Virtual environments All Versions

A Virtual Environment is a tool to keep the dependencies required by different projects in separate places, by creating virtual Python environments for them. It solves the “Project X depends on version 1.x but, Project Y needs 4.x” dilemma, and keeps your global site-packages directory clean and manageable.

This helps isolate your environments for different projects from each other and from your system libraries.

Examples

Creating and using a virtual environment

`virtualenv` is a tool to build isolated Python environments. This program creates a folder which contains all the necessary executables to use the packages that a Python project would need.

Installing the virtualenv tool

This is only required once. The `virtualenv` program may be available through your distribution. On Debian-like distributions, the package is called `python-virtualenv` or `python3-virtualenv` .

You can alternatively install `virtualenv` using `pip` :

```
$ pip install virtualenv
```

Creating a new virtual environment

This only required once per project. When starting a project for which you want to isolate dependencies, you can setup a new virtual environment for this project:

```
$ virtualenv foo
```

This will create a `foo` folder containing tooling scripts and a copy of the `python` binary itself. The name of the folder is not relevant. Once the virtual environment is created, it is self-contained and does not require further manipulation with the `virtualenv` tool. You can now start using the virtual environment.

Activating an existing virtual environment

To *activate* a virtual environment, some shell magic is required so your Python is the one inside `foo` instead of the system one. This is the purpose of the `activate` file, that you must source into your current shell:

```
$ source foo/bin/activate
```

Windows users should type:

```
$ foo\Scripts\activate.bat
```

Once a virtual environment has been activated, the `python` and `pip` binaries and all scripts installed by third party modules are the ones inside `foo` . Particularly, all modules installed with `pip` will be deployed to the virtual environment, allowing for a contained development environment. Activating the virtual environment should also add a prefix to your prompt as seen in the following commands.

```
# Installs 'requests' to foo only, not globally
(foo)$ pip install requests
```

Saving and restoring dependencies

To save the modules that you have installed via `pip` , you can list all of those modules (and the corresponding versions) into a text file by using the `freeze` command. This allows others to quickly install the Python modules needed for the application by using the install command. The conventional name for such a file is `requirements.txt` :

```
(foo)$ pip freeze > requirements.txt
(foo)$ pip install -r requirements.txt
```

Please note that `freeze` lists all the modules, including the transitive dependencies required by the top-level modules you installed manually. As such, you may prefer to craft the `requirements.txt` file by hand , by putting only the top-level modules you need.

Exiting a virtual environment

If you are done working in the virtual environment, you can deactivate it to get back to your normal shell:

```
(foo)$ deactivate
```

Using a virtual environment in a shared host

Sometimes it's not possible to `$ source bin/activate` a virtualenv, for example if you are using mod_wsgi in shared host or if you don't have access to a file system, like in Amazon API Gateway, or Google AppEngine.

For those cases you can deploy the libraries you installed in your local virtualenv and patch your `sys.path` .

Luckly virtualenv ships with a script that updates both your `sys.path` and your `sys.prefix`

```
import os

mydir = os.path.dirname(os.path.realpath(__file__))
activate_this = mydir + '/bin/activate_this.py'
execfile(activate_this, dict(__file__=activate_this))
```

You should append these lines at the very beginning of the file your server will execute.

This will find the `bin/activate_this.py` that `virtualenv` created file in the same dir you are executing and add your `lib/python2.7/site-packages` to `sys.path`

If you are looking to use the `activate_this.py` script, remember to deploy with, at least, the `bin` and `lib/python2.7/site-packages` directories and their content.

Python 3.x ≥3.3

Built-in virtual environments

From Python 3.3 onwards, the venv module will create virtual environments. The `pyvenv` command does not need installing separately:

```
$ pyvenv foo
$ source foo/bin/activate
```

or

```
$ python3 -m venv foo
$ source foo/bin/activate
```

Specifying specific python version to use in script on Unix/Linux

In order to specify which version of python the Linux shell should use the first line of Python scripts can be a shebang line, which starts with '#!':

```
#!/usr/bin/python
```

If you are in a virtual environment, then `python myscript.py` will use the Python from your virtual environment, but `./myscript.py` will use the Python interpreter in the `#!` line. To make sure the virtual environment's Python is used, change the first line to:

```
#!/usr/bin/env python
```

After specifying the shebang line, remember to give execute permissions to the script by doing:

```
chmod +x myscript.py
```

Doing this will allow you to execute the script by running `./myscript.py` (or provide the absolute path to the script) instead of `python myscript.py` or `python3 myscript.py` .

