
PyOxidizer

Release 0.8.0-pre

May 10, 2020

Contents

1	Overview	3
1.1	Benefits of PyOxidizer	3
1.2	Components	4
1.3	How It Works	4
2	Getting Started	7
2.1	Installing	7
2.2	High-Level Project Lifecycle	8
2.3	Your First PyOxidizer Project	9
2.4	The <code>pyoxidizer.bzl</code> Configuration File	9
2.5	Customizing Python and Packaging Behavior	10
3	The <code>pyoxidizer</code> Command Line Tool	13
3.1	Creating New Projects with <code>init-config-file</code>	13
3.2	Creating New Rust Projects with <code>init-rust-project</code>	13
3.3	Adding PyOxidizer to an Existing Project with <code>add</code>	14
3.4	Building PyObject Projects with <code>build</code>	14
3.5	Running the Result of Building with <code>run</code>	14
3.6	Analyzing Produced Binaries with <code>analyze</code>	15
3.7	Inspecting Python Distributions	15
4	Configuration Files	17
4.1	Finding Configuration Files	17
4.2	File Processing Semantics	17
4.3	Targets	18
4.4	Common Operations	18
5	Packaging User Guide	21
5.1	Creating a PyOxidizer Project	21
5.2	Managing Resources and Their Locations	22
5.3	Packaging an Application from a PyPI Package	25
5.4	Packaging Additional Files	27
5.5	Trimming Unused Resources	29
5.6	Performance of Built Binaries	30
5.7	Packaging Pitfalls	32
5.8	Masquerading As Other Packaging Tools	34
5.9	Binary Compatibility	34

5.10	Static Linking	35
5.11	Licensing Considerations	37
5.12	Terminfo Database	38
6	oxidized_importer Python Extension	41
6.1	Getting Started	41
6.2	Python Meta Path Finders	42
6.3	OxidizedFinder Python Type	43
6.4	OxidizedFinder Behavior and Compliance	46
6.5	oxidized_importer Python Resource Types	48
6.6	Resource Scanning APIs	51
6.7	Loading Resource Files	52
6.8	Freezing Applications with oxidized_importer	57
6.9	Common Issues	60
6.10	Security Implications of Loading Resources	60
7	PyOxidizer for Rust Developers	63
7.1	Rust Projects	63
7.2	Crate Configuration	64
7.3	Controlling Python From Rust Code	65
7.4	Adding Extension Modules At Run-Time	68
8	Configuration File API Reference	71
8.1	Global Symbols	71
8.2	Types	72
8.3	Constants	72
8.4	Functions for Manipulating Global State	73
8.5	Functions for Managing Targets	73
8.6	Python Distributions	74
8.7	Python Resources	78
8.8	Python Resources Policy	79
8.9	Python Interpreter Configuration	79
8.10	Python Binaries	83
8.11	Interacting With the Filesystem	87
9	Frequently Asked Questions	89
9.1	Where Can I Report Bugs / Send Feedback / Request Features?	89
9.2	Why Build Another Python Application Packaging Tool?	89
9.3	Can Python 2.7 Be Supported?	90
9.4	No python interpreter found of version 3.* Error When Building	90
9.5	Why Rust?	90
9.6	Why is the Rust Code... Not Great?	91
9.7	What is the <i>Magic Sauce</i> That Makes PyOxidizer Special?	91
9.8	Can Applications Import Python Modules from the Filesystem?	91
9.9	error while loading shared libraries: libcrypt.so.1: cannot open shared object file: No such file or directory When Building	91
10	Project Status	93
10.1	What's Working	93
10.2	Major Missing Features	93
10.3	Lesser Missing Features	95
10.4	Eventual Features	95
11	Comparisons to Other Tools	99
11.1	PyInstaller	99

11.2	py2exe	99
11.3	py2app	100
11.4	cx_Freeze	100
11.5	Shiv	100
11.6	PEX	101
11.7	XAR	101
11.8	Docker / Running a Container	101
11.9	Nuitka	101
11.10	PyRun	101
11.11	pynsist	102
12	Contributing to PyOxidizer	103
12.1	As a User	103
12.2	As a Developer	103
12.3	Financial Contributions	103
13	Project History	105
13.1	Blog Posts	105
13.2	Version History	105
14	Technical Notes	119
14.1	CPython Initialization	119
14.2	CPython Importing Mechanism	120
14.3	<code>sys.modules</code> After Interpreter Init	122
14.4	Modules Imported by <code>site.py</code>	122
14.5	Random Notes	122
14.6	Desired Changes from Python to Aid PyOxidizer	123

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

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

The [pyoxidizer-users](#) mailing list is a forum for users to discuss all things PyOxidizer.

If you want to financially contribute to PyOxidizer, do so [on Patreon](#) or [via PayPal](#).

The creator and maintainer of PyOxidizer is [Gregory Szorc](#).

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 `PythonConfig` struct from the `pyembed` crate is created to define how an embedded Python interpreter should behave. (One of the build-time actions performed by `PyOxidizer` is to convert the 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`.

2.1 Installing

2.1.1 Installing Rust

PyOxidizer is a Rust application and requires Rust (1.36 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 1.38.0 (625451e37 2019-09-23)
```

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

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

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

2.1.2 Other System Dependencies

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

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

2.1.3 Installing PyOxidizer

PyOxidizer can be installed from its latest published crate:

```
$ cargo install pyoxidizer
```

From a Git repository using cargo:

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

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

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

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

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

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

```
$ pyoxidizer
PyOxidizer 0.5
Gregory Szorc <gregory.szorc@gmail.com>
Build and distribute Python applications

USAGE:
    pyoxidizer [SUBCOMMAND]

...
```

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

2.2 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.

2.3 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
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.

2.4 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.

2.5 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 line passing a `run_repl` argument, which configures the embedded interpreter to run a Python REPL. Let's replace that line with the following:

```
run_eval="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:

```
exe.add_in_memory_python_resources(dist.pip_install(["pyflakes==2.1.1"]))
```

In addition, replace the `run_*` argument to execute `pyflakes`:

```
run_eval="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`!

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.

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.

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.

4.1 Finding Configuration Files

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 File Processing Semantics

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 below for general usage and [Configuration File API Reference](#) for a full reference 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.

4.3 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(name, fn, depends=[], default=False, default_build_script=False)` 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(name, fn, depends=[], default=False, default_build_script=False)` for more.

The value returned by a target function is special. If that value is one of the special types defined by our Starlark dialect (e.g. `PythonDistribution` or `PythonExecutable`), PyOxidizer will attempt to invoke special functionality depending on the run mode. For example, when running `pyoxidizer build` to *build* a target, PyOxidizer will invoke any *build* functionality on the value returned by a target function, if present. For example, a `PythonExecutable`'s *build* functionality would compile an executable binary embedding Python.

4.4 Common Operations

4.4.1 Obtain a Python Distribution

A `PythonDistribution` type defines a Python distribution from which you can derive binaries, perform packaging actions, etc. Every configuration file will likely utilize this type.

Instances 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)
```

4.4.2 Creating an Executable File Embedding Python

A `PythonExecutable` type defines an executable file embedding Python.

