PyOxidizer

Release 0.13.2

Apr 16, 2021

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 Read The Docs (unreleased/latest commit, last release).

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.

1.2 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:

- Generating a boilerplate Rust source file to call into the pyembed crate to run a Python interpreter.
- Generating a working pyoxidizer.bzl *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 `OxidizedPythonInterpreterConfig` 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 Starlark 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 Python Requirements

PyOxidizer currently targets Python 3.8 or 3.9. Your Python application will need to already be compatible with 1 of these versions for it to work with PyOxidizer. See *Why is Python 3.8 Required?* for more on the minimum Python requirement.

2.2 Operating System Requirements

PyOxidizer itself is a Rust program and should theoretically be installable on any environment that Rust supports.

However, PyOxidizer needs to run Python interpreters on the machine performing build/packaging actions and the built binary needs to run a Python interpreter for the target architecture and operating system. These Python interpreters need to be built/packaged in a specific way so PyOxidizer can interact with them.

See *Available Python Distributions* for the full list of available Python distributions. The supported operating systems and architectures are:

- Linux x86_64 (glibc 2.19 or musl linked)
- Windows 8+ / Server 2012+ i686 and x86_64
- macOS 10.9+ Intel x86_64 or 11.0+ ARM

2.3 Installing

2.3.1 Installing Rust

PyOxidizer is a Rust application and requires Rust (1.46 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 1.46.0 (04488afe3 2020-08-24)
```

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.

2.3.2 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.

On macOS, you will need an Apple SDK that is at least as new as the SDK used to build the Python distribution embedded in the binary. PyOxidizer will automatically attempt to locate, validate, and use an appropriate SDK. See *Build Machine Requirements* for more.

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.

2.3.3 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.

Danger: A `cargo build` from the repository root directory will likely fail due to how some of the Rust crates are configured.

See *Using Cargo with PyOxidizer Source Checkouts* for instructions on how to invoke `cargo`.

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

```
$ pyoxidizer
PyOxidizer 0.8-pre
Gregory Szorc <gregory.szorc@gmail.com>
Build and distribute Python applications

USAGE:
    pyoxidizer [FLAGS] [SUBCOMMAND]

...
```

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

2.4 High-Level Project Lifecycle

`PyOxidizer` exposes various functionality through the interaction of `pyoxidizer` commands and configuration files.

The first step of any project is to create it. This is achieved with a `pyoxidizer init-*` command to create files required by `PyOxidizer`.

After that, various `pyoxidizer` commands can be used to evaluate configuration files and perform actions from the evaluated file. `PyOxidizer` provides functionality for building binaries, installing files into a directory tree, and running the results of build actions.

2.5 Your First PyOxidizer Project

The `pyoxidizer init-config-file` command will create a new PyOxidizer configuration file in a directory of your choosing:

```
$ pyoxidizer init-config-file 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 configuration created by `pyoxidizer init-config-file` will produce an executable that embeds Python and starts a Python REPL by default. Let's test that:

```
$ cd pyapp
$ pyoxidizer run
resolving 1 targets
resolving target exe
...
    Compiling pyapp v0.1.0 (/tmp/pyoxidizer.nv7QvpNPRgL5/pyapp)
     Finished dev [unoptimized + debuginfo] target(s) in 26.07s
```

(continues on next page)

(continued from previous page)

```
writing executable to /home/gps/src/pyapp/build/x86_64-unknown-linux-gnu/debug/exe/
↪pyapp
>>>
```

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 *The pyoxidizer Command Line Tool* to learn more about the `pyoxidizer` tool. Or read on for a preview of how to customize your application's behavior.

2.6 The `pyoxidizer.bzl` Configuration File

The most important file for a `PyOxidizer` project is the `pyoxidizer.bzl` configuration file. This is a Starlark file evaluated in a context that provides special functionality for `PyOxidizer`.

Starlark is a Python-like interpreted language and its syntax and semantics should be familiar to any Python programmer.

From a high-level, `PyOxidizer`'s configuration files define named `targets`, which are callable functions associated with a name - the *target* - that resolve to an entity. For example, a configuration file may define a `build_exe()` function which returns an object representing a standalone executable file embedding Python. The `pyoxidizer build` command can be used to evaluate just that target/function.

Target functions can call out to other target functions. For example, there may be an `install` target that creates a set of files composing a full application. Its function may evaluate the `exe` target to produce an executable file.

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

2.7 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.

The autogenerated `pyoxidizer.bzl` file created as part of running `pyoxidizer init-config-file` defines how your application is configured and built. It controls everything from what Python distribution to use, which Python packages to install, how the embedded Python interpreter is configured, and what code to run in that interpreter.

Open `pyoxidizer.bzl` in your favorite editor and find the commented lines assigning to `python_config.run_*`. Let's uncomment or add a line to match the following:

```
python_config.run_command = "import uuid; print(uuid.uuid4())"
```

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

```
$ pyoxidizer run
...
   Compiling pyapp v0.1.0 (/home/gps/src/pyapp)
    Finished dev [unoptimized + debuginfo] target(s) in 3.92s
     Running `target/debug/pyapp`
writing executable to /home/gps/src/pyapp/build/x86_64-unknown-linux-gnu/debug/exe/
↪pyapp
96f776c8-c32d-48d8-8c1c-aef8a735f535
```

It works!

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

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

Find the `exe = dist.to_python_executable(` line in the `pyoxidizer.bzl` file. Let's add a new line to `make_exe()` just below where `exe` is assigned:

```
for resource in exe.pip_install(["pyflakes==2.2.0"]):
    resource.add_location = "in-memory"
    exe.add_python_resource(resource)
```

In addition, set the `python_config.run_command` attribute to execute `pyflakes`:

```
python_config.run_command = "from pyflakes.api import main; main()"
```

Now let's try building and running the new configuration:

```
$ pyoxidizer run -- --help
...
   Compiling pyapp v0.1.0 (/home/gps/src/pyapp)
    Finished dev [unoptimized + debuginfo] target(s) in 5.49s
writing executable to /home/gps/src/pyapp/build/x86_64-unknown-linux-gnu/debug/exe/
↪pyapp
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`!

Note: `pyflakes` with no command arguments will read from stdin and will effectively hang until stdin is closed (typically via `CTRL + D`). So the `-- --help` in the above example is important, as it forces the command to produce output.

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 *The pyoxidizer Command Line Tool* to learn more about the `pyoxidizer` tool.

CHAPTER 3

The `pyoxidizer` Command Line Tool

The `pyoxidizer` command line tool is a frontend to the various functionality of `PyOxidizer`. See *Components* for more on the various components of `PyOxidizer`.

3.1 Creating New Projects with `init-config-file`

The `pyoxidizer init-config-file` command will create a new `pyoxidizer.bzl` configuration file in the target directory:

```
$ pyoxidizer init-config-file pyapp
```

This should have printed out details on what happened and what to do next.

3.2 Creating New Rust Projects with `init-rust-project`

The `pyoxidizer init-rust-project` command creates a minimal Rust project configured to build an application that runs an embedded Python interpreter from a configuration defined in a `pyoxidizer.bzl` configuration file. Run it by specifying the directory to contain the new project:

```
$ pyoxidizer init-rust-project pyapp
```

This should have printed out details on what happened and what to do next.

The explicit creation of Rust projects to use `PyOxidizer` is not required. If your produced binaries only need to perform actions configurable via `PyOxidizer` configuration files (like running some Python code), an explicit Rust project isn't required, as `PyOxidizer` can auto-generate a temporary Rust project at build time.

But if you want to supplement the behavior of the binaries built with Rust, an explicit and persisted Rust project can facilitate that. For example, you may want to run custom Rust code before, during, and after a Python interpreter runs in the process.

See *Rust Projects* for more on the composition of Rust projects.

3.3 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.

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.

3.4 Building PyObject Projects with `build`

The `pyoxidizer build` command is probably the most important and used `pyoxidizer` command. This command evaluates a `pyoxidizer.bzl` configuration file by resolving *targets* in it.

By default, the default *target* in the configuration file is resolved. However, callers can specify a list of explicit *targets* to resolve. e.g.:

```
# Resolve the default target.
$ pyoxidizer build

# Resolve the "exe" and "install" targets, in that order.
$ pyoxidizer build exe install
```

`PyOxidizer` configuration files are effectively defining a build system, hence the name *build* for the command to resolve *targets* within.

3.5 Running the Result of Building with `run`

Target functions in `PyOxidizer` configuration files return objects that may be *runnable*. For example, a *PythonExecutable* returned by a target function that defines a Python executable binary can be *run* by executing a new process.

The `pyoxidizer run` command is used to attempt to *run* an object returned by a build target. It is effectively `pyoxidizer build` followed by *running* the returned object. e.g.:

```
# Run the default target.
$ pyoxidizer run

# Run the "install" target.
$ pyoxidizer run --target install
```

3.6 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.

3.7 Inspecting Python Distributions

`PyOxidizer` uses special pre-built Python distributions to build binaries containing Python.

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.

3.8 Debugging Resource Scanning and Identification with `find-resources`

The `pyoxidizer find-resources` command can be used to scan for resources in a given source and then print information on what's found.

PyOxidizer's packaging functionality scans directories and files and classifies them as Python resources which can be operated on. See *Resource Types*. PyOxidizer's run-time importer/loader (*oxidized_importer Python Extension*) works by reading a pre-built index of known resources. This all works in contrast to how Python typically works, which is to put a bunch of files in directories and let the built-in importer/loader figure it out by dynamically probing for various files.

Because PyOxidizer has introduced structure where it doesn't exist in Python and because there are many subtle nuances with how files are classified, there can be bugs in PyOxidizer's resource scanning code.

The `pyoxidizer find-resources` command exists to facilitate debugging PyOxidizer's resource scanning code.

Simply give the command a path to a directory or Python wheel archive and it will tell you what it discovers. e.g.:

```
$ pyoxidizer find-resources dist/oxidized_importer-0.1-cp38-cp38-manylinux1_x86_64.whl
parsing dist/oxidized_importer-0.1-cp38-cp38-manylinux1_x86_64.whl as a wheel archive
PythonExtensionModule { name: oxidized_importer }
PythonPackageDistributionResource { package: oxidized-importer, version: 0.1, name:␣
↪LICENSE }
PythonPackageDistributionResource { package: oxidized-importer, version: 0.1, name:␣
↪WHEEL }
PythonPackageDistributionResource { package: oxidized-importer, version: 0.1, name:␣
↪top_level.txt }
PythonPackageDistributionResource { package: oxidized-importer, version: 0.1, name:␣
↪METADATA }
PythonPackageDistributionResource { package: oxidized-importer, version: 0.1, name:␣
↪RECORD }
```

Or give it the path to a `site-packages` directory:

```
$ pyoxidizer find-resources ~/.pyenv/versions/3.8.6/lib/python3.8/site-packages
...
```

This command needs to use a Python distribution so it knows what file extensions correspond to Python extensions, etc. By default, it will download one of the *built-in distributions* that is compatible with the current machine and use that. You can specify a `--distributions-dir` to use to cache downloaded distributions:

```
$ pyoxidizer find-resources --distributions-dir distributions /usr/lib/python3.8
...
```

CHAPTER 4

Configuration Files

PyOxidizer uses Starlark files to configure run-time behavior.

Starlark is a dialect of Python intended to be used as a configuration language and the syntax should be familiar to any Python programmer.

This documentation section contains both a high-level overview of the configuration files and their semantics as well as low-level documentation for every type and function in the Starlark dialect.

4.1 Automatic File Location Strategy

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

If the `OUT_DIR` environment variable is set (we're building from the context of a Rust project), the ancestor directories will be searched for a `pyoxidizer.bzl` file and the first one found will be used.

Otherwise, `PyOxidizer` will look for a `pyoxidizer.bzl` file starting in either the current working directory or from the directory containing the `pyembed` crate and then will traverse ancestor directories until a file is found.

If no configuration file is found, an error occurs.

4.2 Concepts

4.2.1 Processing

A configuration file is evaluated in a custom Starlark *dialect* which provides primitives used by PyOxidizer. This dialect provides some well-defined global variables (defined in UPPERCASE) as well as some types and functions that can be constructed and called. See *Global Symbols* for a full list of what's available to the Starlark environment.

Since Starlark is effectively a subset of Python, executing a `PyOxidizer` configuration file is effectively running a sandboxed Python script. It is conceptually similar to running `python setup.py` to build a Python package.

As functions within the Starlark environment are called, `PyOxidizer` will perform actions as described by those functions.

4.2.2 Targets

`PyOxidizer` configuration files are composed of functions registered as named *targets*. You define a function that does something then register it as a target by calling the *register_target()* global function provided by our Starlark dialect. e.g.:

```
def get_python_distribution():
    return default_python_distribution()

register_target("dist", get_python_distribution)
```

When a configuration file is evaluated, `PyOxidizer` attempts to *resolve* an ordered list of *targets* This list of targets is either specified by the end-user or is derived from the configuration file. The first `register_target()` target or the last `register_target()` call passing `default=True` is the default target.

When evaluated in *Rust build script mode* (typically via `pyoxidizer run-build-script`), the default target will be the one specified by the last `register_target()` call passing `default_build_script=True`, or the default target if no target defines itself as the default build script target.

`PyOxidizer` calls the registered target functions in order to *resolve* the requested set of targets.

Target functions can depend on other targets and dependent target functions will automatically be called and have their return value passed as an argument to the target function depending on it. See *register_target()* for more.

The value returned by a target function is special. Some types defined by our Starlark dialect have special *build* or *run* behavior associated with them. If you run `pyoxidizer build` or `pyoxidizer run` against a target that returns one of these types, that behavior will be performed.

For example, if you return a *PythonExecutable*, the *build* behavior is to produce that executable file and the *run* behavior is to run that built executable.

See *Types with Target Behavior* for the full list of types with registered target behaviors.

4.2.3 Python Distributions Provide Python

The *PythonDistribution* Starlark type defines a Python distribution. A Python distribution is an entity which contains a Python interpreter, Python standard library, and which PyOxidizer knows how to consume and integrate into a new binary.

`PythonDistribution` instances are arguably the most important type in configuration files because without them you can't perform Python packaging actions or construct binaries with Python embedded.

Instances of `PythonDistribution` are typically constructed from *default_python_distribution()* and are registered as their own target, since multiple targets may want to reference the distribution instance:

```
def make_dist():
   return default_python_distribution()

register_target("dist", make_dist)
```

4.2.4 Python Executables Run Python

The *PythonExecutable* Starlark type defines an executable file embedding Python. Instances of this type are used to build an executable file (and possibly other files needed by it) that contains an embedded Python interpreter and other resources required by it.

Instances of `PythonExecutable` are derived from a `PythonDistribution` instance via the *PythonDistribution.to_python_executable()* method. There is typically a standalone function/target in config files for doing this.

4.2.5 Python Resources

At run-time, Python interpreters need to consult *resources* like Python module source and bytecode as well as resource/data files. We refer to all of these as *Python Resources*.

Configuration files represent *Python Resources* via the following types:

- *PythonModuleSource*
- *PythonPackageResource*
- *PythonPackageDistributionResource*
- *PythonExtensionModule*

4.2.6 Specifying Resource Locations

Various functionality relates to the concept of a *resource location*, or where a resource should be loaded from at run-time. See *Managing How Resources are Added* for more.

Resource locations are represented as strings in Starlark. The mapping of strings to resource locations is as follows:

`in-memory` Load the resource from memory.

`filesystem-relative:<prefix>` Install and load the resource from a filesystem relative path to the build binary. e.g. `filesystem-relative:lib` will place resources in the `lib/` directory next to the build binary.

4.3 Resource Attributes Influencing Adding

Individual Starlark values representing resources expose various attributes prefixed with `add_` which influence what happens when that resource is added to a resource collector. These attributes are derived from the *PythonPackagingPolicy* attached to the entity creating the resource. But they can be modified by Starlark code before the resource is added to a collection.

The following sections describe each attribute that influences how the resource is added to a collection.

4.3.1 `add_include`

This `bool` attribute defines a yes/no filter for whether to actually add this resource to a collection. If a resource with `.add_include = False` is added to a collection, that add is processed as a no-op and no change is made.

4.3.2 add_location

This `string` attributes defines the primary location this resource should be added to and loaded from at run-time.

It can be set to the following values:

in-memory The resource should be loaded from memory.

For Python modules and resource files, the module is loaded from memory using 0-copy by the custom module importer.

For Python extension modules, the extension module may be statically linked into the built binary or loaded as a shared library from memory (the latter is not supported on all platforms).

filesystem-relative:<prefix> The resource is materialized on the filesystem relative to the built entity and loaded from the filesystem at run-time.

`<prefix>` here is a directory prefix to place the resource in. `.` (e.g. `filesystem-relative:.`) can be used to denote the same directory as the built entity.

4.3.3 add_location_fallback

This `string` or `None` value attribute is equivalent to `add_location` except it only comes into play if the location specified by `add_location` could not be satisfied.

Some resources (namely Python extension modules) cannot exist in all locations. Setting this attribute to a different location gives more flexibility for packaging resources with location constraints.

4.3.4 add_source

This `bool` attribute defines whether to add source code for a Python module.

For Python modules, typically only bytecode is required at run-time. For some applications, the presence of source code doesn't provide sufficient value or isn't desired since the application developer may want to obfuscate the source code. Setting this attribute to `False` prevents Python module source code from being added.

4.3.5 add_bytecode_optimization_level_zero

This `bool` attributes defines whether to add Python bytecode for optimization level 0 (the default optimization level).

If `True`, Python source code will be compiled to bytecode at build time.

The default value is whatever `PythonPackagingPolicy.bytecode_optimize_level_zero` is set to.

4.3.6 add_bytecode_optimization_level_one

This `bool` attributes defines whether to add Python bytecode for optimization level 1.

The default value is whatever `PythonPackagingPolicy.bytecode_optimize_level_one` is set to.

4.3.7 add_bytecode_optimization_level_two

This `bool` attributes defines whether to add Python bytecode for optimization level 2.

The default value is whatever `PythonPackagingPolicy.bytecode_optimize_level_two` is set to.

4.4 Global Symbols

This document lists every single global type, variable, and function available in PyOxidizer's Starlark execution environment.

The Starlark environment contains symbols from the following:

- Starlark built-ins
- *Tugger's Starlark Dialect*
- PyOxidizer's Dialect (documented below)

4.4.1 Global Types

PyOxidizer's Starlark dialect defines the following custom types:

File Represents a filesystem path and content.

FileContent Represents the content of a file on the filesystem.

(Unlike *File*, this does not track the filename internally.)

FileManifest Represents a mapping of filenames to file content.

PythonDistribution Represents an implementation of Python.

Used for embedding into binaries and running Python code.

PythonEmbeddedResources Represents resources made available to a Python interpreter.

PythonExecutable Represents an executable file containing a Python interpreter.

PythonExtensionModule Represents a compiled Python extension module.

PythonInterpreterConfig Represents the configuration of a Python interpreter.

PythonPackageDistributionResource Represents a file containing Python package distribution metadata.

PythonPackageResource Represents a non-module *resource* data file.

PythonPackagingPolicy Represents a policy controlling how Python resources are added to a binary.

PythonModuleSource Represents a `.py` file containing Python source code.

4.4.2 Global Constants

The Starlark execution environment defines various variables in the global scope which are intended to be used as read-only constants. The following sections describe these variables.

BUILD_TARGET_TRIPLE

The string Rust target triple that we're currently building for. Will be a value like `x86_64-unknown-linux-gnu` or `x86_64-pc-windows-msvc`. Run `rustup target list` to see a list of targets.

CONFIG_PATH

The string path to the configuration file currently being evaluated.

CONTEXT

Holds build context. This is an internal variable and accessing it will not provide any value.

CWD

The current working directory. Also the directory containing the active configuration file.

4.4.3 Global Functions

PyOxidizer's Starlark dialect defines the following global functions:

default_python_distribution() Obtain the default *PythonDistribution* for the active build configuration.

register_target() Register a named *target* that can be built.

resolve_target() Build/resolve a specific named *target*.

resolve_targets() Triggers resolution of requested build *targets*.

set_build_path() Set the filesystem path to use for writing files during evaluation.

4.4.4 Types with Target Behavior

As described in *Targets*, a function registered as a named target can return a type that has special *build* or *run* behavior.

The following types have special behavior registered:

FileManifest Build behavior is to materialize all files in the file manifest.

Run behavior is to run the last added *PythonExecutable* if available, falling back to an executable file installed by the manifest if there is exactly 1 executable file.

PythonEmbeddedResources Build behavior is to write out files this type represents.

There is no run behavior.

PythonExecutable Build behavior is to build the executable file.

Run behavior is to run that built executable.

4.5 Functions for Manipulating Global State

4.5.1 `set_build_path()`

Configure the directory where build artifacts will be written.

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

If a relative path is passed, it is interpreted as relative to the directory containing the configuration file.

The default value is `$CWD/build`.

Important: This needs to be called before functionality that utilizes the build path, otherwise the default value will be used.

4.6 Functions for Managing Targets

4.6.1 register_target()

Registers a named target that can be resolved by the configuration file.

A target consists of a string name, callable function, and an optional list of targets it depends on.

The callable may return one of the types defined by this Starlark dialect to facilitate additional behavior, such as how to build and run it.

Arguments:

name (`string`) The name of the target being register.

fn (`function`) A function to call when the target is resolved.

depends (`list` of `string` or `None`) List of target strings this target depends on. If specified, each dependency will be evaluated in order and its returned value (possibly cached from prior evaluation) will be passed as a positional argument to this target's callable.

default (`bool`) Indicates whether this should be the default target to evaluate. The last registered target setting this to `True` will be the default. If no target sets this to `True`, the first registered target is the default.

default_build_script (`bool`) indicates whether this should be the default target to evaluate when run from the context of a Rust build script (e.g. from `pyoxidizer run-build-script`. It has the same semantics as `default`.

Note: It would be easier for target functions to call `resolve_target()` within their implementation. However, Starlark doesn't allow recursive function calls. So invocation of target callables must be handled specially to avoid this recursion.

4.6.2 resolve_target()

Triggers resolution of a requested build target.

This function resolves a target registered with `register_target()` by calling the target's registered function or returning the previously resolved value from calling it.

This function should be used in cases where 1 target depends on the resolved value of another target. For example, a target to create a `FileManifest` may wish to add a `PythonExecutable` that was resolved from another target.

4.6.3 resolve_targets()

Triggers resolution of requested build targets.

This is usually the last meaningful line in a config file. It triggers the building of targets which have been requested to resolve by whatever is invoking the config file.

4.7 Extensions to Tugger's Starlark Dialect

PyOxidizer extends *Tugger's Starlark dialect* with addition methods.

4.7.1 FileManifest.add_python_resource()

This method adds a Python resource to a `FileManifest` instance in a specified directory prefix.

Arguments:

prefix (`string`) Directory prefix to add resource to.

value (various) A *Python resource* instance to add. e.g. *PythonModuleSource* or *PythonPackageResource*.

This method can be used to place the Python resources derived from another type or action in the filesystem next to an application binary.

4.7.2 FileManifest.add_python_resources()

This method adds an iterable of Python resources to a `FileManifest` instance in a specified directory prefix. This is effectively a wrapper for `for value in values: self.add_python_resource(prefix, value)`.

For example, to place the Python distribution's standard library Python source modules in a directory named `lib`:

```
m = FileManifest()
dist = default_python_distribution()
for resource in dist.python_resources():
    if type(resource) == "PythonModuleSource":
        m.add_python_resource("lib", resource)
```

4.8 File

This type represents a concrete file in an abstract filesystem. The file has a path and content.

Instances can be constructed by calling methods that emit resources with a *PythonPackagingPolicy* having *file_scanner_emit_files* set to `True`.

4.8.1 Attributes

The following sections describe the attributes available on each instance.

path

(`string`)

The filesystem path represented. Typically relative. Doesn't have to correspond to a valid, existing file on the filesystem.

is_executable

(`bool`)

Whether the file is executable.

add_*

(various)

See *Resource Attributes Influencing Adding*.

4.9 PythonDistribution

The `PythonDistribution` type defines a Python distribution. A Python distribution is an entity that defines an implementation of Python. This entity can be used to create a binary embedding or running Python and can be used to execute Python code.

4.9.1 Constructors

Instances of `PythonDistribution` can be constructed via a constructor function or via *default_python_distribution()*.

default_python_distribution()

Resolves the default `PythonDistribution`.

The following named arguments are accepted:

flavor (`string`) Denotes the *distribution* flavor. See the section below on allowed values.

Defaults to `standalone`.

build_target (`string`) Denotes the machine target triple that we're building for.

Defaults to the value of the `BUILD_TARGET` global constant.

python_version (`string`) `X.Y` *major.minor* string denoting the Python release version to use.

Supported values are `3.8` and `3.9`.

Defaults to `3.9`.

`flavor` is a string denoting the distribution *flavor*. Values can be one of the following:

standalone A distribution produced by the `python-build-standalone` project. The distribution may be statically or dynamically linked, depending on the `build_target` and availability. This option effectively chooses the best available `standalone_dynamic` or `standalone_static` option.

This option is effectively `standalone_dynamic` for all targets except musl libc, where it is effectively `standalone_static`.

standalone_dynamic This is like `standalone` but guarantees the distribution is dynamically linked against various system libraries, notably libc. Despite the dependence on system libraries, binaries built with these distributions can generally be run in most environments.

This flavor is available for all supported targets except musl libc.

standalone_static This is like `standalone` but guarantees the distribution is statically linked and has minimal - possibly none - dependencies on system libraries.

On Windows, the Python distribution does not export Python's symbols, meaning that it is impossible to load dynamically linked Python extensions with it.

On musl libc, statically linked distributions do not support loading extension modules existing as shared libraries.

This flavor is only available for Windows and musl libc targets.

Note: The *static* versus *dynamic* terminology refers to the linking of the overall distribution, not `libpython` or the final produced binaries.

The `pyoxidizer` binary has a set of known distributions built-in which are automatically available and used by this function. Typically you don't need to build your own distribution or change the distribution manually.

PythonDistribution()

Construct a `PythonDistribution` from arguments.

The following arguments are accepted:

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.

flavor (`string`) The distribution flavor. Must be `standalone`.

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.

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

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

Examples:

```
linux = PythonDistribution(
    sha256="11a53f5755773f91111a04f6070a6bc00518a0e8e64d90f58584abf02ca79081",
    local_path="/var/python-distributions/cpython-linux64.tar.zst"
)

macos = PythonDistribution(
     sha256="b46a861c05cb74b5b668d2ce44dcb65a449b9fef98ba5d9ec6ff6937829d5eec",
     url="https://github.com/indygreg/python-build-standalone/releases/download/
↪20190505/cpython-3.7.3-macos-20190506T0054.tar.zst"
)
```

4.9.2 Methods

PythonDistribution.python_resources()

Returns a `list` of objects representing Python resources in this distribution. Returned values can be *PythonModuleSource*, *PythonExtensionModule*, *PythonPackageResource*, etc.

There may be multiple *PythonExtensionModule* with the same name.

PythonDistribution.make_python_interpreter_config()

Obtain a *PythonInterpreterConfig* derived from the distribution.

The interpreter configuration automatically uses settings appropriate for the distribution.

PythonDistribution.make_python_packaging_policy()

Obtain a *PythonPackagingPolicy* derived from the distribution.

The policy automatically uses settings globally appropriate for the distribution.

PythonDistribution.to_python_executable()

This method constructs a *PythonExecutable* instance. It essentially says *build an executable embedding Python from this distribution*.

The accepted arguments are:

name (`string`) The name of the application being built. This will be used to construct the default filename of the executable.

packaging_policy (`PythonPackagingPolicy`) The packaging policy to apply to the executable builder.

This influences how Python resources from the distribution are added. It also influences future resource adds to the executable.

config (`PythonInterpreterConfig`) The default configuration of the embedded Python interpreter.

Default is what *PythonDistribution.make_python_interpreter_config()* returns.

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.

4.10 PythonEmbeddedResources

The `PythonEmbeddedResources` type represents resources made available to a Python interpreter. The resources tracked by this type are consumed by the `pyembed` crate at build and run time. The tracked resources include:

- Python module source and bytecode
- Python package resources
- Shared library dependencies

While the type's name has *embedded* in it, resources referred to by this type may or may not actually be *embedded* in a Python binary or loaded directly from the binary. Rather, the term *embedded* comes from the fact that the data structure describing the resources is typically *embedded* in the binary or made available to an *embedded* Python interpreter.

Instances of this type are constructed by transforming a type representing a Python binary. e.g. *PythonExecutable.to_embedded_resources()*.

If this type is returned by a target function, its build action will write out files that represent the various resources encapsulated by this type. There is no run action associated with this type.

4.11 PythonExecutable

The `PythonExecutable` type represents an executable file containing the Python interpreter, Python resources to make available to the interpreter, and a default run-time configuration for that interpreter.

Instances are constructed from *PythonDistribution* instances using *PythonDistribution.to_python_executable()*.

4.11.1 Attributes

The following sections describe the attributes available on each instance.

PythonExecutable.packed_resources_load_mode

(string)

Defines how the *packed Python resources data* (see *Python Packed Resources*) is written and loaded at run-time by the embedded Python interpreter.

The following values/patterns can be defined:

none No resources data will be serialized or loaded at run-time. (Use this if you are using Python's filesystem based module importer and don't want to use PyOxidizer's custom importer.)

embedded:<filename> The packed resources data will be embedded in the binary and loaded from a memory address at run-time.

`filename` denotes the path of the on-disk file used at build time. This file is written to the *artifacts* directory that PyOxidizer writes required build files to.

binary-relative-memory-mapped:<filename> The packed resources data will be written to a file relative to the built binary and loaded from there at run-time using memory mapped I/O.

The default is `embedded:packed-resources`.

PythonExecutable.tcl_files_path

(Optional[string])

Defines a directory relative to that of the built executable in which to install tcl/tk files.

If set to a value, tcl/tk files present in the Python distribution being used will be installed next to the build executable and the embedded Python interpreter will automatically set the `TCL_LIBRARY` environment variable to load tcl files from this directory.

If `None` (the default), no tcl/tk files will be installed.

PythonExecutable.windows_runtime_dlls_mode

(string)

Controls how Windows runtime DLLs should be managed when building the binary.

Windows binaries often have a dependency on various runtime DLLs, such as `vcruntime140.dll`. The built executable will need access to these DLLs or it won't work.

This setting controls whether to install required Windows runtime DLLs next to the built binary at build time. For example, if you are producing a `myapp.exe`, this setting can automatically install a `vcruntime140.dll` next to that binary.

The following values are recognized:

never Never install Windows runtime DLLs.

when-present Install Windows runtime DLLs when they can be located. Do nothing if they can't be found.

always Install Windows runtime DLLs and fail if they can't be located.

This setting is ignored when the built binary does not have a dependency on Windows runtime DLLs.

See *Distribution Considerations for Windows* for more on runtime DLL requirements.

PythonExecutable.windows_subsystem

(`string`)

Controls the value to use for the Rust `#![windows_subsystem = "..."]` attribute added to the autogenerated Rust program to build the executable.

This attribute only has meaning on Windows. It effectively controls the value passed to the linker's `/SUBSYSTEM` flag.

Rust only supports certain values but PyOxidizer does not impose limitations on what values are used. Common values include:

console Win32 character-mode application. A console window will be opened when the application runs.

This value is suitable for command-line executables.

windows Application does not require a console and may provide its own windows.

This value is suitable for GUI applications that do not wish to launch a console window on start.

Default is `console`.

4.11.2 Methods

PythonExecutable.make_python_module_source()

This method creates a *PythonModuleSource* instance suitable for use with the executable being built.

Arguments are as follows:

name (string) The name of the Python module. This is the fully qualified module name. e.g. `foo` or `foo.bar`.

source (string) Python source code comprising the module.

is_package (bool) Whether the Python module is also a package. (e.g. the equivalent of a `__init__.py` file or a module without a `.` in its name.

PythonExecutable.pip_download()

This method runs `pip download <args>` with settings appropriate to target the executable being built.

This always uses `--only-binary=:all:`, forcing pip to only download wheel based packages.

This method accepts the following arguments:

args (`list` of `string`) Command line arguments to pass to `pip download`. Arguments will be added after default arguments added internally.

Returns a `list` of objects representing Python resources collected from wheels obtained via `pip download`.

PythonExecutable.pip_install()

This method runs `pip install <args>` with settings appropriate to target the executable being built.

args List of strings defining raw process arguments to pass to `pip install`.

extra_envs Optional dict of string key-value pairs constituting extra environment variables to set in the invoked `pip` process.

Returns a `list` of objects representing Python resources installed as part of the operation. The types of these objects can be *PythonModuleSource*, *PythonPackageResource*, etc.

The returned resources are typically added to a `FileManifest` or `PythonExecutable` to make them available to a packaged application.

PythonExecutable.read_package_root()

This method discovers resources from a directory on the filesystem.

The specified directory will be scanned for resource files. However, only specific named *packages* will be found. 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 read.

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 arguments:

path (string) The filesystem path to the directory to scan.

packages (list 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`.

Returns a `list` of objects representing Python resources found in the virtualenv. The types of these objects can be `PythonModuleSource`, `PythonPackageResource`, etc.

The returned resources are typically added to a `FileManifest` or `PythonExecutable` to make them available to a packaged application.

PythonExecutable.read_virtualenv()

This method attempts to read Python resources from an already built virtualenv.

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.

It accepts the following arguments:

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.

Returns a `list` of objects representing Python resources found in the virtualenv. The types of these objects can be `PythonModuleSource`, `PythonPackageResource`, etc.

The returned resources are typically added to a `FileManifest` or `PythonExecutable` to make them available to a packaged application.

PythonExecutable.setup_py_install()

This method runs `python setup.py install` against a package at the specified path.

It accepts the following arguments:

package_path String filesystem path to directory containing a `setup.py` to invoke.

extra_envs={} Optional dict of string key-value pairs constituting extra environment variables to set in the invoked `python` process.

extra_global_arguments=[] Optional list of strings of extra command line arguments to pass to `python setup.py`. These will be added before the `install` argument.

Returns a `list` of objects representing Python resources installed as part of the operation. The types of these objects can be `PythonModuleSource`, `PythonPackageResource`, etc.

The returned resources are typically added to a `FileManifest` or `PythonExecutable` to make them available to a packaged application.

PythonExecutable.add_python_resource()

This method registers a Python resource of various types with the instance.

It accepts a `resource` argument which can be a `PythonModuleSource`, `PythonPackageResource`, or `PythonExtensionModule` and registers that resource with this instance.

The following arguments are accepted:

resource The resource to add to the embedded Python environment.

This method is a glorified proxy to the various `add_python_*` methods. Unlike those methods, this one accepts all types that are known Python resources.

PythonExecutable.add_python_resources()

This method registers an iterable of Python resources of various types. This method is identical to *PythonExecutable.add_python_resource()* except the argument is an iterable of resources. All other arguments are identical.

PythonExecutable.filter_from_files()

This method filters all embedded resources (source modules, bytecode modules, and resource names) currently present on the instance through a set of resource names resolved from files.

This method accepts the following arguments:

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 argument must be the same format as documented by the `files` argument.

All defined files are first read and the resource names encountered are unioned into a set. This set is then used to filter entities currently registered with the instance.

PythonExecutable.to_embedded_resources()

Obtains a *PythonEmbeddedResources* instance representing resources to be made available to the Python interpreter.

See the *PythonEmbeddedResources* type documentation for more.

PythonExecutable.to_file_manifest()

This method transforms the `PythonExecutable` instance to a `FileManifest`. The `FileManifest` is populated with the build executable and any file-based resources that are registered with the resource collector. A `libpython` shared library will also be present depending on build settings.

This method accepts the following arguments:

prefix (`string`) The directory prefix of files in the `FileManifest`. Use `.` to denote no prefix.

PythonExecutable.to_wix_bundle_builder()

This method transforms the `PythonExecutable` instance into a *WiXBundleBuilder* instance. The returned value can be used to generate a Windows `.exe` installer. This installer will install the Visual C++ Redistributable as well as an MSI for the build application.

This method accepts the following arguments:

id_prefix See *Constructors* for usage.

product_name See *Constructors* for usage.

product_version See *Constructors* for usage.

product_manufacturer See *Constructors* for usage.

msi_builder_callback (`function`) A callable function that can be used to modify the *WiXMSIBuilder* constructed for the application.

The function will receive the *WiXMSIBuilder* as its single argument. The return value is ignored.

The returned value can be further customized before it is built. See *WiXBundleBuilder* type documentation for more.

Important: *PythonExecutable.windows_runtime_dlls_mode* can result in DLLs being installed next to the binary in addition to being installed as part of the installer. When using this method, you probably want to set `.windows_runtime_dlls_mode = "never"` to prevent the redundant installation.

PythonExecutable.to_wix_msi_builder()

This method transforms the `PythonExecutable` instance into a *WiXMSIBuilder* instance. The returned value can be used to generate a Windows MSI installer.

This method accepts the following arguments:

id_prefix See *Constructors* for usage.

product_name See *Constructors* for usage.

product_version See *Constructors* for usage.

product_manufacturer See *Constructors* for usage.

The MSI installer configuration can be customized. See the *WiXMSIBuilder* type documentation for more.

The MSI installer will **not** materialize the Visual C++ Runtime DLL(s).

4.12 PythonExtensionModule

This type represents a compiled Python extension module.

4.12.1 Attributes

The following sections describe the attributes available on each instance.

name

(`string`)

Unique name of the module being provided.

is_stdlib

(`bool`)

Whether this module is part of the Python standard library (part of the Python distribution).

add_*

(various)

See *Resource Attributes Influencing Adding*.

4.13 PythonInterpreterConfig

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

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

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

Danger: Some of the settings exposed by Python's initialization APIs are extremely low level and brittle. Various combinations can cause the process to crash/exit ungracefully. Be very cautious when setting these low-level settings.

4.13.1 Constructors

Instances are constructed by calling *PythonDistribution.make_python_interpreter_config()*.

4.13.2 Attributes

The `PythonInterpreterConfig` state is managed via attributes.

There are a ton of attributes and most attributes are not relevant to most applications. The bulk of the attributes exist to give full control over Python interpreter initialization.

Attributes For Controlling `pyembed` Features

This section documents attributes for controlling features provided by the `pyembed` Rust crate, which manages the embedded Python interpreter at run-time.

These attributes provide features and level of control over embedded Python interpreters beyond what is possible with Python's initialization C API.

`allocator_backend`

(`string`)

Configures a custom memory allocator to be used by Python.

Accepted values are:

`default` Let Python choose how to configure the allocator.

This will likely use the `malloc()`, `free()`, etc functions linked to the binary.

`jemalloc` Use the jemalloc allocator.

(Not available on Windows.)

`mimalloc` Use the mimalloc allocator (https://github.com/microsoft/mimalloc).

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

`snmalloc` Use the snmalloc allocator (https://github.com/microsoft/snmalloc).

The `jemalloc`, `mimalloc`, and `snmalloc` allocators require the presence of additional Rust crates. A run-time error will occur if these allocators are configured but the binary was built without these crates. (This should not occur when using `pyoxidizer` to build the binary.)

When a custom allocator is configured, the autogenerated Rust crate used to build the binary will configure the Rust global allocator (`#[global_allocator] attribute`) to use the specified allocator.

Important: The `rust` allocator is not recommended because it introduces performance overhead. But it may help with debugging in some situations.

Note: Both `mimalloc` and `snmalloc` require the `cmake` build tool to compile code as part of their build process. If this tool is not available in the build environment, you will encounter a build error with a message similar to `failed to execute command: The system cannot find the file specified. (os error 2) is `cmake` not installed?`.

The workaround is to install cmake or use a different allocator.

Note: `snmalloc` only supports targeting to macOS 10.14 or newer. You will likely see build errors when building a binary targeting macOS 10.13 or older.

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

allocator_raw

(`bool`)

Controls whether to install a custom allocator (defined by `allocator_backend`) into Python's *raw* allocator domain (`PYMEM_DOMAIN_RAW` in Python C API speak).

Setting this to `True` will replace the system allocator (e.g. `malloc()`, `free()`) for this domain.

A value of `True` only has an effect if `allocator_backend` is some value other than `default`.

Defaults to `True`.

allocator_mem

(`bool`)

Controls whether to install a custom allocator (defined by `allocator_backend`) into Python's *mem* allocator domain (`PYMEM_DOMAIN_MEM` in Python C API speak).

Setting this to `True` will replace `pymalloc` as the allocator for this domain.

A value of `True` only has an effect if `allocator_backend` is some value other than `default`.

Defaults to `False`.

allocator_obj

(`bool`)

Controls whether to install a custom allocator (defined by `allocator_backend`) into Python's *obj* allocator domain (`PYMEM_DOMAIN_OBJ` in Python C API speak).

Setting this to `True` will replace `pymalloc` as the allocator for this domain.

A value of `True` only has an effect if `allocator_backend` is some value other than `default`.

Defaults to `False`.

allocator_pymalloc_arena

(bool)

Controls whether to install a custom allocator (defined by `allocator_backend`) into Python's `pymalloc` to be used as its arena allocator.

The `pymalloc` allocator is used by Python by default and will use the system's allocator functions (`malloc()`, `VirtualAlloc()`, etc) by default.

Setting this to `True` will have no effect if `pymalloc` is not being used (the `allocator_mem` and `allocator_obj` settings are `True` and have replaced `pymalloc` as the allocator backend for these domains).

A value of `True` only has an effect if `allocator_backend` is some value other than `default`.

Defaults to `False`.

allocator_debug

(bool)

Whether to enable debug hooks for Python's memory allocators.

Enabling debug hooks enables debugging of memory-related issues in the Python interpreter. This setting effectively controls whether to call PyMem_SetupDebugHooks() during interpreter initialization. See the linked documentation for more.

Defaults to `False`.

oxidized_importer

(bool)

Whether to install the `oxidized_importer` meta path importer (*oxidized_importer Python Extension*) on `sys.meta_path` during interpreter initialization.

If *filesystem_importer* is also `True`, the importer's *path_hook* method is appended to `sys.path_hooks` at the end of interpreter initialization.

Defaults to `True`.

filesystem_importer

(bool)

Whether to install the standard library path-based importer for loading Python modules from the filesystem.

If *oxidized_importer* is also `True`, the *path_hook* method of the `oxidized_importer` meta path importer (*oxidized_importer Python Extension*) on `sys.meta_path` is appended to `sys.path_hooks` at the end of interpreter initialization.

If not enabled, Python modules will not be loaded from the filesystem (via `sys.path` discovery): only modules indexed by `oxidized_importer` will be loadable.

The filesystem importer is enabled automatically if *module_search_paths* is non-empty.

argvb

(bool)

Whether to expose a sys.argvb attribute containing bytes versions of process arguments.

On platforms where the process receives char * arguments, Python normalizes these values to unicode and makes them available via sys.argv. On platforms where the process receives wchar_t * arguments, Python may interpret the bytes as a certain encoding. This encoding normalization can be lossy.

Enabling this feature will give Python applications access to the raw bytes values of arguments that are actually used. The single or double width bytes nature of the data is preserved.

Unlike sys.argv which may chomp off leading arguments depending on the Python execution mode, sys.argvb has all the arguments used to initialize the process. The first argument is always the executable.

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.

terminfo_resolution

(string)

Defines how the terminal information database (terminfo) should be configured.

See *Terminfo Database* for more about terminal databases.

Accepted values are:

dynamic 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).

none 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.

static:<path> Indicates that a static path should be used for the path to the `terminfo` database.

This values 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.

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

write_modules_directory_env

(`string` or `None`)

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.

Attributes From `PyPreConfig`

Attributes in this section correspond to fields of the PyPreConfig C struct used to initialize the Python interpreter.

config_profile

(`string`)

This attribute controls which set of default values to use for attributes that aren't explicitly defined. It effectively controls which C API to use to initialize the `PyPreConfig` instance.

Accepted values are:

isolated Use the isolated configuration.

This configuration is appropriate for applications existing in isolation and not behaving like `python` executables.

python Use the Python configuration.

This configuration is appropriate for applications attempting to behave like a `python` executable would.

allocator

(`string` or `None`)

Controls the value of PyPreConfig.allocator.

Accepted values are:

None Use the default.

not-set `PYMEM_ALLOCATOR_NOT_SET`

default `PYMEM_ALLOCATOR_DEFAULT`

debug `PYMEM_ALLOCATOR_DEBUG`

malloc `PYMEM_ALLOCATOR_MALLOC`

malloc-debug PYMEM_ALLOCATOR_MALLOC_DEBUG

py-malloc PYMEM_ALLOCATOR_PYMALLOC

py-malloc-debug PYMEM_ALLOCATOR_PYMALLOC_DEBUG

configure_locale

(bool or None)

Controls the value of PyPreConfig.configure_locale.

coerce_c_locale

(string or None)

Controls the value of PyPreConfig.coerce_c_locale.

Accepted values are:

LC_CTYPE Read LC_CTYPE

C Coerce the C locale.

coerce_c_locale_warn

(bool or None)

Controls the value of PyPreConfig.coerce_c_locale_warn.

development_mode

(bool or None)

Controls the value of PyPreConfig.development_mode.

isolated

(bool or None)

Controls the value of PyPreConfig.isolated.

legacy_windows_fs_encoding

(bool or None)

Controls the value of PyPreConfig.legacy_windows_fs_encoding.

parse_argv

(bool or None)

Controls the value of PyPreConfig.parse_argv.

use_environment

(bool or None)

Controls the value of PyPreConfig.use_environment.

utf8_mode

(bool or None)

Controls the value of PyPreConfig.utf8_mode.

Attributes From PyConfig

Attributes in this section correspond to fields of the PyConfig C struct used to initialize the Python interpreter.

base_exec_prefix

(string or None)

Controls the value of PyConfig.base_exec_prefix.

base_executable

(string or None)

Controls the value of PyConfig.base_exectuable.

base_prefix

(string or None)

Controls the value of PyConfig.base_prefix.

buffered_stdio

(bool or None)

Controls the value of PyConfig.buffered_stdio.

bytes_warning

(string or None)

Controls the value of PyConfig.bytes_warning.

Accepted values are:

- None
- none

- `warn`
- `raise`

`check_hash_pycs_mode`

(`string` or `None`)

Controls the value of PyConfig.check_hash_pycs_mode.

Accepted values are:

- `None`
- `always`
- `never`
- `default`

`configure_c_stdio`

(`bool` or `None`)

Controls the value of PyConfig.configure_c_stdio.

`dump_refs`

(`bool` or `None`)

Controls the value of PyConfig.dump_refs.

`exec_prefix`

(`string` or `None`)

Controls the value of PyConfig.exec_prefix.

`executable`

(`string` or `None`)

Controls the value of PyConfig.executable.

`fault_handler`

(`bool` or `None`)

Controls the value of PyConfig.fault_handler.

filesystem_encoding

(`string` or `None`)

Controls the value of PyConfig.filesystem_encoding.

filesystem_errors

(`string` or `None`)

Controls the value of PyConfig.filesystem_errors.

hash_seed

(`int` or `None`)

Controls the value of PyConfig.hash_seed.

`PyConfig.use_hash_seed` will automatically be set if this attribute is defined.

home

(`string` or `None`)

Controls the value of PyConfig.home.

import_time

(`bool` or `None`)

Controls the value of PyConfig.import_time.

inspect

(`bool` or `None`)

Controls the value of PyConfig.inspect.

install_signal_handlers

(`bool` or `None`)

Controls the value of PyConfig.install_signal_handlers.

interactive

(`bool` or `None`)

Controls the value of PyConfig.interactive.

legacy_windows_stdio

(`bool` or `None`)

Controls the value of PyConfig.legacy_windows_stdio.

malloc_stats

(`bool` or `None`)

Controls the value of PyConfig.malloc_stats.

module_search_paths

(`list[string]` or `None`)

Controls the value of PyConfig.module_search_paths.

This value effectively controls the initial value of `sys.path`.

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

Setting this to a non-empty value also has the side-effect of setting `filesystem_importer = True`

optimization_level

(`int` or `None`)

Controls the value of PyConfig.optimization_level.

Allowed values are:

- `None`
- `0`
- `1`
- `2`

This setting is only relevant if `write_bytecode` is `True` and Python modules are being imported from the filesystem using Python's standard filesystem importer.

parser_debug

(`bool` or `None`)

Controls the value of PyConfig.parser_debug.

pathconfig_warnings

(`bool` or `None`)

Controls the value of PyConfig.pathconfig_warnings.

prefix

(`string` or `None`)

Controls the value of PyConfig.prefix.

program_name

(`string` or `None`)

Controls the value of PyConfig.program_name.

pycache_prefix

(`string` or `None`)

Controls the value of PyConfig.pycache_prefix.

python_path_env

(`string` or `None`)

Controls the value of PyConfig.pythonpath_env.

quiet

(`bool` or `None`)

Controls the value of PyConfig.quiet.

run_command

(`string` or `None`)

Controls the value of PyConfig.run_command.

run_filename

(`string` or `None`)

Controls the value of PyConfig.run_filename.

run_module

(`string` or `None`)

Controls the value of PyConfig.run_module.

show_ref_count

(bool or None)

Controls the value of PyConfig.show_ref_count.

site_import

(bool or None)

Controls the value of PyConfig.site_import.

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

skip_first_source_line

(bool or None)

Controls the value of PyConfig.skip_first_source_line.

stdio_encoding

(string or None)

Controls the value of PyConfig.stdio_encoding.

stdio_errors

(string or None)

Controls the value of PyConfig.stdio_errors.

tracemalloc

(bool or None)

Controls the value of PyConfig.tracemalloc.

user_site_directory

(bool or None)

Controls the value of PyConfig.user_site_directory.

verbose

(bool or None)

Controls the value of PyConfig.verbose.

warn_options

(`list[string]` or `None`)

Controls the value of PyConfig.warn_options.

write_bytecode

(`bool` or `None`)

Controls the value of PyConfig.write_bytecode.

This only influences the behavior of Python standard path-based importer (controlled via `filesystem_importer`).

x_options

(`list[string]` or `None`)

Controls the value of PyConfig.xoptions.

4.13.3 Starlark Caveats

The `PythonInterpreterConfig` Starlark type is backed by a Rust data structure. And when attributes are retrieved, a copy of the underlying Rust struct field is returned.

This means that if you attempt to mutate a Starlark value (as opposed to assigning an attribute), the mutation won't be reflected on the underlying Rust data structure.

For example:

