Why not tell people to "simply" use pyenv, poetry, pipx or anaconda

You keep using that word. I don't think it means what you think it means.

MAR 30, 2023

41 28 Share

Summary

In "Relieving your Python packaging pain", I shared what I've witnessed to be the most reliable way to avoid many python packaging problems, for a large number of users. This article listed the steps to take without justifying them, delegating this responsability to today's post.

Among other things, it advised not to use homebrew, pyenv, anaconda, poetry, pipx, and other tools. The reason is mostly because they have too many modes of failure, while solving problems that people won't have as long as they are battling more primitive issues.

Instead, it focused on providing a basic procedure that will let most people do their job and have the highest possible rate of success.

If you disagree, it's likely that you are too good at handling complexity to realize how many problems you solve every day.

Type your email... Subscribe

Targeting the pain

The Python community is extremely diverse. Some users are devs, some are sys admins, some are teachers, some quants, some 3D artists, some data analysts. Some are on Windows, some are on Mac, BSD or Linux. Some are little girls, some are old bearded guys. Some are in Africa, coding between power cuts and internet shutdowns, some are in the Silicon Valley with wifi in their bus to the Googleplex, and some pip install behind the Great Firewall of China.

A geographer working with QGIS is not a Java dev getting into Python. A Django senior is not Blender plugin author.

But the packaging pain is there.

So how do we help as many people as we can?

After many, many years of helping people in Python, I noticed some pattern.

One big problem was the paradox of choice: too many ways to solve your packaging problems, too many docs or tutorials, with various levels of quality or relevance.

Another was that some errors were recurring over and over:

- The users install a command, and when they run it, they get a “command not found”.
- They install a package, and when they import it, they get an “ImportError”.
- They run Python, but the wrong version starts.
- They do one of the above, and it crashes.

Whether or not those are packaging problems is irrelevant, this is what people perceived as such.

There is no possible exhaustive list of all the causes of those issues, but fortunately, we don’t need one to do a lot of good. What we need is a Pareto list:

- The user PATH is wrong.
- The user doesn’t know what Python they are using.
- Python is broken.
- They don’t have the permission to do something.
- They installed incompatible things together.

- They are missing a dependency.

And so the recipe was born, refined as the years went by, from people to failure, from failure to pain. The quest to provide the maximum benefits to the largest sample of users.

Pieces of it went into tweets, forums, hacker news, and one day, I decided it was time to stop repeating it and just write all it down.

But I could not tag this as being “for beginners”, because it is not that simple. I know veterans that still struggle with this stuff, since every time it comes up, they have a deadline, and rush to the fastest way out, not the long-term solution.

Besides it was not a solution, rather, setuping people for less disappointment.

So I called it “Relieving pain”, it seemed the most honest.

You may have noticed what is **not** in that list:

- A dependency fail to install without crashing.
- Python fails to install without crashing.
- The user fails to create a virtual environment.
- The environment cannot be reproduced on another machine.
- The user can’t install exactly the Python version they want.

Those problems do happen. In fact, this is exactly what poetry/anaconda/homebrew/pyenv have been designed for.

However:

- Those problems happen less often than the ones from the other list.
- They don’t have a generic solution you can easily explain in a tutorial.
- Each tool solving those have subtle trade-offs, and add more modes of failures.
- The solution for each is not going to complement the solution for another. Sometimes applying them together creates new problems.

- The recipe from the previous article addresses some of those problems directly or indirectly.

You see, there is no such thing as “removing all packaging problems” in Python.

“relieving pain” is not a cure, because a cure at the user level doesn’t exist.

On top of that, as you will see in another section, the tooling used to solve the second list problems actually increase the risk of being exposed to the first list problems.

How does the procedure deals with those problems

The procedure is trying very hard to limit requirements and dependencies, and yet get the maximum bang for our bucks.

