Episode Book

EPISODE #6638

How Python Wheels Work Without Running Code

PUBLISHED	16 November 2026 at 21:56 UTC
DURATION	23:44
LISTEN	myweirdprompts.com/episode/how-python-wheels-work-73309/
HOSTS	Corn, Herman, Hilbert
SCRIPT MODEL	z-ai/glm-5.3
VOICES	omnivoice
PIPELINE	V5.34
SOURCES	12 cited by the research (listed at the end)
BOOK GENERATED	16 November 2026 at 22:22 UTC

SYNOPSIS

A Python wheel is just a ZIP archive that installs packages by copying files without running any build code. That simple format, defined by PEP 427, split building from installing and ended the era of compiling Fortran and C on every machine. The conversation traces how compatibility tags like py3-none-any, manylinux1, and the RECORD uninstall manifest work, why "Building wheel for..." actually means a fallback to source, and how PEP 517's frontend-backend split enabled today's crowded ecosystem of tools like pip and uv.

TRANSCRIPT



Corn

Every Python developer knows the guilt. You type pip install, the terminal says Building wheel for something, pages of output scroll past, and you just... let it happen. Daniel wants to know what he's been agreeing to all these years. He uses wheels constantly, sees the dot-whl files everywhere, and admits he's never really understood what one is — what it is, why it exists, what pip or uv actually does when it installs one, and why the whole packaging world feels like a filing system designed during an earthquake.



Herman

That last part is the fun part, honestly. Because the answer to the first part is embarrassingly simple, and the answer to the last part is a twenty-year soap opera.



Corn

Let's start with the file itself, because it's less mysterious than it looks.



Herman

A wheel is a ZIP archive with a dot-whl extension. That's it. You can rename one to dot-zip and open it with anything that opens ZIPs. Inside are exactly the files that need to be copied into place to install the package. No build step, no compiler, nothing executes. The spec's own phrasing is that it contains the files "that need to be copied when installing the package," and they mean that almost literally.



Corn

No code runs during the install.



Herman

None. That's the entire point. The filename is where people get spooked, because it looks like noise — pip twenty-three dot three dot one, py3, none, any, dot whl. But that filename is a data structure. The last three fields are compatibility tags: py3 means any Python three implementation, CPython or PyPy, doesn't matter. None means no ABI dependency, no compiled extension coupled to interpreter internals. Any means no platform dependency. So py3-none-any is the common pure-Python case — one wheel serves everybody.



Corn

So the filename is doing the job of an eligibility test before you've even opened the file.



Herman

Right, the installer can filter purely on the name. Should it run on this machine with this interpreter — check the tags, done, no need to unpack anything. Inside the archive you've got the package's own files sitting at the root, a dot-dist-info directory holding the metadata, and optionally a dot-data directory with subdirectories that map to install locations — purelib, platlib, headers, scripts, data.



Corn

Herman, quick show of hands. Before this week, could you have told me what py3-none-any meant?



Herman

The any, yes. The middle one I'd have hand-waved. I'm not proud.



Corn

You, the man who read a spec for a filing cabinet once.



Herman

The filing cabinet spec was interesting and I stand by it. But no, this is the thing about wheels — everyone treats the filename as wallpaper, and it's actually load-bearing. That's the pattern for the whole episode, actually. The file is simple. The story of why it exists and what it displaced is where the complexity lives, and a little pain.



Corn

So take us back to the pain.



Herman

Two thousand. You want to install a package. You download a tarball, you extract it, you run python setup.py install. There is no dependency resolution. There is no uninstall command. There is no record of what got installed or where. If you want rid of it, you delete files by hand and hope you found them all.



Corn

Hope as a package management strategy.



Herman

And it gets worse, because setup.py install doesn't just copy — it runs the project's build code on your machine. PEP 427, the 2012 proposal that defined the wheel, diagnosed this directly: sdists "require the distutils and setuptools build systems, running arbitrary code to build-and-install, and re-compile, code just so it can be installed into a new virtualenv." Their phrase was "conflating build-install" — slow, hard to maintain, and it hobbles both sides, because the build system and the installer are welded together and can't improve separately.



Corn

Give me the concrete version. What did that actually feel like?



