $ deactivate
2 $
```

If you have many environments to choose from, you can list them all with the `workon` function:

```
1 $ workon
2 my-new-project
3 my-django-project
4 web-scraper
```

And finally, to activate:


```
1 $ workon web-scraper
2 (web-scraper) $
```

Now you don't have to remember where you installed your environments, you can easily delete or copy them as you wish, and your project directory is less cluttered!

Using different versions of python

Unlike the old `virtualenv` tool, `pyenv` doesn't support creating environments with arbitrary versions of Python, which means you're stuck using the default Python 3 installation for all of the environments you create. While you *can* upgrade an environment to the latest system version of Python (via the `--upgrade` option) if it changes, you still can't actually specify a particular version.

There are quite a few ways to install Python (<http://stackabuse.com/install-python-on-mac-osx/>), but few of them are easy enough or flexible enough to frequently uninstall and re-install different versions of the binary.

This is where `pyenv` (<https://github.com/yyuu/pyenv>) comes in to play.

Despite the similarity in names (`pyvenv` vs `pyenv`), `pyenv` is different in that its focus is to help you switch between Python versions on a system-level as well as a project-level. So, while `pyvenv`'s purpose is to separate out modules, `pyenv`'s purpose is to separate Python versions.

You can start by installing `pyenv` with either Homebrew (<http://brew.sh/>) (on OS X), or with the `pyenv-installer` (<https://github.com/yyuu/pyenv-installer>) project:

Homebrew

```
1 $ brew install pyenv
```

pyenv-installer

```
1 $ curl -L https://raw.githubusercontent.com/yyuu/pyenv-installer/master/bin
    /pyenv-installer | bash
```

Unfortunately, `pyenv` does not support Windows. A few alternatives to try are `pywin` (<https://github.com/davidmarble/pywin>) and `anyenv` (<https://github.com/mislav/anyenv>).

Once you have `pyenv` on your system, here are a few of the basic commands you're probably interested in:

```
1 $ pyenv install 3.5.0 # Install new version
2 $ pyenv versions     # List installed versions
3 $ pyenv exec python -V # Execute 'python -V' using pyenv version
```

In these few lines we install the 3.5.0 version of Python, ask `pyenv` to show us all of the versions available to us, and then execute the `python -V` command using the `pyenv` -specified version.

To give you even more control, you can then use any of the available versions for either “global” use or “local” use. Using `pyenv` with the `local` command sets the Python version for a specific project or directory by storing the version number in a local `.python-version` file. We can set the “local” version like this:

```
1 $ pyenv local 2.7.11
```

This creates the `.python-version` file in our current directory, as you can see here:

```
1 $ ls -la
2 total 16
3 drwxr-xr-x  4 michaelherman  staff  136 Feb 22 10:57 .
4 drwxr-xr-x  9 michaelherman  staff  306 Jan 27 20:55 ..
5 -rw-r--r--  1 michaelherman  staff    7 Feb 22 10:57 .python-version
6 -rw-r--r--  1 michaelherman  staff   52 Jan 28 17:20 main.py
```

This file only contains the contents “2.7.11”. Now when you execute a script using `pyenv`, it’ll load this file and use the specified version, assuming it’s valid and exists on your system.

Moving on with our example, let’s say we have a simple script called `main.py` in our project directory that looks like this:

```
1 import sys
2 print('Using version:', sys.version[:5])
```

All it does is print out the version number of the Python executable being used. Using `pyenv` and the `exec` command, we can run the script with any of the different versions of Python we have installed.

