



Modern ChimeraX

Zach Pearson
RBVI General Meeting
26 May 2022

Background: ChimeraX



What is ChimeraX, really?

Four Pillars of the ChimeraX Architecture:

- An isolated, internally consistent distribution of Python...
- Including a namespace package for research computing distributed with that runtime...
- Distributed alongside useful 3rd party binaries...
- Launched by a small C binary that presents a friendly facade to end users



What is ChimeraX, really?

- An isolated, internally consistent distribution of Python...
- Including a namespace package for research computing distributed with that runtime...
- Distributed alongside useful 3rd party binaries...
- Launched by a small C binary that presents a friendly facade to end users

This is a fundamentally good architecture; this presentation does not argue for changing it.



What is ChimeraX, really?

- An isolated, internally consistent distribution of Python...
- Including a namespace package for research computing distributed with that runtime...
- Distributed alongside useful 3rd party binaries..
- Launched by a small C binary that presents a friendly facade to end users

This is a fundamentally good architecture; this presentation does not argue for changing it.

Today I want to focus on the namespace package



What is ChimeraX, really?

- An isolated, internally consistent distribution of Python...
- Including a namespace package for research computing distributed with that runtime...
- Distributed alongside useful 3rd party binaries..
- Launched by a small C binary that presents a friendly facade to end users

This is a fundamentally good architecture; this presentation does not argue for changing it.

Today I want to focus on the namespace package, and the C binary that calls it.



What is ChimeraX, really?

- An isolated, internally consistent distribution of Python...
- Including a namespace package for research computing distributed with that runtime...
- Distributed alongside useful 3rd party binaries..
- Launched by a small C binary that presents a friendly facade to end users

This is a fundamentally good architecture; this presentation does not argue for changing it.

Today I want to focus on the namespace package, and the C binary that calls it.

Background: 1.5 Objectives



1.5 Objectives

- ChimeraX in PyPi ([Ticket #1470](#)) – 3 years old
- Automate uploading wheels to the Toolshed ([Ticket #3058](#)) – 2 years old
- Move pip dependencies into bundles ([Ticket #3890](#)) – 19 months old
- Make native Apple M1 CPU distribution ([Ticket #4663](#)) - 1 year old
- Install PyPi packages directly within ChimeraX ([Ticket #4762](#)) – 11 months old
- Support pyproject.toml ([Ticket #6434](#)) – 7 weeks old

We are already moving in this direction, and we are already committed to *some* work. I am here to explain why in my opinion this vision fits best. Why *this* work is worthwhile in addition to, but also often lieu of, some of the work we would otherwise be committed to.

ChimeraX Binary



Currently

- We use ChimeraX to bootstrap itself: a small subset of ChimeraX python packages are used to build the larger ChimeraX
- **ChimeraX** the C binary depends somewhat less-than-obviously on **chimerax.core** the Python package
 - Core **must** be made on Windows before ChimeraX, to get the build info to make an icon
 - Core is initially made on macOS to generate a plist, which is then replaced later
 - ChimeraX the package's main module lives next to the binary's source code

ChimeraX Namespace Package



Currently

- Core and bundles are organized as separate packages, when they're really all subpackages
- When they reach their final destination, though, they are all under the ChimeraX namespace
- Building a bundle involves massaging a lot of ChimeraX metadata into a format acceptable for Python tooling

As a result, ChimeraX itself depends on ChimeraX.



Currently

- To build a bundle:
 - Define `bundle_info.xml`
 - Add metadata for `setup.py` and ChimeraX's custom metadata
 - Bundle builder will generate the arguments that'd normally go in `setup.py:setup` and then call `setup`, building a plain Python wheel
 - This could go on PyPi, but my one of my arguments is it would be somewhere between “quite” and “extremely” annoying for the library to depend on the built ChimeraX distribution and not vice versa
- ChimeraX will use metadata we place in `dist_info/METADATA` to set up bundles at runtime
- Some packages do not have the same installed folder name as their source folder names
 - Linters, test suites, IDEs, all of which operate on source directories and expect a 1:1 correspondence, don't know where our code is – this handicaps us.
- Language servers have no idea where our code is. If you have no prior knowledge of ChimeraX code IDEs cannot come to the rescue to help you discover it.



