Packaging ROOT for Python

Progress & Challenges

Silia Taider[1], Vincenzo Eduardo Padulano[1]

[1] CERN EP-SFT

PyHEP.dev 2026 - 08-09-2026

https://root.cern

Provide a unified software package for the storage, processing, visualisation and analysis of scientific data that is reliable, performant, supported and sustainable, that is easy to use and obtain, and that minimises computing resources and scientists' time needed to achieve results.

The success of experiments and all ROOT users at large is our priority

- **ROOT is a Python package which reaches users through several channels:**

CVMFS / LCG stacks	Pre-built releases on a distributed filesystem
Pre-built binaries	Linux, macOS, Windows - downloaded and unpacked directly
OS package managers	Homebrew, dnf
conda-forge	Community-maintained channel, adopted in HEP
pip	The standard Python installer

Today we focus on conda and `pip`

Today's talk

1. Things to consider when packaging ROOT
2. Conda-forge: status and challenges
3. Pip: status and challenges
4. Outlook and next steps

- **`ROOT` embeds `cling`, a full C++ interpreter, plus `cppJIT` which generates Python-C++ bindings dynamically**
- **Most compiled Python packages need a C++ toolchain once: when the wheel author builds it**
  - `ROOT` needs one again **every time a user runs `import ROOT`**
    → `cling` compiles code on demand on the user's machine
    → This is what offers the feature of parsing, compiling and executing arbitrary C++ code at runtime within a Python session
  - ROOT is not completely alone in needing a C++ toolchain, but others don't go as far
    → `numba` ships a lightweight version of llvm to make jit compilation work, but doesn't expose as much features as ROOT does

conda-forge

https://root.cern

- **conda-forge is the community-maintained channel most of the scientific Python stack lives on**
  - NumPy, SciPy, etc.
- **ROOT joined that shared infrastructure**
  - Thanks to an initial huge contribution by H. Schreiner and C. Burr!
- **For the ecosystem-wide coordination story, see:**
  - HEP Packaging Coordination: Distributing the HEP software ecosystem on conda-forge
    M. Feickert, C. Burr, G. Stark - PyHEP.dev 2026

Two repositories required to make it work:

- clangdev-feedstock: clang on conda-forge, with a separate package variant for patches required by ROOT
- root-feedstock: The actual repository for building ROOT

Clearly ROOT has more dependencies, but the clang feedstock variant needs to be kept in sync with the ROOT feedstock

Clang variants are visible and tracked depending on the corresponding version of clang needed by ROOT, e.g.

- ROOT 6.40.x is based on clang 20, the variant is managed in the clangdev-feedstock:20.x branch

Current status of the project:

- All major ROOT releases are available on conda-forge
- Available on MacOS, Linux (x86, ARM)
- Minor releases with backports and bugfixes are shipped to conda-forge if they are defined **"core"** under the description at https://root.cern/install/all_releases/
- 72.3K downloads in the last 6 months
- > 70 packages on conda-forge depend on ROOT (`root` and `root_base`)
  - `mamba repoquery whoneeds [root_base, root] --channel conda-forge`
- > 40 issues fixed in the past 18 months

Outlook for this ROOT distribution:

- Aim at reducing to zero the conda-forge patches to ROOT proper
- Address the sysroot incompatibilities (see section on pip for more details) → see root-feedstock/issues#336
- Re-adding the conda nightly builds (more complicated to maintain in the long-term)
- As usual, help is welcome as well as feedback on what to improve!

pip

https://root.cern

`pip` searches an index for xyz.whl and picks the right one for your platform

- module → sdist (source archive, built on install) → wheel.whl (precompiled, ready once uncompressed)

Wheels need a build backend

- `pip` only installs, a backend compiles the code and assembles the archive

Wheels are tied to a platform

- A platform tag like `manylinux_2_28_x86_64` names which machine it'll run on

- **V. Padulano & J. Rembser built the first demonstrator for the ROOT experimental wheels in 2024**
  - pip install ROOT: experiences making a complex multi-language package accessible for Python users
    V. Padulano, J. Rembser - CHEP 2024

```
$ pip install ROOT -i https://root-experimental-python-wheels.web.cern.ch
```

