Table of Contents

Overview

- Why This Change?
- Nomenclature

Basic Venv Usage

- Create the venv
- Activate the venv
- Use the venv
- Deactivate the venv

More Venv Details

- What did the activate script do?
- Why was activate "sourced"?
- Use without activating
- Virtual Environment Configuration

Usage With sudo

- NeoPixel Example
- Option 1 - Invoke with sudo passing environment
- Option 2 - Use absolute paths

Automatically Running at Boot

Other Ideas

- Automatically enabling venv at login
- Disabling Prompt Change
- Hide the venv folder
- That --break-system-packages option
- Return to the "good 'ole days"
- Multiple venvs

TLDR
Overview

Starting with the October 10, 2023 Bookworm release () of the Raspberry Pi OS (), the use of Python Virtual Environments (venv) when pip installing packages is required. No more sudo pip. This will break things and require learning new things. Yeah.

This guide tries to help with this transitional pain by covering basic venv usage on Raspberry Pi's.

Why This Change?

This change is being done to protect the system level Python installation. This is not a new issue and using venv has long been the recommended solution. We've just been slow to adopt this on Raspberry Pi boards.

The main issue is this - when Python modules are installed using `pip`, they get installed into the system level Python installation. This can potentially break the system level installation.

Read the PEP for more details:

PEP 668

Using a virtual environment, the pip installed packages get placed there instead of at the system level.
Nomenclature

In this guide, venv is used as short hand for Python Virtual Environment in general. It's also the name of the Python module used to create new Virtual Environments.

Python "modules" may also be referred to as libraries or packages. They're the thing that gets install via pip.

Also, "Python" here always means Python 3. Code examples may show either `python` or `python3`. The commands should be equivalent.

Basic Venv Usage

This part of the guide is generic and covers basic virtual environment usage. However, it is Linux specific since we are focusing on Raspberry Pi usage.

The basic steps are:

- Create the venv - this is done once (per venv)
- Activate the venv - this is done every time a venv is to be used
- Use the venv - run your Python code here
- Deactivate the venv - optional

Create the venv

To create a new virtual environment, use the `venv ()` module and give it the name for the virtual environment.

```
python3 -m venv foobar
```

The virtual environment name can be any valid name. A lot of tutorials show "venv" or ".venv" for the name. That's fine but tends to imply that name is important. It's not. So here we are intentionally picking something silly, `foobar`, to be clear that the virtual environment name is just a name.

So what did that command do? It created a new folder with the virtual environment name and setup a folder structure that mimics the layout the Python interpreter expects.

```
pi@raspberrypi:~ $ python3 -m venv foobar
pi@raspberrypi:~ $ ls
```
Activate the venv

The main way to use the virtual environment is by "activating" it. This is done by "sourcing" the `activate` script found in the virtual environments bin folder.

```
source foobar/bin/activate
```

The prompt should change to include the virtual environments name. This both helps indicate a venv is in use (active) and which one.

```
pi@raspberrypi:~ $ source foobar/bin/activate
(foobar) pi@raspberrypi:~ $
```

Use the venv

Once the virtual environment has been activated, Python usage proceeds in the normal fashion. Running `python` or `pip` will be done in the context of the virtual environment.

Modules installed with pip will be placed in the local venv folders - sudo should not be used.

```
(foobar) pi@raspberrypi:~ $ pip list
Package Version
---------- -------
pip 23.0.1
setuptools 66.1.1
(foobar) pi@raspberrypi:~ $ pip install click
Collecting click
  Downloading https://www.piwheels.org/simple/click/click-8.1.7-py3-none-any.whl (97 kB)

97.9/97.9 kB 47.2 kB/s eta 0:00:00
Installing collected packages: click
Successfully installed click-8.1.7
(foobar) pi@raspberrypi:~ $ pip list
Package Version
---------- -------
click 8.1.7
pip 23.0.1
setuptools 66.1.1
(foobar) pi@raspberrypi:~ $
```

Running Python will have access to the pip installed modules of the venv.
Deactivate the venv

If you ever want to "turn off" the virtual environment and return to the regular state, the deactivate command can be used.

```
$ deactivate
```

This is not a Linux command. This "command" is a shell function that was defined in the activate script when it was originally sourced. It simply undoes what the activate script did.

More Venv Details

Let's start looking a little more under the hood at how Python venv actually works. This will help to better understand the various options for using venv on Raspberry Pi.

What did the activate script do?