Instances are derived from a `PythonDistribution` instance, usually by using target dependencies. In this example, we create an executable that runs a Python REPL on startup:


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

def make_exe(dist):
    return dist.to_python_executable(
        "myapp",
        run_repl=True,
    )

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

See *Packaging User Guide* for more examples.

4.4.3 Copying Files Next To Your Application

The *FileManifest* 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 application.

The *glob(include, exclude=None, strip_prefix=None)* 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(manifest)*. Here's an example:

```
def make_install():
    m = FileManifest()

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

    return m
```

This will take all files */path/to/project/templates/*, strip the path prefix */path/to/project/* from them and then add all those files to your main *FileManifest*. The files should be installed as *templates/** when the *InstallManifest* is materialized.

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 Managing Resources and Their Locations

An important concept in PyOxidizer packaging is how to manage *resources* and their *locations*.

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

A *location* is where that resource will be placed. Examples of *locations* included *embedded in the built binary* and *in a file next to the built binary*.

Resources are typically represented by a dedicated Starlark type. *Locations* are typically expressed through a function name.

5.2.1 Resource Types

The following Starlark types represent individual resources:

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

PythonBytecodeModule Bytecode for a Python module. Roughly equivalent to a `.pyc` file.

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.

FileContent Represents the content of a filesystem file.

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.)

5.2.2 Python Resource Locations

The `PythonEmbeddedResources` type represents a collection of Python resources of varying *resource* types and locations. When adding a Python resource to this type, you have the choice of multiple locations for the resource.

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* `PythonSourceModule` 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.

5.2.3 Python Resource Location Policies

When constructing a Starlark type that represents a collection of Python resources, the caller can specify a *policy* for what *locations* are allowed and how to handle a resource if no explicit *location* is specified. See *Python Resources Policy* for the full documentation.

Here are some examples of how policies are used:

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

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

    # Only allow resources to be added to the filesystem-relative location under
    # a "lib" directory.
    exe = dist.to_python_executable(
        name="myapp",
        resources_policy="filesystem-relative-only:lib",
```

(continues on next page)

(continued from previous page)

```

)

# Try to add resources to in-memory first. If that fails, add them to a
# "lib" directory relative to the built executable.
exe = dist.to_python_executable(
    name="myapp",
    resources_policy="prefer-in-memory-fallback-filesystem-relative:lib"
)

return exe

```

5.2.4 Routing Python Resources to Locations

Python resource collections have various APIs for adding resources to them. For example, to add a `PythonSourceModule` to a `PythonExecutable`:

```

def make_exe():
    dist = default_python_distribution()

    exe = dist.to_python_executable(
        name="myapp",
        resources_policy="prefer-in-memory-fallback-filesystem-relative:lib",
    )

    for resource in dist.pip_install(["my-package"]):
        if type(resource) == "PythonSourceModule":
            exe.add_in_memory_module_source(resource)
            exe.add_filesystem_relative_module_source("site-packages", resource)

```

These *resource addition* APIs are either *location-aware* or *location-agnostic*.

Location-aware APIs route a resource to a specific location, such as *in-memory* or *filesystem-relative*. Examples of these APIs include `PythonExecutable.add_module_source(module)` and `PythonExecutable.add_filesystem_relative_python_resource(prefix, ...)`.

Location-agnostic APIs route a resource to an appropriate location given the *resource location policy* for the container. e.g. if *in-memory-only* is in use, resources will be routed to the *in-memory* location. Examples of these APIs include `PythonExecutable.add_module_bytecode(module, optimize_level=0)` and `PythonExecutable.add_python_resources(...)`.

Resource addition APIs are either *type-aware* or *type-agnostic*.

Type-aware APIs require that the resource being passed in be a specific type or an error occurs. Examples of *type-aware* APIs include `PythonExecutable.add_filesystem_relative_module_source(prefix, module)` and `PythonExecutable.add_in_memory_package_resource(resource)`.

Type-agnostic APIs operate on any instance of an allowed type. It is safe to call these APIs with any accepted type. Examples of *type-agnostic* APIs include `PythonExecutable.add_python_resource(...)` and `PythonExecutable.add_in_memory_python_resources(...)`.

5.2.5 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.

The *location-agnostic* addition APIs will generally try to route a resource to an intelligent location based on the policy. And these APIs are a bit smarter about their actions than what is available in Starlark. For example, these APIs can see that both a static and shared library is available for an extension module and take a course of action that won't result in a build failure.

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](#).

5.3 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_in_memory_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*. In addition to defining the files constituting that distribution, a *PythonDistribution* exposes methods for performing Python packaging. One of those methods is *pip_install()*, which invokes `pip install` using that Python distribution.

To add a new Python package to our executable, we call `dist.pip_install()` then add the results to our *PythonExecutable* instance. This is done like so:

```
exe.add_in_memory_python_resources(dist.pip_install(["pyflakes==2.1.1"]))
```

The inner call to `dist.pip_install()` will effectively run `pip install pyflakes==2.1.1` and collect a set of installed Python resources (like module sources and bytecode data) and return that as an iterable data structure. The `exe.add_in_memory_python_resources()` call will then embed these resources in the built executable binary.

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

```
run_eval="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 = default_python_distribution()

    config = PythonInterpreterConfig(
        run_eval="from pyflakes.api import main; main()",
    )

    exe = dist.to_python_executable(
        name="pyflakes",
        config=config,
        extension_module_filter="all",
        include_sources=True,
        include_resources=False,
        include_test=False,
    )

    exe.add_in_memory_python_resources(dist.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`.

5.4 Packaging Additional Files

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

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

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

We start by creating a new project:

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

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

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

    config = PythonInterpreterConfig(
        run_module="black",
    )

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

    exe.add_in_memory_python_resources(dist.pip_install(["black==19.3b0"]))

    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. So `black` (and every other Python file assuming the existence of `__file__`) is arguably buggy.

