Python Virtual Environments

–= A gentle introduction for Linux users =–

Cedric Bhihe

Last update: Nov. 2020

Foreword

This tutorial can be traced back to two posts: the first is blog post[1] by Bartek Skorulski, the second is a 2017 post by @Flimm on the stackexchange.com forum.

In this tutorial I focus on the environment commonly seen by Arch Linux users, and by extension by all Linux users whose distribution runs on the system and service initialization manager[2]: systemd. This tutorial goes beyond what is usually seen on the Internet, mostly disjoint commentaries, which many times can truly be understood only by people who already are experienced Python virtual environment practitioners. It addresses the down-to-earth needs of both first-time users and users who want to get a sense out of the current Python virtual environment landscape for Linux, as of mid 2020.

So what is a Python virtual environment and what could motivate the creation of virtual environments for Python? I answer those two questions superficially in this introductory section and chose to dedicate the rest of the document to how one may go about creating and managing virtual environments.

If you use Python for anything, either professionally or at home, chances are you already ran into snags related to the need for different versions of the same Python module(s). If you have not yet contended with such predicament, the premise to this tutorial is that sooner or later you will. This tutorial is a step-by-step guide on how to overcome such a difficulty by using Python virtual environments. There are several ways to do so and you may choose the one that appears more practical to you.

At its root the issue for any programmer is that different versions of any Python module normally cannot coexist in a single runtime environment (RTE) and name space. If you did not already tweak your python setup, you may at most have the latest versions of Python2 (v. 2.7.16) and some version of Python3 installed side by side in your environment. But what if you also needed versions (say 3.5.4 ***and*** 3.8.0 ***and*** 2.7.1 ***and*** 2.5.0) for a specific project involving older libraries ? Forcefully installing any older Python version on your platform would normally require you to rebuild older packages on a massive scale, as well as to roll back your system's module versions and many of its applications. It ca n be done but it would be extraordinarily complicated, not least because it would be very time-consuming and prone to breakage. As for installing several such versions side by side, simply forget it. So how does one remain practical ? One way is to embrace the concept of *Python virtual environments*.

A virtual environment is an isolated Python environment, in a specific directory. In it you can install and execute a Python version of your choice, independently of other Python virtual environments already residing on your platform. The relevant Arch wiki explains:

---

1 Data Scientist at Telefónica Alpha, Barcelona, Spain

2 A simplified overview of the entire Linux boot and startup process is: (i) At host's power-up, the BIOS does minimal hardware tests and initialization. It then hands control over to the boot loader. (ii) The boot loader calls the kernel. (iii) The kernel loads an initial RAM disk that loads the system drives and then looks for the root file system. (iv) After kernel set up, *systemd* initialization starts. (v) Finally *systemd* takes over and continues to mount the host's file systems and start services.

> *A virtual environment is a directory in which some binaries and shell scripts are installed. The binaries include python for executing scripts and pip for installing other modules within the environment. There are also shell scripts (at least one for bash, ...) to activate the environment. Essentially, a virtual environment mimics a full system install of Python and all of the desired modules without interfering with any system on which the application might run.*

In this tutorial we will use the abbreviation VE, when referring to the terms “virtual environment”.

Table of Contents

1. Flavors of VEs and ways to set them up

In this tutorial only 2 closely related VE flavors are described: '`venv`' and '`virtualenv`'. In addition I introduce a handy plugin used by many: '`virtualenvwrapper`'. Finally we will see three ways to go about setting up virtual environments on a Linux OS:

- **`venv`**

The **Venv** module's native capability is already built-in in Linux system running Python v3.3+. It is very lightweight and you need not install any new package. It allows testing your code against your OS' Python version, in a way that isolates the installation of any Python packages and their dependencies from those of your system.

- **`virtualenv`**

Virtualenv serves the same purpose as Venv, but has more features (see a comparison here). The corresponding Arch linux users' package is named `python-virtualenv`. It continues to be more popular than Venv. This is because Virtualenv (unlike Venv) supports both Python 2 and 3.

- **`pyenv`**

Pyenv is also used to isolate Python versions. To Virtualenv users (package: '`python-virtualenv`'), the idea of a Python version local to a directory might seem familiar. A *local* Python created from *`pyenv`* is also a virtual environment. The main difference is that Pyenv actually copies an entire Python installation every time you install a new Python version. In contrast, Virtualenv (not described in this Tutorial) makes use of symbolic links to decrease the size of the virtual environment.

Testing against the *current* Python versions 2 and 3 in a sandbox would only require Virtualenv. However testing against any sand-boxed Python versions (e.g. 2.7, 3.6, 3.7 and 3.9), requires a practical way to switch between them. Meet Pyenv ! And if you were not confused yet, Pyenv boasts *two main plugins*, which extend its command set:

- ***Pyenv-virtualenv*** (under the package name `pyenv-virtualenv`)

It integrates nicely in Pyenv and makes possible creating and handling native Pyenv's VEs.

- ***Virtualenvwrapper*** (under the package name `pyenv-virtualenvwrapper`)

It helps interacting with the plugin Virtualenvwrapper itself.

We will only describe the first of the two plugins: Pyenv-virtualenv.