Python venv are "activated" by running the aptly named activate script found in the venv bin folder. It's just a shell script, so it is a plain text file you can open in any text editor. It's not crazy long or complex, only about 70 lines of shell code - half of which define the deactivate function.

This script does some cute things like change the prompt to include the venv name. However, the key thing done is to alter the PATH variable such that the venv Python path shows up before the system path.

Observe the behavior before activating the venv:

```
$ which python
/usr/bin/python
$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/local/games:/usr/games
$
There is no mention of the venv in `PATH` and as a result the python command is finding the system level installation in `/usr/bin`. (pip would have similar behavior)

Now activate the venv:

```
pi@raspberrypi:~ $ source foobar/bin/activate
(foobar) pi@raspberrypi:~ $
```

And note the change in behavior:

```
(foobar) pi@raspberrypi:~ $ which python
/home/pi/foobar/bin/python
(foobar) pi@raspberrypi:~ $ echo $PATH
/home/pi/foobar/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/
usr/local/games:/usr/games
(foobar) pi@raspberrypi:~ $
```

The python command is now pointing to the venv installation. This is a result of `/home/pi/foobar/bin` being added to `PATH`. The paths in `PATH` are search in order, left to right. So by having the venv path before the system path, the venv is found first.

**Why was activate "sourced"?**

Shell scripts are usually "run". But above we instead "sourced" the `activate` script. Why was that done? It has to do with context. When a script is "run", it gets launched in a new shell context, and when the script is done, that context goes away. However, when a script is "sourced", it is run in the current context.

This is important for things like environment variables, which can be set and accessed very simply in bash:

```
pi@raspberrypi:~ $ FOO=23
pi@raspberrypi:~ $ echo $FOO
23
pi@raspberrypi:~ $
```

Now consider this simple script, called `test.sh`, which similarly sets a variable:

```
pi@raspberrypi:~ $ ls -l test.sh
-rwxr-xr-x 1 pi pi 7 Oct 23 11:50 test.sh
pi@raspberrypi:~ $ cat test.sh
BAR=42
pi@raspberrypi:~ $
```

If this script is "run", the `BAR` variable does not persist:
However, if the script is "sourced", the BAR variable does persist:

```
pi@raspberrypi:~ $ source test.sh
pi@raspberrypi:~ $ echo $BAR
42
pi@raspberrypi:~ $
```

Since "activating" a Python virtual environment needs to alter the current context, the activate script should be sourced, not run.

```
In bash, . is a shortcut for source.
```

Use without activating

Activating a Python venv is really just a convenience. It's possible to use a venv without activating it by explicitly invoking the Python interpreter found in the venv. This is done by using absolute paths.

To demonstrate, let's use this simple Python script, called test.py, that tries to import the click module:

```
import click
print("Done.")
```

The click module is pip installed into the foobar venv:

```
(foobar) pi@raspberrypi:~ $ pip install click
Collecting click
  Downloading https://www.piwheels.org/simple/click/click-8.1.7-py3-none-any.whl (97 kB)
Installing collected packages: click
Successfully installed click-8.1.7
(foobar) pi@raspberrypi:~ $ pip list
Package  Version
---------- -------
click     8.1.7
pip       23.0.1
setuptools 66.1.1
(foobar) pi@raspberrypi:~ $
```

So running with the venv activated has the expected output:
However, if the venv is deactivated, the script fails:

```
(foobar) pi@raspberrypi:~ $ deactivate
pi@raspberrypi:~ $ python test.py
Traceback (most recent call last):
  File "/home/pi/test.py", line 1, in &lt;module&gt;
    import click
ModuleNotFoundError: No module named 'click'
pi@raspberrypi:~ $
```

This is because click was only installed in the foobar venv. Once deactivated, we are back to using the system level install, which does not have click installed.

However, by using an absolute path to the venv, it works again:

```
pi@raspberrypi:~ $ /home/pi/foobar/bin/python test.py
Done.
pi@raspberrypi:~ $
```

Virtual Environment Configuration

A file named `pyvenv.cfg` that lives in the venv folder controls some basic virtual environment configuration.

```
pi@raspberrypi:~ $ ls foobar/
bin/  include/  lib/  pyvenv.cfg
```

This file contains a simple list of key = value pairs.

