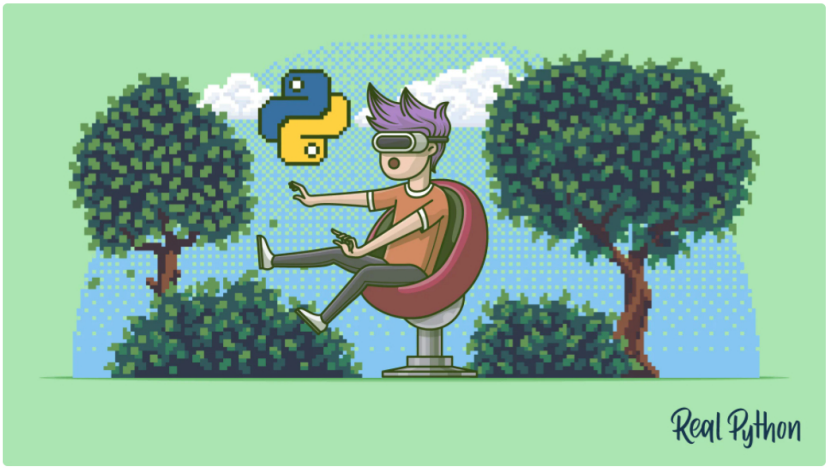


Python Virtual Environments: A Primer

by Real Python Mar 28, 2016 83 Comments best-practices intermediate python tools

Mark as Completed

Table of Contents

Watch Now This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: **Working With Python Virtual Environments**

In this article, we'll show you how to use virtual environments to create and manage separate environments for your Python projects, each using different versions of Python for execution. We'll also take a look at how Python dependencies are stored and resolved.

Free Bonus: Click here to get access to a free 5-day class that shows you how to avoid common dependency management issues with tools like Pip, PyPI, Virtualenv, and requirements files.

- Updated 2018-01-12: Clarified `pyenv` vs. `venv` usage on Python 3.6+
- Updated 2016-06-11: Added section on changing Python versions with virtualenv

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). While this has its advantages, there were some *interesting* decisions made about package storage and resolution, which has lead to some problems—particularly with how and where packages are stored

Table of Contents

- Why the Need for Virtual Environments?
- What Is a Virtual Environment?
- Using Virtual Environments
- How Does a Virtual Environment Work?
- Managing Virtual Environments With virtualenvwrapper
- Using Different Versions of Python
- Conclusion

Mark as Completed

Tweet Share Email

Recommended Video Course

Working With Python Virtual Environments

has lead to some problems - particularly with 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.

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

```
Python >>>

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

More relevant to the topic of this article, third party packages installed using easy_install or pip are typically placed in one of the directories pointed to by site.getsitepackages:

```
Python >>>

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

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** (third party libraries). At first glance, this may not seem like a big deal, and it isn't really, for **system packages** (packages that are 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:

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

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 virtual environments and the virtualenv/venv tools come 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.

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`:

Shell

```
$ pip install virtualenv
```

If you are using Python 3, then you should already have the venv module from the standard library installed.

> **Note:** From here on out, we'll assume you're using the newer `venv` tool, since there are few differences between it and `virtualenv` with regard to the actual commands. In reality, though, they are very different tools.

Start by making a new directory to work with:

Shell

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

Create a new virtual environment inside the directory:

Shell

```
# Python 2:
$ virtualenv env

# Python 3
$ python3 -m venv env
```

> **Note:** By default, this will **not** include any of your existing site packages.

The Python 3 `venv` approach has the benefit of forcing you to choose a specific version of the Python 3 interpreter that should be used to create the virtual environment. This avoids any confusion as to which Python installation the new environment is based on.

From Python 3.3 to 3.4, the recommended way to create a virtual environment was to use the `pyvenv` command-line tool that also comes included with your Python 3 installation by default. But on 3.6 and above, `python3 -m venv` is the way to go.

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

```
├── bin
│   ├── activate
│   ├── activate.csh
│   ├── activate.fish
│   ├── easy_install
│   ├── easy_install-3.5
│   ├── pip
│   ├── pip3
│   ├── pip3.5
│   ├── python -> python3.5
│   ├── python3 -> python3.5
│   └── python3.5 -> /Library/Frameworks/Python.framework/Versions/3.5/bin/python3.5
├── include
├── lib
│   └── python3.5
│       └── site-packages
└── pyvenv.cfg
```

Here's what each folder contains:

- `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 to, a few different Python tools as well as to the Python executables themselves. These files are used to ensure that 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 interesting 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 the following:

Shell

```
$ source env/bin/activate
(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 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`:

Shell

```
(env) $ deactivate
$
```

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:

Shell

```
$ pip -q install bcrypt
$ python -c "import bcrypt; print(bcrypt.hashpw('password'.encode('utf-8'), bcrypt.ge
$2b$12$vWa/VSvxxyQ9d.WGgVTdrell515Ctux36LCga8nM5QTW0.4w8TXXi
```

Here's what happens if we try the same command when the virtual environment is activated:

Shell

```
$ source env/bin/activate
(env) $ python -c "import bcrypt; print(bcrypt.hashpw('password'.encode('utf-8'), bcry
Traceback (most recent call last):
  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?

