5 Virtual environments

When dealing with Python applications, there's always a time where you'll have to deploy, use and/or test your application. But doing that can be really painful, because of the external dependencies. There's a lot of reasons for which that may fail to deploy or operate on your operation system, such as:

- Your system does not have the library you need packaged.
- Your system does not have the right version of the library you need packaged.
- You need two different versions of the same library for two different applications.

This can happen right at the time you deploy your application, or later on while running. Upgrading a Python library installed via your system manager might break your application in a snap without warning you.

The solution to this problem is to use a library directory per application, containing its dependencies. This directory will be used rather than the system installed ones to load the needed Python modules.

The tool `virtualenv` handles these directories automatically for you. Once installed, you just need to run it with a destination directory as argument.

```
$ virtualenv myvenv
Using base prefix '/usr'
New python executable in myvenv/bin/python3
```

```
Also creating executable in myvenv/bin/python
Installing Setuptools.....................done.
Installing Pip............................done.
```

Once ran, `virtualenv` creates a `lib/pythonX.Y` directory and uses it to install `setuptools` and `pip`, that will be necessary to install further Python packages.

You can now activate the virtualenv by "sourcing" the `activate` command:

```
$ source myvenv/bin/activate
```

Once you do that, your shell prompt will be prefixed by the name of your virtual environment. Calling `python` will call the Python that has been copied into the virtual environment. You can check that its working by reading the `sys.path` variable; it will have your virtual environment directory as its first component.

You can stop and leave the virtual environment at any time by calling the `deactivate` command:

```
$ deactivate
```

That's it.

Also not that you're not force to run `activate` if you want to use the Python installed in your virtual environment just once. Calling the `python` binary will also work:

```
$ myvenv/bin/python
```

Now, while we're in our activated virtual environment, we don't have access to any of the module installed and available on the system. That's good, but we probably need to install them. To do that, you just have to use the standard `pip` command, and that will install the packages in the right place, without changing anything to your system:

```
$ source myvenv/bin/activate
(myvenv) $ pip install six
```

```
Downloading/unpacking six
  Downloading six-1.4.1.tar.gz
  Running setup.py egg_info for package six

Installing collected packages: six
  Running setup.py install for six

Successfully installed six
Cleaning up...
```

And voilà. We can install all the libraries we need and then run our application from this virtual environment, without breaking our system. It's then easily imaginable to script this to automatize the installation of a virtual environment based on a list of a dependency with something along these lines:

Example 5.1 Automatic virtual environment creation

```
virtualenv myappvenv
source myappvenv/bin/activate
pip install -r requirements.txt
deactivate
```

In certain situation, it's still useful to have access to your system installed packages. You can enable them when creating your virtual environment by passing the `--system-site-packages` flag to the `virtualenv` command.

As you might guess, virtual environments are utterly useful for automated run of unit test suite. This is a really common pattern, so common that a special tool has been built to solve it, called `tox` (discussed in Section 6.7).

More recently, the PEP 405 [1] which defines a virtual environment mechanism has been accepted and implemented in Python 3.3. Indeed, the usage of virtual envi-

[1] *Python Virtual Environments*, 13th June 2011, Carl Meyer

ronment became so popular that it is now part of the standard Python library.

The `venv` module is now part of Python 3.3 and above, and allows to handle virtual environment without using the `virtualenv` package or any other one. You can call it using the `-m` flag of Python, which loads a module:

```
$ python3.3 -m venv
usage: venv [-h] [--system-site-packages] [--symlinks] [--clear] [--upgrade ↩
   ]
            ENV_DIR [ENV_DIR ...]
venv: error: the following arguments are required: ENV_DIR
```

Building virtual environment is then really simple:

```
$ python3.3 -m venv myvenv
```

And that's it. Inside `myvenv`, you will find a `pyvenv.cfg`, the configuration file for this environment. It doesn't have a lot of configuration option by default. You'll recognize `include-system-site-package`, whose purpose is the same as the `--system-site-packages` of `virtualenv` that we described earlier.