Let's assume we can't easily change the offending source code to work around the issue.

To fix this problem, we change the configuration file to install `black` relative to the built application. This requires changing our approach a little. Before, we ran `dist.pip_install()` from `make_exe()` to collect Python resources and added them to a `PythonEmbeddedResources` instance. This meant those resources were embedded in the self-contained `PythonExecutable` instance returned from `make_exe()`.

Our auto-generated `pyoxidizer.bzl` file also contains an `install` *target* defined by the `make_install()` function. This target produces an `FileManifest`, which represents a collection of relative files and their content. When this type is *resolved*, those files are manifested on the filesystem. To package `black`'s Python resources next to our executable instead of embedded within it, we need to move the `pip_install()` invocation from `make_exe()` to `make_install()`.

Change your configuration file to look like the following:

```
def make_python_dist():
    return default_python_distribution()

def make_exe(dist):
    python_config = PythonInterpreterConfig(
        run_module="black",
        sys_paths=["$ORIGIN/lib"],
    )

    return dist.to_python_executable(
        name="black",
        config=python_config,
        extension_module_filter='all',
        include_sources=True,
        include_resources=False,
        include_test=False,
    )

def make_install(dist, exe):
    files = FileManifest()

    files.add_python_resource(".", exe)

    files.add_python_resources("lib", dist.pip_install(["black==19.3b0"]))

    return files

register_target("python_dist", make_python_dist)
register_target("exe", make_exe, depends=["python_dist"])
register_target("install", make_install, depends=["python_dist", "exe"], default=True)

resolve_targets()
```

There are a few changes here.

We added a new `make_dist()` function and `python_dist` *target* to represent obtaining the Python distribution. This isn't strictly required, but it helps avoid redundant work during execution.

The `PythonInterpreterConfig` construction adds a `sys_paths=["$ORIGIN/lib"]` argument. This argument says *adjust “sys.path” at run-time to include the “lib” directory next to the executable file*. It allows the Python interpreter to import Python files on the filesystem instead of just from memory.

The `make_install()` function/target has also gained a call to `files.add_python_resources()`. This method call takes the Python resources collected from running `pip install black==19.3b0` and adds them to the `FileManifest` instance under the `lib` directory. When the `FileManifest` is resolved, those Python

resources will be manifested as files on the filesystem (e.g. as `.py` and `.pyc` files).

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

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

If you examine the output, you'll see that various Python modules files were written to the output directory, just as our configuration file requested!

Let's try to run the application:

```
$ pyoxidizer run --path black --target install
No paths given. Nothing to do
```

Success!

5.5 Trimming Unused Resources

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

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

Essentially, all strategies for managing the set of packaged resources boil down to crafting 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.

5.6 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.

5.6.1 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
```

(continues on next page)

(continued from previous page)

```
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.

5.6.2 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'
→ 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!

5.7 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.

5.7.1 C and Other Native Extension Modules

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

The way this typically works is some build system (often `distutils` via a `setup.py` script) produces a shared library file containing the extension. On Linux and macOS, the file extension is typically `.so`. On Windows, it is `.pyd`. When an `import` is requested, Python's importing mechanism looks for these files in addition to normal `.py` and `.pyc` files. If an extension module is found, Python will `dlopen()` the file and load the shared library into the process. It will then call into an initialization function exported by that shared library to obtain a Python module instance.

Python packaging has defined various conventions for distributing pre-compiled extension modules in *wheels*. If you see an e.g. `<package>-<version>-cp38-cp38-win_amd64.whl`, `<package>-<version>-cp38-cp38-manylinux2014_x86_64.whl`, or `<package>-<version>-cp38-cp38-macosx_10_9_x86_64.whl` file, you are installing a Python package with a pre-compiled extension module. Inside the *wheel* is a shared library providing the extension module. And that shared library is configured to work with a Python distribution (typically `CPython`) built in a specific way, e.g. with a `libpythonXY` shared library exporting Python symbols.

PyOxidizer currently has *some support* for extension modules. The way this works depends on the platform and Python distribution.

Dynamically Linked Python Distributions on Windows

When using a dynamically linked Python distribution on Windows (e.g. via the `flavor="standalone_dynamic"` argument to `default_python_distribution(flavor="standalone",`

`build_target=None`), PyOxidizer:

- Supports importing shared library extension modules (e.g. `.pyd` files) from memory.
- Automatically detects and uses `.pyd` files from pre-built binary packages installed as part of packaging.
- Automatically detects and uses `.pyd` files produced during package building.

However, there are caveats to this support!

PyOxidizer doesn't currently support resolving additional library dependencies from `.pyd` extension modules / shared libraries when importing from memory. If an extension module depends on another shared library (almost certainly a `.dll`) outside the normal set of libraries (namely the C Runtime and other common Windows system DLLs), you will need to manually package this library next to the application `.exe`. Failure to do this could result in a failure at import time.

PyOxidizer does support loading shared library extension modules from `.pyd` files on the filesystem like a typical Python program. So if you cannot make in-memory extension module importing work, you can fall back to packaging a `.pyd` file in a directory registered on `sys.path`, as set through the [PythonInterpreterConfig\(...\)](#) Starlark primitive.

Extension Modules Everywhere Else

If PyOxidizer is not able to easily reuse a Python extension module built or distributed in a traditional manner, it will attempt to compile the extension module from source in a way that is compatible with the PyOxidizer distribution and application configuration.

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

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

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

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

5.7.2 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
```

(continues on next page)

(continued from previous page)

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

5.8 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 `EmbeddPythonConfig` 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.

5.9 Binary Compatibility

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

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

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

5.9.1 Windows

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

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

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

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

5.9.2 macOS

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

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

5.9.3 Linux

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

If you are linking against `libc.so`, things get more complicated because the binary will probably link against the `glibc` symbol versions that were present on the build machine. To ensure maximum binary compatibility, compile your binary in a Debian 7 or 8 environment, as this will use a sufficiently old version of `glibc` which should work in most Linux environments.

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

5.10 Static Linking

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

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

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

Then you can build away:

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

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

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

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

5.10.2 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](#).

5.11 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.

5.11.1 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.

5.12 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` 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 `pyembd::osutils` Rust module).

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

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 meta-data, 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.

6.1 Getting Started

6.1.1 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.

6.1.2 Installing from PyPI

`oxidized_importer` is [available](#) on PyPI. This means that installing is as simple as:

```
$ pip3 install oxidized_importer
```

6.1.3 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.:

```
$ python3.8 setup.py build_ext -i
$ python3.8 setup.py install

$ pip3.8 install .
$ pip3.8 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 3.8 `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/python3.8 cargo build
```

6.1.4 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.

6.2 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.

6.3 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`.

6.3.1 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).

6.3.2 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](#).

6.3.3 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:

resources_data Bytes-like *packed resources data* to parse. A reference to the passed in value will be stored internally in the constructed instance, as the memory needs to live for the lifetime of the `OxidizedFinder` instance.

resources_file A path-like object defining the filesystem path to a file containing *packed resources data*. If provided, the file will be opened and memory mapped and resources data will be parsed from it.

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.

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.flavor = "module"
    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 *builtin* 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.

6.4 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.

6.4.1 `__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. See [Python Resource Locations](#) for more on registering files for filesystem loading. 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.

6.4.2 `__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.

6.4.3 ResourceReader Compatibility

`ResourceReader` has known compatibility differences with Python's default filesystem-based importer. See [Support for ResourceReader](#) for details.

6.4.4 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.

6.4.5 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:

- `locate_file()` is not defined.
- `@classmethod from_name()` is not defined.
- `@classmethod discover()` 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.

6.5 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.

6.5.1 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 `flavor = "none"` and `name = ""`. All properties will be `None` or `false`.

Properties

The following properties/attributes exist on `OxidizedResource` instances:

flavor A `str` describing the *flavor* of this resource. See [OxidizedResource Flavors](#) for more.

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_opt2 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 Flavors

Each `OxidizedResource` instance describes a particular type of resource. The type is indicated by a `flavor` property on the instance.

The following flavors are defined:

none There is no resource flavor (you shouldn't see this).

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.

built-in A built-in extension module. These represent Python extension modules that are compiled into the application and don't exist as separate shared libraries.

frozen A frozen Python module. These are Python modules whose bytecode is compiled into the application.

extension A Python extension module. These are shared libraries that can be loaded to provide additional modules to Python.

shared_library A shared library. e.g. a `.so` or `.dll`.

6.5.2 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.

6.5.3 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.

6.5.4 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.

6.5.5 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.

6.5.6 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.

6.6 Resource Scanning APIs

The `oxidized_importer` module exposes functions and Python types to facilitate scanning for and collecting Python resources.

6.6.1 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))
```

6.6.2 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 a `policy=<str>` argument defining the resources policy for this collector. The string values are the same as recognized by PyOxidizer's config files and are documented at [Python Resources Policy](#).

e.g. to create a collector that only marks resources for in-memory loading:

```
import oxidized_importer

collector = oxidized_importer.OxidizedResourceCollector(policy="in-memory-only")
```

Instances of `OxidizedResourceCollector` have the following properties:

policy (str) Exposes the policy string this instance was constructed with. This property is read-only.

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.

6.7 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.

6.7.1 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.

6.7.2 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 *Python 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.

6.7.3 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).

6.7.4 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 [Python Resource Locations](#) from in-memory to filesystem-relative, as `__file__` is set when loading modules from the filesystem.

6.7.5 Support for `pkg_resources`

`pkg_resources`'s APIs for loading resources likely do not work with `oxidized_importer`.

6.7.6 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.

6.7.7 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
    # 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.
```

(continues on next page)

(continued from previous page)

```

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.

6.8 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.

6.8.1 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.

6.8.2 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(policy="in-memory-only")

# 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)

# 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.
```

(continues on next page)

(continued from previous page)

```

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.

6.8.3 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(resources_file="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(
    resources=packed_data,
    relative_path_origin=os.path.dirname(os.path.abspath(__file__)),
)

# 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!

```

6.9 Common Issues

6.9.1 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(
    policy="prefer-in-memory-fallback-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)
```

6.9.2 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.

6.10 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>>.

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.

7.1 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.

7.1.1 Layout

Generated Rust projects all have a similar layout:

```
$ find pyapp -type f | grep -v .git
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*.

7.2 Crate Configuration

7.2.1 Build Artifacts for `pyembed`

The `pyembed` crate needs to reference special artifacts as part of its build process in order to compile a Python interpreter into a binary. The most important of these artifacts is a library providing Python symbols.

By default, the `pyembed` crate's `build.rs` build script will run `pyoxidizer run-build-script`, which will attempt to find a PyOxidizer config file and evaluate its default *build script* target. Using environment variables set by `cargo` and by the `build.rs` script, artifacts will be placed in the correct locations and `pyembed` will be built seemingly like any normal Rust crate.

The special build artifacts are generated by resolving a configuration file target returning a *PythonEmbeddedResources* instance. In the auto-generated configuration file, the `embedded` target returns such a type.

7.2.2 Cargo Features to Control Building

The `pyembed` crate and generated Rust projects share 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-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.

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 the heuristics described at [Finding Configuration Files](#). `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`. This directory **must** have a `cargo_metadata.txt` file, which will be printed to stdout by the build script to tell Cargo how to link a Python library.

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 pre-built 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.

7.3 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.

7.3.1 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.

Under the hood, the `pyembed` crate uses the `cpython` and `python3-sys` crates for interacting with Python's C APIs. `pyembed` exposes the `Python` object from `cpython`, which means that once you've initialized a Python interpreter with `pyembed`, you can use all the functionality in `cpython` to interact with that interpreter.

7.3.2 Initializing a Python Interpreter

Initializing an embedded Python interpreter in your Rust process is as simple as calling `pyembed::MainPythonInterpreter::new(config: PythonConfig)`.

The hardest part about this is constructing the `pyembed::PythonConfig` instance.

Using the Default PythonConfig

If the `pyembed` crate is configured to emit build artifacts (the default), its build script will generate a Rust source file containing a `fn default_python_config() -> pyembed::PythonConfig` which emits a `pyembed::PythonConfig` using the configuration as defined by the utilized PyOxidizer *configuration file*. Assuming you are using the boilerplate `Cargo.toml` and `build.rs` script generated with `pyoxidizer init-rust-project`, the path to this generated source file will be in the `PYOXIDIZER_DEFAULT_PYTHON_CONFIG_RS` environment variable.

This all means that to use the auto-generated `pyembed::PythonConfig` 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::PythonConfig = default_python_config();

    pyembed::MainPythonInterpreter::new(config)
}
```

Using a Custom PythonConfig

If you don't want to use the default `pyembed::PythonConfig` instance, that's fine too! However, this will be slightly more complicated.