In no particular order, here is what we get from it:

- Using “-m” forces the users to know what Python they use. It’s too easy to install something for one Python and call it from another one.
- Using a virtualenv removes most PATH, PYTHONPATH, permission and version issues, by design (but not all, hence -m even in venv). It will also prevent users for making more of a mess (like calling sudo).
- Using “py” on windows lets you choose the Python to use, even if you installed several of them from different sources. Some users have insane combinations from so many years of trying to stackoverflow their way out of suffering.
- Using the regular “python.org” installer increases the chance of getting a non-broken Python by the sheer power of statistics. On Windows, it also provides the “py” command, allowing for “-m”. And it’s fewer steps (E.G: you don’t need XCode or CLT for homebrew or install headers for pyenv).
- Using standard pip and venv means no chicken and eggs issue for installing the tools. Installing something outside of an env to manage an env is already setting up people for failure. So we use as few dependencies as possible. They work with “-m”

and you will find tons of materials to help you when things go wrong. They are cross-platform. They will still be here tomorrow.

- Using deadsnake and EPEL on Linux avoid compilation requirements, missing packages, and keep it fairly stable and standardized (to a point).
- All those tools work great with most editors, in CI, or in the most basic cmd.exe terminal with no color, nor fork, and no unicode support.
- Avoiding the latest major version of Python exposes the project to the part of the ecosystem that has a higher chance of compatibility, and a lower chance of bugs.

The most important is not specific to any of the steps: they all have have way, way less numerous and exotic modes of failure than the alternatives.

Not that they don't have any. Not that it will solve all problems.

But it makes a big difference.

Will the procedure work for everybody, in all the cases? Of course no. Nothing can. The combinations are enormous. Some are quite crazy.

But with a single tutorial, it will help the most people, on the biggest number of configurations.

Because the article must be straight to the point, clear, but keep the reader engaged, we have to limit what it says. So we couldn't put those explanations in there.

And even if we wanted to list the type of users it targets, it's not possible. There is no single description of everyone it can help.

This gives the impression the article is a bit pretentious: like it's sort of a divine light of wisdom, the one truth or something.

It's by design.

The people that will understand it's not for them will self-exclude from the recommendations, as they should, since they know what to do anyway.

For the others, it’s important to take them by the hand.

It’s also why the recipe repeats things a lot, giving it weird, but effective style.

Irritating a part of the readers is alright, they are not the target audience, and we had no way to filter them out without making the article less effective for the target audience.

Plus some people will complain that what you say sucks no matter the style or content, this can’t be helped.

More details on why not to promote homebrew/pyenv/poetry/anaconda

After all, they have being designed precisely to solve problems around installing python or python packages, and many devs find them useful.

In short: *using them is not simple*.

It’s ironic, considering they are praised for the things they simplify.

The number and complexity of modes of failure they come with, as well as the resources at the average user’s disposal to deal with those failures, provide a poor ROI.

First, let me start with the fact here is nothing wrong with homebrew, poetry, pyenv or anaconda. They are tools that were born to solve problems, and they do.

A part of the community will benefit from those tools, because the benefit they bring to them is higher than the cost they have.

However, it’s my experience that for the meaty part of the Gaussian curve of the user base, and I write for them as I wished someone wrote for me, the ratio cost/benefit is not in our favor.

I’ve had the pleasure of working as a developer and a trainer in about a hundred entities: companies, NGOs, schools… This is not a typo, a hundred is about right. I’ve seen all skill levels, a large spectrum of motivations or goals, many, many variations on constraints, so I’m confident that my sample is more representative than most.

Let’s talk about pyenv for example.

People that love pyenv are so adamant at telling everybody they should use it. How it solved all their problems, and got rich and lost 5 pounds.

It doesn’t support Windows. *That’s game over right there, for half of the community.*

Even if it did, pyenv compiles Python under the hood when you install it. But compiling can fail in a thousand ways. The world moved to binaries for distributing software for a reason. Plus people ask ChatGPT how to configure VSCode or exit vim, so we can’t settle on a workflow that involves more entropy than a Family Guy episode.