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, and so on.

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

Shell

```
$ which python
/usr/bin/python
```

Now, activate it and run the command again:

Shell

```
$ source env/bin/activate
(env) $ which python
/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:

Shell

```
$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:

$ source env/bin/activate
(env) $ echo $PATH
/Users/michaelherman/python-virtual-environments/env/bin:/usr/local/bin:/usr/bin:/bin
```

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.

> **Note:** Other packages that bundle Python, like Anaconda, 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 raises the following 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. In a virtual environment, it 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/lib/pythonX.X/site-packages`. Finally, this path is stored in the `sys.path` array, which contains all of the locations where 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 tool was created. It's 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
- 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`:

Shell

```
$ pip install virtualenvwrapper
```

> **Note:** For Windows, you should use virtualenvwrapper-win instead.

Once it's installed, we'll need to activate its shell functions. We can do this 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 the following:

Shell

```
$ which virtualenvwrapper.sh
/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 the `~/.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 that these commands are executed when you log in or open a new shell:

Shell

```
export WORKON_HOME=$HOME/.virtualenvs   # Optional
export PROJECT_HOME=$HOME/projects      # Optional
source /usr/local/bin/virtualenvwrapper.sh
```

> **Note:** 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:

Shell

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

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

Shell

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

You'll also now have the shell commands available to you to help you manage the

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

- workon
- deactivate
- mkvirtualenv
- cdvirtualenv
- rmvirtualenv

For more info on commands, installation, and configuring virtualenvwrapper, check out the documentation.

Now, anytime you want to start a new project, you just have to do this:

Shell

```
$ mkvirtualenv my-new-project
(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:

Shell

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

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

Shell

```
$ workon
my-new-project
my-django-project
web-scraper
```

Finally, here's how to activate:

Shell

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

If you would like to be able to use a single tool and switch between Python versions, `virtualenv` will allow you to do just that. virtualenv has a parameter `-p` that allows you to select which version of Python to use. Combine that with the `which` command, and we can easily select your preferred version of Python to use in a simple manner. For example, let's say that we want Python 3 as our preferred version:

Shell

```
$ virtualenv -p $(which python3) blog_virtualenv
```

This will create a new Python 3 environment.

How does this work? The `which` command is used for finding a given command in your `$PATH` variable and returning the full path to that command. So, the full path to `python3` was returned, to the `-p` parameter which takes a `PYTHON_EXE`. This could also be used for `python2` as well. Just substitute `python3` for `python2` (or `python` if you system defaults to `python2`).

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, `pyvenv` 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, 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` 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. While the purpose of `pyvenv` is to separate out modules, the purpose of `pyenv` is to separate Python versions.

You can start by installing `pyenv` with either Homebrew (on OS X) or the pyenv-installer project:

Homebrew

Shell

```
$ brew install pyenv
```

pyenv-installer

Shell

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

> **Note:** Unfortunately, `pyenv` does not support Windows. A few alternatives to try are pywin and anyenv.

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

Shell

```
$ pyenv install 3.5.0   # Install new version
$ pyenv versions        # List installed versions
$ 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:

Shell

```
$ pyenv local 2.7.11
```

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

Shell

```
$ ls -la
total 16
drwxr-xr-x  4 michaelherman  staff  136 Feb 22 10:57 .
drwxr-xr-x  9 michaelherman  staff  306 Jan 27 20:55 ..
-rw-r--r--  1 michaelherman  staff    7 Feb 22 10:57 .python-version
-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:

Python

```
import sys
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.

Shell

```
$ python main.py
Using version: 2.7.5
$ pyenv global 3.5.0
$ pyenv exec python main.py
Using version: 3.5.0
$ pyenv local 2.7.11
$ pyenv exec python main.py
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 not only how Python dependencies are stored and resolved, but also how you can 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 and the folks at Real Python.

your inbox every couple of days. No spam ever. Unsubscribe any time. Curated by the Real Python team.

```
# In Python 3.5+

>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```

Email Address

Send Me Python Tricks »

About The Team

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

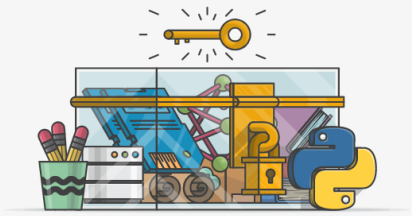

Aldren


Dan


Joanna


Michael

Master Real-World Python Skills With Unlimited Access to Real Python



Join us and get access to hundreds of tutorials, hands-on video courses, and a community of expert Pythonistas:

Level Up Your Python Skills »

What Do You Think?

Tweet Share Email

Real Python Comment Policy: The most useful comments are those written with the goal of learning from or helping out other readers—after reading the whole article and all the earlier comments. Complaints and insults generally won't make the cut here.

What's your #1 takeaway or favorite thing you learned? How are you going to put your newfound skills to use? Leave a comment below and let us know.

Keep Learning

Keep Learning