```
config = dist.make_python_interpreter_config()

# assigns vec!["foo", "bar"].
config.module_search_paths = ["foo", "bar"]

# Creates a copy of the underlying list and appends to that copy.
# The stored value of `module_search_paths` is still `["foo", "bar"]`.
config.module_search_paths.append("baz")
```

To append to a list, do something like the following:

```
value = config.module_search_paths
value.append("baz")
config.module_search_paths = value
```

4.14 PythonModuleSource

This type represents Python source modules, agnostic of location.

Instances can be constructed via *PythonExecutable.make_python_module_source()* or by calling methods that emit Python resources.

4.14.1 Attributes

The following sections describe the attributes available on each instance.

name

(`string`)

Fully qualified name of the module. e.g. `foo.bar`.

source

(`string`)

The Python source code for this module.

is_package

(`bool`)

Whether this module is also a Python package (or sub-package).

is_stdlib

(`bool`)

Whether this module is part of the Python standard library (part of the Python distribution).

add_*

(various)

See *Resource Attributes Influencing Adding*.

4.15 PythonPackageResource

This type represents a resource _file_ in a Python package. It is effectively a named blob associated with a Python package. It is typically accessed using the `importlib.resources` API.

4.15.1 Attributes

The following sections describe the attributes available on each instance.

package

(`string`)

Python package this resource is associated with.

name

(string)

Name of this resource.

is_stdlib

(bool)

Whether this module is part of the Python standard library (part of the Python distribution).

add_*

(various)

See *Resource Attributes Influencing Adding*.

4.16 PythonPackageDistributionResource

This type represents a named resource to make available as Python package distribution metadata. These files are typically accessed using the `importlib.metadata` API.

Each instance represents a logical file in a `<package>-<version>.dist-info` or `<package>-<version>.egg-info` directory. There are specifically named files that contain certain data. For example, a `*.dist-info/METADATA` file describes high-level metadata about a Python package.

4.16.1 Attributes

The following sections describe the attributes available on each instance.

package

(string)

Python package this resource is associated with.

name

(string)

Name of this resource.

is_stdlib

(bool)

Whether this module is part of the Python standard library (part of the Python distribution).

add_*

(various)

See *Resource Attributes Influencing Adding*.

4.17 PythonPackagingPolicy

When building a Python binary, there are various settings that control which Python resources are added, where they are imported from, and other various settings. This collection of settings is referred to as a *Python Packaging Policy*. These settings are represented by the `PythonPackagingPolicy` type.

4.17.1 Attributes

The following sections describe the attributes available on each instance.

allow_files

(`bool`)

Whether to allow the collection of generic *file* resources.

If false, all collected/packaged resources must be instances of concrete resource types (`PythonModuleSource`, `PythonPackageResource`, etc).

If true, *File* instances can be added to resource collectors.

allow_in_memory_shared_library_loading

(`bool`)

Whether to allow loading of Python extension modules and shared libraries from memory at run-time.

Some platforms (notably Windows) allow opening shared libraries from a memory address. This mode of opening shared libraries allows libraries to be embedded in binaries without having to statically link them. However, not every library works correctly when loaded this way.

This flag defines whether to enable this feature where supported. Its true value can be ignored if the target platform doesn't support loading shared library from memory.

bytecode_optimize_level_zero

(`bool`)

Whether to add Python bytecode at optimization level 0 (the default optimization level the Python interpreter compiles bytecode for).

bytecode_optimize_level_one

(`bool`)

Whether to add Python bytecode at optimization level 1.

bytecode_optimize_level_two

(bool)

Whether to add Python bytecode at optimization level 2.

extension_module_filter

(string)

The filter to apply to determine which extension modules to add. The following values are recognized:

all Every named extension module will be included.

minimal Return only extension modules that are required to initialize a Python interpreter. This is a very small set and various functionality from the Python standard library will not work with this value.

no-libraries Return only extension modules that don't require any additional libraries.

Most common Python extension modules are included. Extension modules like `_ssl` (links against OpenSSL) and `zlib` are not included.

no-copyleft Return only extension modules that do not link against *copyleft* 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 *copyleft* licensed. Similarly, the mechanism for determining whether a license is *copyleft* is based on the SPDX license annotations, which could be wrong or out of date.

Default is `all`.

file_scanner_classify_files

(bool)

Whether file scanning should attempt to classify files and emit typed resources corresponding to the detected file type.

If `True`, operations that emit resource objects (such as *PythonExecutable.pip_install()*) will emit specific types for each resource flavor. e.g. *PythonModuleSource*, *PythonExtensionModule*, etc.

If `False`, the file scanner does not attempt to classify the type of a file and this rich resource types are not emitted.

Can be used in conjunction with *file_scanner_emit_files*. If both are `True`, there will be a *File* and an optional non-file resource for each source file.

Default is `True`.

file_scanner_emit_files

(bool)

Whether file scanning should emit file resources for each seen file.

If `True`, operations that emit resource objects (such as *PythonExecutable.pip_install()*) will emit *File* instances for each encountered file.

If `False`, *File* instances will not be emitted.

Can be used in conjunction with *file_scanner_classify_files*.

Default is `False`.

include_classified_resources

(bool)

Whether strongly typed, classified non-File resources have their add_include attribute set to True by default.

Default is True.

include_distribution_sources

(bool)

Whether to add source code for Python modules in the Python distribution.

Default is True.

include_distribution_resources

(bool)

Whether to add Python package resources for Python packages in the Python distribution.

Default is False.

include_file_resources

(bool)

Whether *File* resources have their add_include attribute set to True by default.

Default is False.

include_non_distribution_sources

(bool)

Whether to add source code for Python modules not in the Python distribution.

include_test

(bool)

Whether to add Python resources related to tests.

Not all files associated with tests may be properly flagged as such. This is a best effort setting.

Default is False.

resources_location

(string)

The location that resources should be added to by default.

Default is in-memory.

resources_location_fallback

(`string` or `None`)

The fallback location that resources should be added to if `resources_location` fails.

Default is `None`.

preferred_extension_module_variants

(`dict<string, string>`) (readonly)

Mapping of extension module name to variant name.

This mapping defines which preferred named variant of an extension module to use. Some Python distributions offer multiple variants of the same extension module. This mapping allows defining which variant of which extension to use when choosing among them.

Keys set on this dict are not reflected in the underlying policy. To set a key, call the `set_preferred_extension_module_variant()` method.

4.17.2 Methods

The following sections describe methods on `PythonPackagingPolicy` instances.

PythonPackagingPolicy.register_resource_callback()

This method registers a Starlark function to be called when resource objects are created. The passed function receives 2 arguments: this `PythonPackagingPolicy` instance and the resource (e.g. `PythonModuleSource`) that was created.

The purpose of the callback is to enable Starlark configuration files to mutate resources upon creation so they can globally influence how those resources are packaged.

PythonPackagingPolicy.set_preferred_extension_module_variant()

This method will set a preferred Python extension module variant to use. See the documentation for `preferred_extension_module_variants` above for more.

It accepts 2 `string` arguments defining the extension module name and its preferred variant.

PythonPackagingPolicy.set_resource_handling_mode()

This method takes a string argument denoting the *resource handling mode* to apply to the policy. This string can have the following values:

classify Files are classified as typed resources and handled as such.

Only classified resources can be added by default.

files Files are handled as raw files (as opposed to typed resources).

Only files can be added by default.

This method is effectively a convenience method for bulk-setting multiple attributes on the instance given a behavior mode.

`classify` will configure the file scanner to emit classified resources, configure the `add_include` attribute to only be `True` on classified resources, and will disable the addition of `File` resources on resource collectors.

`files` will configure the file scanner to only emit `File` resources, configure the `add_include` attribute to `True` on `File` and *classified* resources, and will allow resource collectors to add `File` instances.

CHAPTER 5

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.

5.1 Creating a PyOxidizer Project

The process for *oxidizing* every Python application looks the same: you start by creating a new `PyOxidizer` configuration file via the `pyoxidizer init-config-file` command:

```
# Create a new configuration file in the directory "pyapp"
$ pyoxidizer init-config-file pyapp
```

Behind the scenes, `PyOxidizer` works by leveraging a Rust project to build binaries embedding Python. The auto-generated project simply instantiates and runs an embedded Python interpreter. If you would like your built binaries to offer more functionality, you can create a minimal Rust project to embed a Python interpreter and customize from there:

```
# Create a new Rust project for your application in ~/src/myapp.
$ pyoxidizer init-rust-project ~/src/myapp
```

The auto-generated configuration file and Rust project will launch a Python REPL by default. And the `pyoxidizer` executable will look in the current directory for a `pyoxidizer.bzl` configuration file. Let's test that the new configuration file or project works:

```
$ pyoxidizer run
...
   Compiling pyapp v0.1.0 (/home/gps/src/pyapp)
    Finished dev [unoptimized + debuginfo] target(s) in 53.14s
writing executable to /home/gps/src/pyapp/build/x86_64-unknown-linux-gnu/debug/exe/
↪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, you can use `cargo build` and `cargo run` directly. However, Rust's build system is only responsible for build binaries and some of the higher-level functionality from `PyOxidizer`'s configuration files (such as application packaging) will likely not be performed unless tweaks are made to the Rust project's `build.rs`.

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

5.2 Packaging Primitives in `pyoxidizer.bzl` Files

PyOxidizer's run-time behavior is controlled by `pyoxidizer.bzl` Starlark (a Python-like language) configuration files. See *Configuration Files* for documentation on these files, including low-level API documentation.

This document gives a medium-level overview of the important Starlark types and functions and how they all interact.

5.2.1 Targets Define Actions

As detailed at *Targets*, a PyOxidizer configuration file is composed of named *targets*, which are functions returning an object that may have a build or run action attached. Commands like `pyoxidizer build` identify a target to evaluate then effectively walk the dependency graph evaluating dependent targets until the requested target is *built*.

5.2.2 Defining an Executable Embedding Python

In this example, we create an executable embedding Python:

```
def make_dist():
    return default_python_distribution()

def make_exe(dist):
    return dist.to_python_executable("myapp")

register_target("dist", make_dist)
register_target("exe", make_exe, depends=["dist"], default=True)
```

`PythonDistribution.to_python_executable()` accepts an optional `PythonPackagingPolicy` instance that influences how the executable is built and what resources are added where. See the *type documentation* for the list of parameters that can be influenced. Some of this behavior is described in the sections below. Other examples are provided throughout the *Packaging User Guide* documentation.

5.2.3 Configuring the Python Interpreter Run-Time Behavior

The *PythonInterpreterConfig* Starlark type configures the default behavior of the Python interpreter embedded in built binaries.

A `PythonInterpreterConfig` instance is associated with `PythonExecutable` instances when they are created. A custom instance can be passed into `PythonDistribution.to_python_executable()` to use non-default settings.

In this example (similar to above), we construct a custom `PythonInterpreterConfig` instance using non-defaults and then pass this instance into the constructed `PythonExecutable`:

```
def make_dist():
    return default_python_distribution()

def make_exe(dist):
    config = dist.make_python_interpreter_config()
    config.run_command = "print('hello, world')"

    return dist.to_python_executable("myapp", config=config)

register_target("dist", make_dist)
register_target("exe", make_exe, depends=["dist"], default=True)
```

The `PythonInterpreterConfig` type exposes a lot of modifiable settings. See the *API documentation* for the complete list. These settings include but are not limited to:

- Control of low-level Python interpreter settings, such as whether environment variables (like `PYTHONPATH`) should influence run-time behavior, whether stdio should be buffered, and the filesystem encoding to use.
- Whether to enable the importing of Python modules from the filesystem and what the initial value of `sys.path` should be.
- The memory allocator that the Python interpreter should use.
- What Python code to run when the interpreter is started.
- How the `terminfo` database should be located.

Many of these settings are not needed for most programs and the defaults are meant to be reasonable for most programs. However, some settings - such as the `run_*` arguments defining what Python code to run by default - are required by most configuration files.

5.2.4 Adding Python Packages to Executables

A just-created `PythonExecutable` Starlark type contains just the Python interpreter and standard library derived from the `PythonDistribution` from which it came. While you can use PyOxidizer to produce an executable containing just a normal Python *distribution* with nothing else, many people will want to add their own Python packages/code.

The Starlark environment defines various types for representing Python package resources. These include *PythonModuleSource*, *PythonExtensionModule*, *PythonPackageDistributionResource*, and more.

Instances of these types can be created dynamically or by performing common Python packaging operations (such as invoking `pip install`) via various methods on `PythonExecutable` instances. These Python package resource instances can then be added to `PythonExecutable` instances so they are part of the built binary.

See *Managing How Resources are Added* and *Packaging Python Files* for more on this topic, including many examples.

5.2.5 Install Manifests Copy Files Next to Your Application

The *FileManifest* Starlark type represents a collection of files and their content. When `FileManifest` instances are returned from a target function, their build action results in their contents being manifested in a directory having the name of the build target.

`FileManifest` instances can be used to construct custom file *install layouts*.

Say you have an existing directory tree of files you want to copy next to your built executable defined by the `PythonExecutable` type.

The *glob()* function can be used to discover existing files on the filesystem and turn them into a `FileManifest`. You can then return this `FileManifest` directory or overlay it onto another instance using *FileManifest.add_manifest()*. Here's an example:

```
def make_dist():
    return default_python_distribution()

def make_exe(dist):
    return dist.to_python_executable("myapp")

def make_install(exe):
    m = FileManifest()

    m.add_python_resource(".", exe)

    templates = glob(["/path/to/project/templates/**/*"], strip_prefix="/path/to/
↪project/")
    m.add_manifest(templates)

    return m

register_target("dist", make_dist)
register_target("exe", make_exe, depends=["dist"])
register_target("install", make_install, depends=["exe"], default=True)
```

We introduce a new `install` target and `make_install()` function which returns a `FileManifest`. It adds the `PythonExecutable` (represented by the `exe` argument/variable) to that manifest in the root directory, signified by `.`.

Next, it calls `glob()` to find all files in the `/path/to/project/templates/` directory tree, strips the path prefix `/path/to/project/` from them, and then merges all of these files into the final manifest.

When the `InstallManifest` is built, the final layout should look something like the following:

- `install/myapp` (or `install/myapp.exe` on Windows)
- `install/templates/foo`
- `install/templates/...`

See *Packaging Files Instead of In-Memory Resources* for more on this topic.

5.3 Understanding Python Distributions

The *PythonDistribution* Starlark type represents a Python *distribution*, an entity providing a Python installation and build files which PyOxidizer uses to build your applications. See *Python Distributions Provide Python* for more.

5.3.1 Available Python Distributions

PyOxidizer ships with its own list of available Python distributions. These are constructed via the *default_python_distribution()* Starlark method. Under most circumstances, you'll want to use one of these distributions instead of providing your own because these distributions are tested and should have maximum compatibility.

Here are the built-in Python distributions:

Source	Version	Flavor	Build Target
CPython	3.8.8	standalone_dynamic	x86_64-unknown-linux-gnu
CPython	3.9.2	standalone_dynamic	x86_64-unknown-linux-gnu
CPython	3.8.8	standalone_static	x86_64-unknown-linux-musl
CPython	3.9.2	standalone_static	x86_64-unknown-linux-musl
CPython	3.8.8	standalone_dynamic	i686-pc-windows-msvc
CPython	3.9.2	standalone_dynamic	i686-pc-windows-msvc
CPython	3.8.8	standalone_static	i686-pc-windows-msvc
CPython	3.9.2	standalone_static	i686-pc-windows-msvc
CPython	3.8.8	standalone_dynamic	x86_64-pc-windows-msvc
CPython	3.9.2	standalone_dynamic	x86_64-pc-windows-msvc
CPython	3.8.8	standalone_static	x86_64-pc-windows-msvc
CPython	3.9.2	standalone_static	x86_64-pc-windows-msvc
CPython	3.9.2	standalone_dynamic	aarch64-apple-darwin
CPython	3.8.8	standalone_dynamic	x86_64-apple-darwin
CPython	3.9.2	standalone_dynamic	x86_64-apple-darwin

All of these distributions are provided by the python-build-standalone, and are maintained by the maintainer of PyOxidizer.

Here is what those target triple values translate to:

`aarch64-apple-darwin` 64-bit ARM compiled for macOS.

`i686-pc-windows-msvc` 32-bit Windows using the Microsoft Visual C++ Compiler.

`x86-64-pc-windows-msvc` 64-bit Windows using the Microsoft Visual C++ Compiler.

`x86_64-apple-darwin` 64-bit Intel processors compiled for macOS.

`x86_64-pc-unknown-linux-gnu` 64-bit x86 (typically Intel or AMD) targeting Linux, with a dependency on GNU libc (glibc / `libc.so`).

`x86_64-pc-unknown-linux-musl` 64-bit x86 (typically Intel or AMD) targeting Linux using musl libc. (Musl libc uses static linking for libc, unlike glibc.)

5.3.2 Python Version Compatibility

PyOxidizer is capable of working with Python 3.8 and 3.9.

Python 3.9 is the default Python version because it has been around for a while and is relatively stable.

PyOxidizer's tests are run primarily against the default Python version. So adopting a non-default version may risk running into subtle bugs.

5.3.3 Choosing a Python Distribution

The Python 3.9 distributions are the default and are better tested than the Python 3.8 distributions. 3.8 was the default in previous releases and is known to work.

The `standalone_dynamic` distributions behave much more similarly to traditional Python build configurations than do their `standalone_static` counterparts. The `standalone_dynamic` distributions are capable of loading Python extension modules that exist as shared library files. So when working with `standalone_dynamic` distributions, Python wheels containing pre-built Python extension modules often *just work*.

The downside to `standalone_dynamic` distributions is that you cannot produce a single file, statically-linked executable containing your application in most circumstances: you will need a `standalone_static` distribution to produce a single file executable.

But as soon as you encounter a third party extension module with a `standalone_static` distribution, you will need to recompile it. And this is often unreliable.

5.3.4 Binary Portability of Distributions

The built-in Python distributions are built in such a way that they should run on nearly every system for the platform they target. This means:

- All 3rd party shared libraries are part of the distribution (e.g. `libssl` and `libsqlite3`) and don't need to be provided by the run-time environment.
- Some distributions are statically linked and have no dependencies on any external shared libraries.
- On the glibc linked Linux distributions, they use an old glibc version for symbol versions, enabling them to run on Linux distributions created years ago. (The current version is 2.19, which was released in 2014.)
- Any shared libraries not provided by the distribution are available in base operating system installs. On Linux, example shared libraries include `libc.so.6` and `linux-vdso.so.1`, which are part of the Linux Standard Base Core Configuration and should be present on all conforming Linux distros. On macOS, referenced dylibs include `libSystem`, which is part of the macOS core install.
- For Linux, see *Distribution Considerations for Linux* for portability considerations.
- For macOS, see *Distribution Considerations for macOS* for portability considerations.
- For Windows, see *Distribution Considerations for Windows* for portability considerations.

5.3.5 Known Issues with Distributions

There are various known issues with various distributions. The python-build-standalone project documentation at https://python-build-standalone.readthedocs.io/en/latest/ attempts to capture many of them.

PyOxidizer contains workaround for many of the limitations. For example, PyOxidizer (specifically the `pyembed` Rust crate) can automatically configure the terminfo database at run-time.

The `aarch64-apple-darwin` Python distributions are considered beta quality because PyOxidizer does not have continuous CI coverage for this architecture. Releases should be tested before they are released. But there may be undetected breakage on unreleased commits on the `main` branch due to lack of CI coverage. This limitation should go away once GitHub Actions supports running jobs on M1 hardware.

5.4 Managing How Resources are Added

An important concept in PyOxidizer packaging is how to manage *resources* that are added to built applications.

A *resource* is some entity that will be packaged and distributed. Examples of *resources* include Python module source and bytecode, Python extension modules, and arbitrary files on the filesystem.

Resources are represented by a dedicated Starlark type for each resource flavor (see *Resource Types*).

During evaluation of PyOxidizer's Starlark configuration files, *resources* are created and *added* to another Starlark type whose job is to collect all desired *resources* and then do something with them.

5.4.1 Classified Resources Versus Files

All resources in PyOxidizer are ultimately derived from or representable by a file or a file-like primitive. For example, a *PythonModuleSource* is derived from or could be manifested as a `.py` file.

Various PyOxidizer functionality works by scanning existing files and turning those files into *resources*.

This file scanning functionality has two modes of operation: *classified* and *files*. In *files* mode, PyOxidizer simply emits resources corresponding to the raw files it encounters. In *classified* mode, PyOxidizer attempts to *classify* a file as a particular resource and emit a strongly-typed resource like *PythonModuleSource* or *PythonExtensionModule*.

Classified mode is more powerful because PyOxidizer is able to build an *index* of typed resources at packaging time and make this *index* available to *oxidized_importer Python Extension* at run-time to facilitate faster loading of resources.

However, the main downside to *classified* mode is it relies on being able to identify files properly and this is unreliable. Python file layouts are under-specified and there are many edge cases where PyOxidizer fails to properly classify a file. See *Debugging Resource Scanning and Identification with find-resources* for how to identify problems here.

In *files* mode, PyOxidizer simply indexes and manages a named file and its content. There is far less potential for PyOxidizer to make mistakes about a file's type and how it is handled. This means that *files* mode often *just works* when *classified* mode doesn't. The main downside to *files* mode is that *oxidized_importer Python Extension* doesn't have a rich index embedded in the built binary, so you will have to rely on Python's default filesystem-based importer, which is slower than `oxidized_importer`.

5.4.2 Packaging Policies and Adding Resources

The exact mechanism by which *resources* are emitted and added to *resource collectors* is influenced by a *packaging policy* (represented by the *PythonPackagingPolicy* Starlark type) and attributes on each resource object influencing how they are added.

When *resources* are created, the *packaging policy* determines whether emitted resources are *classified* or simply *files*. And the *packaging policy* is applied to each created resource to populate the initial values for the various `add_*` attributes on the Starlark *resource* types.

When a resource is added (e.g. by calling `PythonExecutable.add_python_resource()`), these aforementioned `add_*` attributes are consulted and used to influence exactly how that *resource* is added/packaged.

For example, a *PythonModuleSource* can set attributes indicating to exclude source code and only generate bytecode at a specific optimization level. Or a *PythonExtensionModule* can set attributes saying to prefer to compile it into the built binary or materialize it as a standalone dynamic extension module (e.g. `my_ext.so` or `my_ext.pyd`).

5.4.3 Resource Types

The following Starlark types represent individual resources:

PythonModuleSource Source code for a Python module. Roughly equivalent to a `.py` file.

> This type can also be converted to Python bytecode (roughly equivalent to a `.pyc`) when added to a resource collector.

PythonExtensionModule A Python module defined through compiled, machine-native code. On Linux, these are typically encountered as `.so` files. On Windows, `.pyd` files.

PythonPackageResource A non-module *resource file* loadable by Python resources APIs, such as those in `importlib.resources`.

PythonPackageDistributionResource A non-module *resource file* defining metadata for a Python package. Typically accessed via `importlib.metadata`. This is how files in `*.dist-info` or `*.egg-info` directories are represented.

File Represents a filesystem path and its content.

FileContent Represents the content of a filesystem file.

This is different from *File* in that it only represents file content and doesn't have an associated path. (It is likely these 2 types will be merged someday.)

There are also Starlark types that are logically containers for multiple resources:

FileManifest Holds a mapping of relative filesystem paths to `FileContent` instances. This type effectively allows modeling a directory tree.

PythonEmbeddedResources Holds a collection of Python resources of various types. (This type is often hidden away. e.g. inside a `PythonExecutable` instance.)

5.4.4 Resource Locations

Resources have the concept of a *location*. A resource's *location* determines where the data for that resource is packaged and how that resource is loaded at run-time.

In-Memory

When a Python resource is placed in the *in-memory* location, the content behind the resource will be embedded in a built binary and loaded from there by the Python interpreter.

Python modules imported from memory do not have the `__file__` attribute set. This can cause compatibility issues if Python code is relying on the existence of this module. See *__file__ and __cached__ Module Attributes* for more.

Filesystem-Relative

When a Python resource is placed in the *filesystem-relative* location, the resource will be materialized as a file next to the produced entity. e.g. a *filesystem-relative* `PythonModuleSource` for the `foo.bar` Python module added to a `PythonExecutable` will be materialized as the file `foo/bar.py` or `foo/bar/__init__.py` in a directory next to the built executable.

Resources added to *filesystem-relative* locations should be materialized under paths that preserve semantics with standard Python file layouts. For e.g. Python source and bytecode modules, it should be possible to point `sys.path` of any Python interpreter at the destination directory and the modules will be loadable.

During packaging, PyOxidizer *indexes* all *filesystem-relative* resources and embeds metadata about them in the built binary. While the files on the filesystem may look like a standard Python install layout, loading them is serviced by PyOxidizer's custom importer, not the standard importer that Python uses by default.

5.4.5 Customizing Python Packaging Policies

As described in *Packaging Policies and Adding Resources*, a `PythonPackagingPolicy` Starlark type instance is bound to every entity creating *resource* instances and this *packaging policy* is used to derive the default `add_*` attributes which influence what happens when a resource is added to some entity.

`PythonPackagingPolicy` instances can be customized to influence what the default values of the `add_*` attributes are.

The primary mechanisms for doing this are:

1. Modifying the `PythonPackagingPolicy` instance's internal state. See *PythonPackagingPolicy* for the full list of object attributes and methods that can be set or called.
2. Registering a function that will be called whenever a resource is created. This enables custom Starlark code to perform arbitrarily complex logic to influence settings and enables application developers to devise packaging strategies more advanced than what PyOxidizer provides out-of-the-box.

The following sections give examples of customized packaging policies.

Changing the Resource Handling Mode

As documented in *Classified Resources Versus Files*, PyOxidizer can operate on *classified* resources or *files*-based resources.

PythonPackagingPolicy.set_resource_handling_mode() exists to change the operating mode of a `PythonPackagingPolicy` instance.

```
def make_exe():
    dist = default_python_distribution()

    policy = dist.make_python_packaging_policy()

    # Set policy attributes to only operate on "classified" resource types.
    # (This is the default.)
    policy.set_resource_handling_mode("classify")

    # Set policy attributes to only operate on `File` resource types.
    policy.set_resource_handling_mode("files")
```

PythonPackagingPolicy.set_resource_handling_mode() is just a convenience method for manipulating a collection of attributes on `PythonPackagingPolicy` instances. If you don't like the behavior of its pre-defined modes, feel free to adjust attributes to suit your needs. You can even configure things to emit both *classified* and *files* variants simultaneously!

Customizing Default Resource Locations

The `PythonPackagingPolicy.resources_location` and `PythonPackagingPolicy.resources_location_fallback` attributes define primary and fallback locations that resources should attempt to be added to. These effectively define the default values for the `add_location` and `add_location_fallback` attributes on individual resource objects.

The accepted values are:

`in-memory` Load resources from memory.

`filesystem-relative:prefix` Load resources from the filesystem at a path relative to some entity (probably the binary being built).

Additionally, `PythonPackagingPolicy.resources_location_fallback` can be set to `None` to remove a fallback location.

And here is how you would manage these values in Starlark:

```
def make_exe():
    dist = default_python_distribution()

    policy = dist.make_python_packaging_policy()
    policy.resources_location = "in-memory"
    policy.resources_location_fallback = None

    # Only allow resources to be added to the in-memory location.
    exe = dist.to_python_executable(
        name = "myapp",
        packaging_policy = policy,
    )

    # Only allow resources to be added to the filesystem-relative location under
    # a "lib" directory.

    policy = dist.make_python_packaging_policy()
    policy.resources_location = "filesystem-relative:lib"
    policy.resources_location_fallback = None

    exe = dist.to_python_executable(
        name = "myapp",
        packaging_policy = policy,
    )

    # Try to add resources to in-memory first. If that fails, add them to a
    # "lib" directory relative to the built executable.

    policy = dist.make_python_packaging_policy()
    policy.resources_location = "in-memory"
    policy.resources_location_fallback = "filesystem-relative:lib"

    exe = dist.to_python_executable(
        name = "myapp",
        packaging_policy = policy,
    )

    return exe
```

Using Callbacks to Influence Resource Attributes

The `PythonPackagingPolicy.register_resource_callback(func)` method will register a function to be called when resources are created. This function receives as arguments the active `PythonPackagingPolicy` and the newly created resource.

Functions registered as resource callbacks are called after the `add_*` attributes are derived for a resource but before the resource is otherwise made available to other Starlark code. This means that these callbacks provide a hook point where resources can be modified as soon as they are created.

`register_resource_callback()` can be called multiple times to register multiple callbacks. Registered functions will be called in order of registration.

Functions can be leveraged to unify all resource packaging logic in a single place, making your Starlark configuration

files easier to reason about.

Here's an example showing how to route all resources belonging to a single package to a `filesystem-relative` location and everything else to memory:

```
def resource_callback(policy, resource):
    if type(resource) in ("PythonModuleSource", "PythonPackageResource",
↪"PythonPackageDistributionResource"):
        if resource.package == "my_package":
            resource.add_location = "filesystem-relative:lib"
        else:
            resource.add_location = "in-memory"

def make_exe():
    dist = default_python_distribution()

    policy = dist.make_python_packaging_policy()
    policy.register_resource_callback(resource_callback)

    exe = dist.to_python_executable(
        name = "myapp",
        packaging_policy = policy,
    )

    exe.add_python_resources(exe.pip_install(["my_package"]))
```

5.4.6 `PythonExtensionModule` Location Compatibility

Many resources *just work* in any available location. This is not the case for `PythonExtensionModule` instances!

While there only exists a single `PythonExtensionModule` type to represent Python extension modules, Python extension modules come in various flavors. Examples of flavors include:

- A module that is part of a Python *distribution* and is compiled into `libpython` (a *builtin* extension module).
- A module that is part of a Python *distribution* that is compiled as a standalone shared library (e.g. a `.so` or `.pyd` file).
- A non-*distribution* module that is compiled as a standalone shared library.
- A non-*distribution* module that is compiled as a static library.

Not all extension module *flavors* are compatible with all Python *distributions*. Furthermore, not all *flavors* are compatible with all build configurations.

