PyOxidizer

Release 0.21.0-pre

Gregory Szorc

Jun 05, 2022

CONTENTS

Welcome to the unified documentation of the PyOxidizer Project, a collection of libraries and tools attempting to improve ergonomics around packaging and distributing [Python] applications.

The official home of the project is https://github.com/indygreg/PyOxidizer. Official documentation lives on Read the Docs (unreleased/latest commit, last release).

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

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

MULTIPLE TOOLS UNDER ONE ROOF

The PyOxidizer Project is comprised of discrete pieces of software developed in the same repository. Major pieces of user-facing software have their own documentation, each described in the following sections.

1.1 Apple Code Signing

The `apple-codesign` Rust crate and `rcodesign` CLI tool implement Apple code signing to enable developers to sign, notarize, and staple software without having to use Apple hardware or macOS.

1.1.1 Apple Code Signing

The `apple-codesign` Rust crate and its corresponding `rcodesign` CLI tool implement code signing for Apple platforms.

We believe this crate provides the most comprehensive implementation of Apple code signing outside the canonical Apple tools. We have support for the following features:

- Signing Mach-O binaries (the executable file format on Apple operating systems).

- Signing, notarizing, and stapling directory bundles (e.g. `.app` directories).

- Signing, notarizing, and stapling XAR archives / `.pkg` installers.

- Signing, notarizing, and stapling disk images / `.dmg` files.

What this all means is that you can sign, notarize, and release Apple software from Linux and Windows without needing access to proprietary Apple software!

Other features include:

- Built-in support for using smart cards (e.g. YubiKeys) for signing and key/certificate management.

- A *remote signing* mode that enables you to delegate just the low-level cryptographic signature generation to a remote machine. This allows you to do things like have a CI job initiate signing but use a YubiKey on a remote machine to create cryptographic signatures. See *Remote Code Signing* for more.

- Certificate Signing Request (CSR) support to enable arbitrary private keys (including those generated on smart card devices) to be easily exchanged for Apple-issued code signing certificates.

- Support for dumping and diffing data structures related to code signatures.

- Awareness of Apple's public PKI infrastructure, including CA certificates and custom X.509 extensions and OIDs used by Apple.

- Documentation and code that are likely a treasure trove for others wanting to play with Apple code signing.

The canonical home of this project is https://github.com/indygreg/PyOxidizer/tree/main/apple-codesign. While this project is developed inside a larger monorepository, it is designed to be used as a standalone project.

Getting Started

Installing

To install the latest release version of the `rcodesign` executable using Cargo (Rust's package manager):

```
cargo install apple-codesign
```

To enable smart card integration:

```
cargo install --features smartcard apple-codesign
```

To compile and run from a Git checkout of its canonical repository (developer mode):

```
cargo run --bin rcodesign -- --help
```

To install from a Git checkout of its canonical repository:

```
cargo install --bin rcodesign
```

To install from the latest commit in the canonical Git repository:

```
cargo install --git https://github.com/indygreg/PyOxidizer --branch main rcodesign
```

Obtaining a Code Signing Certificate

Follow the instructions at *Managing Code Signing Certificates* to obtain a code signing certificate.

Installing Apple Transporter for Notarization

Notarization requires using Apple Transporter for uploading artifacts to Apple for notarization. This (Java) tool is distributed for macOS, Windows, and Linux.

You can install it by following Apple's instructions.

If you do not want to perform notarization, you do not need to install Apple Transporter.

Obtaining an Apple Connect API Key

To notarize and staple, you'll need an Apple Connect API Key to authenticate connections to Apple's servers.

You can generate one at https://appstoreconnect.apple.com/access/api.

This requires an Apple Developer account, which requires paying money. You may need to click around in the App Store Connect website to enable the API keys feature.

Apple Transporter looks in various locations for the API Key. Run `iTMSTransporter -help upload` and read the docs for the `-apiKey` argument.

We recommend putting the keys in `~/.appstoreconnect/private_keys/` because that is a descriptive directory name.

Using `rcodesign`

The `rcodesign` executable provided by this project provides a command mechanism to interact with Apple code signing.

Signing with `sign`

The `rcodesign sign` command can be used to sign a filesystem path.

Unless you want to create an ad-hoc signature on a Mach-O binary, you'll need to tell this command what code signing certificate to use.

To sign a Mach-O executable:

```
rcodesign sign \
  --p12-file developer-id.p12 --p12-password-file ~/.certificate-password \
  --code-signature-flags runtime \
  path/to/executable
```

To sign an `.app` bundle (and all Mach-O binaries inside):

```
rcodesign sign \
  --p12-file developer-id.p12 --p12-password-file ~/.certificate-password \
  path/to/My.app
```

To sign a DMG image:

> **rcodesign sign**
> –p12-file developer-id.p12 –p12-password-file ~/.certificate-password path/to/app.dmg

To sign a `.pkg` installer:

```
rcodesign sign \
 --p12-file developer-id-installer.p12 --p12-password-file ~/.certificate-password \
 path/to/installer.pkg
```

Notarizing and Stapling

You can notarize a signed asset via `rcodesign notarize`.

Notarization requires an Apple Connect API Key. See *Obtaining an Apple Connect API Key* for instructions on how to obtain one.

Notarization also requires Apple's Transporter tool. See *Installing Apple Transporter for Notarization* for more about Transporter. The `rcodesign find-transporter` command can be used to see if `rcodesign` can find Transporter.

You will need an API Key `AuthKey_<ID>.p8` file on disk in one of the default locations used by Apple Transporter. These are `$(pwd)/private_keys/`, `~/private_keys/`, `~/.private_keys/`, and `~/.appstoreconnect/private_keys/`.

You need to provide both the Key ID and IssuerID when invoking this command. Both can be found at https://appstoreconnect.apple.com/access/api.

To notarize an already signed asset:

```
rcodesign notarize \
  --api-issuer 68911d4c-110c-4172-b9f7-b7efa30f9680 \
  --api-key DEADBEEF \
  path/to/file/to/notarize
```

By default `notarize` just uploads the asset to Apple. To wait on its notarization result, add `--wait`:

```
rcodesign notarize \
  --api-issuer 68911d4c-110c-4172-b9f7-b7efa30f9680 \
  --api-key DEADBEEF \
  --wait \
  path/to/file/to/notarize
```

Or to wait and automatically staple the file if notarization was successful:

```
rcodesign notarize \
  --api-issuer 68911d4c-110c-4172-b9f7-b7efa30f9680 \
  --api-key DEADBEEF \
  --staple \
  path/to/file/to/notarize
```

If notarization is interrupted or was initiated on another machine and you just want to attempt to staple an asset that was already notarized, you can run `rcodesign staple`. e.g.:

```
rcodesign staple \
  --api-issuer 68911d4c-110c-4172-b9f7-b7efa30f9680 \
  --api-key DEADBEEF \
  path/to/file/to/staple
```

Managing Code Signing Certificates

In order to add cryptographic signatures using this tool, you'll need to use a *Code Signing Certificate*. (Follow the link for what that means.)

In order to perform code signing in a way that is recognized and trusted by Apple operating systems, you will need to obtain a code signing certificate that is signed/issued by Apple. This requires joining the Apple Developer Program, which has an annual membership fee.

Once you are a member, there are various ways to generate and manage your certificates. But first, a primer about flavors of Apple code signing certificates.

Apple Code Signing Certificate Flavors

Apple issues different types/flavors of code signing certificates. Each one is used to sign a different class of software.

If you are logged into your Apple Developer account, you can see Apple's description for these at https://developer.apple.com/account/resources/certificates/add. Here's our concise definitions:

Apple Development
> Sign applications for Apple operating systems that aren't distributed publicly.

Apple Distribution
> Sign applications for submission to the App Store or for Ad Hoc distribution.

iOS App Development
> Legacy version of *Apple Development* just for iOS apps. (We think.)

iOS Distribution
> Legacy version of *Apple Distribution* just for iOS apps. (We think.)

Mac Development
> Legacy version of *Apple Development* just for macOS apps. (We think.)

Mac App Distribution
> Sign macOS applications and configure a Distribution Provisioning Profile for distribution through Mac App Store.

Mac Installer Distribution
> Sign package installers (e.g. `.pkg` files) which will be distributed via the Mac App Store.

Developer ID Installer
> Sign package installers (e.g. `.pkg` files) which will be distributed outside the Mac App Store. i.e. if users fetch your installer via your website, you sign with this.

Developer ID Application
> Sign applications which will be distributed outside the Mac App Store. Used for signing Mach-O binaries, `.app` bundles, and `.dmg` files.

Essentially, if you are distributing macOS software to end-users via non-Apple channels like your website, you need *Developer ID Application* and/or *Developer ID Installer*.

If you are distributing via Apple's App stores, you need *Apple Distribution* or one of the other types having *Distribution* in the name.

Tip: The `rcodesign analyze-certificate` command can be used to print information about Apple code signing certificates. Look for a line with `Certificate Profile` in its output to see which flavor of certificate this software thinks it is.

Generating Certificates with Xcode

Using Xcode from macOS is probably the easiest way to create and manage your certificates as Xcode has built-in UI to facilitate this.

Apple keeps thorough documentation about how to do this. Please follow Apple's documentation to generate a certificate.

Obtaining a Certificate via a Certificate Signing Request

You can obtain a code signing certificate by uploading a *Certificate Signing Request (CSR)* to Apple. Essentially, you generate a CSR, send it to Apple, and Apple will issue a new code signing certificate which you can download.

A CSR is produced by creating a cryptographic signature (using a *private key*) over a small set of metadata describing the *private key* for which a certificate shall be issued.

In order to generate a CSR, you need a *private key*. As of April 2022, Apple appears to require the use of RSA 2048 private keys.

If you have access to macOS, the easiest way to generate a private key and CSR is to use `Keychain Access` using the procedure outlined here.

If you want to generate your own CSR using `rcodesign`, you can! First, you'll need a private key.

To generate an RSA 2048 private key using OpenSSL:

```
openssl genrsa -out private.pem 2048
```

> **Warning:** The RSA private key will be in plain text on your filesystem. This is not very secure!

Then once you have a private key, we can generate a CSR using `rcodesign`:

```
rcodesign generate-certificate-signing-request --pem-source private.pem
rcodesign generate-certificate-signing-request --p12-file key.p12

# Smart cards require generating a new key then creating a CSR from that key.
rcodesign smartcard-generate-key --smartcard-slot 9c
rcodesign generate-certificate-signing-request --smartcard-slot 9c
```

This command will print the CSR to stdout. e.g.:

```
-----BEGIN CERTIFICATE REQUEST-----
MIHeMIGDAgEAMCExHzAdBgNVBAMMFkFwcGxlIENvZGUgU2lnbmluZyBDU1IwWTAT
BgcqhkjOPQIBBggqhkjOPQMBBwNCAAQxluBlPIv/HgBDz0O3GLPhhna/NJU7menq
GzUc9sZFOgZ7XmpR9vQTxHPEyg5D6huBapVQZsDG9IgAXjvSOmimoAAwDAYIKoZI
zj0EAwIFAANIADBFAiEAoZpbfrlm7HgQXByfwuoPt7/V+QM7DCIILcTKCBrkIZUC
IEIp8yA9bSg7bM9XJl8bgFesTjermlSYQI/2JY834/z7
-----END CERTIFICATE REQUEST-----
```

You probably want to use `--csr-pem-path` to write that to a file automatically:

```
rcodesign generate-certificate-signing-request --smartcard-slot 9c --csr-pem-path csr.pem
```

Exchanging a CSR for a Code Signing Certificate

Once you have a CSR file, you can attempt to exchange it for a code signing certificate.

1. Go to https://developer.apple.com/account/resources/certificates/add (you must be logged into Apple's website)

2. Select the certificate *flavor* you want to issue.

3. Click `Continue` to advance to the next form.

4. Select the `G2 Sub-CA (Xcode 11.4.1 or later)` *Profile Type* (we support it).

5. Choose the file containing your CSR.

6. Click `Continue`.

7. If all goes according to plan, you should see a page saying `Download Your Certificate`.

8. Click the `Download` button.

9. Save the certificate somewhere. (The file content is likely not sensitive and doesn't need to be kept secret because this content will be copied to everything you sign with it!)

At this point, you have both a *private key* and a *public certificate*: you can sign Apple software!

Exporting a Code Signing Certificate to a File

`rcodesign` supports consuming code signing certificates from multiple sources, including hardware devices. But sometimes it is desirable to have your code signing certificate exist as a file.

Use the instructions in one of the following sections to export a code signing certificate.

> **Danger:** It is generally accepted that private keys stored in files are less secure than stored in special operating system enclaves like keychains. This is because the operating system has protections around accessing the private keys and these protections are often much stronger than those on a file on the filesystem.
>
> This tool has support for using certificates / keys directly from macOS keychains. So exporting to a file is not always necessary.

Using Keychain Access

(macOS)

1. Open the `Keychain Access` application.
2. Find the certificate you want to export and command click or right click on it.
3. Select the `Export` option.
4. Choose the `Personal Information Exchange (.p12)` format and select a file destination.
5. Enter a password used to protect the contents of the certificate.
6. If prompted to enter your system password to unlock your keychain, do so.

The exported certificate is in the PKCS#12 / PFX / p12 file format. Command arguments with these labels in the same can be used to interact with the exported certificate.

Using Xcode

(macOS)

See [Apple's Xcode documentation](#).

Using `security`

(macOS)

1. Run `security find-identity` to locate certificates available for export.
2. Run `security export -t identities -f pkcs12 -o keys.p12`

If you have multiple identifies (which is common), `security export` will export all of them. `security` doesn't seem to have a command to export just a single certificate pair. You will need to invoke some `openssl` command to extract just the certificate you care about. Please contribute back a fix for this documentation once you figure it out!

Using a Self-Signed Certificate

If you want to cut some corners and play around with certificates not signed by Apple, you can run `rcodesign generate-self-signed-certificate` to generate a self-signed code signing certificate.

This command will include special attributes in the certificate that indicate compatibility with Apple code signing. However, since the certificate isn't signed by Apple, its signatures won't confer the same trust that Apple signed certificates would.

These certificates can be useful for debugging and testing.

Smart Card Support

This project has some support for integrating with Smart Cards. This enables you to perform cryptographic signing using a certificate that is stored in a hardware device.

Certificates stored this way are more secure, as it typically requires that a physical device be unlocked in order to use the private key. And access to the raw private key matter is typically not allowed.

Cargo Feature

Smart card integration requires the optional and disabled-by-default `smartcard` Cargo feature to be enabled.

On macOS and Windows, this feature should *just work*.

On Linux, you'll need a package providing `pcsclite` installed or you may get a cryptic build error due to missing dependencies. On Debian based distros, you want to `apt install libpcsclite1 libpcsclite-dev` (or something of that nature).

Limitations

We currently use yubikey.rs for smart card integration. This likely means that only YubiKeys currently work.

However, we would like to switch to a more generic interface (such as pcsc in the future to allow more flexible usage.

There is currently no support for setting the management key. If you have set a custom management key, you won't be able to import certificates onto your smart card. However, signing should still work.

Validating Smart Card Integration

To see if your smart card device is recognized and certificates can be found:

```
rcodesign smartcard-scan
Device 0: Yubico YubiKey OTP+FIDO+CCID 0
Device 0: Serial: 12345678
Device 0: Version: 5.2.7
Device 0: Certificate in slot Signature / 9c
Subject CN:                 gps
Issuer CN:                  gps
Subject is Issuer?:         true
Team ID:                    <missing>
SHA-1 fingerprint:          c847e830c01845517d7e3775805ab56313aa11c8
SHA-256 fingerprint:        ␣
→7c0bc8fe1a2d7831ca0b0787dc6d5c28c6f562c2723a7eaaab42d39e7a3b7924
```

```
Signed by Apple?:           false
Guessed Certificate Profile: none
Is Apple Root CA?:          false
Is Apple Intermediate CA?:  false
Apple CA Extension:         none
Apple Extended Key Usage Purpose Extensions:
Apple Code Signing Extensions:
```

Pointing Commands at a Smart Card Certificate

`rcodesign` command that operate against certificates expose a `--smartcard-slot` argument to specify which smart-card slot to use.

Slot `9c` is the standard slot for holding certificates used for signing.

To sign with your smart card certificate at slot `9c`, do something like:

```
rcodesign sign \
    --smartcard-slot 9c \
    path/to/entity/to/sign
```

Smartcards often require a PIN on signing operations. You should be prompted for your PIN value if the signing operation is initially unauthenticated.

Importing Certificates Into a Smart Card

The `rcodesign smartcard-import` command can be used to import an existing code signing certificate into your smart card.

