PyOxidizer

Release 0.4.0

Nov 24, 2019

Contents

`PyOxidizer` is a utility that aims to solve the problem of how to distribute Python applications. See *Overview* for more or dive into *Getting Started* to learn how to start using `PyOxidizer`.

The official home of the `PyOxidizer` project is https://github.com/indygreg/PyOxidizer. Official documentation lives at https://pyoxidizer.readthedocs.io/en/latest/index.html.

The pyoxidizer-users mailing list is a forum for users to discuss all things PyOxidizer.

If you want to financially contribute to PyOxidizer, do so on Patreon or via PayPal.

The creator and maintainer of `PyOxidizer` is Gregory Szorc.

CHAPTER 1

Overview

From a very high level, `PyOxidizer` is a tool for packaging and distributing Python applications. The over-arching goal of `PyOxidizer` is to make this (often complex) problem space simple so application maintainers can focus on building quality applications instead of toiling with build systems and packaging tools.

On a lower, more technical level, `PyOxidizer` has a command line tool - `pyoxidizer` - that is capable of building binaries (executables or libraries) that embed a fully-functional Python interpreter plus Python extensions and modules *in a single binary*. Binaries produced with `PyOxidizer` are highly portable and can work on nearly every system without any special requirements like containers, FUSE filesystems, or even temporary directory access. On Linux, `PyOxidizer` can produce executables that are fully statically linked and don't even support dynamic loading.

The *Oxidizer* part of the name comes from Rust: binaries built with `PyOxidizer` are compiled from Rust and Rust code is responsible for managing the embedded Python interpreter and all its operations. But the existence of Rust should be invisible to many users, much like the fact that CPython (the official Python distribution available from www.python.org) is implemented in C. Rust is simply a tool to achieve an end goal (albeit a rather effective and powerful tool).

1.1 Benefits of PyOxidizer

You may be wondering why you should use or care about `PyOxidizer`. Great question!

Python application distribution is generally considered an unsolved problem. At PyCon 2019, Russel Keith-Magee identified code distribution as a potential *black swan* for Python during a keynote talk. In their words, *Python hasn't ever had a consistent story for how I give my code to someone else, especially if that someone else isn't a developer and just wants to use my application.* The over-arching goal of `PyOxidizer` is to solve this problem. If we're successful, we help Python become a more attractive option in more domains and eliminate this potential *black swan* that is an existential threat for Python's longevity.

On a less existential level, there are several benefits to `PyOxidizer`.

1.1.1 Ease of Application Installation

Installing Python applications can be hard, especially if you aren't a developer.

Applications produced with `PyOxidizer` are self-contained - as small as a single file executable. From the perspective of the end-user, they get an executable containing an application that *just works*. There's no need to install a Python distribution on their system. There's no need to muck with installing Python packages. There's no need to configure a container runtime like Docker. There's just an executable containing an embedded Python interpreter and associated Python application code and running that executable *just works*. From the perspective of the end-user, your application is just another platform native executable.

1.1.2 Ease of Packaging and Distribution

Python application developers can spend a large amount of time managing how their applications are packaged and distributed. There's no universal standard for distributing Python applications. Instead, there's a hodgepodge of random tools, typically different tools per operating system.

Python application developers typically need to *solve* the packaging and distribution problem N times. This is thankless work and sucks valuable time away from what could otherwise be spent improving the application itself. Furthermore, each distinct Python application tends to solve this problem redundantly.

Again, the over-arching goal of `PyOxidizer` is to provide a comprehensive solution to the Python application packaging and distribution problem space. We want to make it as turn-key as possible for application maintainers to make their applications usable by novice computer users. If we're successful, Python developers can spend less time solving packaging and distribution problems and more time improving Python applications themselves. That's good for the Python ecosystem.

1.1.3 Faster Python Programs

Binaries built with `PyOxidizer` tend to run faster than those executing via a normal `python` interpreter. There are a few reasons for this.

In its default configuration, binaries produced with `PyOxidizer` configure the embedded Python interpreter differently from how a `python` is typically configured.

Notably, `PyOxidizer` disables the importing of the `site` module by default (making it roughly equivalent to `python -S`). The `site` module does a number of things, such as look for `.pth` files, looks for `site-packages` directories, etc. These activities can contribute substantial overhead, as measured through a normal `python3.7` executable on macOS:

```
$ hyperfine -m 500 -- '/usr/local/bin/python3.7 -c 1' '/usr/local/bin/python3.7 -S -c␣
↪1'
Benchmark #1: /usr/local/bin/python3.7 -c 1
  Time (mean ± σ):      22.7 ms ±   2.0 ms    [User: 16.7 ms, System: 4.2 ms]
  Range (min ... max):    18.4 ms ...  32.7 ms    500 runs

Benchmark #2: /usr/local/bin/python3.7 -S -c 1
  Time (mean ± σ):      12.7 ms ±   1.1 ms    [User: 8.2 ms, System: 2.9 ms]
  Range (min ... max):     9.8 ms ...  16.9 ms    500 runs

Summary
  '/usr/local/bin/python3.7 -S -c 1' ran
    1.78 ± 0.22 times faster than '/usr/local/bin/python3.7 -c 1'
```

Shaving ~10ms off of startup overhead is not trivial!

Another performance benefit comes from importing modules from memory. `PyOxidizer` supports importing Python modules from memory using zero-copy. Traditionally, Python performs filesystem I/O to find and load modules. Filesystem I/O performance is intrinsically inconsistent and depends on several factors. Importing modules from memory removes the filesystem API overhead and the only inconsistency is whether the binary's memory address range containing Python modules is paged in. (On first binary load the first access of a memory address will require the underling file to be paged in by the kernel. But this all happens in the kernel and avoids filesystem API overhead from userland.)

We can attempt to isolate the effect of in-memory module imports by running a Python script that attempts to import the entirety of the Python standard library. This test is a bit contrived. But it is effective at demonstrating the performance difference.

Using a stock `python3.7` executable and 2 `PyOxidizer` executables - one configured to load the standard library from the filesystem and another from memory:

```
$ hyperfine -m 50 -- '/usr/local/bin/python3.7 -S import_stdlib.py' import-stdlib-
↪filesystem import-stdlib-memory
Benchmark #1: /usr/local/bin/python3.7 -S import_stdlib.py
  Time (mean ± σ):     258.8 ms ±   8.9 ms    [User: 220.2 ms, System: 34.4 ms]
  Range (min … max):   247.7 ms … 310.5 ms    50 runs

Benchmark #2: import-stdlib-filesystem
  Time (mean ± σ):     249.4 ms ±   3.7 ms    [User: 216.3 ms, System: 29.8 ms]
  Range (min … max):   243.5 ms … 258.5 ms    50 runs

Benchmark #3: import-stdlib-memory
  Time (mean ± σ):     217.6 ms ±   6.4 ms    [User: 200.4 ms, System: 13.7 ms]
  Range (min … max):   207.9 ms … 243.1 ms    50 runs

Summary
  'import-stdlib-memory' ran
    1.15 ± 0.04 times faster than 'import-stdlib-filesystem'
    1.19 ± 0.05 times faster than '/usr/local/bin/python3.7 -S import_stdlib.py'
```

We see that the `PyOxidizer` executable importing from the filesystem has very similar performance to `python3.7`. But the `PyOxidizer` executable importing from memory is clearly faster. These measurements were obtained on macOS and the `import_stdlib.py` script imports 506 modules.

1.2 Components

The most visible component of `PyOxidizer` is the `pyoxidizer` command line tool. This tool contains functionality for creating new projects using `PyOxidizer`, adding `PyOxidizer` to existing projects, producing binaries containing a Python interpreter, and various related functionality.

The `pyoxidizer` executable is written in Rust. Behind that tool is a pile of Rust code performing all the functionality exposed by the tool. That code is conveniently also made available as a library, so anyone wanting to integrate `PyOxidizer`'s core functionality without using our `pyoxidizer` tool is able to do so.

The `pyoxidizer` crate and command line tool are effectively glorified build tools: they simply help with various project management, build, and packaging.

The run-time component of `PyOxidizer` is completely separate from the build-time component. The run-time component of `PyOxidizer` consists of a Rust crate named `pyembed`. The role of the `pyembed` crate is to manage an embedded Python interpreter. This crate contains all the code needed to interact with the CPython APIs to create and run a Python interpreter. `pyembed` also contains the special functionality required to import Python modules from memory using zero-copy.

1.3 How It Works

The `pyoxidizer` tool is used to create a new project or add `PyOxidizer` to an existing (Rust) project. This entails:

- Adding a copy of the `pyembed` crate to the project.
- Generating a boilerplate Rust source file to call into the `pyembed` crate to run a Python interpreter.
- Generating a working `pyoxidizer.toml` *configuration file*.
- Telling the project's Rust build system about `PyOxidizer`.

When that project's `pyembed` crate is built by Rust's build system, it calls out to `PyOxidizer` to process the active `PyOxidizer` configuration file. `PyOxidizer` will obtain a specially-built Python distribution that is optimized for embedding. It will then use this distribution to finish packaging itself and any other Python dependencies indicated in the configuration file. For example, you can process a pip requirements file at build time to include additional Python packages in the produced binary.

At the end of this sausage grinder, `PyOxidizer` emits an archive library containing Python (which can be linked into another library or executable) and *resource files* containing Python data (such as Python module sources and bytecode). Most importantly, `PyOxidizer` tells Rust's build system how to integrate these components into the binary it is building.

From here, Rust's build system combines the standard Rust bits with the files produced by `PyOxidizer` and turns everything into a binary, typically an executable.

At run time, an instance of the `PythonConfig` struct from the `pyembed` crate is created to define how an embedded Python interpreter should behave. (One of the build-time actions performed by `PyOxidizer` is to convert the TOML configuration file into a default instance of this struct.) This struct is used to instantiate a Python interpreter.

The `pyembed` crate implements a Python *extension module* which provides custom module importing functionality. Light magic is used to coerce the Python interpreter to load this module very early during initialization. This allows the module to service Python `import` requests. The custom module importer installed by `pyembed` supports retrieving data from a read-only data structure embedded in the executable itself. Essentially, the Python `import` request calls into some Rust code provided by `pyembed` and Rust returns a `void *` to memory containing data (module source code, bytecode, etc) that was generated at build time by `PyOxidizer` and later embedded into the binary by Rust's build system.

Once the embedded Python interpreter is initialized, the application works just like any other Python application! The main differences are that modules are (probably) getting imported from memory and that Rust - not the Python distribution's `python` executable logic - is driving execution of Python.

Read on to *Getting Started* to learn how to use `PyOxidizer`.

CHAPTER 2

Getting Started

2.1 Installing

2.1.1 Installing Rust

PyOxidizer is a Rust application and requires Rust 2018 (1.31 or newer) to be installed in order to build PyOxidizer itself as well as Python application binaries.

You can verify your installed version of Rust by running:

```
$ rustc --version
rustc 1.35.0 (3c235d560 2019-05-20)
```

If you don't have Rust installed, https://www.rust-lang.org/ has very detailed instructions on how to install it.

Rust releases a new version every 6 weeks and language development moves faster than other programming languages. It is common for the Rust packages provided by common package managers to lag behind the latest Rust release by several releases. For that reason, use of the `rustup` tool for managing Rust is highly recommended.

If you are a security paranoid individual and don't want to follow the official `rustup` install instructions involving a `curl | sh` (your paranoia is understood), you can find instructions for alternative installation methods at https://github.com/rust-lang/rustup.rs/#other-installation-methods.

2.1.2 Other System Dependencies

You will need a working C compiler/toolchain in order to build some Rust crates and their dependencies. If Rust cannot find a C compiler, it should print a message at build time and give you instructions on how to install one.

There is a known issue with PyOxidizer on Fedora 30+ that will require you to install the `libxcrypt-compat` package to avoid an error due to a missing `libcrypt.so.1` file. See https://github.com/indygreg/PyOxidizer/issues/89 for more info.

2.1.3 Installing PyOxidizer

PyOxidizer can be installed from its latest published crate:

```
$ cargo install pyoxidizer
```

From a Git repository using cargo:

```
# The latest commit in source control.
$ cargo install --git https://github.com/indygreg/PyOxidizer.git --branch main␣
↪pyoxidizer

$ A specific release
$ cargo install --git https://github.com/indygreg/PyOxidizer.git --tag <TAG>␣
↪pyoxidizer
```

Or by cloning the Git repository and building the project locally:

```
$ git clone https://github.com/indygreg/PyOxidizer.git
$ cd PyOxidizer
$ cargo install --path pyoxidizer
```

Note: PyOxidizer's project policy is for the `main` branch to be stable. So it should always be relatively safe to use `main` instead of a released version.

Once the `pyoxidizer` executable is installed, try to run it:

```
$ pyoxidizer
PyOxidizer 0.1
Gregory Szorc <gregory.szorc@gmail.com>
Build and distribute Python applications

USAGE:
    pyoxidizer [SUBCOMMAND]

...
```

Congratulations, PyOxidizer is installed! Now let's move on to using it.

2.2 Your First PyOxidizer Project

The `pyoxidizer init` command will create a new [Rust] project which supports embedding Python. Invoke it with the directory you want to create your new project in:

```
$ pyoxidizer init pyapp
```

This should have printed out details on what happened and what to do next. If you actually ran this in a terminal, hopefully you don't need to continue following the directions here as the printed instructions are sufficient! But if you aren't, keep reading.

The default project created by `pyoxidizer init` will produce an executable that embeds Python and starts a Python REPL by default. Let's test that:

```
$ cd pyapp
$ pyoxidizer run
no existing PyOxidizer artifacts found
processing config file /home/gps/src/pyapp/pyoxidizer.toml
resolving Python distribution...
...
   Compiling pyapp v0.1.0 (/home/gps/src/pyapp)
    Finished dev [unoptimized + debuginfo] target(s) in 53.14s
     Running `target/debug/testapp`
>>>
```

If all goes according to plan, you just started a Rust executable which started a Python interpreter, which started an interactive Python debugger! Try typing in some Python code:

```
>>> print("hello, world")
hello, world
```

It works!

(To exit the REPL, press CTRL+d or CTRL+z.)

Continue reading *Managing Projects with pyoxidizer* to learn more about the `pyoxidizer` tool. Or read on for a preview of how to customize your application's behavior.

2.3 Customizing Python and Packaging Behavior

Embedding Python in a Rust executable and starting a REPL is cool and all. But you probably want to do something more exciting.

Inside the project's root directory is an autogenerated `pyoxidizer.toml` file. This file configures how the embedded Python interpreter is built as well as defines default run-time behavior. See *New Project Layout* if you are interested in what all the files do.

Open `pyoxidizer.toml` in your favorite editor and find the `[[python_run]]` section. This section configures what to do when the interpreter starts. By default, it should have a `mode = "repl"` line. Let's comment that out or delete it and replace it with the following:

```
[[embedded_python_run]]
mode = "eval"
code = "import uuid; print(uuid.uuid4())"
```

We're now telling the interpreter to effectively run the Python statement `eval(import uuid; print(uuid.uuid4())` when it starts. Test that out:

```
$ pyoxidizer run
   Compiling pyembed v0.1.0 (/home/gps/src/pyapp/pyembed)
   Compiling pyapp v0.1.0 (/home/gps/src/pyapp)
    Finished dev [unoptimized + debuginfo] target(s) in 3.92s
     Running `target/debug/pyapp`
96f776c8-c32d-48d8-8c1c-aef8a735f535
```

It works!

This is still pretty trivial. But it demonstrates how the `pyoxidizer.toml` is used to influence the behavior of built binaries.

Let's do something a little bit more complicated, like package an existing Python application!

Find the existing `[[python_packages]]` section in the `pyoxidizer.toml`. Now let's add the following lines after the last of those sections:

```
[[packaging_rule]]
type = "pip-install-simple"
package = "pyflakes==2.1.1"
```

And change the `[[embedded_python_run]]` section to:

```
[[embedded_python_run]]
mode = "eval"
code = "from pyflakes.api import main; main()"
```

This tells PyOxidizer that you want to install version 2.1.1 of the `pyflakes` package. At build time, this will effectively perform a `pip install pyflakes==2.1.1` and take all installed files and add them to the produced binary. Let's try that:

```
$ pyoxidizer run -- --help
   Compiling pyembed v0.1.0 (/home/gps/tmp/pyapp/pyembed)
   Compiling pyapp v0.1.0 (/home/gps/tmp/pyapp)
    Finished dev [unoptimized + debuginfo] target(s) in 5.49s
     Running `target/debug/pyapp --help`
Usage: pyapp [options]

Options:
  --version   show program's version number and exit
  -h, --help  show this help message and exit
```

You've just produced an executable for pyflakes!

There are far more powerful packaging and configuration settings available. Read all about them at *Configuration Files* and *Packaging User Guide*. Or continue on to *Managing Projects with pyoxidizer* to learn more about the `pyoxidizer` tool.

CHAPTER 3

Managing Projects with `pyoxidizer`

The `pyoxidizer` command line tool is used to manage the integration of `PyOxidizer` within a Rust project. See *Components* for more on the various components of `PyOxidizer`.

3.1 High-Level Project Lifecycle and Pipeline

`PyOxidizer` projects conceptually progress through a development pipeline. This pipeline consists of the following phases:

1. Creation
2. Python Building
3. Application Building
4. Application Assembly
5. Validation (manual)
6. Distribution (not yet implemented)

In `Creation`, a new project is created.

In `Python Building`, the Python components of the project are derived. This includes fetching any Python package dependencies.

In `Application Building`, the larger [Rust] application is built. this usually entails producing an executable containing an embedded Python interpreter along with any embedded python resource data.

In `Application Assembly`, the built [Rust] application is assembled with other packaging pieces. These extra pieces could include Python modules not embedded in the [Rust] binary.

In `Validation`, the assembled application is validated, tested, etc.

In `Distribution`, distributable versions of the assembled application are produced. This includes installable packages, etc.

Typically, `Python Building`, `Application Building`, and `Application Assembly` are performed as a single logical step (often via `pyoxidizer build`). But `PyOxidizer` supports performing each action in isolation in order to facilitate more flexible development patterns.

3.2 Creating New Projects with `init`

The `pyoxidizer init` command will create a new [Rust] project which supports embedding Python. Invoke it with the directory you want to create your new project in:

```
$ pyoxidizer init pyapp
```

This should have printed out details on what happened and what to do next. If you actually ran this in a terminal, hopefully you don't need to continue following the directions here as the printed instructions are sufficient!

