Relieving your Python packaging pain

60% of the time, it works every time

MAR 27, 2023

Summary

You can protect yourself against a lot of packaging pain by sticking to one recipe for installing and running Python:

1. *Don't install the latest major version of Python*
2. *Use only the python.org installer on Windows and Mac, or official repositories on Linux.*
3. *Never install or run anything outside of a virtual environment*
4. *Limit yourself to the basics: "pip" and "venv"*
5. *If you run a command, use "-m"*
6. *When creating a virtual environment, be explicit about which Python you use*

The dirty little secret of Python packaging

Now that the Python 3 transition is over and Guido is working on making the language faster, the new topic of complaint for the community is packaging.

What you have tried so far hasn't worked. In fact, most advice on the topic will not help you.

That's because the dirty secret of Python packaging is that most of the problems... don't come from packaging.

They come from the deeper problem of bootstrapping Python, meaning finding, installing, configuring and running it.

I've tried everything under the sun for 15 years, with experts and beginners, professionals and amateurs, and there is no "one true way".

But there is one way that will fail a lot less often, for most people.

In this article, I will give you the steps you need to follow.

In another new article, I explain why I recommend following these steps.

1. Don't install the latest major version of Python

Yes, the latest version is shiny. It's faster, has cool features, and smells good. I'm not saying you should not give it a try, and play with it.

However, don't use it in any project.

So if Python 3.11 is the latest major version, at the maximum, use 3.10. You can use an older one, of course. If it's possible (I understand people don't want to update Python every year), target down to 4 versions as your minimum. In our example, down to 3.7.

In doubt, check the Python status page.

In this page, "feature" means the one being currently in development, "bugfix" are the versions that will receive fixes for bugs, "security" the ones that will receive fixes for security issues, and "end-of-line" the ones that will not be updated any more.

2. A. On Windows and Mac, use the official installers

There are a lot of ways to install Python, and how you choose to install it matters a lot.

Should you use Homebrew, the Windows store, or use Anaconda?

No.

Go to python.org, and use their Python installer for Windows or their installer for Mac.

If you want to know why, that's the role of the other article.

If you want to know how to do it, I made an article dedicated to the full procedure of installing Python.

Also, because I know some users are forced to use Anaconda, you will find a dedicated section about it at the end of this article.

2. B. On Linux, use the official repositories

Use whatever official tool comes with your distribution to install Python, be it "apt", "yum", "dnf", etc.

You will be limited to the versions of Python in the repositories, but resist the temptation to use pyenv to get around that.

If by any chance you happen to use Ubuntu, you are in luck: you can use the deadsnake PPA to extend the number of versions at your disposal.

Ditto if you are on Red Hat, with the EPEL.

You may develop on Windows/Mac and deploy on Linux. In that case, check what is available on the Linux machine, and install the same version of Windows, not the other way around.

And if none of that made sense, get the full tutorial on the topic.

3. To install things, use pip, and only pip

Don't use conda.

Don't use poetry, pipenv, pdm, easy_install

Don't use pipx. pipx is not pip. At all.

Don't use apt, yum, etc.

If you don't know what pip is or how to use it, we cover it in another article.

4. Always use pip in a virtual environment

Since you are reading this article, it likely means you don't have the necessary knowledge to make exceptions to that rule.

So don’t.

This rule is the most important one.

If you need to install something, anything, you should do so in a virtual environment.

Even to install black, jupyter, mypy or whatever you are thinking right now, do it in a virtual environment.

The bottom line is, **if you are not 100% sure you are using a virtual environment when you are installing something, the first thing you should do is making sure you are.**

If you are typing "--user", you are NOT installing in a virtual environment.

If you are typing "sudo", you are NOT installing in a virtual environment.

Once things are installed in a virtual environment, to use them, you need to be in the same environment. So also run all other commands in a virtual environment. Not just pip.

If you don't know what is a virtual environment or how to use one, you guessed it, we cover it in another article.

Yes, it's annoying. You just want to code, not to deal with this pesky virtual environment. Why can't it work like cargo or npm and do the easy thing?

The answer to that is pretty long.

But the situation is such that there is currently no way for you to install anything reliably without a virtual environment.

So you have two choices. Don't use a virtual environment, and suffer a lot. Use one, and suffer a little.

5. To create a virtual environment, use “venv” and only “venv”

“venv” is a command that comes with most Python installations, and it’s the command you should use to create a new virtual environment.

There are other commands such as “virtualenv” and “virtualenvwrapper”. [1]

Don’t use them.

There are other tools such as pipx, pdm, poetry and pipenv.

Don’t use them.

And of course, anaconda comes with their own “env” sub command.

You get it: don’t use it.

At this point, half of the readers who are using those tools should be boiling inside. Once again, remember there is another article about this.

Importantly, “venv” comes bundled with the Python installer from python.org, but Linux, you will have to install a special package, unfortunately, and it’s different for each distro.

6. When running a Python command use "-m"

“-m” is a flag on the “python” command that few users seem to know about. It lets you run any importable Python module, no matter where you are. Because most commands are Python modules, we can use this to say, “run the module X of this particular python”.

