An introduction to Python virtual environments

By Tibs / Tony Ibbs

Presented at CamPUG, on 2020-02-04 (Tuesday 4th February 2020).

Written using reStructuredText.

Converted to PDF slides using pandoc and beamer.

Source and extended notes at https://github.com/tibs/venv-intro

Where we're going today

- A discussion of *why*
- We all work through using `python3 -m venv`
- Wrappers around virtual environments
- IDEs
- Other stuff

Why do we need virtual environments?

Why do we need virtual environments?

My reasons:

- Shouldn't alter the system Python
- *Can't* alter the Python provided
- You need packages with conflicting dependencies
- Being explicit about dependencies of a project
- Being tidy

and maybe a bit of help when upgrading Python (and "losing" installed packages)

Unix, Windows, and so on

When I say "unix" I mean Linux, the BSDs and also Mac OS.

- I assume your shell is something like bash, zsh or even fish.

When I say "windows" I mean Windows 10.

- I assume your shell is CMD.EXE or Powershell.

If you're using the Linux subsystem on Windows, then I think you're "unix".

First, the official way

Since Python 3.3, it has been possible to create virtual environmens with just Python itself.

So that's what we'll look at first.

Somewhere to work

```
tibs ~$ mkdir temp
tibs ~$ cd temp
tibs ~/temp$
```

Check what Python(s) we have available

```
tibs ~/temp$ python --version
Python 2.7.17
tibs ~/temp$ python3 --version
Python 3.7.6
```

and being specific:

```
tibs ~/temp$ which python
/usr/local/bin/python
tibs ~/temp$ which python3
/usr/local/bin/python3
```

Create a virtual environment

```
tibs ~/temp$ python3 -m venv venv
```

python3 -m venv venv

`python3` is the Python I want to use to create the new virtual environment. If I wanted to be specific about exactly which Python I wanted, and assuming I've got them both installed, I could do:

```
$ python3.6 -m venv venv
```

or:

```
$ python3.7.1 -m venv venv
```

`python3 -m venv venv`

`-m venv` tells Python to load the `venv` module and run it.

The Python standard library comes with several modules that you can run in this way. They all end with code like:

```
if __name__ == '__main__':
    do_something()
```

In the case of `venv.py`, that "do something" is to create the setup for a virtual environment for this particular Python.

`python3 -m venv` **venv**

That last `venv` is the name of the directory to create which will hold the "workings" of the virtual environment.

And that got us

```
tibs ~/temp$ ls -F
venv/
```

Activating the virtual environments

Creating the directory doesn't put into the virtual environment.

We need to activate it:

- Bash shell: `source venv/bin/activate`
- CShell: `source venv/bin/activate.csh`
- Fish shell: `source venv/bin/activate.fish`
- Windows CMD.EXE: `venv\Scripts\activate.bat`
- Windows Powershell: `venv\Scripts\Activate.ps1`

```
tibs ~/temp$ source venv/bin/activate.fish
(venv) tibs ~/temp$
```

Note how the prompt has changed.

What has changed? (apart from the prompt)

```
(venv) tibs ~/temp$ python3 --version
Python 3.7.6

(venv) tibs ~/temp$ python --version
Python 3.7.6
```

```
(venv) tibs ~/temp$ which python3
/Users/tibs/temp/venv/bin/python3

(venv) tibs ~/temp$ which python
/Users/tibs/temp/venv/bin/python
```

Let's install requests

```
(venv) tibs ~/temp$ pip install requests
Collecting requests
...
Collecting idna<2.9,>=2.5 (from requests)
...
Collecting urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 (from re
...
Collecting certifi>=2017.4.17 (from requests)
...
Collecting chardet<3.1.0,>=3.0.2 (from requests)
...
Installing collected packages: idna, urllib3, certifi, char
Successfully installed certifi-2019.11.28 chardet-3.0.4 idn
WARNING: You are using pip version 19.2.3, however version
You should consider upgrading via the 'pip install --upgrad
```