Before we move on, let's explore what new projects look like.

3.3 New Project Layout

`pyoxidizer init` essentially does two things:

1. Creates a new Rust executable project by running `cargo init`.
2. Adds PyOxidizer files to that project.

If we run `pyoxidizer init pyapp`, let's explore our newly-created `pyapp` project:

```
$ find pyapp -type f | grep -v .git
pyapp/Cargo.toml
pyapp/src/main.rs
pyapp/pyoxidizer.toml
pyapp/pyembed/src/config.rs
pyapp/pyembed/src/importer.rs
pyapp/pyembed/src/data.rs
pyapp/pyembed/src/lib.rs
pyapp/pyembed/src/pyinterp.rs
pyapp/pyembed/src/pyalloc.rs
pyapp/pyembed/src/pystr.rs
pyapp/pyembed/build.rs
pyapp/pyembed/Cargo.toml
```

3.3.1 The Main Project

The `Cargo.toml` file is the configuration file for the Rust project. Read more in the official Cargo documentation. The magic lines in this file to enable PyOxidizer are the following:

```
[dependencies]
pyembed = { path = "pyembed" }
```

These lines declare a dependency on the `pyembed` package in the directory `pyembed`. `Cargo.toml` is overall pretty straightforward.

Next let's look at `pyapp/src/main.rs`. If you aren't familiar with Rust projects, the `src/main.rs` file is the default location for the source file implementing an executable. If we open that file, we see a `fn main() {` line, which declares the *main* function for our executable. The file is relatively straightforward. We import some symbols from the `pyembed` crate. We then construct a config object, use that to construct a Python interpreter, then we run the interpreter and pass its exit code to `exit()`.

3.3.2 The `pyembed` Package

The bulk of the files in our new project are in the `pyembed` directory. This directory defines a Rust project whose job it is to build and manage an embedded Python interpreter. This project behaves like any other Rust library project: there's a `Cargo.toml`, a `src/lib.rs` defining the main library define, and a pile of other `.rs` files implementing the library functionality. The only functionality you will likely be concerned about are the `PythonConfig` and `MainPythonInterpreter` structs. These types define how the embedded Python interpreter is configured and executed. If you want to learn more about this crate and how it works, run `cargo doc`.

There are a few special properties about the `pyembed` package worth calling out.

First, the package is a copy of files from the PyOxidizer project. Typically, one could reference a crate published on a package repository like https://crates.io/ and we wouldn't need to have local files. However, `pyembed` is currently relying on modifications to some other published crates (we plan to upstream all changes eventually). This means we can't publish `pyembed` on crates.io. So we need to vendor a copy next to your project. Sorry about the inconvenience!

Speaking of modification to the published crates, the `pyembed`'s `Cargo.toml` enumerates those crates. If `pyoxidizer` was run from an installed executable, these modified crates will be obtained from PyOxidizer's canonical Git repository. If `pyoxidizer` was run out of the PyOxidizer source repository, these modified crates will be obtained from the local filesystem path to that repository. **You may want to consider making copies of these crates and/or vendoring them next to your project if you aren't comfortable fetching dependencies from the local filesystem or a Git repository.**

Another property about `pyembed` worth mentioning is its `build.rs` build script. This program runs as part of building the library. As you can see from the source, this program attempts to locate a `pyoxidizer` executable and then calls `pyoxidizer run-build-script`. `pyoxidizer` thus provides the bulk of the build script functionality. This is slightly unorthodox. But it enables you to build applications without building all of PyOxidizer. And since PyOxidizer has a few hundred package dependencies, this saves quite a bit of time!

3.3.3 The `pyoxidizer.toml` Configuration File

The final file in our newly created project is `pyoxidizer.toml`. **It is the most important file in the project.**

The `pyoxidizer.toml` file configures how the embedded Python interpreter is built. This includes choosing which modules to package. It also configures the default run-time settings for the interpreter, including which code to run.

See *Configuration Files* for comprehensive documentation of `pyoxidizer.toml` files and their semantics.

3.4 Adding PyOxidizer to an Existing Project with `add`

Do you have an existing Rust project that you want to add an embedded Python interpreter to? PyOxidizer can help with that too! The `pyoxidizer add` command can be used to add an embedded Python interpreter to an existing Rust project. Simply give the directory to a project containing a `Cargo.toml` file:

```
$ cargo init myrustapp
  Created binary (application) package
$ pyoxidizer add myrustapp
```

This will add required files and make required modifications to add an embedded Python interpreter to the target project. Most of the modifications are in the form of a new `pyembed` crate.

Important: It is highly recommended to have the destination project under version control so you can see what changes are made by `pyoxidizer add` and so you can undo any unwanted changes.

Danger: This command isn't very well tested. And results have been known to be wrong. If it doesn't *just work*, you may want to run `pyoxidizer init` and incorporate relevant files into your project manually. Sorry for the inconvenience.

3.5 Building PyObject Projects with `build`

The `pyoxidizer build` command is probably the most important and used `pyoxidizer` command. This command does the following:

1. Processes the `pyoxidizer.toml` configuration file and derives Python artifacts to incorporate in a larger binary. (The `Python Building` phase of the pipeline described at the top of this document.)
2. Invokes `cargo build` to build the associated Rust project. (The `Application Building` phase.)
3. Performs any post-build actions to assemble extra resources alongside the `cargo`-built binary. (The `Application Assembly` phase.)

In short, `pyoxidizer build` attempts to build your application as you have configured it.

`Application Assembly` is performed into a `build/apps/<app>` directory under the project root. If your project name is `myapp`, the application will be assembled to a `build/apps/myapp` directory. The full path to the executable will be `build/apps/myapp/<target>/<build_type>/myapp` (on Linux and macOS) or `build/apps/myapp/<target>/<build_type>/myapp.exe` (on Windows).

It's worth noting that the ergonomics of `pyoxidizer build` are superior to `cargo build`. With `pyoxidizer build`, the tool prints information about Python-specific activity as it is occurring. While it is possible to build applications with `cargo build` to achieve the same effect, doing so will defer Python build steps until later in the build and will hide that activity from output. This behavior isn't optimal for people whose primary goal is to package Python applications.

3.6 Running Applications with `run`

Once you have produced an application with `pyoxidizer build`, you can run it with `pyoxidizer run`. For example:

```
$ pyoxidizer run -- foo bar'
```

This command will build your application (if needed) then invoke it with the arguments specified.

This command is provided for convenience, as it is certainly possible to run executables directly from their build location.

3.7 Analyzing Produced Binaries with `analyze`

The `pyoxidizer analyze` command is a generic command for analyzing the contents of executables and libraries. While it is generic, its output is specifically tailored for `PyOxidizer`.

Run the command with the path to an executable. For example:

```
$ pyoxidizer analyze build/apps/myapp/x86_64-unknown-linux-gnu/debug/myapp
```

Behavior is dependent on the format of the file being analyzed. But the general theme is that the command attempts to identify the run-time requirements for that binary. For example, for ELF binaries it will list all shared library dependencies and analyze `glibc` symbol versions and print out which Linux distributions it thinks the binary is compatible with.

Note: `pyoxidizer analyze` is not yet implemented for all executable file types that `PyOxidizer` supports.

3.8 Inspecting Python Distributions

The `Python Building` phase of the lifecycle entails downloading special pre-built Python distributions and then linking them into a larger binary. You can find the location of these distributions in your project's `pyoxidizer.toml` configuration file.

These Python distributions are zstandard compressed tar files. Zstandard is a modern compression format that is really, really, really good. (PyOxidizer's maintainer also maintains Python bindings to zstandard and has written about the benefits of zstandard on his blog. You should read that blog post so you are enlightened on how amazing zstandard is.) But because zstandard is relatively new, not all systems have utilities for decompressing that format yet. So, the `pyoxidizer python-distribution-extract` command can be used to extract the zstandard compressed tar archive to a local filesystem path.

Python distributions contain software governed by a number of licenses. This of course has implications for application distribution. See *Licensing Considerations* for more.

The `pyoxidizer python-distribution-licenses` command can be used to inspect a Python distribution archive for information about its licenses. The command will print information about the licensing of the Python distribution itself along with a per-extension breakdown of which libraries are used by which extensions and which licenses apply to what. This command can be super useful to audit for license usage and only allow extensions with licenses that you are legally comfortable with.

For example, the entry for the `readline` extension shows that the extension links against the `ncurses` and `readline` libraries, which are governed by the X11, and GPL-3.0 licenses:

```
readline
--------

Dependency: ncurses
Link Type: library

Dependency: readline
Link Type: library

Licenses: GPL-3.0, X11
License Info: https://spdx.org/licenses/GPL-3.0.html
License Info: https://spdx.org/licenses/X11.html
```

Note: The license annotations in Python distributions are best effort and can be wrong. They do not constitute a legal promise. Paranoid individuals may want to double check the license annotations by verifying with source code distributions, for example.

CHAPTER 4

Packaging User Guide

So you want to package a Python application using PyOxidizer? You've come to the right place to learn how! Read on for all the details on how to *oxidize* your Python application!

First, you'll need to install PyOxidizer. See *Installing* for instructions.

4.1 Creating a PyOxidizer Project

Behind the scenes, PyOxidizer works by creating a Rust project which embeds and runs a Python interpreter.

The process for *oxidizing* every Python application looks the same: you start by creating a new [Rust] project with the PyOxidizer scaffolding. The `pyoxidizer init` command does this:

```
# Create a new project named "pyapp" in the directory "pyapp"
$ pyoxidizer init pyapp

# Create a new project named "myapp" in the directory "~/src/myapp"
$ pyoxidizer init ~/src/myapp
```

The default project created by `pyoxidizer init` will produce an executable that embeds Python and starts a Python REPL. Let's test that:

```
$ pyoxidizer run pyapp
no existing PyOxidizer artifacts found
processing config file /home/gps/src/pyapp/pyoxidizer.toml
resolving Python distribution...
   Compiling pyapp v0.1.0 (/home/gps/src/pyapp)
    Finished dev [unoptimized + debuginfo] target(s) in 53.14s
     Running `target/debug/pyapp`
>>>
```

If all goes according to plan, you just built a Rust executable which contains an embedded copy of Python. That executable started an interactive Python debugger on startup. Try typing in some Python code:

```
>>> print("hello, world")
hello, world
```

It works!

(To exit the REPL, press CTRL+d or CTRL+z or `import sys; sys.exit(0)` from the REPL.)

Note: If you have built a Rust project before, the output from building a PyOxidizer application may look familiar to you. That's because under the hood Cargo - Rust's package manager and build system - is doing a lot of the work to build the application. If you are familiar with Rust development, feel free to use `cargo build` and `cargo run` directly. However, Rust's build system is only responsible for some functionality. Most notable, all the post-build *packaging* steps such as copying binaries to the `build/apps` directory is not performed by the Rust build system, so built applications may be incomplete.

If you are curious about what's inside newly-created projects, read *New Project Layout*.

Now that we've got a new project, let's customize it to do something useful.

4.2 Packaging an Application from a PyPI Package

In this section, we'll show how to package the pyflakes program using a published PyPI package. (Pyflakes is a Python linter.)

First, let's create an empty project:

```
$ pyoxidizer init pyflakes
```

Next, we need to edit the *configuration file* to tell PyOxidizer about pyflakes. Open the `pyflakes/pyoxidizer.toml` file in your favorite editor.

We first tell PyOxidizer to add the `pyflakes` Python package by adding the following lines:

```
[[packaging_rule]]
type = "pip-install-simple"
package = "pyflakes==2.1.1"
```

This creates a packaging rule that essentially translates to running `pip install pyflakes==2.1.1` and then finds and packages the files installed by that command.

Next, we tell PyOxidizer to run pyflakes when the interpreter is executed. Find the `[[embedded_python_run]]` section and change its contents to the following:

```
[[embedded_python_run]]
mode = "eval"
code = "from pyflakes.api import main; main()"
```

This says to effectively run the Python code `eval(from pyflakes.api import main; main())` when the embedded interpreter starts.

The new `pyoxidizer.toml` file should look something like:

```
# Multiple [[python_distribution]] sections elided for brevity.

[[build]]
application_name = "pyflakes"
```

(continues on next page)

(continued from previous page)

```
[[embedded_python_config]]
raw_allocator = "system"

[[packaging_rule]]
type = "stdlib-extensions-policy"
policy = "all"

[[packaging_rule]]
type = "stdlib"
include_source = false

[[packaging_rule]]
type = "pip-install-simple"
package = "pyflakes==2.1.1"

[[embedded_python_run]]
mode = "eval"
code = "from pyflakes.api import main; main()"
```

With the configuration changes made, we can build and run a `pyflakes` native executable:

```
# From outside the ``pyflakes`` directory
$ pyoxidizer run /path/to/pyflakes/project -- /path/to/python/file/to/analyze

# From inside the ``pyflakes`` directory
$ pyoxidizer run -- /path/to/python/file/to/analyze

# Or if you prefer the Rust native tools
$ cargo run -- /path/to/python/file/to/analyze
```

By default, `pyflakes` analyzes Python source code passed to it via stdin.

4.3 What Can Go Wrong

Ideally, packaging your Python application and its dependencies *just works*. Unfortunately, we don't live in an ideal world.

PyOxidizer breaks various assumptions about how Python applications are built and distributed. When attempting to package your application, you will inevitably run into problems due to incompatibilities with PyOxidizer.

The *Packaging Pitfalls* documentation can serve as a guide to identify and work around these problems.

4.4 Packaging Additional Files

By default PyOxidizer will embed Python resources such as modules into the compiled executable. This is the ideal method to produce distributable Python applications because it can keep the entire application self-contained to a single executable and can result in *performance wins*.

But sometimes embedded resources into the binary isn't desired or doesn't work. Fear not: PyOxidizer has you covered!

As documented at *Install Locations*, many packaging rules in PyOxidizer configuration files can define an `install_location` that denotes where resources found by a packaging rule are installed.

Let's give an example of this by attempting to package black, a Python code formatter.

We start by creating a new project:

```
$ pyoxidizer init black
```

Then edit the `pyoxidizer.toml` file to have the following:

```
# Multiple [[python_distribution]] sections elided for brevity.

[[build]]
application_name = "black"

[[embedded_python_config]]
raw_allocator = "system"

[[packaging_rule]]
type = "stdlib-extensions-policy"
policy = "all"

[[packaging_rule]]
type = "stdlib"
include_source = false

[[packaging_rule]]
type = "pip-install-simple"
package = "black==19.3b0"

[[embedded_python_run]]
mode = "module"
module = "black"
```

Then let's attempt to build the application:

```
$ pyoxidizer build black
processing config file /home/gps/src/black/pyoxidizer.toml
resolving Python distribution...
...
packaging application into /home/gps/src/black/build/apps/x86_64-unknown-linux-gnu/
↪debug/black
purging /home/gps/src/black/build/apps/black/x86_64-unknown-linux-gnu/debug
copying /home/gps/src/black/build/target/x86_64-unknown-linux-gnu/debug/black to /
↪home/gps/src/black/build/apps/black/x86_64-unknown-linux-gnu/debug/black
resolving packaging state...
black packaged into /home/gps/src/black/build/apps/black/x86_64-unknown-linux-gnu/
↪debug
```

Looking good so far!

Now let's try to run it:

```
$  black/build/apps/black/x86_64-unknown-linux-gnu/debug/black
Traceback (most recent call last):
  File "black", line 46, in <module>
  File "blib2to3.pygram", line 15, in <module>
NameError: name '__file__' is not defined
SystemError
```

Uh oh - that's didn't work as expected.

As the error message shows, the `blib2to3.pygram` module is trying to access `__file__`, which is not defined. As explained by *Reliance on __file__*, PyOxidizer doesn't set `__file__` for modules loaded from memory. This is perfectly legal as Python doesn't mandate that `__file__` be defined. So `black` (and every other Python file assuming the existence of `__file__`) is buggy.

Let's assume we can't easily change the offending source code.

To fix this problem, we change the packaging rule to install `black` relative to the built application.

Simply change the following rule:

```
[[packaging_rule]]
type = "pip-install-simple"
package = "black==19.3b0"
```

To:

```
[[packaging_rule]]
type = "pip-install-simple"
package = "black==19.3b0"
install_location = "app-relative:lib"
```

The added `install_location = "app-relative:lib"` line says to set the installation location for resources found by that rule to a `lib` directory next to the built application.

In addition, we will also need to adjust the `[[embedded_python_config]]` section to have the following:

```
[[embedded_python_config]]
sys_paths = ["$ORIGIN/lib"]
```

The added `sys_paths = ["$ORIGIN/lib"]` line says to populate Python's `sys.path` list with a single entry which resolves to a `lib` sub-directory in the executable's directory. This configuration change is necessary to allow the Python interpreter to import Python modules from the filesystem and to find the modules that our `[[packaging_rule]]` installed into the `lib` directory.

Now let's re-build the application:

```
$ pyoxidizer build black
...
packaging application into /home/gps/src/black/build/apps/black/x86_64-unknown-linux-
↪gnu/debug
purging /home/gps/src/black/build/apps/black/x86_64-unknown-linux-gnu/debug
copying /home/gps/src/black/build/target/x86_64-unknown-linux-gnu/debug/black to /
↪home/gps/src/black/build/apps/black/x86_64-unknown-linux-gnu/debug/black
resolving packaging state...
installing resources into 1 app-relative directories
installing 46 app-relative Python source modules to /home/gps/src/black/build/apps/
↪black/x86_64-unknown-linux-gnu/debug/lib
...
black packaged into /home/gps/src/black/build/apps/black/x86_64-unknown-linux-gnu/
↪debug
```

If you examine the output, you'll see that various Python modules files were written to the `black/build/apps/black/x86_64-unknown-linux-gnu/debug/lib` directory, just as our packaging rules requested!

Let's try to run the application:

```
$  black/build/apps/black/x86_64-unknown-linux-gnu/debug/black
No paths given. Nothing to do
```

Success!

4.5 Trimming Unused Resources

By default, packaging rules are very aggressive about pulling in resources such as Python modules. For example, the entire Python standard library is embedded into the binary by default. These extra resources take up space and can make your binary significantly larger than it could be.

