Creating Virtual Python Software Environments with Virtualenv

Hans Petter Langtangen[1,2]

Anders Johansen[1]

[1]Center for Biomedical Computing, Simula Research Laboratory
[2]Department of Informatics, University of Oslo

Apr 11, 2016

0.1 Motivation

The most difficult aspect of installing a particular software package is usually to get all the dependencies right. That is, the package requires the existence of a lot of other packages on the computer system. These packages also have their dependencies. In addition, some packages only work with certain versions of other packages. Getting dependencies and software versions right quickly becomes a challenging problem. Even more critical is the fact that installing a new set of packages brings in other versions of some software that affect the behavior of previously installed software on the computer system.

To be specific, think of package `a` that depends on packages `b` and `c` in their respective versions 1.0 and 1.2 or newer. Then we install package `d`, which also requires package `c`, but in an older version 0.6. Package `c` in v0.6 overwrites v1.2, causing package `a` to break.

A good solution to the problem of incompatible dependencies, and in fact a good solution to handle dependencies in general, is to create an isolated virtual environment where we can install the packages in the versions we want. We can have several such isolated environments on our system and none of them interfere with the global software system. Also, we can easily delete an environment when it is no longer needed.

0.2 Creating an isolated environment with Virtualenv

Use virtualenvwrapper!

Since these notes were written, Virtualenvwrapper has a emerged as a tool that simplifies the user of Virtualenv. Check out its documentation[a].

[a]https://virtualenvwrapper.readthedocs.org/en/latest/

Virtualenv is a tool that creates isolated Python environments. More precisely, it allows creating virtual environments that have different versions of Python and Python libraries. This makes it possible to test applications using different set of libraries, or checking if an upgrade of a library will cause errors, without affecting the computer system's global Python installation with all its site-packages.

Note that Virtualenv can only contain Python packages. If an environment needs other types of software, this software must be installed separately and globally on the computer system.

The recommended way to install Virtualenv is from PyPi (the Python Package Index) using `pip`[1]:

```
Terminal> sudo pip install virtualenv
```

The `pip` tool itself can be installed from pure Python source[2] via the standard `sudo python setup.py install` command. On Debian systems (including Ubuntu) one can install `pip` by `sudo apt-get install python-pip`.

The `virtualenv` script creates a new virtual environment in a destination directory here named `venv`:

```
Terminal> virtualenv venv
```

By default, virtualenv will symlink to the system's site-packages if the user installs a package in the virtual environment that is already installed globally on the computer system. To create a totally isolated environment one can use the `--no-site-packages` switch when creating the environment:

```
Terminal> virtualenv --no-site-packages venv
```

One can apply the `-p` flag to specify which Python executable to use as the `python` program in the environment:

```
Terminal> virtualenv -p /usr/bin/python2.6
```

The next step is to activate the virtual environment. To do this, we source the activation script from the `bin` subdirectory of the newly created `venv` directory:

```
Terminal> cd venv/
Terminal> source bin/activate
```

Successful execution of the `activate` script changes the prompt in the terminal window so that the prompt is prefixed with the name of the virtual environment one is using:

[1]https://pypi.python.org/pypi/pip
[2]https://pypi.python.org/pypi/pip

```
Terminal> source bin/activate
(venv)Terminal>
```

Packages we install will now be installed in the `lib/pythonX.Y/site-packages` directory within the environment, where `X.Y` is the Python version. First we install `yolk`, which is a simple tool for listing all installed Python packages:

```
(venv)Terminal> pip install yolk
(venv)Terminal> yolk -l
```

The latter command lists all Python packages installed in the current virtualenv. At this point, there is nothing more than the essential tools like Python and `pip`, and of course yolk itself.

Let us also install the Python web framework Django:

```
(venv)Terminal> pip install Django
```

Using yolk we see that the Django version we installed is 1.5.1:

```
(venv)Terminal> yolk -l | grep Django
Django          - 1.5.1        - active
```

Within a Virtualenv environment, the local `python` interpreter and local packages are always used:

```
(venv)Terminal> which python
/some/path/to/venv/bin/python
(venv)Terminal> python -c 'import django as m; print m'
<module 'django' from '/some/path/to/venv/local/lib/...'>
```

What can be installed by `pip install`? The above examples have installed Python packages whose names and details are present in the Python Package Index (PyPi)[3]. One can also install from tarballs as long as the root directory of the tarball contains a `setup.py` file to do the installation:

```
(venv)Terminal> pip install ../some/dir/package.tar.gz
(venv)Terminal> pip install http://some.net/dir/package.tar.gz
```

Installation directly from the source in a version control system is also possible (if a `setup.py` resides in the root directory):

```
(venv)Terminal> pip install -e \
                git+https://github.com/hplgit/odespy.git#egg=odespy
```

The syntax is `pip install -e vcs+URL#egg=packaname`, where `vcs` is the name of the version control system (`hg`, `git`, `svn`, `cvs`, `bzr`).

[3]`http://pypi.python.org/pypi`

0.3 Creating a slightly different environment

Say that we have written an app for Django v1.5.1 and want to check that it is compatible with an earlier version of Django, e.g., v1.4.1. With Virtualenv this is very easy. All we need to do is deactivating the current virtual environment and create a new one where we install Django v1.4.1. Deactivation of an environment is done by the command deactivate:

```
(venv)Terminal> deactivate
Terminal>
```

Observe that deactivation removes the prompt prefix `(venv)`. To re-activate, just run `source bin/activate`.