Let's assume you created an Apple code signing certificate and exported it to the file `developer_id.p12`. You can import this certificate by doing the following:

```
$ rcodesign smartcard-import \
    --smartcard-slot 9c \
    --p12-file developer_id.p12 --p12-password password

$ rcodesign smartcard-scan
Device 0: Yubico YubiKey OTP+FIDO+CCID 0
Device 0: Serial: 1234567
Device 0: Version: 5.2.7
Device 0: Certificate in slot Signature / 9c
Subject CN:                 Developer ID Application: Gregory Szorc (MK22MZP987)
Issuer CN:                  Developer ID Certification Authority
Subject is Issuer?:         false
Team ID:                    MK22MZP987
SHA-1 fingerprint:          44d7155bcabf3b9a9221b01b8e198040ae04e0ad
SHA-256 fingerprint:        ␣
→8f610de4caea4bc138e85b56726ed4d330f7464d99cfa5957568904b6a6375ec
Signed by Apple?:           true
Apple Issuing Chain:
  - Developer ID Certification Authority
```

```
  - Apple Root CA
  - Apple Root Certificate Authority
Guessed Certificate Profile: DeveloperIdApplication
Is Apple Root CA?:          false
Is Apple Intermediate CA?:  false
Apple CA Extension:         none
Apple Extended Key Usage Purpose Extensions:
  - 1.3.6.1.5.5.7.3.3 (CodeSigning)
Apple Code Signing Extensions:
  - 1.2.840.113635.100.6.1.33 (DeveloperIdDate)
  - 1.2.840.113635.100.6.1.13 (DeveloperIdApplication)
```

Creating a Certificate with a Private Key Exclusive to the Smart Card

It is possible to generate a private key directly on the smart card and create a code signing certificate derived from this private key.

Code signing certificates created this way are theoretically much more secure than other private key generation methods because most smart cards never allow the private key content to be exported/viewed. Assuming operations involving the private key are protected with the appropriate access protections (like pin or touch policies), compromise of the machine or even the smart key itself may not result in unwanted access to the private key.

To create a code signing certificate whose private key has never left the smart card device itself, do something like the following.

First, generate a new private key on the smart card:

```
rcodesign smartcard-generate-key --smartcard-slot 9c
```

Then create a certificate signing request (CSR):

```
rcodesign generate-certificate-signing-request \
    --smartcard-slot 9c \
    --csr-pem-path csr.pem
```

Then follow the instructions at *Exchanging a CSR for a Code Signing Certificate* to submit the CSR file to Apple and obtain a *public certificate*.

Finally, import the Apple-issued public certificate into the smart card:

```
rcodesign smartcard-import \
    --der-source developerID_application.cer \
    --smartcard-slot 9c
```

At this point, the smart card is ready to sign using an Apple issued certificate and the private key never has - and probably never will - leave the smart card itself.

Concepts

Code signing on Apple platforms is complex and has many parts. This document aims to shed some light on things.

Cryptographic Signatures

At the heart of code signing is the use of cryptographic signatures.

The Wikipedia article on digital signatures explains the concept in far more detail than we care to go into.

Essentially, mathematics is used to prove that an entity in possession of a secret *key* digitally attested to the existence of some *signed* entity.

More concretely, an X.509 code signing certificate can be proved to have signed some piece of software by inspecting the cryptographic signature it produced.

Apple's cryptographic signatures use RFC 5652 / Cryptographic Message Syntax (CMS) for representing signatures. This standardized format is used outside the Apple ecosystem and libraries and tools like OpenSSL are capable of interfacing with it.

Code Signing

Code signing (or just *signing*) is the mechanism of producing (and then attaching) a signature to some entity.

Typically signing entails producing a cryptographic signature using a code signing certificate. However, Mach-O files (the binary file format for Apple platforms) has a concept of *ad-hoc* signing where the binary has data structures describing the content of the binary but without the cryptographic signature present.

Notarization

Notarization is the term Apple gives to the process of uploading an asset to Apple for inspection.

In order to help safeguard and control their software ecosystems, Apple imposes requirements that applications and installers be inspected by Apple before they are allowed to run on Apple operating systems - either at all or without scary warning signs.

When you notarize software, you are essentially asking for Apple's blessing to distribute that software. If Apple's systems are appeased, they will issue a *notarization ticket*.

Notarization Ticket

A *notarization ticket* is a blob of data that essentially proves that Apple notarized a piece of software.

The exact format and content of *notarization tickets* is not well known. But they do contain some DER-encoded ASN.1 with data structures that common appear in X.509 certificates. All that matters is that Apple's operating systems know how to read and validate a notarization ticket.

Stapling

Stapling is the term Apple gives to the process of attaching a *notarization ticket* to some entity. It is literally just fetching a *notarization ticket* from Apple's servers and then making that ticket available on the entity that was notarized.

You can think of notarization and stapling as Apple-issued cryptographic signatures. It establishes a chain of trust between some entity to you that also had to be inspected by Apple first.

Mach-O Binaries

Mach-O is the binary executable file format used on Apple operating systems.

When you run an executable like `/usr/bin/zsh` on macOS, you are running a Mach-O file.

Mach-O binaries are either *thin* or *fat*. A *thin* Mach-O contains code for a single architecture, like x86-64 or aarch64 / arm64. A *fat* or *universal* binary contains code for multiple architectures. At run-time, the operating system will decide which one to execute.

Bundles

Bundles are a filesystem based mechanism for encapsulating code and resources.

On macOS, you commonly encounter bundles as `.app` and `.framework` directories in `/Applications` and `/System/Library/Frameworks`.

Bundles are essentially a well-defined set of files that the operating system knows how to interact with. For example, macOS knows that to execute an `.app` bundle it should look for a `Contents/Info.plist` to resolve basic application metadata, such as the name of the main binary for the bundle, which resides in `Contents/MacOS/` within the bundle.

DMGs / Disk Images

Apple Disk Images are a self-contained file format for holding filesystems. Think of DMGs as standalone hard drives that Apple operating systems can recognize.

DMGs are often used to distribute macOS applications.

XARs / Flat Packages / `.pkg` Installers

Flat packages is a mechanism for installing software.

They take the form of `.pkg` files, which are actually XAR archives (a tar-like format for storing content for multiple files within a single file).

Code Signing Certificate

A code signing certificate is used to produce cryptographic signatures over some signed entity.

A code signing certificate consists of a private/secret key (essentially a bunch of large numbers or parameters) and a public certificate which describes it.

Code signing certificates are X.509 certificates. X.509 certificates are the same technology used to secure communication with https:// websites. However, the certificates are used for signing content instead of encrypting it.

The X.509 public certificate contains a bunch of metadata describing the certificate. This includes the name of the person or entity it belongs to, a date range for when it is valid, and a cryptographic signature attesting to its origination.

Apple's operating systems look for special metadata on code signing certificates to authenticate and trust them. There are special properties on certificates indicating what Apple software distribution they are allowed to perform. For example, a `Developer ID Application` certificate is required for signed Mach-O binaries, bundles, and DMG files to be trusted and a `Developer ID Installer` certificate is required to sign `.pkg` installers in order for them to be trusted.

In addition, different Apple code signing certificates are cryptographically signed by different Apple Certificate Authorities (CAs).

Known Issues and Limitations

Apple code signing is complex. While this project strives to provide all the features and compatibility that Apple's official tooling provides, we won't always get it right. This document captures some of the areas where we know we fall short.

Bundle Handling in General

Bundle signing is complex for a few reasons:

- The types and layouts of bundles are highly varied. Application bundles. Frameworks. Kernel extensions. macOS flavored vs iOS flavored bundles. The list goes on.

- Bundles can be nested.

- Signatures in nested bundles often need to propagate to their parent bundle.

- Bundles encapsulate other signable entities, notably Mach-O binaries.

All this complexity means bundle signing is susceptible to a lot of subtle bugs and variation from how Apple's tooling does it.

If you find bugs in bundle signing or have suggestions for improving its ergonomics, please file a GitHub issue!

Cannot Sign File Contents of DMGs

We support signing DMGs. But we can't recursively inspect the files within DMGs and sign those. e.g. if a DMG contains a Mach-O binary, we can't sign that Mach-O by unpacking it from the DMG and writing a new DMG.

The reason we can't do this is because DMGs contain a nested filesystem (likely HFS+) and we don't (yet) have a cross-platform mechanism for reading and writing HFS+ filesystems.

On macOS, we could call out to `hdiutil` to mount a DMG to see its contents and again to create a new DMG. However, this isn't implemented because we don't perceive there to be value in it: if you have access to macOS you should probably just use Apple's official signing tooling!

There are open source libraries for reading and writing HFS+ filesystems. We could potentially integrate those to support reading and writing the contents of DMGs. We could also potentially leverage a pure Rust HFS+ implementation (this is a preferred solution).

DMG also supports multiple embedded filesystem types and it is possible we could leverage one that isn't HFS+ (or APFS) and produce working DMGs. This is an area we haven't yet explored.

If you want to distribute DMGs signed with this tool that themselves have signed files, you'll need to sign the files inside the DMG before the DMG is created. Then you'll need to create the DMG (using `hdiutil` or whatever tool you have access to) then feed that DMG into this tool for signing.

https://github.com/indygreg/PyOxidizer/issues/540 is our tracking issue for DMG writing support. If you have ideas, please comment there!

Cannot Recursively Sign Flat Packages (`.pkg` Installers)

Flat Packages (`.pkg` installers) are a complex file format.

We have support for signing `.pkg` installers by reading the files within a flat package. And we are capable of recursively extracting and signing the `.pkg` installers that themselves are often embedded in `.pkg` installers.

What we don't yet have support for is mutating the file content within flat packages / `.pkg` installers. This means we can't recursively sign nested `.pkg` installers or bundles or Mach-O binaries within.

The main blocker to implementing `.pkg` writing is support for reading and writing Apple's *Bill of Materials* file format. These are the `Bom` files within flat packages. The author of this project has an unpublished Rust crate to read and write bom files but he encountered issues getting it to write files that validate with Apple's implementation.

So if you want to sign `.pkg` files that themselves containable signable entities, you need to sign files going into the `.pkg` before creating the `.pkg`. Then you need to create the `.pkg` and invoke this tool to sign the `.pkg`. For installers that contained nested `.pkg` installers, this process will be quite tedious. Invoking `componentbuild` and `productbuild` will likely be much simpler.

https://github.com/indygreg/PyOxidizer/issues/541 is our tracking issue for flat packages writing support.

Extra Signing or Time-Stamp Token Operations

Signatures often need to encapsulate the size of the resulting signature. This creates a chicken-and-egg problem because how can we know the size of the resulting signature before we actually produce it!

In some cases, this tool will create a *fake* signature and obtain an actual time-stamp token from a server in order to resolve the size of the data so we can better estimate the size of the real signature.

We are not sure if Apple's tooling does this. But ours does and the extra operations can be annoying because they may require extra unlocks of signing keys or communications with a time-stamp token server.

We can likely eliminate the extra use of the signing key for generating these stand-in signatures and we can probably only make 1 request to the time-stamp token server to obtain the size of its signatures. But we haven't implemented this throughout the code base yet.

https://github.com/indygreg/PyOxidizer/issues/542 and https://github.com/indygreg/PyOxidizer/issues/543 track improvements here.

Long Tail of Random Discrepancies from Apple's Tooling

Apple's code signature format is really, really complex. There are tons of data structures and fields with complex values.

There is likely a long tail of minor differences in implementation that result in variations between the behavior of our implementation and Apple's.

In general, we consider differences in behavior in our implementation to be bugs worth filing. So please use `rcodesign diff-signatures` to report behavior differences!

Known areas where discrepancies are likely include:

- The *code requirements* expression embedded into Mach-O binaries. We attempt to derive one based on the signing key. The expression may not be exactly what Apple's tools derive automatically. We consider this a bug.

- Executable segment flags and code signing flags. The exact logic for determining what flags to set when is complex. In general, we consider differences in behavior here to be bugs.

- Size of embedded signatures. You often need to estimate the size of the produced embedded signature before signing because the signature encapsulates its own size. Our estimation method varies from Apple's and can result in signatures with more or less padded null bytes. This difference should be mostly harmless. Improvements to make our signatures use less wasteful extra padding are appreciated.

How to Debug and Report Problems

Apple code signing is complex and there will be cases where this tool behaves differently from Apple's, possibly to the point where Apple rejects the output of this tool.

Important: If Apple software rejects the output of this tool, we consider that a bug. We encourage end-users to report these bugs to the GitHub issue tracker.

Commands to Print Signature Info

The `rcodesign print-signature-info` command can be used to dump YAML describing any signable file entity. Just point it at a Mach-O, bundle, DMG, or `.pkg` installer and it will tell you what it knows about the entity.

The `rcodesign diff-signatures` command will internally execute `print-signature-info` against 2 paths and print the differences between them.

`rcodesign diff-signatures` is exceptionally useful at understanding differences in behavior between this tool and Apple's. If Apple is rejecting the output of this tool, comparing the output of the same operation with Apple's tooling against this tool's is a good way to find the source of the problem.

Reporting Actionable Bugs

Please include the following in bug reports to improve chances for action:

- The released version or Git commit that this tool was built from.

- The command line used.

- The full output of the command.

- The output of `rcodesign diff-signatures` comparing similar operations between Apple's tooling and ours.

- A copy of the entity you were attempting to sign.

- Text copy or screenshot of error from Apple tooling indicating what failed.

It is understandable that some people may not desire to file publish issue reports or submit a copy of their application to be seen by the world. If you send a polite email to gregory.szorc@gmail.com with `apple-codesign` or `rcodesign` in the subject line along with more private/sensitive details, support can be given over email.

Remote Code Signing

This project has support for *remote signing*. This is a feature where cryptographic signature operations (requiring access to the private key) are delegated to a remote machine.

From a high level, two machines establish a secure communications bridge with each other through a central server. The *initiating* machine starts signing operations like normal. But when it gets to an operation that requires producing a cryptographic signature, it sends an end-to-end encrypted message to the bound *signer* peer with the message to sign. The *signer* then uses its private key to create a signature, which it sends back to the *initiator*, who incorporates it into the code signature.

Remote signing is essentially peer-to-peer, not client-server. The central server exists for relaying encrypted messages between peers and not for performing signing operations itself. Each signing *session* is ephemeral and short-lived. Since the signing keys are offline by default and a human must take action to join a signing session and use the signing keys, remote signing is theoretically more secure than solutions like giving a (CI) machine unlimited access to a code signing certificate or HSM.

Remote signing is intended for use cases where the machine initiating signing must not or can not have access to the private key material or unlimited access to it. Popular scenarios include:

- CI environments where you don't want a CI worker to have unlimited access to the signing key because CI workers are notoriously difficult to secure. (If someone can run arbitrary jobs on your CI they can likely exfiltrate any CI secrets with ease.)

- When hardware security devices are used and machines initiating the signing don't have direct access to this device. Think a remote CI machine or coworker wanting to sign with a certificate in a YubiKey or HSM whose access is entrusted to a specific person (or group of people in the case of an HSM).

Important: This feature is considered alpha and will likely change in future versions.

Danger: The custom cryptosystem for remote signing has not yet undergone an audit. The end-to-end message encryption and tampering resistance claims we've made may be undermined by weaknesses in the design of the cryptosystem and its implementation and interaction in code.

In other words, use this feature at your own risk.

Issue 552 tracks performing an audit of this feature.

How It Works

A full overview of the protocol and cryptography involved is available at *Remote Code Signing Protocol* and you can read more about the design and security at *Remote Code Signing Design and Security Considerations*.

From a high-level, signing operations involve 2 parties:

- The *initiator* of the signing request. This is the entity that wants something to be signed but doesn't having the signing certificate / key.

- The *signer*. This is the entity who has access to the private signing key.

The signing procedure is essentially:

1. *Initiator* opens a persistent websocket to a central server and publishes details about that session and how to connect to it.

2. *Signer* follows the instructions from *initiator* and joins the *signing session* by opening a websocket to the same server as the *initiator*. Cryptography is employed to derive encryption keys so all subsequently exchanged messages are end-to-end encrypted, preventing the server or any privileged network actors from eavesdropping on signing operations or forging a signing request.

3. *Initiator* sends a request to *signer* asking them to sign a message.

4. *Signer* inspects the request and issues a cryptographic signature, which it sends back to *initiator*.

5. Steps 3-4 are repeated as long as necessary.

Using

The *initiator* begins a remote signing *session* via `rcodesign sign --remote-signer`. (Some additional arguments are required - see below.)

This command will print out an `rcodesign` command that the *signer* must subsequently run to *join* the signing session. e.g.:

```
$ rcodesign sign --remote-signer --remote-shared-secret-env SHARED_SECRET
...
connecting to wss://ws.codesign.gregoryszorc.com/
session successfully created on server
Run the following command to join this signing session:

rcodesign remote-sign gm1zaGFyZWRzZWNyZXQwg...

(waiting for remote signer to join)
```

At this point, that long opaque string - which we call a *session join string* - needs to be copied or entered on the *signer*. e.g.:

```
$ rcodesign remote-sign --p12-file developer_id.p12 --remote-shared-secret-env SHARED_
→SECRET \
   gm1zaGFyZWRzZWNyZXQwg...
```

If everything goes according to plan, the 2 processes will communicate with each other and *initiator* will delegate all of its signing operations to *signer*, who will issue cryptographic signatures which it sends back to the *initiator*.

Session Agreement

Remote signing currently requires that the *initiator* and *signer* exchange and agree about *something* before signing operations. This ahead-of-time exchange improves the security of signing operations by helping to prevent signers from creating unwanted signatures.

The sections below detail the different types of agreement and how they are used.

Public Key Agreement

Important: This is the most secure and preferred method to use.

In this operating mode, the *signer* possesses a private key that can decrypt messages. When the *initiator* begins a signing operation, it encrypts a message that only the *signer*'s private key can decrypt. This encrypted message is encapsulated in the *session join string* exchanged between the *initiator* and *signer*.

This mode can be activated by passing one of the following arguments defining the public key:

`--remote-public-key`
> Accepts base64 encoded public key data.
>
> Specifically, the value is the DER encoded SubjectPublicKeyInfo (SPKI) data structure defined by RFC 5280.