Here are some of the rules governing extension modules and their locations:

- A *builtin* extension module that's part of a Python *distribution* will always be statically linked into `libpython`.
- A Windows Python distribution with a statically linked `libpython` (e.g. the `standalone_static` *distribution flavor*) is not capable of loading extension modules defined as shared libraries and only supports loading *builtin* extension modules statically linked into the binary.
- A Windows Python distribution with a dynamically linked `libpython` (e.g. the `standalone_dynamic` *distribution flavor*) is capable of loading shared library backed extension modules from the *in-memory* location. Other operating systems do not support the *in-memory* location for loading shared library extension modules.
- If the current build configuration targets Linux MUSL-libc, shared library extension modules are not supported and all extensions must be statically linked into the binary.

- If the object files for the extension module are available, the extension module may be statically linked into the produced binary.
- If loading extension modules from in-memory import is supported, the extension module will have its dynamic library embedded in the binary.
- The extension module will be materialized as a file next to the produced binary and will be loaded from the filesystem. (This is how Python extension modules typically work.)

Note: Extension module handling is one of the more nuanced aspects of PyOxidizer. There are likely many subtle bugs and room for improvement. If you experience problems handling extension modules, please consider filing an issue.

5.5 Packaging Python Files

The most important packaged *resource type* are arguably Python files: source modules, bytecode modules, extension modules, package resources, etc.

For PyOxidizer to recognize these Python resources as Python resources (as opposed to regular files), you will need to use the methods on the *PythonExecutable* Starlark type to use the settings from the thing being built to scan for resources, possibly performing a Python packaging action (such as invoking `pip install`) along the way.

This documentation covers the available methods and how they can be used.

5.5.1 `PythonExecutable` Python Resources Methods

The `PythonExecutable` Starlark type has the following methods that can be called to perform an action and obtain an iterable of objects representing discovered resources:

pip_download(...) Invokes `pip download` with specified arguments and collects resources discovered from downloaded Python wheels.

pip_install(...) Invokes `pip install` with specified arguments and collects all resources installed by that process.

read_package_root(...) Recursively scans a filesystem directory for Python resources in a typical Python installation layout.

setup_py_install(...) Invokes `python setup.py install` for a given path and collects resources installed by that process.

read_virtualenv(...) Reads Python resources present in an already populated virtualenv.

Typically, the Starlark types resolved by these method calls are passed into a method that adds the resource to a to-be-generated entity, such as the *PythonExecutable* Starlark type.

The following sections demonstrate common use cases.

5.5.2 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-config-file pyflakes
```

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

Find the `make_exe()` function. This function returns a *PythonExecutable* instance which defines a standalone executable containing Python. This function is a registered *target*, which is a named entity that can be individually built or run. By returning a `PythonExecutable` instance, this function/target is saying *build an executable containing Python*.

The `PythonExecutable` type holds all state needed to package and run a Python interpreter. This includes low-level interpreter configuration settings to which Python resources (like source and bytecode modules) are embedded in that executable binary. This type exposes an *add_python_resources()* method which adds an iterable of objects representing Python resources to the set of embedded resources.

Elsewhere in this function, the `dist` variable holds an instance of *PythonDistribution*. This type represents a Python distribution, which is a fancy way of saying *an implementation of Python*.

Two of the methods exposed by `PythonExecutable` are *pip_download()* and *pip_install()*, which invoke `pip` commands with settings to target the built executable.

To add a new Python package to our executable, we call one of these methods then add t he results to our `PythonExecutable` instance. This is done like so:

```
exe.add_python_resources(exe.pip_download(["pyflakes==2.2.0"]))
# or
exe.add_python_resources(exe.pip_install(["pyflakes==2.2.0"]))
```

When called, these methods will effectively run `pip download pyflakes==2.2.0` or `pip install pyflakes==2.2.0`, respectively. Actions are performed in a temporary directory and after `pip` runs, PyOxidizer will collect all the downloaded/installed resources (like module sources and bytecode data) and return them as an iterable of Starlark values. The `exe.add_python_resources()` call will then teach the built executable binary about the existence of these resources. Many resource types will be embedded in the binary and loaded from binary. But some resource types (notably compiled extension modules) may be installed next to the built binary and loaded from the filesystem.

Next, we tell PyOxidizer to run `pyflakes` when the interpreter is executed:

```
python_config.run_command = "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 `make_exe()` function should look something like the following (with comments removed for brevity):

```
def make_exe(dist):
    policy = dist.make_python_packaging_policy()
    policy.extension_module_filter = "all"
    policy.include_distribution_sources = True
    policy.include_distribution_resources = True
    policy.include_test = False

    config = dist.make_python_interpreter_config()
    config.run_command = "from pyflakes.api import main; main()"

    exe = dist.to_python_executable(
        name="pyflakes",
        packaging_policy=policy,
```

(continues on next page)

(continued from previous page)

```
        config=config,
    )

    exe.add_python_resources(exe.pip_install(["pyflakes==2.1.1"]))

    return exe
```

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

```
# From outside the ``pyflakes`` directory
$ pyoxidizer run --path /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.

5.5.3 Packaging an Application from an Existing Virtualenv

This scenario is very similar to the above example. So we'll only briefly describe what to do so we don't repeat ourselves.:

```
$ pyoxidizer init-config-file /path/to/myapp
```

Now edit the `pyoxidizer.bzl` so the `make_exe()` function look like the following:

```
def make_exe(dist):
    policy = dist.make_python_packaging_policy()
    policy.extension_module_filter = "all"
    policy.include_distribution_sources = True
    policy.include_distribution_resources = False
    policy.include_test = False

    config = dist.make_python_interpreter_config()
    config.run_command = "from myapp import main; main()"

    exe = dist.to_python_executable(
        name="myapp",
        packaging_policy=policy,
        config=config,
    )

    exe.add_python_resources(exe.read_virtualenv("/path/to/virtualenv"))

    return exe
```

Of course, you need a populated virtualenv!:

```
$ python3.8 -m venv /path/to/virtualenv
$ /path/to/virtualenv/bin/pip install -r /path/to/requirements.txt
```

Once all the pieces are in place, simply run `pyoxidizer` to build and run the application:

```
$ pyoxidizer run --path /path/to/myapp
```

Warning: When consuming a pre-populated virtualenv, there may be compatibility differences between the Python distribution used to populate the virtualenv and the Python distributed used by PyOxidizer at build and application run time.

For best results, it is recommended to use a packaging method like `pip_install(...)` or `setup_py_install(...)` to use PyOxidizer's Python distribution to invoke Python's packaging tools.

5.5.4 Packaging an Application from a Local Python Package

Say you have a Python package/application in a local directory. It follows the typical Python package layout and has a `setup.py` file and Python files in sub-directories corresponding to the package name. e.g.:

```
setup.py
mypackage/__init__.py
mypackage/foo.py
```

You have a number of choices as to how to proceed here. Again, the workflow is very similar to what was explained above. The main difference is the content of the `pyoxidizer.bzl` file and the exact *method* to call to obtain the Python resources.

You could use `pip install <local path>` to use `pip` to process a local filesystem path:

```
exe.add_python_resources(exe.pip_install(["/path/to/local/package"]))
```

If the `pyoxidizer.bzl` file is in the same directory as the directory you want to process, you can derive the absolute path to this directory via the *CWD* Starlark variable:

```
exe.add_python_resources(exe.pip_install([CWD]))
```

If you don't want to use `pip` and want to run `setup.py` directly, you can do so:

```
exe.add_python_resources(exe.setup_py_install(package_path=CWD))
```

Or if you don't want to run a Python packaging tool at all and just scan a directory tree for Python files:

```
exe.add_python_resources(exe.read_package_root(CWD, ["mypackage"]))
```

Note: In this mode, all Python resources must already be in place in their final installation layout for things to work correctly. Many `setup.py` files perform additional actions such as compiling Python extension modules, installing additional files, dynamically generating some files, or changing the final installation layout.

For best results, use a packaging method that invokes a Python packaging tool (like `pip_install(...)` or `setup_py_install(...)`.

5.5.5 Choosing Which Packaging Method to Call

There are a handful of different methods for obtaining Python resources that can be added to a resource collection. Which one should you use?

The reason there are so many methods is because the answer is: *it depends*.

Each method for obtaining resources has its niche use cases. That being said, **the preferred method for obtaining Python resources is pip_download()**. However, `pip_download()` may not work in all cases, which is why other methods exist.

PythonExecutable.pip_download() runs `pip download` and attempts to fetch Python wheels for specified packages, requirements files, etc. It then extracts files from inside the wheel and converts them to Python resources which can be added to resource collectors.

Important: `pip_download()` will only work if a compatible Python *wheel* package (`.whl` file) is available. If the configured Python package repository doesn't offer a compatible wheel for the specified package or any of its dependencies, the operation will fail.

Many Python packages do not yet publish wheels (only `.tar.gz` archives) or don't publish at all to Python package repositories (this is common in corporate environments, where you don't want to publish your proprietary packages on PyPI or you don't run a Python package server).

Important: Not all build targets support `pip_download()` for all published packages. For example, when targeting Linux musl libc, built binaries are fully static and aren't capable of loading Python extension modules (which are shared libraries). So `pip_download()` only supports source-only Python wheels in this configuration.

Another advantage of `pip_download()` is it supports cross-compiling. Unlike `pip install`, `pip download` supports arguments that tell it which Python version, platform, implementation, etc to download packages for. PyOxidizer automatically tells `pip download` to download wheels that are compatible with the target environment you are building for. This means you can do things like download wheels containing Windows binaries when building on Linux.

Note: Cross-compiling is not yet fully supported by PyOxidizer and likely doesn't work in many cases. However, this is a planned feature (at least for some configurations) and `pip_download()` is likely the most future-proof mechanism to support installing Python packages when cross-compiling.

A potential downside with `pip_download()` is that it only supports classical Python binary loading/shipping techniques. If you are trying to produce a statically linked executable containing custom Python extension modules, `pip_download()` won't work for you.

After `pip_download`, *PythonExecutable.pip_install()* and *PythonExecutable.setup_py_install()* are the next most-preferred packaging methods.

Both of these work by locally running a Python packaging action (`pip install` or `python setup.py install`, respectively) and then collecting resources installed by that action.

The advantage over `pip download` is that a pre-built Python wheel does not have to be available and published on a Python package repository for these commands to work: you can run either against say a local version control checkout of a Python project and it should work.

The main disadvantage over `pip download` is that you are running Python packaging operations on the local machine as part of building an executable. If your package contains just Python code, this should *just work*. But if you need to compile extension modules, there's a good chance your local machine may either not be able to build them properly or will build those extension modules in such a way that they aren't compatible with other machines you want to run them on.

The final options for obtaining Python resources are *PythonExecutable.read_package_root()* and *PythonExecutable.read_virtualenv()*. Both of these methods rely on traversing a filesystem tree that is already populated with

Python resources. This should *just work* if only pure Python resources are in play. **But if there are compiled Python extension modules, all bets are off and there is no guarantee that found extension modules will be compatible with PyOxidizer or will have binary compatibility with other machines.** These resource discovery mechanisms also rely on state not under the control of PyOxidizer and therefore packaging results may be highly inconsistent and not reproducible across runs. For these reasons, **read_package_root() and read_virtualenv() are the least preferred methods for Python resource discovery.**

5.6 Packaging Files Instead of In-Memory Resources

By default, PyOxidizer will *classify* files into typed resources and attempt to load these resources from memory (with the exception of compiled extension modules, which require special treatment). Please read *Managing How Resources are Added*, specifically *Classified Resources Versus Files* and *Resource Locations* for more on the concepts of *classification* and *resource locations*.

This is the ideal packaging method because it keeps the entire application self-contained and can result in *performance wins* at run-time.

However, sometimes this approach isn't desired or flat out doesn't work. Fear not: PyOxidizer has you covered.

5.6.1 Examples of Packaging Failures

Let's give some concrete examples of how PyOxidizer's default packaging settings can fail.

black

Let's demonstrate a failure attempting to package black, a Python code formatter.

We start by creating a new project:

```
$ pyoxidizer init-config-file black
```

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

```
def make_exe(dist):
    config = dist.make_python_interpreter_config()
    config.run_module = "black"

    exe = dist.to_python_executable(
        name = "black",
    )

    for resource in exe.pip_install(["black==19.3b0"]):
        resource.add_location = "in-memory"
        exe.add_python_resource(resource)

    return exe
```

Then let's attempt to build the application:

```
$ pyoxidizer build --path black
processing config file /home/gps/src/black/pyoxidizer.bzl
resolving Python distribution...
...
```

Looking good so far!

Now let's try to run it:

```
$ pyoxidizer run --path 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 *__file__ and __cached__ Module Attributes*, `PyOxidizer` doesn't set `__file__` for modules loaded from memory. This is perfectly legal as Python doesn't mandate that `__file__` be defined. But `black` (and many other Python modules) assume `__file__` always exists. So it is a problem we have to deal with.

NumPy

Let's attempt to package NumPy, a popular Python package used by the scientific computing crowd.

```
$ pyoxidizer init-config-file numpy
```

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

```
def make_exe(dist):
    policy = dist.make_python_packaging_policy()
    policy.resources_location_fallback = "filesystem-relative:lib"

    exe = dist.to_python_executable(
        name = "numpy",
        packaging_policy = policy,
    )

    for resource in exe.pip_download(["numpy==1.19.0"]):
        resource.add_location = "filesystem-relative:lib"
        exe.add_python_resource(resource)

    return exe
```

We did things a little differently from the `black` example above: we're explicitly adding NumPy's resources into the `filesystem-relative` location so they are materialized as files instead of loaded from memory. This is to demonstrate a separate failure mode.

Then let's attempt to build the application:

```
$ pyoxidizer build --path numpy
processing config file /home/gps/src/numpy/pyoxidizer.bzl
resolving Python distribution...
...
```

Looking good so far!

Now let's try to run it:

```
$ pyoxidizer run --path numpy
...
```

(continues on next page)

(continued from previous page)

```
Python 3.8.6 (default, Oct  3 2020, 20:48:20)
[Clang 10.0.1 ] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy
Traceback (most recent call last):
  File "numpy.core", line 22, in <module>
  File "numpy.core.multiarray", line 12, in <module>
  File "numpy.core.overrides", line 7, in <module>
ImportError: libopenblasp-r0-ae94cfde.3.9.dev.so: cannot open shared object file: No␣
↪such file or directory

During handling of the above exception, another exception occurred:
...
```

That's not good! What happened?

Well, the hint is in the stack trace: `libopenblasp-r0-ae94cfde.3.9.dev.so: cannot open shared object file: No such file or directory`. So there's a file named `libopenblasp-r0-ae94cfde.3.9.dev.so` that can't be found. Let's look in our install layout:

```
$ find numpy/build/x86_64-unknown-linux-gnu/debug/install/ | grep libopenblasp
numpy/build/x86_64-unknown-linux-gnu/debug/install/lib/numpy/libs/libopenblasp-r0-
↪ae94cfde
numpy/build/x86_64-unknown-linux-gnu/debug/install/lib/numpy/libs/libopenblasp-r0-
↪ae94cfde/3
numpy/build/x86_64-unknown-linux-gnu/debug/install/lib/numpy/libs/libopenblasp-r0-
↪ae94cfde/3/9
numpy/build/x86_64-unknown-linux-gnu/debug/install/lib/numpy/libs/libopenblasp-r0-
↪ae94cfde/3/9/dev.so
```

Well, we found some files, including a `.so` file! But the filename has been mangled.

This filename mangling is actually a bug in PyOxidizer's file/resource classification. See *Incorrect Resource Identification* and *Classified Resources Versus Files* for more.

5.6.2 Installing Classified Resources on the Filesystem

In the *black* example above, we saw how `black` failed to run with modules imported from memory because of `__file__` not being defined.

In scenarios where in-memory resource loading doesn't work, the ideal mitigation is to fix the offending Python modules so they can load from memory. But this isn't always trivial or possible with 3rd party dependencies.

Your next mitigation should be to attempt to place the resource on the filesystem, next to the built binary.

This will require configuration file changes.

The goal of our new configuration is to materialize Python resources associated with `black` on the filesystem instead of in memory.

Change your configuration file so `make_exe()` looks like the following:

```
def make_exe(dist):
    policy = dist.make_python_packaging_policy()
    policy.resources_location_fallback = "filesystem-relative:lib"

    python_config = dist.make_python_interpreter_config()
```

(continues on next page)

(continued from previous page)

```
    python_config.run_module = "black"

    exe = dist.to_python_executable(
        name = "black",
        packaging_policy = policy,
        config = python_config,
    )

    for resource in exe.pip_install(["black==19.3b0"]):
        resource.add_location = "filesystem-relative:lib"
        exe.add_python_resource(resource)

    return exe
```

There are a few changes here.

We constructed a new *PythonPackagingPolicy* via *PythonDistribution.make_python_packaging_policy()* and set its *resources_location_fallback* attribute to `filesystem-relative-lib`. This allows us to install resources on the filesystem, relative to the produced binary.

Next, in the `for resource in exe.pip_install(...)` loop, we set `resource.add_location = "filesystem-relative:lib"`. What this does is tell the subsequent call to *PythonExecutable.add_python_resource()* to add the resource as a filesystem-relative resource in the `lib` directory.

With the new configuration in place, let's re-build and run the application:

```
$ pyoxidizer run --path black
...
adding extra file lib/toml-0.10.1.dist-info/top_level.txt to .
installing files to /home/gps/tmp/myapp/build/x86_64-unknown-linux-gnu/debug/install
No paths given. Nothing to do
```

That `No paths given` output is from `black`: it looks like the new configuration worked!

If you examine the build output, you'll see a bunch of messages indicating that extra files are being installed to the `lib/` directory. And if you poke around in the `install` directory, you will in fact see all these files.

In this configuration file, the Python distribution's files are all loaded from memory but `black` resources (collected via `pip install black`) are materialized on the filesystem. All of the resources are indexed by PyOxidizer at build time and that index is embedded into the built binary so *oxidized_importer Python Extension* can find and load resources more efficiently.

Because only some of the Python modules used by `black` have a dependency on `__file__`, it is probably possible to cherry pick exactly which resources are materialized on the filesystem and minimize the number of files present. We'll leave that as an exercise for the reader.

5.6.3 Installing Unclassified Files on the Filesystem

In *Installing Classified Resources on the Filesystem* we demonstrated how to move *classified* resources from memory to the filesystem in order to work around issues importing a module from memory.

Astute readers may have already realized that this workaround (setting `.add_location` to `filesystem-relative:...`) was attempted in the *NumPy* failure example above. So this workaround doesn't always work.

In cases where PyOxidizer's resource classifier or logic to materialize those classified resources as files is failing (presumably due to bugs in PyOxidizer), you can fall back to using *unclassified*, file-based resources. See *Classified Resources Versus Files* for more on *classified* versus *files* based resources.

Our approach here is to switch from *classified* to *files* packaging mode. Using our NumPy example from above, change the `make_exe()` in your configuration file to as follows:

```
def make_exe(dist):
    policy = dist.make_python_packaging_policy()
    policy.set_resource_handling_mode("files")
    policy.resources_location_fallback = "filesystem-relative:lib"

    python_config = dist.make_python_interpreter_config()
    python_config.module_search_paths = ["$ORIGIN/lib"]

    exe = dist.to_python_executable(
        name = "numpy",
        packaging_policy = policy,
        config = python_config,
    )

    for resource in exe.pip_download(["numpy==1.19.0"]):
        resource.add_location = "filesystem-relative:lib"
        exe.add_python_resource(resource)

    return exe
```

There are a few key lines here.

`policy.set_resource_handling_mode("files")` calls a method on the *PythonPackagingPolicy* to set the resource handling mode to *files*. This effectively enables *File* based resources to work. Without it, resource scanners won't emit *File* and attempts at adding *File* to a resource collection will fail.

Next, we enable file-based resource installs by setting *resources_location_fallback*.

Another new line is `python_config.module_search_paths = ["$ORIGIN/lib"]`. This all-important line to set *module_search_paths* effectively installs the `lib` directory next to the executable on `sys.path` at run-time. And as a side-effect of defining this attribute, Python's built-in module importer is enabled (to supplement `oxidized_importer`). This is important because because when you are operating in *files* mode, resources are indexed as *files* and not classified/typed resources. This means `oxidized_importer` doesn't recognize them as loadable Python modules. But since you enable Python's standard importer and register `lib/` as a search path, Python's standard importer will be able to find the `numpy` package at run-time.

Anyway, let's see if this actually works:

```
$ pyoxidizer run --path numpy
...
adding extra file lib/numpy.libs/libgfortran-2e0d59d6.so.5.0.0 to .
adding extra file lib/numpy.libs/libopenblasp-r0-ae94cfde.3.9.dev.so to .
adding extra file lib/numpy.libs/libquadmath-2d0c479f.so.0.0.0 to .
adding extra file lib/numpy.libs/libz-eb09ad1d.so.1.2.3 to .
installing files to /home/gps/tmp/myapp/build/x86_64-unknown-linux-gnu/debug/install
Python 3.8.6 (default, Oct  3 2020, 20:48:20)
[Clang 10.0.1 ] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy
>>> numpy.__loader__
<_frozen_importlib_external.SourceFileLoader object at 0x7f063da1c7f0>
```

It works!

Critically, we see that the formerly missing `libopenblasp-r0-ae94cfde.3.9.dev.so` file is being installed to the correct location. And we can confirm from the `numpy.__loader__` value that the standard library's module loader is being used. Contrast with a standard library module:

```
>>> import pathlib
>>> pathlib.__loader__
<OxidizedFinder object at 0x7f063dc8f8f0>
```

Enabling *files* mode and falling back to Python's importer is often a good way of working around bugs in PyOxidizer's *resource handling*. But it isn't bulletproof.

Important: Please *file a bug report <https://github.com/indygreg/PyOxidizer/issues>* if you encounter any issues with PyOxidizer's handling of resources and paths.

5.7 Working with Python Extension Modules

Python extension modules are machine native code exposing functionality to a Python interpreter via Python modules.

PyOxidizer has varying levels of support for extension modules. This is because some PyOxidizer configurations break assumptions about how Python interpreters typically run.

This document attempts to capture all the nuances of working with Python extension modules with PyOxidizer.

5.7.1 Extension Module Flavors

Python extension modules exist as either *built-in* or *standalone*. A *built-in* extension module is statically linked into *libpython* and a *standalone* extension module is a shared library that is dynamically loaded at run-time.

Typically, *built-in* extension modules only exist in Python distributions (and are part of the Python standard library by definition) and Python package maintainers only ever produce *standalone* extension modules (e.g. as `.so` or `.pyd` files).

Python distributions typically contain a mix of *built-in* and *standalone* extension modules. e.g. the `_ast` extension module is *built-in* and the `_ssl` extension module is *standalone*.

Important: Because PyOxidizer enables you to build your own binaries embedding Python and because different Python distributions have different levels of support for extension modules, it is important to familiarize yourself with the types of extension modules and how they can be used.

5.7.2 Extension Module Restrictions

PyOxidizer imposes a handful of restrictions on how extension modules work. These restrictions are typically a side-effect of limitations of the *Python distribution* being used/targeted. These restrictions are documented in the sections below.

musl libc Linux Distributions Only Support Built-in Extension Modules

The Python distributions built against musl libc (build target `*-linux-musl`) only support *built-in* extension modules.

This is because musl libc binaries are statically linked and statically linked Linux binaries are incapable of calling `dlopen()` to load a shared library.

This means Python binaries built in this configuration cannot load *standalone* Python extension modules existing as separate files (`.so` files typically). This means PyOxidizer cannot consume Python wheels or other Python resource sources containing pre-built Python extension modules.

In order for PyOxidizer to support a Python extension module built for musl libc, it must compile that extension module from source and link the resulting object files / static library directly into the built binary and expose that extension module as a *built-in*. This is done using *Building with a Custom Distutils*.

Windows Static Distributions Only Support Built-in Extension Modules

The Windows `standalone_static` distribution flavor only supports *built-in* extension modules and doesn't support loading shared library extension modules.

See the above section for implications on this.

The situation of having to rebuild Python extension modules on Windows is often more complicated than on Linux because oftentimes building extension modules on Windows isn't as trivial as on Linux. This is because many Windows environments don't have the correct version of Visual Studio or various library dependencies. If you want a turnkey experience for Windows packaging, it is recommended to use the `standalone_dynamic` distribution flavor.

Loading Extension Modules from `in-memory` Location

When you attempt to add a *PythonExtensionModule* Starlark instance to the `in-memory` *resource location*, the request may or may not work depending on the state of the extension module and support from the Python distribution.

The `in-memory` resource location is interpreted by PyOxidizer as *load this extension from memory, without having a standalone file*. PyOxidizer will try its hardest to satisfy this request.

If the object files / static library of an extension module are known to PyOxidizer, these will be statically linked into the built binary and the extension module will be exposed as a *built-in* extension module.

If only a shared library is available for the extension module, PyOxidizer only supports loading shared libraries from memory on Windows `standalone_dynamic` distributions: in all other platforms the request to load a shared library extension module is rejected.

Some extensions and shared libraries are known to not work when loaded from memory using the custom shared library loader used by PyOxidizer. For this reason, *allow_in_memory_shared_library_loading* exists to control this behavior.

Important: Because the `in-memory` location for extension modules can be brittle, it is recommended to set a resources policy or `add_location_fallback` to allow extension modules to exist as standalone files. This will provide maximum compatibility with built Python extension modules and will reduce the complexity of packaging 3rd party extension modules.

5.7.3 Extension Module Library Dependencies

PyOxidizer doesn't currently support resolving additional library dependencies from discovered extension modules outside of the Python distribution. For example, if your extension module `foo.so` has a run-time dependency on `bar.so`, PyOxidizer doesn't yet detect this and doesn't realize that `bar.so` needs to be handled.

This means that if you add a *PythonExtensionModule* Starlark type and this extension module depends on an additional library, PyOxidizer will likely not realize this and fail to distribute that additional library dependency with your application.

If your Python extensions depend on additional libraries, you may need to manually add these files to your installation via custom Starlark code.

Note that if your shared library exists as a file in Python package (a directory with `__init__.py` somewhere in the hierarchy), PyOxidizer's resource scanning may detect the shared library as a *PythonPackageResource* and package this resource. However, the packaged resource won't be flagged as a shared library. This means that the run-time importer won't identify the shared library dependency and won't take steps to ensure it is available/loaded before the extension is loaded. This means that the shared library loading needs to be handled by the operating system's default rules. And this means that the shared library file must exist on the filesystem, next to a file-based extension module.

5.7.4 Building with a Custom Distutils

If PyOxidizer is not able to reuse an existing shared library extension module or the build configuration is forcing an extension to be built as a *built-in*, PyOxidizer attempts to compile the extension module from source so that it can be statically linked as a *built-in*.

The way PyOxidizer achieves this is a bit crude, but often 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 compiled) such that the build emits artifacts that PyOxidizer can statically link into a custom binary.

For example, on Linux, PyOxidizer copies the intermediate object files produced by the build and links them into the binary containing the generated `libpython`. 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.

The easiest way to avoid the pitfalls of a custom `distutils` build is to not attempt to produce a statically linked binary: use a `standalone_dynamic` distribution flavor that supports loading extension modules from files.

Until PyOxidizer supports telling it additional object files or static libraries to link into a binary, there's no easy workaround aside from giving up on a statically linked binary. Better support will hopefully be present in future versions of PyOxidizer.

5.8 Managing *Packed* Resources Data

PyOxidizer's custom module importer (see *OxidizedFinder Python Type*) reads data in a custom serialization format (see *Python Packed Resources*) to facilitate efficient module importing and resource loading. If you are using this module importer (controlled from the *oxidized_importer* attribute, which is enabled by default), the interpreter will need to reference this *packed resources data* at run-time.

The *PythonExecutable.packed_resources_load_mode* attribute can be used in config files to control how this resources data should be read.

5.8.1 Available Resource Data Load Modes

Embedded

The *embedded* resources load mode (the default) will embed raw resources data into the binary and it will be read from memory at run-time.

This mode is necessary to achieve self-contained, single-file executables. This mode is also useful for single executable applications, where only a single executable file embeds a Python interpreter.

This mode is also likely the fastest mode, as no explicit filesystem I/O needs to be performed to reference resources data at run-time.

Binary Relative Memory Mapped File

The *binary relative memory mapped file* load mode will write resources data into a standalone file that is installed next to the built binary. At run-time, that file will be memory mapped and memory mapped I/O will be used.

This mode is useful for multiple executable applications, as it enables the resources data to be shared across executables without bloating total distribution size.

Here's an example:

```
def make_exe():
    dist = default_python_distribution()

    exe = dist.to_python_executable(
        name = "myapp",
    )

    # Write and load resources from a "myapp.pypacked" file next to
    # the executable.
    exe.packed_resources_load_mode = "binary-relative-memory-mapped:myapp.pypacked"

    return exe
```

None / Disabled

The resources load mode of `none` will disable the writing and loading of this *packed resources data*. This effectively means `OxidizedFinder` can't load anything by default.

This mode can be useful to produce a binary that behaves like `python`, without PyOxidizer's special run-time code. (See *Building an Executable that Behaves Like python* for more on this topic.)

If this mode is in use, you will need to enable Python's filesystem importer (*filesystem_importer*) or define custom Rust code to have `OxidizedFinder` *index* resources or else the embedded Python interpreter will fail to initialize due to missing modules.

5.9 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 config file logic that chooses which resources are packaged.

But maintaining explicit lists of resources can be tedious. `PyOxidizer` offers a more automated approach to solving this problem.

The *PythonInterpreterConfig* type 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 different build *target* to filter resources through a set of *only include* names.

TODO this functionality was temporarily dropped as part of the Starlark port.

5.10 Performance of Built Binaries

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

5.10.1 Resources Data Compiled Into Binary

Traditionally, when Python needs to `import` a module, it traverses the entries on `sys.path` and queries the filesystem to see whether a `.pyc` file, `.py` file, etc are available until it finds a suitable file to provide the Python module data. If you trace the system calls of a Python process (e.g. `strace -f python3 ...`), you will see tons of `lstat()`, `open()`, and `read()` calls performing filesystem I/O.

While filesystems cache the data behind these I/O calls, every time Python looks up data in a file the process needs to context switch into the kernel and then pass data back to Python. Repeated thousands of times - or even millions of times across hundreds or thousands of process invocations - the few microseconds of overhead plus the I/O overhead for a cache miss can add up to significant overhead!

When binaries are built with PyOxidizer, all available Python resources are discovered at build time. An index of these resources along with the raw resource data is packed - often into the executable itself - and made available to PyOxidizer's *custom importer*. When PyOxidizer services an `import` statement, looking up a module is effectively looking up a key in a dictionary: there is no explicit filesystem I/O to discover the location of a resource.

PyOxidizer's packed resources data supports storing raw resource data inline or as a reference via a filesystem path.

If inline storage is used, resources are effectively loaded from memory, often using 0-copy. There is no explicit filesystem I/O. The only filesystem I/O that can occur is indirect, as the operating system pages a memory page on first access. But this all happens in the kernel memory subsystem and is typically faster than going through a functionally equivalent system call to access the filesystem.

If filesystem paths are stored, the only filesystem I/O we require is to `open()` the file and `read()` its file descriptor: all filesystem I/O to locate the backing file is skipped, along with the overhead of any Python code performing this discovery.

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 using Python's default importer 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 using the standard Python importer 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.

A less contrived example is running the test harness for the Mercurial version control tool. Mercurial's test harness creates tens of thousands of new processes that start Python interpreters. So a few milliseconds of overhead starting interpreters or loading modules can translate to several seconds.

We run the full Mercurial test harness on Linux on a Ryzen 3950X CPU using the following variants:

- `hg` script with a `#!/path/to/python3.7` line (traditional)
- `hg` PyOxidizer executable using Python's standard filesystem import (oxidized)
- `hg` PyOxidizer executable using *filesystem-relative* resource loading (filesystem)
- `hg` PyOxidizer executable using *in-memory* resource loading (in-memory)

The results are quite clear:

Variant	CPU Time (s)	Delta (s)	% Orig
traditional	11,287	0	100
oxidized	10,735	-552	95.1
filesystem	10,186	-1,101	90.2
in-memory	9,883	-1,404	87.6

These results help us isolate specific areas of speedups:

- *oxidized* over *traditional* is a rough proxy for the benefits of `python -S` over `python`. Although there are other factors at play that may be influencing the numbers.
- *filesystem* over *oxidized* isolates the benefits of using PyOxidizer's importer instead of Python's default importer. The performance wins here are due to a) avoiding excessive I/O system calls to locate the paths to resources and b) functionality being implemented in Rust instead of Python.
- *in-memory* over *filesystem* isolates the benefits of avoiding explicit filesystem I/O to load Python resources. The Rust code backing these 2 variants is very similar. The only meaningful difference is that *in-memory* constructs a Python object from a memory address and *filesystem* must open and read a file using standard OS mechanisms before doing so.

From this data, one could draw a few conclusions:

- Processing of the `site` module during Python interpreter initialization can add substantial overhead.
- Maintaining an index of Python resources such that you can avoid discovery via filesystem I/O provides a meaningful speedup.
- Loading Python resources from an in-memory data structure is faster than incurring explicit filesystem I/O to do so.

5.10.2 Ignoring `site`

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!

5.11 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.

5.11.1 C and Other Native Extension Modules

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

PyOxidizer has varying levels of support for Python extension modules. In many cases, everything *just works*. But there are known incompatibilities and corner cases. See *Working with Python Extension Modules* for details.

5.11.2 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!')
```

5.11.3 Incorrect Resource Identification

PyOxidizer has custom code for scanning for and indexing files as specific Python resource types. This code is somewhat complex and nuanced and there are known bugs that will cause PyOxidizer to fail to identify or classify a file appropriately.

To help debug problems with this code, the `pyoxidizer find-resources` command can be employed. See *Debugging Resource Scanning and Identification with find-resources* for more.

Important: Please file a bug to report problems!

See *Classified Resources Versus Files* for more on this topic.

5.12 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`. Other applications are discouraged from defining this attribute.

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 `EmbbedPythonConfig` 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.

5.13 Standalone / Single File Applications with Static Linking

This document describes how to produce standalone, single file application binaries embedding Python using static linking.

See also *Working with Python Extension Modules* for extensive documentation about extension modules, which are often a pain point when it comes to static linking.

5.13.1 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. And newer versions of Rust may change which version of musl they use, introducing failures similar to above. If you run into problems with a modern version of Rust, consider reporting an issue against PyOxidizer!

Once Rust's musl target is installed, 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.

5.13.2 Building Statically Linked Binaries on Windows

It is possibly to produce a mostly self-contained `.exe` on Windows. We say *mostly* self-contained here because currently the built binary has some external `.dll` dependencies. However, these DLLs are core Windows / system DLLs and should be present on any Windows installation supported by the Python distribution being used.

The main trick to build a statically linked Windows binary is to switch the Python distribution from the default `standalone_dynamic` flavor to `standalone_static`. This can be done via the following in your config file:

```
def make_dist():
    return default_python_distribution(flavor = "standalone_static")
```

Important: The `standalone_static` Windows distributions build Python in a way that is incompatible with compiled Python extensions (`.pyd` files). So if you use this distribution flavor, you will need to compile all Python extensions from source and cannot use pre-built wheels packages. This can make building applications with many dependencies difficult, as many Python packages don't compile on Windows without installing many dependencies first.

See also *Windows Static Distributions Only Support Built-in Extension Modules*.

See also *Understanding Python Distributions* for more details on the differences between `standalone_dynamic` and `standalone_static` Python distributions.

5.13.3 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 very similar to that of containers (e.g. Docker images). If you are OK distributing and running 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*.

5.14 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.

Consider using a package such as pip-licenses to generate a license report for your Python packages.

5.14.1 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.

5.15 Terminfo Database

Note: This content 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* `PythonInterpreterConfig.terminfo_resolution` 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.

5.16 Using the tkinter Python Module

The tkinter Python standard library module/package provides a Python interface to tcl/tk/tkinter. This interface allows you to create GUI applications.

PyOxidizer has partial support for using tkinter. Since tkinter isn't a commonly used Python feature, you must opt in to enabling it.

5.16.1 Installing tcl Files

tkinter requires both a Python extension module compiled against tcl/tk and tcl support files to be loaded at run-time.

All the *built-in Python distributions* shipping with PyOxidizer provide tkinter support with the exception of the Windows standalone_static distributions.

However, the tcl support files aren't installed by default.

To install tcl support files, you will need to set the *PythonExecutable.tcl_files_path* attribute of a *PythonExecutable* instance to the directory you want to install these files into. e.g.

```
def make_exe(dist):
    exe = dist.to_python_executable(name="myapp")
    exe.tcl_files_path = "lib"

    return exe
```

When tcl_files_path is set to a non-None value, the tcl files required by tkinter are installed in that directory and the built executable will automatically set the TCL_LIBRARY environment variable at run-time so the tcl interpreter uses those files.

5.16.2 tcl Files Prevent Self-Contained Executables

The tcl interpreter needs to load various files off the filesystem at run-time. PyOxidizer does not (yet) support embedding these files in the binary and loading them from memory or extracting them at run-time.

So if you need to use tkinter, you cannot have a single-file executable that works without a dependency on tcl files elsewhere on the filesystem.

5.17 Building an Executable that Behaves Like `python`

It is possible to use PyOxidizer to build an executable that would behave like a typical `python` executable would.

To start, initialize a new config file:

```
$ pyoxidizer init-config-file python
```

Then, we'll want to modify the `pyoxidizer.bzl` configuration file to look something like the following:

```
def make_dist():
    return default_python_distribution()

def make_exe(dist):
    policy = dist.make_python_packaging_policy()
    policy.extension_module_filter = "all"
    policy.include_distribution_resources = True

    # Add resources to the filesytem, next to the built executable.
    # You can add resources to memory too. But this makes the install
    # layout somewhat consistent with what Python expects.
    policy.resources_location = "filesystem-relative:lib"

    python_config = dist.make_python_interpreter_config()

    # This is the all-important line to make the embedded Python interpreter
    # behave like `python`.
    python_config.config_profile = "python"

    # Enable the stdlib path-based importer.
    python_config.filesystem_importer = True

    # You could also disable the Rust importer if you really want your
    # executable to behave like `python`.
    # python_config.oxidized_importer = False

    exe = dist.to_python_executable(
        name="python3",
        packaging_policy = policy,
        config = python_config,
    )

    return exe

def make_embedded_resources(exe):
    return exe.to_embedded_resources()

def make_install(exe):
    files = FileManifest()
    files.add_python_resource(".", exe)

    return files

register_target("dist", make_dist)
register_target("exe", make_exe, depends=["dist"])
register_target("resources", make_embedded_resources, depends=["exe"], default_build_
↪script=True)
```

(continues on next page)

(continued from previous page)

```
register_target("install", make_install, depends=["exe"], default=True)

resolve_targets()
```

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

From there, build/run from the config:

```
$ cd python
$ pyoxidizer build
...
$ pyoxidizer run
...
Python 3.8.6 (default, Oct  3 2020, 20:48:20)
[Clang 10.0.1 ] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

5.17.1 Resource Loading Caveats

PyOxidizer's configuration defaults are opinionated about how resources are loaded by default. In the default configuration, the Python distribution's resources are indexed and loaded via `oxidized_importer` at run-time. This behavior is obviously different from what a standard `python` executable would do.

If you want the built executable to behave like `python` would and use the standard library importers, you can disable `oxidized_importer` by setting *oxidized_importer* to `False`.

Another caveat is that indexed resources are embedded in the built executable by default. This will bloat the size of the executable for no benefit. To disable this functionality, set *PythonExecutable.packed_resources_load_mode* to `none`.

5.17.2 Binary Portability

A `python`-like executable built with PyOxidizer may not *just work* when copied to another machine. See *Portability of Binaries Built with PyOxidizer* to learn more about the portability of binaries built with PyOxidizer.

CHAPTER 6

Distributing User Guide

This documentation covers how to *distribute* or *ship* applications with PyOxidizer.

6.1 Overview

Application *distribution* in PyOxidizer is fundamentally a separate domain from *building* or *packaging* applications. One way to think about this is *building* is concerned with producing files constituting your application - the executables and support files needed at run-time - and *distribution* is concerned with installing those files on other machines.