It is often desirable to *prune* your application of unused resources. For example, you may wish to only include Python modules that your application uses. This is possible with PyOxidizer.

Essentially, all strategies for managing the set of packaged resources boil down to crafting `[[packaging_rule]]`'s that choose which resources are packaged.

The recommended method to manage resources is the *filter-include* `[[packaging_rule]]`. This rule acts as an *allow list* filter against all resources identified for packaging. Using this rule, you can construct an explicit list of resources that should be packaged.

But maintaining explicit lists of resources can be tedious. There's a better way!

The *[[embedded_python_config]]* config section defines a `write_modules_directory_env` setting, which when enabled will instruct the embedded Python interpreter to write the list of all loaded modules into a randomly named file in the directory identified by the environment variable defined by this setting. For example, if you set `write_modules_directory_env = "PYOXIDIZER_MODULES_DIR"` and then run your binary with `PYOXIDIZER_MODULES_DIR=~/tmp/dump-modules`, each invocation will write a `~/tmp/dump-modules/modules-*` file containing the list of Python modules loaded by the Python interpreter.

One can therefore use `write_modules_directory_env` to produce files that can be referenced in a `filter-include` rule's `files` and `glob_files` settings.

While PyOxidizer doesn't yet automate the process, one could use a two phase build to *slim* your binary.

In phase 1, a binary is built with all resources and `write_modules_directory_env` enabled. The binary is then executed and `modules-*` files are written.

In phase 2, the `filter-include` rule is enabled and only the modules used by the instrumented binary will be packaged.

4.6 Adding Extension Modules At Run-Time

Normally, Python extension modules are compiled into the binary as part of the embedded Python interpreter.

PyOxidizer also supports providing additional extension modules at run-time. This can be useful for larger Rust applications providing extension modules that are implemented in Rust and aren't built through normal Python build systems (like `setup.py`).

If the `PythonConfig` Rust struct used to construct an embedded Python interpreter contains a populated `extra_extension_modules` field, the extension modules listed therein will be made available to the Python interpreter.

Please note that Python stores extension modules in a global variable. So instantiating multiple interpreters via the `pyembed` interfaces may result in duplicate entries or unwanted extension modules being exposed to the Python interpreter.

4.7 Masquerading As Other Packaging Tools

Tools to package and distribute Python applications existed several years before PyOxidizer. Many Python packages have learned to perform special behavior when the _fingerprint* of these tools is detected at run-time.

First, PyOxidizer has its own fingerprint: `sys.oxidized = True`. The presence of this attribute can indicate an application running with PyOxidizer.

Since PyOxidizer's run-time behavior is similar to other packaging tools, PyOxidizer supports falsely identifying itself as these other tools by emulating their fingerprints.

The `[[embedded_python_config]]` configuration section defines the boolean flag `sys_frozen` to control whether `sys.frozen = True` is set. This can allow PyOxidizer to advertise itself as a *frozen* application.

In addition, the `sys_meipass` boolean flag controls whether a `sys._MEIPASS = <exe directory>` attribute is set. This allows PyOxidizer to masquerade as having been built with PyInstaller.

Warning: Masquerading as other packaging tools is effectively lying and can be dangerous, as code relying on these attributes won't know if it is interacting with PyOxidizer or some other tool. It is recommended to only set these attributes to unblock enabling packages to work with PyOxidizer until other packages learn to check for `sys.oxidized = True`. Setting `sys._MEIPASS` is definitely the more risky option, as a case can be made that PyOxidizer should set `sys.frozen = True` by default.

CHAPTER 5

Packaging Pitfalls

While PyOxidizer is capable of building fully self-contained binaries containing a Python application, many Python packages and applications make assumptions that don't hold inside PyOxidizer. This section talks about all the things that can go wrong when attempting to package a Python application.

5.1 Reliance on `__file__`

Python modules typically have a `__file__` attribute that defines the path of the file from which the module was loaded. (When a file is executed as a script, it masquerades as the `__main__` module, so non-module scripts can behave as modules too.)

It is relatively common for Python modules in the wild to use `__file__`. For example, modules may do something like `module_dir = os.path.abspath(os.path.dirname(__file__))` to locate the directory that a module is in so they can load a non-Python file from that directory. Or they may use `__file__` to resolve paths to Python source files so that they can be loaded outside the typical `import` based mechanism (various plugin systems do this, for example).

Strictly speaking, the `__file__` attribute on modules is not required. Therefore any Python code that requires the existence of `__file__` is either broken or has made an explicit choice to not support module loaders - like PyOxidizer - that don't store modules as files and may not set `__file__`. **Therefore required use of __file__ is highly discouraged.** It is recommended to instead use a *resources* API for loading *resource* data relative to a Python module and to fall back to `__file__` if a suitable API is unavailable or doesn't work. See the next section for more.

5.2 Resource Reading

Many Python application need to load *resources*. *Resources* are typically non-Python *support* files, such as images, config files, etc. In some cases, *resources* could be Python source or bytecode files. For example, many plugin systems load Python modules outside the context of the normal `import` mechanism and therefore treat standalone Python source/bytecode files as non-module *resources*.

PyOxidizer can break existing resource reading code by invalidating assumptions about where resources are located. Historically, resources almost always translate to individual paths on the filesystem. One can use `__file__` to derive the path to a resource file and `open()` the file. So there is a lot of code in the wild that relies on `__file__` for this use case.

Important: Use of `__file__` will not work for in-memory resources in PyOxidizer applications and Python code will need to use a resource reading API to access resources data within the binary.

Depending on your need to support Python versions older than 3.7, the solution may or may not be simple. That's because for most of its lifetime, Python hasn't had a robust story for loading *resource* data. `pkg_resources` was the recommended solution for a while. Python 3 introduced the ResourceLoader interface on module loaders. But this interface became deprecated in Python 3.7 in favor of the ResourceReader interface and associated APIs in the importlib.resources module But even the modern `ResourceReader` interface isn't perfect, as some of its behavior is seemingly inconsistent.

`ResourceReader` is the best interface for importing non-module *resource* data to date. Unfortunately, that API requires Python 3.7. And a lot of the Python universe hasn't yet fully adopted Python 3.7 and its APIs. This means that Python projects in the wild tend to target the *lowest common denominator* for loading *resource* data. And this solution tends to be to rely on `__file__` (directly or abstracted away) for deriving paths to things because `__file__` has worked nearly everywhere for seemingly forever.

Important: PyOxidizer supports the ResourceReader interface on module loaders and highly encourages Python libraries and applications to adopt it as the preferred mechanism for loading resources data.

Let's talk about what this means in practice.

Say you have resources next to a Python module. Legacy code in a module might do something like the following:

```python
def get_resource(name):
    """Return a file handle on a named resource next to this module."""
    module_dir = os.path.abspath(os.path.dirname(__file__))
    # Warning: there is a path traversal attack possible here if
    # name continues values like ../../../../../etc/password.
    resource_path = os.path.join(module_dir, name)

    return open(resource_path, 'rb')
```

Modern code targeting Python 3.7+ can use the *ResourceReader* API directly:

```python
def get_resource(name):
    """Return a file handle on a named resource next to this module."""
    # get_resource_reader() may not exist or may return None, which this
    # code doesn't handle.
    reader = __loader__.get_resource_reader(__name__)
    return reader.open_resource(name)
```

Alternatively, you can use the functions in importlib.resources:

```python
import importlib.resources

with importlib.resources.open_binary('mypackage', 'resource-name') as fh:
    data = fh.read()
```

The `importlib.resources` functions are glorified wrappers around the low-level interfaces on module loaders. But they do provide some useful functionality, such as additional error checking and automatic importing of modules, making them useful in many scenarios, especially when loading resources outside the current package/module.

See the importlib_resources documentation site for more.

`ResourceReader` and `importlib.resources` were introduced in Python 3.7. So if you want your code to remain compatible with older Python versions, you will need to write an abstraction for obtaining resources. Try something like the following:

```
import importlib

try:
    import importlib.resources
    # Defeat lazy module importers.
    importlib.resources.open_binary
    HAVE_RESOURCE_READER = True
except ImportError:
    HAVE_RESOURCE_READER = False

try:
    import pkg_resources
    # Defeat lazy module importers.
    pkg_resources.resource_stream
    HAVE_PKG_RESOURCES = True
except ImportError:
    HAVE_PKG_RESOURCES = False


def get_resource(package, resource):
    """Return a file handle on a named resource in a Package."""

    # Prefer ResourceReader APIs, as they are newest.
    if HAVE_RESOURCE_READER:
        # If we're in the context of a module, we could also use
        # ``__loader__.get_resource_reader(__name__).open_resource(resource)``.
        # We use open_binary() because it is simple.
        return importlib.resources.open_binary(package, resource)

    # Fall back to pkg_resources.
    if HAVE_PKG_RESOURCES:
        return pkg_resources.resource_stream(package, resource)

    # Fall back to __file__.

    # We need to first import the package so we can find its location.
    # This could raise an exception!
    mod = importlib.import_module(package)

    # Undefined __file__ will raise NameError on variable access.
    try:
        package_path = os.path.abspath(os.path.dirname(mod.__file__))
    except NameError:
        package_path = None

    if package_path is not None:
        # Warning: there is a path traversal attack possible here if
        # resource contains values like ../../../../etc/password. Input
        # must be trusted or sanitized before blindly opening files or
        # you may have a security vulnerability!
        resource_path = os.path.join(package_path, resource)
```

(continues on next page)

(continued from previous page)

```
        return open(resource_path, 'rb')

    # Could not resolve package path from __file__.
    raise Exception('do not know how to load resource: %s:%s' % (
                    package, resource))
```

(The above code is dedicated to the public domain and can be used without attribution.)

The above code is just a demonstration. It may *just work* for your needs. It may need additional tweaking.

The state of resource management in Python has historically been a mess. So don't be surprised if you need to modify code to support the modern resource interfaces. But this effort should be well spent, as the new resource APIs are hopefully the most future compatible. And, using them will enable applications built with PyOxidizer to import resources data from memory!

5.3 C and Other Native Extension Modules

Many Python packages compile *extension modules* to native code. (Typically C is used to implement extension modules.)

The way this typically works is some build system (often `distutils` via a `setup.py` script) produces a shared library file containing the extension. On Linux and macOS, the file extension is typically `.so`. On Windows, it is `.pyd`. Python's importing mechanism looks for these files in addition to normal `.py` and `.pyc` files when an `import` is requested.

PyOxidizer currently has *limited support* for extension modules. Under some circumstances, building extension modules as part of regular package build machinery *just works* and the resulting extension module can be embedded in the produced binary.

The way PyOxidizer achieves this is a bit crude, but effective.

When PyOxidizer invokes `pip` or `setup.py` to build a package, it installs a modified version of `distutils` into the invoked Python's `sys.path`. This modified `distutils` changes the behavior of some key build steps (notably how C extensions are built) such that the build emits artifacts that PyOxidizer can use to integrate the extension module into a custom binary. For example, on Linux, PyOxidizer copies the intermediate object files produced by the build and links them into the same binary containing Python: PyOxidizer completely ignores the shared library that is or would typically be produced.

If `setup.py` scripts are following the traditional pattern of using distutils.core.Extension to define extension modules, things tend to *just work* (assuming extension modules are supported by PyOxidizer for the target platform). However, if `setup.py` scripts are doing their own monkeypatching of `distutils`, rely on custom build steps or types to compile extension modules, or invoke separate Python processes to interact with `distutils`, things may break.

If you run into an extension module packaging problem that isn't recorded here or on the *static page*, please file an issue so it may be tracked.

5.4 Identifying PyOxidizer

Python code may want to know whether it is running in the context of PyOxidizer.

At packaging time, pip and setup.py invocations made by PyOxidizer should set a PYOXIDIZER=1 environment variable. setup.py scripts, etc can look for this environment variable to determine if they are being packaged by PyOxidizer.

At run-time, PyOxidizer will always set a sys.oxidized attribute with value True. So, Python code can test whether it is running in PyOxidizer like so:

```
import sys

if getattr(sys, 'oxidized', False):
    print('running in PyOxidizer!')
```

CHAPTER 6

Distributing Binaries

There are a handful of considerations for distributing binaries built with PyOxidizer.

Foremost, PyOxidizer doesn't yet have a turnkey solution for various distribution problems. PyOxidizer currently produces a binary (typically an executable application) and then leaves the final packaging (like generating installers) up to the user. We eventually want to tackle some of these problems.

6.1 Binary Compatibility

Binaries produced with PyOxidizer should be able to run nearly anywhere. The details and caveats vary depending on the operating system and target platform and are documented in the sections below.

Important: Please create issues at https://github.com/indygreg/PyOxidizer/issues when the content of this section is incomplete or lacks important details.

The `pyoxidizer analyze` command can be used to analyze the contents of executables and libraries. For example, for ELF binaries it will list all shared library dependencies and analyze glibc symbol versions and print out which Linux distributions it thinks the binary is compatible with. Please note that `pyoxidizer analyze` is not yet implemented on all platforms.

6.1.1 Windows

Binaries built with PyOxidizer have a run-time dependency on various DLLs. Most of the DLLs are Windows system DLLs and should always be installed.

Binaries built with PyOxidizer have a dependency on the Visual Studio C++ Runtime. You will need to distribute a copy of `vcruntimeXXX.dll` alongside your binary or trigger the install of the Visual Studio C++ Redistributable in your application installer so the dependency is managed at the system level (the latter is preferred).

There is also currently a dependency on the Universal C Runtime (UCRT).

PyOxidizer will eventually make producing Windows installers from packaged applications turnkey. Until that time arrives, see the Microsoft documentation on deployment considerations for Windows binaries. The Dependency Walker tool is also useful for analyzing DLL dependencies.

6.1.2 macOS

The Python distributions that PyOxidizer consumers are built with `MACOSX_DEPLOYMENT_TARGET=10.9`, so they should be compatible with macOS versions 10.9 and newer.

The Python distribution has dependencies against a handful of system libraries and frameworks. These frameworks should be present on all macOS installations.

6.1.3 Linux

On Linux, a binary built with musl libc should *just work* on pretty much any Linux machine. See *Building Fully Statically Linked Binaries on Linux* for more.

If you are linking against `libc.so`, things get more complicated because the binary will probably link against the `glibc` symbol versions that were present on the build machine. To ensure maximum binary compatibility, compile your binary in a Debian 7 environment, as this will use a sufficiently old version of glibc which should work in most Linux environments.

Of course, if you control the execution environment (like if executables will run on the same machine that built them), then this may not pose a problem to you. Use the `pyoxidizer analyze` command to inspect binaries for compatibility before distributing a binary so you know what the requirements are.

6.1.4 Building Fully Statically Linked Binaries on Linux

It is possible to produce a fully statically linked executable embedding Python on Linux. The produced binary will have no external library dependencies nor will it even support loading dynamic libraries. In theory, the executable can be copied between Linux machines and it will *just work*.

Building such binaries requires using the `x86_64-unknown-linux-musl` Rust toolchain target. Using `pyoxidizer`:

```
$ pyoxidizer build --target x86_64-unknown-linux-musl
```

Specifying `--target x86_64-unknown-linux-musl` will cause PyOxidizer to use a Python distribution built against musl libc as well as tell Rust to target *musl on Linux*.

Targeting musl requires that Rust have the musl target installed. Standard Rust on Linux installs typically do not have this installed! To install it:

```
$ rustup target add x86_64-unknown-linux-musl
info: downloading component 'rust-std' for 'x86_64-unknown-linux-musl'
info: installing component 'rust-std' for 'x86_64-unknown-linux-musl'
```

If you don't have the musl target installed, you get a build time error similar to the following:

```
error[E0463]: can't find crate for `std`
  |
  = note: the `x86_64-unknown-linux-musl` target may not be installed
```

But even installing the target may not be sufficient! The standalone Python builds are using a modern version of musl and the Rust musl target must also be using this newer version or else you will see linking errors due to missing symbols. For example:

```
/build/Python-3.7.3/Python/bootstrap_hash.c:132: undefined reference to `getrandom'
/usr/bin/ld: /build/Python-3.7.3/Python/bootstrap_hash.c:132: undefined reference to␣
↪`getrandom'
/usr/bin/ld: /build/Python-3.7.3/Python/bootstrap_hash.c:136: undefined reference to␣
↪`getrandom'
/usr/bin/ld: /build/Python-3.7.3/Python/bootstrap_hash.c:136: undefined reference to␣
↪`getrandom'
```

Rust 1.37 or newer is required for the modern musl version compatibility. Rust 1.37 is Rust Nightly until July 4, 2019, at which point it becomes Rust Beta. It then becomes Rust Stable on August 15, 2019. You may need to override the Rust toolchain used to build your project so Rust 1.37+ is used. For example:

```
$ rustup override set nightly
$ rustup target add --toolchain nightly x86_64-unknown-linux-musl
```

This will tell Rust that the `nightly` toolchain should be used for the current directory and to install musl support for the `nightly` toolchain.

Then you can build away:

```
$ pyoxidizer build --target x86_64-unknown-linux-musl
$ ldd build/apps/myapp/x86_64-unknown-linux-musl/debug/myapp
     not a dynamic executable
```

Congratulations, you've produced a fully statically linked executable containing a Python application!

Important: There are reported performance problems with Python linked against musl libc. Application maintainers are therefore highly encouraged to evaluate potential performance issues before distributing binaries linked against musl libc.

It's worth noting that in the default configuration PyOxidizer binaries will use `jemalloc` for memory allocations, bypassing musl's apparently slower memory allocator implementation. This *may* help mitigate reported performance issues.

6.2 Licensing Considerations

Any time you link libraries together or distribute software, you need to be concerned with the licenses of the underlying code. Some software licenses - like the GPL - can require that any code linked with them be subject to the license and therefore be made open source. In addition, many licenses require a license and/or copyright notice be attached to works that use or are derived from the project using that license. So when building or distributing **any** software, you need to be cognizant about all the software going into the final work and any licensing terms that apply. Binaries produced with PyOxidizer are no different!

PyOxidizer and the code it uses in produced binaries is licensed under the Mozilla Public License version 2.0. The licensing terms are generally pretty favorable. (If the requirements are too strong, the code that ships with binaries could potentially use a *weaker* license. Get in touch with the project author.)