`--remote-public-key-pem-file`
> The path to a file containing the PEM encoded public key data.
>
> The file can begin with `-----BEGIN PUBLIC KEY-----` or `-----BEGIN CERTIFICATE-----`. The former defines just the SPKI data structure. The latter an X.509 certificate (which has the SPKI data inside of it).

Both the public key and certificate data can be obtained by running the `rcodesign analyze-certificate` command against a (code signing) certificate.

The *signer* needs to use the corresponding private key specified by the *initiator* in order to join the signing session. By default, `rcodesign remote-sign` attempts to use the in-use code signing certificate for decryption.

So, an end-to-end workflow might look like the following:

1. Run `rcodesign analyze-certificate` and locate the `-----BEGIN PUBLIC KEY-----` block.

2. Save this to a file, `signing_public_key.pem`. You can check this file into source control - the contents aren't secret.

3. On the initiator, run `rcodesign sign --remote-signer --remote-public-key-pem-file signing_public_key.pem /path/to/input /path/to/output`.

4. On the signer, run `rcodesign remote-sign --smartcard-slot 9c ``<session join string>`.

We believe this method to be the most secure for establishing sessions because:

- The state required to bootstrap the secure session is encrypted and can only be decrypted by the private key it is encrypted for. If you are practicing proper key management, there is exactly 1 copy of the private key and access to the private key is limited. This means you need access to the private key in order to compromise the security of the signing session.

- The session ID is encrypted and can't be discovered if the session join string is observed. This eliminates a denial of service vector.

Shared Secret Agreement

Important: This method is less secure than the preferred *public key agreement* method.

In this operating mode, *initiator* and *signer* agree on some shared secret value. A password, passphrase, or some random value, such as a type 4 UUID.

This mode is activated by passing one of the following arguments defining the shared secret:

`--remote-shared-secret-env`
> Defines the environment variable holding the value of a shared secret.

`--remote-shared-secret`
> Accepts the raw shared secret string.
>
> This method is not very secure since the secret value is captured in plain text in process arguments!

An end-to-end workflow might look like the following:

1. A secure, random password is generated using a password manager.

2. The secret value is stored in a password manager, registered as a CI secret, etc.

3. The initiator runs `rcodesign sign --remote-signer --remote-shared-secret-env REMOTE_SIGNING_SECRET /path/to/input /path/to/output`.

4. The signer runs `rcodesign remote-sign --remote-shared-secret-env REMOTE_SIGNING_SECRET --smartcard-slot 9c`.

Important security considerations:

- Anybody who obtains the shared password could coerce the signer into signing unwanted content.

- Weak password will undermine guarantees of secure message exchange and could make it easier to decrypt or forge communications.

Because the password exists in multiple locations, must be known by both parties, and the process for generating it are not well defined, the overall security of this solution is not as strong as the preferred *public key agreement* method. However, this method is easier to use and may be preferred by some users.

Using with GitHub Actions

It is pretty simple to initiate remote code signing from GitHub Actions! In fact, this scenario is one of the primary use cases for the design of the feature.

Note: Issue #553 tracks publishing a canonical GitHub Action that formalizes the steps in this documentation. Assistance in building that would be greatly appreciated!

Here are the general steps.

Configuring a Workflow / Actions

First, export the public key data of the signing certificate to a file checked into source control. Use `rcodesign analyze-certificate` and copy the `-----BEGIN PUBLIC KEY----` block to a file in your repository. e.g. https://github.com/indygreg/PyOxidizer/blob/main/ci/developer-id-application.pem defines the `Developer ID Application` public key data for the maintainer of this project.

Note: The public key data is included in the code signatures embedded in signed artifacts so there is generally not a concern with making the public key data widely available in the repository.

Next, create a GitHub workflow or action that invokes `rcodesign sign`. https://github.com/indygreg/PyOxidizer/blob/main/.github/workflows/sign-apple-exe.yml is an example of such a workflow. This particular workflow is using `on.workflow_dispatch` so the workflow is only triggered manually. See the workflow_dispatch documentation and Manually running a workflow docs for more.

Important: A manually triggered workflow is strongly recommended because a signer must take manual action to perform remote signing and an automated trigger will likely hang unless a person is around to attend to it.

Important: For security reasons, you should set `timeout-minutes` on either the job or step initiating remote signing to limit how long a signer will wait.

The important steps in a remote signing action/workflow are:

1. Securely obtain `rcodesign`. We recommend downloading a release artifact from https://github.com/indygreg/PyOxidizer/releases and pinning/verifying the SHA-256 digest on download.

2. Download the artifact you want signed. The Download workflow artifact action can be useful for downloading artifacts from other workflows in the current repository (since the official `download-artifact` action limits you to artifacts in the current workflow).

3. Invoke `rcodesign sign --remote-signer --remote-public-key-pem-file path/to/public_key.pem`.

4. Do something with the signed result (like upload it as an artifact).

Running the Workflow / Action

Now that you have a GitHub workflow or action in place, here's how you use it.

If you followed the recommendations from above, the workflow is manually triggered via `on.workflow_dispatch`. You can trigger the workflow via the GitHub web UI or via API. For API, the path of least resistance is likely the `gh` GitHub CLI tool. e.g.:

```
gh workflow run sign-apple-exe.yml \
  --ref ci-main \
  -f workflow=rcodesign.yml \
  -f run_id=2214520041 \
  -f artifact=exe-rcodesign-macos-universal \
  -f exe_name=rcodesign
```

If your workflow is highly parameterized (like this one), you may want to script its invocation to make it more turnkey.

When `rcodesign sign --remote-signer` runs in GitHub Actions, it will print instructions on how to join the signing session. You will need to follow these instructions in a timely manner to complete the code signing operation.

Here is what you are looking for in the job output:

> ✅ Run actions/checkout@v2

> ✅ Download rcodesign Linux executable

> ✅ Download artifact to sign

∨ ✅ Perform remote code signing

```
 1  ▶ Run chmod +x bins/rcodesign
11  connecting to wss://ws.codesign.gregoryszorc.com/
12  session successfully created on server
13
14  Run the following command to join this signing session:
15
16      rcodesign remote-sign gmpwdWJsaWNrZXkwg1kBAD1edBc6eAcI8kkr7CSR
    yDVlFhc6ZRaNZtckR440pFIAhFJhRjGWbBfp6iJpXDRsvVGP31oZ5EDf_X4YqqtSSs
    AQC3YIHtD2gEOkDo-R3iykYJYX4ts6clQaRQ94Kft-si_wJPoczFoITwqiRO42jydWd
    m6W9P_6CoBL2qedh3a6Bsc22MisPVgrkkXmiLY148NLfu4XhCa4RcYCHd4OUhEOjMCd
    RWTccgDmKTxqHar5R31dp-KnJOPRZpce9ZuUpjjBTixZmsHz-gDvjcPz5LkUw62ZC1d
17
18  Or if this output is too long, paste the following output:
19
20  -----BEGIN SESSION JOIN STRING-----
21  gmpwdWJsaWNrZXkwg1kBAD1edBc6eAcI8kkr7CSR+34Kg0ubiw3bkW/bmef3pZqe
22  iYRmgvGRfv28x1p35tpjossoxxRcyjwyoWXk012d7CxiiZMw5l/PmaIn2Zl3ub91
23  dIwltbpGo1SPpWuGC+SVr1xiRK39e91SdnEN+yDVlFhc6ZRaNZtckR440pFIAhFJ
24  hRjGWbBfp6iJpXDRsvVGP31oZ5EDf/X4YqqtSSsfdqrERj9F6bmaR4YSc0j61sJT
25  RLv2XHlou1EA5MCm2qvKBZuPU3DHcv3ZShf7FHUa2IbePlawJjwFKE7BrnccCvDN
26  e/7JJwRTkgONsVXtApx42Sw8XE6B7fryZvIxlJrzy45ZASYwggEiMA0GCSqGSIb3
27  DQEBAQUAA4IBDwAwggEKAoIBAQC3YIHtD2gEOkDo+R3iykYJYX4ts6clQaRQ94Kf
28  t+si/wJPoczFoITwqiRO42jydWo1b05zL+gUc7ZKInQfYQwWBpYhWB+Im5kogzDM
29  PMgS2BXjxUT1X70BFojkk4Wb5u6K8C2UJ1PO1QBKxwipVvXTQurJ/ygB3rv1vLCz
30  ETKKBVPEAVD6ocV/KTT8Pxny2Dt76iyhcAWgf8Tp90ppQy8RTCgy+m6W9P/6CoBL
31  2qedh3a6Bsc22MisPVgrkkXmiLY148NLfu4XhCa4RcYCHd4OUhEOjMCoLQ56DTM9
32  W4UEVnLWJXLg1Lxphj02JoofcQwNWTPPTvvOfD9wmn37AdlZAgMBAAFY24/HV0CB
33  gP1++soH+cU82atz2H+Ug9NQ9sGvI7JgNJgF0bPNbCsSbM7vUKfn2gKT2bNbMgKR
34  9uWTMdkwIxKnbsWaE42kWbYDt2u+RWTccgDmKTxqHar5R31dp+KnJOPRZpce9ZuU
35  pjjBTixZmsHz+gDvjcPz5LkUw62ZC1qorpRbBGBa2Gv4W0IISqynh6XmQRXU2SPQ
36  kat1dChaA8ykMs4BEwiqFYPiUBcbviaVkdfBUv6KHfrnWLC/IfuwaEm+EmIa+jAV
37  wmF2ntvYAbENpQzCnsxclhtgv4H7Jv//
38  -----END SESSION JOIN STRING-----
39
40  Into an interactive editor using:
41
42      rcodesign remote-sign --editor
43
44  Or into a new file whose path you define with:
45
46      rcodesign remote-sign --sjs-path /path/to/file/you/just/saved
47
48  (waiting for remote signer to join)
```

Then, simply follow instructions on the machine with the signing key to commence signing!

Important: When you view the logs of a running GitHub Actions job, only the output from after the point you started viewing them is visible. This means that if you are *too late* you may not see the printed instructions for joining the signing session!

There are definitely some mitigations we can take for this. For the moment, you need to be quick to open the job output in your browser. Or you can do things like add a `sleep` before running `rcodesign sign`.

If all goes according to plan, you should see progress being printed both in the signing process and from the near real time output from GitHub Actions.

Here is the output from the GitHub Actions (Linux) machine:

```
48   (waiting for remote signer to join)
49   signer joined session; deriving shared encryption key
50   requesting signing certificate info from signer
51   remote signer will sign with certificate: Developer ID Application: Gregory Szorc (MK22MZP
52   registering signing key
53   automatically registered Apple CA certificate: Developer ID Certification Authority
54   automatically registered Apple CA certificate: Apple Root CA
55   using time-stamp protocol server http://timestamp.apple.com/ts01
56   automatically setting team ID from signing certificate: MK22MZP987
57   registering extra X.509 certificate
58   registering extra X.509 certificate
59   signing dist/input/rcodesign to dist/output/rcodesign
60   signing dist/input/rcodesign as a Mach-O binary
61   inferring default signing settings from Mach-O binary
62   preserving existing binary identifier in Mach-O
63   using team ID from settings
64   preserving code signature flags in existing Mach-O signature
65   setting binary identifier to rcodesign
66   parsing Mach-O
67   writing dist/output/rcodesign
68   signing Mach-O binary at index 0
69   attempting to derive code requirements from signing certificate
70   code requirements: 0: (identifier "rcodesign") and ((anchor apple generic) and ((certifica
     leaf[subject.OU] = "MK22MZP987"))));
71   binary targets macOS >= 11.0.0 with SDK 12.1.0
72   code directory version: 132096
73   creating cryptographic signature with certificate Developer ID Application: Gregory Szorc
74   Using time-stamp server http://timestamp.apple.com/ts01
75   sending signing request to remote signer
76   received signature from remote signer
77   total signature size: 186965 bytes
78   signing Mach-O binary at index 1
79   attempting to derive code requirements from signing certificate
80   code requirements: 0: (identifier "rcodesign-8d3eb894c5473a86") and ((anchor apple generic
     (certificate leaf[subject.OU] = "MK22MZP987"))));
81   binary targets macOS >= 11.0.0 with SDK 12.1.0
82   adding code signature flags from signing settings: ADHOC | LINKER_SIGNED
83   removing ad-hoc code signature flag
84   removing linker signed flag from code signature (we're not a linker)
85   code directory version: 132096
86   creating cryptographic signature with certificate Developer ID Application: Gregory Szorc
87   Using time-stamp server http://timestamp.apple.com/ts01
88   sending signing request to remote signer
89   received signature from remote signer
90   total signature size: 174199 bytes
91   terminating signing session on relay
92   relay server confirmed session termination
93   disconnecting from relay server
```

> ✓ Upload

And from the signing Windows machine using a YubiKey for signing:

```
connected to reader: Yubico YubiKey OTP+FIDO+CCID 0
using certificate in smartcard slot 9c
attempt 1/3
failed sign command with code 6982
device refused operation due to authentication error
Please enter device PIN: [hidden]
pin verification successful
attempt 2/3
connecting to wss://ws.codesign.gregoryszorc.com/
joining session...
successfully joined signing session 69bab200-128d-4f6e-a4d1-af609844910f
verifying encrypted communications with peer
waiting for server to send us a message...
waiting for server to send us a message...
creating signature for remote message: MYIB1DAYBgkqhkiG9w0BCQMxCwYJKoZIhvcNA
DYXJlbj9a17ujVhmFwvv/oTODAcBgkqhkiG9w0BCQUxDxcNMjIwNDI0MTg0OTE4WjCCASkGCSqGSI
iBlbmNvZGluZz0iVVRGLTgiPz4KPCFET0NUWVBFIHBsaXN0IFBVQkxJQyAiLS8vQXBwbGUvL0RUR
S5jb20vRFREEcy9Qcm9wZXJ0eUxpc3QtMS4wLmR0ZCI+CjxwbGlzdCB2ZXJzaW9uPSIxLjAiPgo8Z
T4KCQk8ZGF0YT4KCQk3WTVQQVllWVWRHhqREdtWUV3MkZ5Wlc0L1U9CgkJPC9kYXRhPgoJPC9hc
AkCMS8wLQYJYIZIAWUDBAIBBCDtjk9VivJ5UPGMMaZgTDYXJlbj9a17ujVhmFwvv/oTOA==
initial signing attempt may fail if the certificate requires a pin to unlock
attempt 1/3
failed sign command with code 6982
device refused operation due to authentication error
Please enter device PIN: [hidden]
pin verification successful
attempt 2/3
sending signature to peer
relay acknowledged signature message received
```

Remote Code Signing Protocol

Overview

The remote signing protocol facilitates the cryptographic signing of messages involving 2 discrete network peers.

The peer that wants something signed is the **initiator**.

The peer with access to the signing key that produces cryptographic signatures is the **signer**.

Peers establish persistent websocket connections to a central server to enable them to speak with each through firewalls and NATs.

Peers register an ephemeral *session* with the server, which is essentially a binding between 2 connected websocket clients.

Peers derive session-specific encryption keys using mutually agreed upon ahead of time data. They then relay end-to-end encrypted messages through the central server and perform cryptographic signing operations.

Wire Protocol

The protocol entails the exchange of JSON encoded objects via websockets.

The JSON objects sent from clients to the server have the following keys:

request_id
> (string) (required) A unique identifier for this request.

api
> (string) (required) The name of the API / method to invoke on the server.

payload
> (object) (optional) Parameters passed to this API invocation.

The JSON objects sent from servers to clients have the following keys:

request_id
> (string) (optional) Echo of `request_id` from the message that generated this one. The value could be unknown to the receiver if this message was generated from the other peer in the session.

type
> (string) (required) The message type.

ttl
> (number) (optional) Integer number of seconds remaining before the session expires and will be automatically deleted by the server.

payload
> (object) (optional) Payload further describing this message.

All other fields in the top-level object are reserved for future use.

Messages sent from the client to server ALWAYS result in the server responding to that API request.

It is also possible for servers to send messages to clients asynchronously of any client-initiated message.

Initial Connection Protocol

When a client connects to the server, it SHOULD issue a `hello` API message and wait for the server's response.

If the response contains a *message of the day* string, it MUST be displayed to the end-user.

Clients SHOULD also make a best effort attempt to validate the server's advertised capabilities and make a determination about compatibility and error or print warnings if incompatibility is detected.

Session Negotiation

The *initiator* and *signer* pair with each other by forming a *session*.

From the server's perspective, a *session* is an opaque identifier string with associated state, such as the unique websocket connection IDs of the *initiator* and *signer* clients.

Sessions are ephemeral and expire automatically after a duration specified by the initiating client. (The server can impose a maximum duration to prevent service abuse.)

Sessions are generally created by the *initiator*.

The *initiator* creates a unique session ID, `SessionId`. `SessionId` MUST be randomly chosen. It SHOULD have sufficient entropy to prevent server-side collisions. The use of type 4 UUIDs for session IDs is recommended.

Once a server-side session is created, the *initiator* then shares a *session join string* with the signer via an out-of-band mechanism. See *Session Join Strings* for more.

At this point, mechanisms diverge based on the session joining mechanism employed. But generally speaking, the *signer* sends a *join-session* to the server to register itself as the other peer in the session. At this point, both peers derive encryption keys and communicate with each other by issuing *send-message* messages. See *Signing Protocol* for more.

Session Join Strings

The *initiator* and *signer* need to leverage an out-of-band mechanism for communicating metadata with each other in order to join a server-established session. There are various potential solutions for this and we've purposefully designed the mechanism to be extensible.