1-1. 'venv'

As already mentioned, `venv` is a native capability of Python v3.3+. You can run it using the command

```
$ /usr/bin/python3 -m venv
```

(although for some reason some distros separate it out into a separate distro package, such as '`python3-venv`' on Ubuntu/Debian).

1-1-1. Virtual environment activation

To set it up along with `pip` (included by default since Python 3.4+), just issue:

```
$ python -m venv [--system-site-packages] --prompt VENV <dir1> \
[<dir2> [<dir3> […]]]
```

where:

- `-m venv` will *create* the VE
- `--system-site-packages` optionally gives the VE access to the system site packages
- `--prompt VENV` will modify the prompt by prepending `(VENV)` to it, so you know when you operate in an activated VE or not.
- `<dir1>`, and optional `<dir2>, ...` are fully qualified target directories, created if they do not exist already. Inside each one of them them, a VE will have its own Python binary, *which matches the version of the binary used to create the environment*. This is a crucial restriction. If the Python binary of your calling environment has version 3.7.x, so will the Python binary of your VE under *`venv`*. Within that restriction, each *`venv`* VE can have its own independent set of installed Python packages.

The created **`pyvenv.cfg`** file also includes the `include-system-site-packages` key, set to `true` if `venv` is run with the `--system-site-packages` option, `false` otherwise.

Unless the `--without-pip` option is given, `ensurepip` will be invoked by default in order to bootstrap `pip` into the VE.
If the `--without-pip` option was given, but `pip` needs to be installed *a posteriori*, you can always issue:

```
$ python -m ensurepip -upgrade
```

The `--upgrade` option will ensure that the version of `pip` installed is at least as new as the existing one if any.
To know all the available options, issue:

```
$ python -m venv -h
```

To activate and desactivate the Python VE, issue:

```
<pwd> $ source <dir>/bin/activate
(VENV) <dir> $    do your thing here...
(VENV) <dir> $ desactivate
```

where `<dir>` is any previously specified target directory where the VE is to be deployed.

1-1-2. Caveat: changes in executable paths

Quoting for the Python3 docs:

> *When a VE is active (i.e., the VE's Python interpreter is running), the attributes* `sys.prefix` *and* `sys.exec_prefix` *point to the base directory of the VE, whereas* `sys.base_prefix` *and* `sys.base_exec_prefix` *point to the non-VE Python installation which was used to create the VE. If a VE is not active, then* `sys.prefix` *is the same as* `sys.base_prefix` *and* `sys.exec_prefix` *is the same as* `sys.base_exec_prefix` *(they all point to a non-VE Python installation).*
>
> *When a VE is active, any options that change the installation path will be ignored from all* `distutils` *configuration files to prevent projects being inadvertently installed outside of the VE.*
>
> *When working in a command shell, users can make a VE active by running an* `activate` *script in the VE's directory (the precise filename is shell-dependent). This prepends the VE's directory for executables to the* `PATH` *environment variable for the running shell. There should be no need in other circumstances to activate a VE—scripts installed into VEs have a "shebang" line which points to the VE's Python interpreter. This means that the script will run with that interpreter regardless of the value of* `PATH`.

An immediate consequence of the above mentioned changes in path variables manifests itself, for example, when running a Jupyter notebook. There are two possibilities to run such a notebook:

a) It is run from the global (not isolated, non virtual) environment:

In such a case its python3 kernel will be based on `/usr/bin/python3`. Hence all package imported in the notebook should be installed in that global `usr/bin/python3` environment prior to importing them in the notebook.

To do so, first install `pip3` globally:

```
$ /usr/bin/python3 -m ensurepip --user --upgrade
```

followed by, e.g. for the 'seaborn' package:

```
$ /usr/bin/python3 -m pip install seaborn
```

Only then can `'seaborn'` be imported from a Jupyter notebook.

b) it is run explicitly from the activated python VE
In that case we revert to a normal package installation path.

```
(VENV) $ python3 -m pip install seaborn
```

1-1-3. Other limitations

As already mentioned the functionalities of `venv` are a subset of what `virtualenv` (described in the next subsection) does. That subset of tools was integrated upstream into the standard libraries under the `venv module`. Note however that the `venv` module does not offer all features of `virtualenv`. For instance it is not:

- able to create bootstrap scripts,
- able to create VEs for Python versions other than that of the host's base Python,
- relocatable,
- ...

For all those reason, Python3.3+'s `venv module` is recommended for projects that no longer need to support Python2 and just require straightforward isolated environments, simply based on the host's Python version.

1-2. 'virtualenv'

On many Linux distros, Ian Bicking's `virtualenv` along with its plugin, `virtualenvwrapper`, described in subsection 1.2.7, are used extensively, for both development and deployment.

While `virtualenv` is a tool to create VEs, `virtualenvwrapper` is a plugin, i.e. a set of extensions for `virtualenv`. Unlike `venv`, `virtualenv` creates VEs where libraries are not shared with those of other VEs. Optionally they also can be denied access to the globally installed libraries.

1-2-1. Installation

```
$ sudo pacman -Syu python-virtualenv
```

after which *man* pages may be consulted in the usual manner:

```
$ man virtualenv
```

1-2-2. Usage

Quoting largely from the Arch Linux documentation, *virtualenv* has one basic command:

```
$ virtualenv [--python=PYTHON_EXE [options]] <dir>
```

where `<dir>` is a fully qualified directory, which is to house the new Python's VE. `<dir>` will also be the new prompt's default prefix, the which can be modified easily with the *--prompt* option.