First, if you use an explicit `PythonConfig`, 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 `PythonConfig` Rust struct. And if you are using your own custom `PythonConfig` 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 `jemalloc`, you will need to enable a Cargo feature when building the `pyembed` crate or else you will get a run-time error that `jemalloc` is not available.

`pyembed::PythonConfig::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
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 `PythonConfig` instance.

This error can be addressed by working around either.

To enable the default filesystem importer:

```
let mut config = pyembed::PythonConfig::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::PythonConfig::default();
config.packed_resources = ...;
config.use_custom_importlib = true;
```

The `packed_resources` field defines a reference to *packed resources data* (a `& [u8]`). This is a custom serialization format for expressing *resources* to make available to a Python interpreter. See the `python-packed-resources` crate for the format specification and code for serializing it. Again, the easiest way to obtain this data blob is by using PyOxidizer and consuming the `packed-resources` build artifact/file, likely through `include_bytes!`.

Finally, setting `use_custom_importlib = true` is necessary to enable the custom bytecode and meta path importer to be used at run-time.

7.3.3 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().unwrap();

    match pyembed::run_code(py, "print('hello, world')") {
        Ok(_) => print("python code executed successfully"),
        Err(e) => print("python error: {:?}", e),
    }
}
```

The `pyembed` crate exports various `run_*` functions for performing high-level evaluation of various primitives (files, modules, code strings, etc). See the `pyembed` crate's documentation for more.

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.

7.3.4 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.

7.3.5 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!

7.4 Adding Extension Modules At Run-Time

Normally, PyOxidizer assembles all extension modules needed for a built application and the resources data embedded in the binary describes all extension modules.

The `pyembed` crate also supports providing additional extension modules, which are defined outside of PyOxidizer configuration files. This feature can be useful for Rust applications that want to provide extension modules through their own means and don't want to use standard Python packaging tools (like `setup.py`) or PyOxidizer config files for building them.

7.4.1 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 `PythonConfig` 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.

7.4.2 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 build configuration supports loading dynamically linked Python extension modules. See [PythonExtensionModule Location Compatibility](#) for more on this topic.

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`.

Configuration File API Reference

This document describes the low-level API for `PyOxidizer` configuration files. For a higher-level overview of how configuration files work, see [Configuration Files](#).

8.1 Global Symbols

The following are all global symbols available by default in the Starlark environment:

- Starlark built-ins.
- `BUILD_TARGET_TRIPLE`
- `CONFIG_PATH`
- `CONTEXT`
- `CWD`
- `default_python_distribution(flavor="standalone", build_target=None)`
- `FileManifest()`
- `glob(include, exclude=None, strip_prefix=None)`
- `PythonBytecodeModule`
- `PythonDistribution(sha256, local_path=None, url=None, flavor="standalone")`
- `PythonEmbeddedResources`
- `PythonExecutable`
- `PythonExtensionModule`
- `PythonInterpreterConfig(...)`
- `PythonPackageDistributionResource`
- `PythonPackageResource`

- *PythonSourceModule*
- *register_target(name, fn, depends=[], default=False, default_build_script=False)*
- *resolve_target(target)*
- *resolve_targets()*
- *set_build_path(path)*

8.2 Types

The following custom data types are defined in the Starlark environment:

FileContent Represents the content of a file on the filesystem.

FileManifest Represents a mapping of filenames to file content.

PythonBytecodeModule Represents a `.pyc` file containing Python bytecode for a given module.

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.

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

8.3 Constants

PyOxidizer provides global constants as defined by the following sections.

8.3.1 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.

8.3.2 CONFIG_PATH

The string path to the configuration file currently being evaluated.

8.3.3 CONTEXT

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

8.3.4 CWD

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

8.4 Functions for Manipulating Global State

8.4.1 `set_build_path(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.

8.5 Functions for Managing Targets

8.5.1 `register_target(name, fn, depends=[], default=False, default_build_script=False)`

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.

`depends` is an optional 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` 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` 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.

8.5.2 `resolve_target(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.

8.5.3 `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.

8.6 Python Distributions

Python distributions are entities that define an implementation of Python that can be used to create a binary embedding Python and that can be used to execute Python code.

Python distributions are defined by the `PythonDistribution` type. This type can be constructed from parameters or via `default_python_distribution(flavor="standalone", build_target=None)`.

8.6.1 `PythonDistribution(sha256, local_path=None, url=None, flavor="standalone")`

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

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

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

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

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`.

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"
)
```


8.6.2 `default_python_distribution(flavor="standalone", build_target=None)`

Resolves the default `PythonDistribution` for the given distribution flavor and build target, which default to a `standalone` distribution and the active build target as defined by `BUILD_TARGET`, respectively.

`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.

standalone_static This is like `standalone` but the distribution must have a statically linked `libpython`.

standalone_dynamic This is like `standalone` but the distribution must have a dynamically linked `libpython`.

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.

8.6.3 `PythonDistribution` Methods

`PythonDistribution.source_modules()`

Returns a list of `PythonSourceModule` representing Python source modules present in this distribution.

`PythonDistribution.package_resources(include_test=False)`

Returns a list of `PythonPackageResource` representing resource files present in this distribution.

The `include_test` boolean argument controls whether resources associated with test packages are included.

`PythonDistribution.extension_modules(filter='all', preferred_variants=None)`

Returns a list of `PythonExtensionModule` representing extension modules in this distribution.

The `filter` argument denotes how to filter the extension modules. 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-gpl Return only extension modules that do not link against GPL licensed libraries.

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

The `preferred_variants` argument denotes a string to string mapping of extension module name to its preferred variant name. If multiple variants of an extension module meet the filter requirements, the preferred variant from this mapping will be used. Otherwise the first variant will be used.

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.

`PythonDistribution.pip_install(args, extra_envs={})`

This method runs `pip install <args>` with the specified distribution.

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 `PythonSourceModule`, `PythonBytecodeModule`, `PythonPackageResource`, etc.

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

`PythonDistribution.read_package_root(path, packages)`

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 `PythonSourceModule`, `PythonBytecodeModule`, `PythonPackageResource`, etc.

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

`PythonDistribution.read_virtualenv(path)`

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 `PythonSourceModule`, `PythonBytecodeModule`, `PythonPackageResource`, etc.

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

`PythonDistribution.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 `PythonSourceModule`, `PythonBytecodeModule`, `PythonPackageResource`, etc.

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

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

resources_policy (str) The policy to apply when adding resources to the produced instance.

See *Python Resources Policy* for documentation on allowed values. The default value is `in-memory-only`.