Herman

Installing NumPy meant compiling Fortran and C on your machine. Fortran! Installing lxml meant having the libxml2 development headers already present. A failed build produced pages of compiler output that most Python developers had no prayer of reading. And the really galling part in PEP 427's telling — you'd compile it once, then move to a new virtualenv and compile the exact same thing again, because nothing was ever saved.



Corn

Recompiling identical code as a hobby.



Herman

The wheel's three promises, straight from the PEP: it frees installers from knowing anything about the build system, it saves time by amortizing compile time across many installations, and it removes the need to have a build system in the target environment at all. Somebody with a big machine compiles it once, uploads the artifact, and everyone else copies files.



Corn

That's the part I think deserves more credit than it gets. The deep idea isn't "ZIP files are nice." It's splitting building from installing so the halves can evolve separately.



Herman

And that split is exactly what made today's tool zoo possible. Once a build system only has to produce a standard wheel, hatchling, flit, PDM, setuptools and uv build can all coexist, because the interface between them and the installer is a file format, not a piece of code running on your machine. PEP 517 in 2015 cashed that in. The tool diversity people complain about is downstream of the wheel succeeding.



Corn

The format solved the problem so well it created the conditions for the next complaint.



Herman

Each generation solved one problem and created the next. That's the arc of the whole thing.



Corn

You said pure Python was solved immediately. Wasn't there a gap for compiled packages?



Herman

A big one. A wheel containing compiled C works fine on the machine it was built for — but a wheel built on Ubuntu wouldn't run on CentOS, because the two shipped different system libraries. So for years the projects that needed wheels most, the NumPy-type projects, still fell back to compiling locally, and the wheel was this great thing that only helped the easy cases. PEP 513 in 2016 defined manylinux1: a minimal set of system libraries nearly every Linux distribution provides. Build against that set, tag the wheel manylinux1, and one artifact covers basically the whole Linux world. Follow-up PEPs refined it, and cibuildwheel later automated building the whole matrix across Python versions, operating systems and architectures in CI.



Corn

Which is why a random small project can now ship a binary wheel for an ARM Mac without owning one.



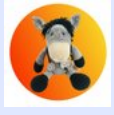
Herman

And there's a lineage here too, because wheels weren't the first attempt. Setuptools introduced the egg in 2004 — also a ZIP, also trying to solve this. But the egg had no official specification, it was both a distribution format and a runtime installation format, you could import straight from the zipped egg, which is a design decision with opinions. Wheel is distribution-only, uses dot-dist-info instead of egg-info, and carries a version. Since August 2023, PyPI rejects egg uploads outright.



Corn

The egg got retired with full honors.



Herman

Full honors and a locked door.



Corn

Okay, that's the why. Now the mechanics. I type pip install, I hit enter, and something arrives. What actually happens?



Herman

Two phases, and the spec names them: Unpack, then Spread. Unpack: the installer reads the WHEEL file inside dot-dist-info — that's metadata about the archive itself, not the package — checks its own wheel spec version against it, and unpacks the archive either into purelib or platlib depending on a field called Root-Is-Purelib. Then Spread: the dot-data subtrees get moved onto their real destinations, scripts starting with the magic shebang line get rewritten to point at your actual interpreter — that's how the pip command itself ends up pointing at the right Python — RECORD gets updated with the installed paths, the now-empty data directory is removed, and any dot-py files are compiled to bytecode.



Corn

Which is why installing never needs a compiler. It's a coordinated copy with a haircut.



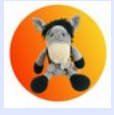
Herman

I'm keeping that for a dinner party. RECORD is the part I find quietly brilliant, though. It's a CSV, one line per installed file: path, hash, size. The hashes are sha256 in a particular encoding. And it's the uninstall manifest. To remove a package, a tool deletes every file in RECORD, the matching bytecode files, and any directories that empty out as a result. If RECORD is missing, tools are forbidden from attempting an uninstall or upgrade at all — they're not allowed to guess.



Corn

So uninstall went from "delete files and hope" to a formal promise.



Herman

One file, one column of hashes, and suddenly deletion is deterministic. It's the most boring possible solution and it's exactly right. Same spirit as the no-bytecode rule — wheels don't generally include dot-pyc files, because the whole point is one wheel working across Python versions, and shipping bytecode would force one wheel per Python version instead of one single wheel. Installers generate it at install time.