The newly created VE will be placed in `<dir>`. By default it will not rely on site-packages. It is modifiable via many options and has the following effects:

- `<dir>/lib/` and `<dir>/include/` are created, containing supporting library files for a new Python's VE. Packages installed in this VE will live under `<dir>/lib/pythonX.X/site-packages/`.
- `<dir>/bin/` is where executable binary files live, among them a new Python binary, whose version depends on the option `--python=PYTHON_EXE`. Thus, running a script with shebang
  `#!<dir>/bin/python`
  would run that script under this very virtualenv's python's binary version.

1-2-3. 'virtualenv' command options

You may modify the build of your VE with options.

- Options `--no-pip` and `–no-setuptools`
  When used, preclude installation of the two crucial packages: pip and setuptools. Otherwise those are installed by default, which allows other packages to be further installed in the VE. This associated pip can be run with `<dir>/bin/pip`. Thus the Python version in your new VE is effectively isolated from the Python used to create it.
- Option `-p PYTHON_EXE`, or equivalently `--python=PYTHON_EXE`
  With this you may specify the Python interpreter to use to create the new environment, e.g.:
  `-p python2.5`
  or equivalently
  `–python=python2.5`.
  In absence of that option, the default interpreter is that with which `virtualenv` was installed. As of 2017, it would most likely be `/usr/bin/python`, symbolic-linked to the system's version of Python 3.x.
- Option `--system-site-packages`
  That makes your VE inherit packages from wherever your global site-packages directory is located, e.g.: `/usr/lib/python2.7/site-packages`.

  This can be used if you have control over the global site-packages directory, and you want to depend on the packages there. If you want complete isolation from the global system, in particular if you want to prevent a system update to wreak havoc in your carefully isolated VE's packages and dependencies' versions, do **not** use this flag.

  If you need to change this option after creating a virtual environment, you can turn if off or on. You do so by respectively adding or deleting the file `no-global-site-packages.txt` to or from the virtual environments directory: `<dir>/lib/python3.7/`
- Option `--prompt MY_PROMPT`
  modifies the activated VE's prompt's prefix.
  `MY_PROMPT` is usually much shorter and more meaningful than its default `<dir>` value.

- Other slightly less common but handy options include:
  `--relocatable`
  `--extra-search-dir=/path/to/distributions`

They are briefly described in the man page as well as in the online *virtualenv* user guide.

1-2-4. Activating the virtual environment

In a newly created VE there will also be an `activate` shell script. The exact `activate` file location may vary with the shell being used (*csh, fish, ...)*. On POSIX and POSIX-like systems, and in particular under for *bash*, *zsh* and *dash,* the activation script resides in `<dir>/bin/`, so you can run:

```
$ source <dir>/bin/activate
```

For some shells (e.g. the original Bourne Shell), or `source` does not exist, you may need to use the '`.`' cmd. Cmd `source` is purely a convenience in that it makes your shell environment change *in-place*. All that activation of the VE does is to change the `PATH` variable, so its first entry becomes the VE's `<dir>/bin/` directory.

The `activate` script will also modify your shell prompt to indicate which environment is currently active. To disable that behaviour, see `VIRTUAL_ENV_DISABLE_PROMPT`.

1-2-5. Not activating the virtual environment

If you choose NOT to activate your VE, and instead run a script or the python interpreter directly from the VE's directory, `<dir>/bin/`, e.g. by doing:

`<dir>/bin/pip` or
`<dir>/bin/python-script.py`

then `sys.path` will automatically be set to use the Python libraries associated with the VE. Bear in mind however, that in that case, unlike what we just saw in the previous section with activation scripts, the environment variables `PATH` and `VIRTUAL_ENV` will *not* be modified. This means that if for example a Python script uses `subprocess` to run another Python script, for instance via the shebang `#!/usr/bin/env python`, the second script *may neither be executed with the same Python binary as the first*, nor see the same libraries. To prevent this from happening your first script will need to modify the environment variables in the same manner as the activation scripts, before the second script is executed.

1-2-6. Deactivating and removing the virtual environment

To undo these changes to PATH (and prompt), and to return to your normal global Python's default environment just run:

```
(<dir>) $ deactivate
```

Removing a virtual environment is simply done by deactivating it and deleting the environment folder with all its contents:

```
(<dir>)$ deactivate
$ cd .. && \rm -r <dir>
```

1-2-7. 'virtualenvwrapper' (plugin)

This MIT licensed plugin by Doug Hellmann is a set of extensions to `virtualenv` (see docs). It gives you commands such as:

```
mkvirtualenv
mktmpenv
lsvirtualenv
lssitepackages
showvirtualenv
rmvirtualenv
allvirtualenv
mkproject
workon
```

and more... The last cited one above is also the most used and allow the user to switch between different `virtualenv` directories and their specific VEs. This tool is especially useful if you usually work with multiple `virtualenv` directories simultaneously, i.e. when your workflow makes you jump from one to the other.

1-2-7-1. Features

The extensions include wrappers for creating and deleting VEs and otherwise managing your development workflow. It makes it easier to work on more than one project at a time, without introducing conflicts in their dependencies.