That last bit said:

```
WARNING: You are using pip version 19.2.3, however
    version 20.0.2 is available.
You should consider upgrading via the 'pip install
    --upgrade pip' command.
```

Let's upgrade pip

```
(venv) tibs ~/temp$ pip install --upgrade pip
Collecting pip
...
Installing collected packages: pip
  Found existing installation: pip 19.2.3
    Uninstalling pip-19.2.3:
      Successfully uninstalled pip-19.2.3
Successfully installed pip-20.0.2
```

And now

```
(venv) tibs ~/temp$ python
Python 3.7.6 (default, Jan 28 2020, 22:16:20)
[Clang 11.0.0 (clang-1100.0.33.16)] on darwin
Type "help", "copyright", "credits" or "license" for more i
>>> import requests
>>>
```

```
(venv) tibs ~/temp$ pip --version
pip 20.0.2 from /Users/tibs/temp/venv/lib/python3.7/site-pa
```

Deactivating

```
(venv) tibs ~/temp$ deactivate
tibs ~/temp$
```

After deactivation

```
tibs ~/temp$ python --version
Python 2.7.17
```

```
tibs ~/temp$ pip --version
pip 19.3.1 from /usr/local/lib/python2.7/site-packages/pip
```

```
tibs ~/temp$ python3
Python 3.7.6 (default, Jan 28 2020, 22:16:20)
[Clang 11.0.0 (clang-1100.0.33.8)] on darwin
Type "help", "copyright", "credits" or "license" for more
>>> import requests
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'requests'
>>>
```

Let's look at the virtual environment directory

```
tibs ~/temp$ ls -F venv
bin/        include/    lib/        pyvenv.cfg
```

```
tibs ~/temp$ more venv/pyvenv.cfg
home = /usr/local/bin
include-system-site-packages = false
version = 3.7.6
```

The bin directory

```
tibs ~/temp$ ls -F venv/bin/
activate          easy_install*         pip3.7*
activate.csh      easy_install-3.7*     python@
activate.fish     pip*                  python3@
chardetect*       pip3*
```

```
tibs ~/temp$ ls -l venv/bin/python
lrwxr-xr-x  1 tibs  staff 7 19 Jan 16:50 venv/bin/python
                                          -> python3
tibs ~/temp$ ls -l venv/bin/python3
lrwxr-xr-x  1 tibs  staff 22 19 Jan 16:50 venv/bin/python3
                                  -> /usr/local/bin/python3
```

The lib directory just after creating a venv

```
tibs ~/temp$ ls -F venv2/lib/python3.7/site-packages/
__pycache__/                    pkg_resources/
easy_install.py                 setuptools/
pip/                            setuptools-41.2.0.dist-info
pip-19.2.3.dist-info/
```

The lib directory after installing requests

```
tibs ~/temp$ ls -F venv/lib/python3.7/site-packages
__pycache__/                        pip-20.0.2.dist-info/
certifi/                            pkg_resources/
certifi-2019.11.28.dist-info/       requests/
chardet/                            requests-2.22.0.dist-info/
chardet-3.0.4.dist-info/            setuptools/
easy_install.py                     setuptools-41.2.0.dist-info/
idna/                               urllib3/
idna-2.8.dist-info/                 urllib3-1.25.8.dist-info/
pip/
```

Finding out what is installed: 1

```
tibs ~/temp$ source venv/bin/activate.fish
```

```
(venv) tibs ~/temp$ pip freeze
certifi==2019.11.28
chardet==3.0.4
idna==2.8
requests==2.22.0
urllib3==1.25.8
```

Finding out what is installed: 2

```
tibs ~/temp$ source venv/bin/activate.fish
(venv) tibs ~/temp$ pip install pipdeptree
...
```

and then:

```
(venv) tibs ~/temp$ pipdeptree
pipdeptree==0.13.2
  - pip [required: >=6.0.0, installed: 20.0.2]
requests==2.22.0
  - certifi [required: >=2017.4.17, installed: 2019.11.28]
  - chardet [required: >=3.0.2,<3.1.0, installed: 3.0.4]
  - idna [required: >=2.5,<2.9, installed: 2.8]
  - urllib3 [required: >=1.21.1,<1.26,!=1.25.1,!=1.25.0, in
setuptools==41.2.0
```

Where to put the venv directory?