The Rust code PyOxidizer produces relies on a handful of 3rd party Rust crates. These crates have various licenses. We recommend using the cargo-license, cargo-tree, and cargo-lichking tools to examine the Rust crate dependency tree and their respective licenses. The `cargo-lichking` tool can even assemble licenses of Rust dependencies automatically so you can more easily distribute those texts with your application!

As cool as these Rust tools are, they don't include licenses for the Python distribution, the libraries its extensions link against, nor any 3rd party Python packages you may have packaged.

Python and its various dependencies are governed by a handful of licenses. These licenses have various requirements and restrictions.

At the very minimum, the binary produced with PyOxidizer will have a Python distribution which is governed by a license. You will almost certainly need to distribute a copy of this license with your application.

Various C-based extension modules part of Python's standard library link against other C libraries. For self-contained Python binaries, these libraries will be statically linked if they are present. That can trigger *stronger* license protections. For example, if all extension modules are present, the produced binary may contain a copy of the GPL 3.0 licensed `readline` and `gdbm` libraries, thus triggering strong copyleft protections in the GPL license.

Important: It is critical to audit which Python extensions and packages are being packaged because of licensing requirements of various extensions.

6.2.1 Showing Python Distribution Licenses

The special Python distributions that PyOxidizer consumes can annotate licenses of software within.

The `pyoxidizer python-distribution-licenses` command can display the licenses for the Python distribution and libraries it may link against. This command can be used to evaluate which extensions meet licensing requirements and what licensing requirements apply if a given extension or library is used.

6.3 Terminfo Database

Note: This section is not relevant to Windows.

If your application interacts with terminals (e.g. command line tools), your application may require the availability of a `terminfo` database so your application can properly interact with the terminal. The absence of a terminal database can result in the inability to properly colorize text, the backspace and arrow keys not working as expected, weird behavior on window resizing, etc. A `terminfo` database is also required to use `curses` or `readline` module functionality without issue.

UNIX like systems almost always provide a `terminfo` database which says which features and properties various terminals have. Essentially, the `TERM` environment variable defines the current terminal [emulator] in use and the `terminfo` database converts that value to various settings.

From Python, the `ncurses` library is responsible for consulting the `terminfo` database and determining how to interact with the terminal. This interaction with the `ncurses` library is typically performed from the `_curses`, `_curses_panel`, and `_readline` C extensions. These C extensions are wrapped by the user-facing `curses` and `readline` Python modules. And these Python modules can be used from various functionality in the Python standard library. For example, the `readline` module is used to power `pdb`.

PyOxidizer applications do not ship a terminfo database. Instead, applications rely on the `terminfo` database on the executing machine. (Of course, individual applications could ship a `terminfo` database if they want: the functionality just isn't included in PyOxidizer by default.) The reason PyOxidizer doesn't ship a `terminfo` database is that terminal configurations are very system and user specific: PyOxidizer wants to respect the configuration of the environment in which applications run. The best way to do this is to use the `terminfo` database on the executing machine instead of providing a static database that may not be properly configured for the run-time environment.

PyOxidizer applications have the choice of various modes for resolving the `terminfo` database location. This is facilitated mainly via the *terminfo_resolution* `[[embedded_python_config]]` config setting.

By default, when Python is initialized PyOxidizer will try to identify the current operating system and choose an appropriate set of well-known paths for that operating system. If the operating system is well-known (such as a Debian-based Linux distribution), this set of paths is fixed. If the operating system is not well-known, PyOxidizer will look for `terminfo` databases at common paths and use whatever paths are present.

If all goes according to plan, the default behavior *just works*. On common operating systems, the cost to the default behavior is reading a single file from the filesystem (in order to resolve the operating system). The overhead should be negligible. For unknown operating systems, PyOxidizer may need to `stat()` ~10 paths looking for the `terminfo` database. This should also complete fairly quickly. If the overhead is a concern for you, it is recommended to build applications with a fixed path to the `terminfo` database.

Under the hood, when PyOxidizer resolves the `terminfo` database location, it communicates these paths to `ncurses` by setting the `TERMINFO_DIRS` environment variable. If the `TERMINFO_DIRS` environment variable is already set at application run-time, PyOxidizer will **never** overwrite it.

The `ncurses` library that PyOxidizer applications ship with is also configured to look for a `terminfo` database in the current user's home directory (`HOME` environment variable) by default, specifically `$HOME/.terminfo`). Support for `termcap` databases is not enabled.

Note: `terminfo` database behavior is intrinsically complicated because various operating systems do things differently. If you notice oddities in the interaction of PyOxidizer applications with terminals, there's a good chance you found a deficiency in PyOxidizer's terminal detection logic (which is located in the `pyembed::osutils` Rust module).

Please report terminal interaction issues at https://github.com/indygreg/PyOxidizer/issues.

CHAPTER 7

Configuration Files

PyOxidizer uses TOML configuration files to configure how Python is packaged and built applications behave.

7.1 Finding Configuration Files

The TOML configuration file is processed as part of building the `pyembed` crate. This is the crate that manages an embedded Python interpreter in a larger Rust project.

If the `PYOXIDIZER_CONFIG` environment variable is set, the path specified by this environment variable will be used as the location of the TOML configuration file.

If `PYOXIDIZER_CONFIG` is not set, the build will look for a `pyoxidizer.toml` starting in the directory of the `pyembed` crate and then traversing ancestor directories until a file is found.

If no configuration file is found, an error occurs.

7.2 File Processing Semantics

Config files are processed by iterating through the various sections within them. Unless specified otherwise, when a new section type is encountered, a set of default values for that section type is initialized. As each new section instance is encountered, the section is examined to see if it is *applicable*. If it is, the settings it defines are set on the configuration object. The final set of values set for a given section type are used.

The configuration file format is designed to be simultaneously used by multiple build *targets*, where a target is a Rust toolchain target triple, such as `x86_64-unknown-linux-gnu` or `x86_64-pc-windows-msvc`. (Run `rustup target list` to see a list of targets.)

Each TOML section accepts an optional `built_target` key that can be used to control whether the section is applied or ignored. If the `built_target` key is not defined or has the special value `all`, it is always applied. Otherwise the section is only applied if its `build_target` value matches the Rust build target.

7.3 Configuration Sections

The following documentation sections describe the various TOML sections.

7.3.1 `[[build]]`

This section configures high-level application build settings.

`application_name` Name of the application being built.

This also corresponds to the name of the Rust binary to be built. A `cargo build --bin <application_name>` must work.

`build_path` Filesystem path to directory where build artifacts will be written.

Build artifacts include Rust build state, files generated by PyOxidizer, staging areas for built binaries, etc.

The special value `$ORIGIN` will be replaced by the directory holding this configuration file.

The default value is `$ORIGIN/build`.

7.3.2 `[[python_distribution]]`

Defines a Python distribution that can be embedded into a binary.

A Python distribution is a zstandard-compressed tar archive containing a specially produced build of Python. These distributions are typically produced by the python-build-standalone project. Pre-built distributions are available at https://github.com/indygreg/python-build-standalone/releases.

The `pyoxidizer` binary has a set of known distributions built-in which are automatically added to generated `pyoxidizer.toml` config files. Typically you don't need to build your own distribution or change the distribution manually: distributions are managed automatically by `pyoxidizer`.

A distribution is defined by a target triple, location, and a hash.

One of `local_path` or `url` MUST be defined.

`build_target` (string)

Target triple this distribution is compiled for.

`sha256` (string)

The SHA-256 of the distribution archive file.

`local_path` (string)

Local filesystem path to the distribution archive.

`url` (string)

URL from which a distribution archive can be obtained using an HTTP GET request.

Examples:

```
[[python_distribution]]
build_target = "x86_64-unknown-linux-gnu"
local_path = "/var/python-distributions/cpython-linux64.tar.zst"
sha256 = "11a53f5755773f91111a04f6070a6bc00518a0e8e64d90f58584abf02ca79081"
```

```
[[python_distribution]]
build_target = "x86_64-apple-darwin"
url = "https://github.com/indygreg/python-build-standalone/releases/download/20190505/
↪cpython-3.7.3-macos-20190506T0054.tar.zst"
sha256 = "b46a861c05cb74b5b668d2ce44dcb65a449b9fef98ba5d9ec6ff6937829d5eec"
```

7.3.3 `[[embedded_python_config]]`

This section configures the default behavior of the embedded Python interpreter.

Embedded Python interpreters are configured and instantiated using a `pyembed::PythonConfig` data structure. The `pyembed` crate defines a default instance of this data structure with parameters defined by the settings in this TOML section.

Note: If you are writing custom Rust code and constructing a custom `pyembed::PythonConfig` instance and don't use the default instance, this config section is not relevant to you and can be omitted from your config file.

The following keys can be defined to control the default `PythonConfig` behavior:

`dont_write_bytecode` (bool)

Controls the value of Py_DontWriteBytecodeFlag.

This is only relevant if the interpreter is configured to import modules from the filesystem.

Default is `true`.

`ignore_environment` (bool)

Controls the value of Py_IgnoreEnvironmentFlag.

This is likely wanted for embedded applications that don't behave like `python` executables.

Default is `true`.

`no_site` (bool)

Controls the value of Py_NoSiteFlag.

The `site` module is typically not needed for standalone Python applications.

Default is `true`.

`no_user_site_directory` (bool)

Controls the value of Py_NoUserSiteDirectory.

Default is `true`.

`optimize_level` (bool)

Controls the value of Py_OptimizeFlag.

Default is `0`, which is the Python default. Only the values `0`, `1`, and `2` are accepted.

This setting is only relevant if `dont_write_bytecode` is `false` and Python modules are being imported from the filesystem.

`stdio_encoding` (string)

Defines the encoding and error handling mode for Python's standard I/O streams (`sys.stdout`, etc). Values are of the form `encoding:error` e.g. `utf-8:ignore` or `latin1-strict`.

If defined, the `Py_SetStandardStreamEncoding()` function is called during Python interpreter initialization. If not, the Python defaults are used.

`unbuffered_stdio` (bool)

Controls the value of Py_UnbufferedStdioFlag.

Setting this makes the standard I/O streams unbuffered.

Default is `false`.

`filesystem_importer` (bool)

Controls whether to enable Python's filesystem based importer. Enabling this importer allows Python modules to be imported from the filesystem.

Default is `false` (since PyOxidizer prefers embedding Python modules in binaries).

`sys_frozen` (bool)

Controls whether to set the `sys.frozen` attribute to `True`. If `false`, `sys.frozen` is not set.

Default is `false`.

`sys_meipass` (bool)

Controls whether to set the `sys._MEIPASS` attribute to the path of the executable.

Setting this and `sys_frozen` to `true` will emulate the behavior of PyInstaller and could possibly help self-contained applications that are aware of PyInstaller also work with PyOxidizer.

Default is `false`.

`sys_paths` (array of strings)

Defines filesystem paths to be added to `sys.path`.

Setting this value will imply `filesystem_importer = true`.

The special token `$ORIGIN` in values will be expanded to the absolute path of the directory of the executable at run-time. For example, if the executable is `/opt/my-application/pyapp`, `$ORIGIN` will expand to `/opt/my-application` and the value `$ORIGIN/lib` will expand to `/opt/my-application/lib`.

If defined in multiple sections, new values completely overwrite old values (values are not merged).

Default is an empty array (`[]`).

`raw_allocator` (string)

Which memory allocator to use for the `PYMEM_DOMAIN_RAW` allocator.

This controls the lowest level memory allocator used by Python. All Python memory allocations use memory allocated by this allocator (higher-level allocators call into this pool to allocate large blocks then allocate memory out of those blocks instead of using the *raw* memory allocator).

Values can be `jemalloc`, `rust`, or `system`.

`jemalloc` will have Python use the jemalloc allocator directly.

`rust` will use Rust's global allocator (whatever that may be).

`system` will use the default allocator functions exposed to the binary (`malloc()`, `free()`, etc).

The `jemalloc` allocator requires the `jemalloc-sys` crate to be available. A run-time error will occur if `jemalloc` is configured but this allocator isn't available.

Important: the `rust` crate is not recommended because it introduces performance overhead.

Default is `jemalloc` on non-Windows targets and `system` on Windows. (The `jemalloc-sys` crate doesn't work on Windows MSVC targets.)

`terminfo_resolution` **(string)** How the terminal information database (`terminfo`) should be configured.

See *Terminfo Database* for more about terminal databases.

The value `dynamic` (the default) looks at the currently running operating system and attempts to do something reasonable. For example, on Debian based distributions, it will look for the `terminfo` database in `/etc/terminfo`, `/lib/terminfo`, and `/usr/share/terminfo`, which is how Debian configures `ncurses` to behave normally. Similar behavior exists for other recognized operating systems. If the operating system is unknown, PyOxidizer falls back to looking for the `terminfo` database in well-known directories that often contain the database (like `/usr/share/terminfo`).

The value `none` indicates that no configuration of the `terminfo` database path should be performed. This is useful for applications that don't interact with terminals. Using `none` can prevent some filesystem I/O at application startup.

The value `static` indicates that a static path should be used for the path to the `terminfo` database. That path should be provided by the `terminfo_dirs` configuration option.

`terminfo` is not used on Windows and this setting is ignored on that platform.

`terminfo_dirs` Path to the `terminfo` database. See the above documentation for `terminfo_resolution` for more on the `terminfo` database.

This value consists of a `:` delimited list of filesystem paths that `ncurses` should be configured to use. This value will be used to populate the `TERMINFO_DIRS` environment variable at application run time.

`write_modules_directory_env` (string)

Environment variable that defines a directory where `modules-<UUID>` files containing a `\n` delimited list of loaded Python modules (from `sys.modules`) will be written upon interpreter shutdown.

If this setting is not defined or if the environment variable specified by its value is not present at run-time, no special behavior will occur. Otherwise, the environment variable's value is interpreted as a directory, that directory and any of its parents will be created, and a `modules-<UUID>` file will be written to the directory.

This setting is useful for determining which Python modules are loaded when running Python code.

7.3.4 `[[embedded_python_run]]`

This section configures the default Python code to be executed by built binaries.

Embedded Python interpreters are configured and instantiated using a `pyembed::PythonConfig` data structure. The `pyembed` crate defines a default instance of this data structure with parameters defined by the settings in this TOML section.

Note: If you are writing custom Rust code and constructing a custom `pyembed::PythonConfig` instance and don't use the default instance, this config section is not relevant to you and can be omitted from your config file.

Instances of this section have a `mode` key that defines the mode of execution for the interpreter. The sections below describe these various modes.

`eval`

This mode will evaluate a string containing Python code after the interpreter initializes.

This mode requires the `code` key to be set to a string containing Python code to run.

Example:

```
[[embedded_python_run]]
mode = "eval"
code = "import mymodule; mymodule.main()"
```

`module`

This mode will load a named Python module as the `__main__` module and then execute that module.

This mode requires the `module` key to be set to the string value of the module to load as `__main__`.

Example:

```
[[embedded_python_run]]
mode = "module"
module = "mymodule"
```

`repl`

This mode will launch an interactive Python REPL connected to stdin. This is similar to the behavior of running a `python` executable without any arguments.

Example:

```
[[embedded_python_run]]
mode = "repl"
```

`noop`

This mode will do nothing. It is provided for completeness sake.

7.3.5 `[[packaging_rule]]`

Defines a rule to control the packaging of Python resources.

A *Python resource* can be one of the following:

- *Extension module*. An extension module is a Python module backed by compiled code (typically written in C).
- *Python module source*. A Python module's source code. This is typically the content of a `.py` file.
- *Python module bytecode*. A Python module's source compiled to Python bytecode. This is similar to a `.pyc` files but isn't exactly the same (`.pyc` files have a header in addition to the raw bytecode).
- *Resource file*. Non-module files that can be accessed via APIs in Python's importing mechanism.

Extension modules are a bit special in that they can have library dependencies. If an extension module has an annotated library dependency, that library will automatically be linked into the produced binary containing Python. Static linking is used, if available. For example, the `_sqlite3` extension module will link the `libsqlite3` library (which should be included as part of the Python distribution).

Each entry of this section describes a specific rule for finding and including or excluding resources. Each section has a `type` key describing the *flavor* of rule this is.

When packaging goes to resolve the set of resources, it starts with an empty set for each resource *flavor*. As sections are read, their results are *merged* with the existing resource sets according to the behavior of that rule `type`. If multiple rules add a resource of the same name and flavor, the last added version is used. i.e. *last write wins*.

Install Locations

Some rules support the concept of *install locations*. This allows resources to be packaged in different locations. For example, some resources can be embedded in the produced binary and others can live as files on the filesystem (like how Python traditionally works).

If a rule supports *install locations*, the string value defining an install location has the following values:

`embedded` Resource will be embedded in the produced binary.

This is usually the default install location.

`app-relative:<path>` Strings prefixed with `app-relative:` denote a path relative to the produced binary. The value following the prefix will be joined with the parent directory of the produced binary to form a base path for resources to be installed into.

For example, `app-relative:lib` would install resources into a `lib` child directory underneath where the produced binary lives.

Different resource types are mapped to different semantics for choosing the exact final path. Using the above example, a Python source module for the `foo.bar` module would be installed to `lib/foo/bar.py` or `lib/foo/bar/__init__.py` if it is a package module.

The following sections describe the various `type`'s of rules.

`stdlib-extension-policy`

This rule defines a base policy for what *extension modules* to include from the Python distribution.

This type has a `policy` key denoting the *policy* to use. This key can have the following values:

`minimal` Include the minimal set of extension modules required to initialize a Python interpreter. This is a very small set and various common functionality from the Python standard library will not work with this value.

`all` Includes all available extension modules in the Python distribution.

`no-libraries` Includes all available extension modules in the Python distribution that do not have an additional library dependency. Most common Python extension modules are included. Extension modules like `_ssl` (links against OpenSSL) and `zlib` are not included.

`no-gpl` Includes all available extension modules in the Python distribution that do not link against GPL licensed libraries.

Not all Python distributions may annotate license info for all extensions or the libraries they link against. If license info is missing, the extension is not included because it *could* be GPL licensed. Similarly, the mechanism for determining whether a license is GPL is based on an explicit list of non-GPL licenses. This ensures new GPL licenses don't slip through.

