PCCS7307 Seminar
Report

Python Virtual Environments

Submitted in partial fulfilment of
the requirements for the award of the degree of

Bachelor of Technology
in
Computer Science and Engineering

Submitted by

Aroonav Mishra [ID: B113008]

Department of Computer Science and Engineering
International Institute of Information Technology Bhubaneswar
Bhubaneswar, Odisha, India – 751 003

Spring 2016

Department of Computer Science and Engineering

International Institute of Information Technology Bhubaneswar

Certificate

This is to certify that this is a bonafide record of the project presented by Aroonav Mishra, bearing student ID B113008 during Spring 2016 in partial fulfilment of the requirements of the degree of Bachelor of Technology in Computer Science and Engineering.

Prof. Tushar Ranjan Sahoo
(Project Guide)

Date:

Undertaking

I declare that the work presented in this thesis titled "Python Virtual Environments", submitted to the Department of Computer Science & Engineering, International Institute of Information Technology, Bhubaneswar, for the award of the Bachelors of Technology degree in Computer Science & Engineering, is my original work. I have not plagiarized or submitted the same work for the award of any other degree.

Aroonav Mishra
Department of Computer Science
IIIT Bhubaneswar

Acknowledgments

I would like to thank Prof. Tushar Ranjan Sahoo, Assistant Professor in CSE Department, IIIT Bhubaneswar for providing useful inputs to improve the content and for reviewing this final report.

Aroonav Mishra

April 2016
International Institute of Information Technology Bhubaneswar

Abstract

virtualenv is a Python tool written by Ian Bicking and is used to create isolated environments for Python in which one can install packages without interfering with the other virtualenvs nor with the system Python's packages. It is a tool designated to address the problem of dealing with packages' dependencies while maintaining different versions that suit projects' needs. The report shows how to use virtual environments to create and manage separate environments for Python projects, each using different versions of Python for execution, as well as how Python dependencies are stored internally and resolved.

Contents

Introduction

The official introduction to Python is:

> Python is an easy to learn, powerful programming language. It has efficient high-level data structures and a simple but effective approach to object-oriented programming. Python's elegant syntax and dynamic typing, together with its interpreted nature, makes it an ideal language for scripting and rapid application development in many areas on most platforms.

Virtualenv is a tool used to create an isolated Python environment. This environment has its own installation directories that doesn't share libraries with other virtualenv environments (and optionally doesn't access the globally installed libraries either). The basic problem being addressed is one of dependencies and versions, and indirectly permissions. Imagine you have an application that needs version 1 of LibFoo, but another application requires version 2. How can you use both these applications? If you install everything into /usr/lib/python2.7/site-packages (or whatever your platforms standard location is), its easy to end up in a situation where you unintentionally upgrade an application that shouldnt be upgraded.

Need for virtualenv

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 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.

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:

```
>>>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?

Its 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 isnt 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 cant 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
```

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/pyvenv tools) comes into play.

Using virtual environments

1 Installing

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

$ pip install virtualenv

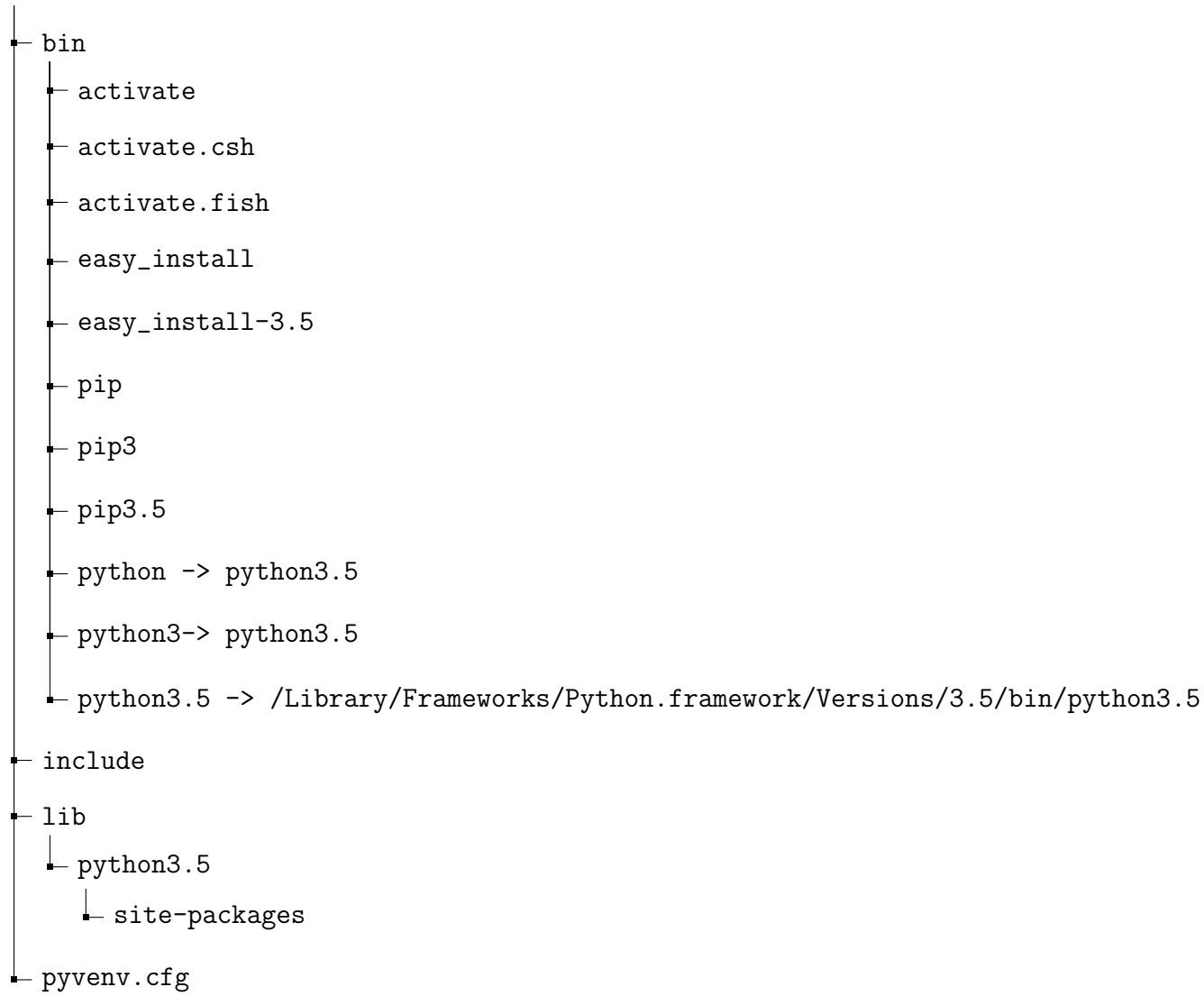
Start by making a new directory to work with:

