Creating and Managing Python Virtual Environments

Virtual Environments – What and Why?

The Python docs tell us that Python applications will often use packages and modules that don’t come as part of the standard library. Applications will sometimes need a specific version of a library, because the application may require that a particular bug has been fixed or the application may be written using an obsolete version of the library’s interface.

This means it may not be possible for one Python installation to meet the requirements of every application. If application A needs version 1.0 of a particular module but application B needs version 2.0, then the requirements are in conflict and installing either version 1.0 or 2.0 will leave one application unable to run.

The solution for this problem is to create a virtual environment, a self-contained directory tree that contains a Python installation for a particular version of Python, plus a number of additional packages.

Python Virtual Environment Docs

Conda

There are other multiple ways to create virtual environments, but in this course we will be using Conda.

Conda is a package manager application that quickly installs, runs, and updates packages and their dependencies. It allows you to easily set up and switch between environments on your local computer. Conda is included in all versions of Anaconda and Miniconda.

Conda Docs

Creating Virtual Environments

1. Verify that Conda is installed by checking it's version with `conda --version`. Next let's update conda using the update command `conda update conda`. If a newer version is available, go ahead and accept the update.

2. Create your first virtual environment with the command `conda create --name dogs`. This will create a brand new environemnt with the name `dogs`.

3. Activate your new environment with the command (Mac) `source activate dogs`, (Windows) `activate dogs`. You can also deactive environments with (Mac) `source deactivate dogs`, (Windows) `deactivate dogs`. Note that your environment will live in `/envs/dogs`.

4. Let's create and activate another environment. Type `conda create --name cats` followed by (Mac) `source activate cats` or (Windows) `activate cats`.

5. Let's list out our new environments with the command `conda info --envs`. You should see both of your new enviroments plus your root `/home/username/miniconda`.

6. Conda puts an asterisk next to your active environment. Again, type `conda info --envs` to verify which environment is active and then type (Mac) `source activate dogs` to change back to `dogs` or (Windows) `activate dogs`.

Additional commands

- COPY: `conda create --name dogs --clone fish`. This will make a copy of `dogs` named `fish`.

- DELETE: `conda remove --name fish --all`. This will delete `fish`.

Managing Python

Conda treats Python the same as any other package, so it's very easy to manage and update multiple installations.

1. Let's check what versions of Python are available to install. In your terminal type `conda search --full-name python`.

2. Let's say you need Python 3 for your homework, but you don't want to overwrite your Python 2.7 environment. You can create and activate a new environment named `py36`, and install the latest version of Python 3 like this.

- `conda create --name py36 python=3.6`
- (Mac) `source activate homework`
- (Windows) `activate homework`

NOTE It helps to give your virtual environments descriptive names, so you can recognize them easily.

3. Verify that your new environment was added with `conda info --envs`.

4. Verify it uses `Python3` with `python --version`.

5. Switch back to `dogs` by typing (Mac) `source activate dogs` (Windows) `activate dogs`.

Managing Packages With Conda

As of right now, the only package we have installed is `Python3`. Let's see what packages we have, which ones are available and then install a specific package.

1. In your terminal type `conda list` to see what packages you have

installed.

2. Let's search for a package called beautifulsoup4 by typing conda search beautifulsoup4.

3. Let's install it to our dogs envoronment with the command conda install --name dogs beautifulsoup4. When prompted with "The following NEW packages will be INSTALLED:", type "yes" and hit Enter to proceed.

NOTE Conda will default package installs to your active environment, unless you give it the name of a different environment.

You can view all of the packages available for conda install here.

Managing Packages With Pip

For packages that are not available from conda or Anaconda.org, we can often install the package with pip (short for “pip install packages”).

TIP: Pip is only a package manager, so it cannot manage environments for you. Pip cannot even update Python, because unlike conda it does not consider Python a package. But it does install some things that conda does not, and vice versa. Both pip and conda are included in Anaconda and Miniconda.

1. Activate the environment where you want pip to install packages.

2. Let's install a popular testing package called Unit Test. In your terminal type pip install unittest2.

3. To verify your installs type conda list.

pip freeze

1. pip freeze will output installed packages in a format that can be used for requirements.txt, which we will go over in shortly.

2. In your terminal, type pip freeze. You should see an output similar to the following. Notice the version numbers next to each package.

```
cffi==1.9.1
conda==4.3.17
cryptography==1.7.1
idna==2.2
pyasn1==0.1.9
pycosat==0.6.1
pycparser==2.17
pyOpenSSL==16.2.0
requests==2.12.4
six==1.10.0
```

pip list

1. pip list will output installed packages including editables installs.

2. In your terminal, type pip list. You should see an output similar to the following. Notice there are no version numbers.

```
cffi (1.9.1)
conda (4.3.17)
cryptography (1.7.1)
idna (2.2)
pip (9.0.1)
pyasn1 (0.1.9)
pycosat (0.6.1)
pycparser (2.17)
pyOpenSSL (16.2.0)
requests (2.12.4)
setuptools (27.2.0)
six (1.10.0)
wheel (0.29.0)
```

pip install -r requirements.txt

1. pip install -r requirements.txt will install a list of requirements contained in a requirements.txt file that you have in your projects. Requirements files are used to hold the result from pip freeze for the purpose of achieving repeatable installations. In this case, your requirement file contains a pinned version of everything that was installed when pip freeze was run.

Example requirements.txt from the Python docs.

```
#
####### example-requirements.txt #######
#
###### Requirements without Version Specifiers ######
nose
nose-cov
beautifulsoup4
#
###### Requirements with Version Specifiers ######
#   See https://www.python.org/dev/peps/pep-
0440/#version-specifiers
docopt == 0.6.1             # Version Matching. Must
be version 0.6.1
keyring >= 4.1.1            # Minimum version 4.1.1
coverage != 3.5             # Version Exclusion.
Anything except version 3.5
Mopidy-Dirble ~= 1.1        # Compatible release.
Same as >= 1.1, == 1.*
#
###### Refer to other requirements files ######
-r other-requirements.txt
#
#
###### A particular file ######
./downloads/numpy-1.9.2-cp34-none-win32.whl
http://wxpython.org/Phoenix/snapshot-
```

```
builds/wxPython_Phoenix-3.0.3.dev1820+49a8884-cp34-
none-win_amd64.whl
#
###### Additional Requirements without Version
Specifiers ######
#   Same as 1st section, just here to show that you
can put things in any order. Note that it is
generally bad practice to list packages without
version numbers.
rejected
green
#
```

2. You can read about the requirements file format https://pip.pypa.io/en/stable/reference/pip_install/#requirements-file-format.