E.G:

Don't do :

```
pip install
```

Do:

```
python -m pip install
```

Don't do :

```
black
```

Do:

```
python -m black
```

Don't do :

```
jupyter notebook
```

Do:

```
python -m jupyter notebook
```

You should do this, even when you are in a virtual environment, despite everything people will tell you.

Yes, it's annoying. You just want to code, not to deal with this pesky "-m". Why can't it work out of the box?

The answer to that is pretty long.

But the situation is such that there is currently no way for you to run any python command reliably without "-m".

So you have two choices. Don't use "-m", and suffer a lot. Use "-m", and suffer a little.

6. When creating a virtual environment, be explicit about the Python you use

It’s common to have several Python installed on your computer. Sometimes without knowing it.

When you create a virtual environment, you should always specify which Python you want to use for the task, because that environment will be forever using this particular Python.

On Windows, this means using the “py” command. This command comes with the python.org installer, and can list all Python currently installed on your machine by doing:

```
py --list-paths
```

Which may show you something like this:

```
 -3.9        C:\Python39\python.exe
 -3.8        C:\Python38\python.exe *
 -3.7        C:\Python37\python.exe
```

You can then choose what Python to run doing:

```
py -X.Y
```

E.G, to run Python 3.7:

```
py -3.7
>>> print('Hello')
Hello
```

On Linux and Mac, you will have to use a version suffix, usually “pythonX.Y”.

E.G, to run Python 3.7:

```
python3.7
>>> print('Hello')
Hello
```

In both cases, those are instructions to select an existing Python and run it. It assumes the Python is installed on the machine. It cannot run a Python that you haven’t installed before.

Most importantly:

If you follow all the steps so far, this means you will use “venv” and “-m” with this.

And so to create a virtual environment you will end up typing some monstrous command.

E.G, on Windows:

```
"py -3.8 -m venv .venv"
```

E.G, on Linux and Mac:

```
"python3.8 -m venv .venv"
```

(If this means nothing to you, again, we will cover that in another article)

It looks hideous. But this is the way.

Yes, it's annoying. You just want to code, not to deal with Mandolarian Sigils. Why do we have to write something that Chat GPT doesn't even recommend?

The answer to that is pretty long.

But the situation is such that there is currently no way for you to create reliably a virtual environment without this.

So you have two choices. Keep doing it the way you did it before and suffer a lot. Or put this magic string in a file to copy / paste it every time you need it, and suffer a little.

Let's review all this

1. Don't install the latest major version of Python
2. Use only the python.org installer on Windows and Mac, or official repositories on Linux.
3. Never install or run anything outside of a virtual environment
4. Limit yourself to the basics: "pip" and "venv"
5. If you run a command, use "-m"
6. When creating a virtual environment, be explicit about which Python you use

And if all of that feels bonkers, read the other articles that explain how to do it.

You may have noted we almost never talked about anything related to packaging. Yet doing this will reduce significantly all problems that most users identify as packaging problems.

At this point, many of you may also have objections to those recommendations.

So it's the perfect time for the explanation of how and why they came to be.

What if you are forced to use Anaconda?

It's important to be 100% sure you have to use Anaconda. I've met many people who thought they did, but after following the above procedures, realized they actually didn't. Since the introduction of wheel files in pypi, packages that used to be a nightmare to install, such as GUI tool-kits or the scientific stack, are now a breeze.

Still, I know some people have to.

You may be stuck with some machine learning Frankenstein project, or the policy of your company is to use Anaconda and nothing else.

In that case, you can still survive by making sure to stay in the Anaconda bubble.

Use "conda", and only "conda", for everything.

Do not use "pip" or "venv". Or any other tool that is not from Anaconda. Certainly, never mix "conda" with them.

This will limit your horizon to what's available in your Anconda channels, but that's the only way to stay sane.

The advice that you should always create a virtual environment and do everything in it still holds, but create it using "conda".

1 The naming around virtual environment is messed up. First, virtual environments are just glorified directories. Do not think they are something special like a virtual machine. Secondly, we tend to abbreviate the concept of "virtual environment" as "venv" or "virtualenv", which is unfortunate, since those are also the name of some tools to create virtual environments. This makes the whole situation confusing.

19 Likes · 5 Restacks

← Previous Next →

Discussion about this post

Comments Restacks

Write a comment...

Julian Aug 9 · *edited Aug 9*

How about using Make to make this all easier? Couldn't it just be the relief you need?

LIKE REPLY SHARE

3 replies by Bite Code! and others

Peter Aug 1 · *edited Aug 1*

I'd love to follow this advice, but different projects in my company use different tools. I work on projects that use venv, poetry, pipenv and pyenv. It's gotten so confusing that I wrote a shell script to deactive all the different environments I might be in, then just activate the one for the project I want to work on.

Any advice on working with several different Python package managers? When I accidentally use one on the wrong project things get very weird.

LIKE REPLY SHARE

1 reply by Bite Code!

23 more comments...