1. Organizes all of your VEs in one place.
2. Wrappers for managing your VEs (create, delete, copy).
3. Use a single cmd to switch between VEs.
4. Tab completion for cmds that take a VE as argument.
5. User-configurable hooks for all operations (see Per-User Customization).
6. Plugin system for more sharable extensions (see Extending virtualenvwrapper).

1-2-7-2. Setup

`virtualenvwrapper` should be installed in the same global-site packages area as `virtualenv`, in the absence of any active VE, so the same release is shared by all Python environment on your platform.

```
$ sudo pacman -Syu python-virtualenvwrapper
```

```
or      $ /usr/bin/python -m pip install virtualenvwrapper
```

Include the following two lines in `~/.profile`:

```
export WORKON_HOME=$HOME/.virtualenvs
export PROJECT_HOME=$HOME/Projects
mkdir -p "$WORKON_HOME"
```

All relevant environmental variables are defined in `~/.profile` (`$WORKON_HOME`, `$PROJECT_HOME`, `$VIRTUAL_ENV`, etc.)

Include the following in `~/.bashrc`:

```
#source /usr/bin/virtualenvwrapper.sh      # loading upon shell creation
source /usr/bin/virtualenvwrapper_lazy.sh  # lazy loading
```

The commented out line above may induce slowness every time sub-shells are spawned. Favor lazy invocation per the next line to only load plugin shell functions when they are first needed.

1-2-7-3. Usage

At startup or when spawning a new shell, `virtualenvwrapper.sh` or `virtualenvwrapper_lazy.sh` finds the first Python and `virtualenv` programs on the `$PATH` and remembers them for later use. This eliminates any conflict as the `$PATH` changes, enabling interpreters inside VEs where `virtualenvwrapper` is not installed or even where different versions of `virtualenv` are installed. To make that behavior possible, it is important for the `$PATH` to be set **before** sourcing `virtualenvwrapper.sh` or `virtualenvwrapper_lazy.sh`.

Make VE `env1` inside `${WORKON_HOME}/`

```
$ mkvirtualenv env1       # creates new VE in $WORKON_HOME and switches to it
(env1)$ ls $WORKON_HOME
env1 hook.log
```

The general syntax is:

```
$ mkvirtualenv [-a project_path] [-i package] [-r requirements] [virtualenv options] ENVNAME
```

Install a package inside `${WORKON_HOME}/env1`

```
(env1)$ pip install django
(env1)$ lssitepackages
Django-1.1.1-py2.6.egg-info    easy-install.pth       setuptools-0.6.10-py2.6.egg
setuptools-0.6.10-py2.6.egg    pip-0.6.3-py2.6.egg    setuptools.pth
django
```

There is no limitation (beyond disc space and memory) as to the number of VEs that can be created:

```
$ mkvirtualenv env2            # makes and switches to newly created VE
(env2)$ ls $WORKON_HOME
env1 env2 hook.log
```

To switch between VEs:

```
(env2)$ workon env1
(env1)$ pwd
/Users/dhellmann/Envs/env1
(env1)$
```

The `workon` command includes tab completion for the environment names. It also invokes customization scripts as a VE is activated or deactivated (see Per-User Customization).

To apply a general command to all already defined VEs under $WORKON_HOME, for instance to update `pip`:

```
$ allvirtualenv python -m pip install -U pip
```

`postactivate` and **`postmkvirtualenv`**, **`postdeactivate`**, **`predeactivate`**, and others are special files sourced before or after a new VE is either activated, created, deactivated or on the verge of being so. They permit the automation of repetitive tasks, such as the installation of commonly used tools, the unaliasing of commands reserved for when a VE is active, etc…

Two short examples are provided:

```
(env1)$ echo 'cd $VIRTUAL_ENV' >> $WORKON_HOME/postactivate
(env1)$ workon env2            # the previous cmd
(env2)$ pwd
/Users/dhellmann/Envs/env2

(env2) $ echo   'pip install sphinx' >> $WORKON_HOME/postmkvirtualenv
(env2)$ mkvirtualenv env3
New python executable in env3/bin/python
Installing setuptools.....................
.........................................
Downloading/unpacking sphinx
.........................................
Successfully installed docutils Jinja2 Pygments sphinx
(env3)$ which sphinx-build
/Users/dhellmann/Envs/env3/bin/sphinx-build
```

More details on cmd shortcuts are available at https://virtualenvwrapper.readthedocs.io/en/latest and https://virtualenvwrapper.readthedocs.io/en/latest/command_ref.html.

1-3. 'pyenv'

1-3-1. Working principle

1-3-1-1. Scope

`pyenv` was previously known as `pythonbrew` and can be found on Github.

When used in conjunction with `virtualenvwrapper` it is one of the most practical way of installing and dealing with different Python versions and their configurations on a Linux platform. Both the Pyenv-virtualenv Virtualenvwrapper plugins extend Pyenv's set of cmds. Both plugins are available either in AUR or on Github.

`pyenv` lets you:

- change the global Python version,
- install multiple Python versions,
- set directory (project)-specific Python versions, and
- generally create / manage multiple Python VEs.

All this is done on *nix-style machines (Linux, Unix and OS X) without depending on Python itself. It also works with regular users privileges. There is no need for any privilege escalation to super-user level with *`sudo`*.

