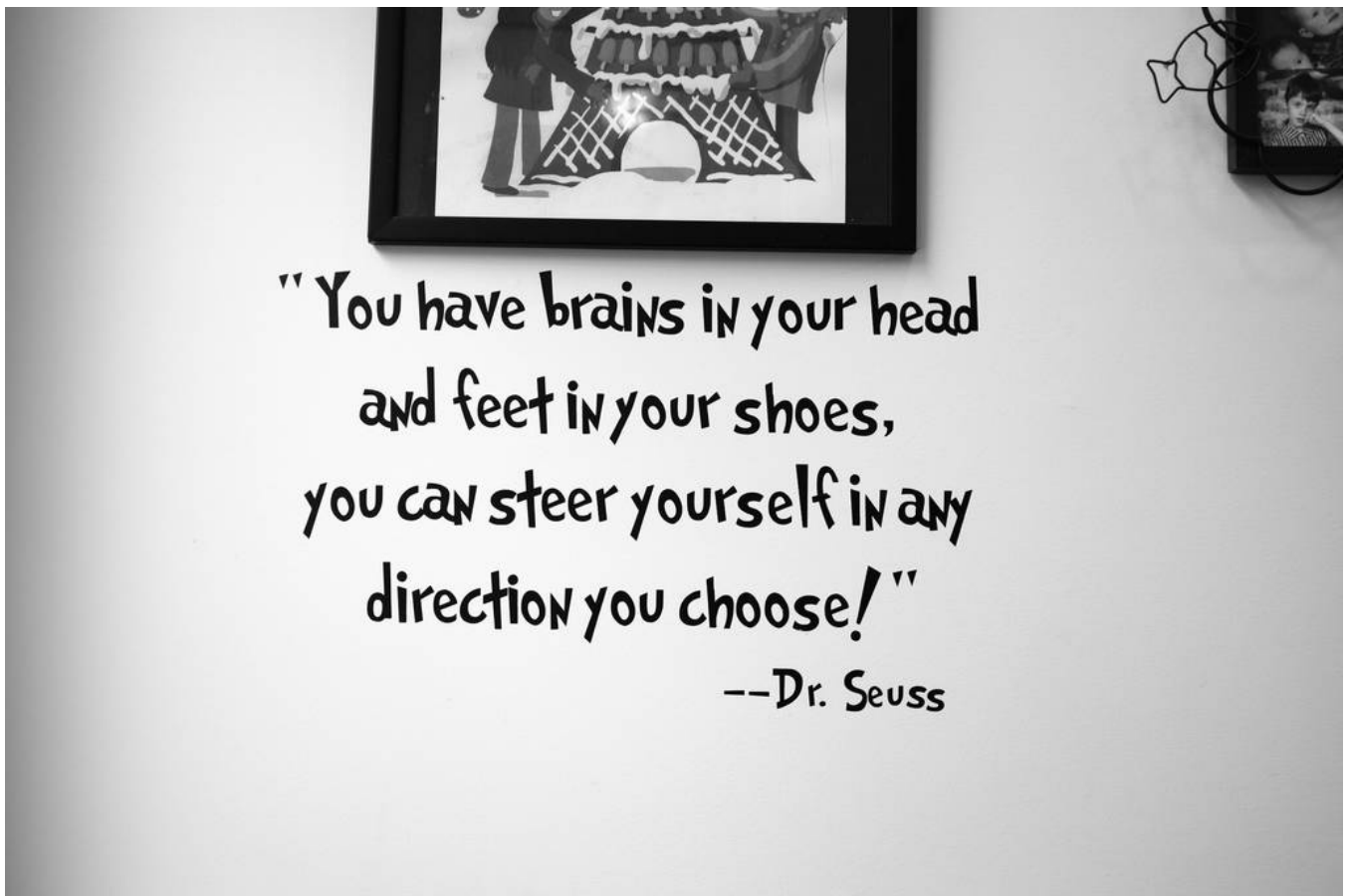




100M+ proxy pool
Residential Proxies

Try now

Pipenv & Virtual Environments



This tutorial walks you through installing and using Python packages.

It will show you how to install and use the necessary tools and make strong recommendations on best practices. Keep in mind that Python is used for a great many different purposes, and precisely how you want to manage your dependencies may change based on how you decide to publish your software. The guidance presented here is most directly applicable to the development and deployment of network services (including web applications), but is also very well suited to managing development and testing environments for any kind of project.

Note:

This guide is written for Python 3, however, these instructions should work fine on Python 2.7—if you are still using it, for some reason.

Make sure you've got Python & pip

Before you go any further, make sure you have Python and that it's available from your command line. You can check this by simply running:

```
$ python --version
```

You should get some output like 3.6.2. If you do not have Python, please install the latest 3.x version from python.org or refer to the [Installing Python](#) section of this guide.

v: latest ▾

Note:

If you're newcomer and you get an error like this:

```
>>> python
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'python' is not defined
```

It's because this command is intended to be run in a *shell* (also called a *terminal* or *console*). See the Python for Beginners [getting started tutorial](#) for an introduction to using your operating system's shell and interacting with Python.