Generically, the mechanism to join a session is expressed through a **session join string**, or SJS.

The SJS is ultimately a CBOR encoded array of length 2. The array's elements are:

- (string) The scheme being used.
- (varied) The payload for that scheme.

But to end-users it is an opaque string.

The SJS can be encoded as:

- Base64 using the RFC 3548 *URL safe* character set with optional = padding.
- PEM using `SESSION JOIN STRING` as the armoring tag.

In general, the *session join string* is shared out-of-band with the other peer, who uses it to join the session.

In general, *session join strings* are designed such that a 3rd party becoming aware of the SJS will not jeopardize the security of the current or future signing operations. However, denial of service could occur if the SJS exposes the session ID and a 3rd party joins the session before the *intended* peer.

The following sections denote the defined *session join string* schemes. Sections names are the `scheme` value.

`publickey0`

The `publickey0` session joining mechanism relies on public key cryptography to authenticate the 2nd peer in a session by leveraging knowledge of the 2nd peer's public encryption key.

The initiating peer, `A`, MUST know the public key of the joining peer, `B`.

`A` generates a random value at least 32 bytes long, `ChallengeSecret`.

`A` generates a new RFC 7748 Curve 25519 private key. Its private / public components are `AAgreementPrivate` and `AAgreementPublic`, respectively.

`A` generates a new random 16 byte value, `SharedAESKey`.

`A` loads the public key of `B`, `BPublic`. It usually does so by extracting the X.509 SubjectPublicKeyInfo (SPKI) (RFC 5280 Section 4.1.2.7) from an X.509 certificate or DER/PEM fragment of just the SPKI.

`A` prepares a plaintext message to be sent to `B`, `AJoinPlaintext`. This message is a CBOR array with the following elements:

serverUrl
    (Index 0) (optional string) URL of the server to connect to.

sessionId
    (Index 1) (string) The session identifier created on the server.

---

challenge

(Index 2) (bytes) The content of `ChallengeSecret`.

agreementPublic

(Index 3) (bytes) `SubjectPublicKeyInfo` for `AAgreementPublic`.

A encrypts `AJoinPlaintext` using AES-128 in GCM with `SharedAESKey`, yielding `AJoinCiphertext`. A 12 byte nonce is used where the bytes are all `0x42`. The 16 byte authentication tag is appended to the raw ciphertext and constitutes the final bytes of `AJoinCiphertext`.

A encrypts `SharedAESKey` using asymmetric encryption targeting `BPublic`, yielding `SharedAESCiphertext`.

For RSA, OAEP padding with SHA-256 digests MUST be used.

The payload of the *session join string* is a CBOR array with the following elements:

aes_ciphertext

(Index 0) (bytes) The `SharedAESCiphertext` generated above.

bPublic

(Index 1) (bytes) The SPKI describing which public key was used to encrypt `SharedAESCiphertext`.

message_ciphertext

(Index 2) (bytes) The `AJoinCiphertext` generated above.

So, the final *session join string* is `["publickey0", [SharedAESCiphertext, BSPKI, AJoinCiphertext]]`.

The *session join string* is summarily CBOR and base64 encoded and made available to B.

B receives and decodes the SJS.

B locates the decryption key from the provided SPKI structure. (B may want to impose restrictions here to prevent clients from fishing for specific keys.)

B decrypts `SharedAESCiphertext` using `BPrivate`, yielding back `SharedAESKey`.

Using `SharedAESKey`, B verifies and decrypts `AJoinCiphertext`, yielding `AJoinPlaintext`.

On success, B generates a new RFC 7748 Curve 25519 private key, `BAgreementPrivate` and `BAgreementPublic`.

B connects to the server and sends a *join-session* message with `context` set to `BAgreementPublic`.

At this point, `A` and B both perform key agreement using their ephemeral ED25519 private key and the public key of the other peer, each mutually deriving `SessionSharedKey`.

At this point, the procedure described in *AEAD Key Derivation* is used to derive new symmetric encryption keys. `ChallengeSecret` is used as the additional value to derive `IdentifierA` and `IdentifierB`.

Security Considerations

The *session join string* consists of 2 discrete encrypted payloads and is generally safe against offline attacks. Unless ciphers are broken, the private key is required to obtain for anything beyond side-channels (like total payload size).

`SessionId` is encrypted, so compromise of the SJS can't easily lead to a DoS by an unwanted peer joining the session.

The server doesn't see anything: the encrypted AES key and AES encrypted peer metadata are both encapsulated in the SJS. We could potentially move some of these to the server to reduce the length of the SJS.

Open Questions for Security Audit

- We don't sign / HMAC the asymmetrically encrypted AES key. Nor do we include an IV or other prepended message. This seems to go against best practices. Does it matter? Does the additional layer of AEAD feeding into the key agreement compensate for this?

- Is the use of a constant nonce for the `SharedAES` -> `AJoinCiphertext` acceptable? The AES key is randomly generated and is used exactly once, so do the nonces even matter?

- Is AES-128 in GCM mode a sufficient key/cipher for encrypting the main message?

- We currently generate 2 distinct private keys: 1 for key agreement and 1 for AES encryption. They are generated independently. Does this make sense or should perhaps HKDF be used against a common key?

- Right now there is no explicit trust anchoring between the asymmetric encryption targeting B and the derived shared secret key. Should B produce a cryptographic signature using `BPrivate` so A doesn't assume that *ability to decrypt* authenticates B? Or is *ability to decrypt* along with the assumption that only B possesses `agreementPublic` sufficient?

sharedsecret0

The `sharedsecret0` session joining mechanism uses SPAKE2 to derive a shared encryption key using an ahead-of-time mutually agreed upon shared secret, `SharedSecret`.

The peer creating the session, henceforth A, generates unique/random `SessionId` and `Identifier` values. These values are used to construct the SPAKE2 identifier strings: `A:{SessionId}:{Identifier}` and `B:{SessionId}:{Identifier}`.

A begins SPAKE2 role A initialization using `SharedSecret` and role A's identifier string. This produces `SpakeAInit`.

A calls *create-session* to register the new session with the server. Its `context` field is empty.

The *session join string* value is a CBOR array with the following elements:

sessionId
> (Index 0) (string) The session identifier string.

identifier
> (Index 1) (bytes) The random `Identifier` value produced earlier.

spakeAInit
> (Index 2) (bytes) The SPAKE2 Role A initialization message.

The final CBOR *session join string* is `["sharedsecret0", [SessionId, Identifier, SpakeAInit]]`.

The *session join string* is summarily CBOR and base64 encoded and made available to B.

B receives and decodes the SJS.

B performs SPAKE2 Role B initialization, producing `SpakeBInit`.

B sends a *join-session* message to the server with `context` set to the base64 encoding of `SpakeBInit`. `SpakeBInit` is relayed to A via the server.

At this point, both A and B are able to finalize SPAKE2 using `SpakeBInit` and `SpakeAInit`, respectively. They should mutually derive a shared encryption key, `SessionSharedKey`.

At this point, the procedure described in *AEAD Key Derivation* is used to derive new symmetric encryption keys. `Identifier` is used as the additional value used to derive `IdentifierA` and `IdentifierB`.

Security Considerations

The *session join string* containing the plaintext `SessionId`, `Identifier`, and `SpakeAInit` generally does not need to be highly secure or made secret.

`SharedSecret` cannot be derived from knowledge of the *session join string*.

The server does not directly observe the value for `Identifier`, only `SpakeBInit`. So it would need knowledge of the *session join string* and `SharedSecret` to decrypt messages.

A 3rd party in a privileged network position (including the server) with knowledge of `SharedSecret`, `SessionId`, and `Identifier` would be able to decrypt and forge messages, as it would be able to derive `RoleAKey` and `RoleBKey`. So it is important to use transport-level encryption, a trusted server, and keep `SharedSecret` a secret value.

Open Questions for Security Audit

- Is SPAKE2 the best mechanism for deriving session encryption keys from a shared secret?

- Should `SpakeAInit` be in the *session join string* or stored on the server and hidden from plaintext view? What are the tradeoffs with each approach?

- As proposed, the SPAKE2 identifier contains `SessionId` and yet another random value. That random value is not sent to the server but is possibly world readable in the *session join string*. Is this second source of entropy necessary? Does attempting to prevent the server from having access to it buy us any security value? Or is just the client-chosen `SessionId` string good enough?

- The SPAKE2 specification seems to insist on the use of key confirmation messages. Since we're using HKDF into AEAD, which has built-in authentication, do we need to perform the SPAKE2 key confirmation since any failures in SPAKE2 land would lead to AEAD failures anyway?

- How sensitive is SPAKE2 to the entropy of `SharedSecret`? While we want to encourage a relatively strong `SharedSecret`, we can't guarantee this. Should we be doing e.g. PBKDF2 on `SharedSecret` before feeding it into SPAKE2 or will SPAKE2 do sufficient *key stretching* on its own?

AEAD Key Derivation

The schemes above commonly detail the steps to enable 2 peers to mutually derive a session-ephemeral shared encryption key, `SessionSharedKey`.

Rather than use `SessionSharedKey` directly for subsequent message exchange, we instead derive additional keys from it for use with Authenticated Encryption and Additional Data (AEAD) encryption / message exchange.

An identifier value is associated with peers assuming roles `A` (the session initiator) and `B` (the session joiner). The value is a bytes concatenation of:

- The role name. e.g. `A` / `0x41` or `B` / `0x42`.

- A colon (`:` / `0x3a`)

- The `SessionId` identifier, UTF-8 encoded.

- A colon (`:` / `0x3a`)

- An additional value communicated in the session join string. e.g. `ChallengeSecret`.

These values are known as `IdentifierA` and `IdentifierB`.

HKDF is used to derive new keys.

Step 1 / HKDF-Extract uses an empty salt and `SessionSharedKey` to produce a pseudorandom key, PRK.

Step 2 / HKDF-Expand is performed twice to derive 2 new keys. The first invocation uses `IdentifierA` for `info` and `32` for L, producing `RoleAKey`. The second invocation uses `IdentifierB` for `info` and `32` for L, producing `RoleBKey`.

`RoleAKey` and `RoleBKey` are used to empower AEAD encryption / message exchange. ChaCha20+Poly1305 is used. Nonces are 12 bytes where the first 4 bytes are a little-endian u32 counter whose initial used value is `0` and the subsequent 8 bytes are always `0`. Additionally authenticated data (`AAD`) is generally not used.

`RoleAKey` is used by `A` to encrypt messages and by `B` to verify/decrypt messages from `A`. `RoleBKey` is used by `B` to encrypt messages and by `A` to verify/decrypt messages from `B`.

Open Questions for Security Audit

- Is ChaCha20+Poly1305 a reasonable cipher choice? Or should we be using block ciphers (e.g. AES)?

- Using a simple, easily guessable counter for nonces seems wrong. Using a random value seems more appropriate. But both parties need to know what the nonce we be. Do we use a random value for the nonce but encode the nonce in plaintext next to the exchanged ciphertext messages? Or do we need something else entirely?

- We could potentially use additionally authenticated data (AAD) to encapsulate more details of the request, such as the request ID. Does that buy us security benefits?

Signing Protocol

Once 2 peers have established a session and derived encryption keys to facilitate end-to-end encrypted communication, they communicate with each other using *peer to peer messages* by invoking the *send-message* API.

This process generally involves a handshake:

1. Both peers simultaneously send *ping* messages.

2. Upon receipt, each peer sends a *pong* in response. This dance confirms peer presence and that the derived encryption keys work.

3. The *initiator* sends a *request-signing-certificate* to request information about the signer's public certificate. This is necessary in order to allow the signer to do things like estimate the sizes of signatures and to derive additional details needed for signing.

4. The *signer* sends a *signing-certificate* in response.

At this point, both peers are ready to commence signing.

5. The *initiator* sends a *sign-request*.

6. The *signer* receives the request, assesses it, creates a cryptographic signature, and sends a *signature* in reply.

7. Steps 5-6 are repeated as necessary.

Finally,

8. Either peer sends a *goodbye* to finalize the session.

Client Issued Messages

The following sections denote the types of messages issued from clients to servers.

Section names denote the value of the `api` key in the messages.

hello

Greets the server and obtains information about the server.

This message type has no payload.

Servers respond to this message with a *error*.

create-session

Requests the creation of a new session on the server.

Sent by the *initiator* as part of session negotiation.

Fields:

session_id
>   (string) (required) Unique identifier to use for this session.

ttl
>   (number) (required) Requested session duration, in seconds.

context
>   (string) (optional) Additional context to be passed to the peer when it joins the session.

Servers SHOULD automatically expire the server-side session state after its TTL duration expires. Servers MAY close connections to connected clients when their session expires. Servers MAY impose a shorter TTL if the requested TTL is too long.

Servers respond to this message with a *session-created*.

join-session

Attempts to join an existing session.

Sent by the *signer* as part of session negotiation.

Fields:

session_id
>   (string) (required) Identifier of session to join.

context
>   (string) (optional) Additional context to pass through to the other peer.

Servers respond to this message with a *session-joined*.

send-message

Sends an (encrypted) message to the other peer in this session.

Fields:

session_id
> (string) (required) Identifier of session to use for peer lookup.

message
> (string) (required) Base64 encoded ciphertext of an AEAD encrypted message to send to the peer.

Server implementations MUST ensure that the client issuing this request are bound to the session they are attempting to send a message to.

Servers react to this message by sending a *peer-message* to the other peer in the specified session.

Servers respond to this message with a *message-sent*.

goodbye

Indicates the client is finished and will be disconnecting.

Fields:

session_id
> (string) (required) Identifier of session to use for peer lookup.

reason
> (string) (option) Reason the client is disconnecting.

Server implementations MUST ensure that the client issuing this request is bound to the session they are attempting to close.

Servers react to this message by sending a *session-closed* to the other peer in the specified session.

Servers respond to this message with a *session-closed*.

Server Sent Messages

The following sections denote the types of messages sent from the server to clients.

Section names denote the value of the `type` field in the message.

error

Conveys information about a server-side error.

Could be sent in reply to any API request or sent asynchronously if some error occurred (such as the peer disconnecting unexpectedly).

Fields:

code
> (string) (required) Value that uniquely identifies this error type.

message
> (string) (required) Human readable error message.

greeting

Conveys information about the server.

Sent in reply to a *hello* request.

Fields:

apis
>   (array of strings) (required) Names of APIs that the server supports.

motd
>   (string) (optional) *Message of the day* conveying messaging that the server operator wishes clients to know about.

session-created

Conveys the successful creation of a session.

Sent in reply to a *create-session* request.

session-joined

Conveys the successful joining into a session.

Sent in reply to a *join-session* request.

Sent asynchronously by servers in response to a *join-session* issued by the joining peer.

Fields:

context
>   (string) (optional) Data from the peer required to finish initializing the session.
>
>   If this message was sent in reply to a *join-session*, the value will be from the initiating peer.
>
>   If this message was sent to the pre-existing peer in reaction to a *join-session*, the value will be from the joining peer.

message-sent

Conveys the successful sending of a message to the session peer.

Sent in reply to a *send-message* request.

peer-message

Delivers an (encrypted) message from the peer in this session.

Sent asynchronously by servers in response to a *send-message* issued by the other peer in a session.

Fields:

message
>   (string) (required) Base64 encoded AEAD message.

session-closed

Conveys that the session has been finalized and can no longer be used.

Sent in reply to a *goodbye* request as well as asynchronously to the peer in its session.

Fields:

reason
> (string) (optional) Provides further context on why the session was closed.

Peer to Peer Messages

Peers within a session communicate with each other by sending and receiving *send-message* and *peer-message*, respectively.

The `message` field denotes a base64 encoded AEAD encrypted message. The message consists of the ciphertext with the authentication tag appended. The plaintext of these messages is the JSON encoding of an object having the following keys:

type
> (string) (required) The message type. This is unique message namespace from server-sent messages.

payload
> (object) (optional) Payload for this message.

The following sections denote the types of peer-to-peer messages. The section names denote the value for the `type` field.

ping

Check on the status of the peer.

Receivers should send a *pong* in response.

pong

Respond to a status check from a peer.

Sent in response to a *ping* message.

request-signing-certificate

Requests the peer to send it information about its signing certificate.

Receivers should send a *signing-certificate* in response.

Should only be sent by the *initiator*.

signing-certificate

Describes the signing certificate(s) that is being used by the signer.

Sent in response to a *request-signing-certificate*.

Fields:

certificates
> (array of object) (required) Contains a list of signing certificates that will potentially be used.
>
> Each entry is an object described below.
>
> Today, there is likely a single certificate in this array. We've left the door open for supporting the use of multiple signing certificates in the future.

Each entry in the `certificatess` array is an object with the following fields:

certificate
> (string) (required) Base64 encoded DER of the public X.509 certificate.

chain
> (array of strings) (optional) Base64 encoded DER of additional public X.509 certificates in the signing chain for this certificate.

sign-request

Requests the cryptographic signing of a message.

Fields:

message
> (string) (required) Base64 encoded message to be signed.

signature

Conveys the cryptographic signature over a message.

Sent in response to a *sign-request*.

Fields:

message
> (string) (required) Base64 encoded message that was signed.

signature
> (string) (required) Base64 encoded signature data.

algorithm_oid
> (string) (required) Base64 encoded DER encoding of OID denoting the signature algorithm.

Remote Code Signing Design and Security Considerations

Design Goals and Constraints

The design of remote signing is influenced with the following primary goals in mind:

- The initiating machine MUST NOT have direct access to the private signing key. Ever. The private key (or ability to create signatures with it) is only ever in possession of the signer.