- "project" - one venv per directory tree

  At the top of the directory tree for this project

- "environment" - one venv per purpose

  Generally, somewhere central, with a name related to the purpose

python3 -m venv help

```
$ python3 -m venv --help
usage: venv [-h] [--system-site-packages] [--symlinks | --c
            [--upgrade] [--without-pip] [--prompt PROMPT]
            ENV_DIR [ENV_DIR ...]

Creates virtual Python environments in one or more target d

positional arguments:
  ENV_DIR               A directory to create the environme

optional arguments:
  -h, --help            show this help message and exit
  --system-site-packages
                        Give the virtual environment access
                        site-packages dir.
  --symlinks            Try to use symlinks rather than cop
                        are not the default for the platfor
  --copies              Try to use copies rather than symli
```

Wrappers to make it easier

- virtualenv
- virtualenvwrapper
- pipenv
- poetry
- conda

virtualenv

virtualenv is essentially where Python virtual envrironments all started.

If you want virtual environments for Python2 or early versions of Python 3, this is still the package to use.

virtualenv example: 1

```
tibs ~/temp$ virtualenv -p python3.7 VENV
Running virtualenv with interpreter /usr/local/bin/python3
Already using interpreter /usr/local/opt/python/bin/python3
Using base prefix '/usr/local/Cellar/python/3.7.6_1/Framew
New python executable in /Users/tibs/temp/VENV/bin/python3
Also creating executable in /Users/tibs/temp/VENV/bin/pytho
Installing setuptools, pip, wheel...
done.
```

virtualenv example: 2

```
tibs ~/temp$ ls -F VENV
bin/     include/ lib/
```

There is also a "hidden" file in there, a link:

```
tibs ~/temp$ ls -l VENV/.Python
lrwxr-xr-x  1 tibs  staff  80  1 Feb 16:43 VENV/.Python ->
```

Note that there is no `pyenv.cfg` file - that's a later invention.

virtualenvwrapper

virtualenvwrapper is a wrapper for virtualenv (well, it's in the name!) that aims to make it easier to use, by providing some extra commands.

Once you've installed it, there's a degree of manual setup, although it's reasonable clearly explained in the documentation.

Once you've set it up, it will:

1. Allow you to keep all of your virtual environment directories under one single directory - typically something like `$HOME/.virtualenvs`.
2. Provide a new command, `mkvirtualenv` to create new virtual environments.
3. Provide a command `workon` that lets you change to a (different) virtual environment.

virtualenvwrapper example

```
tibs ~/temp$ mkvirtualenv use-requests
```

which would create me a virtual environment directory:

`/Users/tibs/.virtualenvs/use-requests`

To use it:

```
tibs ~/temp$ workon use-requests
(use-requests) tibs ~/temp$
```

pipenv

pipenv aims to make using virtual environments easier, but also to help with package management for a project as well.

pipenv example

```
tibs ~/temp$ cd ~/temp
tibs ~/temp$ pipenv install --python 3.7
Creating a virtualenv for this project...
Pipfile: /Users/tibs/temp/Pipfile
Using /usr/local/bin/python3 (3.7.6) to create virtualenv.
? Creating virtual environment...Already using interpreter
Using base prefix '/usr/local/Cellar/python/3.7.6_1/Framew
New python executable in /Users/tibs/.local/share/virtualen
Also creating executable in /Users/tibs/.local/share/virtua
Installing setuptools, pip, wheel...
done.
Running virtualenv with interpreter /usr/local/bin/python3
```