Even simpler methods of installation, like using the Microsoft Store or Homebrew will eventually explode. First, they don’t follow the naming conventions the community chose for their own system. “py” is not provided by the MS store, they don’t override the PATH, no, they use suffixes, unlike any other installation method on the OS. Homebrew will provide the wonderful trap of the “pip3” command, which caused more confusion by itself than easy_install and eggs combined. Worse, the Python in homebrew is not even intended for you to use. (That page recommends using asdf instead. Don’t.) Plus, you need to install Homebrew in the first place. I know you think it’s not an hindrance, but for some people, it is. Espacially because it’s usually never covered in the tutorial or documentation, like it’s obvious.

Both of those methods regularly install broken Python. Yes. Broken. Wrong folder permissions, badly linked dependencies, shadowing commands, you name it.

Poetry? I’ve used it for years, and every month, a new stack trace. I can solve that. That’s my job. But even I got tired of it after a while. And I really wanted to love poetry, it checks all the boxes in my book.

At this point in the article, someone is mumbling they never had any problem with poetry. I have a section dedicated to this.

But even without this, it has one show stopper from the start: how do you install poetry? Outside of a venv? Then you just made sure thousands of users will fail. Inside a venv? But poetry is supposed to manage your venv, which is what everybody tells you it’s great for. Wait, did you notice? You have to answer a difficult question before even using the tool. Provided you have an answer, that’s still one “if/else” branch you add to the whole decision diagram we must push to the user head.

Anaconda is probably the worst offender, because it’s so popular on Windows, particularly with the least technical user base. Their venv model has inheritance. Their channels are limited to a subset of packages. The YAML format(s) they promote come(s) full of gotchas. But the real danger is, if you mix “pip install” and “conda install”, one day, Davy Jones may very well spawn randomly to collect your soul.

When you look at it, it’s so unfair: the victim won’t be able to link their current problem with the decision they made months ago. They can’t.

So they will turn red, and express their new-found faith in Chtulu on reddit, tiktok or whatever will be the quickest way to rage-quit from pypi.

“Python packaging sucks!”

After this article brought many comments about pipx, I had to update it to cover it. It’s the same problem: more modes of failure.

On Mac, the doc tells you to install it with… homebrew. You now why it’s not a good idea now.

On Linux, it says to use the package manager, meaning you have no control about the version of Python you install it with. In fact, you don’t even know.

On Windows it recommand the obscure scoop tool.

Once you have installed it, you more possible ways to fail:

- It create implicit venvs. If you update your Python, those venv can break. Usually, you see the problem since you manage the venv. Here, people won’t have a clue.
- It uses an implicit Python version that will be different that some of your projects’ Python version. The parser behavior and valid syntax, the error messages, the

available libs, all that will be different, and potentially conflicting. Black, mypy or pylint can all fail.

- If your tools use python to define config (nox, doit, etc), you will be under the impression you can import your project files. That's not going to go well.
- If a tool needs to install/import things in your project, like for a plugin system (e.g: pytest), it will be chaos.

I'm not going to list all the tools, all their shortcomings. The list would become long, and out of date fast. Some things will have been fixed since them. New things will have broken.

Because that's another point: pip and venv are slow moving, but the rest of the tools change faster. Faster than a lot of devs can (or want to) keep up. Some come out into fashion only to fall into disgrace later on (remember when everyone loved pipenv?).

The majority of people wants to get things done, not to keep track of every single part of their stack. I read PEP for fun and install the trending Python packages on Github on my free time. But that's the exception, not the rule.

The idea is not that those tools are evil. I used several of them regularly.

The bottom line is: *provide a single procedure that will let most people do their job and have the highest possible rate of success.*

This means limiting the number of indirections. Something we do naturally when we code, so why is it that so many devs don't apply that to infra?

Now I hear you saying: "but they should learn how things work".