PyOxidizer uses the *Tugger* tool to handle most *distribution* functionality. Tugger is a Rust crate and Starlark dialect developed alongside PyOxidizer that specializes in functionality required to *distribute* applications. Tugger is technically a separate project. But PyOxidizer provides full access to Tugger's Starlark functionality and even extends it to make distributing Python applications simpler.

6.1.1 Using Tugger Starlark

Tugger defines a Starlark dialect that enables you to produce distributable artifacts. See *Tugger Starlark Dialect* for the documentation of this dialect.

The full Tugger Starlark dialect is available to PyOxidizer configuration files.

PyOxidizer configuration files have the option of using the generic Tugger Starlark primitives and using supplemental/extended functionality provided by PyOxidizer's Starlark dialect. The Tugger-provided primitives are generally low-level and generic. The PyOxidizer-provided extensions are Python specific and may allow simpler configuration files.

See other documentation in *Distributing User Guide* for details on PyOxidizer's extensions to Tugger's Starlark dialect and how to perform common *distribution* actions.

6.2 Portability of Binaries Built with PyOxidizer

Binary portability refers to the property that a binary built in machine/environment *X* is able to run on machine/environment *Y*. In other words, you've achieved binary portability if you are able to copy a binary to another machine and run it without modifications.

It is exceptionally difficult to achieve high levels of binary portability for various reasons.

PyOxidizer is capable of building binaries that are highly *portable*. However, the steps for doing so can be nuanced and vary substantially by operating system and target platform.

This document outlines some general strategies for tackling binary portability. Please also consult the various platform-specific documentation on this topic:

- *Distribution Considerations for Linux*
- *Distribution Considerations for macOS*
- *Distribution Considerations for Windows*

Important: Please create issues at https://github.com/indygreg/PyOxidizer/issues when documentation on this subject is inaccurate or lacks critical details.

6.2.1 Using `pyoxidizer analyze` For Assessing Binary Portability

The `pyoxidizer analyze` command can be used to analyze the contents of executables and libraries. It can be used as a PyOxidizer-specific tool for assessing the portability of built binaries.

For example, for ELF binaries (the binary format used on Linux), this command will list all shared library dependencies and analyze glibc symbol versions and print out which Linux distribution versions it thinks the binary is compatible with.

Note: `pyoxidizer analyze` is not yet feature complete on all platforms.

6.3 Building Windows Installers with the WiX Toolset

PyOxidizer supports building Windows installers (e.g. `.msi` and `.exe` installer files) using the WiX Toolset. PyOxidizer leverages the *Tugger shipping tool* for integrating with WiX. See *Using the WiX Toolset to Produce Windows Installers* for the full Tugger WiX documentation.

Tugger - and PyOxidizer by extension - are able to automatically create XML files used by WiX to define installers with common features as well as use pre-existing WiX files. This enables Tugger/PyOxidizer to facilitate both simple and arbitrarily complex use cases.

6.3.1 Extensions to Tugger Starlark Dialect

PyOxidizer supplements Tugger's Starlark dialect with additional functionality that makes building Python application installers simpler. For example, instead of manually constructing a WiX installer, you can call a method on a Python Starlark type to convert it into an installer.

PyOxidizer provides the following extensions and integrations with *Tugger's Starlark dialect*:

FileManifest.add_python_resource() Adds a Python resource type to Tugger's *FileManifest* type.

FileManifest.add_python_resources() Adds an iterable of Python resource types to Tugger's *FileManifest* type.

PythonExecutable.to_file_manifest() Converts a *PythonExecutable* to a *FileManifest*. Enables materializing an executable/application as a set of files, which Tugger can easily operate against.

PythonExecutable.to_wix_bundle_builder() Converts a *PythonExecutable* to a *WiXBundleBuilder*.

This method will produce a *WiXBundleBuilder* that is pre-configured with appropriate settings and state for a Python application. The produced `.exe` installer should *just work*.

PythonExecutable.to_wix_msi_builder() Converts a *PythonExecutable* to a *WiXMSIBuilder*.

This method will produce a *WiXMSIBuilder* that is pre-configured to install a Python application and all its support files. The MSI will install all files composing the Python application, excluding system-level dependencies.

6.3.2 Choosing an Installer Creation Method

Tugger provides multiple Starlark primitives for defining Windows installers built with the WiX Toolset. Which one should you use?

See *Tugger's WiX APIs* for a generic overview of this topic. The remainder of this documentation will be specific to Python applications.

It is is important to call out that unless you are using the *static Python distributions*, binaries built with PyOxidizer will have a run-time dependency on the Visual C++ Redistributable runtime DLLs (e.g. `vcruntime140.dll`). Many Windows applications have a dependency on these DLLs and most Windows machines have installed an application that has installed the required DLLs. So not distributing `vcruntimeXXX.dll` with your application may *just work* most of the time. However, on a fresh Windows installation, these required files may not exist. So it is important that they be installed with your application.

When using *PythonExecutable.to_wix_msi_builder()* or *PythonExecutable.to_wix_bundle_builder()*, PyOxidizer will automatically add the Visual C++ Redistributable to the installer if it is required. However, the method varies. For bundle installers, the installer will contain the official `VC_Redist*.exe` installer and this installer will be executed as part of running your application's installer. For MSI installers, Tugger will attempt to locate the `vcruntimeXXX.dll` files on your system (this requires an installation of Visual Studio) and copy these files next to your built/installed executable.

If you are not using one of the aforementioned APIs to create your installer, you will need to explicitly add the Visual C++ Redistributable to your installer. The *WiXMSIBuilder.add_visual_cpp_redistributable()* and *WiXBundleBuilder.add_vc_redistributable()* Starlark methods can be called to do this. (PyOxidizer's Starlark methods for creating WiX installers effectively call these methods.)

6.4 Distribution Considerations for Linux

This document describes some of the considerations when you want to install/run a PyOxidizer-built application on a separate Linux machine from the one that built it.

6.4.1 Exception for musl libc Binaries

Linux binaries built against musl libc (e.g. the `x86_64-unknown-linux-musl` target triple) generally work on any Linux machine supporting the target architecture. This is because musl libc linked binaries are fully statically linked and therefore self-contained.

If you run `ldd /path/to/binary` and it prints `not a dynamic executable`, that binary is likely highly portable.

See *Building Fully Statically Linked Binaries on Linux* for instructions on building binaries with musl libc.

The rest of this document likely doesn't apply if using musl libc.

6.4.2 Python Distribution Dependencies

The default *Python distributions* used by PyOxidizer have dependencies on shared libraries outside of the Python distribution.

However, the python-build-standalone project - the entity building the default Python distributions - has gone to great lengths to ensure that all dependencies are common to nearly every Linux system and that the Python distribution binaries should be highly portable across machines.

The `*-unknown-linux-gnu` builds have a dependency against GNU libc (glibc), specifically `libc.so.6`. However, the python-build-standalone project has build-time validation that glibc version numbers in referenced symbols aren't higher than glibc 19 (released in 2014). This should make binaries compatible with the following common distributions:

- Fedora 21+
- RHEL/CentOS 7+
- openSUSE 13.2+
- Debian 8+ (Jessie)
- Ubuntu 14.04+

In addition to glibc, Python distributions also link to a handful of other system libraries. Most of the libraries are part of the Linux Standard Base specification and should be present on any conforming Linux distribution.

Some shared library dependencies are only pulled in by single Python extensions. For example, `libcrypto.so.1` is likely only needed by the `crypt` extension. Distributors wanting to minimize the number of shared library dependencies can do so by pruning Python extensions from the install set. The `PYTHON.json` file in the extracted Python distribution archive can be used to inspect which libraries are required by which extensions.

6.4.3 Built Application Dependencies

While the default Python distributions used by PyOxidizer are highly portable, the same cannot be said for binaries built with PyOxidizer.

Important: The machine and environment you use to run `pyoxidizer` has critical implications for the portability of built binaries.

When you use PyOxidizer to produce a new binary (an executable or library), you are compiling *new* code and linking it in an environment that is different from the specialized environment used to build the default Python distributions. This often means that the binary portability of your built binary is effectively defined by the environment `pyoxidizer` was run from.

As a concrete example, if you run `pyoxidizer build` on an Ubuntu 20.10 machine and then `pyoxidizer analyze` the resulting ELF binary, you'll find that it has a dependency on `libgcc_s.so.1` and it references glibc 2.32 symbol versions. This despite the default Python distribution not depending on *libgcc_s.so.1*' and only glibc version 2.19.

What's happening here is the compiler/build settings from the building machine are *leaking* into new binaries, likely as part of compiling Rust code.

6.4.4 Managing Binary Portability on Linux

Linux is a difficult platform to tackle for binary portability.

The best way to produce a portable Linux binary is to produce a fully statically-linked binary. There are no shared libraries to worry about and generally speaking these binaries *just work*. See *Building Fully Statically Linked Binaries on Linux* for more.

If you produce a dynamic binary with library dependencies, things are complicated.

Nearly every binary built on Linux will require linking against `libc` and will require a symbol provided by `glibc`. `glibc` versions it symbols. And when the linker resolves those symbols at link time, it usually uses the version of `glibc` being linked against. For example, if you link on a machine with `glibc` 2.19, the symbol versions in the produced binary will be against version 2.19 and the binary will load against `glibc` versions >=2.19. But if you link on a machine with `glibc` 2.29, symbol versions are against version 2.29 and you can only load against versions >= 2.29.

This means that to ensure maximum portability, you want to link against old `glibc` symbol versions. While it is possible to use old symbol versions when a more modern `glibc` is present, the path of least resistance is to build in an environment that has an older `glibc`.

A similar story plays out with a dependency on `libgcc_s.so.1`.

The default Python distributions use Debian 8 (Jessie) as their build environment. So a Debian 8 build environment is a good candidate to build on. Ubuntu 14.04, OpenSUSE 13.2, OpenSUSE 42.1, RHEL/CentOS 7, and Fedora 21 (`glibc` 2.20) are also good candidates for build environments.

Of course, if you are producing distribution-specific binaries and/or control installation (so e.g. dependencies are installed automatically), this matters less to you.

The `pyoxidizer analyze` command can be very useful for inspecting binaries for portability and alerting you to any potential issues.

6.5 Distribution Considerations for macOS

This document describes some of the considerations when you want to install/run a PyOxidizer-built application on a separate macOS machine from the one that built it.

6.5.1 Operating System and Architecture Requirements

PyOxidizer has support for targeting x86_64 (Intel) and aarch64 (ARM) Apple devices. The default *Python distributions* target macOS 10.9+ for x86_64 and 11.0+ for aarch64.

6.5.2 Build Machine Requirements

PyOxidizer needs to link new binaries containing Python. Due to the way linking works on Apple platforms, you **must** use an Apple SDK no older than the one used to build the Python distributions or linker errors (likely undefined symbols) can occur.

PyOxidizer will automatically attempt to locate, validate, and use an appropriate Apple SDK given requirements specified by the Python distribution in use. If you have Xcode or the Xcode Commandline Tools installed, PyOxidizer

should be able to locate Apple SDKs automatically. When building, PyOxidizer will print information about Apple SDK discovery. More details are printed when running `pyoxidizer --verbose`.

PyOxidizer will automatically look for SDKs in the directory specified by `xcode-select --print-path`. This path is often `/Applications/Xcode.app/Contents/Developer`. You can specify an alternative directory by setting the `DEVELOPER_DIR` environment variable. e.g.:

```
DEVELOPER_DIR=/Applications/Xcode-beta.app/Contents/Developer pyoxidizer build
```

You can override PyOxidizer's automatic SDK discovery by setting `SDKROOT` to the base directory of an Apple SDK you want to use. (If you find yourself doing this to work around SDK discovery *bugs*, please consider creating a GitHub issue to track the problem.) e.g.:

```
SDKROOT=/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/
↪Developer/SDKs/MacOSX.sdk pyoxidizer build
```

6.5.3 Python Distribution Dependencies

The default *Python distributions* used by PyOxidizer have dependencies on system libraries outside of the Python distribution.

The python-build-standalone project has gone to great lengths to ensure that the Python distributions only link against external libraries and symbols that are present on a default macOS installation.

The default Python distributions are built to target macOS 10.9 on x86_64 and 11.0 on aarch64. So they should *just work* on those and any newer versions of macOS.

6.5.4 Single Architecture Binaries

PyOxidizer currently only emits single architecture binaries.

Multiple architecture binaries (often referred to as *universal* or *fat* binaries) can not (yet) be emitted natively by PyOxidizer.

This means that if you distribute a binary produced by PyOxidizer and want it to run on both Intel and ARM machines, you will need to maintain separate artifacts for Intel and ARM machines or you will need to produce a *fat* binary outside of PyOxidizer.

https://github.com/indygreg/PyOxidizer/issues/372 tracks implementing support for emitting *fat* binaries from PyOxidizer. Please engage there if this feature is important to you.

6.5.5 Managing Portability of Built Applications

Like Linux, the macOS build environment can *leak* into the built application and introduce additional dependencies and degrade the portability of the default Python distributions.

It is common for built binaries to pull in modern macOS SDK features. A common way to prevent this is to set the `MACOSX_DEPLOYMENT_TARGET` environment variable during the build to the oldest version of macOS you want to support.

The default *Python distributions* target macOS 10.9 on x86_64 and 11.0 on aarch64.

Important: PyOxidizer will automatically set the deployment target to match what the Python distribution was built with, so in many cases you don't need to worry about version targeting.

If you wish to override the default deployment targets, set an alternative value using the appropriate environment variable.:

```
$ MACOSX_DEPLOYMENT_TARGET=10.15 pyoxidizer build
```

Apple's Xcode documentation has various guides useful for further consideration.

6.6 Distribution Considerations for Windows

This document describes some of the considerations when you want to install/run a PyOxidizer-built application on a separate Windows machine from the one that built it.

Important: The restrictions in this document regard the run-time / target environment that a binary will run on: they do not describe the environment used to build that binary. In many cases, a binary built on Windows 10 or Windows Server 2019 will work fine on earlier operating system versions.

Readers may also find the Microsoft documentation on deployment considerations for Windows binaries a useful resource to supplement this document with more generic considerations.

6.6.1 Operating System Requirements

The default *Python distributions* used by PyOxidizer require Windows 8 or Windows 2012 or newer.

The official Python 3.8 Windows distributions available on www.python.org support Windows 7. PyOxidizer has chosen to drop support for Windows 7 to simplify support.

In addition to the restrictions imposed by the Python distribution in use, Rust may impose its own restrictions. However, Rust has historically produced binaries that work on Windows 8 and Windows 2012, so this likely is not an issue.

6.6.2 General Runtime / DLL Dependencies

The default *Python distributions* used by PyOxidizer require the Microsoft Visual C++ Redistributable and Universal CRT (UCRT).

The `standalone_dynamic` distributions (the default distribution flavor) have a run-time dependency on various 3rd party DLLs used by extensions (OpenSSL, SQLite3, etc). However, these 3rd party DLLs are part of the Python distribution and PyOxidizer should automatically install them if they are required.

All other DLL dependencies required by the default Python distributions should be core Windows operating system components and always available, even in a freshly installed Windows machine.

6.6.3 Application Specific Dependencies

When adding custom behavior to your application, PyOxidizer makes some effort to ensure additional dependencies (beyond the operating system, Python distribution, and Microsoft runtimes) are met. However, there are limitations to this.

When installing custom Python packages, PyOxidizer attempts to identify and install compiled Python extensions and `.dll` dependencies distributed with that package. See *Packaging Files Instead of In-Memory Resources* for more. However, there are corner cases and occasional bugs that may prevent this from working correctly.

To ensure are DLL dependencies are properly captured, it is recommend to inspect your binaries for references to missing DLLs before distributing them. The Dependency Walker tool can be used for this. `pyoxidizer analyze` may also provide useful information.

In many cases, installing a missing DLL is a matter of installing the DLL next to your application/binary by treating the DLL as an *additional file* from the Starlark configuration. See *Packaging Files Instead of In-Memory Resources* for more.

When possible, it is recommended to test your application in a freshly installed Windows environment to ensure it works. Please note that many Windows virtual machines already contain additional software and may not reflect real world deployment targets.

6.6.4 Managing the Visual C++ Redistributable Requirement

Binaries built with PyOxidizer often have a run-time dependency on the Microsoft Visual C++ Redistributable. These are DLLs with filenames like `vcruntime140.dll` and `vcruntime140_1.dll`.

Important: The Visual C++ Redistributable is **not** a core Windows operating system component and any distributed Windows application **must take measures to ensure the Visual C++ Redistributable is available on the remote machine** or the application may fail to run with a missing DLL error.

See Microsoft's Redistributing Visual C++ Files documentation for the canonical source on distribution requirements.

PyOxidizer has built-in features to make satisfying these requirements turnkey. Read the sections below for details of each.

Installing the Visual C++ Redistributable as Part of Your Application Installer

PyOxidizer can produce Windows `.exe` application installers that embed a copy of the Microsoft Visual C++ Redistributable installer (files named `vc_redist<arch>.exe`) and automatically run this installer during application install.

The way this works is PyOxidizer contains a reference to the URL and SHA-256 of these `vc_redist<arch>.exe` installers. When your application installer is built, these files are downloaded from Microsoft's servers and embedded in the new meta-installer. At install time, these embedded installers are executed automatically (if they need to be) and the Visual C++ files are installed at the system level, where they are available to any application.

If a newer version of the Visual C++ Redistributable files are already present, the installer should no-op instead of downgrading what's already installed.

The following Starlark functionality can be used to bundle the Visual C++ Redistributable installer as part of your application installer:

- *PythonExecutable.to_wix_bundle_builder()*
- *WiXBundleBuilder.add_vc_redistributable()*

Installing the Visual C++ Redistributable Files Locally Next to Your Binary

Another method of installing the Visual C++ Redistributable files is to distribute copies of the DLLs next to the binary that loads them. e.g. if you produce a `myapp.exe`, there will be a `vcruntime140[_1].dll` in the same directory as `myapp.exe`. Since Windows attempts to load DLLs next to the executable, if the DLLs are present, this should *just work*.

PyOxidizer supports automatically finding and copying the required DLLs in this manner. The Starlark setting controlling this behavior is *PythonExecutable.windows_runtime_dlls_mode*.

This setting effectively instructs the `PythonExecutable` building code to materialize extra files next to the binary. The Visual C++ files are treated just like any other supplementary files (like Python resources). This means that Visual C++ files will be materialized on the filesystem when running `pyoxidizer build`, `pyoxidizer run`. The files will also be present in file lists when using Starlark methods like *PythonExecutable.to_file_manifest()* or *PythonExecutable.to_wix_msi_builder()*.

This *local files* mode relies on locating DLLs on the local system. It does so using `vswhere.exe` to locate a Visual Studio installation containing the `Microsoft.VisualCPP.Redist.<version>.Latest` component (`<version>` is `14` for `vcruntime140.dll`). This should *just work* if you have Visual Studio 2017 or 2019 installed with support for building C/C++ applications. If the files cannot be found, run the Visual Studio Installer, `Modify` your installation, go to `Individual Components`, search for `redistributable`, and make sure all items are checked.

Important: It is possible to include a copy of the Visual C++ Redistributable in both your application installer and as files local to the built binary. This behavior is redundant and will likely result in the local files being used.

When including the Visual C++ Redistributable installer as part of your deployment solution, it is recommended to set `PythonExecutable.windows_runtime_dlls_mode = "never"` to prevent them from being redundantly installed.

6.6.5 Managing the Universal CRT (UCRT) Requirement

Binaries built with PyOxidizer may have a run-time dependency on the Universal C Runtime (UCRT).

The UCRT is a Windows operating system component and is always present in installations of Windows 10, Windows Server 2016, and newer. Combined with PyOxidizer's Windows version requirements, this means you don't need to worry about the UCRT unless you are targeting Windows 8 or Windows Server 2012.

PyOxidizer does not currently support automatically materializing the UCRT. See https://docs.microsoft.com/en-us/cpp/windows/universal-crt-deployment for instructions on deploying the UCRT with your application.

We are receptive to adding a feature to support more turnkey UCRT management if there is interest in it.

CHAPTER 7

oxidized_importer Python Extension

oxidized_importer is a Python extension module maintained as part of the PyOxidizer project that allows you to:

- Install a custom, high-performance module importer (OxidizedFinder) to service Python import statements and resource loading (potentially from memory).
- Scan the filesystem for Python resources (source modules, bytecode files, package resources, distribution metadata, etc) and turn them into Python objects.
- Serialize Python resource data into an efficient binary data structure for loading into an OxidizedFinder instance. This facilitates producing a standalone *resources blob* that can be distributed with a Python application which contains all the Python modules, bytecode, etc required to power that application.

oxidized_importer is automatically compiled into applications built with PyOxidizer. It can also be built as a standalone extension module and used with regular Python installs.

7.1 Getting Started

7.1.1 Requirements

oxidized_importer requires CPython 3.8 or newer. This is because it relies on modern C and Python standard library APIs only available in that version.

Building oxidized_importer from source requires a working Rust toolchain for the target platform.

7.1.2 Installing from PyPI

oxidized_importer is available on PyPI. This means that installing is as simple as:

```
$ pip3 install oxidized_importer
```

7.1.3 Compiling from Source

To build from source, obtain a clone of PyOxidizer's Git repository and run the `setup.py` script or use `pip` to build the Python project in the root of the repository. e.g.:

```
$ python3.9 setup.py build_ext -i
$ python3.9 setup.py install

$ pip3.9 install .
$ pip3.9 wheel .
```

The `setup.py` is pretty minimal and is a thin wrapper around `cargo build` for the underlying Rust project. If you want to build using Rust's standard toolchain, do something like the following:

```
$ cd oxidized-importer
$ cargo build --release
```

If you don't have a Python 3.9 `python3` executable in your `PATH`, you will need to tell the Rust build system which `python3` executable to use to help derive the build configuration for the Python extension:

```
$ PYTHON_SYS_EXECUTABLE=/path/to/python3.9 cargo build
```

7.1.4 Using

To use `oxidized_importer`, simply import the module:

```
import oxidized_importer
```

To register a custom importer with Python, do something like the following:

```
import sys

import oxidized_importer

finder = oxidized_importer.OxidizedFinder()

# You want to register the finder first so it has the highest priority.
sys.meta_path.insert(0, finder)
```

To get performance benefits of loading modules and resources from memory, you'll need to index resources with the `OxidizedFinder`, serialize that data out, then load that data into a new `OxidizedFinder` instance. See *Freezing Applications with oxidized_importer* for more detailed examples.

7.2 Python Meta Path Finders

Python allows providing custom Python types to handle the low-level machinery behind the `import` statement. The way this works is a *meta path finder* instance (as defined by the importlib.abc.MetaPathFinder interface) is registered on sys.meta_path. When an `import` is serviced, Python effectively iterates the objects on `sys.meta_path` and asks each one *can you service this request* until one does.

These *meta path finder* not only service basic Python module loading, but they can also facilitate loading resource files and package metadata. There are a handful of optional methods available on implementations.

This documentation will often refer to a *meta path finder* as an *importer*, because it is primarily used for *importing* Python modules.

Normally when you start a Python process, the Python interpreter itself will install 3 *meta path finders* on `sys.meta_path` before your code even has a chance of running:

`BuiltinImporter` Handles importing of *built-in* extension modules, which are compiled into the Python interpreter. These include modules like `sys`.

`FrozenImporter` Handles importing of *frozen* bytecode modules, which are compiled into the Python interpreter. This *finder* is typically only used to initialize Python's importing mechanism.

`PathFinder` Handles filesystem-based loading of resources. This is what is used to import `.py` and `.pyc` files. It also handles `.zip` files. This is the *meta path finder* that most imports are traditionally serviced by. It queries the filesystem at `import` time to find and load resources.

7.3 `OxidizedFinder` Python Type

`oxidized_importer.OxidizedFinder` is a Python type that implements a custom *meta path finder*. *Oxidized* is in its name because it is implemented in Rust.

Unlike traditional *meta path finders* which have to dynamically discover resources (often by scanning the filesystem), `OxidizedFinder` instances maintain an *index* of known resources. When a resource is requested, `OxidizedFinder` can retrieve that resource by effectively performing 1 or 2 lookups in a Rust `HashMap`. This makes resource resolution extremely efficient.

Instances of `OxidizedFinder` are optionally bound to a binary blob holding *packed resources data*. This is a custom serialization format for expressing Python modules (source and bytecode), Python extension modules, resource files, shared libraries, etc. This data format along with a Rust library for interacting with it are defined by the python-packed-resources crate.

When an `OxidizedFinder` instance is created, the *packed resources data* is parsed into a Rust data structure. On a modern machine, parsing this resources data for the entirety of the Python standard library takes ~1 ms.

`OxidizedFinder` instances can index *built-in* extension modules and *frozen* modules, which are compiled into the Python interpreter. This allows `OxidizedFinder` to subsume functionality normally provided by the `BuiltinImporter` and `FrozenImporter` *meta path finders*, allowing you to potentially replace `sys.meta_path` with a single instance of `OxidizedFinder`.

7.3.1 `OxidizedFinder` in PyOxidizer Applications

When running from an application built with PyOxidizer (or using the `pyembed` crate directly), an `OxidizedFinder` instance will (likely) be automatically registered as the first element in `sys.meta_path` when starting a Python interpreter.

You can verify this inside a binary built with PyOxidizer:

```
>>> import sys
>>> sys.meta_path
[<OxidizedFinder object at 0x7f16bb6f93d0>]
```

Contrast with a typical Python environment:

```
>>> import sys
>>> sys.meta_path
[
    <class '_frozen_importlib.BuiltinImporter'>,
    <class '_frozen_importlib.FrozenImporter'>,
```

(continues on next page)

(continued from previous page)

```
    <class '_frozen_importlib_external.PathFinder'>
]
```

The `OxidizedFinder` instance will (likely) be associated with resources data embedded in the binary.

This `OxidizedFinder` instance is constructed very early during Python interpreter initialization. It is registered on `sys.meta_path` before the first `import` requesting a `.py`/`.pyc` is performed, allowing it to service every `import` except those from the very few *built-in extension modules* that are compiled into the interpreter and loaded as part of Python initialization (e.g. the `sys` module).

At the end of initialization, the *path_hook* method of the `OxidizedFinder` instance on `sys.meta_path` is appended to `sys.path_hooks` if both the `OxidizedFinder` and standard library filesystem based importer are enabled.

7.3.2 Python API

`OxidizedFinder` instances implement the following interfaces:

- `importlib.abc.MetaPathFinder`
- `importlib.abc.Loader`
- `importlib.abc.InspectLoader`
- `importlib.abc.ExecutionLoader`

See the importlib.abc documentation for more on these interfaces.

In addition to the methods on the above interfaces, the following methods defined elsewhere in `importlib` are exposed:

- `get_resource_reader(fullname: str) -> importlib.abc.ResourceReader`
- `find_distributions(context: Optional[DistributionFinder.Context]) -> [Distribution]`

`ResourceReader` is documented alongside other `importlib.abc` interfaces. `find_distribution()` is documented in importlib.metadata.

7.3.3 Non-`importlib` API

`OxidizedFinder` instances have additional functionality beyond what is defined by `importlib`. This functionality allows you to construct, inspect, and manipulate instances.

`__new__(cls, ...)`

New instances of `OxidizedFinder` can be constructed like normal Python types:

```
finder = OxidizedFinder()
```

The constructor takes the following named arguments:

`relative_path_origin` A path-like object denoting the filesystem path that should be used as the *origin* value for relative path resources. Filesystem-based resources are stored as a relative path to an *anchor* value. This is that *anchor* value. If not specified, the directory of the current executable will be used.

See the python_packed_resources Rust crate for the specification of the binary data blob defining *packed resources data*.

Important: The *packed resources data* format is still evolving. It is recommended to use the same version of the `oxidized_importer` extension to produce and consume this data structure to ensure compatibility.

`index_bytes(self, data: bytes) -> None`

This method parses any bytes-like object and indexes the resources within.

`index_file_memory_mapped(self, path: Path) -> None`

This method parses the given Path-like argument and indexes the resources within. Memory mapped I/O is used to read the file. Rust managed the memory map via the `memmap` crate: this does not use the Python interpreter's memory mapping code.

`index_interpreter_builtins(self) -> None`

This method indexes Python resources that are built-in to the Python interpreter itself. This indexes built-in extension modules and frozen modules.

`index_interpreter_builtin_extension_modules(self) -> None`

This method will index Python extension modules that are compiled into the Python interpreter itself.

`index_interpreter_frozen_modules(self) -> None`

This method will index Python modules whose bytecode is frozen into the Python interpreter itself.

`indexed_resources(self) -> List[OxidizedResource]`

This method returns a list of resources that are indexed by the `OxidizedFinder` instance. It allows Python code to inspect what the finder knows about.

See *OxidizedResource* for more on the returned type.

`add_resource(self, resource: OxidizedResource)`

This method registers an *OxidizedResource* instance with the finder, enabling the finder to use it to service lookups.

When an `OxidizedResource` is registered, its data is copied into the finder instance. So changes to the original `OxidizedResource` are not reflected on the finder. (This is because `OxidizedFinder` maintains an index and it is important for the data behind that index to not change out from under it.)

Resources are stored in an invisible hash map where they are indexed by the `name` attribute. When a resource is added, any existing resource under the same name has its data replaced by the incoming `OxidizedResource` instance.

If you have source code and want to produce bytecode, you can do something like the following:

```
def register_module(finder, module_name, source):
    code = compile(source, module_name, "exec")
    bytecode = marshal.dumps(code)

    resource = OxidizedResource()
    resource.name = module_name
    resource.is_module = True
    resource.in_memory_bytecode = bytecode
    resource.in_memory_source = source

    finder.add_resource(resource)
```

add_resources(self, resources: List[OxidizedResource])

This method is syntactic sugar for calling `add_resource()` for every item in an iterable. It is exposed because function call overhead in Python can be non-trivial and it can be quicker to pass in an iterable of `OxidizedResource` than to call `add_resource()` potentially hundreds of times.

serialize_indexed_resources(self, ...) -> bytes

This method serializes all resources currently indexed by the instance into an opaque `bytes` instance. The returned data can be fed into a separate `OxidizedFinder` instance by passing it to *__new__(cls, ...)*.

Arguments:

ignore_builtin (bool) Whether to ignore `builtin` extension modules from the serialized data.

Default is `True`

ignore_frozen (bool) Whether to ignore `frozen` extension modules from the serialized data.

Default is `True`.

Entries for *built-in* and *frozen* modules are ignored by default because they aren't portable, as they are compiled into the interpreter and aren't guaranteed to work from one Python interpreter to another. The serialized format does support expressing them. Use at your own risk.

path_hook(path: Union[str, bytes, os.PathLike[AnyStr]]) -> OxidizedPathEntryFinder

When `path_hook`, bound to an `OxidizedFinder` instance `self`, is in `sys.path_hooks`, `pkgutil.iter_modules` can search `self`'s embedded resources, filtering by its `path` argument. Additionally, if you add `sys.executable` to `sys.path`, the meta-path finder `importlib.machinery.PathFinder` can find `self`'s embedded resources.

`path`'s semantics match those of *__path__ Module Attribute*. After normalization, `path` must be or be in `sys.executable`; otherwise `path_hook` raises an `ImportError`. If `path` is `sys.executable`, top-level modules are accessible. Otherwise `path_hook` computes the requested package by stripping `sys.executable` from the beginning of `path` and replacing path separators with dots. The result is decoded to a `str` using the filesystem encoding. If that fails, `path_hook` raises an `ImportError` from the `UnicodeDecodeError`.[1]

class oxidized_finder.**OxidizedPathEntryFinder**

A path-entry finder that can find modules embedded in an `OxidizedFinder` instance by searching paths at or under `sys.executable`. Each *OxidizedPathEntryFinder* instance is associated

[1] This is required by the path-entry finder protocol.

with the `path` argument to *`OxidizedPathEntryFinder`*'s only constructor, *OxidizedFinder.path_hook*. Only modules embedded in the `OxidizedFinder` instance in the top level of the path are *visible* to the *`OxidizedPathEntryFinder`* instance. For example, if `path` were `os.path.join(sys.executable, 'a')`, then module `a.b` would be visible, but neither modules `a` nor `a.b.c` would be visible. Further, `a.b` would be visible only if it were embedded in the `OxidizedFinder` instance that constructed the instance.

This class complies with the path-entry finder protocol by providing compliant *`find_spec()`* and *`invalidate_caches()`* methods. However, support for the long-deprecated methods `importlib.abc.PathEntryFinder.find_loader` and `importlib.abc.PathEntryFinder.find_module` may be missing or incomplete.

Direct use of *`OxidizedPathEntryFinder`* is generally unnecessary. It exists primarily to support `pkgutil.iter_modules` via *OxidizedFinder.path_hook*.

`find_spec`(*fullname: str*, *target: Optional[types.ModuleType] = None*) → Optional[importlib.machinery.ModuleSpec]
Search for modules visible to the instance.

`invalidate_caches`() → None
Invoke the same method on the `OxidizedFinder` instance with which the *`OxidizedPathEntryFinder`* instance was constructed.

`iter_modules`(*prefix: str = ""*) → List[pkgutil.ModuleInfo]
Iterate over the visible modules. This method complies with `pkgutil.iter_modules`'s protocol.

7.4 `OxidizedFinder` Behavior and Compliance

`OxidizedFinder` strives to be as compliant as possible with other *meta path importers*. So generally speaking, the behavior as described by the importlib documentation should be compatible. In other words, things should mostly *just work* and any deviance from the `importlib` documentation constitutes a bug worth reporting.

That being said, `OxidizedFinder`'s approach to loading resources is drastically different from more traditional means, notably loading files from the filesystem. PyOxidizer breaks a lot of assumptions about how things have worked in Python and there is some behavior that may seem odd or in violation of documented behavior in Python.

The sections below attempt to call out known areas where `OxidizedFinder` deviates from typical behavior.

7.4.1 `__file__` and `__cached__` Module Attributes

Python modules typically have a `__file__` attribute holding a `str` defining the filesystem path the source module was imported from (usually a path to a `.py` file). There is also the similar - but lesser known - `__cached__` attribute holding the filesystem path of the bytecode module (usually the path to a `.pyc` file).

Important: `OxidizedFinder` will not set either attribute when importing modules from memory.

These attributes are not set because it isn't obvious what the values should be! Typically, `__file__` is used by Python as an anchor point to derive the path to some other file. However, when loading modules from memory, the traditional filesystem hierarchy of Python modules does not exist. In the opinion of PyOxidizer's maintainer, exposing `__file__` would be *lying* and this would cause more potential for harm than good.

While we may make it possible to define `__file__` (and `__cached__`) on modules imported from memory someday, we do not yet support this.

`OxidizedFinder` does, however, set `__file__` and `__cached__` on modules imported from the filesystem. So, a workaround to restore these missing attributes is to avoid in-memory loading.

Note: Use of `__file__` is commonly encountered in code loading *resource files*. See *Loading Resource Files* for more on this topic, including how to port code to more modern Python APIs for loading resources.

7.4.2 `__path__` Module Attribute

Python modules that are also packages must have a `__path__` attribute containing an iterable of `str`. The iterable can be empty.

If a module is imported from the filesystem, `OxidizedFinder` will set `__path__` to the parent directory of the module's file, just like the standard filesystem importer would.

If a module is imported from memory, `__path__` will be set to the path of the current executable joined with the package name. e.g. if the current executable is `/usr/bin/myapp` and the module/package name is `foo.bar`, `__path__` will be `["/usr/bin/myapp/foo/bar"]`. On Windows, paths might look like `C:\dev\myapp.exe\foo\bar`.

Python's `zipimport` importer uses the same approach for modules imported from zip files, so there is precedence for `OxidizedFinder` doing things this way.

7.4.3 Support for `__init__` in Module Names

There exists Python code that does things like `from .__init__ import X`.

`__init__` is special in Python module names because it is the filename used to denote a Python package's filename. So syntax like `from .__init__ import X` is probably intended to be equivalent to `from . import X`. Or `import foo.__init__` is probably intended to be written as `import foo`.

Python's filesystem importer doesn't treat `__init__` in module names as special. If you attempt to import a module named `foo.__init__`, it will attempt to locate a file named `foo/__init__.py`. If that module is a package, this will succeed. However, the module name seen by the importer has `__init__` in it and the name on the created module object will have `__init__` in it. This means that you can have both a module `foo` and `foo.__init__`. These will both be derived from the same file but are actually separate module objects.

PyOxidizer will automatically remove trailing `.__init__` from module names. This will enable PyOxidizer to work with syntax such as `import foo.__init__` and `from .__init__ import X` and therefore be compatible with Python code in the wild. However, PyOxidizer may not preserve the `.__init__` in the module name. For example, with Python's path based importer, you could have both `foo` and `foo.__init__` in `sys.modules` but PyOxidizer will only have `foo`.

A limitation of PyOxidizer module name normalization is it only normalizes the single trailing `.__init__` from the module name: `__init__` appearing inside the module name are not normalized. e.g. `foo.__init__.bar` is not normalized to `foo.bar`. This may introduce incompatibilities with Python code in the wild. However, for this to be true, the filesystem layout would have to be something like `foo/__init__/bar.py`. This hopefully does not occur in the wild. But it is conceivable it does.

See https://github.com/indygreg/PyOxidizer/issues/317 and https://bugs.python.org/issue42564 for more discussion on this issue.

7.4.4 `ResourceReader` Compatibility

`ResourceReader` has known compatibility differences with Python's default filesystem-based importer. See *Support for ResourceReader* for details.

7.4.5 `ResourceLoader` Compatibility

The `ResourceLoader` interface is implemented but behavior of `get_data(path)` has some variance with Python's filesystem-based importer.

See *Support for ResourceLoader* for details.

Note: `ResourceLoader` is deprecated as of Python 3.7. Code should be ported to `ResourceReader` / `importlib.resources` if possible.

7.4.6 `importlib.metadata` Compatibility

`OxidizedFinder` implements `find_distributions()` and therefore provides the required hook for `importlib.metadata` to resolve `Distribution` instances. However, the returned objects do not implement the full `Distribution` interface.

Here are the known differences between `OxidizedDistribution` and `importlib.metadata.Distribution` instances:

- `OxidizedDistribution` is not an instance of `importlib.metadata.Distribution`.
- `locate_file()` is not defined.
- `@staticmethod at()` is not defined.
- `@property files` raises `NotImplementedError`.

There are additional _ prefixed attributes of `importlib.metadata.Distribution` that are not implemented. But we do not consider these part of the public API and don't feel they are worth calling out.

In addition, `OxidizedFinder.find_distributions()` ignores the `path` attribute of the passed `Context` instance. Only the `name` attribute is consulted. If `name` is `None`, all packages with registered distribution files will be returned. Otherwise the returned `list` contains at most 1 `PyOxidizerDistribution` corresponding to the requested package `name`.

7.4.7 `pkgutil` Compatibility

The pkgutil package in Python's standard library reacts to special functionality on `MetaPathFinder` instances.

`pkgutil.iter_modules()` attempts to use an `iter_modules()` method to obtain results.

`OxidizedFinder` implements `iter_modules(prefix="")` and `pkgutil.iter_modules()` should work. However, there are some differences in behavior:

- `iter_modules()` is defined to be a generator but `OxidizedFinder.iter_modules()` returns a `list`. `list` is iterable and this difference should hopefully be a harmless implementation detail.
- Support for the `path` argument to `pkgutil.iter_modules()` requires that `OxidizedFinder`'s *path_hook* is installed in `sys.path_hooks`. For automatic registration of this at interpreter initialization time, both `OxidizedFinder` and the stdlib filesystem-based importer have to be enabled.

7.5 `oxidized_importer` Python Resource Types

The `oxidized_importer` module defines Python types beyond `OxidizedFinder`. This page documents those types and their APIs.

Important: All types are backed by Rust structs and all properties return copies of the data. This means that if you mutate a Python variable that was obtained from an instance's property, that mutation won't be reflected in the backing Rust struct.

7.5.1 `OxidizedResource`

The `OxidizedResource` Python type represents a *resource* that is indexed by a `OxidizedFinder` instance.

Each instance represents a named entity with associated metadata and data. e.g. an instance can represent a Python module with associated source and bytecode.

