Using virtual environments to manage software

Henry Schreiner • 19-9-2025 • HSF-India

About me

Henry Schreiner

RSE/Computational Physicist
@ Princeton University

https://iscinumpy.dev

henryiii

PyPA member

Scikit-build admin

Scikit-HEP admin

pybind11 maintainer

CLI11 author

Scientific Python Development Guide author

About me

Henry Schreiner

https://iscinumpy.dev

henryiii

RSE/Computational Physicist
@ Princeton University

PyPA member

Python packaging

Scikit-build admin

Scikit-HEP admin

HEP & Python

pybind11 maintainer

Compiled Python interface

CLI11 author

Scientific Python Development Guide author

ISciNumPy.dev

A random collection of Science, Numerics, C++, CMake, and Python related topics.

Favorite posts and series

C++ 11 14 17 20 23 • macOS (AS) / Windows Setup • Azure DevOps (Python Wheels) • Conda-Forge ROOT • CLI11 • GooFit • cibuildwheel • Hist • Python Bindings • Python 2→3 3.7 3.8 3.9 3.10 3.11 3.12 3.13 3.14 • SSH

My classes and books

Modern CMake • CompClass • se-for-sci

My workshops

CMake Workshop • Python CPU, GPU, Compiled minicourses • Level Up Your Python

CMake Workshop • Python CPU, GPU, Compiled minicourses • Level Up Your Python • Packaging

Python libraries

pybind11 (python_example, cmake_example, scikit_build_example) • cibuildwheel • build • packaging • pipx • dependency-groups • pyproject-metadata • nox • scikit-build (core, cmake, ninja, moderncmakedomain) • meson-python • boost-histogram • Hist • UHI • Vector • GooFit • Particle • DecayLanguage • Conda-Forge ROOT • uproot-browser • Scientific-Python/cookie • repo-review • validate-pyproject(-schema-store) • flake8-errmsg • check-sdist • pytest GHA annotate-failures • Plumbum

Other projects

CLI11 • beautifulhugo • Jekyll-Indico • POVM • hypernewsviewer • AoC 2023 • AoC 2024

My sites

Scientific-Python Development Guide • IRIS-HEP • Scikit-HEP • CLARIPHY

Why did we use Binderhub?

Easy to get everyone on the same page

Can give you access to Linux, GPUs

Don't have to download over the shared network

But what about after this workshop?

Why can't I just ask an LLM?

Instead of listening to this talk

AI helps when you know you what to do

But it's kind of terrible at best practices
when there are a lot of bad examples!

This will give you an overview of how it works
(then you can get the LLM to help!)

Places to get packages

PyPI vs. conda-forge

PyPI	conda
Supported by PSF / standards	Supported by Anaconda
Install with uv or pip	Install with conda/mamba/pixi
Python focused	Language agnostic
Open source, PyPA	Open source recipes: conda-forge
Built by package authors	Built by shared infrustructure

Where do you get Python itself?

PyPI vs. conda-forge

PyPI	conda
Official installers	Just a another package!
Your system package manager	
Third party package managers	
uv python install	
conda	Warning: conda has it's own compiler stack

You can mix, but avoid if possible!

Size and performance

PyPI vs. conda-forge

PyPI	conda
No external dynamic linking	Can link between packages
Self contained	Binary dependencies
Very small packages	Very large installs
Can be slower*	Can be faster*
Immediate access to new versions	Can be delayed

*Conda can target arch levels, shared threadpools, etc, but package specific.

So which one?

If it was easy, there wouldn't be two :)

Package selection varies

Tooling varies

Environment practices varies

Performance varies

Compiling code varies

Which tool?

Conda family

Conda

Written in Python

Resolver now fast (libmamba)

Has environment setup function

Mamba

Written in C++

Fast

Pixi

Written in Rust

Very Fast

High level interface (only)

Low level interface

Conda / Mamba

```
mamba env create -n myenv
mamba activate myenv
mamba install -c conda-forge numpy
```

Low level interface

From file

```yaml
name: myenv
channels:
  - conda-forge
dependencies:
  - jupyterlab >=1.2
  - nb_conda_kernels
  - numba >=0.57
  - root >=6.18.04
  - pip:
    - awkward >=2.0.0
```

```
mamba env create -f environment.yml
```

Reproducible environments?

Why a high-level interface is nice

What if you want the same environment ~~next year~~?

tomorrow?

Standard solution: lock files
(some tools now allow limiting date, too)

High level interfaces make this easy!