Currently

- To build a bundle:
 - Define bundle_info.xml
 - Add metadata for setup.py and ChimeraX's custom metadata
 - Bundle builder will generate a setup.py and execute it, building a plain Python wheel
 - This could go on PyPi, but my one of my arguments is it would be somewhere between “quite” and “extremely” annoying for the library to depend on the built ChimeraX distribution and not vice versa
- ChimeraX will use metadata we place in dist_info/METADATA to set up bundles at runtime
- Some packages do not have the same installed folder name as their source folder names
 - Linters, test suites, IDEs, all of which operate on source directories and expect a 1:1 correspondence, don't know where our code is – this handicaps us.
- Language servers have no idea where our code is. If you have no prior knowledge of ChimeraX code IDEs cannot come to the rescue to help you discover it.

The native-packaging branch



native-packaging from 10,000ft

- Resolve the dependency between the C binary and the namespace package
 - Each one can optionally be developed, distributed separately
- Preserve the core's ability to be built independently from the distribution build system
- Convert bundles to native tooling: pyproject.toml and naked setup.py
 - No more bundle_info.xml *by default* – it was converted to setup.py anyway
- Provide a metadata writer in **ChimeraX-Core** sufficient to build compliant wheels...
- ...then force bundles to include setuptools and at least (but not necessarily only*)
ChimeraX-Core as build dependencies in pyproject.toml

*If you need to link against some package's shared object then make it a build dependency



Immediate advantages of the branch

Two tickets become more or less the same ticket:

- ChimeraX in PyPi ([Ticket #1470](#)) reduces to uploading wheels (except UI) to PyPi
 - This can be automated with Python's own `cibuildwheel` package
 - `cibuildwheel` also sidesteps the M1 multi-arch compile problem
- Automate uploading wheels to the Toolshed ([Ticket #3058](#)) reduces to an “upload it here too” clause at the end of 1470's solution

Two tickets are naturally completed through the routine work of native packaging:

- Move pip dependencies into bundles ([Ticket #3890](#))
- Support pyproject.toml ([Ticket #6434](#))



Immediate advantages, cont'd

One ticket nearly solves itself:

- Make native Apple M1 CPU distribution ([Ticket #4663](#))
 - Wheels get their tags programmatically. We can currently multi-arch compile, and still get a specific tag: arm64/x64, because we assign tags ourselves
 - To build a universal2 wheel requires a universal2 version of Python
 - We can make a virtual environment from the Python prereq, then Python does the heavy lifting for us



What this solution enables

Decoupling ChimeraX the C binary from chimeraX the namespace package should let us use native tooling to build every part of ChimeraX, including core.

But let's say we don't even decouple the version info; it's still useful to add it to the launcher even if we just duplicate it.

If we put it in the launcher and the core, then whenever we try to read the version from the launcher and can't, and have to use the core version as plan B, we know we're in the PyPi library and not the distribution.

We can know when the core build date differs from the launcher build date, and adjust accordingly.

Which means more useful bug reports. It's not going to be tremendously often that it makes the difference, but it's useful additional information.

***build.py** in **poetry**'s case



What this solution enables

Decoupling ChimeraX the C binary from `chimerax` the namespace package should let us use native tooling to build every part of ChimeraX, including core.

We will eventually *have* to change the bundle builder to support `setuptools`'s deprecation of direct invocations of `setup.py`. We once thought that would be `setup.cfg` for pure python packages; however...

We **must** support `pyproject.toml`, as `setuptools` will eventually deprecate `setup.cfg` too

ChimeraX bundles will always require a `setup.py*`, but we can store package metadata, ChimeraX metadata, and most importantly *read* ChimeraX's metadata from `pyproject.toml` files

So why don't we just use the native config files instead of `bundle_info.xml`?

We can read the bundle name for the toolshed, for example, from `[tool.chimerax.bundle-name]`

Classifiers come from `[tool.chimerax.classifiers]`

*`build.py` in `poetry`'s case



What this solution enables

One more thing:

(Almost) NO MORE MANUAL BUNDLE BUILD DEPENDENCY TRACKING

Right now we have to build bundles in a specific order.

Once each bundle is built once by hand and uploaded, each bundle should be able to depend on any other bundle and the build order should not matter anymore.

The build system will download those bundles at build-time.

We will not need to algorithmically track bundle dependencies anymore.

Fast failures: Versioning mistakes will be caught early in builds.

A New Distribution Scheme



Supporting ChimeraX-Y.Z and PyPi in One Codebase

- With a native-first package, we could have a point release that specifies what versions of our packages are *stable* and *supported for that release*.