Additionally, you'll need to make sure you have [pip](#) available. You can check this by running:

```
$ pip --version
```

If you installed Python from source, with an installer from [python.org](#), or via [Homebrew](#) you should already have pip. If you're on Linux and installed using your OS package manager, you may have to [install pip](#) separately.

Installing Pipenv

[Pipenv](#) is a dependency manager for Python projects. If you're familiar with Node.js' [npm](#) or Ruby's [bundler](#), it is similar in spirit to those tools. While [pip](#) can install Python packages, Pipenv is recommended as it's a higher-level tool that simplifies dependency management for common use cases.

Use pip to install Pipenv:

```
$ pip install --user pipenv
```

Note:

This does a [user installation](#) to prevent breaking any system-wide packages. If pipenv isn't available in your shell after installation, you'll need to add the [user base](#)'s binary directory to your PATH.


On Linux and macOS you can find the user base binary directory by running `python -m site --user-base` and adding bin to the end. For example, this will typically print `~/local` (with `~` expanded to the absolute path to your home directory) so you'll need to add `~/local/bin` to your PATH. You can set your PATH permanently by [modifying ~/.profile](#).

On Windows you can find the user base binary directory by running `py -m site --user-site` and replacing site-packages with Scripts. For example, this could return `C:\Users\Username\AppData\Roaming\Python36\site-packages` so you would need to set your PATH to include `C:\Users\Username\AppData\Roaming\Python36\Scripts`. You can set your user PATH permanently in the [Control Panel](#). You may need to log out for the PATH changes to take effect.

Installing packages for your project

Pipenv manages dependencies on a per-project basis. To install packages, change into your project's directory (or just an empty directory for this tutorial) and run:

```
$ cd myproject
$ pipenv install requests
```

Pipenv will install the excellent [Requests](#) library and create a `Pipfile` for you in your project's directory. T  `v: latest` used to track which dependencies your project needs in case you need to re-install them, such as when you share your project with others. You should get output similar to this (although the exact paths shown will vary):

```

Creating a Pipfile for this project...
Creating a virtualenv for this project...
Using base prefix '/usr/local/Cellar/python3/3.6.2/Frameworks/Python.framework/Versions/3.6'
New python executable in ~/.local/share/virtualenvs/tmp-agwWamBd/bin/python3.6
Also creating executable in ~/.local/share/virtualenvs/tmp-agwWamBd/bin/python
Installing setuptools, pip, wheel...done.

Virtualenv location: ~/.local/share/virtualenvs/tmp-agwWamBd
Installing requests...
Collecting requests
  Using cached requests-2.18.4-py2.py3-none-any.whl
Collecting idna<2.7,>=2.5 (from requests)
  Using cached idna-2.6-py2.py3-none-any.whl
Collecting urllib3<1.23,>=1.21.1 (from requests)
  Using cached urllib3-1.22-py2.py3-none-any.whl
Collecting chardet<3.1.0,>=3.0.2 (from requests)
  Using cached chardet-3.0.4-py2.py3-none-any.whl
Collecting certifi>=2017.4.17 (from requests)
  Using cached certifi-2017.7.27.1-py2.py3-none-any.whl
Installing collected packages: idna, urllib3, chardet, certifi, requests
Successfully installed certifi-2017.7.27.1 chardet-3.0.4 idna-2.6 requests-2.18.4 urllib3-1.22

Adding requests to Pipfile's [packages]...
P.S. You have excellent taste! 🍷 📦 🍷

```

Using installed packages

Now that Requests is installed you can create a simple `main.py` file to use it:

```

import requests

response = requests.get('https://httpbin.org/ip')

print('Your IP is {0}'.format(response.json()['origin']))

```

Then you can run this script using `pipenv run`:

```
$ pipenv run python main.py
```

You should get output similar to this:

```
Your IP is 8.8.8.8
```

Using `$ pipenv run` ensures that your installed packages are available to your script. It's also possible to spawn a new shell that ensures all commands have access to your installed packages with `$ pipenv shell`.

Next steps

Congratulations, you now know how to install and use Python packages! 🍷 📦 🍷

Lower level: 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.

It can be used standalone, in place of `Pipenv`.

Install `virtualenv` via `pip`:

```
$ pip install virtualenv
```

 v: latest ▾

Test your installation:

```
$ virtualenv --version
```

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.

Note:

‘venv’ is the general convention used globally. As it is readily available in ignore files (eg: `.gitignore`)

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 the Python interpreter of your choice (like `python2.7`).

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

or change the interpreter globally with an env variable in `~/.bashrc`:

```
$ export VIRTUALENVWRAPPER_PYTHON=/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.

For Windows, same command which is mentioned in step 1 can be used for creation of virtual environment. But, to activate, we use the following command.

Assuming that you are in project directory:

```
PS C:\Users\suryav> \venv\Scripts\activate
```

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.

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

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

Note:

Python has included `venv` module from version 3.3. For more details: [venv](#).

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 (see [Version Control Ignores](#)).

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 my_project
```

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

2. Work on a virtual environment:

```
$ workon my_project
```

 v: latest ▾

Alternatively, you can make a project, which creates the virtual environment, and also a project directory inside `$WORKON_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](#) automatically 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
```