Once activated, it prefixes the `$PATH` environment variable with `$HOME/.pyenv/shims`, where special files matching the Python commands (*`python`*, *`pip`*) are found. Those are not copies of the Python-shipped commands; they are special scripts that decide on the fly which version of Python to run based on the `PYENV_VERSION` environment variable, or the `.python-version` file, or the `$HOME/.pyenv/version` file. `pyenv` also makes the process of downloading and installing multiple Python versions easier, using the cmd:

```
$ pyenv install <python_version_number>
```

In a nutshell *`pyenv`* works by inserting a ***directory of shims*** at the beginning of `$PATH`:

```
$ echo $PATH
/home/USER/.pyenv/shims:/usr/local/bin:/usr/bin:/bin
```

provided `$PYENV_ROOT` is equal to `/home/USER/.pyenv`.

Through a process called *rehashing*, `pyenv` maintains shims in that directory to match every Python cmd across all its installed versions. Shims are lightweight executables that simply pass your cmd along to `pyenv`. So with `pyenv` installed, when you run *`pip`*, your operating system will do the following:

- Search `$PATH` for an executable file named *`pip`*
- Find the `pyenv` shim named *`pip`* at the beginning of `$PATH`
- Run the shim named *`pip`*, which in turn passes the command along to `pyenv`

Setting a local project-specific Python version is possible by writing the version name to a `.python-version` file in the current VE-directory. That *local version* overrides the *global version*, and can be overridden itself by setting the `$PYENV_VERSION` environment variable or with the *`pyenv`* shell command.

1-3-1-2. Plugins: '`pyenv-virtualenv`' and '`pyenv-virtualenvwrapper`'

We already mentioned the two Pyenv plugins:

The first named '**`pyenv-virtualenv`**', only requires `pyenv-git` as a dependency. It comes with various features to help `pyenv` users manage VEs created by `virtualenv` or `anaconda`. Because the `activate` script of those VEs changes the user's interactive shell's `$PATH`, it intercept `pyenv`'s shim style cmd execution hooks.

The second plugin is '**`pyenv-virtualenvwrapper`**'. It requires dependencies `pyenv` as well as `python-virtualenvwrapper` which in turn requires `python-virtualenv`. You may check the PKGBUILD file in the AUR repo for confirmation or for any possible future change in the dependencies.

1-3-2. Installation of 'pyenv' (pyenv-git)

Either
```
$ cd /path/to/your/local_AUR_builds
$ git clone https://aur.archlinux.org/pyenv-git.git
$ cd pyenv-git; makepkg -sric
```

or
```
$ git clone https://github.com/pyenv/pyenv.git ~/.pyenv
```

or, if `pyenv` is available in your usual package repository, e.g. for Archlinux with *`pacman`*:
```
$ sudo pacman -Syu pyenv
```

For Debian-like systems with `apt-get` and `curl`:
```
$ sudo apt-get install -y make build-essential git libssl-dev zlib1g-dev \
        libbz2-dev libreadline-dev libsqlite3-dev wget curl \
        llvm libncurses5-dev libncursesw5-dev xz-utils tk-dev
$ curl -L https://raw.githubusercontent.com/pyenv/pyenv-installer/master/bin/pyenv-installer | bash
```

To get help, issue the cmd:
```
$ pyenv help
```

1-3-3. Configuration and update of 'pyenv'

Add the following lines in `~/.profile`:
```
export PYENV_ROOT=${HOME}/.pyenv
export PATH=${PYENV_ROOT}/shims:${PYENV_ROOT}/bin:$PATH" # prepend PATH !!
export WORKON_HOME="$HOME/.virtualenvs"          # VEs' local repo
mkdir -p "$WORKON_HOME"
export PROJECT_HOME="${HOME}/path_to/project_directory"    # project directory
```

Add the following lines in `~/.bashrc:`
```
[ -n "$(command -v pyenv)" ] && eval "$(pyenv init -)"
pyenv global "$(/usr/bin/python --version | cut -d ' ' -f2 2>/dev/null)"
/usr/bin/echo "$(pyenv global)" >| "${PYENV_ROOT}"/version
source /usr/bin/virtualenvwrapper.sh
```
or
```
source /usr/bin/virtualenvwrapper_lazy.sh
```

Note that when *lazy-sourcing*, tab-completion of arguments to `virtualenvwrapper` commands (such as environment names) is not enabled until after the first command has been run. Also there seems to be a bug that causes the virtualenvwrapper plugin cmd *`workon`* to make your terminal session crash the plugin is invoked lazily.

`pyenv init -` loads extra commands into your shell. Here's what it actually does:

1. **Sets up your shims path.** This is the only requirement for pyenv to function properly. You can do this by hand by pre-pending `$(pyenv root)/shims` to PATH.
2. **Installs autocompletion.** Sourcing `$(pyenv root)/completions/pyenv.bash` will set that up.
3. **Rehashes shims.** From time to time you'll need to rebuild your shim files. Doing this on init makes sure everything is up to date. Instead you can run *`pyenv rehash`* manually.
4. **Installs the sh dispatcher.** This bit is also optional, but allows Pyenv and plugins to change variables in your current shell, making commands such as *`pyenv shell`* possible. The sh dispatcher doesn't do anything crazy such as overriding *`cd`* or hacking your shell prompt, but if for some reason you need *`pyenv`* to be a real script rather than a shell function, you can skip it safely.