pipenv example, continued

```
? Successfully created virtual environment!
Virtualenv location: /Users/tibs/.local/share/virtualenvs/t
Creating a Pipfile for this project...
Pipfile.lock not found, creating...
Locking [dev-packages] dependencies...
Locking [packages] dependencies...
Updated Pipfile.lock (a65489)!
Installing dependencies from Pipfile.lock (a65489)...
? |||||||||||||||||||||||||||||||| 0/0 - 00:00:00
To activate this project's virtualenv, run pipenv shell.
Alternatively, run a command inside the virtualenv with
pipenv run.
```

pipenv virtual environment

If we look in the virtual environment directory:

```
(temp) tibs ~/temp$ ls -aF \
            ~/.local/share/virtualenvs/temp--1EXmzEU/
./          .Python@  bin/      lib/
../         .project  include/
```

it is a virtualenv style virtual environment, not a `venv` style.

pipenv files

pipenv also creates two files in the current directory:

```
tibs ~/temp$ ls -F
Pipfile       Pipfile.lock
```

pipenv Pipfile

```
[[source]]
name = "pypi"
url = "https://pypi.org/simple"
verify_ssl = true

[dev-packages]

[packages]

[requires]
python_version = "3.7"
```

pipenv Pipfile.lock

```
{
  "_meta": {
    "hash": {
      "sha256": "7e7ef69da7248742e869378f8421880cf8f0017f96c
    },
    "pipfile-spec": 6,
    "requires": {
      "python_version": "3.7"
    },
    "sources": [
      {
        "name": "pypi",
        "url": "https://pypi.org/simple",
        "verify_ssl": true
      }
    ]
  },
  "default": {},
```

pipenv: entering the virtual environment

The normal way to use the virtual environment is then (as it suggests) to do:

```
tibs ~/temp$ pipenv shell
Launching subshell in virtual environment...
Welcome to fish, the friendly interactive shell
tibs ~/temp$  source /Users/tibs/.local/share/virtualenvs/t

(temp) tibs ~/temp$
```

This actually starts a new shell with the virtual environment enabled in it.

pipenv: installing requests

We use `pipenv install` (and not `pip`) to install new packages:

```
(temp) tibs ~/temp$ pipenv install requests
Installing requests...
Adding requests to Pipfile's [packages]...
? Installation Succeeded
Pipfile.lock (444a6d) out of date, updating to (a65489)...
Locking [dev-packages] dependencies...
Locking [packages] dependencies...
? Success!
Updated Pipfile.lock (444a6d)!
Installing dependencies from Pipfile.lock (444a6d)...
? |||||||||||||||||||||||||||||||| 5/5 - 00:00:00
```

pipenv: after installing requests

Now the `Pipfile` and `Pipfile.lock` have been updated - the `Pipfile` to:

```
[[source]]
name = "pypi"
url = "https://pypi.org/simple"
verify_ssl = true

[dev-packages]

[packages]
requests = "*"

[requires]
python_version = "3.7"
```

and the `Pipfile.lock` to something rather longer and more complicated, but which basically uniquely identifies the packages that were installed.

poetry

I built Poetry because I wanted a single tool to manage my Python projects from start to finish. I wanted something reliable and intuitive that the community could use and enjoy.
-- Sébastien Eustace

poetry new

If you want to create a new project, then the `poetry new` command will create the project directory and a sensible starting layout.

I don't really want to go quite that far (although actually it's a good idea in general), so I shall just use `poetry init` to get started.

poetry init

This takes the user through some questions to generate the `pyproject.toml` file that poetry requires:

```
tibs ~/temp$ poetry init

This command will guide you through creating your
  pyproject.toml config.

Package name [temp]:
Version [0.1.0]:
Description []:
Author [Tibs <tibs@tonyibbs.co.uk>, n to skip]:
License []:  MIT
Compatible Python versions [^3.7]:
```

poetry init, continued

```
Would you like to define your main dependencies
  interactively? (yes/no) [yes] no
Would you like to define your dev dependencies
  (require-dev) interactively (yes/no) [yes] no

Generated file

[tool.poetry]
name = "temp"
version = "0.1.0"
description = ""
authors = ["Tibs <tibs@tonyibbs.co.uk>"]
license = "MIT"
```