Corn

I want to put the scary message on the table, because it's the thing Daniel explicitly flagged. Building wheel for such-and-such. Everyone reads that as "installing a wheel."



Herman

It's the opposite. That message means pip could not find a compatible wheel for your platform and interpreter, so it fell back: download the sdist, invoke the project's build backend in an isolated environment, build a wheel from source on your machine, then install that wheel. The message is the sdist path wearing the wheel's name. It's misleading wording, and it explains basically every "why is this slow," "why did it need a compiler," "why did it fail" experience people have.



Corn

So when the terminal is spewing compiler errors under a banner that says wheel, the honest version of that banner is "no wheel existed for you."



Herman

You're watching a fallback happen in real time.



Corn

And uv? We've done pip versus uv to death — the Rust rewrite, the speed, all of it — so let's not rehash. Same format underneath?



Herman

Same format, same preference order. Both prefer wheels and fall back to sdists. The difference is speed of resolution and install, not what gets installed or where it lands. uv is not installing a different kind of object.



Corn

So the format is settled. The install is unpack-and-spread. Then why, Herman, does the ecosystem still feel like an unattended group project?



Herman

Ha. Several layers to that. Layer one: the frontend-backend split is elegant but invisible. PEP 517 says a build frontend takes source trees or sdists and produces wheels, and the actual building is done by each project's build backend, named in pyproject.toml. When you run pip install lxml, pip is what they call an integration frontend — it goes and finds the backend and delegates.



Corn

And pyproject.toml is the contract between them.



Herman

The build-system table names the module. Setuptools dot build-meta, hatchling dot build, flit core dot buildapi, pdm dot backend, uv build. If the file's absent or the backend's missing, tools revert to legacy setup.py behaviour — the old ghost still lives under the floorboards.



Corn

The floorboards creak in setup.py.



Herman

They do. And there's a rule in PEP 517 that I find telling: the build-requirements graph must not contain cycles, and front ends should use wheels for build requirements where practical, to avoid deeply nested builds. Because your build backend is itself a Python package that may need building, which needs a backend — the snake eats its own tail unless the format breaks the cycle. Bootstrapping a build backend is a special case for exactly this reason.



Corn

Layer two, then. If the format's settled, why fourteen tools?



Herman

Chris Warrick counted fourteen packaging tools in 2023 and called that "at least twelve too many," which is a great line and also a diagnosis. Part of it is that the format enabled the diversity — once the interface is a file, anyone can write a producer or consumer. But part of it is structural. PEP 517 laid out the trap with unusual candour: distutils and setuptools are missing features, extending them is difficult, and yet it's very hard to use anything else because they're the standard interface expected by users and installers alike. The stated goal of the PEP was to get the standards body out of the business of being a gatekeeper for build systems. They opened the gate, and the crowd that came through is the crowd.



Corn

And there's no bouncer.



Herman

LWN's survey put it flatly: the PyPA has no real authority in the community. It leads by example, and its recommendations, even in the form of PEPs, are simply that. Tool authors can and do ignore or skirt them. There's a whole parallel universe — conda uses its own format that predates and ignores the standards pip uses. NumPy's own install page steers you toward conda first.



Corn

Two currencies, one economy, no exchange rate set by law.



Herman

And one more structural thing that people constantly misattribute: only one version of a package can be active per Python instance. If application A needs version one of a dependency and application B needs version two, only one can be satisfied — the import statement can't pick per application. That constraint, not any tool's incompetence, is why virtual environments exist at all. And then pip only looks for dependencies at PyPI, while your system package manager has its own ideas, and the two can install the same thing twice and get crosswise.



Corn

Herman, let me push on one thing, because I don't quite buy the "format is settled, friction moved upstairs" story clean. Wheels are specialized per operating system, architecture, C library, Python version. One sdist per release, many wheels. Who builds the many?



Herman

Each individual project. PyPI relies on the projects themselves to build all the wheels users might need, which distributes the burden but leaves gaps for projects without the resources of a large build farm. So the format is settled, but the coverage is only as good as whoever's got CI configured.



Corn

So the gap is exactly where the money isn't. That's not a settled format, that's a settled format with holes in it where the maintainer has a day job.