To see exactly what happens under the hood, just look at the standard output of *`pyenv init -`*.

Back in the console, check the list of available Python versions, pick a couple (e.g. `UVW, [XYZ]`) and install them:

```
$ pyenv root
/home/$USER/.pyenv
$ cd $PYENV_ROOT/plugins/python-build/../.. && git pull && cd -
$ pyenv install -l
[… list of available python versions]
$ pyenv install UVW [XYZ]        # where XYZ may be “3.7.3” or “stackless-2.7.5” or …
$ pyenv versions                 # check list of Python version(s) installed in $PYENV_ROOT
```

After installing at least one version of Python, you can set it globally with a shell snippet in `~/.bashrc`. If not set `pyenv`'s global python version defaults to that of your OS, denoted as “`system`” in the output of:

```
$ pyenv versions

$ pyenv global XYZ       # where, as noted above, XYZ can be any valid Python version
                         # available to your pyenv setup
```

or, if you want the system-wide Pythonv ersion to be the one to be available as the *`pyenv global`* version:

```
        $ pyenv global “$(/usr/bin/python --version | cut -d’ ‘ -f2)”
or      $ echo “$(/usr/bin/python --version | cut -d’ ‘ -f2 2>/dev/null)” \
               >| “${PYENV_ROOT}”/version
```

Caution:
Check that the latest version available on your platform (given to you by '*/usr/bin/python -version*') is also listed in the output of *pyenv install --list*. If not, update either the cloned git repo in ~/.pyenv and choose the closest prior version available:

```
$ cd $(pyenv root)
$ git pull
```

or update your installed distro package 's version.

Back to the console, check that your globally set Python version is correct:

```
$ pyenv global
```

and that it corresponds to the right default executable path, outside any activated VE:

```
$ which python
/home/$USER/$(pyenv root)/shims/python
$ eval $(which python) -version    # should yield the same version as pyenv global
```

1-3-4. Uninstalling 'pyenv'

The simplicity of Pyenv makes disabling it temporarily or uninstalling it easy.

1. To **disable** Pyenv, that is to prevent it from managing your Python VEs, simply remove
   eval "$(pyenv init -)
   from *~/.bashrc*. This will remove the $HOME/pyenv/shims directory from $PATH, and future invocations of *python* will execute your platform's Python version, as it did prior to Pyenv installation. Pyenv will still be accessible on the command line, but your Python apps won't be affected by version switching.
2. To completely **uninstall** Pyenv, go through step (1) and remove Pyenv's root directory. This will **delete all Python versions** installed under directory $(pyenv root)/versions/:

   ```
   $ rm -rf $(pyenv root)
   ```

1-3-5. Installation of 'pyenv-virtualenv'

Create the $(pyenv root)/plugins directory if it does not already exist and switch to it. Then clone the Github repo in it.

```
$ mkdir -p $(pyenv root)/plugins
$ cd $(pyenv root)/plugins
$ git clone https://github.com/pyenv/pyenv-virtualenv
$ cd pyenv-virtualenv; makepkg -sric
```

In ~/.bashrc, add *eval "$(pyenv virtualenv-init -)"* after *eval "$(pyenv init -)"*.

`pyenv virtualenv-init -` will automatically activate and deactivate VEs upon entering a directory, as long as `$PWD` contains the file `.python-version` listing the name of a valid VE as shown in the output of *`pyenv virtualenvs`*.

2. Using Python virtual environments

2-1. 'pyenv' usage

2-1-1. Selection of the virtual environment's Python version

When you try to run Python, Python must first decide which one of its versions to run. To do that it first looks for a file named `.python-version` in the current directory. If it doesn't find this file, it looks for the user-level file `~/$(pyenv root)/version`. The latter contains the global default Python version, the same that obtains with:

```
$ pyenv global          # displays the global default Python version
$ pyenv prefix          # displays the currently selected Python version
```

List all available Python versions available in the *`$(pyenv root)/versions`* directory with:

```
$ pyenv versions
```

Alternatively the directory `~/$(pyenv root)/versions` contains all Python versions installed as sub-directories.

To remove old Python versions, you can automate the removal process, with:

```
$ pyenv uninstall <version_to_remove>
```

Alternatively, simply removing the directory of the version is expedient:

```
$ rm -rf $HOME/$(pyenv root)/versions/<Python_version>
```

You can find the version's directory to be deleted by issuing:

```
$ pyenv prefix 2.6.8.
/home/$USER/$(pyenv root)/versions/2.6.8
```

If any particular version is not present or not installed, a warning message appears. You can add it with:

```
$ pyenv install -list | less     # displays all Python versions available for installation
$ pyenv install --list | sed -En 's/^\s+//; /^[3-9]\.[0-9]+\.[0-9]+$/p'
```

where the second command lists all Python 3 versions excluding versions whose major and minor include a letter e.g. alpha and beta versions and non standard build releases.