New instances can be constructed via `OxidizedResource()`. This will return an instance whose `name = ""` and all properties will be `None` or `false`.

Properties

The following properties/attributes exist on `OxidizedResource` instances:

`is_module` A `bool` indicating if this resource is a Python module. Python modules are backed by source or bytecode.

`is_builtin_extension_module` A `bool` indicating if this resource is a Python extension module built-in to the Python interpreter.

`is_frozen_module` A `bool` indicating if this resource is a Python module whose bytecode is frozen into the Python interpreter.

`is_extension_module` A `bool` indicating if this resource is a Python extension module.

`is_shared_library` A `bool` indicating if this resource is a shared library.

`name` The `str` name of the resource.

`is_package` A `bool` indicating if this resource is a Python package.

`is_namespace_package` A `bool` indicating if this resource is a Python namespace package.

`in_memory_source` `bytes` or `None` holding Python module source code that should be imported from memory.

`in_memory_bytecode` `bytes` or `None` holding Python module bytecode that should be imported from memory.

This is raw Python bytecode, as produced from the `marshal` module. `.pyc` files have a header before this data that will need to be stripped should you want to move data from a `.pyc` file into this field.

`in_memory_bytecode_opt1` `bytes` or `None` holding Python module bytecode at optimization level 1 that should be imported from memory.

This is raw Python bytecode, as produced from the `marshal` module. `.pyc` files have a header before this data that will need to be stripped should you want to move data from a `.pyc` file into this field.

`in_memory_bytecode_opt2` `bytes` or `None` holding Python module bytecode at optimization level 2 that should be imported from memory.

This is raw Python bytecode, as produced from the `marshal` module. `.pyc` files have a header before this data that will need to be stripped should you want to move data from a `.pyc` file into this field.

in_memory_extension_module_shared_library `bytes` or `None` holding native machine code defining a Python extension module shared library that should be imported from memory.

in_memory_package_resources `dict[str, bytes]` or `None` holding resource files to make available to the `importlib.resources` APIs via in-memory data access. The `name` of this object will be a Python package name. Keys in this dict are virtual filenames under that package. Values are raw file data.

in_memory_distribution_resources `dict[str, bytes]` or `None` holding resource files to make available to the `importlib.metadata` API via in-memory data access. The `name` of this object will be a Python package name. Keys in this dict are virtual filenames. Values are raw file data.

in_memory_shared_library `bytes` or `None` holding a shared library that should be imported from memory.

shared_library_dependency_names `list[str]` or `None` holding the names of shared libraries that this resource depends on. If this resource defines a loadable shared library, this list can be used to express what other shared libraries it depends on.

relative_path_module_source `pathlib.Path` or `None` holding the relative path to Python module source that should be imported from the filesystem.

relative_path_module_bytecode `pathlib.Path` or `None` holding the relative path to Python module bytecode that should be imported from the filesystem.

relative_path_module_bytecode_opt1 `pathlib.Path` or `None` holding the relative path to Python module bytecode at optimization level 1 that should be imported from the filesystem.

relative_path_module_bytecode_opt1 `pathlib.Path` or `None` holding the relative path to Python module bytecode at optimization level 2 that should be imported from the filesystem.

relative_path_extension_module_shared_library `pathlib.Path` or `None` holding the relative path to a Python extension module that should be imported from the filesystem.

relative_path_package_resources `dict[str, pathlib.Path]` or `None` holding resource files to make available to the `importlib.resources` APIs via filesystem access. The `name` of this object will be a Python package name. Keys in this dict are filenames under that package. Values are relative paths to files from which to read data.

relative_path_distribution_resources `dict[str, pathlib.Path]` or `None` holding resource files to make available to the `importlib.metadata` APIs via filesystem access. The `name` of this object will be a Python package name. Keys in this dict are filenames under that package. Values are relative paths to files from which to read data.

`OxidizedResource` Resource Types

Each `OxidizedResource` instance describes a particular type of resource. If a resource identifies as a type, it sets one of the following `is_*` attributes to `True`:

is_module A Python module. These typically have source or bytecode attached.

Modules can also be packages. In this case, they can hold additional data, such as a mapping of resource files.

is_builtin_extension_module A built-in extension module. These represent Python extension modules that are compiled into the application and don't exist as separate shared libraries.

is_frozen_module A frozen Python module. These are Python modules whose bytecode is compiled into the application.

is_extension_module A Python extension module. These are shared libraries that can be loaded to provide additional modules to Python.

is_shared_library A shared library. e.g. a `.so` or `.dll`.

7.5.2 PythonModuleSource

The `oxidized_importer.PythonModuleSource` type represents Python module source code. e.g. a `.py` file.

Instances have the following properties:

module (str) The fully qualified Python module name. e.g. `my_package.foo`.

source (bytes) The source code of the Python module.

> Note that source code is stored as `bytes`, not `str`. Most Python source is stored as `utf-8`, so you can `.encode("utf-8")` or `.decode("utf-8")` to convert between `bytes` and `str`.

is_package (bool) This this module is a Python package.

7.5.3 PythonModuleBytecode

The `oxidized_importer.PythonModuleBytecode` type represents Python module bytecode. e.g. what a `.pyc` file holds (but without the header that a `.pyc` file has).

Instances have the following properties:

module (str) The fully qualified Python module name.

bytecode (bytes) The bytecode of the Python module.

> This is what you would get by compiling Python source code via something like `marshal.dumps(compile(source, "exe"))`. The bytecode does **not** contain a header, like what would be found in a `.pyc` file.

optimize_level (int) The bytecode optimization level. Either `0`, `1`, or `2`.

is_package (bool) Whether this module is a Python package.

7.5.4 PythonExtensionModule

The `oxidized_importer.PythonExtensionModule` type represents a Python extension module. This is a shared library defining a Python extension implemented in native machine code that can be loaded into a process and defines a Python module. Extension modules are typically defined by `.so`, `.dylib`, or `.pyd` files.

Instances have the following properties:

name (str) The name of the extension module.

Note: Properties of this type are read-only.

7.5.5 PythonPackageResource

The `oxidized_importer.PythonPackageResource` type represents a non-module *resource* file. These are files that live next to Python modules that are typically accessed via the APIs in `importlib.resources`.

Instances have the following properties:

package (str) The name of the leaf-most Python package this resource is associated with.

With `OxidizedFinder`, an `importlib.abc.ResourceReader` associated with this package will be used to load the resource.

name (str) The name of the resource within its `package`. This is typically the filename of the resource. e.g. `resource.txt` or `child/foo.png`.

data (bytes) The raw binary content of the resource.

7.5.6 PythonPackageDistributionResource

The `oxidized_importer.PythonPackageDistributionResource` type represents a non-module *resource* file living in a package distribution directory (e.g. `<package>-<version>.dist-info` or `<package>-<version>.egg-info`). These resources are typically accessed via the APIs in `importlib.metadata`.

Instances have the following properties:

package (str) The name of the Python package this resource is associated with.

version (str) Version string of Python package this resource is associated with.

name (str) The name of the resource within the metadata distribution. This is typically the filename of the resource. e.g. `METADATA`.

data (bytes) The raw binary content of the resource.

7.6 Resource Scanning APIs

The `oxidized_importer` module exposes functions and Python types to facilitate scanning for and collecting Python resources.

7.6.1 find_resources_in_path(path)

The `oxidized_importer.find_resources_in_path()` function will scan the specified filesystem path and return an iterable of objects representing found resources. Those objects will be 1 of the types documented in *oxidized_importer Python Resource Types*.

Only directories can be scanned.

To discover all filesystem based resources that Python's `PathFinder` *meta path finder* would (with the exception of `.zip` files), try the following:

```
import os
import oxidized_importer
import sys

resources = []
for path in sys.path:
    if os.path.isdir(path):
        resources.extend(oxidized_importer.find_resources_in_path(path))
```

7.6.2 `OxidizedResourceCollector` Python Type

The `oxidized_importer.OxidizedResourceCollector` type provides functionality for turning instances of Python resource types into a collection of `OxidizedResource` for loading into an `OxidizedFinder` instance. It exists as a convenience, as working with individual `OxidizedResource` instances can be rather cumbersome.

Instances can be constructed by passing an `allowed_locations=<list[str]>` argument defining locations that resources can be loaded from. The accepted string values are `in-memory` and `filesystem-relative`.

e.g. to create a collector that only marks resources for in-memory loading:

```
import oxidized_importer

collector = oxidized_importer.OxidizedResourceCollector(
    allowed_locations=["in-memory"]
)
```

Instances of `OxidizedResourceCollector` have the following properties:

`allowed_locations(list[str])` Exposes allowed locations where resources can be loaded from.

Methods are documented in the following sections.

`add_in_memory(resource)`

`OxidizedResourceCollector.add_in_memory(resource)` adds a Python resource type (`PythonModuleSource`, `PythonModuleBytecode`, etc) to the collector and marks it for loading via in-memory mechanisms.

`add_filesystem_relative(prefix, resource)`

`OxidizedResourceCollector.add_filesystem_relative(prefix, resource)` adds a Python resource type (`PythonModuleSource`, `PythonModuleBytecode`, etc) to the collector and marks it for loading via a relative path next to some *origin* path (as specified to the `OxidizedFinder`). That relative path can have a `prefix` value prepended to it. If no prefix is desired and you want the resource placed next to the *origin*, use an empty `str` for `prefix`.

`oxidize()`

`OxidizedResourceCollector.oxidize()` takes all the resources collected so far and turns them into data structures to facilitate later use.

The return value is a tuple of `(List[OxidizedResource], List[Tuple[pathlib.Path, bytes, bool]])`.

The first element in the tuple is a list of `OxidizedResource` instances.

The second is a list of 3-tuples containing the relative filesystem path for a file, the content to write to that path, and whether the file should be marked as executable.

7.7 Loading Resource Files

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*.

`oxidized_importer` has support for loading resource files. But compatibility with Python's expected behavior may vary.

7.7.1 Python Resource Loading Mechanisms

Before we talk about `oxidized_importer`'s support for resource loading, it is important to understand how Python code in the wild can load resources.

We'll overview them in the chronological order they were introduced into the Python ecosystem.

The most basic and oldest mechanism to load resources is to perform raw filesystem I/O. Typically, Python code looks at `__file__` to get the filename of the current module. Then, it calculates the directory name and derives paths to resource files using e.g. `os.path.join()`. It will usually then `open()` these paths directly.

Python packaging evolved over time. Packaging tools could express various metadata at build time, such as supplementary *resource* files. This metadata would be installed next to a package and APIs could be used to access it. One such API was pkg_resources. Using e.g. `pkg_resources.resource_string("foo", "bar.txt")`, you could obtain the content of the resource `bar.txt` in the `foo` package.

`pkg_resources` had useful functionality. And it was the recommended mechanism for loading resource files for several years. But it wasn't part of the Python standard library and needed to be explicitly installed. So not everyone used it.

Python 3.1 added the `importlib` package, which is the primary home for all core functionality related to `import`. Python importers were now defined via interfaces. One of those interfaces is `ResourceLoader`. It has a single method `get_data(path)`. Given a Python module's `loader` (e.g. via the `__loader__` attribute on the module), you could call `get_data(path)` and load a resource. e.g. `import foo; foo.__loader__.get_data("bar.txt")`.

The standard library only had `ResourceLoader` for several years. And `ResourceLoader` wasn't exactly a convenient API to use because it was so low-level. Many Python applications continued to use `pkg_resources` or direct file-based I/O.

Python 3.7 introduced significant improvements to resource loading in the standard library.

At a low level, module loaders could now implement a `get_resource_reader(name)` method, which would return an object implementing the ResourceReader interface. This interface defined methods like `open_resource(name)` and `contents()` to open a file-like handle on a named resource and obtain a list of all available resources.

At a high level, the importlib.resources package provided a user-friendly API for interacting with `ResourceReader` instances. You could call e.g. `importlib.resources.open_binary(package, name)` to obtain a file-like handle on a specific resource within a package.

Python 3.7's new resource APIs finally gave the Python standard library access to powerful APIs for loading resources without using a 3rd party package (like `pkg_resources`).

At the time of writing this in April 2020, it looks like Python 3.9 will invent yet another low-level resource loading API.

Because Python hasn't had a robust resource loading API in the standard library for much of its history, lots of Python code in the wild does not make use of the APIs in the standard library. It is not uncommon to see code in 2020 that still uses `__file__` to load resources. Furthermore, because Python 3.7 is still relatively young and code may wish to maintain compatibility with older Python versions, the newer APIs may be actively avoided.

Important: As of Python 3.8, `ResourceReader` and `importlib.resources` are the most robust mechanisms for loading resources and we recommend adopting these APIs if possible.

7.7.2 Support for `ResourceReader`

`oxidized_importer` implements the `ResourceReader` interface for loading resource files.

However, compatibility with Python's default filesystem-based implementation can vary. Unfortunately, various behavior with `ResourceReader` is undefined, so it isn't clear if CPython or `oxidized_importer` is buggy here.

`oxidized_importer` maintains an index of known resource files. This index is logically a `dict` of `dict``s, where the outer key is the Python package name and the inner key is the resource name. Package names are fully qualified. e.g. ``foo` or `foo.bar`. Resource names are effectively relative filesystem paths. e.g. `resource.txt` or `subdir/resource.txt`. The relative paths always use `/` as the directory separator, even on Windows.

`OxidizedFinder.get_resource_reader()` returns instances of `OxidizedResourceReader`. Each instance is bound to a specific Python package: that's how they are defined. When an `OxidizedResourceReader` receives the name of a resource, it performs a simple lookup in the global resources index. If the string key is found, it is used. Otherwise, it is assumed the resource doesn't exist.

The `OxidizedResourceReader.contents()` method will return a list of all keys in the internal resources index.

`OxidizedResourceReader` works the same way for in-memory and filesystem-relative resource locations because internally both use the same index of resources to drive execution: only the location of the resource content varies.

`OxidizedResourceReader`'s implementation varies from the standard library filesystem-based implementation in the following ways:

- `OxidizedResourceReader.contents()` will return keys from the package's resources dictionary, not all the files in the same directory as the underlying Python package (the standard library uses `os.listdir()`). `OxidizedResourceReader` will therefore return resource names in sub-directories as long as those sub-directories aren't themselves Python packages.
- Resources must be explicitly registered with `OxidizedFinder` as such in order to be exposed via the resources API. By contrast, the filesystem-based importer - relying on `os.listdir()` - will expose all files in a directory as a resource. This includes `.py` files.
- `OxidizedResourceReader.is_resource()` will return `True` for resource names containing a slash. Contrast with Python's, which returns `False` (even though you can open a resource with `ResourceReader.open_resource()` for the same path). `OxidizedResourceReader`'s behavior is more consistent.

7.7.3 Support for `ResourceLoader`

`OxidizedFinder` implements the deprecated `ResourceLoader` interface and `get_data(path)` will return `bytes` instances for registered resources or raise `OSError` on request of an unregistered resource.

The path passed to `get_data(path)` MUST be an absolute path that has the prefix of either the currently running executable file or the directory containing it.

If the resource path is prefixed with the current executable's path, the path components after the current executable path are interpreted as the path to a resource registered for in-memory loading.

If the resource path is prefixed with the current executable's directory, the path components after this directory are interpreted as the path to a resource registered for application-relative loading.

All other resource paths aren't recognized and an `OSError` will be raised. There is no fallback to loading from the filesystem, even if a valid filesystem path pointing to an existing file is passed in.

Note: The behavior of not servicing paths that actually exist but aren't registered with `OxidizedFinder` as resources may be overly opinionated and undesirable for some applications.

If this is a legitimate use case for your application, please create a GitHub issue to request this feature.

Once a path is recognized as having the prefix of the current executable or its directory, the remaining path components will be interpreted as the resource path. This resource path logically contains a package name component and a resource name component. `OxidizedFinder` will traverse all potential package names starting from the longest/deepest up until the top-level package looking for a known Python package. Once a known package name is encountered, its resources will be consulted. At most 1 package will be consulted for resources.

Here is a concrete example.

If the `path` is `/usr/bin/myapp/foo/bar/resource.txt` and the current executable is `/usr/bin/myapp`, the requested resource will be `foo/bar/resource.txt`. Since the path was prefixed with the executable path, only resources registered for in-memory loading will be consulted.

Our candidate package names are `foo.bar` and `foo`, in that order.

If `foo.bar` is a known package and `resource.txt` is registered for in-memory loading, that resource's contents will be returned.

If `foo.bar` is a known package and `resource.txt` is not registered in that package, `OSError` is raised.

If `foo.bar` is not a known package, we proceed to check for package `foo`.

If `foo` is a known package and `bar/resource.txt` is registered for in-memory loading, its contents will be returned.

Otherwise, we're out of possible packages, so `OSError` is raised.

Similar logic holds for resources registered for filesystem-relative loading. The difference here is the stripped path prefix and we are only looking for resources registered for filesystem-relative loading. Otherwise, the traversal logic is exactly the same.

If `OSError` is raised due to a missing resource, its `errno` is `ENOENT` and its `filename` is the passed in `path`. Python should automatically translate this to a `FileNotFoundError` exception. But callers should catch `OSError`, as other `OSError` variants can be raised (e.g. for file permission errors).

7.7.4 Support for `__file__`

`OxidizedFinder` may or may not set the `__file__` attribute on loaded modules. See *__file__ and __cached__ Module Attributes* for details.

Therefore, Python code relying on the presence of `__file__` to derive paths to resource files may or may not work with `oxidized_importer`.

Code utilizing `__file__` for resource loading is highly encouraged to switch to the `importlib.resources` API. If this is not possible, you can change packaging settings to move the resource locations from in-memory to filesystem-relative, as `__file__` is set when loading modules from the filesystem.

7.7.5 Support for `pkg_resources`

`pkg_resources`'s APIs for loading resources likely do not work with `oxidized_importer`.

7.7.6 Porting Code to Modern Resources APIs

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

```
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:

```
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)
```

The `ResourceReader` interface is quite low-level. If you want something higher level or want to access resources outside the current module, it is recommended to use the importlib.resources APIs. e.g.:

```
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.

7.7.7 Maintaining Compatibility With Python <3.7

If you want to maintain compatibility with Python <3.7, you can't use `ResourceReader` or `importlib.resources`, as they are not available. The recommended solution here is to use a shim.

The best shim to use is importlib_resources. This is a standalone Python package that is a backport of `importlib.resources` to older Python versions. Essentially, you can always get the APIs from the latest Python version. This shim knows about the various APIs available on `Loader` instances and chooses the best available one. It should *just work* with `oxidized_importer`.

If you want to implement your own shim without introducing a dependency on `importlib_resources`, the following code can be used as a starting implementation:

```
import importlib

try:
    import importlib.resources
```

(continues on next page)

(continued from previous page)

```
    # 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)

        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.)

This code is provided for example purposes only. It may or may not be sufficient for your needs.

7.8 *Freezing* Applications with `oxidized_importer`

`oxidized_importer` can be used to create and run *frozen* Python applications, where Python resources data (module source and bytecode, etc) is *frozen*/packaged and distributed next to your *application*.

This is conceptually similar to what PyOxidizer does. The major difference is that PyOxidizer will package and distribute a Python distribution with your application: when only `oxidized_importer` is being used, the Python distribution is provided by some other means (it is typically already installed on the system). This makes `oxidized_importer` a light-weight alternative to PyOxidizer for scenarios where PyOxidizer isn't suitable or viable.

7.8.1 High-Level Freezing Workflow

The steps for *freezing* an application all look the same:

1. Load `OxidizedResource` instances into an `OxidizedFinder` instance so they are indexed.
2. Serialize indexed resources.
3. Write the serialized resources blob somewhere along with any files (if using filesystem-based loading).
4. Somehow make that resources blob available to others (you could add it as a *resource* file in your Python package for example).
5. From your application, construct an `OxidizedFinder` instance and load the resources blob you generated.
6. Register the `OxidizedFinder` instance as the first element on `sys.meta_path`.

The next sections show what this may look like.

7.8.2 Indexing and Serializing Resources

In your *build* process, you'll need to index resources and serialize them. You can construct `OxidizedResource` instances directly and hand them off to an `OxidizedFinder` instance. But you'll probably want to use `OxidizedResourceCollector` to make this simpler.

Try something like the following:

```
import os
import stat
import sys

import oxidized_importer

# Create a collector to help with managing resources.
collector = oxidized_importer.OxidizedResourceCollector(
    allowed_locations=["in-memory"]
)

# Add all known Python resources by scanning sys.path.
# Note: this will pull in the Python standard library and
# any other installed packages, which may not be desirable!
for path in sys.path:
    # Only directories can be scanned by oxidized_importer.
    if os.path.isdir(path):
        for resource in oxidized_importer.find_resources_in_path(path):
            collector.add_in_memory(resource)
```

(continues on next page)

(continued from previous page)

```
# Turn the collected resources into ``OxidizedResource`` and file
# install rules.
resources, file_installs = collector.oxidize()

# Now index the resources so we can serialize them.
finder = oxidized_importer.OxidizedFinder()
finder.add_resources(resources)

# Turn the indexed resources into an opaque blob.
packed_data = finder.serialize_indexed_resources()

# Write out that data somewhere.
with open("oxidized_resources", "wb") as fh:
    fh.write(packed_data)

# Then for all the file installs, materialize those files.
for (path, data, executable) in file_installs:
    path.parent.mkdir(parents=True, exist_ok=True)

    with path.open("wb") as fh:
        fh.write(data)

    if executable:
        path.chmod(path.stat().st_mode | stat.S_IEXEC)
```

At this point, you've collected all known Python resources and written out a data structure describing them all. For resources targeting in-memory loading, the content of those resources is embedded in the data structure. For resources targeting filesystem-relative loading, the data structure contains the relative path to those resources. And you've written out the files in the locations where those relative paths point to.

7.8.3 Loading Serialized Resources in Your Application

Now, from our *application* code, we need to load the resources and register the custom importer with Python:

```
import os
import sys

import oxidized_importer

# Load those resources into an instance of our custom importer. This
# will read the index in the passed data structure and make all
# resources immediately available for importing.
finder = oxidized_importer.OxidizedFinder()
finder.index_file_memory_mapped("oxidized_resources")

# If the relative path of filesystem-based resources is not relative
# to the current executable (which is likely the ``python3`` executable),
# you'll need to set ``origin`` to the directory the resources are
# relative to.
finder = oxidized_importer.OxidizedFinder(
    relative_path_origin=os.path.dirname(os.path.abspath(__file__)),
)
finder.index_bytes(packed_data)
```

(continues on next page)

(continued from previous page)

```
# Register the meta path finder as the first item, making it the
# first finder that is consulted.
sys.meta_path.insert(0, finder)

# At this point, you should be able to ``import`` modules defined
# in the resources data!
```

7.9 Common Issues

7.9.1 Extension Modules Support

Unlike PyOxidizer, `OxidizedResourceCollector` isn't (yet) as intelligent about how to handle extension modules (standalone machine native shared libraries). And even PyOxidizer's support for extension modules can be brittle.

One notable difference between PyOxidizer and `OxidizedResourceCollector` is PyOxidizer is able to determine whether importing extension modules from memory is supported and is able to automatically redirect an extension module to filesystem-based loading if not supported. `OxidizedResourceCollector` is *dumb* and adds resources where you tell it to.

`OxidizedFinder` supports loading extension modules from memory on Windows. But everywhere else, this isn't supported and will result in an `ImportError` if you index an extension module for in-memory loading.

To work around this deficiency, you'll want to mark extension modules as loaded from the filesystem unless you are on Windows. Try something like this:

```
import oxidized_importer

collector = oxidized_importer.OxidizedResourceCollector(
    allowed_locations=["in-memory", "filesystem-relative"],
)

# Redirect extension modules to the filesystem and everything else to
# memory.
for resource in oxidized_importer(find_resources_in_path("/path/to/resources")):
    if isinstance(resource, oxidized_importer.PythonExtensionModule):
        collector.add_filesystem_relative("lib", resource)
    else:
        collector.add_in_memory(resource)
```

7.9.2 Resource Scanning Descends Into `site-packages`

`oxidized_importer.find_resources_in_path()` descends into `site-packages` directories. This is arguably not the desired behavior, especially when in the context of virtualenvs, which may want to not inherit the resources in the `site-packages` of the *outer* Python installation. This will likely be fixed in a future release.

7.10 Security Implications of Loading Resources

`OxidizedFinder` allows Python code to define its own `OxidizedResource` instances to be made available for loading. This means Python code can define its own Python module source or bytecode that could later be executed. It also allows registration of extension modules and shared libraries, which give a vector for allowing execution of native machine code.

This feature has security implications, as it provides a vector for arbitrary code execution.

While it might be possible to restrict this feature to provide stronger security protections, we have not done so yet. Our thinking here is that it is extremely difficult to sandbox Python code. Security sandboxing at the Python layer is effectively impossible: the only effective mechanism to sandbox Python is to add protections at the process level. e.g. by restricting what system calls can be performed. We feel that the capability to inject new Python modules and even shared libraries via `OxidizedFinder` doesn't provide any new or novel vector that doesn't already exist in Python's standard library and can't already be exploited by well-crafted Python code. Therefore, this feature isn't a net regression in security protection.

If you have a use case that requires limiting the features of `OxidizedFinder` so security isn't sacrificed, please *file an issue <https://github.com/indygreg/PyOxidizer/issues>*.

CHAPTER 8

Python Packed Resources

PyOxidizer has defined a custom data format for storing resources useful to the execution of a Python interpreter. We call this data format *Python packed resources*.

The way it works is that some producer collects resources required by a Python interpreter. These resources include Python module source and bytecode, non-module resource/data files, extension modules, and shared libraries. Metadata about these resources and sometimes the raw resource data itself is serialized to a binary data structure.

At Python interpreter run time, this data structure is loaded (it can be embedded in a binary or exist as a standalone file) and parsed. A custom *Python Meta Path Finders* (*OxidizedFinder* from *oxidized_importer Python Extension*) then uses the parsed data structure to power Python module importing.

This functionality is similar to using a `.zip` file for holding Python modules. However, the *Python packed resources* data structure is far more advanced.

8.1 Implementation

The canonical implementation of the writer and parser of this data structure lives in the `python-packed-resources` Rust crate. The canonical home of this crate is https://github.com/indygreg/PyOxidizer/tree/main/python-packed-resources.

This crate is published to crates.io at https://crates.io/crates/python-packed-resources.

8.2 Specification

From a high level, the data structure defines an iterable of *resources*. A *resource* is an entity with a name, metadata, and blob fields. Typically the most common *resource* is a Python module/package. But other resource types (such as shared libraries) are defined.

The first 8 bytes of the data structure are a magic header identifying the content as our data structure and the version of it. The first 7 bytes are `pyembed` and the following 1 byte denotes a version. Semantics of each version are denoted in sections below.

8.2.1 High-Level Layout

From a high-level, the serialized format consists of:

- A *global header* describing the overall payload.
- An index describing the blob sections present in the payload.
- An index describing each resource and its content.
- A series of blob sections holding the data referenced by the resources index.

A resource is composed of various *fields* that describe it. Examples of fields include the resource name, source code, and bytecode. The resources index describes which fields are present and where to find them in the payload.

The actual content of fields (e.g. the raw bytes containing source code) is stored in field-specific sections after the index. Each field has its own section and data for all resources is stored next to each other. e.g. you will have all the data for resource names followed by all data for module sourcecode.

The low-level data format is described below. All integers are little-endian.

The first 13 bytes after the magic header denote a global header. The global header consists of:

- A `u8` denoting the number of blob sections, `blob_sections_count`.
- A `u32` denoting the length of the blob index, `blob_index_length`.
- A `u32` denoting the total number of resources in this data, `resources_count`.
- A `u32` denoting the length of the resources index, `resources_index_length`.

Following the *global header* is the *blob index*. The blob index describes the various blob sections present in the payload following the *resources index*.

Each entry in the *blob index* logically consists of a set of fields defining metadata about each *blob section*. This is encoded by a *start of entry* `u8` marker followed by N `u8` field type values and their corresponding metadata, followed by an *end of entry* `u8` marker. The *blob index* is terminated by an *end of index* `u8` marker. The total number of bytes in the *blob index* including the *end of index* marker should be `blob_index_length`.

Following the *blob index* is the *resources index*. Each entry in this index defines a sparse set of metadata describing a single resource. Entries are composed of a series of `u8` identifying pieces of metadata, followed by field-specific supplementary descriptions. For example, a value of `0x02` denotes the length of the resources's name and is immediately followed by a `u16` holding said length. See the section below for each field tracked by this index.

Following the *resources index* is blob data. Blob data is logically consisted of different sections holding data for different fields for different resources. But there is no internal structure or separators: all the individual blobs are just laid out next to each other.

8.2.2 Blob Field Types

The Blob Index allows attributing a sparse set of metadata with every blob section entry. The type of metadata being conveyed is defined by a `u8`. Some field types have additional metadata following that field.

The various field types and their semantics follow.

`0x00` End of index. This field indicates that there are no more blob index entries and we've reached the end of the *blob index*.

`0x01` Start of blob section entry. Encountering this value signals the beginning of a new blob section. From a specification standpoint, this isn't strictly required. But it helps ensure parser state.

`0xff` End of blob section entry. Encountering this value signals the end of the current blob section definition. The next encountered `u8` in the index should be `0x01` to denote a new entry or `0x00` to denote end of index.

0x02 Resource field type. This field defines which resource field this blob section is holding data for. A `u8` following this one will contain the resource field type value (see section below).

0x03 Raw payload length. This field defines the raw length in bytes of the blob section in the payload. The `u64` containing that length will immediately follow this `u8`.

0x04 Interior padding mechanism. This field defines interior padding between elements in the blob section. Following this `u8` is another `u8` denoting the padding mechanism.

`0x01` indicates no padding. `0x02` indicates NULL padding (a `0x00` between elements).

If not present, *no padding* is assumed. If the payload data logically consists of discrete resources (e.g. Python package resource files), then padding applies to these sub-elements as well.

8.2.3 Resource Field Types

The Resources Index allows attributing a sparse set of metadata with every resource. A `u8` indicates what metadata is being conveyed. Some field types have additional metadata following this `[u8]` further defining the field. The values of each defined metadata type follow.

0x00 End of index. Special type to denote the end of an index.

0x01 Start of resource entry. Signals the beginning of a new resource. From a specification standpoint this isn't strictly required. But it helps ensure parser state.

0x02 Resource flavor. Declares the type of resource this entry represents. A `u8` defining the resource flavor immediately follows this byte. See the section below for valid resource flavors.

This field is deprecated in version 2 in favor of the individual fields expressing presence of a resource type. (See fields starting at `0x16`.)

0xff End of resource entry. The next encountered `u8` in the index should be an *end of index* or *start of resource* marker.

0x03 Resource name. A `u16` denoting the length in bytes of the resource name immediately follows this byte. The resource name *must* be valid UTF-8.

0x04 Package flag. If encountered, the resource is identified as a Python package.

0x05 Namespace package flag. If encountered, the resource is identified as a Python *namespace package*.

0x06 In-memory Python module source code. A `u32` denoting the length in bytes of the module's source code immediately follows this byte.

0x07 In-memory Python module bytecode. A `u32` denoting the length in bytes of the module's bytecode immediately follows this byte.

0x08 In-memory Python module optimized level 1 bytecode. A `u32` denoting the length in bytes of the module's optimization level 1 bytecode immediately follows this byte.

0x09 In-memory Python module optimized level 2 bytecode. Same as previous, except for bytecode optimization level 2.

0x0a In-memory Python extension module shared library. A `u32` denoting the length in bytes of the extension module's machine code immediately follows this byte.

0x0b In-memory Python resources data. If encountered, the module/package contains non-module resources files and the number of resources is contained in a `u32` that immediately follows. Following this `u32` is an array of `(u16, u64)` denoting the resource name and payload size for each resource in this package.

0x0c In-memory Python distribution resource. Defines resources accessed from `importlib.metadata` APIs. If encountered, the module/package contains distribution metadata describing the package. The number of files

being described is contained in a `u32` that immediately follows this byte. Following this `u32` is an array of `(u16, u64)` denoting the distribution file name and payload size for each virtual file in this distribution.

0x0d In-memory shared library. If set, this resource is a shared library and not a Python module. The resource name field is the name of this shared library, with file extension (as it would appear in a dynamic binary's loader metadata to indicate a library dependency). A `u64` denoting the length in bytes of the shared library data follows. This shared library should be loaded from memory.

0x0e Shared library dependency names. This field indicates the names of shared libraries that this entity depends on. The number of library names is contained in a `u16` that immediately follows this byte. Following this `u16` is an array of `u16` denoting the length of the library name for each shared library dependency. Each described shared library dependency may or may not be described by other entries in this data structure.

0x0f Relative filesystem path to Python module source code. A `u32` holding the length in bytes of a filesystem path encoded in the platform-native file path encoding follows. The source code for a Python module will be read from a file at this path.

0x10 Relative filesystem path to Python module bytecode. Similar to the previous except the filesystem path holds Python module bytecode.

0x11 Relative filesystem path to Python module bytecode at optimization level 1. Similar to the previous except for what is being pointed to.

0x12 Relative filesystem path to Python module bytecode at optimization level 2. Similar to the previous except for what is being pointed to.

0x13 Relative filesystem path to Python extension module shared library. Similar to the previous except the file holds a Python extension module loadable as a shared library.

0x14 Relative filesystem path to Python package resources. The number of resources is contained in a `u32` that immediately follows. Following this `u32` is an array of `(u16, u32)` denoting the resource name and filesystem path to each resource in this package.

0x15 Relative filesystem path to Python distribution resources.

Defines resources accessed from `importlib.metadata` APIs. If encountered, the module/package contains distribution metadata describing the package. The number of files being described is contained in a `u32` that immediately follows this byte. Following this `u32` is an array of `(u16, u32)` denoting the distribution file name and filesystem path to that distribution file.

0x16 Is Python module flag. If set, this resource contains data for an importable Python module or package. Resource data is associated with Python packages and is covered by this type.

0x17 Is builtin extension module flag. This type represents a Python extension module that is built in (compiled into) the interpreter itself or is otherwise made available to the interpreter via `PyImport_Inittab` such that it should be imported with the *builtin* importer.

0x18 Is frozen Python module flag. This type represents a Python module whose bytecode is *frozen* and made available to the Python interpreter via the `PyImport_FrozenModules` array and should be imported with the *frozen* importer.

0x19 Is Python extension flag. This type represents a compiled Python extension. Extensions have specific requirements around how they are to be loaded and are differentiated from regular Python modules.

0x1a Is shared library flag. This type represents a shared library that can be loaded into a process.

0x1b Is utf-8 filename data flag. This type represents an arbitrary filename. The resource name is a UTF-8 encoded filename of the file this resource represents. The file's data is either embedded in memory or referred to via a relative path reference.

0x1c File data is executable flag.

If set, the arbitrary file this resource tracks should be marked as executable.

0x1d Embedded file data.

If present, the resource should be a file resource and this field holds its raw file data in memory.

A `u64` containing the length of the embedded data follows this field.

0x1e UTF-8 relative path file data.

If present, the resource should be a file resource and this field defines the relative path containing that file's data. The relative path filename is UTF-8 encoded.

A `u32` denoting the length of the UTF-8 relative path (in bytes) follows.

8.2.4 Resource Flavors

Important: Enumerated resource flavors are deprecated after version 1. You should use individual fields to express resource identity instead.

The data format allows defining different types/flavors of resources. This flavor of a resource is identified by a `u8`. The declared flavors are:

0x00 No flavor. Should not be encountered.

0x01 Python module/package. This is equivalent to resource field `0x16` being set.

0x02 Builtin Python extension module. This is equivalent to resource field `0x17` being set.

0x03 Frozen Python module. This is equivalent to resource field `0x18` being set.

0x04 Python extension. This is equivalent to resource field `0x19` being set.

0x05 Shared library. This is equivalent to resource field `0x1a` being set.

8.2.5 pyembed\x01 Format

The initially released/formalized packed resources data format.

Supports resource field types up to and including `0x15`.

8.2.6 pyembed\x02 Format

Version 2 of the packed resources data format.

This version introduces field type values `0x16` to `0x1a`. The resource flavor field type (`0x02`) is deprecated and the individual field types denoting resource types should be used instead.

(PyOxidizer removed run-time code looking at field type `0x02` when this format was introduced.)

8.2.7 pyembed\x03 Format

Version 3 of the packed resources data format.

This version introduces field type values `0x1b` to `0x1e`.

These fields provide the ability for a resource to identify itself as an arbitrary filename and for the arbitrary file data to be embedded within the data structure or referenced via a relative path.

Unlike previous fields that use OS-native encoding of filesystem paths (`[u8]` on POSIX and `[u16]` on Windows), the paths for these new fields use UTF-8. This can't represent all valid paths on all platforms. But it is portable and works for most paths encountered in the wild.

8.3 Design Considerations

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

Performance is a significant consideration. We want everything to be as fast as possible. Possible dimensions influencing performance include parse time, payload size, and I/O access patterns.

The payload is designed such that the *index* data is at the beginning so a reader only has to read a contiguous slice of data to fully understand the data within. This is in opposition to jumping around the entire data structure to extract metadata of the data within. This means that we only need to page in a fraction of the total backing data structure in order to initialize our custom importer. In addition, the index data is read sequentially. Sequential I/O should always be faster than random access I/O.

x86 is little endian, so we use little endian integers so we don't need to waste cycles on endian transformation.

We store all data for the same field next to each other in the data structure. This is in opposition to say packing all of resource A's data then resource B's, etc. We do this to help maximize locality for similar data. This can help with performance because often the same field for multiple resources is accessed together. e.g. an importer will access a bunch of module bytecode entries at the same time. This locality helps minimize the number of pages that must be read. Locality can also help yield higher compression ratios.

Everything is designed to facilitate a reader leveraging 0-copy. If a reader has the data structure in memory, we don't want to require it to copy memory in order to reference entries. In Rust speak, we should be able to hold `&[u8]` references everywhere.

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.

8.4 Potential Future Features

This data structure is robust enough to be used by PyOxidizer to power importing of every Python module used by a Python interpreter. However, there are various aspects that could be improved.

8.4.1 Compression

A potential area for optimization is use of general compression. Various fields should compress well - either in streaming mode or by utilizing compression dictionaries. Compression would undermine 0-copy, of course. But in environments where we want to optimize for size, it could be desirable.

8.4.2 Platform Portability

Currently, filesystem paths are encoded as platform native. That means `[u8]` on POSIX and `[u16]` on Windows. This isn't portable.

Most filenames are likely ASCII or UTF-8 safe. For the common case where we don't need platform-native filenames to preserve subtle encoding differences, we could express paths as a simpler string type.

CHAPTER 9

The pyembed Rust Crate

The pyembed Rust crate facilitates the embedding of a Python interpreter in a Rust binary.

The crate provides an API for instantiating and controlling an embedded Python interpreter. It also defines a custom *meta path importer* that can be used to import Python resources (such as module bytecode) from memory.

9.1 Crate Configuration

9.1.1 Cargo Features to Control Building

The pyembed crate has a set of build-mode-* Cargo feature flags to control how build artifacts are created and consumed.

The features are described in the following sections.

build-mode-default

This is the default build mode. It is enabled by default.

This build mode uses default Python linking behavior and feature detection as implemented by the cpython and python3-sys crates. It will attempt to find a python in PATH or from the PYTHON_SYS_EXECUTABLE environment variable and dynamically link against it.

This is the default mode for convenience, as it enables the pyembed crate to build in the most environments. However, the built binaries will have a dependency against a foreign libpython and likely aren't suitable for distribution.

pyembed has a dependency on Python 3.8+. If an older Python is detected, it can result in build errors, including unresolved symbol errors.

build-mode-pyoxidizer-exe

A pyoxidizer executable will be run to generate build artifacts.

The path to this executable can be defined via the `PYOXIDIZER_EXE` environment variable. Otherwise `PATH` will be used.

At build time, `pyoxidizer run-build-script` will be run. A `PyOxidizer` configuration file will be discovered using PyOxidizer's heuristics for doing so. `OUT_DIR` will be set if running from `cargo`, so a `pyoxidizer.bzl` next to the main Rust project being built should be found and used.

`pyoxidizer run-build-script` will resolve the default build script target by default. To override which target should be resolved, specify the target name via the `PYOXIDIZER_BUILD_TARGET` environment variable. e.g.:

```
$ PYOXIDIZER_BUILD_TARGET=build-artifacts cargo build
```

build-mode-prebuilt-artifacts

This mode tells the build script to reuse artifacts that were already built. (Perhaps you called `pyoxidizer build` or `pyoxidizer run-build-script` outside the context of a normal `cargo build`.)

In this mode, the build script will look for artifacts in the directory specified by `PYOXIDIZER_ARTIFACT_DIR` if set, falling back to `OUT_DIR`. See *Build Artifacts* for documentation on the required artifacts.

build-mode-standalone

Do not attempt to invoke `pyoxidizer` or find artifacts it would have built. It is possible to build the `pyembed` crate in this mode if the `rust-cpython` and `python3-sys` crates can find a Python interpreter. But, the `pyembed` crate may not be usable or work in the way you want it to.

This mode is intended to be used for performing quick testing on the `pyembed` crate. It is quite possible that linking errors will occur in this mode unless you take additional actions to point Cargo at appropriate libraries.

cpython-link-unresolved-static

Configures the link mode of the `cpython` crate to use a static `pythonXY` library without resolving the symbol at its own build time. The `pyembed` crate or a crate building it will need to emit `cargo:rustc-link-lib=static=pythonXY` and any `cargo:rustc-link-search=native={}` lines to specify an explicit `pythonXY` library to link against.

This is the link mode used to produce self-contained binaries containing `libpython` and `pyembed` code.

cpython-link-default

Configures the link mode of the `cpython` crate to use default semantics. The crate's build script will find a prebuilt Python library by querying the `python` defined by `PYTHON_SYS_EXECUTABLE` or found on `PATH`. See the `cpython` crate's documentation for more.

This link mode should be used when linking against an existing `libpython` that can be found by the `cpython` crate's build script.

9.1.2 Build Artifacts

When using `build-mode-prebuilt-artifacts` or `build-mode-pyoxidizer-exe`, the `pyembed` crate consumes special artifacts as part of its build process to provide the embedded Python interpreter. These artifacts are typically generated by PyOxidizer. However, there is nothing stopping anyone from producing equivalent artifacts via other means and having `pyembed` consume them.

The way this mode works is the build script is pointed at a directory containing artifacts. The only required artifact is a `cargo_metadata.txt` file. This file contains lines which will be printed to stdout by the crate build script. These lines typically contain `cargo:` lines, which influence Cargo's configuration for the crate.

The `cargo:` lines **must** define a pre-built `pythonXY` library to link against. That library name is literally `pythonXY` and `XY` is not a placeholder for a version string!

Use cases like PyOxidizer derive a custom library containing Python's core symbols. The `cargo:` lines for this use case will look something like the following:

```
cargo:rustc-link-lib=depend0
cargo:rustc-link-lib=depend1
cargo:rustc-link-lib=static=depend2
cargo:rustc-link-lib=static=depend3
cargo:rustc-link-lib=static=pythonXY
cargo:rustc-link-search=native=/path/to/libraries
```

Essentially what PyOxidizer does is compile a custom library containing Python. This will be named `pythonXY.lib` or `pythonXY.dll` on Windows and `libpythonXY.a` or `libpythonXY.so` on UNIX platforms. It then lists link library dependencies as needed and registers the generated `pythonXY` library to be linked from the context of the `pyembed` crate.

Deriving a custom library containing Python is fairly complex! From the perspective of `build-mode-prebuilt-artifacts`, all that is strictly needed is for the `cargo_metadata.txt` to define how to link against a `pythonXY` library. It is even possible to alias `pythonXY` to an existing Python library already on your system (this is effectively what `build-mode-default` does). So a minimal `cargo_metadata.txt` might look something like this:

cargo:rustc-link-lib=pythonXY:python3.9 cargo:rustc-link-search=native=/path/to/directory/containing/python/library

9.2 Controlling Python from Rust Code

9.2.1 Initializing a Python Interpreter

Initializing an embedded Python interpreter in your Rust process is as simple as calling `pyembed::MainPythonInterpreter::new(config: OxidizedPythonInterpreterConfig)`.

The hardest part about this is constructing the `pyembed::OxidizedPythonInterpreterConfig` instance.

9.2.2 Using a Python Interpreter

Once you've constructed a `pyembed::MainPythonInterpreter` instance, you can obtain a `cpython::Python` instance via `.acquire_gil()` and then use it:

```
fn do_it(interpreter: &MainPythonInterpreter) -> {
    let py = interpreter.acquire_gil();

    match py.eval("print('hello, world')") {
        Ok(_) => print("python code executed successfully"),
        Err(e) => print("python error: {:?}", e),
    }
}
```

Since CPython's API relies on static variables (sadly), if you really wanted to, you could call out to CPython C APIs directly (probably via the bindings in the `python3-sys` crate) and they would interact with the interpreter started by the `pyembed` crate. This is all `unsafe`, of course, so tread at your own peril.

9.2.3 Finalizing the Interpreter

`pyembed::MainPythonInterpreter` implements `Drop` and it will call `Py_FinalizeEx()` when called. So to terminate the Python interpreter, simply have the `MainPythonInterpreter` instance go out of scope or drop it explicitly.

9.2.4 A Note on the `pyembed` APIs

The `pyembed` crate is highly tailored towards PyOxidizer's default use cases and the APIs are not considered extremely well polished.

While the functionality should work, the ergonomics may not be great.

It is a goal of the PyOxidizer project to support Rust programmers who want to embed Python in Rust applications. So contributions to improve the quality of the `pyembed` crate will likely be greatly appreciated!

9.3 Adding Extension Modules At Run-Time

A Python extension module is effectively a callable function defined in a library somewhere.

The `pyembed` crate supports registering Python extension modules multiple ways.

9.3.1 Statically Linked Extension Modules

You can inform the `pyembed` crate about the existence of additional Python extension modules which are statically linked into the binary.

To do this, you will need to populate the `extra_extension_modules` field of the `OxidizedPythonInterpreterConfig` Rust struct used to construct the Python interpreter. Simply add an entry defining the extension module's `import` name and a pointer to its C initialization function (often named `PyInit_<name>`. e.g. if you are defining the extension module `foo`, the initialization function would be `PyInit_foo` by convention.

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.

9.3.2 Dynamically Linked Extension Modules

If you have an extension module provided as a shared library (this is typically how Python extension modules work), it will be possible to load this extension module provided that the Python interpreter supports loading dynamically linked Python extension modules.

There is not yet an explicit Rust API for loading additional dynamically linked extension modules. It is theoretically possible to add an entry to the parsed embedded resources data structure. The path of least resistance is likely to enable the standard filesystem importer and put your shared library extension module somewhere on Python's `sys.path`. (This is how extension modules are typically loaded.)

CHAPTER 10

PyOxidizer for Rust Developers

PyOxidizer is implemented in Rust. Binaries built with PyOxidizer are also built with Rust using standard Rust projects.

While the existence of Rust should be abstracted away from most users (aside from the existence of the install dependency and build output), a target audience of PyOxidizer is Rust developers who want to embed Python in a Rust project or Python developers who want to leverage more Rust in their Python applications.

Follow the links below to learn how PyOxidizer uses Rust and how Rust can be leveraged to build more advanced applications embedding Python.

10.1 Using Cargo with PyOxidizer Source Checkouts

PyOxidizer's source repository consists of multiple Rust projects/crates. At the root of the repository is a `Cargo.toml` defining a workspace consisting of all these crates.

Important: Building various Rust crates from source can be extremely brittle and a top-level `cargo build` will likely encounter multiple build failures.

If you want to run `cargo` from a PyOxidizer source checkout, you will likely want to limit the invocation to a single crate at a time to ensure things can build.

The following sections detail how to build various crates inside a source checkout.

10.1.1 `pyoxidizer` Crate

Building the `pyoxidizer` crate in isolation (e.g. `cargo build -p pyoxidizer`) should *just work*, as it is a pretty typical Rust crate.

Perhaps the only special property of this crate is that it defines both a library and an executable. So you may want to limit operations to a specific binary. e.g. `cargo build --bin pyoxidizer` or `cargo test --bin pyoxidizer`.

10.1.2 `python-packed-resources` Crate

This is a standard Rust crate and should always build without issue. e.g. `cargo build -p python-packed-resources`.

10.1.3 `python-packaging` Crate

This is a standard Rust crate and should always build without issue. e.g. `cargo build -p python-packaging` or `cargo test -p python-packaging`.

10.1.4 `pyembed` Crate

The `pyembed` crate provides the bulk of the run-time functionality for binaries embedding Python interpreters. Because the crate needs to consult with a Python interpreter at build time and link against it and because it needs to exchange state with PyOxidizer, its build configuration is... *special*.

Important: Almost all workspace build failures are somehow related to the `pyembed` crate.

The `pyembed` crate defines various *features* to control how it is built. See *Crate Configuration* for details.

In its default configuration, a Python 3.9 executable needs to be found on `PATH`. If said executable can't be found, you'll get a `No python interpreter found of version 3.*` error at build time.

To work around this, add a `python3.9` or `python3` executable to `PATH` or run `cargo build` with the `PYTHON_SYS_EXECUTABLE` environment variable pointing to a specific Python 3 executable. e.g.

```
$ PYTHON_SYS_EXECUTABLE=/path/to/python3.9 cargo build -p pyembed
```

10.1.5 `oxidized-importer` Crate

This crate is a very small shim around the `pyembed` crate which builds the `pyembed` crate in a specific manner so it provides just the functionality needed for *oxidized_importer Python Extension*.

Because this crate is a thin shim, the caveats that apply to building `pyembed` apply to it as well.

10.2 Rust Projects

PyOxidizer uses Rust projects to build binaries embedding Python.

If you just have a standalone configuration file (such as when running `pyoxidizer init-config-file`), a temporary Rust project will be created as part of building binaries. That project will be built, its build artifacts copied, and the temporary project will be deleted.

If you use `pyoxidizer init-rust-project` to initialize a `PyOxidizer` application, the Rust project exists side-by-side with the `PyOxidizer` configuration file and can be modified like any other Rust project.

10.2.1 Layout

Generated Rust projects all have a similar layout:

```
$ find pyapp -type f | grep -v .git
.cargo/config
Cargo.toml
build.rs
pyoxidizer.bzl
src/main.rs
```

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:

```
[package]
build = "build.rs"

[dependencies]
pyembed = ...
```

These lines declare a dependency on the `pyembed` package, which holds the smarts for embedding Python in a binary.

In addition, the `build = "build.rs"` tells runs a script that hooks up the output of the `pyembed` crate with this project.

Next let's look at `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()`. Succinctly, we instantiate and run an embedded Python interpreter. That's our executable.

The `pyoxidizer.bzl` is our auto-generated *PyOxidizer configuration file*.

10.2.2 Cargo Configuration

Linking a custom libpython into the final Rust binary can be finicky, especially when statically linking on Windows.

The auto-generated `.cargo/config` file defines some custom compiler settings to enable things to work. However, this only works for some configurations. The file contains some commented out settings that may need to be set for some configurations (e.g. the `standalone_static` Windows distributions). Please consult this file if running into build errors when not building through `pyoxidizer`.

10.3 Controlling Python From Rust Code

PyOxidizer can be used to embed Python in a Rust application.

This page documents what that looks like from a Rust code perspective.

10.3.1 Interacting with the `pyembed` Crate

When writing Rust code to interact with a Python interpreter, your primary area of contact will be with the `pyembed` crate.

The `pyembed` crate is a standalone crate maintained as part of the PyOxidizer project. This crate provides the core run-time functionality for PyOxidizer, such as the implementation of *PyOxidizer's custom importer*. It also exposes a high-level API for initializing a Python interpreter and running code in it.

See *The pyembed Rust Crate* for full documentation on the `pyembed` crate. *Controlling Python from Rust Code* in particular describes how to interface with the embedded Python interpreter.

The following documentation will be unique to PyOxidizer's use of the `pyembed` crate.

10.3.2 Using the Default `OxidizedPythonInterpreterConfig`

When using a PyOxidizer-generated Rust project and that project is configured to use PyOxidizer to build (the default), that project/crate's build script will call into PyOxidizer to emit various build artifacts. This will process the PyOxidizer configuration file and write some files somewhere.

One of the files generated is a Rust source file containing a `fn default_python_config() -> pyembed::OxidizedPythonInterpreterConfig` which emits a `pyembed::OxidizedPythonInterpreterConfig` using the configuration from the PyOxidizer configuration file. This configuration is based off the *PythonInterpreterConfig* defined in the PyOxidizer Starlark configuration file.

The crate's build script will set the `PYOXIDIZER_DEFAULT_PYTHON_CONFIG_RS` environment variable to the path to this file, exposing it to Rust code.

This all means that to use the auto-generated `pyembed::OxidizedPythonInterpreterConfig` instance with your Rust application, you simply need to do something like the following:

```
include!(env!("PYOXIDIZER_DEFAULT_PYTHON_CONFIG_RS"));

fn create_interpreter() -> Result<pyembed::MainPythonInterpreter> {
    // Calls function from include!()'d file.
    let config: pyembed::OxidizedPythonInterpreterConfig = default_python_config();

    pyembed::MainPythonInterpreter::new(config)
}
```

Using a Custom `OxidizedPythonInterpreterConfig`

If you don't want to use the default `pyembed::OxidizedPythonInterpreterConfig` instance, that's fine too! However, this will be slightly more complicated.

First, if you use an explicit `OxidizedPythonInterpreterConfig`, the *PythonInterpreterConfig* Starlark type defined in your PyOxidizer configuration file doesn't matter that much. The primary purpose of this Starlark type is to derive the default `OxidizedPythonInterpreterConfig` Rust struct. And if you are using your own custom `OxidizedPythonInterpreterConfig` instance, you can ignore most of the arguments when creating the `PythonInterpreterConfig` instance.

An exception to this is the `raw_allocator` argument/field. If you are using a custom allocator (like jemalloc, mimalloc, or snmalloc), you will need to enable a Cargo feature when building the `pyembed` crate or else you will get a run-time error that the specified allocator is not available.

`pyembed::OxidizedPythonInterpreterConfig::default()` can be used to construct a new instance, pre-populated with default values for each field. The defaults should match what the *PythonInterpreterConfig* Starlark type would yield.

The main catch to constructing the instance manually is that the custom *meta path importer* won't be able to service Python `import` requests unless you populate a few fields. In fact, if you just use the defaults, things will blow up pretty hard at run-time:

```
$ myapp
Fatal Python error: initfsencoding: Unable to get the locale encoding
```

(continues on next page)

(continued from previous page)

```
ModuleNotFoundError: No module named 'encodings'

Current thread 0x00007fa0e2cbe9c0 (most recent call first):
Aborted (core dumped)
```

What's happening here is that Python interpreter initialization hits a fatal error because it can't `import encodings` (because it can't locate the Python standard library) and Python's C code is exiting the process. Rust doesn't even get the chance to handle the error, which is why we're seeing a segfault.

The reason we can't `import encodings` is twofold:

1. The default filesystem importer is disabled by default.
2. No Python resources are being registered with the `OxidizedPythonInterpreterConfig` instance.

This error can be addressed by working around either.

To enable the default filesystem importer:

```
let mut config = pyembed::OxidizedPythonInterpreterConfig::default();
config.filesystem_importer = true;
config.sys_paths.push("/path/to/python/standard/library");
```

As long as the default filesystem importer is enabled and `sys.path` can find the Python standard library, you should be able to start a Python interpreter.

Hint: The `sys_paths` field will expand the special token `$ORIGIN` to the directory of the running executable. So if the Python standard library is in e.g. the `lib` directory next to the executable, you can do something like `config.sys_paths.push("$ORIGIN/lib")`.

If you want to use the custom *PyOxidizer Importer* to import Python resources, you will need to update a handful of fields:

```
let mut config = pyembed::OxidizedPythonInterpreterConfig::default();
config.packed_resources = ...;
config.oxidized_importer = true;
```

The `packed_resources` field defines a reference to *packed resources data* (a `PackedResourcesSource` enum. This is a custom serialization format for expressing *resources* to make available to a Python interpreter. See *Python Packed Resources* for more. The easiest way to obtain this data blob is by using PyOxidizer and consuming the `packed-resources` build artifact/file, likely though `include_bytes!`. *OxidizedFinder Python Type* can also be used to produce these data structures.

Finally, setting `oxidized_importer = true` is necessary to enable `OxidizedFinder`.

10.4 Porting a Python Application to Rust

PyOxidizer can be used to gradually port a Python application to Rust. What we mean by this is that Python code in an application would slowly be rewritten in Rust.

10.4.1 Overview

When porting a Python application to Rust, the goal is to port Python code - and possibly Python C extension code - to Rust. Parts of the Rust code will presumably need to call into Python code and vice-versa.

When porting code to Rust, there are essentially two *flavors* of Rust code that will be written and executed:

1. *Vanilla* Rust code
2. *Python-flavored* Rust code

Vanilla Rust code is standard Rust code. It is what you would write if authoring a Rust-only project.

Python-flavored Rust code is Rust code that interacts with the Python C API. It is regular Rust code, of course, but it is littered with references to *PyObject* and function calls into the Python C API (although these function calls may be abstracted so you don't have to use `unsafe`).

These different *flavors* of Rust code dictate different approaches to porting. Both *flavors*/approaches can be used simultaneously when porting an application to Rust.

Vanilla Rust code will supplement the boilerplate Rust code that PyOxidizer uses to define and build a standalone executable embedded Python. See *Extending Rust Projects* for more.

Python-flavored Rust code typically involves writing Python extension modules in Rust. In this approach, you create a Python extension modules implemented in Rust and then make them available to the Python interpreter, which is managed by a Rust project.

10.4.2 Extending Rust Projects

When building an application from a standalone `pyoxidizer.bzl` file, PyOxidizer creates and builds a temporary, boilerplate Rust project behind the scenes. This Rust project has just enough code to initialize and run an embedded Python interpreter. That's the extent of the Rust code.

PyOxidizer also supports persistent Rust projects. In this mode, you have full control over the Rust project and can add custom Rust code to it as you desire. In this mode, you can run Rust code independent of the Python interpreter.

Supplementing the Rust code contained in your executable gives you the power to run arbitrary Rust code however you see fit. Here are some common scenarios this can enable:

- Implementing argument parsing in Rust instead of Python. This could allow you to parse out the sub-command being invoked and dispatch to pure Rust code paths if possible, falling back to running Python code only if necessary.
- Running a *forking* server, which doesn't start a Python interpreter until an event occurs.
- Starting a thread with a high-performance application component implemented in Rust. For example, you could run a thread servicing a high-performance logging subsystem or HTTP server implemented in Rust and have that thread interact with a Python interpreter via a pipe or some other handle.

Getting Started

To extend a Rust project with custom Rust code, you'll first want to materialize the boilerplate Rust project used by PyOxidizer:

```
$ pyoxidizer init-rust-project myapp
```

See *Rust Projects* for details on the files materialized by this command.

If you are using version control, now would be a good time to add the created files to version control. e.g.:

```
$ git add myapp
$ git commit -m 'create boilerplate PyOxidizer project'
```

From here, your next steps are to modify the Rust project to do something new and different.

The auto-generated `src/main.rs` file contains the `main()` function used as the entrypoint for the Rust executable. The default file will simply instantiate a Python interpreter from a configuration, run that interpreter, then exit the process.

To extend your application with custom Rust code, simply add custom code to `main()`. e.g.

```
fn main() {
    println!("hello from Rust!")

    // Code auto-generated by ``pyoxidizer init-rust-project`` goes here.
    // ...
}
```

That is literally all there is to it!

To build your custom Rust project, `pyoxidizer build` is the most robust way to do that. But it is also possible to use `cargo build`.

What Can Go Wrong

`pyoxidizer` Not Found or Rust Code Version Mismatch

When using `cargo build`, the `pyoxidizer` executable will be invoked behind the scenes. This requires that executable to be on `PATH` and for the version to be compatible with the Rust code you are trying to build. (The Rust APIs do change from time to time.)

If the `pyoxidizer` executable is not on `PATH` or its version doesn't match the Rust code, you can forcefully tell the Rust build system which `pyoxidizer` executable to use:

```
$ PYOXIDIZER_EXE=/path/to/pyoxidizer cargo build
```

`thread 'main' panicked at 'jemalloc is not available in this build configuration'`

If you see this error, the problem is that the Python interpreter configuration says to use *jemalloc* as the memory allocator but the Rust project was built without *jemalloc* support. This is likely because the default Rust project features in `Cargo.toml` don't include *jemalloc* by default.

You can resolve this issue by either disabling jemalloc in the Python configuration or by enabling jemalloc in Rust.

To disable jemalloc, open your `pyoxidizer.bzl` file and find the definition of `allocator_backend`. You can set it to `raw_allocator="default"` so Python uses the system memory allocator instead of jemalloc.

To enable jemalloc, you have a few options.

First, you could build the Rust project with jemalloc support:

```
$ cargo build --features allocator-jemalloc
```

Or, you modify `Cargo.toml` so the *allocator-jemalloc* feature is enabled by default:

```
.. code-block:: toml
```

[features] default = ["build-mode-pyoxidizer-exe", "allocator-jemalloc"]

jemalloc is typically a faster allocator than the system allocator. So if you care about performance, you may want to use it.

10.4.3 Implementing Python Extension Modules in Rust

If you want to port a Python application to Rust, chances are that you will need to have Rust and Python code interact with each other. A common way to do this is to implement Python extensions in Rust so that Rust code will be invoked as a Python interpreter is running.

There are two ways Rust-implemented Python extension modules can be consumed by PyOxidizer:

1. Define them via Python packaging tools (e.g. via a `setup.py` file for your Python package).
2. Define them in Rust code and register them as a *built-in* extension module.

Python Built Rust Extension Modules

If you've defined a Rust Python extension module via a Python package build tool (e.g. inside a `setup.py`), PyOxidizer should automatically detect said extension module as part of packaging the corresponding Python package: there is no need to take special action to tell PyOxidizer it is a Rust extension, as this is all handled by Python packaging tools invoked as part of processing your `pyoxidizer.bzl` file.

See *Packaging User Guide* for more.

The topic of authoring Python extension modules implemented in Rust is arguably outside the scope of this documentation. A search engine search for `Rust Python extension` should set you on the right track.

Built-in Rust Extension Modules

A Python extension module is defined as a `PyInit__<name>` function which is called to initialize an extension module. Typically, Python extension modules are compiled as standalone shared libraries, which are then loaded into a process, after which their `PyInit__<name>` function is called.

But Python has an additional mechanism for defining extension modules: *built-ins*. A *built-in* extension module is simply an extension module whose `PyInit__<name>` function is already present in the process address space. Typically, these are extensions that are part of the Python distribution itself and are compiled directly into `libpython`.

When you instantiate a Python interpreter, you give it a list of the available *built-in* Python extension modules. And PyOxidizer's `pyembed` crate allows you to supplement the default list with custom extensions.

To use *built-in* extension modules implemented in Rust, you'll need to implement said extension module in Rust, either as part of your application's Rust crate or as part of a different crate. Either way, you'll need to extend the boilerplate Rust project code (see *Extending Rust Projects*) and tell it about additional *built-in* extension modules. See *Adding Extension Modules At Run-Time* for instructions on how to do this.

The tricky part here is implementing your Rust extension module.

You probably want to use the cpython or PyO3 Rust crates for interfacing with the CPython API, as these provide an interface that is more ergonomic and doesn't require use of `unsafe { }`. Use of these crates is beyond the scope of the PyOxidizer documentation.

If you attempt to use the `cpython` or `PyO3` macros for defining a Python extension module, you'll likely run into problems because these assume that extension modules are standalone shared libraries, which isn't the case for *built-in* extension modules!

If you attempt to use a separate Rust crate to define your extension module, you may run into Python symbol issues at link time because the build system for the `cpython` and `PyO3` crates will use their own logic for locating a

Python interpreter and that interpreter may not have a configuration that is compatible with the one embedded in your PyOxidizer binary!

At the end of the day, all you need to register a *built-in* extension module with PyOxidizer is an `extern "C" fn () -> *mut python3_sys::PyObject`. Here is the boilerplate for defining a Python extension module in Rust (this uses the `cpython` crate).

```
use python3_sys as pyffi;
use cpython::{PyErr, PyModule, PyObject};

static mut MODULE_DEF: pyffi::PyModuleDef = pyffi::PyModuleDef {
    m_base: pyffi::PyModuleDef_HEAD_INIT,
    m_name: std::ptr::null(),
    m_doc: std::ptr::null(),
    m_size: std::mem::size_of::<ModuleState>() as isize,
    m_methods: 0 as *mut _,
    m_slots: 0 as *mut _,
    m_traverse: None,
    m_clear: None,
    m_free: None,
};

#[allow(non_snake_case)]
pub extern "C" fn PyInit_my_module() -> *mut pyffi::PyObject {
    let py = unsafe { cpython::Python::assume_gil_acquired() };

    unsafe {
        if MODULE_DEF.m_name.is_null() {
            MODULE_DEF.m_name = "my_module".as_ptr() as *const _;
            MODULE_DEF.m_doc = "usage docs".as_ptr() as *const _;
        }
    }

    let module = unsafe { pyffi::PyModule_Create(&mut MODULE_DEF) };

    if module.is_null() {
        return module;
    }

    let module = match unsafe { pyffi::from_owned_ptr(py, module).cast_into::
↪<PyModule>(py) } {
        Ok(m) => m,
        Err(e) => {
            PyErr::from(e).restore(py);
            return std::ptr::null_mut();
        }
    };