poetry init, continued some more

```
[tool.poetry.dependencies]
python = "^3.7"

[tool.poetry.dev-dependencies]

[build-system]
requires = ["poetry>=0.12"]
build-backend = "poetry.masonry.api"


Do you confirm generation? (yes/no) [yes]
```

pyproject.toml file

The pyproject.toml file is indeed as described:

```
[tool.poetry]
name = "temp"
version = "0.1.0"
description = ""
authors = ["Tibs <tibs@tonyibbs.co.uk>"]
license = "MIT"

[tool.poetry.dependencies]
python = "^3.7"

[tool.poetry.dev-dependencies]

[build-system]
requires = ["poetry>=0.12"]
build-backend = "poetry.masonry.api"
```

poetry install

```
tibs ~/temp$ poetry install
Creating virtualenv temp-PD0d5gaI-py3.7 in
    /Users/tibs/Library/Caches/pypoetry/virtualenvs
Updating dependencies
Resolving dependencies... (0.1s)

Writing lock file

No dependencies to install or update
```

Where the virtual environment directory goes is dependent on the operating system. On a Mac, `~/Library/Caches` is a fairly traditional sort of place.

poetry: the virtual environment directory

```
tibs ~/temp$ pushd /Users/tibs/Library/Caches/pypoetry
```

```
tibs ~/temp$ ls -aF virtualenvs/temp-PD0d5gaI-py3.7/
./          bin/        lib/        src/
../         include/    pyvenv.cfg
```

```
tibs ~/temp$ popd
```

which tells us we've created a (modern) `venv` virtual environment.

The name of the virtual environment includes our starting directory name, a hash, and the version of Python.

poetry: in the current directory

Meanwhile, in the current directory, we have:

```
tibs ~/temp$ ls -F
poetry.lock     pyproject.toml
```

The `pyproject.toml` hasn't changed, and the `poetry.lock` contains:

```
package = []

[metadata]
content-hash = "669741988c507fb04697bdb0c9077fa1b2342c356d
python-versions = "^3.7"

[metadata.files]
```

poetry: starting the virtual environment

We get into our virtual environment by starting a new shell using `poetry shell`:

```
tibs ~/temp$ poetry shell
Spawning shell within /Users/tibs/Library/Caches/pypoetry/v
Welcome to fish, the friendly interactive shell
tibs ~/temp$ source /Users/tibs/Library/Caches/pypoetry/vi
(temp-PD0d5gaI-py3.7) tibs ~/temp$
```

poetry add requests

```
(temp-PD0d5gaI-py3.7) tibs ~/temp$ poetry add requests
Using version ^2.22.0 for requests

Updating dependencies
Resolving dependencies... (1.0s)

Writing lock file


Package operations: 0 installs, 5 updates, 0 removals

  - Updating certifi (2019.11.28 /usr/local/Cellar/poetry/1
  - Updating chardet (3.0.4 /usr/local/Cellar/poetry/1.0.3/
  - Updating idna (2.8 /usr/local/Cellar/poetry/1.0.3/libex
  - Updating urllib3 (1.25.8 /usr/local/Cellar/poetry/1.0.3
  - Updating requests (2.22.0 /usr/local/Cellar/poetry/1.0.
```

pyproject.toml after adding requests

```
[tool.poetry]
name = "temp"
version = "0.1.0"
description = ""
authors = ["Tibs <tibs@tonyibbs.co.uk>"]
license = "MIT"

[tool.poetry.dependencies]
python = "^3.7"
requests = "^2.22.0"

[tool.poetry.dev-dependencies]

[build-system]
requires = ["poetry>=0.12"]
build-backend = "poetry.masonry.api"
```

poetry.lock after adding requests

...specifies the dependencies for `requests`, the exact versions of packages, and various other things.

conda

conda comes out of the Anaconda project, which started as a means of providing easy installation of scientific/numeric Python on Windows. It's now a lot more than that, but still aimed at the scientific / big data worlds.