- The private key cannot be used without the signer's knowledge (and optional consent to each use).

- The initiating machine must be able to run remotely / non-interactively.

We also imposed the following constraints when considering designs:

- The initiating machine is partially trusted. We assume that if you trust the initiating machine to invoke a signing operation then you trust that machine to e.g. not lie about the signing requests it subsequently presents to the signer.

- We should place minimal trust in any 3rd party servers or machines. Assume all 3rd parties are malicious and will attempt to coerce signers into signing arbitrary content.

- 3rd party servers should have access to as little information about signing activity as possible. e.g. 3rd party servers should not be able to observe the messages that are signed, the produced signatures, or the certificates used to sign. They may observe details that leak through side channels, such as the number of messages exchanged and the sizes of encrypted ciphertexts.

- We assume the existence of an out-of-band side-channel for 2 peers to exchange information at signing time. This means we require some synchronous activity by the signer in order to fulfill signing requests. (The signer isn't just running an always-running server that responds to signing requests.)

Threat Models

The following threat models dictate some design choices:

- A malicious brokering server or man-in-the-middle could coerce the signer into signing unwanted content.

- A malicious 3rd party could disrupt signing operations by sending garbage messages to the brokering server, either in general or directed at established sessions. i.e. DoS against the server.

- A malicious brokering server or man-in-the-middle could fulfill signature requests using the *wrong* certificate.

If signing sessions were conducted without any prior knowledge of the peer, neither peer would be able to trust or authenticate the other. You could securely exchange end-to-end encrypted messages with a peer. But the *initiator* wouldn't be able to answer the question *is this signed by who I want it to be signed by*. And more importantly, the *signer* wouldn't be able to answer *do I trust the initiator to send me content that I want to sign*.

You can't establish a trust relationship without a trust anchor. **So in order to establish trust we require that peers share pre-existing knowledge of the other before signing operations.** The exact mechanism can vary. But *some* pre-existing knowledge needs to be conveyed to the other peer in order to serve as a trust anchor.

Since all designs rule out the possibility of the private key being directly accessed or used by the *initiator*, the next best attack vector is tricking the *signer* into signing untrusted/malicious content.

The easiest way to conduct this attack is for a malicious server or man-in-the-middle to intercept communications and/or issue a malicious signing request. There are a few mitigations for this.

First, *signers* must have presence in order to create signatures. When signers go offline, they can't produce signatures. So attacks against signers must occur when the signer is online.

Second, we employ end-to-end encryption of peer-to-peer messages using ephemeral encryption keys unique to the session and logically derived from a pre-existing trust anchor. A malicious 3rd party would need access to data never transmitted in plaintext through the server in order to decrypt messages or issue fake/malicious messages.

Security Analysis in the Bigger Picture

When considering the overall security of remote code signing, we have to consider the broader ecosystem in which it exists.

Without remote code signing, the following are all commonly true:

- Signing keys are copied to multiple machines to make it easier to access them.

- Signing keys are made available as secrets on CI workers.

- Access to perform operations on the signing key is always on. e.g. anybody who can talk to the HSM can create a signature.

- Security conscious people (those who want to minimize risk for private keys) need to impose a more complicated release pipeline - one that typically entails copying assets to a separate machine, signing them, then copying elsewhere. These steps are often tedious and effectively constitute a barrier to good security hygiene.

There are general principles of private key management:

- You should have as few copies of the private key as possible. Ideally 1.

- Keys should be as short lived as possible or access to them should be limited in time duration.

Traditional solutions to code signing violate these principles because there's not an easy-to-use / viable alternative. So in the absence of remote code signing, commonly practiced code signing key management is generally not great.

We believe that our design of remote code signing is intrinsically more secure than what is commonly practiced because:

- The signer in possession of the private key must be present. There is no unlimited access to the private key outside an active signing session.

- You can have exactly 1 copy of the private key without compromising on usability. The urge to make copies to streamline CI/CD is largely mitigated via an easy-to-use remote signing UI.

In addition, the design and implementation of the relay server further bolsters security by:

- Purging sessions after a maximum time to live (measured in minutes).

- Refusing to allow N>2 peers from sending messages to a session.

- Requiring active presence for message exchange. The server doesn't store a copy of relayed signing messages so there isn't a potential for someone to deposit a malicious message for later retrieval.

And these security properties are delivered without even factoring in end-to-end message encryption! The end-to-end encryption is effectively protections against a malicious server or man-in-the-middle. These are arguably necessary protections - especially when using a server hosted by an (untrusted) 3rd party. But for scenarios where you run your own server and you trust the network, end-to-end encryption isn't buying you much beyond what signer presence requirements and server design already deliver.

Default Remote Code Signing Server

By default, this project uses the remote code signing server at `wss://ws.codesign.gregoryszorc.com/`.

This service is operated by the maintainer of this project and is provided for free for use by the community. However, there is no formal or legal agreement around the availability of its service or its operation.

The service is hosted on AWS and uses API Gateway + Lambda + DynamoDB and should be highly reliable, as these services rarely experience outages.

The *Remote Code Signing Protocol* and implementation of the server have been purposefully designed to be respectful of privacy of its users.

Meaningful messages between clients are end-to-end encrypted and the server is unable to determine the contents of those messages. The server only has access to protocol-level details, such as which APIs are being invoked and the sizes of the payloads.

The server does have access to client IPs and any additional metadata in HTTP requests and websocket frames. However, IPs or other identifying information is not read by our custom code powering the websocket server or retained in any logs to the best of our knowledge. (We believe user data to be toxic and don't want anything to do with it.)

Some metrics to monitor the health of the service and help prevent abuse are recorded. These include the counts of different API invocations and the sizes of message payloads.

The code powering the server and the Terraform for deploying it on AWS are open source and available to audit. See *Running Your Own Server* for details. Of course, there's no way to prove that `ws.codesign.gregoryszorc.com` is running the same configuration as the provided open source code. You *just* have to trust that the maintainer of this project values the privacy of his users.

Running Your Own Server

If you are unable or unwilling to use the default remote signing server operated by the maintainer of this project, it is possible to deploy your own server instance.

The source code for the server and a Terraform module for deploying it into AWS are available in this repository in the `terraform-modules/remote-code-signing` directory. The canonical location is https://github.com/indygreg/PyOxidizer/tree/main/terraform-modules/remote-code-signing.

See its README for instructions on how to use. Once deployed at a different hostname, you'll need to provide the `--remote-signing-url` argument to relevant commands to override the default signing server URL.

A Primer on Gatekeeper

Gatekeeper is the name Apple gives to a set of technologies that enforce application execution policies at the operating system level. Essentially, Gatekeeper answers the question *is this software allowed to run*.

When Gatekeeper runs, it performs a *security assessment* against the binary and the currently configured system policies from the system policy database (see `man syspolicyd`). If the binary fails to meet the requirements, Gatekeeper prevents the binary from running.

The `spctl` Tool

The `spctl` program distributed with macOS allows you to query and manipulate the assessment policies.

If you run `sudo spctl --list`, it will print a list of rules. e.g.:

```
$ sudo spctl --list
8[Apple System] P20 allow lsopen
        anchor apple
3[Apple System] P20 allow execute
        anchor apple
2[Apple Installer] P20 allow install
        anchor apple generic and certificate 1[subject.CN] = "Apple Software Update
→Certification Authority"
17[Testflight] P10 allow execute
        anchor apple generic and certificate 1[field.1.2.840.113635.100.6.2.1] exists
→and certificate leaf[field.1.2.840.113635.100.6.1.25.1] exists
10[Mac App Store] P10 allow install
        anchor apple generic and certificate leaf[field.1.2.840.113635.100.6.1.10] exists
5[Mac App Store] P10 allow install
        anchor apple generic and certificate leaf[field.1.2.840.113635.100.6.1.10] exists
4[Mac App Store] P10 allow execute
        anchor apple generic and certificate leaf[field.1.2.840.113635.100.6.1.9] exists
16[Notarized Developer ID] P5 allow lsopen
        anchor apple generic and certificate 1[field.1.2.840.113635.100.6.2.6] exists
→and certificate leaf[field.1.2.840.113635.100.6.1.13] exists and notarized
12[Notarized Developer ID] P5 allow install
        anchor apple generic and certificate 1[field.1.2.840.113635.100.6.2.6] exists
→and (certificate leaf[field.1.2.840.113635.100.6.1.14] or certificate leaf[field.1.2.
→840.113635.100.6.1.13]) and notarized
11[Notarized Developer ID] P5 allow execute
        anchor apple generic and certificate 1[field.1.2.840.113635.100.6.2.6] exists
→and certificate leaf[field.1.2.840.113635.100.6.1.13] exists and notarized
9[Developer ID] P4 allow lsopen
        anchor apple generic and certificate 1[field.1.2.840.113635.100.6.2.6] exists
→and certificate leaf[field.1.2.840.113635.100.6.1.13] exists and legacy
7[Developer ID] P4 allow install
        anchor apple generic and certificate 1[field.1.2.840.113635.100.6.2.6] exists
→and (certificate leaf[field.1.2.840.113635.100.6.1.14] or certificate leaf[field.1.2.
→840.113635.100.6.1.13]) and legacy
6[Developer ID] P4 allow execute
        anchor apple generic and certificate 1[field.1.2.840.113635.100.6.2.6] exists
→and certificate leaf[field.1.2.840.113635.100.6.1.13] exists and (certificate
→leaf[timestamp.1.2.840.113635.100.6.1.33] absent or certificate leaf[timestamp.1.2.840.
→113635.100.6.1.33] < timestamp "20190408000000Z")
2718[GKE] P0 allow lsopen [(gke)]
        cdhash H"975d9247503b596784dd8a9665fd3ff43eb7722f"
2717[GKE] P0 allow execute [(gke)]
        cdhash H"cf782d6467be86b73a83d86cd6d8c9f87d9d9ce5"
...
18[GKE] P0 allow lsopen [(gke)]
        cdhash H"cf5f88b3b2ff4d8612aabb915f6d1f712e16b6f2"
15[Unnotarized Developer ID] P0 deny lsopen
```

```
        anchor apple generic and certificate 1[field.1.2.840.113635.100.6.2.6] exists␣
→and certificate leaf[field.1.2.840.113635.100.6.1.13] exists
14[Unnotarized Developer ID] P0 deny install
        anchor apple generic and certificate 1[field.1.2.840.113635.100.6.2.6] exists␣
→and (certificate leaf[field.1.2.840.113635.100.6.1.14] or certificate leaf[field.1.2.
→840.113635.100.6.1.13])
13[Unnotarized Developer ID] P0 deny execute
        anchor apple generic and certificate 1[field.1.2.840.113635.100.6.2.6] exists␣
→and certificate leaf[field.1.2.840.113635.100.6.1.13] exists and (certificate␣
→leaf[timestamp.1.2.840.113635.100.6.1.33] exists and certificate leaf[timestamp.1.2.
→840.113635.100.6.1.33] >= timestamp "20190408000000Z")
```
```

The first line of each item identifies the policy. The second line is a *code requirement language expression*. This is a DSL that compiles to a binary expression tree for representing a test to perform against a binary. See `man csreq` for more.

Some of these expressions are pretty straightforward. For example, the following entry says to allow executing a binary with a code signature whose *code directory* hash is `cf782d6467be86b73a83d86cd6d8c9f87d9d9ce5`:

```
2717[GKE] P0 allow execute [(gke)]
        cdhash H"cf782d6467be86b73a83d86cd6d8c9f87d9d9ce5"
```

The *code directory* refers to a data structure within the code signature that contains (among other things) content digests of the binary. The hash/digest of the code directory itself is effectively a chained digest to the actual binary content and theoretically a unique way of identifying a binary. So `cdhash H"cf782d6467be86b73a83d86cd6d8c9f87d9d9ce5"` is a very convoluted way of saying *allow this specific binary (specified by its content hash) to execute*.

Other rules are more interesting. For example:

```
11[Notarized Developer ID] P5 allow execute
        anchor apple generic and certificate 1[field.1.2.840.113635.100.6.2.6] exists
        and certificate leaf[field.1.2.840.113635.100.6.1.13] exists and notarized
```

We see the description (`Notarized Developer ID`) but what does that expression mean?

Well, first this expression parses into a tree. We won't attempt to format the tree here. But essentially the following conditions must `all` be true:

- `anchor apple generic`

- `certificate 1[field.1.2.840.113635.100.6.2.6] exists`

- `certificate leaf[field.1.2.840.113635.100.6.1.13] exists`

- `notarized`

`anchor apple generic` and `notarized` are essentially special expressions that expand to mean *the certificate signing chain leads back to an Apple root certificate authority (CA)* and *there is a supplemental code signature from Apple that can only come from Apple's notarization service*.

But what about those `certificate` expressions? That `certificate <position>[field.*]` syntax essentially says *the code signature certificate at* ``<position>``*in the certificate chain has an X.509 certificate extension with OID* ``X`` (where `X` is a value like `A.B.C.D.E.F`).

This is all pretty low level. But essentially X.509 certificates can have a series of *extensions* that further describe the certificate. Apple code signing uses these extensions to convey metadata about the certificate. And since code signing

certificates are signed, whoever signed those certificates is effectively also approving of whatever is conveyed by the extensions within.

But what do these extensions actually mean? Running `rcodesign x509-oids` may give us some help:

```
$ rcodesign x509-oids`
...
Code Signing Certificate Extension OIDs
...
1.2.840.113635.100.6.1.13        DeveloperIdApplication
...
Certificate Authority Certificate Extension OIDs
...
1.2.840.113635.100.6.2.6         DeveloperId
```

We see `1.2.840.113635.100.6.2.6` is the OID of an extension on certificate authorities indicating they act as the *Apple Developer ID* certificate authority. We also see that `1.2.840.113635.100.6.1.13` is the OID of an extension saying the certificate acts as a code signing certificate for *applications* associated with an *Apple Developer ID*.

So, what this expression translates to is essentially:

- Trust code signatures whose certificate signing chain leads back to an Apple CA.

- The signer of the code signing certificate must have the extension that identifies it as the *Apple Developer ID* certificate authority.

- The code signing certificate itself must have the extension that says it is an *Apple Developer ID* for use with *application* signing.

- The binary is *notarized*.

In simple terms, this is saying *allow execution of binaries that were signed by a Developer ID code signing certificate which was signed by Apple's Developer ID certificate authority and are also notarized*.

Selectively Bypassing Gatekeeper with Custom Assessment Policies

By default, Apple locks down their operating systems such that the default assessment policies enforced by Gatekeeper restrict what can be run. The restrictions vary by operating system (iOS is more locked down than macOS for example).

On macOS, it is possible to change the system assessment policies via the `spctl` tool. By injecting your own rules, you can allow binaries through meeting criteria expressible via *code requirements language expressions*. This allows you to allow binaries having:

- A specific *code directory hash* (uniquely identifies the binary).

- A specific code signing certificate identified by its certificate hash.

- Any code signing certificate whose trust/signing chain leads to a trusted certificate.

- Any code signing certificate signed by a certificate containing a certain X.509 extension OID.

- A code signing certificate with specific values in its subject field.

- And many more possibilities. See Apple's docs on the requirements language for more possibilities.

Defining custom rules is possible via the under-documented `spctl --add --requirement` mode. In this mode, you can register a code requirements expression into the system database for Gatekeeper to utilize. The following sections give some examples of this.

Verifying Assessment Policies

The sections below document how to define custom assessment policies to allow execution of binaries/installers/etc signed by certificates that aren't normally supported.

When doing this, you probably want a way to verify things work as expected.

The `spctl --assess` mode puts `spctl` in *assessment mode* and tells you what verdict Gatekeeper would render. e.g.:

```
$ spctl --assess --type execute -vv /Applications/Firefox.app
/Applications/Firefox.app: accepted
source=Notarized Developer ID
```

Do note that this only works on app bundles (not standalone executable binaries)! If you run `spctl --assess` on a standalone executable, you get an error:

```
$ spctl --assess -vv /usr/bin/ssh
/usr/bin/ssh: rejected (the code is valid but does not seem to be an app)
origin=Software Signing
```

In addition, macOS uses the `com.apple.quarantine` extended file attribute to *quarantine* files and prevent them from running via the graphical UI. It can sometimes be handy to add this attribute back to a file to simulate a fresh quarantine. You can do this by running a command like the following:

```
xattr -w com.apple.quarantine "0001;$(printf %x $(date +%s));manual;$(/usr/bin/uuidgen)"␣
↪/path/to/file
```

(This extended attribute isn't added to files downloaded by tools like `curl` or `wget` which is why you can execute binaries obtained via these tools but can't run the same binary downloaded via a web browser.)

Allowing Execution of Binaries Signed by a Specific Certificate

Say you have a single code signing certificate and want to be able to run all binaries signed by that certificate. We can construct a *code requirement expression* that refers to this specific certificate.

The most reliable way to specify a single certificate is via a digest of its content. Assuming no two certificates have the same digest, this uniquely identifies a certificate.

You can use `rcodesign analyze-certificate` to locate a certificate's content digest.:

```
rcodesign analyze-certificate --pem-source path/to/cert | grep fingerprint
SHA-1 fingerprint:          0b724bcd713c9f3691b0a8b0926ae0ecf9e7edd8
SHA-256 fingerprint:        ␣
↪ac5c4b5936677942e017bca1570aaa9e763674c4b66709231b15118e5842aeca
```

The *code requirement* language only supports SHA-1 hashes. So we construct our expression referring to this certificate as `certificate leaf H"0b724bcd713c9f3691b0a8b0926ae0ecf9e7edd8"`.

Now, we define an assessment rule to allow execution of binaries signed with this certificate:

```
sudo spctl --add --type execute --label 'My Cert' --requirement \
  'certificate leaf H"0b724bcd713c9f3691b0a8b0926ae0ecf9e7edd8"'
```

Now Gatekeeper should allow execution of all binaries signed with this exact code signing certificate!

If the signing certificate hash is registered in the system assessment policy database, there is no need to register the certificate in a *keychain* or mark that certificate as *trusted* in a keychain. The signing certificate also does not need to

chain back to an Apple certificate. And since the requirement expression doesn't say `and notarized`, binaries don't need to be notarized by Apple either. **This effectively allows you to sidestep the default requirement that binaries be signed and notarized by certificates that Apple is aware of.** Congratulations, you've just escaped Apple's walled garden (at your own risk of course).

Do note that for files with the `com.apple.quarantine` extended attribute, you may see a dialog the first time you run this file. You can prevent that by removing the extended attribute via `xattr -d com.apple.quarantine /path/to/file`.

Allowing Execution of Binaries Signed by a Trusted CA

Say you are an enterprise or distributed organization and want to have multiple code signing certificates. Using the approach in the section above you could individually register each code signing certificate you want to allow. However, the number of certificates can quickly grow and become unmanageable.

To solve this problem, you can employ the strategy that Apple itself uses for code signing certificates associated with Developer ID accounts: trust code signing certificates themselves issued/signed by a trusted certificate authority (CA).

To do this, we'll again craft a *code requirement expression* referring to our trusted CA certificate.

This looks very similar to above except we change the position of the trusted certificate:

```
sudo spctl --add --type execute --label 'My Trusted CA' --requirement \
  'certificate 1 H"0b724bcd713c9f3691b0a8b0926ae0ecf9e7edd8"'
```

That `certificate 1` says to apply to the certificate that signed the certificate that produced the code signature. By trusting the CA certificate, you implicitly trust all certificates signed by that CA certificate.

Note that if you use a custom CA for signing code signing certificates, you'll probably want to follow some best practices for running your own Public Key Infrastructure (PKI) like publishing a Certificate Revocation List (CRL). This is a complex topic outside the scope of this documentation. Ask someone with *Security* in their job title for assistance.

For CA certificates issuing/signing code signing certificates, you'll want to enable a few X.509 certificate extensions:

- Key Usage (`2.5.29.15`): *Digital Signature* and *Key Cert Sign*
- Basic Constraints (`2.5.29.19`): CA=yes
- Extended Key Usage (`2.5.29.37`): Code Signing (`1.3.6.1.5.5.7.3.3`); critical=true

You can create CA certificates in the `Keychain Access` macOS application. If you create CA certificates another way, you may want to compare certificate extensions and other fields against those produced via `Keychain Access` to make sure they align. It is unknown how much Apple's operating systems enforce requirements on the X.509 certificates. But it is a good idea to keep things as similar as possible.

1.2 oxidized_importer

A Python extension module [implemented in Rust] providing a highly performant alternate module and resource importing mechanism. `oxidzed_importer` can be used to import Python modules and resources from memory, enabling Python applications to be single file executables.

`oxidized_importer` is usable as a standalone Python package and can be installed from PyPI.

1.2.1 `oxidized_importer` Python Extension

`oxidized_importer` is a Python extension module maintained as part of the PyOxidizer project that allows you to:

- Install a custom, high-performance module importer (`OxidizedFinder`) to service Python `import` statements and resource loading (potentially from memory).

- Scan the filesystem for Python resources (source modules, bytecode files, package resources, distribution metadata, etc) and turn them into Python objects.

- Serialize Python resource data into an efficient binary data structure for loading into an `OxidizedFinder` instance. This facilitates producing a standalone *resources blob* that can be distributed with a Python application which contains all the Python modules, bytecode, etc required to power that application.

`oxidized_importer` is automatically compiled into applications built with PyOxidizer. It can also be built as a standalone extension module and used with regular Python installs.

Getting Started

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.

Installing from PyPI

`oxidized_importer` is available on PyPI. This means that installing is as simple as:

```
$ pip3 install oxidized_importer
```

Compiling from Source

To build from source, obtain a clone of PyOxidizer's Git repository and run the `setup.py` script or use `pip` to build the Python project in the root of the repository. e.g.:

```
$ python3.9 setup.py build_ext -i
$ python3.9 setup.py install

$ pip3.9 install .
$ pip3.9 wheel .
```

The `setup.py` is pretty minimal and is a thin wrapper around `cargo build` for the underlying Rust project. If you want to build using Rust's standard toolchain, do something like the following:

```
$ cd oxidized-importer
$ cargo build --release
```

If you don't have a Python 3.9 `python3` executable in your `PATH`, you will need to tell the Rust build system which `python3` executable to use to help derive the build configuration for the Python extension:

```
$ PYO3_PYTHON=/path/to/python3.9 cargo build
```

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.

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.

`OxidizedFinder` Meta Path Finder

OxidizedFinder is a Python type implementing a custom and fully-featured *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, as no filesystem probing or other explicit I/O is performed.

Instances of *OxidizedFinder* are optionally bound to binary blobs 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*.

`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'>,
    <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).

If *OxidizedFinder* is being installed on `sys.meta_path`, its *path_hook* method will be registered as the first item on `sys.path_hooks`.

If filesystem importing is disabled, all entries of `sys.meta_path` and `sys.path_hooks` not related to *OxidizedFinder* will be removed.

Python API

See *OxidizedFinder* for the Python API documentation.

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. oxidized_finder 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.

__file__ and __cached__ Module Attributes

Python modules typically have a __file__ attribute holding a str defining the filesystem path the source module was imported from (usually a path to a .py file). There is also the similar - but lesser known - __cached__ attribute holding the filesystem path of the bytecode module (usually the path to a .pyc file).

Important: *OxidizedFinder* will not set either attribute when importing modules from memory.

These attributes are not set because it isn't obvious what the values should be! Typically, __file__ is used by Python as an anchor point to derive the path to some other file. However, when loading modules from memory, the traditional filesystem hierarchy of Python modules does not exist. In the opinion of PyOxidizer's maintainer, exposing __file__ would be *lying* and this would cause more potential for harm than good.

While we may make it possible to define __file__ (and __cached__) on modules imported from memory someday, we do not yet support this.

OxidizedFinder does, however, set __file__ and __cached__ on modules imported from the filesystem. So, a workaround to restore these missing attributes is to avoid in-memory loading.

Note: Use of __file__ is commonly encountered in code loading *resource files*. See *Loading Resource Files* for more on this topic, including how to port code to more modern Python APIs for loading resources.

__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.

Support for `__init__` in Module Names

There exists Python code that does things like `from .__init__ import X`.

`__init__` is special in Python module names because it is the filename used to denote a Python package's filename. So syntax like `from .__init__ import X` is probably intended to be equivalent to `from . import X`. Or `import foo.__init__` is probably intended to be written as `import foo`.

Python's filesystem importer doesn't treat `__init__` in module names as special. If you attempt to import a module named `foo.__init__`, it will attempt to locate a file named `foo/__init__.py`. If that module is a package, this will succeed. However, the module name seen by the importer has `__init__` in it and the name on the created module object will have `__init__` in it. This means that you can have both a module `foo` and `foo.__init__`. These will both be derived from the same file but are actually separate module objects.

PyOxidizer will automatically remove trailing `.__init__` from module names. This will enable PyOxidizer to work with syntax such as `import foo.__init__` and `from .__init__ import X` and therefore be compatible with Python code in the wild. However, PyOxidizer may not preserve the `.__init__` in the module name. For example, with Python's path based importer, you could have both `foo` and `foo.__init__` in `sys.modules` but PyOxidizer will only have `foo`.

A limitation of PyOxidizer module name normalization is it only normalizes the single trailing `.__init__` from the module name: `__init__` appearing inside the module name are not normalized. e.g. `foo.__init__.bar` is not normalized to `foo.bar`. This may introduce incompatibilities with Python code in the wild. However, for this to be true, the filesystem layout would have to be something like `foo/__init__/bar.py`. This hopefully does not occur in the wild. But it is conceivable it does.

See https://github.com/indygreg/PyOxidizer/issues/317 and https://bugs.python.org/issue42564 for more discussion on this issue.

ResourceReader Compatibility

`ResourceReader` has known compatibility differences with Python's default filesystem-based importer. See *Support for ResourceReader* for details.

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.

`importlib.metadata` Compatibility

OxidizedFinder implements `find_distributions()` and therefore provides the required hook for `importlib.metadata` to resolve `Distribution` instances. However, the returned objects do not implement the full `Distribution` interface.

Here are the known differences between `OxidizedDistribution` and `importlib.metadata.Distribution` instances:

- `OxidizedDistribution` is not an instance of `importlib.metadata.Distribution`.

- `locate_file()` is not defined.

- `@staticmethod at()` is not defined.

- `@property files` raises `NotImplementedError`.

There are additional _ prefixed attributes of `importlib.metadata.Distribution` that are not implemented. But we do not consider these part of the public API and don't feel they are worth calling out.

In addition, `OxidizedFinder.find_distributions()` ignores the `path` attribute of the passed `Context` instance. Only the `name` attribute is consulted. If `name` is `None`, all packages with registered distribution files will be returned. Otherwise the returned `list` contains at most 1 `PyOxidizerDistribution` corresponding to the requested package `name`.

`pkgutil` Compatibility

The [pkgutil](#) package in Python's standard library reacts to special functionality on `MetaPathFinder` instances.

`pkgutil.iter_modules()` attempts to use an `iter_modules()` method to obtain results.

OxidizedFinder implements `iter_modules(prefix="")` and `pkgutil.iter_modules()` should work. However, there are some differences in behavior:

- `iter_modules()` is defined to be a generator but `OxidizedFinder.iter_modules()` returns a `list`. `list` is iterable and this difference should hopefully be a harmless implementation detail.

- Support for the `path` argument to `pkgutil.iter_modules()` requires that *OxidizedFinder*'s *path_hook* is installed in `sys.path_hooks`. This will be done automatically if *OxidizedFinder* is installed at interpreter initialization time.

Paths Hooks Compatibility

The *OxidizedFinder.path_hook* method from an instantiated instance can be installed on `sys.path_hooks` to enable a *OxidizedFinder* to function as a [path entry finder](#).

As a brief refresher, callables on `sys.path_hooks` are called with *paths*, giving them the opportunity to service a particular *path*. If a *path hook* responds to a *path* by returning a *path entry finder*, that returned object will service that *path*. Often, the *paths* passed to *path hooks* are from `sys.path`. However, arbitrary *paths* can be passed in. A property of the returned *path entry finder* is it only targets a particular level in the *package hierarchy*. Unlike *meta path finders* (which can service any named resource it knows about), *path entry finders* are *bound* to a specific package target level and will only return resources existing at that level.

path hooks are used by the following mechanisms:

- The standard library [PathFinder](#) (the meta path finder that Python uses to load resources from the filesystem) uses `sys.path_hooks` as part of resolving a *finder* for a given `sys.path` entry.

- `pkgutil.get_importer()` for resolving the finder for a given `sys.path` entry. This in turn is used by various code, including other `pkgutil` APIs.

- `pkg_resources` maps *path entry finder* types to functions to enable a resolution of `pkg_resources.Distribution` instances for individual *paths*.

When installed on `sys.path_hooks`, *OxidizedFinder.path_hook* will respond to the following path values:

- The path to the current executable, as defined by *OxidizedFinder.path_hook_base_str*.

- A virtual sub-directory of the path to the current executable, as defined by *OxidizedFinder.path_hook_base_str*.

Important: *path_hook* is very strict about what values it will respond to.

The value **must** be a `str` and be equal to *OxidizedFinder.path_hook_base_str* or have *OxidizedFinder.path_hook_base_str* plus a directory separator as the exact string prefix.

path_hook will **not** respond to `bytes`, `pathlib.Path`, or any other path-like type.

OxidizedFinder.path_hook_base_str **may not** be the same value as `sys.executable`. Always use *OxidizedFinder.path_hook_base_str* to derive `sys.path` values to ensure the path hook will respond.

When *path_hook* is called with its *OxidizedFinder.path_hook_base_str* value, a *OxidizedPathEntryFinder* bound to the source *OxidizedFinder* is returned. This finder is able to service *root resources* (i.e. top-level modules and packages).

When *path_hook* is called with a virtual sub-directory of *OxidizedFinder.path_hook_base_str*, the same thing happens except the returned *OxidizedPathEntryFinder* will only service resources at the exact package hierarchy specified by that virtual sub-directory.

The validation and normalization of path values is similar to the following:

```python
def path_hook(self, path: str):
    # Path exactly matching current_exe will be bound to resources at root.
    if path == self.path_hook_base_str:
        return ...

    # Virtual sub-directories must begin with self.current_exe + directory
    # separator.
    if not path.startswith((self.path_hook_base_str + "/", self.path_hook_base_str + "\\
")):
        raise ImportError

    # Part after directory separator.
    package_part = path[len(self.path_hook_base_str) + 1:]

    # Normalize to UNIX style directory separators, allowing Windows
    # separators to exist.
    package_part = package_part.replace("\\", "/")

    # Ban leading, trailing, and consecutive directory separators.
    if package_part.startswith("/") or package_part.endswith("\\") or package_part.
contains("//"):
        raise ImportError()

    # Ban dots in directory components.
```

(continues on next page)

```
    for part in package_part.split("/"):
        if part.startswith(".") or part.endswith(".") or part.contains(".."):
            raise ImportError()

    # Normalize directory tree to package hierarchy. e.g. foo/bar -> foo.bar.
    package = package_part.replace("/", ".")

    # When converting the package string to a Rust string to facilitate
    # resource name comparisons, it is encoded to UTF-8, replacing
    # "bad" code points with the Unicode replacement code point.
    rust_package_string = package.encode("utf-8", "replace")
```

Note that when the package component of virtual sub-directories is converted to a Rust string, we use the UTF-8 encoding, not Python's active filesystem encoding. This is to keep things simpler. And since *OxidizedFinder* indexes resource names using Rust's UTF-8 backed string type anyway, this seems semantically correct from the perspective of `oxidized_importer`.

As an example, if `path` were `os.path.join(finder.path_hook_base_str, "a")`, the finder would only service modules of the form `a.*`. So `a`, `a.b` would match but `a.b.c` and `d` would not.

For best results, use `os.path.join(finder.path_hook_base_str, str)` to define values that will be accepted by the path hook.

OxidizedPathEntryFinder complies with the PathEntryFinder protocol and implements *OxidizedPathEntryFinder.find_spec()* and *OxidizedPathEntryFinder.invalidate_caches()*. However, support for the deprecated methods `find_loader` and `find_module` is not implemented. Instances also implement *OxidizedPathEntryFinder.iter_modules()*, enabling it to be used by `pkgutil.iter_modules()`.

pkg_resources Compatibility

OxidizedFinder can be registered as a provider for `pkg_resources`, enabling `pkg_resources` APIs to be used with resources tracked by *OxidizedFinder* instances.

However, there are known compatibility differences. See *Support for pkg_resources* for more.

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.

OxidizedResource

Represents a *resource* that is indexed by a *OxidizedFinder* instance.

See *OxidizedResource* for API documentation.

OxidizedResource Resource Types

Each `OxidizedResource` instance describes a particular type of resource. If a resource identifies as a type, it sets one of the following `is_*` attributes to `True`:

OxidizedResource.is_module
> A Python module. These typically have source or bytecode attached.
>
> Modules can also be packages. In this case, they can hold additional data, such as a mapping of resource files.

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

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

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

OxidizedResource.is_shared_library
> A shared library. e.g. a `.so` or `.dll`.

PythonModuleSource

The *PythonModuleSource* type represents Python module source code. e.g. a `.py` file. See its linked API documentation for more.

PythonModuleBytecode

The *PythonModuleBytecode* type represents Python module bytecode. e.g. what a `.pyc` file holds (but without the header that a `.pyc` file has).

PythonExtensionModule

The *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.

Note: Properties of this type are read-only.

PythonPackageResource

The *PythonPackageResource* type represents a non-module *resource* file.

PythonPackageDistributionResource

The *PythonPackageDistributionResource* type represents a non-module *resource* file living in a package distribution directory

Resource Scanning APIs

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

find_resources_in_path(path)

This function scans a filesystem path and returns discovered resources. See *find_resources_in_path()* for the API documentation.

To discover all filesystem based resources that Python's `PathFinder` *meta path finder* would (with the exception of `.zip` files), try the following:

```python
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))
```

OxidizedResourceCollector Python Type

The *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.

To create a collector that only marks resources for in-memory loading:

```python
import oxidized_importer

collector = oxidized_importer.OxidizedResourceCollector(
    allowed_locations=["in-memory"]
)
```

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.

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.

Support for `ResourceReader`

`oxidized_importer` implements the `ResourceReader` interface for loading resource files.

However, compatibility with Python's default filesystem-based implementation can vary. Unfortunately, various behavior with `ResourceReader` is undefined, so it isn't clear if CPython or `oxidized_importer` is buggy here.

`oxidized_importer` maintains an index of known resource files. This index is logically a `dict` of dict``s, where the outer key is the Python package name and the inner key is the resource name. Package names are fully qualified. e.g. ``foo or `foo.bar`. Resource names are effectively relative filesystem paths. e.g. `resource.txt` or `subdir/resource.txt`. The relative paths always use / as the directory separator, even on Windows.

`OxidizedFinder.get_resource_reader()` returns instances of *OxidizedResourceReader*. Each instance is bound to a specific Python package: that's how they are defined. When an *OxidizedResourceReader* receives the name of a resource, it performs a simple lookup in the global resources index. If the string key is found, it is used. Otherwise, it is assumed the resource doesn't exist.

The `OxidizedResourceReader.contents()` method will return a list of all keys in the internal resources index.

OxidizedResourceReader works the same way for in-memory and filesystem-relative resource locations because internally both use the same index of resources to drive execution: only the location of the resource content varies.

OxidizedResourceReader's implementation varies from the standard library filesystem-based implementation in the following ways:

- `OxidizedResourceReader.contents()` will return keys from the package's resources dictionary, not all the files in the same directory as the underlying Python package (the standard library uses `os.listdir()`). *OxidizedResourceReader* will therefore return resource names in sub-directories as long as those sub-directories aren't themselves Python packages.

- Resources must be explicitly registered with *OxidizedFinder* as such in order to be exposed via the resources API. By contrast, the filesystem-based importer - relying on `os.listdir()` - will expose all files in a directory as a resource. This includes `.py` files.

- `OxidizedResourceReader.is_resource()` will return `True` for resource names containing a slash. Contrast with Python's, which returns `False` (even though you can open a resource with `ResourceReader.open_resource()` for the same path). *OxidizedResourceReader*'s behavior is more consistent.

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).

Support for `__file__`

OxidizedFinder may or may not set the `__file__` attribute on loaded modules. See *__file__ and __cached__ Module Attributes* for details.

Therefore, Python code relying on the presence of `__file__` to derive paths to resource files may or may not work with `oxidized_importer`.

Code utilizing `__file__` for resource loading is highly encouraged to switch to the `importlib.resources` API. If this is not possible, you can change packaging settings to move the resource locations from in-memory to filesystem-relative, as `__file__` is set when loading modules from the filesystem.

Support for `pkg_resources`

`oxidized_importer` has support for working with `pkg_resources`.

`oxidized_importer` integration with `pkg_resources` is enabled by calling *register_pkg_resources()*.

If an *OxidizedFinder* imports the `pkg_resources` module, *register_pkg_resources()* may be called automatically.

The `pyembed` crate and PyOxidizer both have this functionality enabled by default and will likely have *OxidizedFinder* servicing the `pkg_resources` import. So there are likely no additional steps needed to enable `pkg_resources` support in these scenarios.

If you are using `oxidized_importer` as a standalone extension module in the context of a regular Python interpreter, you may need to call *register_pkg_resources()* manually to ensure integration is enabled.

To test whether integration is enabled, look for an *<class 'OxidizedFinder'>: <class 'OxidizedPkgResource-sProvider'>* entry in `pkg_resources._provider_factories`.

Distribution Resolving

OxidizedPathEntryFinder is a *path entry finder* type that responds to *paths* via the `sys.path_hooks` mechanism.

Distribution resolution support requires *OxidizedFinder.path_hook* to be registered on `sys.path_hook` and for *register_pkg_resources()* to have been called. If both these conditions are satisfied, `pkg_resources` should be able to find package distributions indexed by *OxidizedFinder* instances.

pkg_resources_find_distributions() is the callable registered with `pkg_resources` for resolving distributions. It respects path targeting and the `only` flag, per the behavior documented by `pkg_resources`.

Metadata and Resource Resolving

If `pkg_resources` derives the *provider* for any module loaded with *OxidizedFinder* or *OxidizedPathEntryFinder*, it should create an instance of *OxidizedPkgResourcesProvider* to resolve package metadata and resource info.

There are known behavior differences with *OxidizedPkgResourcesProvider* that may result in runtime errors. See that type's API documentation for more.

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:

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

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

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

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

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.:

```python
import importlib.resources

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

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

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:

```python
import importlib

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

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


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

(continues on next page)

```python
    # 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.

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.

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.

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:

```python
import os
import stat
import sys

import oxidized_importer

# Create a collector to help with managing resources.
collector = oxidized_importer.OxidizedResourceCollector(
    allowed_locations=["in-memory"]
)

# Add all known Python resources by scanning sys.path.
# Note: this will pull in the Python standard library and
# any other installed packages, which may not be desirable!
for path in sys.path:
    # Only directories can be scanned by oxidized_importer.
    if os.path.isdir(path):
        for resource in oxidized_importer.find_resources_in_path(path):
            collector.add_in_memory(resource)

# 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.
```

```python
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.

Loading Serialized Resources in Your Application

Now, from our *application* code, we need to load the resources and register the custom importer with Python:

```python
import os
import sys

import oxidized_importer

# Load those resources into an instance of our custom importer. This
# will read the index in the passed data structure and make all
# resources immediately available for importing.
finder = oxidized_importer.OxidizedFinder()
finder.index_file_memory_mapped("oxidized_resources")

# If the relative path of filesystem-based resources is not relative
# to the current executable (which is likely the ``python3`` executable),
# you'll need to set ``origin`` to the directory the resources are
# relative to.
finder = oxidized_importer.OxidizedFinder(
    relative_path_origin=os.path.dirname(os.path.abspath(__file__)),
)
finder.index_bytes(packed_data)

# 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!
```

`OxidizedZipFinder` Meta Path Finder

`oxidized_importer` contains a pure Rust implementation of a *meta path finder* that can load Python resources from zip files. Its goal is to be a compatible reimplementation of `zipimport.zipimporter` from the Python standard library.

Usage

Instances of *`OxidizedZipFinder`* are bound to zip archive data.

Instances can be constructed by calling *`OxidizedZipFinder.from_zip_data()`* or *`OxidizedZipFinder.from_path()`*.

`OxidizedZipFinder` is a *meta path finder* and instances should be registered on `sys.meta_path`. e.g.

```python
import os
import sys
import oxidized_importer

HERE = os.dirname(os.path.abspath(__file__))
zip_path = os.path.join(HERE, "archive.zip")

zip_importer = OxidizedZipFinder.from_path(zip_path)
sys.meta_path.insert(0, zip_importer)
```

Once an instance is registered on `sys.meta_path`, it will be consulted when an `import` is serviced by Python's importing mechanism.

Behavior

`OxidizedZipFinder` is similar to - but critically different from - the standard library `zipimport.zipimporter`.

`OxidizedZipFinder` is a *meta path finder*, not a *path entry finder*. This means instances are bound to `sys.meta_path` and not `sys.path_hooks`. Support for enabling use as a *path hook* is planned. The lack of `sys.path_hooks` support means this importer can't be used as a replacement for `zipimport.zipimporter`.

All I/O and zip reading in *`OxidizedZipFinder`* is implemented in Rust. Subtle differences in behavior as a result of zip parsing implementations could occur.

`OxidizedZipFinder` doesn't yet implement support for resource reading (e.g. the `importlib.abc.ResourceReader` interface). Only loading of `.py` and `.pyc` files is supported.

`OxidizedZipFinder` doesn't validate the header of `.pyc` files. If it sees a `.pyc` version of a module, its bytecode will be used as-is. (`zipimport.zipimporter` validates that the content in the `.pyc` matches expectations.)

Support for opening just sub-directories within zip files is not yet implemented.

Performance

OxidizedZipFinder should perform substantially better than `zipimport.zipimporter`.

A test importing the ~450 modules that constitute the Python standard library yielded the following results:

Environment	`zipimporter`	Us (memory)	Us (file)	`OxidizedFinder`
Ryzen 5950X Linux	205.07 ms	168.70 ms	184.74 ms	126.33 ms
Ryzen 5950X Windows	235.73 ms	147.14 ms	167.10 ms	140.21 ms

(The exact set of modules and Python versions were different between the environments so it isn't fair to compare numbers across environments: only within the same environment.)

Python API

See *OxidizedZipFinder* for the Python API documentation.

Common Issues

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:

```python
import oxidized_importer

collector = oxidized_importer.OxidizedResourceCollector(
    allowed_locations=["in-memory", "filesystem-relative"],
)

# Redirect extension modules to the filesystem and everything else to
# memory.
for resource in oxidized_importer(find_resources_in_path("/path/to/resources")):
    if isinstance(resource, oxidized_importer.PythonExtensionModule):
        collector.add_filesystem_relative("lib", resource)
    else:
        collector.add_in_memory(resource)
```

Resource Scanning Descends Into `site-packages`

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.

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>*.

API Reference

Module Level Functions

oxidized_importer.**decode_source**(*io_module*, *source_bytes*) → str

> Decodes Python source code `bytes` to a `str`.
>
> This is effectively a reimplementation of `importlib._bootstrap_external.decode_source()`

oxidized_importer.**find_resources_in_path**(*path*) → List

> This 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.

oxidized_importer.**register_pkg_resources**()

> Enables `pkg_resources` integration.
>
> This function effectively does the following:
>
> - Calls `pkg_resources.register_finder()` to map *OxidizedPathEntryFinder* to :py:func:pkg_resources_find_distributions`.
> - Calls `pkg_resources.register_load_type()` to map *OxidizedFinder* to *OxidizedPkgResourcesProvider*.
>
> It is safe to call this function multiple times, as behavior should be deterministic.

oxidized_importer.**pkg_resources_find_distributions**(*finder:* OxidizedPathEntryFinder, *path_item: str*, *only=false*) → list

> Resolve pkg_resources.Distribution instances given a *OxidizedPathEntryFinder* and search criteria.
>
> This function is what is registered with pkg_resources for distribution resolution and you likely don't need to call it directly.

The OxidizedFinder Class

class oxidized_importer.**OxidizedFinder**

> A meta path finder that resolves indexed resources. See See *OxidizedFinder Meta Path Finder* for more high-level documentation.
>
> This type implements 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.
>
> Instances have additional functionality beyond what is defined by importlib. This functionality allows you to construct, inspect, and manipulate instances.
>
> **multiprocessing_set_start_method**
>
> > (Opional[str]) Value to pass to multiprocessing.set_start_method() on import of multiprocessing module.
> >
> > None means the method won't be called.
>
> **origin**
>
> > (str) The path this instance is using as the anchor for relative path references.
>
> **path_hook_base_str**
>
> > (str) The base path that the path hook handler on this instance will respond to.
> >
> > This value is often the same as sys.executable but isn't guaranteed to be that exact value.
>
> **pkg_resources_import_auto_register**
>
> > (bool) Whether this instance will be registered via pkg_resources.register_finder() upon this instance importing the pkg_resources module.
>
> **__new__**(*cls*, *relative_path_origin: Optional[os.PathLike]*) → *OxidizedFinder*
>
> > Construct a new instance of *OxidizedFinder*.
> >
> > New instances of *OxidizedFinder* can be constructed like normal Python types:

```
finder = OxidizedFinder()
```

The constructor takes the following named arguments:

relative_path_origin
> A path-like object denoting the filesystem path that should be used as the *origin* value for relative path resources. Filesystem-based resources are stored as a relative path to an *anchor* value. This is that *anchor* value. If not specified, the directory of the current executable will be used.

See the python_packed_resources Rust crate for the specification of the binary data blob defining *packed resources data*.

Important: The *packed resources data* format is still evolving. It is recommended to use the same version of the `oxidized_importer` extension to produce and consume this data structure to ensure compatibility.

---

index_bytes(*data: bytes*) → None
> This method parses any bytes-like object and indexes the resources within.

index_file_memory_mapped(*path: pathlib.Path*) → None
> This method parses the given Path-like argument and indexes the resources within. Memory mapped I/O is used to read the file. Rust managed the memory map via the `memmap` crate: this does not use the Python interpreter's memory mapping code.

index_interpreter_builtins() → None
> This method indexes Python resources that are built-in to the Python interpreter itself. This indexes built-in extension modules and frozen modules.

index_interpreter_builtin_extension_modules() → None
> This method will index Python extension modules that are compiled into the Python interpreter itself.

index_interpreter_frozen_modules() → None
> This method will index Python modules whose bytecode is frozen into the Python interpreter itself.

indexed_resources() → List[*OxidizedResource*]
> This method returns a list of resources that are indexed by the instance. It allows Python code to inspect what the finder knows about.
>
> Any mutations to returned values are not reflected in the finder.
>
> See *OxidizedResource* for more on the returned type.

add_resource(*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:

```python
def register_module(finder, module_name, source):
    code = compile(source, module_name, "exec")
    bytecode = marshal.dumps(code)

    resource = OxidizedResource()
    resource.name = module_name
    resource.is_module = True
    resource.in_memory_bytecode = bytecode
    resource.in_memory_source = source

