6. Modules - Python in a Nutshell, 3rd Edition

Python Environments

A typical Python programmer works on several projects concurrently, each with its own list of dependencies (typically, third-party libraries and data files). When the dependencies for all projects are installed into the same Python interpreter, it is very difficult to determine which projects use which dependencies, and impossible to handle projects with conflicting versions of certain dependencies.

Early Python interpreters were built on the assumption that each computer system would have "a Python interpreter" installed on it, which would be used to process all Python that ran on that system. Operating system distributions started to include Python in their base installation, but, because Python was being actively developed, users often complained that they would like to use a more up-to-date version of the language than their operating system provided.

Techniques arose to let multiple versions of the language be installed on a system, but installation of third-party software remained nonstandard and intrusive. This problem was eased by the introduction of the *site-packages* directory as the repository for modules added to a Python installation, but it was still not possible to maintain projects with conflicting requirements using the same interpreter.

Programmers accustomed to command-line operations are familiar with the concept of a *shell environment*. A shell program running in a process has a current directory, variables that can be set by shell commands (very similar to a Python namespace), and various other pieces of process-specific state data. Python programs have access to the shell environment through `os.environ`.

Various aspects of the shell environment affect Python's operation, as mentioned in "Environment Variables". For example, the interpreter executed in response to `python` and other commands is determined by the `PATH` environment variable. You can think of those aspects of your shell environment that affect Python's operation as your *Python environment*. By modifying it you can determine which Python interpreter runs in response to the `python` command, which packages and modules are available under certain names, and so on.

Leave the system's Python to the system

We recommend taking control of your Python environment. In particular, do not build applications on top of a system's distributed Python. Instead, install another Python distribution independently and adjust your shell environment so that the `python` command runs your locally installed Python rather than the system's Python.

Enter the Virtual Environment

The introduction of the `pip` utility created a simple way to install (and, for the first time, to uninstall) packages and modules in a Python environment. Modifying the system Python's *site-packages* still requires administrative privileges, and hence so does `pip` (although it can optionally install somewhere other than *site-packages*). Installed modules are still visible to all programs.

The missing piece is the ability to make controlled changes to the Python environment, to direct the use of a specific interpreter and a specific set of Python libraries. That is just what *virtual environments* (*virtualenvs*) give you. Creating a virtualenv based on a specific Python interpreter copies or links to components from that interpreter's installation. Critically, though, each one has its own *site-packages* directory, into which you can install the Python resources of your choice.

Creating a virtualenv is *much* simpler than installing Python, and requires far less system resources (a typical newly created virtualenv takes less than 20 MB). You can easily create and activate them on demand, and deactivate and destroy them just as easily. You can activate and deactivate a virtualenv as many times as you like during its lifetime, and if necessary use `pip` to update the installed resources. When you are done with it, removing its directory tree reclaims all storage occupied by the virtualenv. A virtualenv's lifetime can be from minutes to months.

What Is a Virtual Environment?

A virtualenv is essentially a self-contained subset of your Python environment that you can switch in or out on demand. For a Python X.Y interpreter it includes, among other things, a *bin* directory containing a Python X.Y interpreter and a *lib/pythonX.Y/site-packages* directory containing preinstalled versions of `easy-install`, `pip`, `pkg_resources`, and `setuptools`. Maintaining separate copies of these important distribution-related resources lets you update them as necessary rather than forcing reliance on the base Python distribution.

A virtualenv has its own copies of (on Windows), or symbolic links to (on other platforms), Python distribution files. It adjusts the values of `sys.prefix` and `sys.exec_prefix`, from which the interpreter and various installation utilities determine the location of some libraries. This means that `pip` can install dependencies in isolation from other environments, in the virtualenv's *site-packages* directory. In effect the virtualenv redefines which interpreter runs when you run the `python` command and which libraries are available to it, but leaves most aspects of your Python environment (such as the `PYTHONPATH` and `PYTHONHOME` variables) alone. Since its changes affect your shell environment they also affect any subshells in which you run commands.

With separate virtualenvs you can, for example, test two different versions of the same library with a project, or test your project with multiple versions of Python (very useful to check v2/v3 compatibility of your code). You can also add dependencies to your Python projects without needing any special privileges, since you normally create your virtualenvs somewhere you have write permission.

For a long time the only way to create virtual environments was the third-party `virtualenv` package, with or without help from `virtualenvwrapper`, both of which are still available for v2. You can read more about these tools in the Python Packaging User Guide. They also work with v3, but the 3.3 release added the `venv` module, `python -m venv envpath` making virtual environments a native feature of Python for the first time. ***New in 3.6***: use in preference to the `pyvenv` command, which is now deprecated.

Creating and Deleting Virtual Environments

In v3 the command `python -m venv envpath` creates a virtual environment (in the *envpath* directory, which it also creates if necessary) based on the Python interpreter used to run the command. You can give multiple directory arguments to create with a single command several virtual environments, into which you then install different sets of dependencies. `venv` can take a number of options, as shown in Table 6-1.