Possible commands:

- `make build-pypi`
 - Recursively build individual bundles and upload them to PyPi (except ui)
- `make build-distribution`
 - Could be exactly what we do now

Alternatively:

- Build the ChimeraX environment, then download ChimeraX packages from the toolshed

On the toolshed:

- Host empty package, e.g. `chimerax-1.4-dist` on toolshed
- This package will depend on every bundle that comprises a stable point release



Testing to the Best of Our Abilities

- This way of doing things would enable us to audit ChimeraX via automated testing
 - Each package is buildable, installable, and testable without the ChimeraX environment, as any library package should be, so...
- We could gatekeep uploads to the toolshed behind passing tests, as we already should for PyPi
- ChimeraX-daily *could* become whatever complete suite of mutually compatible bundles managed to pass testing
- Won't be complete at first, but over time will dramatically cut down or possibly eliminate broken daily builds

Alternatively, we could just do what we do now: ship code.



ChimeraX on the Toolshed

- Toolshed is a PyPi repository
- ...and *already* in need of an update, regardless of the adoption of these changes

If we're going to put bundles on the toolshed anyway...

- Offer users two options for updating ChimeraX in the GUI
 - Only install stable updates (same minor version)
 - Install any mutually compatible updates
- Allow users to check for updates to specific packages

The distribution cannot have the same version as the core bundle for this to happen, because the core bundle should be able to be upgraded.

For users who elect to take the latest stable bundles, it solves the “oh crap we shipped a bug” problem.

All bundles compatible with the same Python point release as the original distribution release should be updatable within ChimeraX.



But let's not get carried away...

I'd like to reiterate:

We do **not** need a new build system, new repo, nor a new fundamental paradigm for reasoning about ChimeraX.

Besides replacing `bundle_info.xml`, most of the changes in **native-packaging** are paths: the location of `$(TOP)`, where the core Makefile is, and corrections to the gitignore, for example.

This proposed changeset just makes it easy to reason about ChimeraX the distribution and ChimeraX the package at the same time.

The Timing Is Right

Our Tech Debt is NOT Intractable

2-Pass Plan for PyPi Packaging



2-Pass Plan

Pass 1:

- Build all the bundles without any dependencies specified

- Move dependencies into bundles

Pass 2:

- Mark features as optional in bundles, upload complete ones to PyPi one-by-one

- Gradually use build dependencies from PyPi to build bundles

In Summary



Summary

- Native packaging tools can support ChimeraX distribution and ChimeraX library simultaneously with minimal overhead
- Modern packaging is a prerequisite to making a PyPi package
- Decoupling the binary distribution from the core enables many modern features impossible under our current scheme, both for developers and users
- Modern package architectures make it easier for people who know Python but *not ChimeraX* to apply their knowledge to ChimeraX: we can expect more bundle developers in the future.

Best of all:

- This future is much closer than previously anticipated.
- We compile 24 bundles. **14** of them are compiling without bundle builder *today*.



Thank you!

Supplemental Slides



What native-packaging does

- Move `ChimeraX_main.py` → `core/__main__.py`
- Point the ChimeraX binary at `chimerax.core`
- Read `argv` from `sys.argv` as the first call in `__main__:init` vs passing in `argv`
- Define `CX_DIST_VER` as a variable that can be used by `make/C` preprocessor
- On macOS: delay installing the plist until the very end, do it once
- On Windows: Get icon info from the build system!



What native-packaging does, cont'd

- Use the ChimeraX binary to modify the ChimeraX module at runtime using the Python C API
 - Cleanly inject the buildinfo variables when it's run from the distribution

```
Py_Initialize(); // blank Python environment
// "chimerax" module is empty because it contains no code
PyObject* py_cx = PyImport_ImportModule("chimerax");
PyModule_AddStringConstant(py_cx, "_CHIMERAX_C_DIST_BUILD_DATE", __DATE__)
```

Could also use `PyImport_AddModule` so as not to touch the top-level ChimeraX module

Reading from the injected variables instead of buildinfo lets us tell whether a user has the dist or the library for a bug report.



What native-packaging does, cont'd

- How do we know what kind of build we're running with variable injection?