To install one or more new Python version(s), available in the previous list:

```
$ pyenv install <Python_version1> [<Python_version2>]
```

`pyenv install`
To install a new Python version (using python-build), use:

```
$ pyenv install [-f] [-kvp] <version>
$ pyenv install [-f] [-kvp] <definition_file>
$ pyenv install -l|--list
```

install options:

```
 -l/--list                   List all available versions
 -f/--force                 Install even if the version appears to be installed already
 -s/--skip-existing    Skip the installation if the version appears to be installed already
```

python-build options:

```
-k/--keep        Keep source tree in $PYENV_BUILD_ROOT after installation
(defaults to $PYENV_ROOT/sources)
-v/--verbose     Verbose mode: print compilation status to stdout
-p/--patch       Apply a patch from stdin before building
-g/--debug       Build a debug version
```

`pyenv rehash`
Installs shims for all Python binaries known to pyenv (i.e., `~/.pyenv/versions/*/bin/*`). Run this command after you install a new version of Python, or install a package that provides binaries.

```
$ pyenv rehash
```

`pyenv which`
Displays the full path to the executable that Pyenv will invoke when you run the given command.

```
$ pyenv which python3.3
/home/$USER/.pyenv/versions/3.3.3/bin/python3.3
```

`pyenv whence`
Lists all Python versions with the given command installed.

```
$ pyenv whence 2to3
2.6.8
2.7.6
3.3.3
```

2-1-2. Activation of the Python virtual environment

After installing a Python version available to Pyenv, you can create and simultaneously activate a virtual environment based on that Python version in `my_directory`, like so:

```
$ cd /path/to/my_directory
$ pyenv local <your choice of pyenv-installed Python version>
```

This will create the `/path/to/my_directory/.python-version` file containing the Python version for the virtual environment. The virtual environment is limited to the directory that contains the `.python-version` file.

You can specify one version (2.x or 3.x) or multiple versions (2.x and 3.x) as local Python at once.

2-1-3. Removing a Python virtual environment

Running `pyenv local` without a version number will just query its immediate environment and report the local configuration virtual environment version if any. In case a local virtual environment configuration exist , run:

```
$ cd /path/to/my_directory
$ pyenv local --unset
```

or

```
$ rm -f .python-version
```

to remove that local configuration.

2-2. 'pyenv-virtualenv' usage with 'pyenv'

2-2-1. Creation of Python virtual environments

To create a virtual environment for the Python version used with Pyenv, run *`pyenv virtualenv`*, specifying the Python version you want and the name of the virtual environment directory:

```
$ pyenv virtualenv 2.7.10 ve2710
```

creates a virtual environment based on Python 2.7.10 under `$(pyenv root)/versions/ve2710`. If the Python version is not explicitly specified, the virtual environment is created by default, based on the current version.

```
$ pyenv version
3.4.3 (set by /home/yyuu/.pyenv/version)
$ pyenv virtualenv venv34
```

2-2-2. Listing of Python virtual environments

To list all existing Python (and Conda) virtual environments, issue:

```
$ pyenv virtualenvs
…
```

Note that there are two entries for each virtual environment; the shorter one is just a symlink.

2-2-3. Automatic / manual virtual environments activation / deactivation

Automatic virtual environment activation is possible upon entering in a directory, if `eval "$(pyenv virtualenv-init -)"` is configured in your shell (`~/.bashrc`), and provided the `$PWD` contain a `.python-version` file that contains the name of a valid virtual environment as shown in the output of:

```
$ pyenv virtualenvs
```

`.python-version` files are used by Pyenv to denote local Python versions and can be created and deleted with the *pyenv local* command. You can also activate and deactivate a Pyenv VE manually:

```
$ pyenv activate <VE_name>
$ pyenv deactivate
```

2-2-4. Deletion of an existing virtual environments

Removing the directories in `$(pyenv root)/versions` and `$(pyenv root)/versions/{version}/envs` will delete the virtual environment. Another option is to run:

```
$ pyenv uninstall <my_virtual environment>
```

or

```
$ pyenv virtualenv-delete <my_virtual environment>
```

2-3. Working with shell-specific Python versions

To set a shell-specific Python version, just set the `PYENV_VERSION` env-var in your shell. This Python version overrides local application-specific versions as well as the global version.

```
$ pyenv shell <Python_version1> [<Python_version2>]
```

where `<Python_version>` should be either "`system`" or any "`XYZ`" string matching a Python version known to (i.e. installed in) '`pyenv`'.

To restore to the previously set shell-specific value of `PYENV_VERSION`:

```
$ pyenv shell -
```

To unset the shell-specific python version altogether, issue:

```
$ pyenv shell --unset
```

2-4. Run a Jupyter notebook in a Python virtual environment

On a platform configured with Pyenv, shims will preempt any call to `python` on the CLI by placing a new path to Python, e.g. `/home/<USER>/.pyenv/shims/python3`, before the standard Python3 entries `/bin` and `/usr/bin` in the `PATH` environment variable. Invoking the platform default Python version is always possible by specifying the full path, as in:

```
$ /usr/bin/python
```

However running a Jupyter notebook in a specific Python environment requires the configuration of a new iPython kernel, which must be later invoked from within the notebook.
The procedure to follow is simple:

```
$ cd /path/to/my_directory              # go to the virtual environment specific directory
$ pyenv local 3.7.0                     # specify the virtual environment's python version
$ python -m pip install ipykernel       # install ipykernel in that virtual environment
```