Table 6-1. venv options

Option	Purpose
`--clear`	Removes any existing directory content before installing the virtual environment
`--copies`	Installs files by copying on the Unix-like platforms where using symbolic links is the default
`--h` or `--help`	Prints out a command-line summary and a list of available options

Option	Purpose
`--system-site-packages`	Adds the standard system *site-packages* directory to the environment's search path, making modules already installed in the base Python available inside the environment
`--symlinks`	Installs files by using symbolic links on platforms where copying is the system default
`--upgrade`	Installs the running Python in the virtual environment, replacing whichever version the environment was created with
`--without-pip`	Inhibits the usual behavior of calling `ensurepip` to bootstrap the `pip` installer utility into the environment

v2 users must use the `python -m virtualenv` command, which does not accept multiple directory arguments.

The following terminal session shows the creation of a virtualenv and the structure of the directory tree created. The listing of the *bin* subdirectory shows that this particular user by default uses a v3 interpreter installed in */usr/local/bin*.

```
machine:~ user$ python3 -m venv /tmp/tempenv
machine:~ user$ tree -dL 4 /tmp/tempenv
/tmp/tempenv
├── bin
├── include
└── lib
    └── python3.5
        └── site-packages
            ├── __pycache__
            ├── pip
            │   └── pkg_resources
            ├── pip-8.1.1.dist-info
            ├── setuptools
            └── setuptools-20.10.1.dist-info

11 directories
machine:~ user$ ls -l /tmp/tempenv/bin/
total 80
-rw-r--r-- 1 sh wheel 1250 Oct 24 15:26 activate
-rw-r--r-- 1 sh wheel 2134 Oct 24 15:26 activate.csh
-rw-r--r-- 1 sh wheel 2388 Oct 24 15:26 activate.fish
-rwxr-xr-x 1 sh wheel  249 Oct 24 15:26 easy_install
-rwxr-xr-x 1 sh wheel  249 Oct 24 15:26 easy_install-3.5
-rwxr-xr-x 1 sh wheel  221 Oct 24 15:26 pip
-rwxr-xr-x 1 sh wheel  221 Oct 24 15:26 pip3
-rwxr-xr-x 1 sh wheel  221 Oct 24 15:26 pip3.5
lrwxr-xr-x 1 sh wheel    7 Oct 24 15:26 python->python3
lrwxr-xr-x 1 sh wheel   22 Oct 24 15:26 python3->/usr/local/bin/python3
```

Deletion of the virtualenv is as simple as removing the directory in which it resides (and all subdirectories and files in

the tree: `rm -rf envpath` in Unix-like systems). Ease of removal is a helpful aspect of using virtualenvs.

The `venv` module includes features to help the programmed creation of tailored environments (e.g., by preinstalling certain modules in the environment or performing other post-creation steps). It is comprehensively documented online, and we therefore do not cover the API further in this book.

Working with Virtual Environments

To use a virtualenv you *activate* it from your normal shell environment. Only one virtualenv can be active at a time—activations don't "stack" like function calls. Activation conditions your Python environment to use the virtualenv's Python interpreter and *site-packages* (along with the interpreter's full standard library). When you want to stop using those dependencies, deactivate the virtualenv and your standard Python environment is once again available. The virtualenv directory tree continues to exist until deleted, so you can activate and deactivate it at will.

Activating a virtualenv in Unix-based environments requires use of the `source` shell command so that the commands in the activation script make changes to the current shell environment. Simply running the script would mean its commands were executed in a subshell, and the changes would be lost when the subshell terminated. For bash and similar shells, you activate an environment located at path *`envpath`* with the command:

```
source
envpath/bin/activate
```

Users of other shells are accommodated with *activate.csh* and *activate.fish* scripts located in the same directory. On Windows systems, use *activate.bat*:

```
envpath/Scripts/activate.bat
```

Activation does several things, most importantly:

- Adds the virtualenv's *bin* directory at the beginning of the shell's `PATH` environment variable, so its commands get run in preference to anything of the same name already on the `PATH`
- Defines a `deactivate` command to remove all effects of activation and return the Python environment to its former state
- Modifies the shell prompt to include the virtualenv's name at the start
- Defines a `VIRTUAL_ENV` environment variable as the path to the virtualenv's root directory (scripting can use this to introspect the virtualenv)

As a result of these actions, once a virtualenv is activated the `python` command runs the interpreter associated with that virtualenv. The interpreter sees the libraries (modules and packages) that have been installed in that environment, and `pip`—now the one from the virtualenv, since installing the module also installed the command in the virtualenv's *bin* directory—by default installs new packages and modules in the environment's *site-packages* directory.

Those new to virtualenvs should understand that a virtualenv is not tied to any project directory. It's perfectly possible to work on several projects, each with its own source tree, using the same virtualenv. Activate it, then move around your filestore as necessary to accomplish your programming tasks, with the same libraries available (because the virtualenv determines the Python environment).