    match module_init(py, &module) {
        Ok(()) => module.into_object().steal_ptr(),
        Err(e) => {
            e.restore(py);
            std::ptr::null_mut()
        }
    }
}
```

If you want a concrete example of what this looks like and how to do things like define Python types and have Python functions implemented in Rust, do a search for `PyInit_oxidized_importer` in the source code of the

`pyembed` crate (which is part of the PyOxidizer repository) and go from there.

The documentation for authoring Python extension modules and using the Python C API is well beyond the scope of this document. A good place to start is the official documentation.

CHAPTER 11

Shipping Applications with `tugger`

The Tugger project aims to make it easy to ship applications. It does so by implementing generic functionality related to application distribution in a myriad (fleet?) of individual, domain-specific crates. See *Modular Crate Architecture* for more. Tugger supports generating distributable artifacts in common formats such as Windows `.msi` installers, Debian `.deb` files, and Snapcraft `.snap` files.

Tugger's Rust crates can be consumed as regular Rust library crates by any project and are explicitly designed for this use case. Tugger also defines a Starlark dialect (Starlark is a Python-like configuration language), enabling applications to define packaging functionality in configuration files, which Tugger can execute. The Starlark dialect is effectively a scriptable interface to Tugger's Rust internals.

Tugger is part of the PyOxidizer Project and is developed inside the PyOxidizer repository at https://github.com/indygreg/PyOxidizer. However, Tugger is designed to be a standalone project and doesn't require PyOxidizer.

11.1 Overview

Tugger aims to be a generic tool to help application maintainers ship their applications to end-users.

Tugger can be thought of a specialized build system for distributable artifacts (Windows MSI installers, Debian packages, RPMs, etc). However, Tugger itself is generally not concerned with details of how a particular file is built: Tugger's role is to consume existing files and *package* them into artifacts that are distributed/installed on other machines.

11.1.1 Designed to Be Platform Agnostic

An explicit goal of Tugger is to be platform agnostic and to have as much functionality implemented in-process. For example, it should be possible to produce a Linux `.deb` from Windows, a Windows MSI installer from macOS, or a macOS DMG from Linux without any out-of-process dependencies.

Tugger attempts to implement packaging functionality in Rust with minimal dependence on external tools. For example, RPMs and Debian packages are built by constructing the raw archive files using Rust code rather than calling out to tools like `rpmbuild` or `debuild`. This enables Tugger to build artifacts that don't target the current architecture or operating system.

While Tugger may not achieve this goal for all distributable formats and architectures, it is something that Tugger strives to do.

11.1.2 File Centric View

Tugger attempts to take a file-centric view towards packaging. This helps achieve platform independent and *cross-compiling*. What this means in practice is many of Tugger's packaging facilities operate by taking an input set of files and assembling them into some other distributable format. Contrast this with specialized tools for each distributable format, which generally invoke a custom build system and have domain-specific configuration files.

A side-effect of this decision is that Tugger is often not aware of build systems: it is often up to you to script Tugger to produce the files you wish to distribute.

11.1.3 Modular Crate Architecture

Tugger is composed of a series - a *fleet* if you will - of Rust crates. Each Rust crate provides domain-specific functionality. While the Rust crates are part of the Tugger project, an attempt is made to implement them such that they can be used outside of Tugger. For example, the `tugger-debian` crate contains generic code for building `.deb` files from scratch.

The following crates compose Tugger's crate *fleet*:

`tugger-binary-analysis` Analyze platform native binaries. Finds library dependencies. Identifies Linux distribution compatibility. Etc.

`tugger-common` Shared functionality required by multiple crates. This entails things like downloading files, shared test code, etc.

`tugger-debian` Debian packaging primitives. Parsing and serializing control files. Writing `.deb` files.

`tugger-file-manifest` A virtual manifest of a collection of files. Virtual file manifests are used throughout Tugger to represent a collection of files, their content, and file metadata.

`tugger-licensing` Functionality related to software licensing.

`tugger-licensing-net` Functionality related to software licensing requiring network access.

`tugger-rpm` RPM packaging primitives.

`tugger-snapcraft` Snapcraft packaging. Represent `snapcraft.yaml` files. Invoke `snapcraft` to produce `.snap` files.

`tugger-windows` Windows-specific functionality. Finding the Microsoft SDK and Visual C++ Redistributable files. Signing Windows binaries.

`tugger-wix` Interface to the WiX Toolset (produces Windows `.msi` and `.exe` installers). Can build Windows installers with little-to-no knowledge about how the WiX Toolset works.

`tugger` The primary crate. Implements Starlark dialect and driver code for running it. This crate has minimal use as a library, as most library functionality is within the domain-specific crates.

11.2 Tugger Starlark Dialect

Tugger uses Starlark files to configure run-time behavior.

Starlark is a subset of Python intended to be used as a configuration language and the syntax should be familiar to any Python programmer.

Tugger defines its own *dialect* of Starlark - types and functions - specific to Tugger.

11.2.1 Global Symbols

This document lists every single global type, variable, and function available in Tugger's Starlark execution environment.

The Starlark environment contains symbols from the following:

- Starlark built-ins
- Tugger's Dialect (documented below)

Global Types

Tugger's Starlark dialect defines the following custom types:

FileContent Represents the content of a file on the filesystem.

FileManifest Represents a mapping of filenames to file content.

SnapApp Represents an application inside a `snapcraft.yaml` file.

SnapPart Represents a part inside a `snapcraft.yaml` file.

Snap Represents a `snapcraft.yaml` file.

SnapcraftBuilder Manages the environment and invocations of the `snapcraft` command.

WiXBundleBuilder Produce a Windows exe installer containing multiple installers using WiX.

WiXInstaller Produce a Windows installer using WiX.

WiXMSIBuilder Produce a Windows MSI installer with common installer features using WiX.

Global Functions

Tugger's Starlark dialect defines the following global functions:

glob() Collect files from the filesystem.

11.2.2 Functions for Interacting with the Filesystem

`glob()`

The `glob()` function resolves file patterns to a *FileManifest*.

This function accepts the following arguments:

`include` (`list` of `string`) Defines file patterns that will be matched using the `glob` Rust crate. If patterns begin with `/` or look like a filesystem absolute path, they are absolute. Otherwise they are evaluated relative to the directory of the current config file.

`exclude` (`list` of `string` or `None`) File patterns used to exclude files from the result. All patterns in `include` are evaluated before `exclude`.

`strip_prefix` (`string` or `None`) Prefix to strip from the beginning of matched files. `strip_prefix` is stripped after `include` and `exclude` are processed.

Returns a *FileManifest*.

11.2.3 FileContent

This type represents the content of a single file.

11.2.4 FileManifest

The `FileManifest` type represents a set of files and their content.

`FileManifest` instances are used to represent things like the final filesystem layout of an installed application.

Conceptually, a `FileManifest` is a dict mapping relative paths to file content.

Methods

FileManifest.add_manifest()

This method overlays another `FileManifest` on this one. If the other manifest provides a path already in this manifest, its content will be replaced by what is in the other manifest.

FileManifest.add_path()

This method adds a file on the filesystem to the manifest.

The following arguments are accepted:

path (`string`) The filesystem path to add.

strip_prefix (`string`) The string prefix to strip from the path. The remaining path will be stored in the manifest.

force_read (`bool`) Whether to read the file data into memory now.

This can be set when reading temporary files.

Defaults to `False`.

FileManifest.install()

This method writes the content of the `FileManifest` to a directory specified by `path`. The path is evaluated relative to the path specified by `BUILD_PATH`.

If `replace` is True (the default), the destination directory will be deleted and the final state of the destination directory should exactly match the state of the `FileManifest`.

11.2.5 SnapApp

The `SnapApp` type represents an application entry in a `snapcraft.yaml` file. Specifically, this type represents the values of `apps.<app-name>` keys.

See https://snapcraft.io/docs/snapcraft-yaml-reference for more documentation.

Constructors

SnapApp()

`SnapApp()` creates an empty instance. It accepts no arguments.

Attributes

Instances of `SnapApp` expose attributes that map to the keys within `apps.<app-name>` entries in `snapcraft.yaml` configuration files.

Currently the attributes are write only.

Setting an attribute value to `None` has the side-effect of removing that attribute from the serialized `snapcraft.yaml` file.

See https://snapcraft.io/docs/snapcraft-yaml-reference for detailed documentation about what each attribute means.

adapter

(Optional[string])

autostart

(Optional[string])

command_chain

(Optional[list[string]])

command

(Optional[string])

common_id

(Optional[string])

daemon

(Optional[string])

desktop

(Optional[string])

environment

(Optional[list[string]])

extensions

(Optional[list[string]])

listen_stream

(Optional[string])

passthrough

(Optional[dict[string, string]])

plugs

(Optional[list[string]])

post_stop_command

(Optional[string])

restart_condition

(Optional[string])

slots

(Optional[list[string]])

stop_command

(Optional[string])

stop_timeout

(Optional[string])

timer

(Optional[string])

socket_mode

(Optional[int])

socket

(Optional[dict[string]])

11.2.6 SnapPart

The SnapPart type represents a part entry in a snapcraft.yaml file. Specifically, this type represents the values of parts.<part-name> keys.

See https://snapcraft.io/docs/snapcraft-yaml-reference for more documentation.

Constructors

SnapPart()

SnapPart() creates an empty instance. It accepts no arguments.

Attributes

Instances of SnapPart expose attributes that map to the keys within parts.<part-name> entries in snapcraft.yaml configuration files.

Currently the attributes are write only.

Setting an attribute value to None has the side-effect of removing that attribute from the serialized snapcraft.yaml file.

See https://snapcraft.io/docs/snapcraft-yaml-reference for detailed documentation about what each attribute means.

after

(Optional[list[string]])

build_attributes

(Optional[list[string]])

build_environment

(Optional[list[dict[string, string]]])

build_packages

(Optional[list[string]])

build_snaps

(Optional[list[string]])

filesets

(Optional[dict[string, list[string]]])

organize

(Optional[dict[string, string]])

override_build

(Optional[string])

override_prime

(Optional[string])

override_pull

(Optional[string])

override_stage

(Optional[string])

parse_info

(Optional[string])

plugin

(Optional[string])

prime

(Optional[list[string]])

source_branch

(Optional[string])

source_checksum

(Optional[string])

source_commit

(Optional[string])

source_depth

(Optional[int])

source_subdir

(Optional[string])

source_tag

(Optional[string])

source_type

(Optional[string])

source

(Optional[string])

stage_packages

(Optional[list[string]])

stage_snaps

(Optional[list[string]])

stage

(Optional[list[string]])

11.2.7 Snap

The `Snap` type represents an entire `snapcraft.yaml` file.

See https://snapcraft.io/docs/snapcraft-yaml-reference for more documentation.

Constructors

Snap()

`Snap()` creates an instance initialized with required parameters. It accepts the following arguments:

name (`string`)

version (`string`)

summary (`string`)

description (`string`)

Attributes

Instances of `Snapt` expose attributes that map to the keys within `snapcraft.yaml` files.

Currently the attributes are write only.

Setting an attribute value to `None` has the side-effect of removing that attribute from the serialized `snapcraft.yaml` file.

See https://snapcraft.io/docs/snapcraft-yaml-reference for detailed documentation about what each attribute means.

adopt_info

(`Optional[string]`)

apps

(`Optional[dict[string, SnapApp]]`)

architectures

(`Optional[dict["build_on" | "run_on", string]]`)

assumes

(`Optional[list[string]]`)

base

(`Optional[string]`)

confinement

(`Optional[string]`)

description

(string)

grade

(Optional[string])

icon

(Optional[string])

license

(Optional[string])

name

(string)

passthrough

(Optional[dict[string, string]])

parts

(Optional[dict[string, SnapPart]])

plugs

(Optional[dict[string, list[string]]])

slots

(Optional[dict[string, list[string]]])

summary

(string)

title

(Optional[string])

type

(Optional[string])

version

(string)

Methods

Snap.to_builder()

Converts this instance into a *SnapcraftBuilder*.

This method accepts no arguments and returns a *SnapcraftBuilder*. It is equivalent to calling `SnapcraftBuilder(self)`.

11.2.8 SnapcraftBuilder

The `SnapcraftBuilder` type coordinates the invocation of the `snapcraft` command.

Constructors

SnapcraftBuilder()

`SnapcraftBuilder()` constructs a new instance from a *Snap*.

It accepts the following arguments:

snap (`Snap`) The *Snap* defining the configuration to be used.

Methods

SnapcraftBuilder.add_invocation()

This method registers an invocation of `snapcraft` with the builder. When this instance is built, all registered invocations will be run sequentially.

The following arguments are accepted:

args (`List[String]`) Arguments to pass to `snapcraft` executable.

purge_build (`Optional[bool]`) Whether to purge the build directory before running this invocation.

> If not specified, the build directory is purged for the first registered invocation and not purged for all subsequent invocations.

SnapcraftBuilder.add_file_manifest()

This method registers the content of a *FileManifest* with the build environment for this builder.

When this instance is built, the content of the passed manifest will be materialized in a directory next to the `snapcraft.yaml` file this instance is building.

The following arguments are accepted:

manifest (`FileManifest`) A *FileManifest* defining files to install in the build environment.

SnapcraftBuilder.build()

This method invokes the builder and runs `snapcraft`.

The following arguments are accepted:

target (`String`) The name of the build target.

This method returns a `ResolvedTarget`. That target is not runnable.

11.2.9 WiXBundleBuilder

The `WiXBundleBuilder` type allows building simple *bundle* installers with the WiX Toolset.

`WiXBundleBuilder` instances allow you to create `.exe` installers that are composed of a chain of actions. At execution time, each action in the chain is evaluated. See the WiX Toolset documentation for more.

Constructors

WiXBundleBuilder()

`WiXBundleBuilder()` is called to construct new instances. It accepts the following arguments:

id_prefix (`string`) The string prefix to add to auto-generated IDs in the `.wxs` XML.

The value must be alphanumeric and - cannot be used.

The value should reflect the application whose installer is being defined.

name (`string`) The name of the application being installed.

version (`string`) The version of the application being installed.

This is a string like `X.Y.Z`, where each component is an integer.

manufacturer (`string`) The author of the application.

Methods

Sections below document methods available on `WiXBundleBuilder` instances.

WiXBundleBuilder.add_condition()

Defines a `<bal:Condition>` that must be satisfied to run this installer.

See the WiX Toolkit documentation for more.

This method accepts the following arguments:

condition (`string`) The condition expression that must be satisfied.

message (`string`) The message that will be displayed if the condition is not met.

WiXBundleBuilder.add_vc_redistributable()

This method registers the Visual C++ Redistributable to be installed.

This method accepts the following arguments:

platform (`string`) The architecture to install for. Valid values are `x86`, `x64`, and `arm64`.

The bundle can contain Visual C++ Redistributables for multiple runtime architectures. The bundle installer will only install the Redistributable when running on a machine of that architecture. This allows a single bundle installer to target multiple architectures.

WiXBundleBuilder.add_wix_msi_builder()

This method adds a *WiXMSIBuilder* to be installed by the produced installer.

This method accepts the following arguments:

builder (`WiXMSIBuilder`) The *WiXMSIBuilder* representing an MSI to install.

display_internal_ui (`Optional[bool]`) Whether to display the UI of the MSI. This is `False` by default.

install_condition (`Optional[string]`) An expression that must be true for this MSI to be installed.

This method effectively coerces the `WiXMSIBuilder` instance to an `<MsiPackage>` element and adds it to the `<Chain>` in the bundle XML. See the WiX Toolset documentation for more.

WiXBundleBuilder.build()

This method will build an exe using the WiX Toolset.

This method accepts the following arguments:

target (`string`) The name of the target being built.

11.2.10 WiXInstaller

The `WiXInstaller` type represents a Windows installer built with the WiX Toolset.

`WiXInstaller` instances allow you to collect `.wxs` files for processing and to turn these into an installer using the `light.exe` tool in the WiX Toolset.

Constructors

WiXInstaller()

`WiXInstaller()` is called to construct a new instance. It accepts the following arguments:

id (`string`) The name of the installer being built.

This value is used in `Id` attributes in WiX XML files and must conform to limitations imposed by WiX. Notably, this must be alphanumeric and - cannot be used.

This value is also used to derive GUIDs for the installer.

This value should reflect the name of the entity being installed and should be unique to prevent collisions with other installers.

filename (`string`) The name of the file that will be built.

WiX supports generating multiple installer file types depending on the content of the `.wxs` files. You will have to provide a filename that is appropriate for the installer type.

File extensions of `.msi` and `.exe` are common. If using `add_simple_installer()`, you will want to provide an `.msi` filename.

Methods

Sections below document methods available on `WiXInstaller` instances.

WiXInstaller.add_build_files()

This method registers additional files to make available to the build environment. Files will be materialized next to `.wxs` files that will be processed as part of building the installer.

Accepted arguments are:

manifest (`FileManifest`) The file manifest defining additional files to install.

WiXInstaller.add_build_file()

This method registers a single additional file to make available to the build environment.

Accepted arguments are:

build_path (`string`) The relative path to materialize inside the build environment

filesystem_path (`string`) The filesystem path of the file to copy into the build environment.

force_read (`bool`) Whether to read the content of this file into memory when this function is called.

Defaults to `False`.

WiXInstaller.add_install_file()

Add a file from the filesystem to be installed by the installer.

This methods accepts the following arguments:

install_path (`string`) The relative path to materialize inside the installation directory.

filesystem_path (string) The filesystem path of the file to install.

force_read (bool) Whether to read the content of this file into memory when this function is called.

Defaults to False.

WiXInstaller.add_install_files()

Add files defined in a *FileManifest* to be installed by the installer.

This method accepts the following arguments:

manifest (FileManifest) A *FileManifest* defining files to materialize in the installation directory. All these files will be installed by the installer.

WiXInstaller.add_msi_builder()

This method adds a *WiXMSIBuilder* instance to this instance, marking it for processing/building.

This method accepts the following arguments:

builder (WiXMSIBuilder) A *WiXMSIBuilder* representing a .wxs file to build.

WiXInstaller.add_simple_installer()

This method will populate the installer configuration with a pre-defined and simple/basic configuration suitable for simple applications. This method effectively derives a .wxs which will produce an MSI that materializes files in the Program Files directory.

Accepted arguments are:

product_name (string) The name of the installed product. This becomes the value of the <Product Name="..."> attribute in the generated .wxs file.

product_version (string) The version string of the installed product. This becomes the value of the <Product Version="..."> attribute in the generated .wxs file.

product_manufacturer (string) The author of the product. This becomes the value of the <Product Manufacturer="..."> attribute in the generated .wxs file.

program_files (FileManifest) Files to materialize in the Program Files/<product_name> directory upon install.

WiXInstaller.add_wxs_file()

Adds an existing .wxs file to be processed as part of building this installer.

Accepted arguments are:

path (string) The filesystem path to the .wxs file to add. The file will be copied into a temporary directory as part of building the installer and the destination filename will be the same as the file's name.

preprocessor_parameters (Optional[dict[string, string]]) Preprocessor parameters to define when invoking candle.exe for this .wxs file. These effectively constitute -p arguments to candle.exe.

WiXInstaller.set_variable()

Defines a variable to be passed to `light.exe` as `-d` arguments.

Accepted arguments are:

key (`string`) The name of the variable.

value (`Optional[string]`) The value of the variable. If `None` is used, the variable has no value and is simply defined.

11.2.11 WiXMSIBuilder

The `WiXMSIBuilder` type allows building simple MSI installers using the WiX Toolset.

`WiXMSIBuilder` instances allow you to create and build a `.wxs` file with common features. A goal of this type is to allow simple applications - without complex installer needs - to generate MSI installers without having to author your own `.wxs` files.

Constructors

WiXMSIBuilder()

`WiXMSIBuilder()` is called to construct new instances. It accepts the following arguments:

id_prefix (`string`) The string prefix to add to auto-generated IDs in the `.wxs` XML.

The value must be alphanumeric and `-` cannot be used.

The value should reflect the application whose installer is being defined.

product_name (`string`) The name of the application being installed.

product_version (`string`) The version of the application being installed.

This is a string like `X.Y.Z`, where each component is an integer.

product_manufacturer (`string`) The author of the application.

Attributes

Sections below document attributes available on instances. Attributes are write only.

banner_bmp_path

(`string`)

The path to a 493 x 58 pixel BMP file providing the banner to display in the installer.

dialog_bmp_path

(`string`)

The path to a 493 x 312 pixel BMP file providing an image to be displayed in the installer.

eula_rtf_path

(string)

The path to a RTF file containing the EULA that will be shown to users during installation.

help_url

(string)

A URL that will be presented to provide users with help.

license_path

(string)

Path to a file containing the license for the application being installed.

msi_filename

(string)

The filename to use for the built MSI.

If not set, the default is `<product_name>-<product_version>.msi`.

package_description

(string)

A description of the application being installed.

package_keywords

(string)

Keywords for the application being installed.

product_icon_path

(string)

Path to a file providing the icon for the installed application.

target_triple

(string)

The Rust target triple the MSI is being built for.

upgrade_code

(string)

A GUID defining the upgrade code for the application.

If not provided, a stable GUID derived from the application name will be derived automatically.

Methods

Sections below document methods available on `WiXMSIBuilder` instances.

WiXMSIBuilder.add_program_files_manifest()

This method registers the content of a *FileManifest* to be installed in the *Program Files* directory for this application.

This method accepts the following arguments:

manifest (`FileManifest`) A *FileManifest* containing files to register for installation.

WiXMSIBuilder.add_visual_cpp_redistributable()

This method will locate and add the Visual C++ Redistributable runtime DLL files (e.g. `vcruntime140.dll`) to the *Program Files* manifest in the builder, effectively materializing these files in the installed file layout.

This method accepts the following arguments:

redist_version (`string`) The version of the Visual C++ Redistributable to search for and add. `14` is the version used for Visual Studio 2015, 2017, and 2019.

platform (`string`) Identifies the Windows run-time architecture. Must be one of the values `x86`, `x64`, or `arm64`.

This method uses `vswhere.exe` to locate the `vcruntimeXXX.dll` files inside a Visual Studio installation. This should *just work* if a modern version of Visual Studio is installed. However, it may fail due to system variance.

WiXMSIBuilder.build()

This method will build an MSI using the WiX Toolset.

This method accepts the following arguments:

target (`string`) The name of the target being built.

11.3 Using the WiX Toolset to Produce Windows Installers

The WiX Toolset is an open source collection of tools used for building Windows installers (`.msi` files, `.exe`, etc). The WiX Toolset is incredibly powerful and enables building anything from simple to complex installers.

Tugger defines interfaces to the WiX Toolset via Rust APIs and exposes much of this functionality to Starlark.

11.3.1 Concepts

With the WiX Toolset, you define your installer through `.wxs` XML files. You use the `candle.exe` program to *compile* these files into `.wixobj` files. These *compiled* files are then *linked* together using `light.exe` to produce an installer (`.msi`, `.exe`, etc).

The goal of Tugger's Rust API is to expose the low-level control over WiX Toolset that the most demanding applications will need while also providing high-level and simpler interfaces for performing common tasks (such as producing a simple `.msi` installer that simply materializes files into the `Program Files` directory).

11.3.2 Tugger's WiX APIs

Tugger implements various interfaces for interacting with WiX. This section attempts to document them at a high level and talks about when to use which.

`WxsBuilder` The `WxsBuilder` Rust struct is used to build a single `.wxs` file. You provide the path of the `.wxs` and build settings and it knows how to invoke `candle.exe` for this file.

`WiXInstallerBuilder` The `WiXInstallerBuilder` Rust struct and *WiXInstaller* Starlark type are used to manage the end-to-end building and linking of `.wxs` files. This type knows how to register multiple `WxsBuilder` instances and build them as a collection. This type holds all the logic for invoking `candle.exe` and `light.exe`.

`WiXSimpleMSIBuilder` The `WiXSimpleMSIBuilder` Rust struct and *WiXMSIBuilder* Starlark type provide a high-level interface for generating an MSI based installer with common features. It enables you to generate a `.wxs` file by providing a few parameters, without having to know WiX XML.

A `WiXSimpleMSIBuilder` ultimately is converted to a `WiXInstallerBuilder`.

`WiXBundleInstallerBuilder` The `WiXBundleInstallerBuilder` Rust struct and *WiXBundleBuilder* Starlark type provide a high-level interface for generating an `.exe` based installed with common features.

A `WiXBundleInstallerBuilder` ultimately is converted to a `WiXInstallerBuilder`.

If your application only needs the limited functionality exposed by the high-level `WiXSimpleMSIBuilder` and `WiXBundleInstallerBuilder` interfaces, you are encouraged to use these for building your installer, as you won't need to concern yourself with the low-level WiX XML details.

If your application needs what you think is simple or common functionality not provided by the aforementioned high-level builders, consider filing a feature request to request the missing functionality.

Complex applications that have outgrown the limited capabilities of the high-level *builder* interfaces will need to use the lower level `WiXInstallerBuilder` / *WiXInstaller* interface. This interface allows you to provide your own `.wxs` files. This means you can still use Tugger for invoking WiX, even if all of your `.wxs` files are maintained outside of Tugger, enabling Tugger to grow with your needs. Note that it is possible to use one of the higher-level interfaces for automatically generating a `.wxs` file and then supplement this automatically-generated file with other `.wxs` files that you maintain.

Note: Ideally no WiX installer should be too complicated to be handled by Tugger. If Tugger's functionality is not sufficient, consider creating an issue to request a feature to close the feature gap.

11.3.3 How Tugger Invokes WiX

Tugger's Rust APIs collects which `.wxs` files to compile and their compilation settings. It also collects additional files needed to compile `.wxs` files.

When you *build* your installer, Tugger copies all the registered `.wxs` files plus other registered files into a common directory. It then invokes `candle.exe` on each `.wxs` file followed by `light.exe` to link them together. This is different from a traditional environment, where `.wxs` files are often processed in place: Tugger always makes copies to try to ensure results are reproducible and the full build environment is captured.

11.3.4 Automatic `<Fragment>` Generation for Files

Tugger supports automatically generating a `.wxs` file with `<Fragment>`'s describing a set of files. Given a set of input files, it will produce a deterministic `.wxs` file with `<DirectoryRef>` holding `<Component>` and `<File>` of every file therein as well as `<ComponentGroup>` for each distinct directory tree.

This functionality is similar to what WiX Toolset's `heat.exe` tool can do. However, Tugger uses a deterministic mechanism to derive GUIDs and IDs for each item. This enables the produced elements to be referenced in other `.wxs` files more easily. And the generated file doesn't need to be saved or manually updated, as it does with the use of `heat.exe`.

You simply give Tugger a manifest of files to index and the prefix for `Id` attributes in XML, and it will emit a deterministic `.wxs` file!

11.4 Project History

11.4.1 Version History

0.4.0

Not yet released.

0.3.0

Released March 4, 2021.

New Features

- The `FileManifest` Starlark type now exposes an `add_path()` method.
- The Starlark dialect now exposes `SnapApp`, `Snappart`, and `Snap` types representing Snapcraft configuration files.
- The Starlark dialect now has a `SnapcraftBuilder` type that serves as an interface to invoking `snapcraft`.
- The Starlark dialect now exposes `WiXBundleBuilder`, `WiXInstaller`, and `WiXMSIBuilder` types for defining Windows installers using the WiX Toolset.

0.2.0

Version 0.2 was released November 8, 2020.

Version 0.2 marked the beginning of a complete rewrite of Tugger. The canonical source code repository was moved to the PyOxidizer repository.

Not all features from version 0.1 were ported to version 0.2.

0.1.0

Version 0.1 was released on August 25, 2019.

Version 0.1 was mostly a proof of concept to demonstrate the viability of Starlark configuration files. But Tugger was usable in this release.

CHAPTER 12

Frequently Asked Questions

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

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

12.2 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.

12.3 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.

12.4 Why is Python 3.8 Required?

Python 3.8 contains a new C API for controlling how embedded Python interpreters are started. This makes the run-time code that native binaries execute much, much simpler.

PyOxidizer versions up to 0.7 supported Python 3.7. But a decision was made to require Python 3.8 because the run-time code to manage the Python interpreter was vastly simpler and less prone to bugs. Given that Python 3.8 is mostly backwards compatible with Python 3.7, this wasn't perceived as a significant annoyance.

12.5 `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.

12.6 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.

12.7 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.

12.8 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.

During Python interpreter initialization, a custom Rust-implemented Python importer is registered and takes over all imports. Requests for modules are serviced from the parsed data structure defining known modules.

Follow the *Documentation* link for the pyembed crate for an overview of how the in-memory import machinery works.

12.9 Can Applications Import Python Modules from the Filesystem?

Yes!

While PyOxidizer supports importing Python resources from in-memory, it also supports filesystem-based import like traditional Python applications.

This can be achieved by adding Python resources to a non *in-memory* resource location (see *Managing How Resources are Added*) or by enabling Python's standard filesystem-based importer by enabling `filesystem_importer=True` (see *PythonInterpreterConfig*).

12.10 `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.

12.11 `vcruntime140.dll was not found` Error on Windows

Binaries built with PyOxidizer often have a dependency on the Visual C++ Redistributable Runtime, or `vcruntime140.dll`. If this file is not present on your system or in a path where the built binary can find it, you'll get an error about this missing file when attempting to run/load the binary.

PyOxidizer has some support for managing this file for you. See *Managing the Visual C++ Redistributable Requirement* for more.

If PyOxidizer is not materializing this file next your built binary, either you've disabled this functionality via your configuration file (see *PythonExecutable.windows_runtime_dlls_mode*) or PyOxidizer could not find the Visual Studio component providing this file.

The quick fix for this is to install the Visual C++ Redistributable runtime globally on your system. Simply go to https://support.microsoft.com/en-us/topic/the-latest-supported-visual-c-downloads-2647da03-1eea-4433-9aff-95f26a218cc0 and download and install the appropriate platform installer for `Visual Studio 2015, 2017 and 2019`.

If you want PyOxidizer to materialize the DLL(s) next to your built binary, you'll need to install Visual Studio with the `Microsoft.VisualCPP.Redist.14.Latest` component (you will typically get this component if installing support for building C/C++ applications).

12.12 `ld: unsupported tapi file type '!tapi-tbd' in YAML file` on macOS When Building

If you see this error when building on macOS, it means that the linker (likely Clang) being used is not able to read the `.tbd` files from a more modern Apple SDK.

PyOxidizer requires using an Apple SDK no older than the one used to build the Python distributions being embedded (see *Build Machine Requirements*). So the only recourse to this problem is to use a more modern linker.

On Apple platforms, it is common to use the clang/linker from an Xcode or Xcode Commandline Tools install. So the problem can usually be fixed by upgrading Xcode or the Xcode Commandline Tools.

CHAPTER 13

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!

13.1 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.

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

13.2 Major Missing Features

13.2.1 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.

13.2.2 Native Extension Modules

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 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*.

13.2.3 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.

13.2.4 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:

- Support for `__file__`.
- 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.

13.2.5 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.

13.2.6 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.

13.3 Lesser Missing Features

13.3.1 Python Version Support

Python 3.8 and 3.9 are currently supported. Older versions of PyOxidizer (through version 0.7) supported Python 3.7. See *Why is Python 3.8 Required?* for why we require these Python versions.

13.3.2 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.

13.3.3 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).

13.3.4 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.

13.3.5 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.

13.3.6 Configuration Files

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

13.4 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.

13.4.1 Lazy Module Loading

When a Python module is imported, 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 importable modules at build time, `PyOxidizer` could return a *lazy* module object after performing a simple Rust `HashMap` lookup. This would be extremely fast.

13.4.2 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.

13.4.3 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).

13.4.4 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.

13.4.5 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.

13.4.6 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.

13.4.7 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.

CHAPTER 14

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. Even better, submit a pull request!

14.1 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 to a temporary directory then import modules from the filesystem.

`PyOxidizer` often skips this step and loads modules directly from memory using zero-copy. This makes `PyOxidizer` executables significantly faster to start when this feature is employed.

When `PyOxidizer` is running in single-file mode, it needs to build all binary dependencies from source to facilitate static linking. Although this behavior is optional and `PyOxidizer` can also work with pre-built binary Python packages.

A current difference between the tools is that `PyInstaller` generally has better support for binary dependencies. `PyInstaller` knows how to find runtime dependencies and allows a lot of not-easy-to-build packages like PyQT to work out of the box. With `PyOxidizer`, you could need to add sufficient complexity to its configuration files to get things to work.

14.2 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.

`py2exe` and `PyOxidizer` both employ a clever trick on Windows that allows loading DLLs from memory. This enables DLLs to be embedded in an executable so you can ship a single `.exe` and not have to worry about bundling DLLs as separate files. (`PyOxidizer` is using the same in-memory DLL loading library as `py2exe`.)

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`.

14.3 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`.

14.4 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.

14.5 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 environments where Python is installed and Python shebangs work. It isn't acceptable for environments where you can't guarantee an appropriate Python executable is installed/available.

By distributing its 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.

14.6 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.

14.7 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.

14.8 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 executables 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.

14.9 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 or 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`.

14.10 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.

14.11 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.

14.12 Bazel

Bazel has Python rules for building Python binaries and libraries. From a high level, it works similarly to how PyOxidizer's Starlark config files allow you to perform much of the same actions.

The executables produced by `py_binary` are significantly different from what PyOxidizer does, however.

An executable produced by `py_binary` is a glorified self-executing zip file. At run time, it extracts Python resources to a temporary directory and then runs a Python interpreter against them. The approach is similar in nature to what Shiv and PEX do.

PyOxidizer, by contrast, produces a specialized binary containing the Python interpreter and allows you to embed Python resources inside that binary, enabling Python modules to be imported without the overhead of writing a temporary directory and extracting a zip file.

CHAPTER 15

Contributing to PyOxidizer

This page documents how to contribute to PyOxidizer.

15.1 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.

15.2 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.

The `Cargo.toml` in the root of the repository defines a Cargo workspace containing many crates. If you attempt to `cargo build` or `cargo test`, you will likely get errors, as different crates have different, conflicting build requirements. The `oxidized-importer` crate is particularly troublesome.

Try building/testing everything with `cargo build --workspace --exclude oxidized-importer` or `cargo test --workspace --exclude oxidized-importer`. Or just target the crate you want by adding the `-p` argument. e.g. `cargo build -p pyembed` or `cargo test -p pyoxidizer`.

15.3 Financial Contributions

If you would like to thank the PyOxidizer maintainer via a financial contribution, you can do so via GitHub Sponsors 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 16

Project History

Work on PyOxidizer started in November 2018 by Gregory Szorc.

16.1 Blog Posts

- Announcing the 0.9 Release of PyOxidizer (2020-10-18)
- Announcing the 0.8 Release of PyOxidizer (2020-10-12)
- Using Rust to Power Python Importing with oxidized_importer (2020-05-10)
- PyOxidizer 0.7 (2020-04-09)
- 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)

16.2 Version History

16.2.1 0.13.2

Released April 15, 2021.

Bug Fixes

- Fixes a build failure on Windows.

16.2.2 0.13.1

Released April 15, 2021.

Bug Fixes

- The 0.13.0 release contained improper crate paths in `Cargo.toml` files due to a bug in our automated release mechanism. This release should fix those issues.

16.2.3 0.13.0

Released April 15, 2021.

Bug Fixes

- `WiXSimpleMsiBuilder` now properly writes XML when a license file is provided.
- `WixBundleInstallerBuilder` now handles the *already installed* exit code from the VC++ Redistributable installer as a success condition. Previously, installs would abort.
- `WixBundleInstallerBuilder` no longer errors on a missing build directory when attempting to download the Visual C++ Redistributable runtime files.

New Features

- Per-platform Windows MSI and multi-platform Windows exe installers for PyOxidizer are now available. The installers are built with PyOxidizer, using its built-in support for producing Windows installers.

Other Relevant Changes

- Default CPython distributions upgraded from 3.9.3 to 3.9.4.
- Default Python distributions upgraded setuptools from 54.2.0 to 56.0.0.

16.2.4 0.12.0

Released April 14, 2021.

Danger: The 0.12.0 release uses CPython 3.9.3, which inadvertently shipped an ABI incompatible change, causing some extension modules to not work or crash. Please avoid this release if you use pre-built Python extension modules.

Backwards Compatibility Notes

- The minimum Rust version has been changed from 1.45 to 1.46 to facilitate use of *const fn*.

- On Apple platforms, PyOxidizer now validates that the Apple SDK being used is compatible with the Python distribution being used and will abort the build if not. Previously, PyOxidizer would blindly use whatever SDK was the default and this could lead to cryptic error messages when building (likely undefined symbol errors when linking). The current default Python distributions impose a requirement of the macosx10.15+ SDK for Python 3.8 and macosx11.0+ for Python 3.9. See issue #373 for a comprehensive discussion of this topic.
- On Apple platforms, binaries built with PyOxidizer now automatically target the OS version that the Python distribution was built to target. Previously, binaries would likely target the OS version of the building machine unless explicit action was taken. The practical effect of this change is binaries targeting x86_64 should now work on macOS 10.9 without any end-user action required.
- Documentation URLs for PyOxidizer now all consistently begin with `pyoxidizer_`. Many old documentation URLs no longer work.

Bug Fixes

- The autogenerated `pyoxidizer.bzl` correctly references the `no-copyleft` extension module filter instead of the old `no-gpl` value.
- Linux binaries using the `libedit` variant of the `readline` Python extension (occurs when using the `no-copyleft` extension module filter) no longer encounter an undefined symbol error when linking. (#376)
- The *ctypes* extension was previously compiled incorrectly, leading to run-time errors on various platforms. These issues should be fixed.

New Features

- On Apple platforms, PyOxidizer now automatically locates, validates, and uses an appropriate SDK given the settings of the Python distribution being used. PyOxidizer will reject building with an SDK older than the one used to produce the Python distribution. PyOxidizer will automatically use the newest installed SDK compatible with the target configuration. The SDK and targeting information is printed during builds. See *Build Machine Requirements* for details on how to override default behavior.
- `OxidizedFinder` now implements `path_hook()` and a path hook is automatically registered on `sys.path_hooks` during interpreter initialization when an `OxidizedFinder` is being used. Feature contributed by William Schwartz in #343.

Other Relevant Changes

- The `snmalloc` allocator now uses the C API directly and avoids going through an allocation tracking layer, improving the performance of this allocator. Improvement contributed by Ryan Clanton.
- Python distributions updated to latest versions. Changes include: macOS Python 3.8 is now built against the 10.15 SDK instead of 11.1; musl libc upgraded to 1.2.2; setuptools upgraded to 54.2.0; LibreSSL upgraded to 3.2.5; OpenSSL upgraded to 1.1.1k; SQLite upgraded to 3.35.4.

16.2.5 0.11.0

Released March 4, 2021.

Backwards Compatibility Notes

- The default Python distribution is now CPython 3.9 instead of 3.8. To use 3.8, pass the `python_version="3.8"` argument to *default_python_distribution()* in your configuration file. We don't anticipate dropping support for 3.8 any time soon. However, this may be necessary in order to more easily support new Python features.
- The Python 3.8 distributions no longer support Windows 7 and require Windows 8, Windows 2012, or newer. The Python 3.9 distributions already required these Windows versions.
- The minimum Rust version has been changed from 1.41 to 1.45 to facilitate the use of procedural macros.
- The `pyembed::MainPythonInterpreter::run_as_main()` method has been renamed to `py_runmain()` to reflect that it always calls `Py_RunMain()`.
- The `py-module-names` file is no longer written as part of the files comprising an embedded Python interpreter.
- `OxidizedFinder.__init__()` no longer accepts `resources_data` and `resources_file` argument to specify the resources to load. Instead, call one of the new `index_*` methods on constructed instances.
- `OxidizedFinder.__init__()` no longer automatically indexes builtin extension modules and frozen modules. Instead, you must now call one of the `index_*` methods to index these resources.
- The `pyembed::OxidizedPythonInterpreterConfig.packed_resources` field is now a `Vec<pyembed::PackedResourcesSource>` instead of `Vec<&[u8]>`. The new enum allows specifying files as alternative resources sources.
- The `no-gpl` value of `PythonPackagingPolicy.extension_module_filter` has been changed to `no-copyleft` and it operates on the SPDX license annotations instead of a list we maintained.
- `show_alloc_count` has been removed from types representing Python interpreter configuration because support for this feature was removed in Python 3.9.
- `pyembed::MainPythonInterpreter.acquire_gil()`'s signature has changed, now returning a `Python` value directly without wrapping it in a `Result`.
- `pyembed::OxidizedPythonInterpreterConfig` had its memory allocator fields refactored to support new features and to help prevent bad configs (like defining multiple custom memory allocators).
- The Starlark `PythonInterpreterConfig.raw_allocator` field has been renamed to `allocator_backend`. The `system` value has been renamed to `default`.
- The `pyembed` crate now canonicalizes the current executable's path and uses this canonicalized path when resolving values with `$ORIGIN` in them. Previously, the path passed into the program was used without resolving symlinks, etc. If that path were a symlink or hardlink, unexpected results could ensue.
- `OxidizedFinder.find_distributions()` now returns an iterator of `OxidizedDistribution` instead of a `list`. Code in the standard library of older versions of CPython expected an iterator to be returned and the new behavior is more compatible. This change enables `importlib.metadata.metadata()` to work with `OxidizedFinder`.

Bug Fixes

- Escaping of string and path values when emitting Rust code for the embedded Python interpreter configuration should now be more robust. Previously, special characters (like `\`) were not escaped properly. (#321)
- The `load()` Starlark function should now work. (#328)
- `pyembed::OxidizedPythonInterpreterConfig.argv` is now always used when set, even if `self.interpreter_config.argv` is also set.

- `OxidizedFinder` now normalizes trailing `.__init__` in module names to be equivalent to the parent package to partially emulate CPython's behavior. See *Support for __init__ in Module Names* for more. (#317)
- The lifetime of `pyembed::MainPythonInterpreter.acquire_gil()`'s return value has been adjusted so the Rust compiler will refuse to compile code that could crash due to attempting to use a finalized interpreter. (#345)
- `pyembed::MainPythonInterpreter.py_runmain()`'s signature has changed, now consuming ownership of the receiver. Subsequent borrows of `self` now fail to compile rather than causing runtime errors.
- The optional `rust` memory allocator is now thread-safe. Previously, it wasn't and releasing of the GIL could lead to memory corruption and crashes.
- `OxidizedResourceCollector.oxidize()` should now properly clean up the temporary directory it uses during execution. Before, premature Python interpreter termination (such as during failing tests) could cause the temporary directory to not be removed. Closes #346. Fix contributed by William Schwartz in #347.
- `OxidizedFinder.find_distributions()` now properly handles the default/empty `Context` instance (specifically instances where `.name = None`). Previously, `name = None` would filter as if `.name = "None"`. This means that all distributions should now be returned with the default/empty `Context` instance.
- `OxidizedFinder.find_distributions()` now properly filters when the passed `Context`'s `name` attribute is set to a string. Previously, the `name` and `path` attributes had their order swapped in a function call, leading to incorrect filtering.
- The Windows `standalone_static` distributions should now work again. They had been broken for a few releases and likely never worked with Python 3.9. Test coverage of this build configuration has been added to help prevent future regressions. (#360)

New Features

- Support added for `aarch64-apple-darwin` (Apple M1 machines). Only Python 3.9 is supported on this architecture. Because we do not have CI coverage for this architecture (due to GitHub Actions not yet having M1 machines), support is considered beta quality at this time.
- The `FileManifest` Starlark type now exposes an `add_path()` to add a single file to the manifest.
- The `PythonExecutable` Starlark type now exposes a `to_file_manifest()` to convert the instance to a `FileManifest`.
- The `PythonExecutable` Starlark type now exposes a `to_wix_msi_builder()` method to obtain a `WiXMSIBuilder`, which can be used to generate an MSI installer for the application.
- The `PythonExecutable` Starlark type now exposes a `to_wix_bundle_builder()` method to obtain a `WiXBundleBuilder`, which can be used to generate an `.exe` installer for the application.
- The `pyembed` crate and `OxidizedFinder` importer now support indexing multiple resources sources. You can have multiple in-memory data blobs, multiple file-based resources, or a mix of all of the above.
- The `OxidizedFinder` Python type now exposed various `index_*` methods to facilitate loading/indexing of resource data in arbitrary byte buffers or files. You can call these methods multiple times to chain multiple resources blobs together.
- The `PythonExecutable` Starlark type now exposes a `packed_resources_load_mode` attribute allowing control over where *packed resources data* is written and how it is loaded at run-time. This attribute facilitates disabling the embedding of packed resources data completely (enabling you to produce an executable that behaves very similarly to `python`) and allows writing and loading resources data to a standalone file installed next to the binary (enabling multiple binaries to share the same resources file). See *Managing Packed Resources Data* for more on this feature.

- PyOxidizer now scans for licenses of Python packages processed during building and prints a report about what it finds when writing build artifacts. This feature is best effort and relies on packages properly advertising their license metadata.
- Support for configuring Python's memory allocators has been expanded. The Starlark *PythonInterpreterConfig.allocator_debug* field has been added and allows enabling Python memory allocator debug hooks. The Starlark *PythonInterpreterConfig.allocator_mem*, *PythonInterpreterConfig.allocator_obj*, and *PythonInterpreterConfig.allocator_pymalloc_arena* fields have been added to control whether to install a custom allocator for the *mem* and *obj* domains as well as `pymalloc`'s arena allocator.
- The *mimalloc* and *snmalloc* memory allocators can now be used as Python's memory allocators. See documentation for *PythonInterpreterConfig.allocator_backend*. Code contributed by Ryan Clanton in #358.
- The *mimalloc* and *snmalloc* memory allocators will now automatically be used as Rust's global allocator when configured to be used by Python.
- The `@classmethod OxidizedDistribution.find_name()` and `OxidizedDistribution.discover()` are now implemented, filling in a feature gap in `importlib.metadata` functionality.
- There is a new *PythonExecutable.windows_runtime_dlls_mode* attribute to control how required Windows runtime DLL files should be materialized during application building. By default, if a built binary requires the Visual C++ Redistributable Runtime (e.g. `vcruntime140.dll`), PyOxidizer will attempt to locate and copy those files next to the built binary. See *Managing the Visual C++ Redistributable Requirement* for more.
- Documentation around portability of binaries produced with PyOxidizer has been reorganized and overhauled. See *Portability of Binaries Built with PyOxidizer* for the new documentation.

Other Relevant Changes

- Python distributions upgraded to CPython 3.8.8 and 3.9.2 (from 3.8.6 and 3.9.0). See https://github.com/indygreg/python-build-standalone/releases/tag/20210103 and https://github.com/indygreg/python-build-standalone/releases/tag/20210227 for a full list of changes in these distributions.
- CI has been moved from Azure Pipelines to GitHub Actions.
- Low level code in the `pyembed` crate for loading and indexing resources has been significantly refactored. This code has historically been a bit brittle, as it needs to do *unsafe* things. We think the new code is much more robust. But there's a chance that crashes could occur.
- When using the `no-copyleft` (formerly `no-gpl`) extension module filter, some system library dependencies are now allowed, enabling various extension modules to be present in this mode.
- The `pyembed` and `oxidized-importer` crates had their SPDX license expression changed from `Python-2.0 AND MPL-2.0` to `Python-2.0 OR MPL-2.0`. The author misunderstood what `AND` did and after realizing his mistake, corrected it to `OR` so the crates can one license or the other.
- When using dynamically linked Python distributions on Windows, the `python3.dll` file is automatically installed if it is present. (#336)
- `libclang_rt.osx.a` is now linked into Python binaries on macOS. This was necessary to avoid undefined symbols errors from symbols which Python 3.9.1+ relies on.
- The `oxidized_importer` Python module now exports the `OxidizedDistribution` symbol, which is the custom `importlib.metadata` *distribution* type used by `OxidizedFinder`.
- When building with Windows `standalone_static` distributions, `pyoxidizer` now sets `RUSTFLAGS=-C target-feature=+crt-static -C link-args=/FORCE:MULTIPLE` to force static CRT linkage and ignore duplicate symbol errors. Previously, the Python distribution would be using static CRT linkage and the Rust application would use dynamic/DLL CRT linkage. Furthermore, many `standalone_static` distributions have build configurations that lead to duplicate symbols and this

would lead to a linker error. Suppressing the duplicate symbol error is not ideal, but it restores building with `standalone_static` until a more appropriate workaround can be devised.

16.2.6 0.10.3

Released November 10, 2020.

Bug Fixes

- The `run_as_main()` function on embedded Python interpreters now always calls `Py_RunMain()`. This fixes a regression in previous 0.10 releases that prevented a REPL from running when no explicit `run_*` attribute was set on the Python interpreter configuration.

16.2.7 0.10.2

Released November 10, 2020.

Bug Fixes

- Fixes a version mismatch between the `pyoxidizer` and `pyembed` crates that could cause builds to fail.

16.2.8 0.10.1

Released November 9, 2020.

Danger: The 0.10.1 release has a serious bug where the version of the `pyembed` crate needed to build binaries may not be correct, preventing the build from working. Please use a newer release.

Bug Fixes

16.2.9 0.10.0

Released November 8, 2020.

Danger: The 0.10.0 release has a serious Starlark bug preventing PyOxidizer from working correctly in many scenarios. Please use a newer release.

Backwards Compatibility Notes

- A lot of unused Rust functions for running Python code have been removed from the `pyembed` crate. The deleted code has not been used since the `PyConfig` data structure was adopted for running code during interpreter initialization. The deleted code was reimplementing functionality in CPython and much of it was of questionable quality.
- The built-in Python distributions have been updated to use version `6` of the standalone distribution format. PyOxidizer only recognizes version `6` distributions.

- The `pyembed::OxidizedPythonInterpreterConfig` Rust struct now contains a `tcl_library` field to control the value of the *TCL_LIBRARY* environment variable.
- The `pyembed::OxidizedPythonInterpreterConfig` Rust struct no longer has a `run_mode` field.
- The `PythoninterpreterConfig` Starlark type no longer has a `run_mode` attribute. To define what code to run at interpreter startup, populate a `run_*` attribute or leave all `None` with `.parse_argv = True` (the default for `profile = "python"`) to start a REPL.
- Minimum Rust version changed from 1.40 to 1.41 to facilitate using a new crate which requires 1.41.
- The default Cargo features of the `pyembed` crate now use the default Python interpreter detection and linking configuration as determined by the `cpython` crate. This enables the `cargo build` or `cargo test` to *just work* without having to explicitly specify features.
- The `python-distributions-extract` command now receives the path to an existing distribution archive via the `--archive-path` argument instead of an unnamed argument.

Bug Fixes

- Fixed a broken documentation example for `glob()`. (#300)
- Fixed a bug where generated Rust code for *Option<PathBuf>* interpreter configuration fields was not being generated correctly.
- Fixed serialization of string config options to Rust code that was preventing the following attributes of the `PythonInterpreterConfig` Starlark type from working: `filesystem_encoding`, `filesystem_errors`, `python_path_env`, `run_command`, `run_module`, `stdio_encoding`, `stdio_errors`, `warn_options`, and `x_options`. (#309)

New Features

- The `PythonExecutable` Starlark type now exposes a `windows_subsystem` attribute to control the value of Rust's `#![windows_subsystem = "..."]` attribute. Setting this to `windows` prevents Windows executables from opening a console window when run. (#216)
- The `PythonExecutable` Starlark type now exposes a `tcl_files_path` attribute to define a directory to install tcl/tk support files into. Setting this attribute enables the use of the `tkinter` Python module with compatible Python distributions. (#25)
- The `python-distribution-extract` CLI command now accepts a `--download-default` flag to download the default distribution for the current platform.

Other Relevant Changes

- The Starlark types with special *build* or *run* behavior are now explicitly documented.
- The list of glibc and GCC versions used by popular Linux distributions has been updated.
- The built-in Linux and macOS Python distributions are now compiled with LLVM/Clang 11 (as opposed to 10).
- The built-in Python distributions now use pip 20.2.4 and setuptools 50.3.2.
- The Starlark primitives for defining build system targets have been extracted into a new `starlark-dialect-build-targets` crate.
- The code for resolving how to reference PyOxidizer's Git repository has been rewritten. The resolution is now performed at build time in the pyoxidizer crate's `build.rs`. There now exist environment variables that can be specified at crate build time that influence how PyOxidizer constructs these references.

16.2.10 0.9.0

Released October 18, 2020.

Backwards Compatibility Notes

- The `pyembed::OxidizedPythonInterpreterConfig` Rust struct now contains an `argv` field that can be used to control the population of `sys.argv`.
- The `pyembed::OxidizedPythonInterpreterConfig` Rust struct now contains a `set_missing_path_configuration` field that can be used to control the automatic run-time population of missing *path configuration* fields.
- The `configure_locale` interpreter configuration setting is enabled by default. (#294)
- The `pyembed::OxidizedPythonInterpreterConfig` Rust struct now contains an `exe` field holding the path of the currently running executable.
- At run-time, the `program_name` and `home` fields of the embedded Python interpreter's path configuration are now always set to the currently running executable and its directory, respectively, unless explicit values have been provided.
- The packed resource data version has changed from 2 to 3 in order to support storing arbitrary file data. Support for reading and writing version 2 has been removed. Packed resources blobs will need to be regenerated in order to be compatible with new versions of PyOxidizer.
- The `pyembed::OxidizedPythonInterpreterConfig` Rust struct had its `packed_resources` field changed from `Option<&'a [u8]>` to `Vec<&'a [u8]>` so multiple resource inputs can be specified.
- The `PythonDistribution` Starlark type no longer has `extension_modules()`, `package_resources()` and `source_modules()` methods. Use `PythonDistribution.python_resources()` instead.

New Features

- A `print(*args)` function is now exposed to Starlark. This function is documented as a Starlark built-in but isn't provided by the Rust Starlark implementation by default. So we've implemented it ourselves. (#292)
- The new `pyoxidizer find-resources` command can be used to invoke PyOxidizer's code for scanning files for resources. This command can be used to debug and triage bugs related to PyOxidizer's custom code for finding and handling resources.
- Executables built on Windows now embed an application manifest that enables long paths support. (#197)
- The Starlark `PythonPackagingPolicy` type now exposes an `allow_files` attribute controlling whether files can be added as resources.
- The Starlark `PythonPackagingPolicy` type now exposes `file_scanner_classify_files` and `file_scanner_emit_files` attributes controlling whether file scanning attempts to classify files and whether generic file instances are emitted, respectively.
- The Starlark `PythonPackagingPolicy` type now exposes `include_classified_resources` and `include_file_resources` attributes to control whether certain classes of resources have their `add_include` attribute set by default.
- The Starlark `PythonPackagingPolicy` type now has a `set_resources_handling_mode()` method to quickly apply a mode for resource handling.
- The Starlark `PythonDistribution` type now has a `python_resources()` method for obtaining all Python resources associated with the distribution.

- Starlark `File` instances can now be added to resource collections via `PythonExecutable.add_python_resource()` and `PythonExecutable.add_python_resources()`.

Bug Fixes

- Fix some documentation references to outdated Starlark configuration syntax (#291).
- Emit only the `PythonExtensionModule` built with our patched distutils instead of emitting 2 `PythonExtensionModule` for the same named module. This should result in compiled Python extension modules being usable as built-in extensions instead of being recognized as only shared libraries.
- Fix typo preventing the Starlark method `PythonExecutable.read_virtualenv()` from being defined. (#297)
- The default value of the Starlark `PythonInterpreterConfig.configure_locale` field is `True` instead of `None` (effectively `False` since the default `.profile` value is `isolated`). This results in Python's encodings being more reasonable by default, which helps ensure non-ASCII arguments are interpreted properly. (#294)
- Properly serialize `module_search_paths` to Rust code. Before, attempting to set `PythonInterpreterConfig.module_search_paths` in Starlark would result in malformed Rust code being generated. (#298)

Other Relevant Changes

- The `pyembed` Rust crate now calls `PyConfig_SetBytesArgv` or `PyConfig_SetArgv()` to initialize argv instead of `PySys_SetObject()`. The encoding of string values should also behave more similarly to what `python` does.
- The `pyembed` tests exercising Python interpreters now run in separate processes. Before, Rust would instantiate multiple interpreters in the same process. However, CPython uses global variables and APIs (like `setlocale()`) that also make use of globals and process reuse resulted in tests not having pristine execution environments. All tests now run in isolated processes and should be much more resilient.
- When PyOxidizer invokes a subprocess and logs its output, stderr is now redirected to stdout and logged as a unified stream. Previously, stdout was logged and stderr went to the parent process stderr.
- There now exists *documentation* on how to create an executable that behaves like `python`.
- The documentation on binary portability has been overhauled to go in much greater detail.
- The list of standard library test packages is now obtained from the Python distribution metadata instead of a hardcoded list in PyOxidizer's source code.

16.2.11 0.8.0

Released October 12, 2020.

Backwards Compatibility Notes

- The default Python distributions have been upgraded to CPython 3.8.6 (from 3.7.7) and support for Python 3.7 has been removed.
- On Windows, the `default_python_distribution()` Starlark function now defaults to returning a `standalone_dynamic` distribution variant, meaning that it picks a distribution that can load standalone `.pyd` Python extension modules by default.

- The *standalone* Python distributions are now validated to be at least version 5 of the distribution format. If you are using the default Python distributions, this change should not affect you.
- Support for packaging the official Windows embeddable Python distributions has been removed. This support was experimental. The official Windows embeddable distributions are missing critical support files that make them difficult to integrate with PyOxidizer.
- The `pyembed` crate now defines a new `OxidizedPythonInterpreterConfig` type to configure Python interpreters. The legacy `PythonConfig` type has been removed.
- Various `run_*` functions on `pyembed::MainPythonInterpreter` have been moved to standalone functions in the `pyembed` crate. The `run_as_main()` function (which is called by the default Rust program that is generated) will always call `Py_RunMain()` and finalize the interpreter. See the extensive crate docs for move.
- Python resources data in the `pyembed` crate is no longer annotated with the `'static` lifetime. Instances of `PythonConfig` and `OxidizedPythonInterpreterConfig` must now be annotated with a lifetime for the resources data they hold such that Rust lifetimes can be enforced.
- The type of the custom Python importer has been renamed from `PyOxidizerFinder` to `OxidizedFinder`.
- The name of the module providing our custom importer has been renamed from `_pyoxidizer_importer` to `oxidized_importer`.
- Minimum Rust version changed from 1.36 to 1.40 to allow for upgrading various dependencies to modern versions.
- Windows static extension building is possibly broken due to changes to `distutils`. However, since we changed the default configuration to not use this build mode, we've deemed this potential regression acceptable for the 0.8 release. If it exists, it will hopefully be fixed in the 0.9 release.
- The `pip_install()`, `read_package_root()`, `read_virtualenv()` and `setup_py_install()` methods of the `PythonDistribution` Starlark type have been moved to the `PythonExecutable` type. Existing Starlark config files will need to change references accordingly (often by replacing `dist.` with `exe.`).
- The `PythonDistribution.extension_modules()` Starlark function no longer accepts arguments `filter` and `preferred_variants`. The function now returns every extension in the distribution. The reasons for this change were to make code simpler and the justification for removing it was rather weak. Please file an issue if this feature loss affects you.
- The `PythonInterpreterConfig` Starlark type now interally has most of its fields defined to `None` by default instead of their default values.
- The following Starlark methods have been renamed: `PythonExecutable.add_module_source()` -> `PythonExecutable.add_python_module_source()`; `PythonExecutable.add_module_bytecode()` -> `PythonExecutable.add_python_module_bytecode()`; `PythonExecutable.add_package_resource()` -> `PythonExecutable.add_python_package_resource()`; `PythonExecutable.add_package_distribution_resource()` -> `PythonExecutable.add_python_package_distribution_resource()`; `PythonExecutable.add_extension_module()` -> `PythonExecutable.add_python_extension_module()`.
- The location-specific Starlark methods for adding Python resources have been removed. The functionality can be duplicated by modifying the `add_location` and `add_location_fallback` attributes on Python resource types. The following methods were removed: `PythonExecutable.add_in_memory_module_source()`; `PythonExecutable.add_filesystem_relative_module_source()`,

PythonExecutable.add_in_memory_module_bytecode(); PythonExecutable.
add_filesystem_relative_module_bytecode(); PythonExecutable.
add_in_memory_package_resource();PythonExecutable.add_filesystem_relative_package_resou
PythonExecutable.add_in_memory_package_distribution_resource()
PythonExecutable.add_filesystem_relative_package_distribution_resource();
PythonExecutable.add_in_memory_extension_module(); PythonExecutable.
add_filesystem_relative_extension_module(); PythonExecutable.
add_in_memory_python_resource();PythonExecutable.add_filesystem_relative_python_resourc
PythonExecutable.add_in_memory_python_resources(); PythonExecutable.
add_filesystem_relative_python_resources().

- The Starlark `PythonDistribution.to_python_executable()` method no longer accepts the arguments `extension_module_filter`, `preferred_extension_module_variants`, `include_sources`, `include_resources`, and `include_test`. All of this functionality has been replaced by the optional `packaging_policy`, which accepts a `PythonPackagingPolicy` instance. The new type represents all settings influencing executable building and control over resources added to the executable.
- The Starlark type `PythonBytecodeModule` has been removed. Previously, this type was internally a request to convert Python module source into bytecode. The introduction of `PythonPackagingPolicy` and underlying abilities to derive bytecode from a Python source module instance when adding that resource type rendered this Starlark type redundant. There may still be the need for a Starlark type to represent actual Python module bytecode (not derived from source code at build/packaging time). However, this functionality did not exist before so the loss of this type is not a loss in functionality.
- The Starlark methods `PythonExecutable.add_python_resource()` and `PythonExecutable.add_python_resources()` no longer accept the arguments `add_source_module`, `add_bytecode_module`, and `optimize_level`. Instead, set various `add_*` attributes on resource instances being passed into the methods to influence what happens when they are added.
- The Starlark methods `PythonExecutable.add_python_module_source()`, `PythonExecutable.add_python_module_bytecode()`, `PythonExecutable.add_python_package_resource()`,`PythonExecutable.add_python_package_distribution_resource` and `PythonExecutable.add_python_extension_module()` have been removed. The remaining `PythonExecutable.add_python_resource()` and `PythonExecutable.add_python_resources()` methods are capable of handling all resource types and should be used. Previous functionality available via argument passing on these methods can be accomplished by setting `add_*` attributes on individual Python resource objects.
- The Starlark type `PythonSourceModule` has been renamed to `PythonModuleSource`.
- Serialized Python resources no longer rely on the `flavor` field to influence how they are loaded at run-time. Instead, the new `is_*` fields expressing individual type affinity are used. The `flavor` attributes from the `OxidizedResource` Python type has been removed since it does nothing.
- The packed resources data format version has been changed from 1 to 2. The parser has dropped support for reading version 1 files. Packed resources blobs will need to be written and read by the same version of the Rust crate to be compatible.
- The autogenerated Rust file containing the Python interpreter configuration now emits a `pyembed::OxidizedPythonInterpreterConfig` instance instead of `pyembed::PythonConfig`. The new type is more powerful and what is actually used to initialize an embedded Python interpreter.
- The concept of a *resources policy* in Starlark has now largely been replaced by attributes denoting valid locations for resources.
- **`oxidized_importer.OxidizedResourceCollector.__init__()` now** accepts an `allowed_locations` argument instead of `policy`.

- The `PythonInterpreterConfig()` constructor has been removed. Instances of this Starlark type are now created via `PythonDistribution.make_python_interpreter_config()`. In addition, instances are mutated by setting attributes rather than passing perhaps dozens of arguments to a constructor function.
- The default build configuration for Windows no longer forces extension modules to be loaded from memory and materializes some extension modules as standalone files. This was done because some some extension modules weren't working when loaded from memory and the configuration caused lots of problems in the wild. The new default should be much more user friendly. To use the old settings, construct a custom `PythonPackagingPolicy` and set `allow_in_memory_shared_library_loading = True` and `resources_location_fallback = None`.

New Features

- Python distributions upgraded to CPython 3.8.6.
- CPython 3.9 distributions are now supported by passing `python_version="3.9"` to the `default_python_distribution()` Starlark function. CPython 3.8 is the default distribution version.
- Embedded Python interpreters are now managed via the new apis defined by PEP-587. This gives us much more control over the configuration of interpreters.
- A `FileManifest` Starlark instance will now have its default `pyoxidizer run` executable set to the last added Python executable. Previously, it would only have a run target if there was a single executable file in the `FileManifest`. If there were multiple executables or executable files (such as Python extension modules) a run target would not be available and `pyoxidizer run` would do nothing.
- Default Python distributions upgraded to version 5 of the standalone distribution format. This new format advertises much more metadata about the distribution, enabling PyOxidizer to take fewer guesses about how the distribution works and will help enable more features over time.
- The `pyembed` crate now exposes a new `OxidizedPythonInterpreterConfig` type (and associated types) allowing configuration of every field supported by Python's interpreter configuration API.
- Resources data loaded by the `pyembed` crate can now have a non-`'static` lifetime. This means that resources data can be more dynamically obtained (e.g. by reading a file). PyOxidizer does not yet support such mechanisms, however.
- `OxidizedFinder` instances can now be *constructed from Python code*. This means that a Python application can instantiate and install its own oxidized module importer.
- The resources indexed by `OxidizedFinder` instances are now representable to Python code as `OxidizedResource` instances. These types can be created, queried, and mutated by Python code. See *OxidizedResource* for the API.
- `OxidizedFinder` instances can now have custom `OxidizedResource` instances registered against them. This means Python code can collect its own Python modules and register them with the importer. See *add_resource(self, resource: OxidizedResource)* for more.
- `OxidizedFinder` instances can now serialize indexed resources out to a `bytes`. The serialized data can be loaded into a separate `OxidizedFinder` instance, perhaps in a different process. This facility enables the creation and reuse of packed resources data structures without having to use `pyoxidizer` to collect Python resources data.
- The types returned by `OxidizedFinder.find_distributions()` now implement `entry_points`, allowing *entry points* to be discovered.
- The types returned by `OxidizedFinder.find_distributions()` now implement `requires`, allowing package requirements to be discovered.

- `OxidizedFinder` is now able to load Python modules when only source code is provided. Previously, it required that bytecode be available.
- `OxidizedFinder` now implements `iter_modules()`. This enables `pkgutil.iter_modules()` to return modules serviced by `OxidizedFinder`.
- The `PythonModuleSource` Starlark type now exposes module source code via the `source` attribute.
- The `PythonExecutable` Starlark type now has a `make_python_module_source()` method to allow construction of `PythonModuleSource` instances.
- The `PythonModuleSource` Starlark type now has attributes `add_include`, `add_location`, `add_location_fallback`, `add_source`, `add_bytecode_optimization_level_zero`, `add_bytecode_optimization_level_one`, and `add_bytecode_optimization_level_two` to influence what happens when instances are added to to binaries.
- The Starlark methods for adding Python resources now accept an optional `location` argument for controlling the load location of the resource. This functionality replaces the prior functionality provided by location-specific APIs such as `PythonExecutable.add_in_memory_python_resource()`. The following methods gained this argument: `PythonExecutable.add_python_module_source()`; `PythonExecutable.add_python_module_bytecode()`; `PythonExecutable.add_python_package_resource()`; `PythonExecutable.add_python_package_distribution_resource()`; `PythonExecutable.add_python_extension_module()`; `PythonExecutable.add_python_resource()`; `PythonExecutable.add_python_resources()`.
- Starlark now has a `PythonPackagingPolicy` type to represent the collection of settings influencing how Python resources are packaged into binaries.
- The `PythonDistribution` Starlark type has gained a `make_packaging_policy()` method for obtaining the default `PythonPackagingPolicy` for that distribution.
- The `PythonPackagingPolicy.register_resource_callback()` method can be used to register a Starlark function that will be called whenever resources are created. The callback allows a single function to inspect and manipulate resources as they are created.
- Starlark types representing Python resources now expose an `is_stdlib` attribute denoting whether they came from the Python distribution.
- The new `PythonExecutable.pip_download()` method will run `pip download` to obtain Python wheels for the requested package(s). Those wheels will then be parsed for Python resources, which can be added to the executable.
- The Starlark function `default_python_distribution()` now accepts a `python_version` argument to control the *X.Y* version of Python to use.
- The `PythonPackagingPolicy` Starlark type now exposes a flag to control whether shared libraries can be loaded from memory.
- The `PythonDistribution` Starlark type now has a `make_python_interpreter_config()` method to obtain instances of `PythonInterpreterConfig` that are appropriate for that distribution.
- `PythonInterpreterConfig` Starlark types now expose attributes to query and mutate state. Nearly every setting exposed by Python's initialization API can be set.

Bug Fixes

- Fixed potential process crash due to illegal memory access when loading Python bytecode modules from the filesystem.

- Detection of Python bytecode files based on registered suffixes and cache tags is now more robust. Before, it was possible for modules to get picked up having the cache tag (e.g. `cpython-38`) in the module name.
- In the custom Python importer, `read_text()` of distributions returned from `find_distributions()` now returns `None` on unknown file instead of raising `IOError`. This matches the behavior of `importlib.metadata`.
- The `pyembed` Rust project build script now reruns when the source Starlark file changes.
- Some Python resource types were improperly installed in the wrong relative directory. The buggy behavior has been fixed.
- Python extension modules and their shared library dependencies loaded from the filesystem should no longer have the library file suffix stripped when materialized on the filesystem.
- On Windows, the `sqlite` module can now be imported. Before, the system for serializing resources thought that `sqlite` was a shared library and not a Python module.
- The build script of the pyoxidizer crate now uses the `git2` crate to try to resolve the Git commit instead of relying on the `git` command. This should result in fewer cases where the commit was being identified as `unknown`.
- `$ORIGIN` is properly expanded in `sys.path`. (This was a regression during the development of version 0.8 and is not a regression from the 0.7 release.)

Other Relevant Changes

- The registration of the custom Python importer during interpreter initialization no longer relies on running custom frozen bytecode for the `importlib._bootstrap_external` Python module. This simplifies packaging and interpreter configuration a bit.
- Packaging documentation now gives more examples on how to use available Starlark packaging methods.
- The modified `distutils` files used when building statically linked extensions have been upgraded to those based on Python 3.8.3.
- The default `pyoxidizer.bzl` now has comments for the `packaging_policy` argument to `PythonDistribution.to_python_executable()`.
- The default `pyoxidizer.bzl` now uses `add_python_resources()` instead of `add_in_memory_python_resources()`.
- The Rust Starlark crate was upgraded from version 0.2 to 0.3. There were numerous changes as part of this upgrade. While we think behavior should be mostly backwards compatible, there may be some slight changes in behavior. Please file issues if any odd behavior or regressions are observed.
- The configuration documentation was reorganized. The unified document for the complete API document (which was the largest single document) has been split into multiple documents.
- The serialized data structure for representing Python resources metadata and its data now allows resources to identify as multiple types. For example, a single resource can contain both Python module source/bytecode and a shared library.
- `pyoxidizer --version` now prints verbose information about where PyOxidizer was installed, what Git commit was used, and how the `pyembed` crate will be referenced. This should make it easier to help debug installation issues.
- The autogenerated/default Starlark configuration file now uses the `install` target as the default build/run target. This allows extra files required by generated binaries to be available and for built binaries to be usable.

16.2.12 0.7.0

Released April 9, 2020.

Backwards Compatibility Notes

- Packages imported from memory using PyOxidizer now set `__path__` with a value formed by joining the current executable's path with the package name. This mimics the behavior of `zipimport`.
- Resolved Python resource names have changed behavior. See the note in the bug fixes section below.
- The `PythonDistribution.to_python_executable()` Starlark method has added a `packaging_policy` named argument as its 2nd argument / 1st named argument. If you were affected by this, you should add argument names to all arguments passed to this method.
- The default Rust project for built executables now builds executables such that dynamic symbols are exported from the executable. This change is necessary in order to support executables loading Python extension modules, which are shared libraries which need access to Python symbols defined in executables.
- The `PythonResourceData` Starlark type has been renamed to `PythonPackageResource`.
- The `PythonDistribution.resources_data()` Starlark method has been renamed to `PythonDistribution.package_resources()`.
- The `PythonExecutable.to_embedded_data()` Starlark method has been renamed to `PythonExecutable.to_embedded_resources()`.
- The `PythonEmbeddedData` Starlark type has been renamed to `PythonEmbeddedResources`.
- The format of Python resource data embedded in binaries has been completely rewritten. The separate modules and resource data structures have been merged into a single data structure. Embedded resources data can now express more primitives such as package distribution metadata and different bytecode optimization levels.
- The *pyembed* crate now has a *dev* dependency on the *pyoxidizer* crate in order to run tests.

Bug Fixes

- PyOxidizer's importer now always sets `__path__` on imported packages in accordance with Python's stated behavior (#51).
- The mechanism for resolving Python resource files from the filesystem has been rewritten. Before, it was possible for files like `package/resources/foo.txt` to be normalized to a (package, resource_name) tuple of *(package, resources.foo.txt)*, which was weird and not compatible with Python's resource loading mechanism. Resources in sub-directories should no longer encounter munging of directory separators to `.`. In the above example, the resource path will now be expressed as `(package, resources/foo.txt)`.
- Certain packaging actions are only performed once during building instead of twice. The user-visible impact of this change is that some duplicate log messages no longer appear.
- Added a missing *)* for *add_python_resources()* in auto-generated *pyoxidizer.bzl* files.

New Features

- Python resource scanning now recognizes `*.dist-info` and `*.egg-info` directories as package distribution metadata. Files within these directories are exposed to Starlark as *PythonPackageDistributionResource* instances. These resources can be added to the embedded resources payload and made available for loading from memory or the filesystem, just like any other resource. The custom Python importer implements

`get_distributions()` and returns objects that expose package distribution files. However, functionality of the returned *distribution* objects is not yet complete. See *importlib.metadata Compatibility* for details.

- The custom Python importer now implements `get_data(path)`, allowing loading of resources from filesystem paths (#139).
- The `PythonDistribution.to_python_executable()` Starlark method now accepts a `packaging_policy` argument to control a policy and default behavior for resources on the produced executable. Using this argument, it is possible to control how resources should be materialized. For example, you can specify that resources should be loaded from memory if supported and from the filesystem if not. The argument can also be used to materialize the Python standard library on the filesystem, like how Python distributions typically work.
- Python resources can now be installed next to built binaries using the new Starlark functions `PythonExecutable.add_filesystem_relative_module_source()`, `PythonExecutable.add_filesystem_relative_module_bytecode()`, `PythonExecutable.add_filesystem_relative_package_resource()`, `PythonExecutable.add_filesystem_relative_extension_module()`, `PythonExecutable.add_filesystem_relative_python_resource()`, `PythonExecutable.add_filesystem_relative_package_distribution_resource()`, and `PythonExecutable.add_filesystem_relative_python_resources()`. Unlike adding Python resources to `FileManifest` instances, Python resources added this way have their metadata serialized into the built executable. This allows the special Python module importer present in built binaries to service the `import` request without going through Python's default filesystem-based importer. Because metadata for the file-based Python resources is *frozen* into the application, Python has to do far less work at run-time to load resources, making operations faster. Resources loaded from the filesystem in this manner have attributes like `__file__`, `__cached__`, and `__path__` set, emulating behavior of the default Python importer. The custom import now also implements the `importlib.abc.ExecutionLoader` interface.
- Windows binaries can now import extension modules defined as shared libraries (e.g. `.pyd` files) from memory. PyOxidizer will detect `.pyd` files during packaging and embed them into the binary as resources. When the module is imported, the extension module/shared library is loaded from memory and initialized. This feature enables PyOxidizer to package pre-built extension modules (e.g. from Windows binary wheels published on PyPI) while still maintaining the property of a (mostly) self-contained executable.
- Multiple bytecode optimization levels can now be embedded in binaries. Previously, it was only possible to embed bytecode for a given module at a single optimization level.
- The `default_python_distribution()` Starlark function now accepts values `standalone_static` and `standalone_dynamic` to specify a *standalone* distribution that is either statically or dynamically linked.
- Support for parsing version 4 of the `PYTHON.json` distribution descriptor present in standalone Python distribution archives.
- Default Python distributions upgraded to CPython 3.7.7.

Other Relevant Changes

- The directory for downloaded Python distributions in the build directory now uses a truncated SHA-256 hash instead of the full hash to help avoid path length limit issues (#224).
- The documentation for the `pyembed` crate has been moved out of the Sphinx documentation and into the Rust crate itself. Rendered docs can be seen by following the *Documentation* link at https://crates.io/crates/pyembed or by running `cargo doc` from a source checkout.

16.2.13 0.6.0

Released February 12, 2020.

Backwards Compatibility Notes

- The `default_python_distribution()` Starlark function now accepts a `flavor` argument denoting the distribution flavor.
- The `pyembed` crate no longer includes the auto-generated default configuration file. Instead, it is consumed by the application that instantiates a Python interpreter.
- Rust projects for the main executable now utilize and require a Cargo build script so metadata can be passed from `pyembed` to the project that is consuming it.
- The `pyembed` crate is no longer added to created Rust projects. Instead, the generated `Cargo.toml` will reference a version of the `pyembed` crate identical to the `PyOxidizer` version currently running. Or if `pyoxidizer` is running from a Git checkout of the canonical `PyOxidizer` Git repository, a local filesystem path will be used.
- The fields of `EmbeddedPythonConfig` and `pyembed::PythonConfig` have been renamed and reordered to align with Python 3.8's config API naming. This was done for the Starlark type in version 0.5. We have made similar changes to 0.6 so naming is consistent across the various types.

Bug Fixes

- Module names without a `.` are now properly recognized when scanning the filesystem for Python resources and a package allow list is used (#223). Previously, if filtering scanned resources through an explicit list of allowed packages, the top-level module/package without a dot in its full name would not be passed through the filter.

New Features

- The `PythonDistribution()` Starlark function now accepts a `flavor` argument to denote the distribution type. This allows construction of alternate distribution types.
- The `default_python_distribution()` Starlark function now accepts a `flavor` argument which can be set to `windows_embeddable` to return a distribution based on the zip file distributions published by the official CPython project.
- The `pyembed` crate and generated Rust projects now have various `build-mode-*` feature flags to control how build artifacts are built. See *Rust Projects* for more.
- The `pyembed` crate can now be built standalone, without being bound to a specific `PyOxidizer` configuration.
- The `register_target()` Starlark function now accepts an optional `default_build_script` argument to define the default target when evaluating in *Rust build script* mode.
- The `pyembed` crate now builds against published `cpython` and `python3-sys` crates instead of a a specific Git commit.
- Embedded Python interpreters can now be configured to run a file specified by a filename. See the `run_file` argument of *PythonInterpreterConfig*.

Other Relevant Changes

- Rust internals have been overhauled to use traits to represent various types, namely Python distributions. The goal is to allow different Python distribution flavors to implement different logic for building binaries.
- The `pyembed` crate's `build.rs` has been tweaked so it can support calling out to `pyoxidizer`. It also no longer has a build dependency on `pyoxidizer`.

16.2.14 0.5.1

Released January 26, 2020.

Bug Fixes

- Fixed bad Starlark example for building `black` in docs.
- Remove resources attached to packages that don't exist. (This was a regression in 0.5.)
- Warn on failure to annotate a package. (This was a regression in 0.5.)
- Building embedded Python resources now emits warnings when `__file__` is seen. (This was a regression in 0.5.)
- Missing parent packages are now automatically added when constructing embedded resources. (This was a regression in 0.5.)

16.2.15 0.5.0

Released January 26, 2020.

General Notes

This release of PyOxidizer is significant rewrite of the previous version. The impetus for the rewrite is to transition from TOML to Starlark configuration files. The new configuration file format should allow vastly greater flexibility for building applications and will unlock a world of new possibilities.

The transition to Starlark configuration files represented a shift from static configuration to something more dynamic. This required refactoring a ton of code.

As part of refactoring code, we took the opportunity to shore up lots of the code base. PyOxidizer was the project author's first real Rust project and a lot of bad practices (such as use of *.unwrap()*/panics) were prevalent. The code mostly now has proper error handling. Another new addition to the code is unit tests. While coverage still isn't great, we now have tests performing meaningful packaging activities. So regressions should hopefully be less common going forward.

Because of the scale of the rewritten code in this release, it is expected that there are tons of bugs of regressions. This will likely be a transitional release with a more robust release to follow.

Backwards Compatibility Notes

- Support for building distributions/installers has been temporarily dropped.
- Support for installing license files has been temporarily dropped.

- Python interpreter configuration setting names have been changed to reflect names from Python 3.8's interpreter initialization API.
- `.egg-info` directories are now ignored when scanning for Python resources on the filesystem (matching the behavior for `.dist-info` directories).
- The `pyoxidizer init` sub-command has been renamed to `init-rust-project`.
- The `pyoxidizer app-path` sub-command has been removed.
- Support for building distributions has been removed.
- The minimum Rust version to build has been increased from 1.31 to 1.36. This is mainly due to requirements from the `starlark` crate. We could potentially reduce the minimum version requirements again with minimal changes to 3rd party crates.
- PyOxidizer configuration files are now Starlark instead of TOML files. The default file name is `pyoxidizer.bzl` instead of `pyoxidizer.toml`. All existing configuration files will need to be ported to the new format.

Bug Fixes

- The `repl` run mode now properly exits with a non-zero exit code if an error occurs.
- Compiled C extensions now properly honor the `ext_package` argument passed to `setup()`, resulting in extensions which properly have the package name in their extension name (#26).

New Features

- A *glob()*' function has been added to config files to allow referencing existing files on the filesystem.
- The in-memory `MetaPathFinder` now implements `find_module()`.
- A `pyoxidizer init-config-file` command has been implemented to create just a `pyoxidizer.bzl` configuration file.
- A `pyoxidizer python-distribution-info` command has been implemented to print information about a Python distribution archive.
- The `EmbeddedPythonConfig()` config function now accepts a `legacy_windows_stdio` argument to control the value of `Py_LegacyWindowsStdioFlag` (#190).
- The `EmbeddedPythonConfig()` config function now accepts a `legacy_windows_fs_encoding` argument to control the value of `Py_LegacyWindowsFSEncodingFlag`.
- The `EmbeddedPythonConfig()` config function now accepts an `isolated` argument to control the value of `Py_IsolatedFlag`.
- The `EmbeddedPythonConfig()` config function now accepts a `use_hash_seed` argument to control the value of `Py_HashRandomizationFlag`.
- The `EmbeddedPythonConfig()` config function now accepts an `inspect` argument to control the value of `Py_InspectFlag`.
- The `EmbeddedPythonConfig()` config function now accepts an `interactive` argument to control the value of `Py_InteractiveFlag`.
- The `EmbeddedPythonConfig()` config function now accepts a `quiet` argument to control the value of `Py_QuietFlag`.
- The `EmbeddedPythonConfig()` config function now accepts a `verbose` argument to control the value of `Py_VerboseFlag`.

- The `EmbeddedPythonConfig()` config function now accepts a `parser_debug` argument to control the value of `Py_DebugFlag`.
- The `EmbeddedPythonConfig()` config function now accepts a `bytes_warning` argument to control the value of `Py_BytesWarningFlag`.
- The `Stdlib()` packaging rule now now accepts an optional `excludes` list of modules to ignore. This is useful for removing unnecessary Python packages such as `distutils`, `pip`, and `ensurepip`.
- The `PipRequirementsFile()` and `PipInstallSimple()` packaging rules now accept an optional `extra_env` dict of extra environment variables to set when invoking `pip install`.
- The `PipRequirementsFile()` packaging rule now accepts an optional `extra_args` list of extra command line arguments to pass to `pip install`.

Other Relevant Changes

- PyOxidizer no longer requires a forked version of the `rust-cpython` project (the `python3-sys` and `cpython` crates. All changes required by PyOxidizer are now present in the official project.

16.2.16 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 final binary rather than being ignored.

16.2.17 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.

16.2.18 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.

16.2.19 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)

16.2.20 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.

16.2.21 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)

16.2.22 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 17

Technical Notes

17.1 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")``
```