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

Default is what `PythonInterpreterConfig()` returns.

extension_module_filter (str) The filter to apply to determine which extension modules to add.

See *PythonDistribution.extension_modules(filter='all', preferred_variants=None)* for what values are accepted and their behavior.

Default is `all`.

preferred_extension_module_variants (dict of string to string) Preferred extension module variants to use. See *PythonDistribution.extension_modules(filter='all', preferred_variants=None)* for behavior.

Default is `None`, which will use the first variant.

include_sources (bool) Boolean to control whether sources of Python modules are added in addition to bytecode.

Default is `True`.

include_resources (bool) Boolean to control whether non-module resource data from the distribution is added.

Default is `False`.

include_test (bool) Boolean to control whether test-specific objects are included.

Default is `False`.

8.7 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 types *PythonSourceModule*, *PythonBytecodeModule*, *PythonPackageResource*, *PythonPackageDistributionResource*, and *PythonExtensionModule*.

These are described in detail in the following sections.

8.7.1 PythonSourceModule

This type represents Python source modules, agnostic of location.

Each instance has the following attributes:

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

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

Instances cannot be manually constructed.

8.7.2 PythonBytecodeModule

This type represents a Python module defined through bytecode.

Each instance has the following attributes:

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

optimize_level (int) Optimization level of compiled bytecode. Must be the value 0, 1, or 2.

is_package (bool) Whether the module is also a Python package (or sub-package).

8.7.3 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.

Each instance has the following attributes:

package (string) Python package this resource is associated with.

name (string) Name of this resource.

8.7.4 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.

Each instance has the following attributes:

package (string) Python package this resource is associated with.

name (string) Name of this resource.

8.7.5 PythonExtensionModule

This type represents a compiled Python extension module.

Each instance has the following attributes:

name (string) Unique name of the module being provided.

8.8 Python Resources Policy

There are various ways to add resources (typically Python resources) to a binary. For example, you can import modules from memory or the filesystem. Often, configuration files may wish to be explicit about what behavior is and is not allowed. A *Python Resources Policy* is used to apply said behavior.

A *Python Resources Policy* is defined by a `str`. The following values are recognized.

in-memory-only Resources are to be loaded from in-memory only. If a resource cannot be loaded from memory (e.g. dynamically linked Python extension modules in some configurations), an error will (likely) occur.

filesystem-relative-only:<prefix> Values starting with `filesystem-relative-only:` specify that resources are to be loaded from the filesystem from paths relative to the produced binary. Files will be installed at the path prefix denoted by the value after the `..` e.g. `filesystem-relative-only:lib` will install resources in a `lib/` directory.

prefer-in-memory-fallback-filesystem-relative:<prefix> Values starting with `prefer-in-memory-fallback-filesystem-relative` represent a hybrid between `in-memory-only` and `filesystem-relative-only:<prefix>`. Essentially, if in-memory resource loading is supported, it is used. Otherwise we fall back to loading from the filesystem from paths relative to the produced binary.

8.9 Python Interpreter Configuration

A Python interpreter has settings to control how it runs. Configuration files represent these settings through the *PythonInterpreterConfig(...)* type.

8.9.1 PythonInterpreterConfig(...)

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

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

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

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

bytes_warning (int) Controls the value of `Py_BytesWarningFlag`.

Default is 0.

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

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

ignore_environment (bool) Controls the value of `Py_IgnoreEnvironmentFlag`.

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

Default is `True`.

inspect (bool) Controls the value of `Py_InspectFlag`.

Default is `False`.

interactive (bool) Controls the value of `Py_InteractiveFlag`.

Default is `False`.

isolated (bool) Controls the value of `Py_IsolatedFlag`.

legacy_windows_fs_encoding (bool) Controls the value of `Py_LegacyWindowsFSEncodingFlag`.

Only affects Windows.

Default is `False`.

legacy_windows_stdio (bool) Controls the value of `Py_LegacyWindowsStdioFlag`.

Only affects Windows.

Default is `False`.

optimize_level (bool) Controls the value of `Py_OptimizeFlag`.

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

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

parser_debug (bool) Controls the value of `Py_DebugFlag`.

Default is `False`.

quiet (bool) Controls the value of `Py_QuietFlag`.

raw_allocator (string) Which memory allocator to use for the `PYMEM_DOMAIN_RAW` allocator.

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

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

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

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

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

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

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

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

`run_eval (string)` Will cause the interpreter to evaluate a Python code string defined by this value after the interpreter initializes.

An example value would be `import mymodule; mymodule.main()`.

`run_file (string)` Will cause the interpreter to evaluate a file at the specified filename.

The filename is resolved at run-time using whatever mechanisms the Python interpreter applies. i.e. this is little different from running `python <path>`.

`run_module (string)` The Python interpreter will load a Python module with this value's name as the `__main__` module and then execute that module.

This mode is similar to `python -m <module>` but isn't exactly the same. `python -m <module>` has additional functionality, such as looking for the existence of a `<module>.__main__` module. PyOxidizer does not do this. The value of this argument will be the exact module name that is imported and run as `__main__`.

`run_noop (bool)` Instructs the Python interpreter to do nothing after initialization.

`run_repl (bool)` The Python interpreter will launch an interactive Python REPL connected to `stdio`. This is similar to the default behavior of running a `python` executable without any arguments.

`site_import (bool)` Controls the inverse value of `Py_NoSiteFlag`.

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

Default is `False`.

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

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

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

Default is `False`.

`sys_meipass (bool)` Controls whether to set the `sys._MEIPASS` attribute to the path of the executable.

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

Default is `False`.

`sys_paths (array of strings)` Defines filesystem paths to be added to `sys.path`.

Setting this value will imply `filesystem_importer = true`.

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

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

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

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

See *Terminfo Database* for more about terminal databases.

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

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

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

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

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

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

unbuffered_stdio (bool) Controls the value of `Py_UnbufferedStdioFlag`.

Setting this makes the standard I/O streams unbuffered.

Default is `False`.

use_hash_seed (bool) Controls the value of `Py_HashRandomizationFlag`.

Default is `False`.

user_site_directory (bool) Controls the inverse value of `Py_NoUserSiteDirectory`.

Default is `False`.

write_bytecode (bool) Controls the inverse value of `Py_DontWriteBytecodeFlag`.

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

Default is `False`.

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

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

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

8.10 Python Binaries

Binaries containing an embedded Python interpreter can be defined by configuration files. They are defined via the *PythonExecutable* type. In addition, the *PythonEmbeddedResources* type represents the collection of resources made available to an embedded Python interpreter.