Creating a virtual environment for a different version of python

Assuming `python` and `python3` are both installed, it is possible to create a virtual environment for Python 3 even if `python3` is not the default Python:

```
virtualenv -p python3 foo
```

or

```
virtualenv --python=python3 foo
```

or

```
python3 -m venv foo
```

or

```
pyvenv foo
```

Actually you can create virtual environment based on any version of working python of your system. You can check different working python under your `/usr/bin/` or `/usr/local/bin/` (In Linux) OR in `/Library/Frameworks/Python.framework/Versions/X.X/bin/` (OSX), then figure out the name and use that in the `--python` or `-p` flag while creating virtual environment.

Installing packages in a virtual environment

Once your virtual environment has been activated, any package that you install will now be installed in the `virtualenv` & not globally. Hence, new packages can be without needing root privileges.

To verify that the packages are being installed into the `virtualenv` run the following command to check the path of the executable that is being used :

```
(<Virtualenv Name) $ which python
/<Virtualenv Directory>/bin/python

(Virtualenv Name) $ which pip
/<Virtualenv Directory>/bin/pip
```

Any package then installed using pip will be installed in the `virtualenv` itself in the following directory :

```
/<Virtualenv Directory>/lib/python2.7/site-packages/
```

Alternatively, you may create a file listing the needed packages.

requirements.txt :

```
requests==2.10.0
```

Executing:

```
# Install packages from requirements.txt
pip install -r requirements.txt
```

will install version 2.10.0 of the package `requests` .

You can also get a list of the packages and their versions currently installed in the active virtual environment:

```
# Get a list of installed packages
pip freeze

# Output list of packages and versions into a requirement.txt file so you can recreate the virtu
pip freeze > requirements.txt
```

Alternatively, you do not have to activate your virtual environment each time you have to install a package. You can directly use the pip executable in the virtual environment directory to install packages.

```
$ /<Virtualenv Directory>/bin/pip install requests
```

More information about using pip can be found on the PIP topic .

Since you're installing without root in a virtual environment, this is *not* a global install, across the entire system - the installed package will only be available in the current virtual environment.

Making virtual environments using Anaconda

A powerful alternative to `virtualenv` is Anaconda - a cross-platform, `pip` -like package manager bundled with features for quickly making and removing virtual environments. After installing Anaconda, here are some commands to get started:

Create an environment

```
conda create -name <envname> python=<version>
```

where `<envname>` in an arbitrary name for your virtual environment, and `<version>` is a specific Python version you wish to setup.

Activate and deactivate your environment

```
# Linux, Mac
source activate <envname>
source deactivate
```

or

```
# Windows
activate <envname>
deactivate
```

Viewa list of created environments

```
conda env list
```

Remove an environment

```
conda env remove -n <envname>
```

Find more commands and features in the official conda documentation .

Managing multiple virtual enviroments with virtualenvwrapper

The virtualenvwrapper utility simplifies working with virtual environments and is especially useful if you are dealing with many virtual environments/projects.

Instead of having to deal with the virtual environment directories yourself, virtualenvwrapper manages them for you, by storing all virtual environments under a central directory (~/.virtualenvs by default).

Installation

Install virtualenvwrapper with your system's package manager.

Debian/Ubuntu-based:

```
apt-get install virtualenvwrapper
```

Fedora/CentOS/RHEL:

```
yum install python-virtualenvrwapper
```

Arch Linux:

```
pacman -S python-virtualenvwrapper
```

Or install it from PyPI using pip :

```
pip install virtualenvwrapper
```

Under Windows you can use either virtualenvwrapper-win or virtualenvwrapper-powershell instead.