Example:

```
[[packaging_rule]]
type = "stdlib-extension-policy"
policy = "no-libraries"
```

Important: Libraries that extension modules link against have various software licenses, including GPL version 3. Adding these extension modules will also include the library. This typically exposes your program to additional licensing requirements, including making your application subject to that license and therefore open source. See *Licensing Considerations* for more.

stdlib-extensions-explicit-includes

This rule allows including explicitly delimited extension modules from the Python distribution.

The section must define an `includes` key, which is an array of strings of extension module names.

This policy is typically combined with the `minimal` `stdlib-extension-policy` to cherry pick individual extension modules for inclusion.

Example:

```
[[packaging_rule]]
type = "stdlib-extensions-explicit-includes"
includes = ["binascii", "errno", "itertools", "math", "select", "_socket"]
```

stdlib-extensions-explicit-excludes

This rule allows excluding explicitly delimited extension modules from the Python distribution.

The section must define an `excludes` key, which is an array of strings of extension module names.

Every known extension module not in `excludes` will be added. If an extension module with a name in `excludes` has already been added, it will be removed.

Example:

```
[[packaging_rule]]
type = "stdlib-extensions-explicit-excludes"
excludes = ["_ssl"]
```

stdlib-extension-variant

This rule specifies the inclusion of a specific extension module *variant*.

Some Python distributions offer multiple variants for an individual extension module. For example, the `readline` extension module may offer a `libedit` variant that is compiled against `libedit` instead of `libreadline` (the default).

By default, the first listed extension module variant in a Python distribution is used. By defining rules of this type, one can use an alternate or explicit extension module variation.

Extension module variants are defined the the `extension` and `variant` keys. The former defines the extension module name. The latter its variant name.

Example:

```
[[packaging_rule]]
type = "stdlib-extension-variant"
extension = "readline"
variant = "libedit"
```

stdlib

This rule controls packaging of non-extension modules Python resources from the Python distribution's standard library. Presence of this rule will pull in the Python standard library in its entirety.

Important: A `stdlib` rule is required, as Python can't be initialized without some modules from the standard library. It should be one of the first `[[packaging_rule]]` entries so the standard library forms the base of the set of Python modules to include.

The following keys can exist in this rule type:

`exclude_test_modules` (bool)

Indicates whether test-only modules should be included in packaging. The Python standard library ships various packages and modules that are used for testing Python itself. These modules are not referenced by *real* modules in the Python standard library and can usually be safely excluded.

Default is `true`.

`optimize_level` (int)

The optimization level for packaged bytecode. Allowed values are `0`, `1`, and `2`.

Default is `0`, which is the Python default.

`include_source` (bool)

Whether to include the source code for modules in addition to bytecode.

Default is `true`.

`include_resources` (bool)

Whether to include non-module resource files.

These are files like `lib2to3/Grammar.txt` which are present in the standard library but aren't typically used for common functionality.

Default is `false`.

`install_location` (string)

Where to package these resources. See *Install Locations*.

package-root

This rule discovers resources from a directory on the filesystem.

The specified directory will be scanned for resource files. However, only specific named *packages* will be packaged. e.g. if the directory contains sub-directories `foo/` and `bar`, you must explicitly state that you want the `foo` and/or `bar` package to be included so files from these directories will be included.

This rule is frequently used to pull in packages from local source directories (e.g. directories containing a `setup.py` file). This rule doesn't involve any packaging tools and is a purely driven by filesystem walking. It is primitive, yet effective.

This rule has the following keys:

`path` (string)

The filesystem path to the directory to scan.

`optimize_level` (int)

The module optimization level for packaged bytecode.

Allowed values are `0`, `1`, and `2`.

Defaults to `0`, which is the Python default.

`packages` (array of string)

List of package names to include.

Filesystem walking will find files in a directory `<path>/<value>/` or in a file `<path>/<value>.py`.

`excludes` (array of string)

An array of package or module names to exclude.

A value in this array will match on an exact full resource name match or on a package prefix match. e.g. `foo` will match the module `foo`, the package `foo`, and any sub-modules in `foo`. e.g. it will match `foo.bar` but will not match `foofoo`.

Default is an empty array.

`include_source` (bool)

Whether to include the source code for modules in addition to the bytecode.

Default is `true`.

`install_location` (string)

Where to package resources associated with this rule. See *Install Locations*.

pip-install-simple

This rule runs `pip install` for a single package and will automatically package all Python resources associated with that operation, including resources associated with dependencies.

Using this rule, one can easily add multiple Python packages with a single rule.

`package` (string)

Name of the package to install. This is added as a positional argument to `pip install`.

`optimize_level` (int)

The module optimization level for packaged bytecode.

Allowed values are `0`, `1`, and `2`.

Default is `0`, which is the Python default.

`include_source` (bool)

Whether to include the source code for Python modules in addition to the byte code.

Default is `true`.

`excludes` (array of string)

An array of package or module names to exclude. See the documentation for `excludes` for `package-root` rules for more.

Default is an empty array.

`install_location` (string)

Where to package resources associated with this rule. See *Install Locations*.

`extra_args` (optional array of string)

An array of arguments added to the pip install command.

This will include the `pyflakes` package and all its dependencies as embedded resources:

```
[[packaging_rule]]
type = "pip-install-simple"
package = "pyflakes"
```

This will include the `black` package and all its dependencies in a directory next to the produced binary:

```
[[packaging_rule]]
type = "pip-install-simple"
package = "black"
install_location = "app-relative:lib"
```

`pip-requirements-file`

This rule runs `pip install -r <path>` for a given pip requirements file. This allows multiple Python packages to be downloaded/installed in a single operation.

`requirements_path` (string)

Filesystem path to pip requirements file.

`optimize_level` (int)

The module optimization level for packaged bytecode.

Allowed values are `0`, `1`, and `2`.

Defaults to `0`, which is the Python default.

`include_source` (bool)

Whether to include the source code for Python modules in addition to the bytecode.

Default is `true`.

Example:

```
[[packaging_rule]]
type = "pip-requirements-file"
requirements_path = "/home/gps/src/myapp/requirements.txt"
```

`setup-py-install`

This rule runs `python setup.py install` for a given directory containing a `setup.py` `distutils`/`setuptools` packaging script.

The target package will be installed to a temporary directory and its installed resources will be collected and packaged.

`package_path` (string)

Local filesystem to the directory containing a `setup.py` file.

Can be a relative or absolute path. If relative, it is evaluated relative to the PyOxidizer configuration file.

The `setup.py` invocation will run with its current working directory set to this path.

`extra_env` (table)

Extra environment variables to pass to the `setup.py` invocation.

Some `setup.py` scripts accept environment variables to customize execution behavior. This option can be defined to pass those along to the invocation.

Typically inline table syntax is used. e.g. `extra_env = { FOO = "bar" }`.

`extra_global_arguments` (array of string)

Extra arguments to pass to `setup.py` before the `install` command.

Some `setup.py` scripts accept global arguments to control how the distribution is installed. This option can be defined to specify additional process arguments to the `setup.py` command.

`optimize_level` (int)

The module optimization level for packaged bytecode.

Allowed values are `0`, `1`, and `2`.

Defaults to `0`, which is the Python default.

`include_source` (bool)

Whether to include the source code for Python modules in addition to the bytecode.

Default is `true`.

`install_location` (string)

Where to package resources associated with this rule. See *Install Locations*.

`excludes` (array of string)

An array of package or module names to exclude. See the documentation for `excludes` for `package-root` rules for more.

Default is an empty array.

`virtualenv`

This rule will include resources found in a pre-populated *virtualenv* directory.

Important: PyOxidizer only supports finding modules and resources populated via *traditional* means (e.g. `pip install` or `python setup.py install`). If `.pth` or similar mechanisms are used for installing modules, files may not be discovered properly.

`path` (string)

The filesystem path to the root of the virtualenv.

Python modules are typically in a `lib/pythonX.Y/site-packages` directory (on UNIX) or `Lib/site-packages` directory (on Windows) under this path.

`optimize_level` (int)

The module optimization level for packaged bytecode.

Allowed values are `0`, `1`, and `2`.

Defaults to `0`, which is the Python default.

`excludes` (array of string)

An array of package or module names to exclude. See the documentation for `excludes` for `package-root` rules for more.

Default is an empty array.

`include_source` (bool)

Whether to include the source code for modules in addition to the bytecode.

Default is `true`.

`install_location` (string)

Where to package resources associated with this rule. See *Install Locations*.

Example:

```
[[packaging_rule]]
type = "virtualenv"
path = "/home/gps/src/myapp/venv"
```

write-license-files

This rule instructs packaging to write license files to a directory as denoted by this rule.

`path` (string) Filesystem path to directory where licenses should be written.

Value is relative to the application binary. An empty string denotes to write files in the same directory as the application binary.

filter-include

This rule filters all resource names resolved so far through a set of resource names resolved from sources defined by this section. Resources not contained in the set defined by this section will be removed.

This rule is effectively an *allow list*. This rule allows earlier rules to aggressively pull in resources only to filter them via this rule. This approach is often easier than adding a cherry picked set of resources via highly granular addition rules.

The section has keys that define various sources for resource names:

`files` (array of string)

List of filesystem paths to files containing resource names. The file must be valid UTF-8 and consist of a `\n` delimited list of resource names. Empty lines and lines beginning with `#` are ignored.

`glob_files` (array of string)

List of glob matching patterns of filter files to read. `*` denotes all files in a directory. `**` denotes recursive directories. This uses the Rust `glob` crate under the hood and the documentation for that crate contains more pattern matching info.

The files read by this key must be the same format as documented by the `files` key.

All defined keys have their resolved resources combined into a set of resource names. Each read entity has its values unioned with the set of values resolved so far.

Example:

```
[[packaging_rule]]
type = "filter-include"
files = ["allow-modules"]
glob_files = ["module-dumps/modules-*"]
```

7.3.6 `[[distribution]]`

Instances of the `[[distribution]]` section define application distributions that can be produced. An application distribution is an entity that can be shared across machines to *distribute* the application. Application distributions include archives, installers, packages, etc.

Each `[[distribution]]` section **must** define a `type` key. The value of this key defines the *flavor* of the distribution being produced. The various distribution `type`'s are described in the sections below.

`tarball`

The `type = "tarball"` distribution will produce a tar archive from the contents of the application directory.

This type accepts the following keys:

`path_prefix` String value that will be prepended to paths in the archive. By default, archive members have no path prefix and extraction of the archive will typically place files in the current directory. Specify this option to prefix all archive members with a path prefix.

`wix`

The `type = "wix"` distribution will produce Windows installers via the WiX Toolset. These installers allow the application to be easily installed on Windows.

This type accepts the following keys:

`msi_upgrade_code_x86` UUID to use for the x86 MSI installer. If not defined, a deterministic UUID based on the application name will be used.

`msi_upgrade_code_amd64` UUID to use for the x64 MSI installer. If not defined, a deterministic UUID based on the application name will be used.

`bundle_upgrade_code` UUID to use for the unified `.exe` bundle installer. The bundle installer contains the application's MSI installer as well as other dependencies (such as the Visual C++ Redistributable). This is typically the installer given to users.

If not defined, a deterministic UUID based on the application name will be used.

CHAPTER 8

Frequently Asked Questions

8.1 Where Can I Report Bugs / Send Feedback / Request Features?

At https://github.com/indygreg/PyOxidizer/issues

8.2 Why Build Another Python Application Packaging Tool?

It is true that several other tools exist to turn Python code into distributable applications! *Comparisons to Other Tools* attempts to exhaustively compare `PyOxidizer` to these myriad of tools. (If a tool is missing or the comparison incomplete or unfair, please file an issue so Python application maintainers can make better, informed decisions!)

The long version of how `PyOxidizer` came to be can be found in the Distributing Standalone Python Applications blog post. If you really want to understand the motivations for starting a new project rather than using or improving an existing one, read that post.

If you just want the extra concise version, at the time `PyOxidizer` was conceived, there were no Python application packaging/distribution tool which satisfied **all** of the following requirements:

- Works across all platforms (many tools target e.g. Windows or macOS only).
- Does not require an already-installed Python on the executing system (rules out e.g. zip file based distribution mechanisms).
- Has no special system requirements (e.g. SquashFS, container runtimes).
- Offers startup performance no worse than traditional `python` execution.
- Supports single file executables with none or minimal system dependencies.

8.3 Can Python 2.7 Be Supported?

In theory, yes. However, it is considerable more effort than Python 3. And since Python 2.7 is being deprecated in 2020, in the project author's opinion it isn't worth the effort.

8.4 `No python interpreter found of version 3.*` Error When Building

This is due to a dependent crate insisting that a Python executable exist on `PATH`. Set the `PYTHON_SYS_EXECUTABLE` environment variable to the path of a Python 3.7 executable and try again. e.g.:

```
# UNIX
$ export PYTHON_SYS_EXECUTABLE=/usr/bin/python3.7
# Windows
$ SET PYTHON_SYS_EXECUTABLE=c:\python37\python.exe
```

Note: The `pyoxidizer` tool should take care of setting `PYTHON_SYS_EXECUTABLE` and prevent this error. If you see this error and you are building with `pyoxidizer`, it is a bug that should be reported.

8.5 Why Rust?

This is really 2 separate questions:

- Why choose Rust for the run-time/embedding components?
- Why choose Rust for the build-time components?

`PyOxidizer` binaries require a *driver* application to interface with the Python C API and that *driver* application needs to compile to native code in order to provide a *native* executable without requiring a run-time on the machine it executes on. In the author's opinion, the only appropriate languages for this were C, Rust, and maybe C++.

Of those 3, the project's author prefers to write new projects in Rust because it is a superior systems programming language that has built on lessons learned from decades working with its predecessors. The author prefers technologies that can detect and eliminate entire classes of bugs (like buffer overflow and use-after-free) at compile time. On a less-opinionated front, Rust's built-in build system support means that we don't have to spend considerable effort solving hard problems like cross-compiling. Implementing the embedding component in Rust also creates interesting opportunities to embed Python in Rust programs. This is largely an unexplored area in the Python ecosystem and the author hopes that PyOxidizer plays a part in more people embedding Python in Rust.

For the non-runtime packaging side of `PyOxidizer`, pretty much any programming language would be appropriate. The project's author initially did prototyping in Python 3 but switched to Rust for synergy with the the run-time driver and because Rust had working solutions for several systems-level problems, such as parsing ELF, DWARF, etc executables, cross-compiling, integrating custom memory allocators, etc. A minor factor was the author's desire to learn more about Rust by starting a *real* Rust project.

8.6 Why is the Rust Code. . . Not Great?

This is the project author's first real Rust project. Suggestions to improve the Rust code would be very much appreciated!

Keep in mind that the `pyoxidizer` crate is a build-time only crate and arguably doesn't need to live up to quality standards as crates containing run-time code. Things like aggressive `.unwrap()` usage are arguably tolerable.

The run-time code that produced binaries run (`pyembed`) is held to a higher standard and is largely `panic!` free.

8.7 What is the *Magic Sauce* That Makes PyOxidizer Special?

There are 2 technical achievements that make `PyOxidizer` special.

First, `PyOxidizer` consumes Python distributions that were specially built with the aim of being used for standalone/distributable applications. These custom-built Python distributions are compiled in such a way that the resulting binaries have very few external dependencies and run on nearly every target system. Other tools that produce standalone Python binaries often rely on an existing Python distribution, which often doesn't have these characteristics.

Second is the ability to import `.py`/`.pyc` files from memory. Most other self-contained Python applications rely on Python's `zipimporter` or do work at run-time to extract the standard library to a filesystem (typically a temporary directory or a FUSE filesystem like SquashFS). What `PyOxidizer` does is expose the `.py`/`.pyc` modules data to the Python interpreter via a Python extension module built-in to the binary. In addition, the `importlib._bootstrap_external` module (which is *frozen* into `libpython`) is replaced by a modified version that defines a custom module importer capable of loading Python modules from the in-memory data structures exposed from the built-in extension module.

The custom `importlib_bootstrap_external` frozen module trick is probably the most novel technical achievement of `PyOxidizer`. Other Python distribution tools are encouraged to steal this idea!

See *pyembed Crate* for an overview of how the in-memory import machinery works.

8.8 Can Applications Import Python Modules from the Filesystem?

Yes. While the default is to import all Python modules from in-memory data structures linked into the binary, it is possible to configure `sys.path` to allow importing from additional filesystem paths. Support for importing compiled extension modules is also possible.

8.9 What are the Implications of Static Linking?

Most Python distributions rely heavily on dynamic linking. In addition to `python` frequently loading a dynamic `libpython`, many C extensions are compiled as standalone shared libraries. This includes the modules `_ctypes`, `_json`, `_sqlite3`, `_ssl`, and `_uuid`, which provide the native code interfaces for the respective non-_ prefixed modules which you may be familiar with.

These C extensions frequently link to other libraries, such as `libffi`, `libsqlite3`, `libssl`, and `libcrypto`. And more often than not, that linking is dynamic. And the libraries being linked to are provided by the system/environment Python runs in. As a concrete example, on Linux, the `_ssl` module can be provided by `_ssl.cpython-37m-x86_64-linux-gnu.so`, which can have a shared library dependency against `libssl.so.1.1` and `libcrypto.so.1.1`, which can be located in `/usr/lib/x86_64-linux-gnu` or a similar location under `/usr`.

When Python extensions are statically linked into a binary, the Python extension code is part of the binary instead of in a standalone file.

If the extension code is linked against a static library, then the code for that dependency library is part of the extension/binary instead of dynamically loaded from a standalone file.

When `PyOxidizer` produces a fully statically linked binary, the code for these 3rd party libraries is part of the produced binary and not loaded from external files at load/import time.

There are a few important implications to this.

One is related to security and bug fixes. When 3rd party libraries are provided by an external source (typically the operating system) and are dynamically loaded, once the external library is updated, your binary can use the latest version of the code. When that external library is statically linked, you need to rebuild your binary to pick up the latest version of that 3rd party library. So if e.g. there is an important security update to OpenSSL, you would need to ship a new version of your application with the new OpenSSL in order for users of your application to be secure. This shifts the security onus from e.g. your operating system vendor to you. This is less than ideal because security updates are one of those problems that tend to benefit from greater centralization, not less.

It's worth noting that PyOxidizer's library security story is the same as it is for e.g. Docker images. Docker images have the same security properties. If you are OK distributing Docker images, you should be OK with distributing executables built with PyOxidizer.

Another implication of static linking is licensing considerations. Static linking can trigger stronger licensing protections and requirements. Read more at *Licensing Considerations*.