- if you've got anaconda, you're already using this - so just keep doing so
- support for many different languages
- there is miniconda which is `conda` without *all* of the packages - this is closer to just using `pip`.

Summary: Which wrapper to use?

You don't *have* to use any of these

- virtualenv - if you must work with Python < 3.3
- virtualenvwrapper - nice if you're using virtualenv
- pipenv
- poetry - if you want something that manages more project details
- conda - if you're already using it or Anaconda

IDEs

This is going to be a brief summary, since I don't use any of these.

VS Code

VS Code (Visual Studio Code) supports Python virtual environments.

If you are editing a Python file, the Python interpreter being used is shown at the bottom left of the screen.

The VS Code documentation explains how it decides where to look.

If you are working with a VS Code "workspace", then it will automatically find a `.venv` directory in that workspace.

Also, VS Code understands the locations that virtualenvwrapper and pipenv use to store virtual environments, and its simple to use with poetry as well.

PyCharm

PyCharm: always thinks in terms of "projects".

Configure a virtual environment explains how to use and create virtual environments in PyCharm, and Conda virtual environment explains how to use conda virtual environments.

When setting up the Python interpreter for use in a PyCharm project, you need to specify the full path to the Python executable. So, for instance:

`~/tibs/temp/venv/bin/python3`

Atom

There appear to be multiple packages that support virtual environments for Python in atom. I'm assuimg that if you use atom you know your way around the package system.

Jupyter notebook

Jupyter notebook isn't really an IDE, but virtual environments are still relevant when using it.

The simplest thing to do is to create your virtual environment, then install jupyter notebook within it. When you run that jupyter notebook, it will automatically use the Python it was installed for.

For instance:

```
$ source .venv/bin/activate
$ pip install jupyter
$ jupyter notebook
```

It *is* possible to run multiple Python "backends" for Jupyter notebook, but that's a bit beyond this document.

Awkward questions

- What happens if I activate a virtual environment while I've got one activated?
- Can I (deliberately) create a virtual environment that depends on another?
- How do I stop `pip` from installing outside a virtual environment?
- Do I *need* to activate the virtual environment?
- What happens when I upgrade Python?

What happens if I activate a virtual environment while I've got one activated?

The new activation will "take over".

In particular, the old virtual environment binary directory is removed from the PATH and the new one is added instead.

However, I don't know if anything *promises* that this will work, so it's perhaps best not to rely on it.

Can I (deliberately) create a virtual environment that depends on another?

Yes. Simply do `python -m venv <name>` inside an already activated virtual environment.

If you inspect the `bin/python` entry (on unix, at least) you will see it links to the Python from the earlier virtual environment.

Why you might want to do that, and how useful it might be, is another discussion.

How do I stop `pip` from installing outside a virtual environment?

It's not very well documented, but the simplest way to do this is to set the environment variable:

```
PIP_REQUIRE_VIRTUALENV=true
```

For instance, in your `.bashrc` you would add:

```
export PIP_REQUIRE_VIRTUALENV=true
```

and that would then take effect when you open a new shell.

When that is set, any attempt to use `pip install <something>` outside a virtual environment will give the error message:

```
ERROR: Could not find an activated virtualenv (required).
```

Do I *need* to activate the virtual environment?

Well, actually, no. It just makes things more convenient. If you run the Python in the virtual environment `bin` directory (`Scripts` for Wndows) explicitly, then that Python will "look around itself" and use the virtual environment.

So:

```
(venv) tibs ~/temp$ deactivate
tibs ~/temp$ venv/bin/python
Python 3.7.6 (default, Jan 28 2020, 22:16:20)
[Clang 11.0.0 (clang-1100.0.33.16)] on darwin
Type "help", "copyright", "credits" or "license" for more i
>>> import requests
>>>
```

That *also* means that if you install a Python program to the virtual environment `bin` directory, and run it directly (using its full path) then it too will know what environment to use, without your needing to activate the virtual environment.