Usage

Virtual environments are created with mkvirtualenv . All arguments of the original virtualenv command are accepted as well.

```
mkvirtualenv my-project
```

or e.g.

```
mkvirtualenv --system-site-packages my-project
```

The new virtual environment is automatically activated. In new shells you can enable the virtual environment with workon

```
workon my-project
```

The advantage of the workon command compared to the traditional . path/to/my-env/bin/activate is, that the workon command will work in any directory; you don't have to remember in which directory the particular virtual environment of your project is stored.

Project Directories

You can even specify a project directory during the creation of the virtual environment with the -a option or later with the setvirtualenvproject command.

```
mkvirtualenv -a /path/to/my-project my-project
```

or

```
workon my-project
cd /path/to/my-project
setvirtualenvproject
```

Setting a project will cause the `workon` command to switch to the project automatically and enable the `cdproject` command that allows you to change to project directory.

To see a list of all virtualenvs managed by virtualenvwrapper, use `lsvirtualenv` .

To remove a virtualenv, use `rmvirtualenv` :

```
rmvirtualenv my-project
```

Each virtualenv managed by virtualenvwrapper includes 4 empty bash scripts: `preactivate` , `postactivate` , `predeactivate` , and `postdeactivate` . These serve as hooks for executing bash commands at certain points in the life cycle of the virtualenv; for example, any commands in the `postactivate` script will execute just after the virtualenv is activated. This would be a good place to set special environment variables, aliases, or anything else relevant. All 4 scripts are located under `.virtualenvs/<virtualenv_name>/bin/` .

For more details read the virtualenvwrapper documentation .

Discovering which virtual environment you are using

If you are using the default `bash` prompt on Linux, you should see the name of the virtual environment at the start of your prompt.

```
(my-project-env) user@hostname:~$ which python
/home/user/my-project-env/bin/python
```

Checking if running inside a virtual environment

Sometimes the shell prompt doesn't display the name of the virtual environment and you want to be sure if you are in a virtual environment or not.

Run the python interpreter and try:

```
import sys
sys.prefix
sys.real_prefix
```

- Outside a virtual, environment `sys.prefix` will point to the system python installation and `sys.real_prefix` is not defined.
- Inside a virtual environment, `sys.prefix` will point to the virtual environment python installation and `sys.real_prefix` will point to the system python installation.

For virtual environments created using the standard library venv module there is no `sys.real_prefix` . Instead, check whether `sys.base_prefix` is the same as `sys.prefix` .

Using virtualenv with fish shell

Fish shell is friendlier yet you might face trouble while using with `virtualenv` or `virtualenvwrapper` . Alternatively `virtualfish` exists for the rescue. Just follow the below sequence to start using Fish shell with virtualenv.

- Install virtualfish to the global space

  ```
  sudo pip install virtualfish
  ```

- Load the python module virtualfish during the fish shell startup

  ```
  $ echo "eval (python -m virtualfish)" > ~/.config/fish/config.fish
  ```

- Edit this function `fish_prompt` by `$ funced fish_prompt --editor vim` and add the below lines and close the vim editor

  ```
  if set -q VIRTUAL_ENV
      echo -n -s (set_color -b blue white) "(" (basename "$VIRTUAL_ENV") ")" (set_color norm
  end
  ```

Note: If you are unfamiliar with vim, simply supply your favorite editor like this `$ funced fish_prompt --editor nano` or `$ funced fish_prompt --editor gedit`

- Save changes using `funcsave`

```
funcsave fish_prompt
```

- To create a new virtual environment use `vf new`

```
vf new my_new_env # Make sure $HOME/.virtualenv exists
```

- If you want create a new python3 environment specify it via `-p` flag

```
vf new -p python3 my_new_env
```

- To switch between virtualenvironments use `vf deactivate` & `vf activate another_env`

Official Links:

- https://github.com/adambrenecki/virtualfish
- http://virtualfish.readthedocs.io/en/latest/

Syntax

Parameters

Remarks

Virtual environments are sufficiently useful that they probably should be used for every project. In particular, virtual environments allow you to:

1. Manage dependencies without requiring root access
2. Install different versions of the same dependency, for instance when working on different projects with varying requirements
3. Work with different python versions