```
home = /usr/bin
include-system-site-packages = false
version = 3.11.2
executable = /usr/bin/python3.11
command = /usr/bin/python3 -m venv /home/pi/foobar
```

Probably the most important one is the `include-system-site-packages` option. This does pretty much what it says. If set to `true`, then the venv has access to modules (aka libraries, aka packages) installed at the system level. The default is `false`.

For example, `gpiozero` is usually pre-installed at the system level on most Raspberry Pi's:
But with the default `false` setting of `include-system-site-packages`, the fooar venv does not have access to it:

```bash
pi@raspberrypi:~ $ source foobar/bin/activate
(foobar) pi@raspberrypi:~ $ python
File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'gpiozero'
```

If the pyvenv.cfg in the fooar venv folder is changed to (only change is false to true):

```python
home = /usr/bin
include-system-site-packages = true
version = 3.11.2
executable = /usr/bin/python3.11
command = /usr/bin/python3 -m venv /home/pi/foobar
```

Then the venv does have access:

```bash
pi@raspberrypi:~ $ source foobar/bin/activate
(foobar) pi@raspberrypi:~ $ python
```

Enabling access to system site packages can also be done when initially creating the venv by using the `--system-site-packages` command line argument:

```bash
python3 -m venv --system-site-packages fooar
```

Most setups will likely want to enable the `--system-site-packages` feature.

---

**Usage With sudo**

Now things are getting more Raspberry Pi specific. Sometimes elevated privileges are needed to access certain hardware on the Raspberry Pi (ex: /dev/mem). The typical way to do this is by invoking the Python script using sudo. However, sudo usage with virtual environments requires some special attention.
It’s not as simple as `sudo make me a sandwich`().

**NeoPixel Example**

Let’s use a simple NeoPixel script as an example, since NeoPixels generally require running with sudo. Here’s the simple NeoPixel code we want to run, which we will call `neo_test.py`:

```python
import board
import neopixel

pixels = neopixel.NeoPixel(board.D18, 10)
pixels.fill(0xADAF00)
```

We start with a Raspberry Pi setup with Blinka:

**Installing Blinka on Raspberry Pi**

For this example, Blinka is installed into a venv called blinka (remember venv names are just names) using the [manual process](https://adafruit.github.io/circuitpython manuals/installation#the-manual-process)...

```
pi@raspberrypi:~ $ python3 -m venv blinka
(blinka) pi@raspberrypi:~ $ source blinka/bin/activate
(blinka) pi@raspberrypi:~ $ pip3 install --upgrade adafruit-blinka
```

And then the NeoPixel library is also installed:

```
(blinka) pi@raspberrypi:~ $ pip3 install adafruit-circuitpython-neopixel
```

If we try running `neo_test.py` without sudo, we get a permission error:

```
(blinka) pi@raspberrypi:~ $ python3 neo_test.py
Can’t open /dev/mem: Permission denied
```

The full traceback is shown in the code snippet below.

---

© Adafruit Industries
raise RuntimeError(
RuntimeError: NeoPixel support requires running with sudo, please try again!
swig/python detected a memory leak of type 'ws2811_t *', no destructor found.
(blinka) pi@raspberrypi:~ $

The error message says to run with sudo, so let's try that:

(blinka) pi@raspberrypi:~ $ sudo python3 neo_test.py
Traceback (most recent call last):
  File "/home/pi/neo_test.py", line 1, in &lt;module&gt;
    import board
ModuleNotFoundError: No module named 'board'
(blinka) pi@raspberrypi:~ $

That just throws a different error. And it is confusing. Now it can't find board? What's going on?

The general issue is that sudo launches a new environment when invoked. So the script is running in that new environment, which generally does not known about the virtual environment. That's why the board module (which is part of Blinka installed in the venv) can not be found.

So how do we get around this?

Option 1 - Invoke with sudo passing environment

The `sudo` command has the appealing `-E` option which allows users to "preserve the existing environment variables". Since most of the magic with a venv comes from its altering of the `PATH` environment variable, it seems like this should work. But it doesn't:

(blinka) pi@raspberrypi:~ $ sudo -E python3 neo_test.py
Traceback (most recent call last):
  File "/home/pi/neo_test.py", line 1, in &lt;module&gt;
    import board
ModuleNotFoundError: No module named 'board'
(blinka) pi@raspberrypi:~ $

The `-E` did not preserve everything. As a result, the system level Python install ends up being used:

(blinka) pi@raspberrypi:~ $ sudo which python3
/usr/bin/python3
(blinka) pi@raspberrypi:~ $ sudo -E which python3
/usr/bin/python3

To get around this, the `env` command can be use inline to allow specifying the `PATH` variable be explicitly copied into the invoking environment:
Now it's finding the Python in the venv path. Using this to run the script works:

```
(blinka) pi@raspberrypi:~ $ sudo -E env PATH=$PATH python3 /home/pi/blinka/bin/python3
(blinka) pi@raspberrypi:~ $ sudo -E env PATH=$PATH python3 neo_test.py
(blinka) pi@raspberrypi:~ $
```

Make an Alias

The command `sudo -E env PATH=$PATH python3` is a bit klunky. To make invoking this easier, an alias can be used.