But what things? Where is the line? Why is this thing more important than this other thing? Days have 24H, and remember to do a 7 hours morning routine to start your day, cook organic locally sourced intermittent fasting, and have a Github activity history that looks like a bathroom wall. Now learn how to fix your car. You should know how it works… Wait, is it an EV or a gas car?

So python.org, "-m", pip and venv?

Yeah.

If this is all true, why didn’t I read that somewhere else?

For the same reason you will always find a lot of geeks that will tell you we all installed Arch Linux, and it worked without any problem.

We lie.

Sometimes knowingly, for the same reason people brag about their sports team.

But also without realizing it.

Technical people are blind to the fact they automatically solve dozens of problems every day in their regular workflow, any single one big enough to block another user for a few hours. Without even thinking about it.

E.G:

Sometimes my movie theater doesn’t output sound. I stand up, unplug a particular cable and put it back, and the sound roars again. One day, a friend of mine tried to use my system. I wasn’t here to tell her about this one-second-trick. She couldn’t use the system for a day.

Whether it’s moving to the proper directory, setting the PATH, changing one configuration value or running this cryptic command in a particular order, I guarantee **most good developers perform many, many, one-second-tricks a day** to have their stuff working.

And they are completely oblivious to it.

Sure, it’s not a big deal to install the Python headers for pyenv to work, or making sure PYTHONPATH points to the root directory of your project.

Except for most users it is.

There are usually two kinds of coders giving advises. A fresh one that has no idea how complex things really are, yet. Or an experienced one, that forgot it.

If you disagree with what I wrote so far, you are probably in the latter group. The only way to change your perspective is to give a Python training to a bunch of accountants that want to migrate part of their workflow from Excel, behind a corporate firewall and no admin rights on their machine. Also they use notepad++. Then, come back a year later to see how they are doing.

Remember: a huge number of Python coders are not devs, they are likely on windows, many of them never touched a terminal in their life.

As for the devs, believe it or not, a lot of them don't enjoy tinkering with their setup. They just want it to work, not try dozens of tools that will fail in surprising ways one rainy Friday morning. They may use <proprietary IDE you don't like> or <cloud provider you don't know exist> and their Python stack must work with it.

We, the people that do like tinkering, are actually a minority. A prolific vocal one, giving the illusion that what we do is how it should be done.

I heard you like modes of failure, so I put modes of failure in your modes of failure, so you can fail when you fail

The problem with being good at dealing with complexity, or being experienced in a particular field, which I suspect is not that different, is not only you can deal with a problem, but you can deal with the cascades that follows.

However, when you give somebody a way to do something, and it fails, you are usually not here to catch them when they fall, not from one step, but the whole stairway.

One day, one user, frustrated with not being able to run Python X.Y, hears that pyenv will solve that.

She decides to give it a try.

Now she has new problems to solve: learn to install pyenv, learn to use it, learn to adapt her workflow and toolchain for it and solve any error on the way.

She goes on, and notice it doesn't exist on windows, but the doc says you can use WSL.

So she decides to give it a try.

Now she has new problems to solve: learn to install WSL, learn to use it, learn to adapt her workflow and toolchain for it and solve any errors on the way.

But wait, WSL means she must get used to a whole new OS.

Now show has new problems to solve: learn to use bash, sudo, apt, one $EDITOR, learn to adapt her workflow and toolchain for it and solve any errors on the way.

Every time she digs, there is one more thing to learn, and for each thing, hundreds of ways to fail.

E.G. (one single very small step):

She is on Ubuntu WSL, how does she install Python 3 headers with apt, which are required for pyenv?

sudo apt install python3-dev ?

sudo apt install python3.x-dev ?

sudo apt install python-dev ?

Depending of the versions of Ubuntu, Python and the phase of the moon, some will exist, some not. It will also die when you run any of them, because WSL boots with a stale repository cache. If you don't run "sudo apt update" first, you'll get an error. If it works at all without deadsnake, because not all headers are in Ubuntu official repos.