8.10.1 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.

8.10.2 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(...)*.

`PythonExecutable.add_in_memory_module_source(module)`

This method registers a Python source module with a *PythonExecutable* instance. The module will be imported from memory at run-time. The argument must be a *PythonSourceModule* instance.

If called multiple times for the same module, the last write wins.

`PythonExecutable.add_filesystem_relative_module_source(prefix, module)`

This method registers a Python source module with a *PythonExecutable* instance. The module will be imported from the filesystem from a file relative to the built executable. The directory prefix for the generated file is defined by *prefix*.

If called multiple times for the same module, the last write wins.

`PythonExecutable.add_module_source(module)`

This method registers a Python source module with a `PythonExecutable` instance.

This method is a glorified proxy to `PythonExecutable.add_in_memory_module_source(module)` or `PythonExecutable.add_filesystem_relative_module_source(prefix, module)` depending on the *Python Resources Policy* in effect.

`PythonExecutable.add_in_memory_module_bytecode(module, optimize_level=0)`

This method registers Python module bytecode with a `PythonExecutable` instance. The module will be imported from memory at run-time.

The first argument must be a `PythonSourceModule` instance. The 2nd argument the value 0, 1, or 2.

`PythonExecutable.add_filesystem_relative_module_bytecode(prefix, module, optimize_level=0)`

This method registers Python module bytecode with a `PythonExecutable` instance. The module will be imported from the filesystem from a file relative to the built executable. The directory prefix for the generated file is defined by `prefix`.

The `module` argument must be a `PythonSourceModule` instance. The `optimize_level` argument must be the value 0, 1, or 2.

If called multiple times for the same module, the last write wins.

`PythonExecutable.add_module_bytecode(module, optimize_level=0)`

This method registers a Python module bytecode with a `PythonExecutable` instance.

This method is a glorified proxy to `PythonExecutable.add_in_memory_module_bytecode(module, optimize_level=0)` or `PythonExecutable.add_filesystem_relative_module_bytecode(prefix, module, optimize_level=0)` depending on the *Python Resources Policy* in effect. See these other methods for documentation of behavior.

`PythonExecutable.add_in_memory_package_resource(resource)`

This method adds a `PythonPackageResource` instance to the `PythonExecutable` instance, making that resource available via in-memory access.

If multiple resources sharing the same `(package, name)` pair are added, the last added one is used.

`PythonExecutable.add_filesystem_relative_package_resource(prefix, resource)`

This method adds a `PythonPackageResource` instance to the `PythonExecutable` instance. The resource will be materialized on the filesystem next to the produced executable at a path derived from the resource's attributes. The directory prefix for the generated file is defined by `prefix`.

If multiple resources sharing the same `(prefix, package, name)` tuple are added, the last added one is used.

PythonExecutable.add_package_resource(resource)

This method adds a `PythonPackageResource` instance to the `PythonExecutable` instance.

This method is a glorified proxy to `PythonExecutable.add_in_memory_package_resource(resource)` or `PythonExecutable.add_filesystem_relative_package_resource(prefix, resource)` depending on the *Python Resources Policy* in effect. See these other methods for documentation of behavior.

PythonExecutable.add_in_memory_package_distribution_resource(resource)

This method adds a `PythonPackageDistributionResource` instance to the `PythonExecutable` instance, making that resource available via in-memory access.

If multiple resources sharing the same (package, name) pair are added, the last added one is used.

PythonExecutable.add_filesystem_relative_package_distribution_resource(prefix, resource)

This method adds a `PythonPackageDistributionResource` instance to the `PythonExecutable` instance. The resource will be materialized on the filesystem next to the produced executable at a path derived from the resource's attributes. The directory prefix for the generated file is defined by `prefix`.

If multiple resources sharing the same (prefix, package, name) tuple are added, the last added one is used.

PythonExecutable.add_package_distribution_resource(resource)

This method adds a `PythonPackageDistributionResource` instance to the `PythonExecutable` instance.

This method is a glorified proxy to `PythonExecutable.add_in_memory_package_distribution_resource(resource)` or `PythonExecutable.add_filesystem_relative_package_distribution_resource(prefix, resource)` depending on the *Python Resources Policy* in effect. See these other methods for documentation of behavior.

PythonExecutable.add_in_memory_extension_module(module)

This method registers a `PythonExtensionModule` instance with a `PythonExecutable` instance. The extension module will be loaded from memory via one of the following mechanisms:

- Linking the extension module's symbols directly into the produced binary.
- Embedded the extension module's shared library into the produced binary and loading it from memory.

If multiple extension modules with the same name are added, the last added one is used.

PythonExecutable.add_filesystem_relative_extension_module(prefix, module)

This method registers a `PythonExtensionModule` instance with a `PythonExecutable` instance. The extension module will be loaded from the filesystem from a path relative to the produced executable.

If multiple extension modules with the same name are added, the last added one is used.

`PythonExecutable.add_extension_module(module)`

This method registers a `PythonExtensionModule` instance with a `PythonExecutable` instance. The extension module will be made available to the executable using whatever means are possible, constrained by the *Python Resources Policy* set by this `PythonExecutable`.

The extension module could be packaged in the following manner depending on the capabilities of the underlying Python distribution and resources policy:

- 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.)

If multiple extension modules with the same name are added, the last added one is used.

`PythonExecutable.add_in_memory_python_resource(...)`

This method registers a Python resource of various types for in-memory loading. It accepts a `resource` argument which can be a `PythonSourceModule`, `PythonBytecodeModule`, `PythonPackageResource`, or `PythonExtensionModule` and registers that resource with this instance. This method is a glorified proxy to the appropriate `add_in_memory_*` method.

The following arguments are accepted:

resource The resource to add to the embedded Python environment.

add_source_module (bool) When the resource is a `PythonSourceModule`, this flag determines whether to add the source for that resource.

Default is `True`.

add_bytecode_module (bool) When the resource is a `PythonSourceModule`, this flag determines whether to add the bytecode for that module source.

Default is `True`.

optimize_level (int) Bytecode optimization level when compiling bytecode.

`PythonExecutable.add_filesystem_relative_python_resource(prefix, ...)`

This method registers a Python resource of various types for filesystem loading in a path relative to the produced executable. The arguments for this method are the same as `PythonExecutable.add_in_memory_python_resource(...)` except the first argument is the `str` `prefix` to install files into.