Further install any Python package needed in that virtual environment, before creating a new kernel ("`my_kernel`") based on the virtual environment Python version:

```
$ python -m pip install matplotlib pandas seaborn tensorflow==1.14.0 keras==2.2.5
$ python -m pip list                    # list local environment's packages
$ awk 'NR==2' < <(python -m pip show pandas) # list pandas' installed version
Version: 1.1.5
$ ipython kernel install --user --name <my_kernel> --display-name "Python3.7
(<my_kernel>)"
```

The new kernel is now available to iPython sessions, such as Jupyter notebooks. You can check that with:

```
$ jupyter kernelspec list
Available kernels:
Python3.7 (<my_kernel>)    /home/USER/.local/share/jupyter/kernels/<my_kernel>
python3     /home/USER/.pyenv/versions/3.6.0/share/jupyter/kernels/python3
```

Finally create a new iPython session with Jupyter from within the virtual environment specific directory:

```
$ cd /path/to/my_directory
$ jupyter notebook
```

This will cause a new window to open in your default browser. In that window you may select "`my_kernel`" under the `New notebook` field in the top-right region of the browser page.

Within a Jupyter notebook cell, check that the iPython kernel used in the Jupyter notebook points to the right Python version:

```
[1] import sys
[2] sys.executable
    '/home/USER/.pyenv/versions/3.7.0/bin/python'
```

References

- pyenv: https://github.com/pyenv/pyenv
- pyenv's complete cmds' reference: https://github.com/pyenv/pyenv/blob/master/COMMANDS.md
- pyenv-virtualenvwrapper: https://github.com/pyenv/pyenv-virtualenvwrapper
- virtualenvwrapper: https://virtualenvwrapper.readthedocs.io/en/latest/
- https://opencafe.readthedocs.io/en/latest/getting_started/pyenv/

Appendix A: Work with '`virtualenvwrapper`'

Prior to creating a Python virtual environment, pick which version of Python you need for that environment:

```
$ pyenv versions
```

To create a new isolated virtual environment based on your systems global default Python version, you may simply start a new project. The minimal requirement is that the project's name be specified – e.g. '`pytf`' below:

```
$ mkproject pytf
```

Given *`PROJECT_HOME`* as defined in *`~/.bashrc`*, a new empty project directory will be automatically created at *`$PROJECT_HOME/pytf`*, and a new virtual environment will be created at *`$WORKON_HOME/pytf`* based on the Python version's global default for your system. You are also automatically transferred to the new project's directory.

The complete syntax for the above cmd is:

```
$ mkproject [-f | --force] [-t template] [-p <python_executable>] \
        [+ other virtualenv options] pytf
```

wher

`-f, --force` Create the virtualenv even if the project directory already exists

`-t <template>` Multiple templates can be selected, applied following cli order

`-p <python_executable>` as defined by existing Python version installations, e.g.

`$(pyenv root)/versions/2.7.16/bin/python2`

The Python executable must be provided with an absolute path or as above.

Instead you can also use, with an already ***created*** project, *`pytf`*:

```
$ mkvirtualenv [-a existing_project_dir_path] [-i package_to_install] \
        [-r pip_packages_list_file] [-p <python_executable>] \
        [+ other virtualenv options] /path/to/project/pytf/directory
```

where you may specify:

`-a <project_directory_absolute_path>`, when the project directory already exists

`-i <package_to_install>,` to specify one or several packages to be install after virtualenv creation, repeating '`-i ...`' as many time as needed,

`-r <pip_packages_list_file>`, a file containing a list of packages to be processed as arguments of *`` `pip -r` ``* for installation in the new environment,

-{virtualenv options}, e.g.: ***`-p <python_executable>`*** as defined by existing Python version installations, *`$(pyenv root)/versions/2.7.16/bin/python`*

The Python executable must be provided with an absolute path, or following query of available Python flavors and installation in the available versions directory:

```
$ pyenv install -l
[... list of available python versions]
$ pyenv install XYZ     # where XYZ may be “3.6.0” or “stackless-2.7.5” or ...
$ pyenv versions
```

This would create a new Python's installation in the *~/.virtualenvs/pytf/* directory based on Python version 2.7.16. Upon its creation '`pyenv`' switches directly to that virtual environment. In other circumstances, manual switching to any already created virtual environment is done by calling:

```
$ workon pytf           # switch to project directory and activate virtualenv
(pytf) $ _
```

Now the prompt should reflect the fact that you are in the project's directory, in an activated virtual environment.

In principle the *workon* cmd should switch your *PWD* to the project's directory automatically, unless the inline option ***-n*** or ***--no-cd*** is present.

Package installation may proceed with *pip*:

```
(pytf) $ pip install tensorflow
```

This package will be installed only in the `pytf` environment. You can list all installed packages by calling:

```
(pytf) $ lssitepackages
```

A list of built virtual environments can be obtained, and any deactivated virtual environment can be removed quite simply, by calling:

```
(pytf) $ deactivate

$ lsvirtualenv  [-b | -l] or [-h]  # to list virtual environments or to get help

$ rmvirtualenv pytf    # must be deactivated first
```

For more details on cmd's syntax, consult the '`virtualenvwrapper`' documentation.