Herman

That's fair. I'd still say the remaining wheel-shaped pain is much smaller than the tooling-shaped pain. On the wheel side, we did an episode a while back on safetensors, the format model weights ship in, and the same shape showed up there — the format itself was the boring part, and all the arguing lived in the tooling around it. Same lesson, different file extension. The arguing now is over project management, dependency resolution, environment handling — the layer above the artifact, not the artifact.



Hilbert

Medieval Pottage Facts. The church took a tenth of every crop as tithe, owed whether the harvest had failed or not. People paid it. The weather did not care.



Corn

Cheery.



Herman

So the wheel matrix problem — that's also why the Building wheel message clusters on unusual setups. You're on a niche Python build, or an odd architecture, and the tags don't match anything on the index, and down you go the sdist path.



Corn

Which is a compiler arriving at your door because your door was the wrong shape.



Herman

And that's why the mismatch usually isn't anyone's fault in particular. You're just the one holding the shovel.



Corn

Let me ask you something practical, then. You've got a machine in the flat — the one you set up last month — and it sat on Building wheel for eleven minutes.



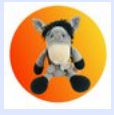
Herman

It did, and I want to defend it.



Corn

Eleven minutes, Herman.



Herman

It was building lxml, and I was on a Python point release the project hadn't built wheels for yet, so it fell back and compiled libxml2 bindings locally, and yes, it took eleven minutes, and then it worked perfectly and I never thought about it again. That's the sdist path doing its job. The fallback exists so that the answer is "slow" instead of "no."



Corn

Fair. Corn, your test — would you rather it failed instantly?



Herman

No. I'd rather read a book. But I'd also like it to tell me that's what's happening, in words, before I've made tea.



Corn

Right, the message is the real usability crime. The mechanism is honest, the signage is a lie.



Herman

Guido van Rossum had a line about packaging once — nothing ever happens, and then somebody ships a terrible hack and everyone adopts it and progress stalls for another seven years. The wheel is one of the rare counterexamples. It went through a real PEP, it's good, and it stuck.



Hilbert

The hash isn't ordinary base64. It's urlsafe, no padding. You'd know that if you'd ever edited a RECORD file by hand, which I have.



Herman

Hilbert, have you actually...



Hilbert

Sat with a printed RECORD file? Yes. I had a package that kept reinstalling itself every time I removed it. Drove me spare for an afternoon. I was deleting the package files by hand, thinking that was that, and there was a stray dot-dist-info directory still sitting there with RECORD listing every path. So the machine kept concluding it was still installed. Corn would have found it in ten minutes. Corn naps through most of the afternoon, but the ten minutes he's awake are well spent.



Corn

That's the most efficient compliment I've ever received.



Hilbert

I kept the RECORD file. Printed it out, it's in a drawer. I can still tell you the first four characters of the sha256 of a file I deleted years ago. Later I tried fixing a broken install by editing RECORD in a text editor, got the hash format wrong, and spent another hour convinced the package was cursed before I found the padding.



Corn

You hand-edited a hash manifest.



Hilbert

I hand-edited a hash manifest. It was that or reinstall, and the reinstall had already lied to me once.



Herman

The thing I love about that is you went straight past the ZIP and the tags to the uninstall manifest. Everyone talks about the archive and the compatibility tags, and the bit that actually saves you is the CSV listing where everything went. He's right, Corn, that's the underrated corner of the whole spec.



Corn

It's also the part with consequences. Get the tags wrong, pip shrugs and finds another wheel. Get RECORD wrong and the machine starts lying to you about what exists.



Hilbert

The manifest never lies. It just gets lied to. The installer believes it, and the installer is the one with the hands.



Corn

So the design lesson is the boring one. The list of files is what you keep, not the files.



Hilbert

The list of files is why I still have the drawer.



Corn

Where does that leave the fourteen tools, though? If the format enabled the diversity, what stops the next one?



Herman

Nothing, really, and that might be the honest answer. A standards body with no enforcement power, a parallel conda universe ignoring the standards, and the one-version-per-interpreter constraint that forces virtual environments to exist regardless of what any tool does. The Hacker News question from a few years back — why does every package and module system become a Rube Goldberg machine — got two hundred answers and no satisfying one.