8.10 `error while loading shared libraries: libcrypt.so.1: cannot open shared object file: No such file or directory` When Building

If you see this error when building, it is because your Linux system does not conform to the Linux Standard Base Specification, does not provide a `libcrypt.so.1` file, and the Python distribution that PyOxidizer attempts to run to compile Python source modules to bytecode can't execute.

Fedora 30+ are known to have this issue. A workaround is to install the `libxcrypt-compat` on the machine running `pyoxidizer`. See https://github.com/indygreg/PyOxidizer/issues/89 for more info.

CHAPTER 9

Project Status

PyOxidizer is functional and works for many use cases. However, there are still a number of rough edges, missing features, and known limitations. Please file issues at https://github.com/indygreg/PyOxidizer/issues!

9.1 What's Working

The basic functionality of creating binaries that embed a self-contained Python works on Linux, Windows, and macOS. The general approach should work for other operating systems.

TOML configuration files allow extensive customization of packaging and run time behavior. Many projects can be successfully packaged with PyOxidizer today.

9.2 Major Missing Features

9.2.1 An Official Build Environment

Compiling binaries that work on nearly every target system is hard. On Linux, things like `glibc` symbol versions from the build machine can leak into the built binary, effectively requiring a new Linux distribution to run a binary.

In order to make the binary build process robust, we will need to provide an execution environment in which to build portable binaries. On Linux, this likely entails making something like a Docker image available. On Windows and macOS, we might have to provide a tarball. In all cases, we want this environment to be integrated into `pyoxidizer build` so end users don't have to worry about jumping through hoops to build portable binaries.

9.2.2 Native Extension Modules

Building and using compiled extension modules (e.g. C extensions) is partially supported.

Building C extensions to be embedded in the produced binary works for Windows, Linux, and macOS.

Support for installing extension modules in app-relative paths is not yet implemented.

Support for extension modules that link additional macOS frameworks not used by Python itself is not yet implemented (but should be easy to do).

Support for cross-compiling extension modules (including to MUSL) does not work. (It may appear to work and break at linking or run-time.)

We also do not yet provide a build environment for C extensions. So unexpected behavior could occur if e.g. a different compiler toolchain is used to build the C extensions from the one that produced the Python distribution.

See also *C and Other Native Extension Modules*.

9.2.3 Incomplete `pyoxidizer` Commands

`pyoxidizer add` and `pyoxidizer analyze` aren't fully implemented.

There is no `pyoxidizer upgrade` command.

Work on all of these is planned.

9.2.4 More Robust Packaging Support

Currently, we produce an executable via Cargo. Often a self-contained executable is not suitable. We may have to run some Python modules from the filesystem because of limitations in those modules. In addition, some may wish to install custom files alongside the executable.

We want to add a myriad of features around packaging functionality to facilitate these things. This includes:

- Copying arbitrary files to live next to the executable.
- Specifying that certain modules should not be embedded in the binary.
- Support for `__file__`.
- A `pyoxidizer` command for turnkey building and assembling of all files.
- A build mode that produces an instrumented binary, runs it a few times to dump loaded modules into files, then builds it again with a pruned set of resources.

9.2.5 Making Distribution Easy

We don't yet have a good story for the *distributing* part of the application distribution problem. We're good at producing executables. But we'd like to go the extra mile and make it easier for people to produce installers, `.dmg` files, tarballs, etc.

This includes providing build environments for e.g. non-MUSL based Linux executables.

It also includes support for auditing for license compatibility (e.g. screening for GPL components in proprietary applications) and assembling required license texts to satisfy notification requirements in those licenses.

9.2.6 Partial Terminfo and Readline Support

PyOxidizer has partial support for detecting `terminfo` databases. See *Terminfo Database* for more.

There's a good chance PyOxidizer's ability to locate `terminfo` databases in the long tail of Python distributions is lacking. And PyOxidizer doesn't currently make it easy to distribute a `terminfo` database alongside the application.

At this time, proper terminal interaction in PyOxidizer applications may be hit-or-miss.

Please file issues at https://github.com/indygreg/PyOxidizer/issues reporting known problems with terminal interaction or to request new features for terminal interaction, `terminfo` database support, etc.

9.2.7 Test Coverage

The test coverage for `PyOxidizer` is pretty bad. We need to write a lot of tests.

9.3 Lesser Missing Features

9.3.1 Python Version Support

Only Python 3.7 is currently supported. Support for older Python 3 releases is possible. But the project author hopes we only need to target the latest/greatest Python release.

9.3.2 Reordering Resource Files

There is not yet support for reordering `.py` and `.pyc` files in the binary. This feature would facilitate linear read access, which could lead to faster execution.

9.3.3 Compressed Resource Files

Binary resources are currently stored as raw data. They could be stored compressed to keep binary size in check (at the cost of run-time memory usage and CPU overhead).

9.3.4 Nightly Rust Required on Windows

Windows currently requires a Nightly Rust to build (you can set the environment variable `RUSTC_BOOTSTRAP=1` to work around this) because the `static-nobundle` library type is required. https://github.com/rust-lang/rust/issues/37403 tracks making this feature stable. It *might* be possible to work around this by adding an `__imp_` prefixed symbol in the right place or by producing a empty import library to satisfy requirements of the `static` linkage kind. See https://github.com/rust-lang/rust/issues/26591#issuecomment-123513631 for more.

9.3.5 Cross Compiling

Cross compiling is not yet supported. We hope to and believe we can support this someday. We would like to eventually get to a state where you can e.g. produce Windows and macOS executables from Linux. It's possible.

9.3.6 TOML Configuration File

Naming and semantics in the TOML configuration files can be significantly improved. There's also various missing packaging functionality.

9.3.7 Poor Rust Error Handling

Error handling in build-time Rust code isn't great. Expect to see the `pyoxidizer` executable to crash from time to time. The code that runs in binaries built with PyOxidizer is held to a higher standard. Crashes should not occur and will be treated as serious bugs!

9.4 Eventual Features

The immediate goal of `PyOxidizer` is to solve packaging and distribution problems for Python applications. But we want `PyOxidizer` to be more than just a packaging tool: we want to add additional features to `PyOxidizer` to bring extra value to the tool and to demonstrate and/or experiment with alternate ways of solving various problems that Python applications frequently encounter.

9.4.1 Lazy Module Loading

When a Python module is ```import``ed, its code is evaluated. When applications consist of dozens or even hundreds of modules, the overhead of executing all this code at ``import``` time can be substantial and add up to dozens of milliseconds of overhead - all before your application runs a meaningful line of code.

We would like `PyOxidizer` to provide lazy module importing so Python's `import` machinery can defer evaluating a module's code until it is actually needed. With features in modern versions of Python 3, this feature could likely be enabled by default. And since many `PyOxidizer` applications are *frozen* and have total knowledge of all ```import``able modules at build time, ``PyOxidizer``` could return a *lazy* module object after performing a simple Rust `HashMap` lookup. This would be extremely fast.

9.4.2 Alternate Module Serialization Techniques

Related to lazy module loading, there is also the potential to explore alternate module serialization techniques. Currently, the way `PyOxidizer` and `.pyc` files work is that a Python code object is serialized with the `marshal` module. At module load time, the code object is deserialized and then executed. This deserialization plus code execution has overhead.

It is possible to devise alternate serialization and load techniques that don't rely on `marshal` and possibly bypass having to run as much code at module load time. For example, one could devise a format for serializing various `PyObject` types and then adjusting pointers inside the structs at run time. This is kind of a crazy idea. But it could work.

9.4.3 Module Order Tracing

Currently, resource data is serialized on disk in alphabetical order according to the resource name. e.g. the `bar` module is serialized before the `foo` module.

We would like to explore a mechanism to record the order in which modules are loaded as part of application execution and then reorder the serialized modules such that they are stored in load order. This will facilitate linear reads at application run time and possibly provide some performance wins (especially on devices with slow I/O).

9.4.4 Module Import Performance Tracing

`PyOxidizer` has near total visibility into what Python's module importer is doing. It could be very useful to provide forensic output of what modules import what, how long it takes to import various modules, etc.

CPython does have some support for module importing tracing. We think we can go a few steps farther. And we can implement it more easily in Rust than what CPython can do in C. For example, with Rust, one can use the inferno crate to emit flame graphs directly from Rust, without having to use external tools.

9.4.5 Built-in Profiler

There's potential to integrate a built-in profiler into `PyOxidizer` applications. The excellent py-spy sampling profiler (or the core components of it) could potentially be integrated directly into `PyOxidizer` such that produced applications could self-profile with minimal overhead.

It should also be possible for `PyOxidizer` to expose mechanisms for Rust to receive callbacks when Python's profiling and tracing hooks fire. This could allow building a powerful debugger or tracer in Rust.

9.4.6 Command Server

A known problem with Python is its startup overhead. The maintainer of `PyOxidizer` has raised this issue on Python's mailing list a few times.

`PyOxidizer` helps with this problem by eliminating explicit filesystem I/O and allowing modules to be imported faster. But there's only so much that can be done and startup overhead can still be a problem.

One strategy to combat this problem is the use of persistent *command server daemons*. Essentially, on the first invocation of a program you spawn a background process running Python. That process listens for *command requests* on a pipe, socket, etc. You send the current command's arguments, environment variables, other state, etc to the background process. It uses its Python interpreter to execute the command and send results back to the main process. On the 2nd invocation of your program, the Python process/interpreter is already running and meaningful Python code can be executed immediately, without waiting for the Python interpreter and your application code to initialize.

This approach is used by the Mercurial version control tool, for example, where it can shave dozens of milliseconds off of `hg` command service times.

`PyOxidizer` could potentially support *command servers* as a built-in feature for *any* Python application.

9.4.7 PyO3

PyO3 are alternate Rust bindings to Python from rust-cpython, which is what `pyembed` currently uses.

The `PyO3` bindings seem to be ergonomically better than *rust-cpython*. `PyOxidizer` may switch to `PyO3` someday. A hard blocker is that as of at least June 2019, `PyO3` requires Nightly Rust. We do not wish to make Nightly Rust a requirement to run `PyOxidizer`.

CHAPTER 10

Comparisons to Other Tools

What makes `PyOxidizer` different from other Python packaging and distribution tools? Read on to find out!

If you are curious why PyOxidizer's creator felt the need to create a new tool, see *Why Build Another Python Application Packaging Tool?* in the FAQ.

Important: It is important for Python application maintainers to make informed decisions about their use of packaging tools. If you feel the comparisons in this document are incomplete or unfair, please file an issue so this page can be improved.

10.1 PyInstaller

PyInstaller is a tool to convert regular python scripts to "standalone" executables. The standard packaging produces a tiny executable and a custom directory structure to host dynamic libraries and Python code (zipped compiled bytecode). `PyInstaller` can produce a self-contained executable file containing your application, however, at run-time, PyInstaller will extract binary files and a custom ZlibArchive <https://pyinstaller.readthedocs.io/en/latest/advanced-topics.html#zlibarchive>'_ to a temporary directory then import modules from the filesystem. ''PyOxidizer' typically skips this step and loads modules directly from memory using zero-copy. This makes `PyOxidizer` executables significantly faster to start.

Currently a big difference is that `PyOxidizer` needs to build all the binary dependecies from stratch to facilitate linking into single file, `PyInstaller` can work with normal Python packages with a complex system of hooks to find the runtime dependencies, this allow a lot of not easy to build packages like PyQt to work out of the box.

10.2 py2exe

py2exe is a tool for converting Python scripts into Windows programs, able to run without requiring an installation.

The goals of `py2exe` and `PyOxidizer` are conceptually very similar.

One major difference between the two is that `py2exe` works on just Windows whereas `PyOxidizer` works on multiple platforms.

One trick that `py2exe` employs is that it can load `libpython` and Python extension modules (which are actually dynamic link libraries) and other libraries from memory - not filesystem files. They employ a really clever hack to do this! This is similar in nature to what Google does internally with a custom build of glibc providing a dlopen_from_offset(). Essentially, `py2exe` embeds DLLs and other entities as *resources* in the PE file (the binary executable format for Windows) and is capable of loading them from memory. This allows `py2exe` to run things from a single binary, just like `PyOxidizer`! The main difference is `py2exe` relies on clever DLL loading tricks rather than `PyOxidizer`'s approach of using custom builds of Python (which exist as a single binary/library) to facilitate this. This is a really clever solution and `py2exe`'s authors deserve commendation for pulling this off!

The approach to packaging that `py2exe` and `PyOxidizer` take is substantially different. py2exe embeds itself into `setup.py` as a `distutils` extension. `PyOxidizer` wants to exist at a higher level and interact with the output of `setup.py` rather than get involved in the convoluted mess of `distutils` internals. This enables `PyOxidizer` to provide value beyond what `setup.py/distutils` can provide.

`py2exe` is a mature Python packaging/distribution tool for Windows. It offers a lot of similar functionality to `PyOxidizer`.

10.3 py2app

py2app is a setuptools command which will allow you to make standalone application bundles and plugins from Python scripts.

`py2app` only works on macOS. This makes it like a macOS version of `py2exe`. Most *comparisons to py2exe* are analogous for `py2app`.

10.4 cx_Freeze

cx_Freeze is a set of scripts and modules for freezing Python scripts into executables.

The goals of `cx_Freeze` and `PyOxidizer` are conceptually very similar.

Like other tools in the *produce executables* space, `cx_Freeze` packages Python traditionally. On Windows, this entails shipping a `pythonXY.dll`. `cx_Freeze` will also package dependent libraries found by binaries you are shipping. This introduces portability problems, especially on Linux.

`PyOxidizer` uses custom Python distributions that are built in such a way that they are highly portable across machines. `PyOxidizer` can also produce single file executables.

10.5 Shiv

Shiv is a packager for zip file based Python applications. The Python interpreter has built-in support for running self-contained Python applications that are distributed as zip files.

Shiv requires the target system to have a Python executable and for the target to support shebangs in executable files. This is acceptable for controlled *NIX environments. It isn't acceptable for Windows (which doesn't support shebangs) nor for environments where you can't guarantee an appropriate Python executable is available.

Also, by distributing our own Python interpreter with the application, PyOxidizer has stronger guarantees about the run-time environment. For example, your application can aggressively target the latest Python version. Another benefit of distributing your own Python interpreter is you can run a Python interpreter with various optimizations, such as profile-guided optimization (PGO) and link-time optimization (LTO). You can also easily configure custom memory allocators or tweak memory allocators for optimal performance.

10.6 PEX

PEX is a packager for zip file based Python applications. For purposes of comparison, PEX and Shiv have the same properties. See *Shiv* for this comparison.

10.7 XAR

XAR requires the use of SquashFS. SquashFS requires Linux.

`PyOxidizer` is a target native executable and doesn't require any special filesystems or other properties to run.

10.8 Docker / Running a Container

It is increasingly popular to distribute applications as self-contained container environments. e.g. Docker images. This distribution mechanism is effective for Linux users.

`PyOxidizer` will almost certainly produce a smaller distribution than container-based applications. This is because many container-based applications contain a lot of extra content that isn't needed by the processes within.

`PyOxidizer` also doesn't require a container execution environment. Not every user has the capability to run certain container formats. However, nearly every user can run an executable.

At run time, `PyOxidizer` executes a native binary and doesn't have to go through any additional execution layers. Contrast this with Docker, which uses HTTP requests to create containers, set up temporary filesystems and networks for the container, etc. Spawning a process in a new Docker container can take hundreds of milliseconds or more. This overhead can be prohibitive for low latency applications like CLI tools. This overhead does not exist for `PyOxidizer` executables.

10.9 Nuitka

Nuitka can compile Python programs to single executables. And the emphasis is on *compile*: Nuitka actually converts Python to C and compiles that. Nuitka is effectively an alternate Python interpreter.

Nuitka is a cool project and purports to produce significant speed-ups compared to CPython!

Since Nuitka is effectively a new Python interpreter, there are risks to running Python in this environment. Some code has dependencies on CPython behaviors. There may be subtle bugs are lacking features from Nuitka. However, Nuitka supposedly supports every Python construct, so many applications should *just work*.

Given the performance benefits of Nuitka, it is a compelling alternative to `PyOxidizer`.

10.10 PyRun

PyRun can produce single file executables. The author isn't sure how it works. PyRun doesn't appear to support modern Python versions. And it appears to require shared libraries (like bzip2) on the target system. `PyOxidizer` supports the latest Python and doesn't require shared libraries that aren't in nearly every environment.

10.11 pynsist

pynsist is a tool for building Windows installers for Python applications. pynsist is very similar in spirit to PyOxidizer.

A major difference between the projects is that pynsist focuses on solving the application distribution problem on Windows where `PyOxidizer` aims to solve larger problems around Python application distribution, such as performance optimization (via loading Python modules from memory instead of the filesystem).

`PyOxidizer` has yet to invest significantly into making producing distributable artifacts (such as Windows installers) simple, so pynsist still has an advantage over `PyOxidizer` here.

CHAPTER 11

Contributing to PyOxidizer

This page documents how to contribute to PyOxidizer.

11.1 As a User

PyOxidizer is currently a relative young project and could substantially benefit from reports from its users.

Try to package applications with PyOxidizer. If things break or are hard to learn, file an issue on GitHub.

You can also join the pyoxidizer-users mailing list to report your experience, get in touch with other users, etc.

11.2 As a Developer

If you would like to contribute to the code behind PyOxidizer, you can do so using a standard GitHub workflow through the canonical project home at https://github.com/indygreg/PyOxidizer.

Please note that PyOxidizer's maintainer can be quite busy from time to time. So please be patient. He will be patient with you.

The documentation around how to hack on the PyOxidizer codebase is a bit lacking. Sorry for that!

The most important command for contributors to know how to run is `cargo run --bin pyoxidizer`. This will compile the `pyoxidizer` executable program and run it. Use it like `cargo run --bin pyoxidizer -- init ~/tmp/myapp` to run `pyoxidizer init ~/tmp/myapp` for example. If you just run `cargo build`, it will also build the `pyapp` project, which is an in-repo project that attempts to use PyOxidizer.

11.3 Financial Contributions

If you would like to thank the PyOxidizer maintainer via a financial contribution, you can do so on his Patreon or via PayPal.

Financial contributions of any amount are appreciated. Please do not feel obligated to donate money: only donate if you are financially able and feel the maintainer deserves the reward for a job well done.