```
1 $ python main.py
2 Using version: 2.7.5
3 $ pyenv global 3.5.0
4 $ pyenv exec python main.py
5 Using version: 3.5.0
6 $ pyenv local 2.7.11
7 $ pyenv exec python main.py
8 Using version: 2.7.11
```

Notice how `pyenv exec python main.py` uses our “global” Python version by default, but then it uses the “local” version after one is set for the current directory.




This can be very powerful for developers who have lots of projects with varying version requirements. Not only can you easily change the default version for all projects (via `global`), but you can also override it to specify special cases.

Conclusion

In this article you learned about how Python dependencies are stored and resolved, and how to use different community tools to help get around various packaging and versioning problems.

As you can see, thanks to the huge Python community there are quite a few tools at your disposal to help with these common problems. As you progress as a developer, be sure to take time to learn how to use these tools to your advantage. You may even find unintended uses for them, or learn to apply similar concepts to other languages you use.

This is a collaboration piece between Scott Robinson, author of Stack Abuse (<http://stackabuse.com>) and the folks at Real Python.

 Posted by Real Python  Mar 28th, 2016  fundamentals (</blog/categories/fundamentals/>), python (</blog/categories/python/>)

See an error in this post? Please submit a pull request on Github (<https://github.com/realpython/realpython-blog>).

Want to learn more? Download the Real Python course.

Download Now » \$60 (<https://app.simplegoods.co/i/IQCZADOY>)

Or, click here (<http://www.realpython.com/>) to learn more about the course.

[« Python for Social Scientists \(/blog/python/python-for-social-scientists/\)](/blog/python/python-for-social-scientists/)

[Deploying Django + Python 3 + PostgreSQL to AWS Elastic Beanstalk » \(/blog/python/deploying-a-django-app-and-postgresql-to-aws-elastic-beanstalk/\)](/blog/python/deploying-a-django-app-and-postgresql-to-aws-elastic-beanstalk/)

Comments

19 Comments

Real Python

 Login ▾ Recommend 4 Share

Sort by Best ▾



Join the discussion...

**Stephanie M. Davis** • 3 days ago

This is fantastic information! Thanks so much; this is an area of much importance.

1 ^ | ▾ • Reply • Share >

**gilgamezh** • 2 months ago

Nice introduction. :)

I would like your opinion about Fades.

It's another way to use virtualenvs. All is automated, you only have to run your program and Fades will handle the venv creation, activation, etc :)

It works with python3 and you can create virtualenvs for any python version.

check the link! <https://github.com/PyAr/fades>

1 ^ | ▾ • Reply • Share >

**Jesús Gómez** → gilgamezh • 2 months ago

This seems interesting. It's another way to work with dependencies. What worries me is how long it takes for a program to start with fades.

1 ^ | ▾ • Reply • Share >

**michaelherman** Mod → gilgamezh • 2 months ago

I have never used Fades so I cannot provide any advice/options/etc. I can find very little about it. Try the mailing list -> <http://listas.python.org.ar/ma...>

^ | ▾ • Reply • Share >

**gilgamezh** → michaelherman • 2 months ago

I'm one of the developers of the project. Maybe my English played tricks on me. I was trying to recommend the project to you. ;)

^ | ▾ • Reply • Share >

**michaelherman** Mod → gilgamezh • 2 months ago

My bad. It looks interesting. Any interest in writing a blog post about it?

^ | ▾ • Reply • Share >

**santagada** • 3 months ago

In your "very different" link you point to a stackoverflow question about pyenv no pyvenv, I think pyvenv is also python code and similar to virtualenv... not "very different" :)

1 ^ | ▾ • Reply • Share >

Categories

- analytics (/blog/categories/analytics/) [4]
- api (/blog/categories/api/) [6]
- bottle (/blog/categories/bottle/) [2]
- data science (/blog/categories/data-science/) [8]
- devops (/blog/categories/devops/) [15]
- django (/blog/categories/django/) [29]
- docker (/blog/categories/docker/) [6]
- editors (/blog/categories/editors/) [3]
- flask (/blog/categories/flask/) [30]
- front-end (/blog/categories/front-end/) [10]
- fundamentals (/blog/categories/fundamentals/) [16]
- migrations (/blog/categories/migrations/) [3]
- opencv (/blog/categories/opencv/) [3]
- pyramid (/blog/categories/pyramid/) [1]
- scraping (/blog/categories/scraping/) [2]
- sql (/blog/categories/sql/) [1]
- testing (/blog/categories/testing/) [9]
- web2py (/blog/categories/web2py/) [1]

© Copyright 2012-2016 Real Python (<https://realpython.com>).

Questions? info@realpython.com (<mailto:info@realpython.com>).

[Back to top](#)