# Virtual Environments

Fork me on GitHub

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.

For example, you can work on a project which requires Django 1.3 while also maintaining a project which requires Django 1.0.

## virtualenv

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

Install virtualenv via pip:

```
$ pip install virtualenv
```

### Basic Usage

1. Create a virtual environment for a project:

```
$ cd my_project_folder
$ virtualenv venv
```

`virtualenv venv` will create a folder in the current directory which will contain the Python executable files, and a copy of the `pip` library which you can use to install other packages. The name of the virtual environment (in this case, it was `venv`) can be anything; omitting the name will place the files in the current directory instead.

This creates a copy of Python in whichever directory you ran the command in, placing it in a folder named `venv`.

You can also use a Python interpreter of your choice.

```
$ virtualenv -p /usr/bin/python2.7 venv
```

This will use the Python interpreter in `/usr/bin/python2.7`

2. To begin using the virtual environment, it needs to be activated:

```
$ source venv/bin/activate
```

The name of the current virtual environment will now appear on the left of the prompt (e.g. `(venv)Your-Computer:your_project UserName$)` to let you know that it's active. From now on, any package that you install using pip will be placed in the `venv` folder, isolated from the global Python installation.

Install packages as usual, for example:

```
$ pip install requests
```

3. If you are done working in the virtual environment for the moment, you can deactivate it:

```
$ deactivate
```

This puts you back to the system's default Python interpreter with all its installed libraries.

v: latest

To delete a virtual environment, just delete its folder. (In this case, it would be `rm -rf venv`.)

After a while, though, you might end up with a lot of virtual environments littered across your system, and its possible you'll forget their names or where they were placed.

### Other Notes

Running `virtualenv` with the option **`--no-site-packages`** will not include the packages that are installed globally. This can be useful for keeping the package list clean in case it needs to be accessed later. [This is the default behavior for `virtualenv` 1.7 and later.]

In order to keep your environment consistent, it's a good idea to "freeze" the current state of the environment packages. To do this, run

```
$ pip freeze > requirements.txt
```

This will create a `requirements.txt` file, which contains a simple list of all the packages in the current environment, and their respective versions. You can see the list of installed packages without the requirements format using "pip list". Later it will be easier for a different developer (or you, if you need to re-create the environment) to install the same packages using the same versions:

```
$ pip install -r requirements.txt
```

This can help ensure consistency across installations, across deployments, and across developers.

Lastly, remember to exclude the virtual environment folder from source control by adding it to the ignore list.

## virtualenvwrapper

virtualenvwrapper provides a set of commands which makes working with virtual environments much more pleasant. It also places all your virtual environments in one place.

To install (make sure **virtualenv** is already installed):

```
$ pip install virtualenvwrapper
$ export WORKON_HOME=~/Envs
$ source /usr/local/bin/virtualenvwrapper.sh
```

(Full virtualenvwrapper install instructions.)

For Windows, you can use the virtualenvwrapper-win.

To install (make sure **virtualenv** is already installed):

```
$ pip install virtualenvwrapper-win
```

In Windows, the default path for WORKON_HOME is %USERPROFILE%Envs

### Basic Usage

1. Create a virtual environment:

```
$ mkvirtualenv venv
```

This creates the `venv` folder inside `~/Envs`.

2. Work on a virtual environment:

v: latest

```
$ workon venv
```

Alternatively, you can make a project, which creates the virtual environment, and also a project directory inside `$PROJECT_HOME`, which is `cd` -ed into when you `workon myproject`.

```
$ mkproject myproject
```

**virtualenvwrapper** provides tab-completion on environment names. It really helps when you have a lot of environments and have trouble remembering their names.

`workon` also deactivates whatever environment you are currently in, so you can quickly switch between environments.

3. Deactivating is still the same:

```
$ deactivate
```

4. To delete:

```
$ rmvirtualenv venv
```

### Other useful commands

`lsvirtualenv`
: List all of the environments.

`cdvirtualenv`
: Navigate into the directory of the currently activated virtual environment, so you can browse its `site-packages`, for example.

`cdsitepackages`
: Like the above, but directly into `site-packages` directory.

`lssitepackages`
: Shows contents of `site-packages` directory.

Full list of virtualenvwrapper commands.

## virtualenv-burrito

With virtualenv-burrito, you can have a working virtualenv + virtualenvwrapper environment in a single command.

## autoenv

When you `cd` into a directory containing a `.env`, autoenv automagically activates the environment.

Install it on Mac OS X using `brew`:

```
$ brew install autoenv
```

And on Linux:

```
$ git clone git://github.com/kennethreitz/autoenv.git ~/.autoenv
$ echo 'source ~/.autoenv/activate.sh' >> ~/.bashrc
```

v: latest