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

Create a new virtual environment inside the directory:

$ pyvenv 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:



2 Usage

The activate scripts in the bin directory are used to set up your shell to use the environments Python executable and its site-packages by default.

In order to use this environments packages/resources in isolation, you need to activate it. To do this, just run:

```
$ 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 environments packages and settings.

To show the package isolation in action, we can use the bcrypt module as an example. Lets 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:

```
(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 youre done using a specific virtual environment.

Now, install bcyrpt and use it to hash a password:

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

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

```
$ source env/bin/activate
(env) $ python -c "import bcrypt; print(bcrypt.hashpw('password'.encode('utf-8'),
bcrypt.gensalt()))"
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 dont. This is the kind of separation we're looking to achieve with virtual environments, which is now easily achieved.

Internals of virtualenv

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

To explain how this works, lets first check out the locations of the different python executables. With the environment "deactivated", run:

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

Now activate it and run the command again:

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

After activating the environment were 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:

```
$ 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:
/usr/sbin:/sbin:
```

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

Other packages that bundle Python, like Anaconda, also tend to manipulate your path when you activate them. In case you run into problems with other environments, it might be because you start activating multiple environments at once.
This begs the questions:

> Whats the difference between these two executables anyway?
> How is the virtual environments Python executable able to use something other than the systems site-packages?

This can be explained by how Python starts up and where it is located on the system. There actually isnt 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 systems 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.

1 site-packages

Python imports site.py at startup. e.g. usr/lib/python2.6/site.py or Lib/site.py in the Python source. Where does sys.prefix in site.py come from? It comes from Python/sysmodule.c and subsequently from Modules/getpath.c. If PYTHONHOME is set, that becomes sys.prefix or else starting from the location of the Python binary, search upwards. At each step, lib/pythonX.X/os.py is looked for. If it exists, we've found sys.prefix. If we never find it, we fallback on hardcoded configure –prefix from build.

os.py is the hardcoded landmark in Modules/getpath.c. It has been defined as follows:

```
# ifndef LANDMARK
# define LANDMARK "os.py"
# endif
```

Python Runtime Services

The sys module provides access to some variables used or maintained by the interpreter and to functions that interact strongly with the interpreter. It is always available. Following are some of the most important parameters that is required for the understanding of virtualenv.

sys.exec_prefix

> A string giving the site-specific directory prefix where the platform-dependent Python files are installed; by default, this is also '/usr/local'. This can be set at build time with the –exec-prefix argument to the configure script. Specifically, all configuration files (e.g. the pyconfig.h header file) are installed in the directory exec_-prefix/lib/pythonX.Y/config, and shared library modules are installed in exec_-prefix/lib/pythonX.Y/lib-dynload, where X.Y is the version number of Python, for example 2.7.

sys.path

> A list of strings that specifies the search path for modules. Initialized from the environment variable PYTHONPATH, plus an installation-dependent default.
> As initialized upon program startup, the first item of this list, path[0], is the directory containing the script that was used to invoke the Python interpreter. If the script directory is not available (e.g. if the interpreter is invoked interactively or if the script is read from standard input), path[0] is the empty string, which directs Python to search modules in the current directory first. Notice that the script directory is inserted before the entries inserted as a result of PYTHONPATH.
> A program is free to modify this list for its own purposes.

sys.prefix

> A string giving the site-specific directory prefix where the platform independent Python files are installed; by default, this is the string '/usr/local'. This can be set at build time with the –prefix argument to the configure script. The main collection of Python library modules is installed in the directory prefix/lib/pythonX.Y while the platform independent header files (all except pyconfig.h) are stored in prefix/include/pythonX.Y, where X.Y is the version number of Python, for example 2.7.

Using different versions of python

1 pyenv vs pyvenv

Unlike the old virtualenv tool, pyvenv doesnt support creating environments with arbitrary versions of Python, which means youre 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 cant 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. So, while pyvenvs purpose is to separate out modules, pyenvs purpose is to separate Python versions.

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

> $ brew install pyenv

pyenv-installer

> $ 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 and anyenv.

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

> $ 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:

```
$ pyenv local 2.7.11
```

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

```
$ 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 will load this file and use the specified version, assuming its valid and exists on your system.

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

```
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.

```
$ 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

This report mainly dealt with the storing and resolving of python dependencies, and how to use different tools to help get around various packaging and versioning problems. As one can see, thanks to the huge Python community there are quite a few tools at ones disposal to help with these common problems.