When you want to disable the virtualenv and stop using that set of resources, simply issue the command `deactivate`.

This undoes the changes made on activation, removing the virtualenv's *bin* directory from your `PATH`, so the `python` command once again runs your usual interpreter. As long as you don't delete it, the virtualenv remains available for future use by repeating the invocation to activate it.

Managing Dependency Requirements

Since virtualenvs were designed to complement installation with `pip`, it should come as no surprise that `pip` is the preferred way to maintain dependencies in a virtualenv. Because `pip` is already extensively documented, we mention only enough here to demonstrate its advantages in virtual environments. Having created a virtualenv, activated it, and installed dependencies, you can use the `pip freeze` command to learn the exact versions of those dependencies:

```
(tempenv) machine:~ user$ pip freeze
appnope==0.1.0
decorator==4.0.10
ipython==5.1.0
ipython-genutils==0.1.0
pexpect==4.2.1
pickleshare==0.7.4
prompt-toolkit==1.0.8
ptyprocess==0.5.1
Pygments==2.1.3
requests==2.11.1
simplegeneric==0.8.1
six==1.10.0
traitlets==4.3.1
wcwidth==0.1.7
```

If you redirect the output of this command to a file called *filename*, you can re-create the same set of dependencies in a different virtualenv with the command `pip install -r filename`.

To distribute code for use by others, Python developers conventionally include a *requirements.txt* file listing the necessary dependencies. When you are installing software from the Python Package Index, `pip` installs the packages you request along with any indicated dependencies. When you're developing software it's convenient to have a requirements file, as you can use it to add the necessary dependencies to the active virtualenv (unless they are already installed) with a simple `pip install -r requirements.txt`.

To maintain the same set of dependencies in several virtualenvs, use the same requirements file to add dependencies to each one. This is a convenient way to develop projects to run on multiple Python versions: create virtualenvs based on each of your required versions, then install from the same requirements file in each. While the preceding example uses exactly versioned dependency specifications as produced by `pip freeze`, in practice you can specify dependencies and constrain version requirements in quite complex ways.

Best Practices with virtualenvs

There is remarkably little advice on how best to manage your work with virtualenvs, though there are several sound tutorials: any good search engine gives you access to the most current ones. We can, however, offer a modest amount of advice that we hope will help you to get the most out of them.

When you are working with the same dependencies in multiple Python versions, it is useful to indicate the version in the environment name and use a common prefix. So for project *mutex* you might maintain environments called *mutex_35* and *mutex_27* for v3 and v2 development. When it's obvious which Python is involved (and remember you see the environment name in your shell prompt), there's less chance of testing with the wrong version. You maintain dependencies using common requirements to control resource installation in both.

Keep the requirements file(s) under source control, not the whole environment. Given the requirements file it's easy to re-create a virtualenv, which depends only on the Python release and the requirements. You distribute your project, and let your consumers decide which version(s) of Python to run it on and create the appropriate (hopefully virtual) environment(s).

Keep your virtualenvs outside your project directories. This avoids the need to explicitly force source code control systems to ignore them. It really doesn't matter where you store them—the `virtualenvwrapper` system keeps them all in a central location.

Your Python environment is independent of your process's location in the filesystem. You can activate a virtual environment and then switch branches and move around a change-controlled source tree to use it wherever convenient.

To investigate a new module or package, create and activate a new virtualenv and then `pip install` the resources that interest you. You can play with this new environment to your heart's content, confident in the knowledge that you won't be installing rogue dependencies into other projects.

You may find that experiments in a virtualenv require installation of resources that aren't currently project requirements. Rather than pollute your development environment, fork it: create a new virtualenv from the same requirements plus the testing functionality. Later, to make these changes permanent, use change control to merge your source and requirements changes back in from the fork.

If you are so inclined, you can create virtual environments based on debug builds of Python, giving you access to a wealth of instrumentation information about the performance of your Python code (and, of course, the interpreter).

Developing your virtual environment itself requires change control, and their ease of creation helps here too. Suppose that you recently released version 4.3 of a module, and you want to test your code with new versions of two of its dependencies. You *could*, with sufficient skill, persuade `pip` to replace the existing copies of dependencies in your existing virtualenv.

It's much easier, though, to branch your project, update the requirements, and create an entirely new virtual environment based on the updated requirements. You still have the original virtualenv intact, and you can switch between virtualenvs to investigate specific aspects of any migration issues that might arise. Once you have adjusted your code so that all tests pass with the updated dependencies, you check in your code *and* requirement changes, and merge into version 4.4 to complete the update, advising your colleagues that your code is now ready for the updated versions of the dependencies.

Virtual environments won't solve all of a Python programmer's problems. Tools can always be made more sophisticated, or more general. But, by golly, they work, and we should take all the advantage of those that we can.