High level interface

Pixi

pixi.toml

```toml
[workspace]
authors = ["Jane Doe <jane.doe@example.com>"]
channels = ["conda-forge"]
name = "my_project"
platforms = ["osx-arm64"]
version = "0.1.0"

[dependencies]

[tasks]
```

Will make a lockfile automatically!

```
pixi run <task>
```

High level interface

Pixi

You don't even have to write the file!

`pixi init`	Makes pixi.toml
`pixi add pytest`	Updates pixi.toml
`pixi run pytest`	Makes pixi.lock

This is all that is needed
for an existing project!

Examples

Pixi

```
git clone https://github.com/se-for-sci/se-for-sci.github.io

cd se-for-sci.github.io

pixi run lab

pixi run book

pixi run pdf
```


~/g/t/se-for-sci
$…
7.7s
Wed Sep 17 17:16:16 2025

Which tool?

PyPI family

Pip	uv
Written in Python	Written in Rust
Low level interface	High and low level
Easily available	Single binary
Slow	Very fast (10-50x)
Used by other tools	Replaces other tools (used by some, too)

Low level interface

pip / uv

```
python -m venv .venv
. .venv/bin/activate
pip install numpy
```

```
uv venv
uv pip install numpy
. .venv/bin/activate
```

Also lockfiles, building packages, etc.

High level interface

uv

```
uv init
uv add pytest
uv run pytest
```

This is all that's needed
for an existing project!

Use uv.toml or pyproject.toml

No task support (yet)

Supports PEP 723: Single file scripts

Automatic lock file generation/usage

Can be used with standardized config

Example: compiled code uv project

1: example.cpp

```cpp
#include <pybind11/pybind11.h>

PYBIND11_MODULE(example, m) {
  m.def("add", [](int i, int j){return i + j;});
}
```

Example: compiled code uv project

2: CMakeLists.txt

```cmake
cmake_minimum_required(VERSION 3.15...4.0)
project(${SKBUILD_PROJECT_NAME} LANGUAGES CXX)

find_package(pybind11 CONFIG REQUIRED)

pybind11_add_module(example MODULE example.cpp)
install(TARGETS example DESTINATION .)
```

Example: compiled code uv project

3: pyproject.toml

```toml
[build-system]
requires = ["scikit-build-core", "pybind11"]
build-backend = "scikit_build_core.build"

[project]
name = "example"
version = "0.1.0"
dependencies = ["numpy"]

[dependency-groups]
dev = ["pytest"]
```

Using it

With a quick pytest file

```python
import example

def test_add():
    assert example.add(1, 2) == 3
```

```
uv run pytest
```

That's all that is needed!

Examples

uv

```
git clone https://github.com/scikit-hep/hist
cd hist
uv run pytest
```

Bonus: single file scripts

PEP 723

```python
# /// script
# dependencies = ["rich"]
# ///

import rich

rich.print("[blue]Hello world")
```

Other runners too!

```
uv run myscript.py
```

Bonus: single file scripts

PEP 723

```
#!/usr/bin/env -S uv run –script

# /// script
# dependencies = ["rich"]
# ///

import rich

rich.print("[blue]Hello world")
```

./myscript.py

Bonus: Running PyPI applications

uvx / uv tool run

```
uvx <PyPI application>

uv tool install <PyPI application>
<PyPI application>
```

Creates an isolated environment,
and exposes the executable!

Pixi supports this too!

uv: bonus features

It can do a lot that pip can't!

uv can install Python

uv can limit packages by date

uv defaults to .venv, requires `--system` flag to break stuff

Doesn't install itself into venvs

Can resolve oldest requested packages instead of newest

Can build and publish packages, too

Other things of note

Task runners:
Tools that set up environments and run specific tasks, like "tests" or "docs". See "nox" (also "tox", "hatch", "pixi")

Build backends:
The library that puts together a Python package. "hatchling", "uv_build", and "scikit-build-core" are examples.

Linting:
Checks for possible problems by static analysis of code. "ruff"

Type checker
Verifies static types in your code (like a compiler). "mypy", also "ty" (new)

Linter runner:
Runs quick checks, integrates with git. "pre-commit"

Making a new pip-installable project

Template

`uv init` Simple, has options

`uvx cookiecutter gh:scientific-python/cookie`

Complete, also has options

Other languages

Interpreted languages all have similar issues and similar solutions

npx is like uvx for Javascript, for example

Compiled languages only need to do this at compile time (see Rust)

uv was designed based on cargo, Rust's package manager

Other things

WebAssembly, binder

There are online services for running notebooks

Binder • Google CoLab

You can also use WebAssembly, with some limitations

Pyodide on JupyterLite