The mechanism to activate the virtual environment is the same as described earlier, "sourcing" the activate script:

```
$ source myvenv/bin/activate
(myvenv) $
```

Also here, you can call `deactivate` to leave the virtual environment.

The downside of this `venv` module is that it doesn't install `setuptools` nor `pip` by default. We will have to bootstrap the environment by ourself, contrary to `virtualenv` that does that for us.

Example 5.2 Boostraping a `venv` environment

```
(myvenv) $ wget https://bitbucket.org/pypa/setuptools/raw/bootstrap/ ↩
   ez_setup.py -O - | python
```

```
-2013-09-02 22:26:07--  https://bitbucket.org/pypa/setuptools/raw/bootstrap ↩
   /ez_setup.py
Resolving bitbucket.org (bitbucket.org)... 131.103.20.168, 131.103.20.167
Connecting to bitbucket.org (bitbucket.org)|131.103.20.168|:443...  ↩
   connected.
HTTP request sent, awaiting response... 200 OK
Length: 11835 (12K) [text/plain]
Saving to: ‘STDOUT’

100%[=============================================>] 11,835      --.-K/s    ↩
   in 0s

2013-09-02 22:26:08 (184 MB/s) - written to stdout [11835/11835]

Downloading https://pypi.python.org/packages/source/s/setuptools/setuptools ↩
   -1.1.tar.gz
Extracting in /tmp/tmp228fqm
Now working in /tmp/tmp228fqm/setuptools-1.1
Installing Setuptools
running install
running bdist_egg
running egg_info
writing dependency_links to setuptools.egg-i
[…]
Adding setuptools 1.1 to easy-install.pth file
Installing easy_install script to /home/jd/myvenv/bin
Installing easy_install-3.3 script to /home/jd/myvenv/bin

Installed /home/jd/myvenv/lib/python3.3/site-packages/setuptools-1.1-py3.3. ↩
   egg
```

```
Processing dependencies for setuptools==1.1
Finished processing dependencies for setuptools==1.1
```

We can then install *pip* via `easy_install`:

```
(myvenv) $ easy_install pip
Searching for pip
Reading https://pypi.python.org/simple/pip/
Best match: pip 1.4.1
Downloading https://pypi.python.org/packages/source/p/pip/pip-1.4.1.tar.gz# ↩
   md5=6afbb46aeb48abac658d4df742bff714
Processing pip-1.4.1.tar.gz
Writing /tmp/easy_install-hxo3b0/pip-1.4.1/setup.cfg
Running pip-1.4.1/setup.py -q bdist_egg --dist-dir /tmp/easy_install-hxo3b0 ↩
   /pip-1.4.1/egg-dist-tmp-efgi80
warning: no files found matching '*.html' under directory 'docs'
warning: no previously-included files matching '*.rst' found under  ↩
   directory 'docs/_build'
no previously-included directories found matching 'docs/_build/_sources'
Adding pip 1.4.1 to easy-install.pth file
Installing pip script to /home/jd/myvenv/bin
Installing pip-3.3 script to /home/jd/myvenv/bin

Installed /home/jd/myvenv/lib/python3.3/site-packages/pip-1.4.1-py3.3.egg
Processing dependencies for pip
Finished processing dependencies for pip
```

We can then use `pip` to install any further package we would need.

So while Python 3.3 includes `venv` by default, one has to admit that it has this little drawback to not come with what you would expect by default. It's easy enough to write a tool using the `venv` library that would mimic the default behaviour of `virtu`

`alenv`, but on the other side, there's little point working on that unless you are only targeting Python 3.3 and above. On the other hand, the *pip* bootstrapping code has been merged into Python 3.4, meaning that this bootstrap problem is solved by the latest Python version.

Anyway, since like most projects, you probably target Python 2 and Python 3, relying only on the `venv` module isn't really an option. Sticking with `virtualenv` for now is probably the best solution. Considering that they both function in an identical manner, this shouldn't be a problem.