Corn

My suspicion is the Rube Goldberg machine is what a real dependency graph looks like when you're forced to look at it. Every language gets there eventually because the problem is combinatorial — versions, platforms, interpreters, system libraries — and the machine is just the shape of the truth drawn badly.



Herman

Or drawn as well as anyone's managed. Guido's seven-year stall, Warrick's fourteen tools — that's what a field with no authority looks like between real breakthroughs. The wheel was a breakthrough. Nothing since has been at that scale, so the layer above it churns.



Corn

The wheel didn't make packaging simple. It made one honest artifact and left the argument standing.



Herman

And the argument got more productive, because at least now it's about tools rather than about whether your install finishes at all. In the sdist era the floor was "hope you have Fortran." Today's floor is "pick one of fourteen doors." Ugly, but every door leads somewhere.



Corn

The forward thought for me is the gap Herman flagged on the wheel matrix. Everything works beautifully as long as somebody with spare CI hours builds the artifact for your corner of the matrix — and the moment nobody does, you're back to compiling Fortran. The format can be perfect and still fail you through absence, not design. That's the thing to watch: whether building the full matrix gets cheap enough that the gaps close on their own.



Herman

Producer credit to Hilbert Flumingtop, as ever, who apparently also functions as our in-house manifest auditor.



Corn

If this was your kind of episode, go back for episode nineteen, AI Images; and episode twelve, The AI Breakthrough. This has been My Weird Prompts. If today's episode finally explained a message you've been scrolling past for years, the review button is right there — tell another person who's scared of their terminal.



Herman

Reviews, the website, my weird prompts dot com, all of it. See you soon, everyone.

SOURCES

What the research for this episode read before the script was written. Primary sources first.

1. **Python Packaging User Guide, Wheel distribution format specification , retrieved 2026-11-16. [primary]**
   https://packaging.python.org/en/latest/specifications/binary-distribution-format/

2. **PEP 427, The Wheel Binary Package Format 1.0, Daniel Holth, created 2012-09-20, accepted 2013-02-16, status Final , retrieved 2026-11-16. [primary]**
   https://peps.python.org/pep-0427/

3. **PEP 517, A build-system independent format for source trees, Nathaniel J. Smith & Thomas Kluyver, created 2015-09-30, status Final , retrieved 2026-11-16. [primary]**
   https://peps.python.org/pep-0517/

4. **Python Packaging User Guide, Package Formats , retrieved 2026-11-16. [primary]**
   https://packaging.python.org/en/latest/discussions/package-formats/

5. **Python Packaging User Guide, Recording installed projects (.dist-info, RECORD, INSTALLER) , retrieved 2026-11-16. [primary]**
   https://packaging.python.org/en/latest/specifications/recording-installed-packages/

6. **Python Packaging User Guide, Packaging Python Projects (build backends, pyproject.toml) , retrieved 2026-11-16. [primary]**
   https://packaging.python.org/en/latest/tutorials/packaging-projects/

7. **pydevtools handbook, How Python Package Formats Evolved: From tar.gz to .whl, retrieved 2026-11-16.**
   https://pydevtools.com/handbook/explanation/how-python-package-formats-evolved/

8. **LWN.net, A survey of the Python packaging landscape, 2023, retrieved 2026-11-16.**
   https://lwn.net/Articles/920132/

9. **Wikipedia, Python Package Index, last edited 2026-09-22, retrieved 2026-11-16.**
   https://en.wikipedia.org/wiki/Python_Package_Index

10. **Hacker News, Ask HN: Why does every package+module system become a Rube Goldberg machine?, 2023-01-30, retrieved 2026-11-16.**
    https://news.ycombinator.com/item?id=34577844

11. **Chris Warrick, How to improve Python packaging, or why fourteen tools are at least twelve too many, 2023-01-15 (seen only via search snippet; full text not fetched), retrieved 2026-11-16.**
    https://chriswarrick.com/blog/2023/01/15/how-to-improve-python-packaging/

12. **Carpentries Incubator, A History of Python Build Tools (seen only via search snippet), retrieved 2026-11-16.**
    https://carpentries-incubator.github.io/python_packaging/04-history-of-packaging.html