CHAPTER 12

Project History

Work on PyOxidizer started in November 2018 by Gregory Szorc.

12.1 Blog Posts

- C Extension Support in PyOxidizer (2019-06-30)
- Building Standalone Python Applications with PyOxidizer (2019-06-24)
- PyOxidizer Support for Windows (2019-01-06)
- Faster In-Memory Python Module Importing (2018-12-28)
- Distributing Standalone Python Applications (2018-12-18)

12.2 Version History

12.2.1 0.4.0

Released October 27, 2019.

Backwards Compatibility Notes

- The `setup-py-install` packaging rule now has its `package_path` evaluated relative to the PyOxidizer config file path rather than the current working directory.

Bug Fixes

- Windows now explicitly requires dynamic linking against `msvcrt`. Previously, this wasn't explicit. And sometimes linking the final executable would result in unresolved symbol errors because the Windows Python distributions used external linkage of CRT symbols and for some reason Cargo wasn't dynamically linking the CRT.
- Read-only files in Python distributions are now made writable to avoid future permissions errors (#123).
- In-memory `InspectLoader.get_source()` implementation no longer errors due to passing a `memoryview` to a function that can't handle it (#134).
- In-memory `ResourceReader` now properly handles multiple resources (#128).

New Features

- Added an `app-path` command that prints the path to a packaged application. This command can be useful for tools calling PyOxidizer, as it will emit the path containing the packaged files without forcing the caller to parse command output.
- The `setup-py-install` packaging rule now has an `excludes` option that allows ignoring specific packages or modules.
- `.py` files installed into app-relative locations now have corresponding `.pyc` bytecode files written.
- The `setup-py-install` packaging rule now has an `extra_global_arguments` option to allow passing additional command line arguments to the `setup.py` invocation.
- Packaging rules that invoke `pip` or `setup.py` will now set a `PYOXIDIZER=1` environment variable so Python code knows at packaging time whether it is running in the context of PyOxidizer.
- The `setup-py-install` packaging rule now has an `extra_env` option to allow passing additional environment variables to `setup.py` invocations.
- `[[embedded_python_config]]` now supports a `sys_frozen` flag to control setting `sys.frozen = True`.
- `[[embedded_python_config]]` now supports a `sys_meipass` flag to control setting `sys._MEIPASS = <exe directory>`.
- Default Python distribution upgraded to 3.7.5 (from 3.7.4). Various dependency packages also upgraded to latest versions.

All Other Relevant Changes

- Built extension modules marked as app-relative are now embedded in the finaly binary rather than being ignored.

12.2.2 0.3.0

Released on August 16, 2019.

Backwards Compatibility Notes

- The `pyembed::PythonConfig` struct now has an additional `extra_extension_modules` field.
- The default musl Python distribution now uses LibreSSL instead of OpenSSL. This should hopefully be an invisible change.

- Default Python distributions now use CPython 3.7.4 instead of 3.7.3.
- Applications are now built into directories named `apps/<app_name>/<target>/<build_type>` rather than `apps/<app_name>/<build_type>`. This enables builds for multiple targets to coexist in an application's output directory.
- The `program_name` field from the `[[embedded_python_config]]` config section has been removed. At run-time, the current executable's path is always used when calling `Py_SetProgramName()`.
- The format of embedded Python module data has changed. The `pyembed` crate and `pyoxidizer` versions must match exactly or else the `pyembed` crate will likely crash at run-time when parsing module data.

Bug Fixes

- The `libedit` extension variant for the `readline` extension should now link on Linux. Before, attempting to link a binary using this extension variant would result in missing symbol errors.
- The `setup-py-install [[packaging_rule]]` now performs actions to appease `setuptools`, thus allowing installation of packages using `setuptools` to (hopefully) work without issue (#70).
- The `virtualenv [[packaging_rule]]` now properly finds the `site-packages` directory on Windows (#83).
- The `filter-include [[packaging_rule]]` no longer requires both `files` and `glob_files` be defined (#88).
- `import ctypes` now works on Windows (#61).
- The in-memory module importer now implements `get_resource_reader()` instead of `get_resource_loader()`. (The CPython documentation steered us in the wrong direction - https://bugs.python.org/issue37459.)
- The in-memory module importer now correctly populates `__package__` in more cases than it did previously. Before, whether a module was a package was derived from the presence of a `foo.bar` module. Now, a module will be identified as a package if the file providing it is named `__init__`. This more closely matches the behavior of Python's filesystem based importer. (#53)

New Features

- The default Python distributions have been updated. Archives are generally about half the size from before. Tcl/tk is included in the Linux and macOS distributions (but PyOxidizer doesn't yet package the Tcl files).
- Extra extension modules can now be registered with `PythonConfig` instances. This can be useful for having the application embedding Python provide its own extension modules without having to go through Python build mechanisms to integrate those extension modules into the Python executable parts.
- Built applications now have the ability to detect and use `terminfo` databases on the execution machine. This allows applications to interact with terminals properly. (e.g. the backspace key will now work in interactive `pdb` sessions). By default, applications on non-Windows platforms will look for `terminfo` databases at well-known locations and attempt to load them.
- Default Python distributions now use CPython 3.7.4 instead of 3.7.3.
- A warning is now emitted when a Python source file contains `__file__`. This should help trace down modules using `__file__`.
- Added 32-bit Windows distribution.
- New `pyoxidizer distribution` command for producing distributable artifacts of applications. Currently supports building tar archives and `.msi` and `.exe` installers using the WiX Toolset.

- Libraries required by C extensions are now passed into the linker as library dependencies. This should allow C extensions linked against libraries to be embedded into produced executables.
- `pyoxidizer --verbose` will now pass verbose to invoked `pip` and `setup.py` scripts. This can help debug what Python packaging tools are doing.

All Other Relevant Changes

- The list of modules being added by the Python standard library is no longer printed during rule execution unless `--verbose` is used. The output was excessive and usually not very informative.

12.2.3 0.2.0

Released on June 30, 2019.

Backwards Compatibility Notes

- Applications are now built into an `apps/<appname>/(debug|release)` directory instead of `apps/<appname>`. This allows debug and release builds to exist side-by-side.

Bug Fixes

- Extracted `.egg` directories in Python package directories should now have their resources detected properly and not as Python packages with the name `*.egg`.
- `site-packages` directories are now recognized as Python resource package roots and no longer have their contents packaged under a `site-packages` Python package.

New Features

- Support for building and embedding C extensions on Windows, Linux, and macOS in many circumstances. See *Native Extension Modules* for support status.
- `pyoxidizer init` now accepts a `--pip-install` option to pre-configure generated `pyoxidizer.toml` files with packages to install via `pip`. Combined with the `--python-code` option, it is now possible to create `pyoxidizer.toml` files for a ready-to-use Python application!
- `pyoxidizer` now accepts a `--verbose` flag to make operations more verbose. Various low-level output is no longer printed by default and requires `--verbose` to see.

All Other Relevant Changes

- Packaging now automatically creates empty modules for missing parent packages. This prevents a module from being packaged without its parent. This could occur with *namespace packages*, for example.
- `pip-install-simple` rule now passes `--no-binary :all:` to pip.
- Cargo packages updated to latest versions.

12.2.4 0.1.3

Released on June 29, 2019.

Bug Fixes

- Fix Python refcounting bug involving call to `PyImport_AddModule()` when `mode = module` evaluation mode is used. The bug would likely lead to a segfault when destroying the Python interpreter. (#31)
- Various functionality will no longer fail when running `pyoxidizer` from a Git repository that isn't the canonical `PyOxidizer` repository. (#34)

New Features

- `pyoxidizer init` now accepts a `--python-code` option to control which Python code is evaluated in the produced executable. This can be used to create applications that do not run a Python REPL by default.
- `pip-install-simple` packaging rule now supports `excludes` for excluding resources from packaging. (#21)
- `pip-install-simple` packaging rule now supports `extra_args` for adding parameters to the pip install command. (#42)

All Relevant Changes

- Minimum Rust version decreased to 1.31 (the first Rust 2018 release). (#24)
- Added CI powered by Azure Pipelines. (#45)
- Comments in auto-generated `pyoxidizer.toml` have been tweaked to improve understanding. (#29)

12.2.5 0.1.2

Released on June 25, 2019.

Bug Fixes

- Honor `HTTP_PROXY` and `HTTPS_PROXY` environment variables when downloading Python distributions. (#15)
- Handle BOM when compiling Python source files to bytecode. (#13)

All Relevant Changes

- `pyoxidizer` now verifies the minimum Rust version meets requirements before building.

12.2.6 0.1.1

Released on June 24, 2019.

Bug Fixes

- `pyoxidizer` binaries built from crates should now properly refer to an appropriate commit/tag in PyOxidizer's canonical Git repository in auto-generated `Cargo.toml` files. (#11)

12.2.7 0.1

Released on June 24, 2019. This is the initial formal release of PyOxidizer. The first `pyoxidizer` crate was published to `crates.io`.

New Features

- Support for building standalone, single file executables embedding Python for 64-bit Windows, macOS, and Linux.
- Support for importing Python modules from memory using zero-copy.
- Basic Python packaging support.
- Support for jemalloc as Python's memory allocator.
- `pyoxidizer` CLI command with basic support for managing project lifecycle.

CHAPTER 13

pyembed Crate

The pyembed crate contains functionality for managing a Python interpreter embedded in a binary. This crate is typically used along PyOxidizer for producing self-contained binaries containing Python.

pyembed provides significant additional functionality over what is covered by the official Embedding Python in Another Application docs and provided by the CPython C API. For example, pyembed defines a custom Python *meta path importer* that can import Python module bytecode from memory using 0-copy. This added functionality is the *magic sauce* that makes pyembed/PyOxidizer stand out from other tools in this space.

From a very high level, this crate serves as a bridge between Rust and various Python C APIs for interfacing with an in-process Python interpreter. This crate *could* potentially be used as a generic interface to any linked/embedded Python distribution. However, this crate is optimized for use with embedded Python interpreters produced with PyOxidizer. Use of this crate without PyOxidizer is strongly discouraged at this time.

13.1 Dependencies

Under the hood, pyembed makes direct use of the python-sys crate for low-level Python FFI bindings as well as the cpython crate for higher-level interfacing. Due to our special needs, **we currently require a fork of these crates**. These forks are maintained in the canonical Git repository. Customizations to these crates are actively upstreamed and the requirement to use a fork should go away in time.

It is an explicit goal of this crate to rely on as few external dependencies as possible. This is because we want to minimize bloat in produced binaries. At this time, we have required direct dependencies on published versions of the byteorder, libc, and uuid crates and on unpublished/forked versions of the python3-sys and cpython crates. We also have an optional direct dependency on the jemalloc-sys crate. Via the cpython crate, we also have an indirect dependency on the num-traits crate.

This crate requires linking against a library providing CPython C symbols. (This dependency is via the python3-sys crate.) On Windows, this library must be named pythonXY. This library is typically generated with PyOxidizer and its linking is managed by the build.rs build script.

13.2 Features

The optional `jemalloc-sys` feature controls support for using jemalloc as Python's memory allocator. Use of Jemalloc from Python is a run-time configuration option controlled by the `PythonConfig` type and having `jemalloc` compiled into the binary does not mean it is being used!

13.3 Technical Implementation Details

When trying to understand the code, a good place to start is `MainPythonInterpreter.new()`, as this will initialize the CPython runtime and Python initialization is where most of the magic occurs.

A lot of initialization code revolves around mapping `PythonConfig` members to C API calls. This functionality is rather straightforward. There's nothing really novel or complicated here. So we won't cover it.

13.3.1 Python Memory Allocators

There exist several CPython APIs for memory management. CPython defines multiple memory allocator *domains* and it is possible to use a custom memory allocator for each using the `PyMem_SetAllocator()` API.

We support having the *raw* memory allocator use either `jemalloc` or Rust's global allocator.

The `pyalloc` module defines types that serve as interfaces between the `jemalloc` library and Rust's allocator. The reason we call into `jemalloc-sys` directly instead of going through Rust's allocator is overhead: why involve an extra layer of abstraction when it isn't needed. To register a custom allocator, we simply instantiate an instance of the custom allocator type and tell Python about it via `PyMem_SetAllocator()`.

13.3.2 Module Importing

The module importing mechanisms provided by this crate are one of the most complicated parts of the crate. This section aims to explain how it works. But before we go into the technical details, we need an understanding of how Python module importing works.

High Level Python Importing Overview

A *meta path importer* is a Python object implementing the importlib.abc.MetaPathFinder interface and is registered on sys.meta_path. Essentially, when the `__import__` function / `import` statement is called, Python's importing internals traverse entities in `sys.meta_path` and ask each *finder* to load a module. The first *meta path importer* that knows about the module is used.

By default, Python configures 3 *meta path importers*: an importer for built-in extension modules (`BuiltinImporter`), frozen modules (`FrozenImporter`), and filesystem-based modules (`PathFinder`). You can see these on a fresh Python interpreter:

```
$ python3.7 -c 'import sys; print(sys.meta_path)`
[<class '_frozen_importlib.BuiltinImporter'>, <class '_frozen_importlib.FrozenImporter
↪'>, <class '_frozen_importlib_external.PathFinder'>]
```

These types are all implemented in Python code in the Python standard library, specifically in the `importlib._bootstrap` and `importlib._bootstrap_external` modules.

Built-in extension modules are compiled into the Python library. These are often extension modules required by core Python (such as the `_codecs`, `_io`, and `_signal` modules). But it is possible for other extensions - such as those provided by Python's standard library or 3rd party packages - to exist as built-in extension modules as well.

For importing built-in extension modules, there's a global `PyImport_Inittab` array containing members defining the extension/module name and a pointer to its C initialization function. There are undocumented functions exported to Python (such as `_imp.exec_builtin()` that allow Python code to call into C code which knows how to e.g. instantiate these extension modules. The `BuiltinImporter` calls into these C-backed functions to service imports of built-in extension modules.

Frozen modules are Python modules that have their bytecode backed by memory. There is a global `PyImport_FrozenModules` array that - like `PyImport_Inittab` - defines module names and a pointer to bytecode data. The `FrozenImporter` calls into undocumented C functions exported to Python to try to service import requests for frozen modules.

Path-based module loading via the `PathFinder` meta path importer is what most people are likely familiar with. It uses `sys.path` and a handful of other settings to traverse filesystem paths, looking for modules in those locations. e.g. if `sys.path` contains `['', '/usr/lib/python3.7', '/usr/lib/python3.7/lib-dynload', '/usr/lib/python3/dist-packages']`, `PathFinder` will look for `.py`, `.pyc`, and compiled extension modules (`.so`, `.dll`, etc) in each of those paths to service an import request. Path-based module loading is a complicated beast, as it deals with all kinds of complexity like caching bytecode `.pyc` files, differentiating between Python modules and extension modules, namespace packages, finding search locations in registry entries, etc. Altogether, there are 1500+ lines constituting path-based importing logic in `importlib._bootstrap_external`!

Default Initialization of Python Importing Mechanism

CPython's internals go through a convoluted series of steps to initialize the importing mechanism. This is because there's a bit of chicken-and-egg scenario going on. The *meta path importers* are implemented as Python modules using Python source code (`importlib._bootstrap` and `importlib._bootstrap_external`). But in order to execute Python code you need an initialized Python interpreter. And in order to execute a Python module you need to import it. And how do you do any of this if the importing functionality is implemented as Python source code and as a module?!

A few tricks are employed.

At Python build time, the source code for `importlib._bootstrap` and `importlib._bootstrap_external` are compiled into bytecode. This bytecode is made available to the global `PyImport_FrozenModules` array as the `_frozen_importlib` and `_frozen_importlib_external` module names, respectively. This means the bytecode is available for Python to load from memory and the original `.py` files are not needed.

During interpreter initialization, Python initializes some special built-in extension modules using its internal import mechanism APIs. These bypass the Python-based APIs like `__import__`. This limited set of modules includes `_imp` and `sys`, which are both completely implemented in C.

During initialization, the interpreter also knows to explicitly look for and load the `_frozen_importlib` module from its frozen bytecode. It creates a new module object by hand without going through the normal import mechanism. It then calls the `_install()` function in the loaded module. This function executes Python code on the partially bootstrapped Python interpreter which culminates with `BuiltinImporter` and `FrozenImporter` being registered on `sys.meta_path`. At this point, the interpreter can import compiled built-in extension modules and frozen modules. Subsequent interpreter initialization henceforth uses the initialized importing mechanism to import modules via normal import means.

Later during interpreter initialization, the `_frozen_importlib_external` frozen module is loaded from bytecode and its `_install()` is also called. This self-installation adds `PathFinder` to `sys.meta_path`. At this point, modules can be imported from the filesystem. This includes `.py` based modules from the Python standard library as well as any 3rd party modules.

Interpreter initialization continues on to do other things, such as initialize signal handlers, initialize the filesystem encoding, set up the `sys.std*` streams, etc. This involves importing various `.py` backed modules (from the filesystem). Eventually interpreter initialization is complete and the interpreter is ready to execute the user's Python code!

Our Importing Mechanism

We have made significant modifications to how the Python importing mechanism is initialized and configured. (Note: we do not require these modifications. It is possible to initialize a Python interpreter with *default* behavior, without support for in-memory module importing.)

The `importer` Rust module of this crate defines a Python extension module. To the Python interpreter, an extension module is a C function that calls into the CPython C APIs and returns a `PyObject*` representing the constructed Python module object. This extension module behaves like any other extension module you've seen. The main differences are it is implemented in Rust (instead of C) and it is compiled into the binary containing Python, as opposed to being a standalone shared library that is loaded into the Python process.

This extension module provides the `_pyoxidizer_importer` Python module, which provides a global `_setup()` function to be called from Python.

The `PythonConfig` instance used to construct the Python interpreter contains a `&[u8]` referencing bytecode to be loaded as the `_frozen_importlib` and `_frozen_importlib_external` modules. The bytecode for `_frozen_importlib_external` is compiled from a **modified** version of the original `importlib._bootstrap_external` module provided by the Python interpreter. This custom module version defines a *new* `_install()` function which effectively runs `import _pyoxidizer_importer; _pyoxidizer_importer._setup(...)`.