```
alias supy='sudo -E env PATH=$PATH python3'
```

The name `supy` can be changed to whatever you want. Adding it to `.bashrc` or `.bashrc_aliases` will make the alias available with every login. Once the alias is set, can then use it:

```
(blinka) pi@raspberrypi:~ $ supy neo_test.py
```

Option 2 - Use absolute paths

As mentioned previously, a venv can be used without activating by using absolute paths to point to the venv's Python install. This also works with `sudo`.

An easy way to get the absolute path is to activate the venv and check that way:

```
pi@raspberrypi:~ $ source blinka/bin/activate
(pi) pi@raspberrypi:~ $ which python3 /home/pi/blinka/bin/python3
```

And then all that is needed is to invoke with `sudo` using that absolute path. To demonstrate, first deactivate the venv:

```
(blinka) pi@raspberrypi:~ $ deactivate
pi@raspberrypi:~ $
```

Now use `sudo` and the absolute path, no extra parameters are needed:

```
pi@raspberrypi:~ $ sudo /home/pi/blinka/bin/python3 neo_test.py
pi@raspberrypi:~ $
```
Make an Alias

An alias can be made for this as well:

```bash
alias supy='sudo /home/pi/blinka/bin/python3'
```

Keep in mind the virtual environment is baked into the absolute path. So this command invokes with that specific virtual environment.

But once set, the alias can be used:

```bash
pi@raspberrypi:~ $ supy neo_test.py
pi@raspberrypi:~ $
```

Note that the virtual environment is not active in the above example.

Automatically Running at Boot

Now to see how a Python script can be run in a virtual environment at boot time. This can be done by creating a normal systemd unit file and being careful to use absolute paths.

See this guide for more details about writing and enabling a systemd service:

[systemd: Writing and Enabling a Service](#)

This example assumes Blinka has been installed on the Pi in a venv named `blinka` and that the [NeoPixel library](#) has been pip installed into that venv.

[Installing Blinka on Raspberry Pi](#)

Here’s an example NeoPixel script that will run a little spin animation on 12 RGBW NeoPixels attached to pin 18 on the Pi’s GPIO header.

Save this as `neopix_spinner.py` in the `/home/pi` directory.