For example:

```
tibs ~/temp$ python3 -m venv pydep
tibs ~/temp$ source pydep/bin/activate.fish
(pydep) tibs ~/temp$ pip install pipdeptree
Collecting pipdeptree
  Using cached https://files.pythonhosted.org/packages/12/6
Requirement already satisfied: pip>=6.0.0 in ./pydep/lib/py
Installing collected packages: pipdeptree
Successfully installed pipdeptree-0.13.2
(pydep) tibs ~/temp$ ls pydep/bin/pipdeptree
pydep/bin/pipdeptree
(pydep) tibs ~/temp$ deactivate
tibs ~/temp$ pydep/bin/pipdeptree --version
0.13.2
```

What happens when I upgrade Python?

That is, if the older Python "disappears" (as will generally happen with a `homebrew` upgrade on a Mac, for instance), do my virtual environments just stop working?

Well, generally, yes, but...

- pipenv: edit the Pipfile and change the version of Python, and then:

  ```
  pipenv --rm
  pipenv shell
  ```

- poetry: edit the pyproject.toml and change the version of Python, and then:

  ```
  poetry env remove python3.7
  poetry shell
  ```

End of *awkward* questions

The venv directory and version control systems

Broadly, don't commit the `venv` directory to your version control system. It doesn't contain anything portable (by definition).

The `--system-site-packages` switch

Normally, when I create a new virtual environment, it starts without anything installed (except `pip` and other basic infrastructure). So if the Python I used to create the virtual environment (the `python3` in `python3 -m venv`) had (for instance) `docutils` installed, the new virtual environment would not.

The `--system-site-packages` switch lets the new virtual environment "see" the packages in the original Python.

```
tibs ~/temp$ python3 -m venv secondary
tibs ~/temp$ source secondary/bin/activate.fish
(secondary) tibs ~/temp$ python
Python 3.7.6 (default, Jan 28 2020, 22:16:20)
[Clang 11.0.0 (clang-1100.0.33.16)] on darwin
Type "help", "copyright", "credits" or "license" for more
>>> import docutils
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'docutils'
>>>
(secondary) tibs ~/temp$ deactivate
```

```
tibs ~/temp$ python3 -m venv tertiary \
                        --system-site-packages
tibs ~/temp$ cat tertiary/pyvenv.cfg
home = /usr/local/bin
include-system-site-packages = true
version = 3.7.6
tibs ~/temp$ source tertiary/bin/activate.fish
(tertiary) tibs ~/temp$ python
Python 3.7.6 (default, Jan 28 2020, 22:16:20)
[Clang 11.0.0 (clang-1100.0.33.16)] on darwin
Type "help", "copyright", "credits" or "license" for more i
>>> import docutils
>>>
```

Multiple Pythons

Sometimes you need more than one version of Python - for instance, to test that a new version of Python is still compatible with existing code.

System package managers cannot always help with this - they typically only support a subset of the possible versions (homebrew on the Mac supports one Python per major version), and it can take some time for a new version to be provided (particularly a problem with some enterprise linuxes).

The solution is to use pyenv, which makes it easy to build Python at different versions.

(For Windows, you may want to look at pyenv-win instead)

Remember that this is *not* the same as virtual environments, but is complementary.

Some other tools

- venv_manager is intended for bash and zsh users, and detects and activates virtual environments as you `cd` into the directories that contain them (by default it looks for `.venv` directories).
- direnv is a more powerful tool that takes actions when you `cd` into a directory, and it too can be used to activate virtual environments. I confess that its documentation intimidates me.
- upm is a "universal package manager", which is meant to act as a consistent front end (command line tool) for various different programming languages. For Python it wraps poetry.
- DepHell is a project management tool for Python that is meant to be an all-in-one solution that can (for instance) work with pip, pipenv and poetry. If you're needing to convert beween tools, or work with multiple tools, it may be a good solution.

Fin

Written using reStructuredText.

Converted to PDF slides using pandoc and beamer.

Source and extended notes at https://github.com/tibs/venv-intro