But like with my sound system, you are not here to tell her to just unplug and plug back all those particular cables. All those one-second-tricks.

Most tutorials don't even mention that you have to do that.

Every single step is like this. And they branch. They compound. And all of them can fail.

All failure modes cascade.

And while she is doing all that, growing more and more frustrated, here is what she is not doing: what she wanted to code in the first place with Python.

Yes, but...

> -m causes other problems

It can indeed. But less than it solves. And more for people that can solve them.

> anaconda deals with compiled extensions

Thanks to wheels, pip handles most of them fine now.

> pyenv let me install all the python versions

In the same way nix lets you revert any update: provided you got it to work in the first place.

> pyenv let me switch the main version of python

It's a Pandora box. You should use a venv for that.

> poetry has a better dependency resolver

pip has had a new dependency resolver since 2020, on part with poetry's.

In fact, pip's is more flexible and won't die on you if the package specifies bad metadata, while poetry strictness may mean you can't install some packages at all.

> Tool x has better ergonomics

You got a point. The question is whether the ROI on that is favorable to the target users. My expectation is that people that apply the procedure will eventually growth and try new tooling if they have the resources and will, once they decide they need better ergonomics over simplicity and reliability.

> You should really use docker

I think you missed the whole point.

> pip-tools is great, why not use that?

I specifically don't talk about pip-tools, because I think once people have the procedure mastered, that's what they should use. But that's a separate article, that will mention you should get master pip and venv first. Otherwise it's like telling people to use an ORM without knowing SQL.

Why don't the core devs solve those problems?

> "Just copy cargo"
>
> "Provide an installer for linux"
>
> "Put the 'py' command everywhere"
>
> "Do something like node_modules".
>
> "Why don't you just…?"

I would need an entire article to cleanly answer this, but it boils down to:

- Python is very old and very popular. You can't be a cowboy.
- Python 2 => 3 took 15 years of outrage.
- Resources to work on Python used to be extremely limited. Now they are just limited.
- There are challenges you have not considered, such as political issues, security problems, frictions with OS maintainers, etc.
- There are many implementations of Python.

- Python is interpreted, with the added trick of having many compiled 3rd party modules containing Fortran, C, or even assembly.
- Python is in weird places. CI. OS. Plugins. WASM. JVM. .NET VM. Embedded devices…
- The packaging story actually improved a lot, believe it or not :)

“Just” is a terrible world to reflect the complexity of the solutions that need to be put in place.

The good news is there are very nice people that are working on it. Dive into the debate of PEP 582, give a try to the rust py launcher, go read Pradyun Gedam’s blog on the topic (pip maintainer), for some interesting insights into the ever-bubbling packaging brainstorming.

Discussion about this post

JustAQ Mar 26

I'm a C and C++ dev, using some python tooling here and there to make my life easier. I've been using -m since forever. I kinda settled into it naturally. What issues can it cause?

LIKE REPLY SHARE

1 reply by Bite Code!

Michał Sep 6, 2023 Liked by Bite Code!

I have to tell, first you make me mad! I did not agree. pyenv - works, packaging work, installing on system wide with pip works... Then I realize you kind of right...

I'm maintainer a small windows desktop app (probably used by less then 100 people). It available as python package at PyPI, but be installed with pip. I did what most developer would do release as whl, allow to install and been updated with pip, simple stuff.

But it turn out most of users are non-developers... I realize it when I got really simple question like: that is terminal, what is cmd, where to put command (pip install dcspy), I got weird output, it is correct or not.... Once, one user reinstall whole windows because he got some strange message/question (normal from developer perspective behavior).

So, yes... we developers solve dozen of simple problem daily, don't realize it... love to tinkering...

So now I just deliver one-file executable (done with pyinstaller/nuitka) and keep wheel as backward compatibility.

Great reed...

LIKE (2) REPLY SHARE

26 more comments...