Luckily we already have a way to do this: **make**! We can even use our automated system!

```
$ make -D$(BUILD_TYPE) apps/ChimeraX
```

```
//launcher.c
#ifdef daily
PyModule_AddStringConstant(py_cx, "_CHIMERAX_C_DIST_VERSION", "1.5-daily")
#else
PyModule_AddStringConstant(py_cx, "_CHIMERAX_C_DIST_VERSION", "1.5")
#endif
int result = PyMain(new_argc, new_argv);
```



What native-packaging does, cont'd

- Move src/apps to /apps – they are not part of the namespace package
- Move src/examples to /examples - Not sure where else it should go
- Move **all** bundles to src/
- Merge src/Makefile and src/bundles/Makefile
 - Both get **much** simpler with native packaging, but with some coercion worked on the old system too
- Move metadata from bundle_info.xml to pyproject.toml: `[tools.chimerax]`
- Make classifiers a dynamic category in pyproject.toml



What native-packaging does, cont'd

- Move src/apps to /apps – they are not part of the namespace package
- Move src/examples to /examples - Not sure where else it should go
- Move **all** bundles to src/
- Merge src/Makefile and src/bundles/Makefile
 - Both get **much** simpler with native packaging, but with some coercion worked on the old system too
- Move metadata from bundle_info.xml to pyproject.toml: `[tools.chimerax]`
- Make classifiers a dynamic category in pyproject.toml



What native-packaging does, cont'd

- Add a toml-to-classifier converter to `chimerax.core` that can be used standalone without *any* other bundle
- Use that module to write `classifiers.txt` which `setup.py` reads
- Other bundles should include “`chimerax.core >= 1.5.0`” in `[build-system.requires]`
- `setup.py` should `from chimerax.core.write_bundle_classifiers import write_bundle_classifiers`, then call it

Wheels are then built which have *all* of the correct ChimeraX metadata.

I'm not saying it's the best solution we can find, it's just my proof-of-concept.



Tricking 'python -m build' into using local packages


- First, make a virtual environment somewhere and activate it
- Install **pypiserver**
- Make a directory somewhere called “**packages**”
- “**python -m build --wheel**” in the core source directory
- Copy the built wheel into the packages directory
- Run pypiserver and point it at the packages directory
- Put this in `~/.config/pip/pip.conf`:

```
[global]
```


```
extra-package-index=http://localhost:8080/simple/
```

Can't wait to retire this hack – which the maintainers say is a bug – when **core** is on PyPi.

Other Miscellaneous Changes

- 
- mk/os.make sets \$(CC) and \$(CXX) on macOS to clang **PLUS** what should be \$(CFLAGS) and \$(CXXFLAGS) but macOS already knows it should use clang as it's Apple's preferred compiler
 - This caused issues compiling md_crds/ because a dependency thinks all the \$(CC) or \$(CXX) definition is the compiler and not just the first space-separated field of the definition
 - To fix: remove \$(CC) and \$(CXX) definitions from macOS, make them \$(CFLAGS) and \$(CXXFLAGS)
 - We do not need two sync directories
 - Uploaded slightly modified segger tarball with extension only recognized by my branch to change its internal \$(TOP) value
 - Adjust spacenavigator makefile to use the \$(TOP) variable, and the new path to copyright.txt
 - **chimerax.core.main**: It's now two more folders deep so needs two more iterations of the **dn ()** basedir finding hack

Other Miscellaneous Suggestions

- 
- `rg "ioutil"` reveals that `ioutil` C++ code is only referenced in two places:
 - *Defined* in `core`
 - *Included* in `atomic_lib`


Source code that *can* live where it's used needs to live where it's used.

- `rg "pysupport"` reveals that it's used in many places
- `rg "chutil"` reveals that it's used in many places
- `rg "include.*logger"` reveals that it's used in many places

If source code must be abstracted, let's place it in a common top-level "include" directory, or in `src/_lib`, like other Python packages; it's just confusing and not very discoverable that it's polluting the core directory.

Native tooling can then be pointed at that include directory to include code in source distributions, which will then be turned into compiled C/++ extensions.

Note: In light of "numpy.get_include()", this slide is woefully wrong; see next slide

- 
- Make folders containing header files into subpackages of the containing packages
 - Add an `__init__.py` with the following contents:

```
import os

def get_include() -> str:

    return os.path.dirname(__file__)
```



What if a user or developer borks their environment?

- `{toolshed,pip} reinstall chimeraX-1.4-py39-none-any.whl`
 - Clear the internal site-packages directory
 - Pull the ChimeraX distribution wheel from the toolshed
 - Reinstall it



Programmmator Cave

- I don't pretend to have all the answers to all the unknown unknowns that this implies...
- ...but I have tried my hardest to anticipate, discover, and provisionally solve as many of them as possible
- Nothing said today is set in stone without the consent of the team
- I used `setuptools` for this talk but `poetry` conversions are easy!
 - (that's the point of `pyproject.toml`!)
 - Poetry lacks dynamic metadata which takes it out of the running *for now*
- While I mentioned some hypothetical possibilities, we should evaluate the talk by how much it reduces already-planned work – the status quo is suboptimal but not *exactly* “broken” – before we get excited about the possibilities