Next steps defined at the time:
Strengthen the build & split the package

setuptools → scikit-build-core

- **ROOT migrated away from setuptools to `scikit-build-core` as it's build backend, chosen for native CMake interoperability**
  - Expresses build options directly instead of assembling strings and shelling out by hand
  - A wider range of functionality like of dynamic metadata - `scikit-build-core 1.0`
  - Solved issues for certain CMake-based projects linking against the wheels
- **ROOT uses `cibuildwheel` to orchestrate the build workflow**
  - Runs that backend across every supported platform from one pipeline
  - Runs nightly

ROOT is building a standard robust packaging pipeline

- **Linux [x86, ARM] wheels build inside manylinux containers**
  → Docker images maintained by PyPA that pin an old-enough glibc, so the wheels runs on nearly any Linux distribution, not just the one that built it

- **MacOS [new!] wheels build directly on native GitHub Actions macOS runners (`macos-15`, `macos-26`)**
  → there's no container equivalent, so the runners SDK becomes part of the compatibility contract

ROOT 6.42

Run `pip install ROOT` on your mac!

- The same install, on two systems that differ just enough, see conda-forge/root-feedstock/issues/336
  → Issue observed with both `conda-forge` and `pip` distributions

✅ ALMALINUX:9

```
$ pip install ROOT
$ python -c “import ROOT;
ROOT.gInterpreter.Declare(‘#include <sys/socket.h>’)”

→ True
```

❌ ALMALINUX:10

```
$ pip install ROOT
$ python -c “import ROOT;
ROOT.gInterpreter.Declare(‘#include <sys/socket.h>’)”

→ error: unexpected type name ‘sa_family_t”:
expected expression
```

Context:

- A user install ROOT as a Python package
- On a system with a different glibc or C++ stdlib, different enough that:
  - API changes in headers
  - Missing symbol in the library
- ROOT exposes a full C++ interpreter
  → Compilation can fail for the reasons above

Break the assumption that the build environment and run environment are exactly the same

Challenges - Build Machine vs Run Machine

- **Every C++ project shipping via PyPI faces this: `manylinux` picks an old-enough glibc, `auditwheel` checks symbols, C++ itself isn't really covered by these tools, this is how Pytorch survives it:**
  - pip matches **platform tags** against your machine (`manylinux_2_34` beats `manylinux_2_28` when available); `libstdc++.so.6` is assumed present
  - and **`GLIBCXX`** symbol versioning does the rest
  - On macOS: the same idea, via **`MACOSX_DEPLOYMENT_TARGET`** tags

- **ROOT has all of that, plus things most C++ packages don't:**
  - **`cling`** compiles arbitrary user code against the system's own headers, at runtime
  - ROOT's **`clang C++ modules`** (.pcm *) serialize the AST from the build machines sysroot*
    → loading them against a different one at runtime is where things break

**PCMS - Precompiled Clang Modules, a cached, serialized snapshot of parsed C++ headers that lets `cling` skip re-parsing them every time*
**sysroot - the system headers and libraries a compiler builds against*

The fundamental problem of building on one platform, running on another, is not solved yet

- Open discussion, solutions we're considering:

Disallow system headers in jitted code	Simplest, but breaks some of ROOT's own tutorials
Bundle glibc/libstdc++ headers in the install tree	Reliable, but can clash with any other library doing the same
Ship one wheel per manylinux tag	Narrows the gap but doesn't fix the root of the problem
Use package variants for cross combinations between glibc and C++ versions	conda has package variants, pip has no equivalent concept
Rebuild PCMs on the user's machine on first run	**pip doesn't have post-install hooks**
Download prebuilt PCMs from a registry ROOT controls	Needs PCMs published for many glibc/libstdc++ combinations