When we initialize the Python interpreter, the `_pyoxidizer_importer` extension module is appended to the global `PyImport_Inittab` array, allowing it to be recognized as a *built-in* extension module and imported as such. In addition, the global `PyImport_FrozenModules` array is modified so the `_frozen_importlib` and `_frozen_importlib_external` modules point at our modified bytecode provided by `PythonConfig`.

When `Py_Initialize()` is called, the initialization proceeds as before. `_frozen_importlib._install()` is called to register `BuiltinImporter` and `FrozenImporter` on `sys.meta_path`. This is no different from vanilla Python. When `_frozen_importlib_external._install()` is called, our custom version/bytecode runs. It performs an `import _pyoxidizer_importer`, which is serviced by `BuiltinImporter`. Our Rust-implemented module initialization function runs and creates a module object. We then call `_setup()` on this module to complete the logical initialization.

The role of the `_setup()` function in our extension module is to add a new *meta path importer* to `sys.meta_path`. The chief goal of our importer is to support importing Python modules from memory using 0-copy.

Our extension module grabs a handle on the `&[u8]` containing modules data embedded into the binary. (See below for the format of this blob.) The in-memory data structure is parsed into a Rust collection type (basically a `HashMap<&str, (&[u8], &[u8])>`) mapping Python module names to their source and bytecode data.

The extension module defines a `PyOxidizerFinder` Python type that implements the requisite `importlib.abc.*` interfaces for providing a *meta path importer*. An instance of this type is constructed from the parsed data structure containing known Python modules. That instance is registered as the first entry on `sys.meta_path`.

When our module's `_setup()` completes, control is returned to `_frozen_importlib_external._install()`, which finishes and returns control to whatever called it.

As `Py_Initialize()` and later user code runs its course, requests are made to import non-built-in, non-frozen modules. (These requests are usually serviced by `PathFinder` via the filesystem.) The standard `sys.meta_path` traversal is performed. The Rust-implemented `PyOxidizerFinder` converts the requested Python module name to a Rust `&str` and does a lookup in a `HashMap<&str, ...>` to see if it knows about the module. Assuming the module is found, a `&[u8]` handle on that module's source or bytecode is obtained. That pointer is used to construct a Python `memoryview` object, which allows Python to access the raw bytes without a memory copy. Depending on the type, the source code is decoded to a Python `str` or the bytecode is sent to `marshal.loads()`, converted into a Python `code` object, which is then executed via the equivalent of `exec(code, module.__dict__)` to populate an empty Python module object.

In addition, `PyOxidizerFinder` indexes the built-in extension modules and frozen modules. It removes `BuiltinImporter` and `FrozenImporter` from `sys.meta_path`. When `PyOxidizerFinder` sees a request for a built-in or frozen module, it dispatches to `BuiltinImporter` or `FrozenImporter` to complete the request. The reason we do this is performance. Imports have to traverse `sys.meta_path` entries until a registered finder says it can service the request. So the more entries there are, the more overhead there is. Compounding the problem is that `BuiltinImporter` and `FrozenImporter` do a `strcmp()` against the global module arrays when trying to service an import. `PyOxidizerFinder` already has an index of module name to data. So it was not that much effort to also index built-in and frozen modules so there's a fixed, low cost for finding modules (a Rust `HashMap` key lookup).

It's worth explicitly noting that it is important for our custom code to run *before* `_frozen_importlib_external._install()` completes. This is because Python interpreter initialization relies on the fact that `.py` implemented standard library modules are available for import during initialization. For example, initializing the filesystem encoding needs to import the `encodings` module, which is provided by a `.py` file on the filesystem in standard installations.

It is impossible to provide in-memory importing of the entirety of the Python standard library without injecting custom code while ‘‘Py_Initialize()‘‘ is running. This is because `Py_Initialize()` imports modules from the filesystem. And, a subset of these standard library modules don't work as *frozen* modules. (The `FrozenImporter` doesn't set all required module attributes, leading to failures relying on missing attributes.)

13.4 Packed Modules Data

The custom meta path importer provided by this crate supports importing Python modules data (source and bytecode) from memory using 0-copy. The `PythonConfig` simply references a `&[u8]` (a generic slice over bytes data) providing modules data in a packed format.

The format of this packed data is as follows.

The first 4 bytes are a little endian u32 containing the total number of modules in this data. Let's call this value `total`.

Following is an array of length `total` with each array element being a 3-tuple of packed (no interior or exterior padding) composed of 4 little endian u32 values. These values correspond to the module name length (`name_length`), module source data length (`source_length`), module bytecode data length (`bytecode_length`), and a `flags` field to denote special behavior, respectively.

The least significant bit of the `flags` field is set if the corresponding module name is a package.

Following the lengths array is a vector of the module name strings. This vector has `total` elements. Each element is a non-NULL terminated `str` of the *name_length* specified by the corresponding entry in the lengths array. There is no padding between values. Values MUST be valid UTF-8 (they should be ASCII).

Following the names array is a vector of the module sources. This vector has `total` elements and behaves just like the names vector, except the `source_length` field from the lengths array is used.

Following the sources array is a vector of the module bytecodes. This behaves identically to the sources vector except the `bytecode_length` field from the lengths array is used.

Example (without literal integer encoding and spaces for legibility):

```
2                       # Total number of elements

[                       # Array defining 2 modules. 24 bytes total because 2 12
                        # byte members.
   (3, 0, 1024),        # 1st module has name of length 3, no source data,
                        # 1024 bytes of bytecode

   (4, 192, 4213),      # 2nd module has name length 4, 192 bytes of source
                        # data, 4213 bytes of bytecode
]

foomain                 # "foo" + "main" module names, of lengths 3 and 4,
                        # respectively.

# This is main.py.\n  # 192 bytes of source code for the "main" module.

<binary data>           # 1024 + 4213 bytes of Python bytecode data.
```

The design of the format was influenced by a handful of considerations.

Performance is a significant consideration. We want everything to be as fast as possible.

The *index* data is located at the beginning of the structure so a reader only has to read a contiguous slice of data to fully parse the index. This is in opposition to jumping around the entire backing slice to extract useful data.

x86 is little endian, so little endian integers are used so integer translation doesn't need to be performed.

It is assumed readers will want to construct an index of known modules. All module names are tightly packed together so a reader doesn't need to read small pieces of data from all over the backing slice. Similarly, it is assumed that similar data types will be accessed together. This is why source and bytecode data are packed with each other instead of packed per-module.

Everything is designed to facilitate 0-copy. So Rust need only construct a `&[u8]` into the backing slice to reference raw data.

Since Rust is the intended target, string data (module names) are not NULL terminated / C strings because Rust's `str` are not NULL terminated.

It is assumed that the module data is baked into the binary and is therefore trusted/well-defined. There's no *version header* or similar because data type mismatch should not occur. A version header should be added in the future because that's good data format design, regardless of assumptions.

There is no checksumming of the data because we don't want to incur I/O overhead to read the entire blob. It could be added as an optional feature.

Currently, the format requires the parser to perform offset math to compute slices of data. A potential area for improvement is for the index to contain start offsets and lengths so the parser can be more *dumb*. It is unlikely this has performance implications because integer math is fast and any time spent here is likely dwarfed by Python interpreter startup overhead.

Another potential area for optimization is module name encoding. Module names could definitely compress well. But use of compression will undermine 0-copy properties. Similar compression opportunities exist for source and bytecode data with similar caveats.

13.5 Packed Resources Data

The custom meta path importer provided by this crate supports loading _resource_ data via the `importlib.abc.ResourceReader` interface. Data is loaded from memory using 0-copy.

Resource file data is embedded in the binary and is represented to `PythonConfig` as a `&[u8]`.

The format of this packed data is as follows.

The first 4 bytes are a little endian u32 containing the total number of packages in the data blob. Let's call this value `package_count`.

Following are `package_count` segments that define the resources in each package. Each segment begins with a pair of little endian u32. The first integer is the length of the package name string and the 2nd is the number of resources in this package. Let's call these `package_name_length` and `resource_count`, respectively.

Following the package header is an array of `resource_count` elements. Each element is composed of 2 little endian u32 defining the resource's name length and data size, respectively.

Following this array is the index data for the next package, if there is one.

After the final package index data is the raw name of the 1st package. Following it is a vector of strings containing the resource names for that package. This pattern repeats for each package. All strings MUST be valid UTF-8. There is no NULL terminator or any other padding between values.

Following the *index* metadata is the raw resource values. Values occur in the order they were referenced in the index. There is no padding between values. Values can contain any arbitrary byte sequence.

Example (without literal integer encoding and spaces for legibility):

```
2                           # There are 2 packages total.

(3, 1)                      # Length of 1st package name is 3 and it has 1 resource.
(3, 42)                     # 1st resource has name length 3 and is 42 bytes long.

(4, 2)                      # Length of 2nd package name is 4 and it has 2 resources.
(5, 128)                    # 1st resource has name length 5 and is 128 bytes long.
(8, 1024)                   # 2nd resource has name length 8 and is 1024 bytes long.

foo                         # 1st package is named "foo"
bar                         # 1st resource name is "bar"
acme                        # 2nd package is named "acme"
hello                       # 1st resource name is "hello"
blahblah                    # 2nd resource name is "blahblah"

foo.bar raw data            # 42 bytes of raw data for "foo.bar".
acme.hello                  # 128 bytes of raw data for "acme.hello".
acme.blahblah               # 1024 bytes of raw data for "acme.blahblah"
```

Rationale for the design of this data format is similar to the reasons given for *Packed Modules Data* above.

CHAPTER 14

Technical Notes

14.1 CPython Initialization

Most code lives in `pylifecycle.c`.

Call tree with Python 3.7:

```
``Py_Initialize()``
  ``Py_InitializeEx()``
    ``_Py_InitializeFromConfig(_PyCoreConfig config)``
      ``_Py_InitializeCore(PyInterpreterState, _PyCoreConfig)``
        Sets up allocators.
        ``_Py_InitializeCore_impl(PyInterpreterState, _PyCoreConfig)``
          Does most of the initialization.
          Runtime, new interpreter state, thread state, GIL, built-in types,
          Initializes sys module and sets up sys.modules.
          Initializes builtins module.
          ``_PyImport_Init()``
            Copies ``interp->builtins`` to ``interp->builtins_copy``.
          ``_PyImportHooks_Init()``
            Sets up ``sys.meta_path``, ``sys.path_importer_cache``,
            ``sys.path_hooks`` to empty data structures.
          ``initimport()``
            ``PyImport_ImportFrozenModule("_frozen_importlib")``
            ``PyImport_AddModule("_frozen_importlib")``
            ``interp->importlib = importlib``
            ``interp->import_func = interp->builtins.__import__``
            ``PyInit__imp()``
              Initializes ``_imp`` module, which is implemented in C.
            ``sys.modules["_imp"} = imp``
            ``importlib._install(sys, _imp)``
            ``_PyImportZip_Init()``

      ``_Py_InitializeMainInterpreter(interp, _PyMainInterpreterConfig)``
```

(continues on next page)

(continued from previous page)

```
        ``_PySys_EndInit()``
          ``sys.path = XXX``
          ``sys.executable = XXX``
          ``sys.prefix = XXX``
          ``sys.base_prefix = XXX``
          ``sys.exec_prefix = XXX``
          ``sys.base_exec_prefix = XXX``
          ``sys.argv = XXX``
          ``sys.warnoptions = XXX``
          ``sys._xoptions = XXX``
          ``sys.flags = XXX``
          ``sys.dont_write_bytecode = XXX``
        ``initexternalimport()``
          ``interp->importlib._install_external_importers()``
        ``initfsencoding()``
          ``_PyCodec_Lookup(Py_FilesystemDefaultEncoding)``
            ``_PyCodecRegistry_Init()``
              ``interp->codec_search_path = []``
              ``interp->codec_search_cache = {}``
              ``interp->codec_error_registry = {}``
              # This is the first non-frozen import during startup.
              ``PyImport_ImportModuleNoBlock("encodings")``
            ``interp->codec_search_cache[codec_name]``
            ``for p in interp->codec_search_path: p[codec_name]``
        ``initsigs()``
        ``add_main_module()``
          ``PyImport_AddModule("__main__")``
        ``init_sys_streams()``
          ``PyImport_ImportModule("encodings.utf_8")``
          ``PyImport_ImportModule("encodings.latin_1")``
          ``PyImport_ImportModule("io")``
          Consults ``PYTHONIOENCODING`` and gets encoding and error mode.
          Sets up ``sys.__stdin__``, ``sys.__stdout__``, ``sys.__stderr__``.
        Sets warning options.
        Sets ``_PyRuntime.initialized``, which is what ``Py_IsInitialized()``
        returns.
        ``initsite()``
          ``PyImport_ImportModule("site")``
```

14.2 CPython Importing Mechanism

`Lib/importlib` defines importing mechanisms and is 100% Python.

`Programs/_freeze_importlib.c` is a program that takes a path to an input `.py` file and path to output `.h` file. It initializes a Python interpreter and compiles the `.py` file to marshalled bytecode. It writes out a `.h` file with an inline `const unsigned char _Py_M__importlib` array containing bytecode.

`Lib/importlib/_bootstrap_external.py` compiled to `Python/importlib_external.h` with `_Py_M__importlib_external[]`.

`Lib/importlib/_bootstrap.py` compiled to `Python/importlib.h` with `_Py_M__importlib[]`.

`Python/frozen.c` has `_PyImport_FrozenModules[]` effectively mapping `_frozen_importlib` to `importlib._bootstrap` and `_frozen_importlib_external` to `importlib._bootstrap_external`.

`initimport()` calls `PyImport_ImportFrozenModule("_frozen_importlib")`, effectively `import importlib._bootstrap`. Module import doesn't appear to have meaningful side-effects.

`importlib._bootstrap.__import__` is installed as `interp->import_func`.

C implemented `_imp` module is initialized.

`importlib._bootstrap._install(sys, _imp` is called. Calls `_setup(sys, _imp)` and adds `BuiltinImporter` and `FrozenImporter` to `sys.meta_path`.

`_setup()` defines globals `_imp` and `sys`. Populates `__name__`, `__loader__`, `__package__`, `__spec__`, `__path__`, `__file__`, `__cached__` on all `sys.modules` entries. Also loads builtins `_thread`, `_warnings`, and `_weakref`.

Later during interpreter initialization, `initexternal()` effectively calls `importlib._bootstrap._install_external_importers()`. This runs `import _frozen_importlib_external`, which is effectively `import importlib._bootstrap_external`. This module handle is aliased to `importlib._bootstrap._bootstrap_external`.

`importlib._bootstrap_external` import doesn't appear to have significant side-effects.

`importlib._bootstrap_external._install()` is called with a reference to `importlib._bootstrap`. `_setup()` is called.

`importlib._bootstrap._setup()` imports builtins `_io`, `_warnings`, `_builtins`, `marshal`. Either `posix` or `nt` imported depending on OS. Various module-level attributes set defining run-time environment. This includes `_winreg`. `SOURCE_SUFFIXES` and `EXTENSION_SUFFIXES` are updated accordingly.

`importlib._bootstrap._get_supported_file_loaders()` returns various loaders. `ExtensionFileLoader` configured from `_imp.extension_suffixes()`. `SourceFileLoader` configured from `SOURCE_SUFFIXES`. `SourcelessFileLoader` configured from `BYTECODE_SUFFIXES`.

`FileFinder.path_hook()` called with all loaders and result added to `sys.path_hooks`. `PathFinder` added to `sys.meta_path`.

14.3 `sys.modules` After Interpreter Init

Module	Type	Source
`__main__`		`add_main_module()`
`_abc`	builtin	`abc`
`_codecs`	builtin	`initfsencoding()`
`_frozen_importlib`	frozen	`initimport()`
`_frozen_importlib_external`	frozen	`initexternal()`
`_imp`	builtin	`initimport()`
`_io`	builtin	`importlib._bootstrap._setup()`
`_signal`	builtin	`initsigs()`
`_thread`	builtin	`importlib._bootstrap._setup()`
`_warnings`	builtin	`importlib._bootstrap._setup()`
`_weakref`	builtin	`importlib._bootstrap._setup()`
`_winreg`	builtin	`importlib._bootstrap._setup()`
`abc`	py	
`builtins`	builtin	`_Py_InitializeCore_impl()`
`codecs`	py	`encodings` via `initfsencoding()`
`encodings`	py	`initfsencoding()`
`encodings.aliases`	py	`encodings`
`encodings.latin_1`	py	`init_sys_streams()`
`encodings.utf_8`	py	`init_sys_streams()` + `initfsencoding()`
`io`	py	`init_sys_streams()`
`marshal`	builtin	`importlib._bootstrap._setup()`
`nt`	builtin	`importlib._bootstrap._setup()`
`posix`	builtin	`importlib._bootstrap._setup()`
`readline`	builtin	
`sys`	builtin	`_Py_InitializeCore_impl()`
`zipimport`	builtin	`initimport()`

14.4 Modules Imported by `site.py`

`_collections_abc` `_sitebuiltins` `_stat` `atexit` `genericpath` `os` `os.path` `posixpath` `rlcompleter` `site` `stat`

14.5 Random Notes

Frozen importer iterates an array looking for module names. On each item, it calls `_PyUnicode_EqualToASCIIString()`, which verifies the search name is ASCII. Performing an O(n) scan for every frozen module if there are a large number of frozen modules could contribute performance overhead. A better frozen importer would use a map/hash/dict for lookups. This //may// require CPython API breakages, as the `PyImport_FrozenModules` data structure is documented as part of the public API and its value could be updated dynamically at run-time.

`importlib._bootstrap` cannot call `import` because the global import hook isn't registered until after `initimport()`.

`importlib._bootstrap_external` is the best place to monkeypatch because of the limited run-time functionality available during `importlib._bootstrap`.

It's a bit wonky that `Py_Initialize()` will import modules from the standard library and it doesn't appear possible to disable this. If `site.py` is disabled, non-extension builtins are limited to `codecs`, `encodings`, `abc`, and whatever `encodings.*` modules are needed by `initfsencoding()` and `init_sys_streams()`.

An attempt was made to freeze the set of standard library modules loaded during initialization. However, the built-in extension importer doesn't set all of the module attributes that are expected of the modules system. The `from . import aliases` in `encodings/__init__.py` is confused without these attributes. And relative imports seemed to have issues as well. One would think it would be possible to run an embedded interpreter with all standard library modules frozen, but this doesn't work.