```python
import time
import board
import neopixel

NEO_PIN = board.D18
NUM_PIXELS = 12
COLOR = 0xADAF00
```
DELAY = 0.1

pixels = neopixel.NeoPixel(NEO_PIN, NUM_PIXELS, pixel_order="GRBW")

while True:
    for p in range(NUM_PIXELS):
        pixels.fill(0)
        pixels[p] = COLOR
        time.sleep(DELAY)

To access Python installed in the blinka venv, the absolute path is 
/home/pi/blinka/bin/python3. The script is saved to /home/pi/
nepix_spinner.py.

Do a test run from the command line to make sure it works:

pi@raspberrypi:~ $ sudo /home/pi/blinka/bin/python3 /home/pi/neopix_spinner.py

This command should not return since the program uses an infinite loop to spin
forever. However, if you actually have NeoPixel hardware connected, it should be
lighting up. Pressing <CTRL><C> will break out of the code and return to the prompt.

pi@raspberrypi:~ $ sudo /home/pi/blinka/bin/python3 /home/pi/neopix_spinner.py
^CTraceback (most recent call last):
  File "/home/pi/neopix_spinner.py", line 16, in &lt;module&gt;
    time.sleep(DELAY)
KeyboardInterrupt
pi@raspberrypi:~ $

Now to create and enable a systemd unit file to run this. Use sudo with a text editor to
create a file named /lib/systemd/system/neopixel.service with the following
contents:

[Unit]
Description=NeoPixel Spinner

[Service]
ExecStart=/home/pi/blinka/bin/python3 /home/pi/neopix_spinner.py

[Install]
WantedBy=multi-user.target

Note that sudo is *not* used inside the systemd unit file.

With this approach, the venv does not need to be "activated".

Now run the following commands to enable and start the neopixel service:
sudo systemctl enable neopixel
sudo systemctl start neopixel

If these commands run as expected, they don't generate much output:

pi@raspberrypi:~ $ sudo systemctl enable neopixel
Created symlink /etc/systemd/system/multi-user.target.wants/neopixel.service → /lib/systemd/system/neopixel.service.
pi@raspberrypi:~ $ sudo systemctl start neopixel
pi@raspberrypi:~ $ 

But the NeoPixels should be lighting up now. Further, this process should run everytime the Pi is booted.

If you want to stop the script from running at boot, then stop (stops current execution) and disable (removes sym link used to boot at start) the service:

sudo systemctl stop neopixel
sudo systemctl disable neopixel

The status can be checked using status.

pi@raspberrypi:~ $ sudo systemctl status neopixel
● neopixel.service - NeoPixel Spinner
   Loaded: loaded (/lib/systemd/system/neopixel.service; enabled; preset: enabled)
   Active: active (running) since Mon 2023-10-23 16:58:53 PDT; 1s ago
   Main PID: 900 (python3)
   Tasks: 1 (limit: 1586)
   CPU: 124ms
   CGroup: /system.slice/neopixel.service
   └─900 /home/pi/blinka/bin/python3 /home/pi/neopix_spinner.py


pi@raspberrypi:~ $ 

Other Ideas

Automatically enabling venv at login

If your main use for a Raspberry Pi is running Python scripts, then needing to activate a virtual environment every time can get tedious. By adding the venv activation to your .bashrc file, it will be activated everytime you log in.

For example, if a venv named foobar has been previously created, add this line to the end of your .bashrc file:

source ~/foobar/bin/activate
The venv will activate along with the changed prompt with each login.

**Disabling Prompt Change**

If you don't want the venv name to show up in the prompt, it can be disabled by setting the environment variable `VIRTUAL_ENV_DISABLE_PROMPT` before activating the venv.

```
pi@raspberrypi:~ $ VIRTUAL_ENV_DISABLE_PROMPT=1
pi@raspberrypi:~ $ source foobar/bin/activate
pi@raspberrypi:~ $ which python
/home/pi/foobar/bin/python
pi@raspberrypi:~ $
```

This can be done interactively as shown above or placed in `.bashrc` so it's active with every login.

```
This is potentially confusing since it makes it less obvious which "Python" is actually running.
```

**Hide the venv folder**

This is simple - just add a `.` to the start of the venv name. This takes advantage of the Linux behavior treating dot files as hidden and not showing them by default with `ls`. In a lot of guides, the name `.venv` is used, but it's up to you what to call it. It's just the name you'll use when activated or referring to the venv.

This is entirely cosmetic.

**That --break-system-packages option**

This option can be used with `sudo pip` to force the installation despite the warning. However, this is not recommended. It should be possible to do everything using venv.

**Return to the "good 'ole days"**

If you're totally against venv's and don't care about possibly breaking your system's Python setup and just want to continuing doing things like you used to, then system wide pip installs can be re-enabled (undisabled?) by deleting a file named `EXTERNALLY-MANAGED` found in the system's Python setup. This is really just the same as using the --break-system-packages option mentioned above.
For example, on a freshly booted Pi running Bookworm, the file can be found at:

```
/usr/lib/python3.11/EXTERNALLY-MANAGED
```

So just `sudo rm` it:

```
sudo rm /usr/lib/python3.11/EXTERNALLY-MANAGED
```

Now you can `sudo pip install` all you want. At least until it finally corrupts the Python setup.

**Multiple venvs**

Keep in mind that more than one venv can be created. Each just ends up being its own folder with all the venv contents inside. This can come in handy when dealing with module conflicts or specific requirements. Like FooEditor requires PyQt5 5.9.2 *only* while BarEditor requires PyQt 5.12.1 *only*. A venv can be setup for each to keep the PyQt module installs separate.

---

**TLDR**

- **NO** = `sudo pip`
  - NEVER RUN PIP WITH SUDO
- **YES** = `sudo apt` or `sudo apt-get`
  - installing Debian packages is OK
  - many Python modules are available as Debian packages
- **YES** = `sudo /absolute/path/to/venv/bin/python foo.py`
  - invoking Python with sudo is OK
  - this is required in some situations to access underlying hardware (ex: neopixel)

- **PEP 668 ()**
- **xkcd 1987 ()**