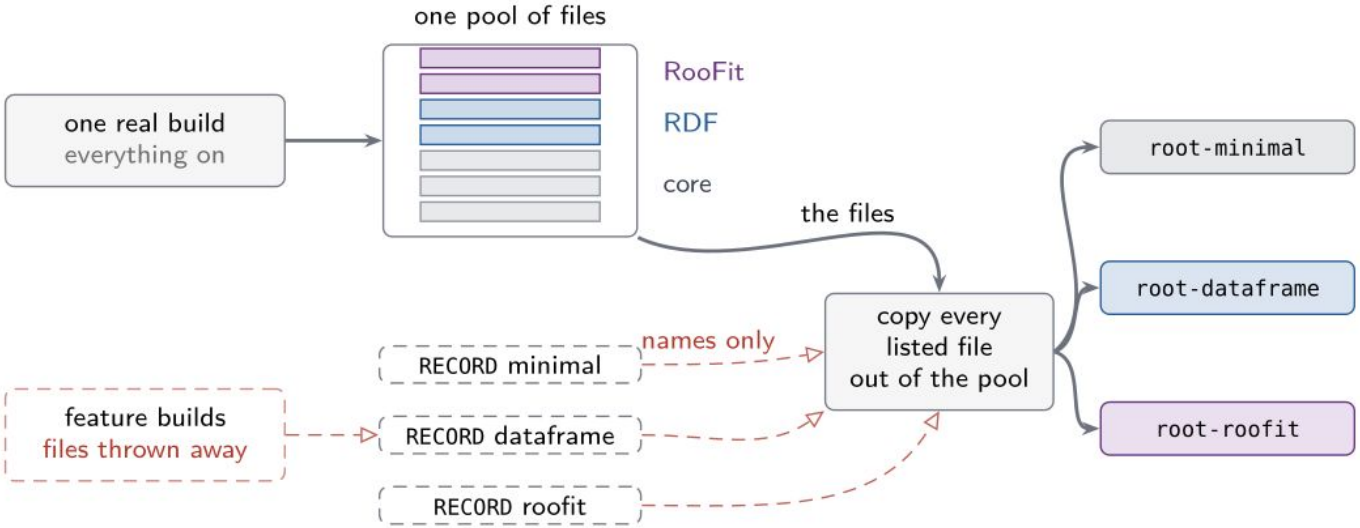
Outlook: Modular Wheels

- **ROOT ships as one wheel today**
  - What if you only need a fraction of the functionality exposed in your analyses?
- **We have recently investigated splitting ROOT into independent, composable wheels**
- **It's feasible!**

```
$ pip install root-minimal
$ pip install root-dataframe
$ pip install root-roofit
```

Outlook: The Trick behind Modular Wheels

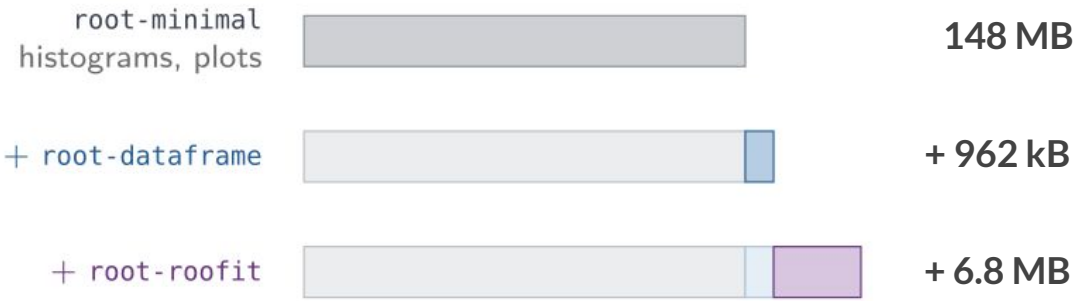
- In short: implicit namespaces let multiple wheels share one install tree;
  a post-install bootstrap script generates the shared modulemap at first init



Outlook: Takeaways From Modular Wheels

- **Needs a *writable* `site-packages` to work**
- **No real size win on the base install: `root-minimal` alone is already ~148 MB (~84% of the full wheel)**

  → `root-minimal` is built with the `-Dgminimal` build flag: includes the interpreter, root core, io, math, graphics
- **The implementation needs to be made robust first before considering discussing production**



Conclusion

- **Packaging ROOT has come a long way**

  → ROOT on conda-forge is a stable production distribution

  → ROOT on PyPI is evolving from an *alpha* stage to a *beta* release

  - Open discussions about the set of features the wheels should support
  - Modular ROOT wheels are feasible but not yet ready to discuss production

- **The hard problem is still open: building on one platform, running on another**

  → actively being worked on, not yet solved

ROOT is working to be a more natural fit in the Python packaging ecosystem building with standard tools
Let us know if you have ideas to make it better

Thank you!

https://root.cern