17.2 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`.

17.3 `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()`

17.4 Modules Imported by `site.py`

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

17.5 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.

17.6 Desired Changes from Python to Aid PyOxidizer

As part of implementing PyOxidizer, we've encountered numerous shortcomings in Python that have made implementation more difficult. This section attempts to capture those along with our desired outcomes.

17.6.1 General Lack of Clear Specifications

PyOxidizer has had to implement a lot of low-level functionality, notably around interpreter initialization and module/resource importing. We have also had to reinvent aspects of packaging so it can be performed in Rust.

Various Python functionality is not defined in specifications. Rather, it is defined by PEPs plus implementations in code. And when there are PEPs, often there isn't a single PEP outlining the clear current state of the world: many PEPs are stated like *builds on top of PEP XYZ*. Often the only canonical source of how something works is the implementation in code. And when there are questions for clarification, it isn't clear whether code or a PEP is wrong because oftentimes there isn't a single PEP that is the canonical source of truth.

It would be highly preferred for Python to publish clear specifications for how various mechanisms work. A PEP would be a diff to a specification (possibly creating a new specification) and a discussion around it. That way there would be a clear specification that can be consulted as the source of truth for how things should behave.

17.6.2 `__file__` Ambiguity

It isn't clear whether `__file__` is actually required and what all is derived from existence of `__file__`. It also isn't clear what `__file__` should be set to if it wouldn't be a concrete filesystem path. Can `__file__` be virtual? Can it refer to a binary/archive containing the module?

Semantics of `__file__` need more clarification.

17.6.3 `importlib.metadata` Documentation Deficiencies

See https://bugs.python.org/issue38594.

17.6.4 `importlib` Resources Directory Ambiguity

See https://bugs.python.org/issue36128, https://gitlab.com/python-devs/importlib_resources/issues/58, and https://gitlab.com/python-devs/importlib_resources/-/issues/90.

17.6.5 Standardizing a Python Distribution Format

PyOxidizer consumes Python distributions and repackages them. e.g. it takes an archive containing a Python executable, standard library, support libraries, etc and transforms them into new binaries or distributable artifacts.

There is no standard for representing a Python distribution. This is something that PyOxidizer has had to invent itself via the `python-build-standalone` project and its `PYTHON.json` files.

Should Python have a standardized way of describing Python distribution archives and should CPython distribute said distributions, it would make PyOxidizer largely agnostic of the distributor flavor being consumed and allow PyOxidizer (and other Python packaging tools) to more easily target other distribution flavors. e.g. you could swap out CPython for PyPy and tooling largely wouldn't care.

17.6.6 Ability to Install Meta Path Importers Before `Py_Initialize()`

`Py_Initialize()` will import some standard library modules during its execution. It does so using the default meta path importers available to the distribution. This means that standard library modules must come from the filesystem (`PathImporter`), frozen modules, built-in extension modules, or zip files (via `PathImporter`).

This restriction prevents importing the entirety of the standard library from the binary containing Python, in effect preventing the use of self-contained executables. PyOxidizer works around this by patching the `importlib._bootstrap` and `importlib._bootstrap_external` source code, compiling that to bytecode, and making said bytecode available as a frozen module. The patched code (which runs as part of `Py_Initialize()`) installs a `sys.meta_path` importer which imports modules from memory. This solution is extremely hacky, but is necessary to achieve single file executables with all imports serviced from memory.

In order for this to work, PyOxidizer needs a copy of these `importlib` modules so it can patch them and compile them to bytecode. This is problematic in some cases because e.g. the Windows embeddable Python distributions ship only the bytecode of these modules in a `pythonXY.zip` file. So PyOxidizer needs to find the source code from another location when consuming these distributions.

But patching the `importlib` bootstrap modules is hacky itself. It would be better if PyOxidizer didn't need to do this at all. This could be achieved by splitting up the interpreter initialization APIs to give embedding applications the opportunity to muck with `sys.meta_path` before any `import` is performed. It could also be achieved by introducing an initialization config option to somehow inject code at the right point during startup to register the `sys.meta_path` importer. This could be done by importing a named module (presumably serviced by the frozen or built-in importer) and having that module run code to modify `sys.meta_path` as a side-effect of module evaluation at import time. A variation would be to define a callable in said module to call after the module is importer. Whatever the solution, there needs to be a way to somehow inject a `sys.meta_path` importer before any `import` not serviced by the frozen or built-in importers is performed.

17.6.7 Lacking Support for Statically Linked Builds

Python really wants you to be using shared libraries for `libpython` and extension modules seem to strongly insist on this.

On Windows, there is no official Visual Studio project configuration for static builds. Actually achieving one requires a lot of hacks to the build system (see `python-build-standalone` project).

There is ~0 support for building statically linked extension modules in packaging tools, from the build step itself all the way up to distribution. (PyOxidizer's approach is to hack `distutils` to record and save the object files that were compiled and then `PyOxidizer` manually links these object files into the final binary.)

To achieve a statically linked executable containing `libpython` and extension modules, you effectively need to build everything from source. And if you want to distribute that executable, you often need to build with special toolchains to ensure binary portability.

There is tons of room for Python to better support static linking. A possible good place to start would be for packaging tools to support building extension modules which don't rely on a dynamic `libpython`. If artifacts containing the raw object files designed for static linking were made available on PyPI, PyOxidizer could download pre-built binaries and link them directly into an executable or custom `libpython`. This would avoid having to recompile said extension modules at repackaging time. The compatibility guarantees would likely look a lot like existing binary wheels.

On a related front, it would be nice if musl libc based binary wheels were standardized. There are some concerns about the performance and compatibility of musl libc when it comes to Python. But musl libc is a valid deploy target nonetheless and it would be nice if Python officially supported it. (FWIW the performance concerns seem to stem from memory allocator performance and PyOxidizer supports using jemalloc as the allocator, bypassing this problem.)

17.6.8 Windows Embeddable Distributions Missing Functionality

The Windows embeddable zip file distributions of CPython are missing certain functionality.

The distributions do not contain source code for Python modules in the standard library. This means PyOxidizer can't easily bundle sources from these distributions.

The `ensurepip` module is not present in the distribution. So there is no way to install `pip` using the distribution itself.

The `venv` module is also not present in the distribution. So there's no way to create virtualenvs using the distribution itself.

The Python C development headers are not part of the distribution, so even if you install packaging tools, you can't build C extensions.

17.6.9 Extension Module / Shared Library Filename Ambiguity

On some platforms, Python extension modules and shared libraries have the same filename extension. e.g. on Linux, both are named `foo.so`.

PyOxidizer's packaging functionality needs to classify files as specific resource types (source modules, bytecode modules, resource files, extension modules, shared libraries, etc). Because certain file patterns (like `.so`) are ambiguous, PyOxidizer cannot perform this classification trivially.

It would be much preferred if there were unique file extensions that distinguished Python extension modules from regular shared libraries.

On Windows, this is already the case with the `.pyd` extension. However, POSIX architectures aren't so fortunate.

17.6.10 Ambiguous File Classification

This is somewhat related to the previous section but is more generic.

Python's default path-based importer dynamically looks for presence of various files on the filesystem and loads the first type variant (extension module, bytecode, source, etc) discovered.

PyOxidizer's importer indexes resources during packaging and its import-time resource resolution is static: the type of resource is baked into the definition of the resource.

These approaches are somewhat at odds with each other. The path-based importer is dynamic in nature: it defers answering questions until a specific resource is requested. PyOxidizer's importer is static / pre-compiled: it must classify a resource based on its filename/path so it can bake that knowledge into an immutable data structure. It does not have knowledge of what names will be requested at run-time.

Bridging this divide has revealed various ambiguities and corner cases in the filenames of Python resources.

The Python extension module or shared library ambiguity is described above.

There is also an ambiguity with extra files that aren't part of a known Python package. If you attempt to classify every file in a `sys.path` directory, it is tempting to classify a file as a Python module (`.py`, `.pyc`, or extension module), package resource (`importlib.resources`), or package metadata (e.g. `.dist-info` files accessed via `importlib.metadata`). However, there exists the possibility that a file is not obviously classified as one of these.

For example, a file `foo/libfoo.so` without the presence of a `foo/__init__.py` file is ambiguous. We could say this is an extension module (`foo.libfoo`) due to the extension module shared library ambiguity. We could also consider this a package resource `foo:libfoo.so` or `"":foo/libfoo.so`. Although the latter case of using an empty string for the package name doesn't make much sense. And we arguably shouldn't consider it a resource of `foo` because no obvious `foo` Python package exists!

This is relevant in the real world because various Python packages rely on installing arbitrary files in `sys.path` directories. For example, `numpy` installs files like `numpy.libs/libz-eb09ad1d.so.1.2.3`, where the `numpy.libs` directory only contains file extensions `*.so[.*]`. Note that this example is particularly confusing because the directory names in `sys.path` directories are typically split on `.` and correspond to Python [sub-]packages.

Because there is no unambiguous way to classify all files in a `sys.path` directory and because Python packaging tools allow the presence of files not contained within a known Python package (identified by the presence of an `__init__` file/module), this externalizes the requirement to introduce an *other* classification of files. And because a specific file can't easily be classified as a specific type, this effectively prevents the use of *resource* loading techniques not involving explicit filesystem I/O without significant smarts. I.e. because PyOxidizer cannot easily unambiguously identify file X as a specific type, it is forced to materialize that file at a similar location on the run-time system. However, if runtimes like PyOxidizer were able to identify the type of a file by its file extension and/or presence of other files, it would know exactly how to load/treat the file at run-time without having to resort to heuristics.

This ambiguity effectively means that PyOxidizer needs to:

- Determine if a file is a shared library or not (because shared libraries are treated specially and we can't unambiguously identify a shared library from its file extension).
- Examine symbols within shared libraries to see if a Python extension module is present (via presence of `PyInit_*` symbols).
- Preserve *extra* files not present in a Python package. (In the case of numpy, there are no *obvious* links to the shared libraries in the `numpy.libs` directory: this relative path is encoded within the extension module shared library via e.g. `DT_NEEDED`.)

The most robust mitigation to this ambiguity is for all files associated with an installable Python package/distribution to be annotated with their type and for Python package installers to refuse to process files that aren't identified. This could be achieved by having a `.dist-info/` file annotating the *role* of each file.

17.6.11 Push Harder for Wheels

Wheels are superior for Python packaging distribution because they are more *static* and follow a finite set of rules for how they should be installed. In theory, one could write code to install a wheel in any programming language. Non-wheel distributions, however, are a different matter entirely. A `.tar.gz` source distribution often relies on running a `setup.py` file, which requires a Python interpreter.

In the ideal world, PyOxidizer doesn't care about how a package is built: just the files that comprise the installed package. So wheels are a more desirable distribution format. In fact, PyOxidizer has Rust code for extracting wheels and repackaging their contents: no Python necessary. This means PyOxidizer can do things like download wheels targeting non-native architectures and it *just works*.

As good as wheels are, they are universal in Python land. There are tons of packages that don't have wheel distributions and continue to offer the older `.tar.gz` distribution format.

We would like to see a concerted effort to push harder for the presence of wheels. For example, PyPI could encourage/nag package maintainers to upload wheels.

Index