`PythonExecutable.add_python_resource(...)`

This method registers a Python resource of various types with the instance. The location the resource will be loaded from will be determined by the *Python Resources Policy* in effect for the `PythonExecutable`.

The arguments are the same as for `PythonExecutable.add_in_memory_python_resource(...)`.

`PythonExecutable.add_in_memory_python_resources(...)`

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

`PythonExecutable.add_filesystem_relative_python_resources(prefix, ...)`

This method registers an iterable of Python resources of various types. This method is identical to `PythonExecutable.add_filesystem_relative_python_resource(prefix, ...)` except the first argument is a path prefix to install files to and the second argument is an iterable of resources. All other arguments are identical.

`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(files=[], glob_patterns=[])`

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.

8.11 Interacting With the Filesystem

8.11.1 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.

FileManifest.add_manifest(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_python_resource(prefix, value)

This method adds a Python resource to a `FileManifest` instance in a specified directory prefix. A *Python resource* here can be a `PythonSourceModule`, `PythonBytecodeModule`, `PythonPackageResource`, `PythonPackageDistributionResource`, or `PythonExtensionModule`.

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

FileManifest.add_python_resources(prefix, values)

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()
m.add_python_resources(dist.source_modules())
```

FileManifest.install(path, replace=True)

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`.

8.11.2 FileContent

This type represents the content of a single file.

8.11.3 glob(include, exclude=None, strip_prefix=None)

The `glob()` function resolves file patterns to a `FileManifest`.

`include` is a list of `str` containing 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` is an optional list of `str` and is used to exclude files from the result. All patterns in `include` are evaluated before `exclude`.

`strip_prefix` is an optional `str` to strip from the beginning of matched files. `strip_prefix` is stripped after `include` and `exclude` are processed.

Returns a `FileManifest`.

Frequently Asked Questions

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

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

9.2 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.

9.3 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.

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

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

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

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

9.5 Why Rust?

This is really 2 separate questions:

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

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

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

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

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

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

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

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

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

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

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

Second is the ability to import `.py/.pyc` files from memory. Most other self-contained Python applications rely on Python's `zipimporter` or do work at run-time to extract the standard library to a filesystem (typically a temporary directory or a FUSE filesystem like `SquashFS`). What `PyOxidizer` does is expose the `.py/.pyc` modules data to the Python interpreter via a Python extension module built-in to the binary.

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.

9.8 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 Resources and Their Locations*) or by enabling Python's standard filesystem-based importer by enabling `filesystem_importer=True` (see `PythonInterpreterConfig(...)`).

9.9 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.

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>!

10.1 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.

10.2 Major Missing Features

10.2.1 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.

10.2.2 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*.

10.2.3 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.

10.2.4 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.

10.2.5 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.

10.2.6 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.

10.3 Lesser Missing Features

10.3.1 Python Version Support

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

10.3.2 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.

10.3.3 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).

10.3.4 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 an empty import library to satisfy requirements of the `static` linkage kind. See <https://github.com/rust-lang/rust/issues/26591#issuecomment-123513631> for more.

10.3.5 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.

10.3.6 Configuration Files

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

10.4 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.

10.4.1 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.

10.4.2 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.

10.4.3 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).

10.4.4 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 [infern crate](#) to emit flame graphs directly from Rust, without having to use external tools.

10.4.5 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.

10.4.6 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.

10.4.7 PyO3

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

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

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.

11.1 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 byte-code). `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` typically skips this step and loads modules directly from memory using zero-copy. This makes `PyOxidizer` executables significantly faster to start.

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

11.2 py2exe

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

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

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

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

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

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

11.3 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`.

11.4 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.

11.5 Shiv

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

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

Also, by distributing our own Python interpreter with the application, `PyOxidizer` has stronger guarantees about the run-time environment. For example, your application can aggressively target the latest Python version. Another benefit of distributing your own Python interpreter is you can run a Python interpreter with various optimizations,

such as profile-guided optimization (PGO) and link-time optimization (LTO). You can also easily configure custom memory allocators or tweak memory allocators for optimal performance.

11.6 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.

11.7 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.

11.8 Docker / Running a Container

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

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

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

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

11.9 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`.

11.10 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.

11.11 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.

Contributing to PyOxidizer

This page documents how to contribute to PyOxidizer.

12.1 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.

12.2 As a Developer

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

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

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

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

12.3 Financial Contributions

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

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

Work on PyOxidizer started in November 2018 by Gregory Szorc.

13.1 Blog Posts

- [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 \(2018-12-18\)](#)

13.2 Version History

13.2.1 0.8.0-pre

Not yet released.

Backwards Compatibility Notes

- 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 default Python distributions have been upgraded to CPython 3.8.2 (from 3.7.7) and support for Python 3.7 has been removed.
- Windows static distributions have been removed (temporary regression which should be corrected before a release).
- The `pyembed` crate now defines a new `OxidizedPythonInterpreterConfig` type to configure Python interpreters. `PythonConfig` still exists and can be converted into a `OxidizedPythonInterpreterConfig` using `.into()`.
- 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 1.39 to allow for upgrading various dependencies to modern versions.

New Features

- 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.
- Python distributions upgraded to CPython 3.8.2.
- 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.
- 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.

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`.

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.

13.2.2 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 `resources_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 `resources_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.

13.2.3 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 re-ordered 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 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`.

13.2.4 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.)

13.2.5 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(include, exclude=None, strip_prefix=None)` 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.

13.2.6 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.

13.2.7 0.3.0

Released on August 16, 2019.

Backwards Compatibility Notes

- The `pyembded::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.

13.2.8 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.

13.2.9 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)

13.2.10 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.

13.2.11 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)

13.2.12 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.

14.1 CPython Initialization

Most code lives in `pylifecycle.c`.

Call tree with Python 3.7:

```

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

  ``_Py_InitializeMainInterpreter(interp, _PyMainInterpreterConfig)``

```

(continues on next page)

(continued from previous page)

```

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

```

14.2 CPython Importing Mechanism

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

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

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

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

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

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

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

C implemented `_imp` module is initialized.

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

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

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

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

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

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

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

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

14.3 `sys.modules` After Interpreter Init

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

14.4 Modules Imported by `site.py`

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

14.5 Random Notes

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

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

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

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

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

14.6 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.

14.6.1 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.

14.6.2 `__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.

14.6.3 `importlib.metadata` Documentation Deficiencies

See <https://bugs.python.org/issue38594>.

14.6.4 `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.

14.6.5 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.

14.6.6 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 imported. 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.

14.6.7 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.)

14.6.8 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.