With the same procedure as before we create a new virtual environment, now called `venv2`, and source its `activate` file. This time we install a specific Django version:

```
Terminal> virtualenv venv2
Terminal> cd venv2
Terminal> source bin/activate
(venv2)Terminal> pip install yolk Django==1.4.1
```

As always, `yolk` is our tool to assure that the correct version a software is installed:

```
(venv2)Terminal> yolk -l | grep Django
Django          - 1.4.1        - active
```

Now we have two Python environments, `venv` with Django 1.5.1 and `venv2` with Django 1.4.1. This makes it easy to test how the same app behaves on different versions of Django without making any changes to the system's configuration. Just copy the files for the app to a directory in the virtual environment and run.

0.4 Copying an environment

There is not a good and easy way to fully share a virtual environment across machines. The most convenient solution is first to create the environment and then re-install this environment in a new environment. A *requirements file* can be used for this cause. First we use `pip freeze` to save a list of requirements to file:

```
(venv2)Terminal> pip freeze > requirements.txt
```

If all the above steps in creating the `venv2` environment have been followed, the requirements file should have the following content:

```
Django==1.4.1
argparse==1.2.1
wsgiref==0.1.2
yolk==0.4.3
```

In case one has also installed a Python package from the repository of a version control system, the particular commit version (and of course the URL) is recorded as data for that package.

To replicate exactly the same environment inside another Virtualenv environment, we create a new environment, say it is called `venv3`, copy the `requirements.txt` file to the `venv3` environment, and use `pip` to install all the packages and their versions specified in `requirements.txt` at once:

```
(venv3)Terminal> pip install -r requirements.txt
```

A `yolk -l` command can be used to check the success of the multiple installations.

A word of caution is necessary here. Distributing a `requirements.txt` file produced by `pip freeze` will not always re-create an environment by a simple `pip install -r requirements.txt`. For example, the output from `pip freeze` may not account for the fact that some packages must be installed before others. When creating a Python environment for doing scientific computing, `numpy` is a package that must be installed before most other packages. Some packages can be challenging to compile via `pip install` (ScientificPython is an example, although manual execution of `setup.py` runs fine). Also, many Python scientific computing packages depend on much non-Python software that cannot be installed by `pip`. For such complex environments it is recommended to create a script that performs the manual installation tasks, but it can utilize `pip` to as large extent as possible.

0.5 Installing a scientific computing environment

Basic Python packages for scientific computing include `numpy`, `sympy`, `matplotlib`, `scipy`, `ipython`, `nose`, and `odespy`. Some of these packages depend on a series of non-Python software packages that must be installed on the system. These and other relevant Debian packages are

```
gcc g++ gfortran
libatlas-base-dev libsuitesparse-dev
tcl8.5-dev tk8.5-dev
subversion mercurial cvs git gitk
libfreetype6-dev libpng-dev
mayavi2 tcl-vtk
libsqlite3-dev
```

With this software in place, we can go on with `pip install` of Python packages:

```
Terminal> packages="numpy sympy matplotlib scipy ipython nose"
Terminal> for p in packages; do pip install $p; done
Terminal> pip install -e \
git+https://github.com/hplgit/odespy.git#egg=odespy
```

Note that `pip install` is preferred over `apt-get install` of Debian packages because `pip` will usually install a newer version of the package. It also opens up the possibility for installing the development version directly from the package's repository on (e.g.) GitHub.

A `pip freeze > requirements.txt` results as usual in a list of the packages in the environment, but this file is not so useful.

Warning.

This requirement file cannot be used to recreate the environment. The reason is that there is no way to impose a certain sequence of the packages for installation. This is demanded, because `numpy` must be installed before `scipy`, `matplotlib`, and most other packages for numerical computing. One must therefore provide a Bash script or Python program for installing the environment. Virtualenv is still useful for having multiple environments with different versions of, e.g., `numpy` and `matplotlib`, but `pip install` via a `requirements.txt` file is not possible.

A Bash script for installing the environment above may look like

```
#!/bin/sh
apt="yes | sudo apt-get install"
$apt gcc g++ gfortran
$apt libatlas-base-dev libsuitesparse-dev
$apt tcl8.5-dev tk8.5-dev
$apt subversion mercurial cvs git gitk
$apt libfreetype6-dev libpng-dev
$apt mayavi2 tcl-vtk
$apt libsqlite3-dev

packages="numpy sympy matplotlib scipy ipython nose"
for p in packages; do pip install $p; done
pip install -e git+https://github.com/hplgit/odespy.git#egg=odespy
```

Other packages can be added to the script as well:

```
pip install -e git+https://github.com/hplgit/scitools.git#egg=scitools
# Do manual install of Scientific Python
if [ ! -d srclib ]; then mkdir scrlib; fi
cd srclib
hg clone https://bitbucket.org/khinsen/scientificpython
cd scientificpython
sudo python setup.py install
cd ../..
```

hpl 1: Should refer to the Vagrant document for how to make a list of packages and then autogenerate scripts.

References.

- Virtualenv documentation[4]
- Pip documentation[5]

[4] `http://www.virtualenv.org/en/latest/`
[5] `http://guide.python-distribute.org/pip.html`