    finder.add_resource(resource)
```

add_resources(*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(*ignore_builtin=true*, *ignore_frozen=true*) → 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 *OxidizedFinder.__new__()*.

Arguments:

ignore_builtin (**bool**)
Whether to ignore `builtin` extension modules from the serialized data.

Default is `True`

ignore_frozen (**bool**)
Whether to ignore `frozen` extension modules from the serialized data.

Default is `True`.

Entries for *built-in* and *frozen* modules are ignored by default because they aren't portable, as they are compiled into the interpreter and aren't guaranteed to work from one Python interpreter to another. The serialized format does support expressing them. Use at your own risk.

path_hook(*path: Union[str, bytes, os.PathLike[AnyStr]]*) → *OxidizedPathEntryFinder*

Implements a *path hook* for obtaining a PathEntryFinder from a `sys.path` entry. See *Paths Hooks Compatibility* for details.

Raises `ImportError` if the given path isn't serviceable. The exception should have `.__cause__` set to an inner exception with more details on why the path was rejected.

The `OxidizedDistribution` Class

class oxidized_importer.**OxidizedDistribution**

Represents the metadata of a Python package. Comparable to `importlib.metadata.Distribution`. Instances of this type are emitted by `OxidizedFinder.find_distributions`.

from_name(*cls*, *name: str*) → *OxidizedDistribution*

Classmethod

Resolve the instance for the given package name.

discover(*cls*, ***kwargs*) → list[*OxidizedDistribution*]

> **Classmethod**
>
> Resolve instances for all known packages.

read_text(*filename*) → str

> Attempt to read metadata file given its filename.

property metadata

> **Type**
>> email.message.EmailMessage
>
> Return the parsed metadata for this distribution.

property name

> **Type**
>> str
>
> Return the Name metadata for this distribution package.

property _normalized_name

> **Type**
>> str
>
> Return the normalized version of the Name.

property version

> **Type**
>> str
>
> Return the Version metadata for this distribution package.

property entry_points

> Resolve entry points for this distribution package.

property files

> Not implemented. Always raises when called.

property requires

> Generated requirements specified for this distribution.

The `OxidizedResourceReader` Class

class oxidized_importer.**OxidizedResourceReader**

> importlib.abc.ResourceReader implementer for *OxidizedFinder*.

open_resource(*resource: str*)

resource_path(*resource: str*)

is_resource(*name: str*) → bool

contents() → list[str]

The `OxidizedPathEntryFinder` Class

class oxidized_importer.**OxidizedPathEntryFinder**

> A path entry finder that can find resources contained in an associated *OxidizedFinder* instance.
>
> Instances are created via *OxidizedFinder.path_hook*.
>
> Direct use of *OxidizedPathEntryFinder* is generally unnecessary: *OxidizedFinder* is the primary interface to the custom importer.
>
> See *Paths Hooks Compatibility* for more on path hook and path entry finder behavior in `oxidized_importer`.
>
> **find_spec**(*fullname: str*, *target: Optional[types.ModuleType] = None*) →
> > Optional[importlib.machinery.ModuleSpec]
>
> > Search for modules visible to the instance.
>
> **invalidate_caches**() → None
>
> > Invoke the same method on the *OxidizedFinder* instance with which the *OxidizedPathEntryFinder* instance was constructed.
>
> **iter_modules**(*prefix: str = ''*) → List[pkgutil.ModuleInfo]
>
> > Iterate over the visible modules. This method complies with `pkgutil.iter_modules`'s protocol.

The `OxidizedPkgResourcesProvider` Class

class oxidized_importer.**OxidizedPkgResourcesProvider**

> A pkg_resources.IMetadataProvider and pkg_resources.IResourceProvider enabling `pkg_resources` to access package metadata and resources.
>
> All members of the aforementioned interfaces are implemented. Divergence from `pkg_resources` defined behavior is documented next to the method.
>
> **has_metadata**(*name: str*) → bool
>
> **get_metadata**(*name: str*) → str
>
> **get_metadata_lines**(*name: str*) → List[str]
>
> > Returns a `list` instead of a generator.
>
> **metadata_isdir**(*name: str*) → bool
>
> **metadata_listdir**(*name: str*) → List[str]
>
> **run_script**(*script_name: str*, *namespace: Any*)
>
> > Always raises `NotImplementedError`.
> >
> > Please leave a comment in #384 if you would like this functionality implemented.
>
> **get_resource_filename**(*manager*, *resource_name: str*)
>
> > Always raises `NotImplementedError`.
> >
> > This behavior appears to be allowed given code in `pkg_resources`. However, it means that `pkg_resources.resource_filename()` will not work. Please leave a comment in #383 if you would like this functionality implemented.
>
> **get_resource_stream**(*manager*, *resource_name: str*) → io.BytesIO
>
> **get_resource_string**(*manager*, *resource_name: str*) → bytes

has_resource(*resource_name: str*) → bool

resource_isdir(*resource_name: str*) → bool

resource_listdir(*resource_name: str*) → List[str]

> Returns a `list` instead of a generator.

The `OxidizedResource` Class

class `oxidized_importer.`**OxidizedResource**

> Represents a *resource* that is indexed by a `OxidizedFinder` instance.
>
> Each instance represents a named entity with associated metadata and data. e.g. an instance can represent a Python module with associated source and bytecode.
>
> New instances can be constructed via `OxidizedResource()`. This will return an instance whose `name = ""` and all properties will be `None` or `false`.
>
> **is_module**
>
> > A `bool` indicating if this resource is a Python module. Python modules are backed by source or bytecode.
>
> **is_builtin_extension_module**
>
> > A `bool` indicating if this resource is a Python extension module built-in to the Python interpreter.
>
> **is_frozen_module**
>
> > A `bool` indicating if this resource is a Python module whose bytecode is frozen into the Python interpreter.
>
> **is_extension_module**
>
> > A `bool` indicating if this resource is a Python extension module.
>
> **is_shared_library**
>
> > A `bool` indicating if this resource is a shared library.
>
> **name**
>
> > The `str` name of the resource.
>
> **is_package**
>
> > A `bool` indicating if this resource is a Python package.
>
> **is_namespace_package**
>
> > A `bool` indicating if this resource is a Python namespace package.
>
> **in_memory_source**
>
> > `bytes` or `None` holding Python module source code that should be imported from memory.
>
> **in_memory_bytecode**
>
> > `bytes` or `None` holding Python module bytecode that should be imported from memory.
> >
> > This is raw Python bytecode, as produced from the `marshal` module. `.pyc` files have a header before this data that will need to be stripped should you want to move data from a `.pyc` file into this field.
>
> **in_memory_bytecode_opt1**
>
> > `bytes` or `None` holding Python module bytecode at optimization level 1 that should be imported from memory.
> >
> > This is raw Python bytecode, as produced from the `marshal` module. `.pyc` files have a header before this data that will need to be stripped should you want to move data from a `.pyc` file into this field.

in_memory_bytecode_opt2

> `bytes` or `None` holding Python module bytecode at optimization level 2 that should be imported from memory.
>
> This is raw Python bytecode, as produced from the `marshal` module. `.pyc` files have a header before this data that will need to be stripped should you want to move data from a `.pyc` file into this field.

in_memory_extension_module_shared_library

> `bytes` or `None` holding native machine code defining a Python extension module shared library that should be imported from memory.

in_memory_package_resources

> `dict[str, bytes]` or `None` holding resource files to make available to the `importlib.resources` APIs via in-memory data access. The `name` of this object will be a Python package name. Keys in this dict are virtual filenames under that package. Values are raw file data.

in_memory_distribution_resources

> `dict[str, bytes]` or `None` holding resource files to make available to the `importlib.metadata` API via in-memory data access. The `name` of this object will be a Python package name. Keys in this dict are virtual filenames. Values are raw file data.

in_memory_shared_library

> `bytes` or `None` holding a shared library that should be imported from memory.

shared_library_dependency_names

> `list[str]` or `None` holding the names of shared libraries that this resource depends on. If this resource defines a loadable shared library, this list can be used to express what other shared libraries it depends on.

relative_path_module_source

> `pathlib.Path` or `None` holding the relative path to Python module source that should be imported from the filesystem.

relative_path_module_bytecode

> `pathlib.Path` or `None` holding the relative path to Python module bytecode that should be imported from the filesystem.

relative_path_module_bytecode_opt1

> `pathlib.Path` or `None` holding the relative path to Python module bytecode at optimization level 1 that should be imported from the filesystem.

relative_path_module_bytecode_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.

The `OxidizedResourceCollector` Class

class `oxidized_importer.`**`OxidizedResourceCollector`**

>Provides functionality for turning instances of Python resource types into a collection of *OxidizedResource* for loading into an *OxidizedFinder* instance.

>**__new__**(*cls*, *allowed_locations: list[str]*)

>>Construct an instance by defining locations that resources can be loaded from.

>>The accepted string values are `in-memory` and `filesystem-relative`.

>**allowed_locations**

>>(list[str]) Exposes allowed locations where resources can be loaded from.

>**add_in_memory_resource**(*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*)

>>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**() → tuple[list[*OxidizedResource*], list[tuple[pathlib.Path, bytes, bool]]]

>>Takes all the resources collected so far and turns them into data structures to facilitate later use.

>>The first element in the returned 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.

The `OxidizedResourceReader` Class

class `oxidized_importer.`**`OxidizedResourceResource`**

>An implementation of importlib.abc.ResourceReader to facilitate resource reading from an *OxidizedFinder*.

>See *Support for ResourceReader* for more.

The `OxidizedZipFinder` Class

class `oxidized_importer.`**`OxidizedZipFinder`**

>A meta path finder that operates on zip files.

>This type attempts to be a pure Rust reimplementation of the Python standard library `zipimport.zipimporter` type.

>This type implements the following interfaces:

>- importlib.abc.MetaPathFinder
>- importlib.abc.Loader
>- importlib.abc.InspectLoader

from_zip_data(*cls*, *source:* *bytes*, *path: Union[bytes, str, pathlib.Path, None]* = *None*) → *OxidizedZipFinder*

> Construct an instance from zip archive data.
>
> The source argument can be any bytes-like object. A reference to the original Python object will be kept and zip I/O will be performed against the memory tracked by that object. It is possible to trigger an out-of-bounds memory read if the source object is mutated after being passed into this function.
>
> The `path` argument denotes the path to the zip archive. This path will be advertised in `__file__` attributes. If not defined, the path of the current executable will be used.

from_path(*cls*, *path: Union[bytes, str, pathlib.Path]*) → *OxidizedZipFinder*

> Construct an instance from a filesystem path.
>
> The source represents the path to a file containing zip archive data. The file will be opened using Rust file I/O. The content of the file will be read lazily.
>
> If you don't already have a copy of the zip data and the zip file will be immutable for the lifetime of the constructed instance, this method may yield better performance than opening the file, reading its content, and calling `OxidizedZipFinder.from_zip_data()` because it may incur less overall I/O.

The `PythonModuleSource` Class

class `oxidized_importer.PythonModuleSource`

> Represents Python module source code. e.g. a `.py` file.
>
> **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`) Whether this module is a Python package.

The `PythonModuleBytecode` Class

class `oxidized_importer.PythonModuleBytecode`

> Represents Python module bytecode. e.g. what a `.pyc` file holds (but without the header that a `.pyc` file has).
>
> **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.

The `PythonPackageResource` Class

class oxidized_importer.**PythonPackageResource**

> 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`.
>
> **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.

The `PythonPackageDistributionResource` Class

class oxidized_importer.**PythonPackageDistributionResource**

> 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`.
>
> **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.

The `PythonExtensionModule` Class

class oxidized_importer.**PythonExtensionModule**

> 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.

Note: Properties of this type are read-only.

Python Packed Resources

This project has defined a custom data format for storing resources useful to the execution of a Python interpreter. We call this data format *Python packed resources*.

The way it works is that some producer collects resources required by a Python interpreter. These resources include Python module source and bytecode, non-module resource/data files, extension modules, and shared libraries. Metadata about these resources and sometimes the raw resource data itself is serialized to a binary data structure.

At Python interpreter run time, an instance of the `OxidizedFinder` *meta path finder* parses this data structure and uses it to power Python module importing.

This functionality is similar to using a `.zip` file for holding Python modules. However, the *Python packed resources* data structure is far more advanced.

Implementation

The canonical implementation of the writer and parser of this data structure lives in the `python-packed-resources` Rust crate. The canonical home of this crate is https://github.com/indygreg/PyOxidizer/tree/main/python-packed-resources.

This crate is published to crates.io at https://crates.io/crates/python-packed-resources.

The `oxidized_importer` Rust crate / Python extension defines the `OxidizedFinder` Python class for using this data structure to power importing. That extension also exposes APIs to interact with instances of the data structure.

Concepts

The data structure is logically an iterable of *resources*.

A *resource* is a sparse collection of *attributes* or *fields*.

Each *attribute* describes behavior of the *resource* or defines data for that resource. For example, there are *attributes* that denote the type of a resource. A *Python module resource* might have an attribute holding its Python sourcecode or bytecode.

In Rust speak, a *resource* is a `struct` and *attributes* are fields in that `struct`. Many fields are `Option<T>` because they are optional and not always defined.

Serialization Format

High-Level Overview

The serialization format consists of:

- A *global header* containing identifying magic and describing the overall payload.
- An index describing data for each distinct *attribute* type. This is called the *blob index*.
- An index describing each resource and its attributes. This is called the *resources index*.
- A series of sections holding data for each distinct *attribute* type. We call these *blob sections*.

All integers are little-endian.

Global Header

The first 8 bytes of the data structure are a magic header identifying the content as our data structure and the version of it. The first 7 bytes are `pyembed` and the following 1 byte denotes a version. Semantics of each version are denoted in sections below.

The first 13 bytes after the magic header describe the *blob* and *resource* indices as follows:

- A `u8` denoting the number of blob sections, `blob_sections_count`.
- A `u32` denoting the length of the blob index, `blob_index_length`.
- A `u32` denoting the total number of resources in this data, `resources_count`.
- A `u32` denoting the length of the resources index, `resources_index_length`.

Blob Index

Following the *global header* is the *blob index*, which describes the *blob sections* present later in the data structure.

Each entry in the *blob index* logically consists of a set of fields defining metadata about each *blob section*. This is encoded by a *start of entry* `u8` marker followed by N `u8` field type values and their corresponding metadata, followed by an *end of entry* `u8` marker.

The *blob index* is terminated by an *end of index* `u8` marker.

The total number of bytes in the *blob index* including the *end of index* marker should be `blob_index_length`.

The *blob index* allows attributing a sparse set of metadata with every blob section entry. The type of metadata being conveyed is defined by a `u8`. Some field types have additional metadata following that field.

The various field types and their semantics follow.

0x00

    End of index. This field indicates that there are no more blob index entries and we've reached the end of the *blob index*.

0x01

    Start of blob section entry. Encountering this value signals the beginning of a new blob section. From a specification standpoint, this isn't strictly required. But it helps ensure parser state.

0xff

    End of blob section entry. Encountering this value signals the end of the current blob section definition. The next encountered `u8` in the index should be `0x01` to denote a new entry or `0x00` to denote end of index.

0x02

    Resource field type. This field defines which resource field this blob section is holding data for. A `u8` following this one will contain the resource field type value (see section below).

0x03

    Raw payload length. This field defines the raw length in bytes of the blob section in the payload. The `u64` containing that length will immediately follow this `u8`.

0x04

    Interior padding mechanism. This field defines interior padding between elements in the blob section. Following this `u8` is another `u8` denoting the padding mechanism.

    `0x01` indicates no padding. `0x02` indicates NULL padding (a `0x00` between elements).

    If not present, *no padding* is assumed. If the payload data logically consists of discrete resources (e.g. Python package resource files), then padding applies to these sub-elements as well.

For example, a *blob index* byte sequence of `0x01 0x02 0x03 0x03 0x0000000000000042 0x04 0x01 0xff 0x00` would be decoded as:

- `0x01` - Start of blob section entry.

- `0x02 0x03` - Resource field type definition (`0x02`) for field `0x03`.

- `0x03 0x0000000000000042` - Blob section length (`0x03`) of `0x42` bytes long.

- `0x04 0x01` - Interior padding in blob section (`0x04`) is defined as no padding (`0x01`).

- `0xff` - End of blob section entry.

- `0x00` - End of index.

Resources Index

Following the *blob index* is the *resources index*.

Each entry in this index defines a sparse set of metadata describing a single resource.

Entries are composed of a series of `u8` identifying pieces of metadata, followed by field-specific supplementary descriptions.

The following `u8` fields and their behavior/payloads are as follows:

0x00

> End of index. Special type to denote the end of an index.

0x01

> Start of resource entry. Signals the beginning of a new resource. From a specification standpoint this isn't strictly required. But it helps ensure parser state.

0x02

> Previously held the resource *flavor*. This field is deprecated in version 2 in favor of the individual fields expressing presence of a resource type. (See fields starting at `0x16`.)

0xff

> End of resource entry. The next encountered `u8` in the index should be an *end of index* or *start of resource* marker.

0x03

> Resource name. A `u16` denoting the length in bytes of the resource name immediately follows this byte. The resource name *must* be valid UTF-8.

0x04

> Package flag. If encountered, the resource is identified as a Python package.

0x05

> Namespace package flag. If encountered, the resource is identified as a Python *namespace package*.

0x06

> In-memory Python module source code. A `u32` denoting the length in bytes of the module's source code immediately follows this byte.

0x07

> In-memory Python module bytecode. A `u32` denoting the length in bytes of the module's bytecode immediately follows this byte.

0x08

> In-memory Python module optimized level 1 bytecode. A `u32` denoting the length in bytes of the module's optimization level 1 bytecode immediately follows this byte.

0x09

In-memory Python module optimized level 2 bytecode. Same as previous, except for bytecode optimization level 2.

0x0a

In-memory Python extension module shared library. A `u32` denoting the length in bytes of the extension module's machine code immediately follows this byte.

0x0b

In-memory Python resources data. If encountered, the module/package contains non-module resources files and the number of resources is contained in a `u32` that immediately follows. Following this `u32` is an array of `(u16, u64)` denoting the resource name and payload size for each resource in this package.

0x0c

In-memory Python distribution resource. Defines resources accessed from `importlib.metadata` APIs. If encountered, the module/package contains distribution metadata describing the package. The number of files being described is contained in a `u32` that immediately follows this byte. Following this `u32` is an array of `(u16, u64)` denoting the distribution file name and payload size for each virtual file in this distribution.

0x0d

In-memory shared library. If set, this resource is a shared library and not a Python module. The resource name field is the name of this shared library, with file extension (as it would appear in a dynamic binary's loader metadata to indicate a library dependency). A `u64` denoting the length in bytes of the shared library data follows. This shared library should be loaded from memory.

0x0e

Shared library dependency names. This field indicates the names of shared libraries that this entity depends on. The number of library names is contained in a `u16` that immediately follows this byte. Following this `u16` is an array of `u16` denoting the length of the library name for each shared library dependency. Each described shared library dependency may or may not be described by other entries in this data structure.

0x0f

Relative filesystem path to Python module source code. A `u32` holding the length in bytes of a filesystem path encoded in the platform-native file path encoding follows. The source code for a Python module will be read from a file at this path.

0x10

Relative filesystem path to Python module bytecode. Similar to the previous except the filesystem path holds Python module bytecode.

0x11

Relative filesystem path to Python module bytecode at optimization level 1. Similar to the previous except for what is being pointed to.

0x12

Relative filesystem path to Python module bytecode at optimization level 2. Similar to the previous except for what is being pointed to.

0x13

Relative filesystem path to Python extension module shared library. Similar to the previous except the file holds a Python extension module loadable as a shared library.

0x14

Relative filesystem path to Python package resources. The number of resources is contained in a `u32` that immediately follows. Following this `u32` is an array of `(u16, u32)` denoting the resource name and filesystem path to each resource in this package.

0x15

Relative filesystem path to Python distribution resources.

Defines resources accessed from `importlib.metadata` APIs. If encountered, the module/package contains distribution metadata describing the package. The number of files being described is contained in a `u32` that immediately follows this byte. Following this `u32` is an array of `(u16, u32)` denoting the distribution file name and filesystem path to that distribution file.

0x16

Is Python module flag. If set, this resource contains data for an importable Python module or package. Resource data is associated with Python packages and is covered by this type.

0x17

Is builtin extension module flag. This type represents a Python extension module that is built in (compiled into) the interpreter itself or is otherwise made available to the interpreter via `PyImport_Inittab` such that it should be imported with the *builtin* importer.

0x18

Is frozen Python module flag. This type represents a Python module whose bytecode is *frozen* and made available to the Python interpreter via the `PyImport_FrozenModules` array and should be imported with the *frozen* importer.

0x19

Is Python extension flag. This type represents a compiled Python extension. Extensions have specific requirements around how they are to be loaded and are differentiated from regular Python modules.

0x1a

Is shared library flag. This type represents a shared library that can be loaded into a process.

0x1b

Is utf-8 filename data flag. This type represents an arbitrary filename. The resource name is a UTF-8 encoded filename of the file this resource represents. The file's data is either embedded in memory or referred to via a relative path reference.

0x1c

File data is executable flag.

If set, the arbitrary file this resource tracks should be marked as executable.

0x1d

Embedded file data.

If present, the resource should be a file resource and this field holds its raw file data in memory.

A `u64` containing the length of the embedded data follows this field.

0x1e

UTF-8 relative path file data.

If present, the resource should be a file resource and this field defines the relative path containing that file's data. The relative path filename is UTF-8 encoded.

A `u32` denoting the length of the UTF-8 relative path (in bytes) follows.

Blob Sections

Following the *resources index* is blob data.

Blob data is logically composed of different sections holding data for different fields for different resources. But there is no internal structure or separators: all the individual blobs are just laid out next to each other. The *resources index* for a given field will describe where in a blob section a particular value occurs.

pyembed\x01 Format

The initially released/formalized packed resources data format.

Supports resource field types up to and including `0x15`.

pyembed\x02 Format

Version 2 of the packed resources data format.

This version introduces field type values `0x16` to `0x1a`. The resource flavor field type (`0x02`) is deprecated and the individual field types denoting resource types should be used instead.

(PyOxidizer removed run-time code looking at field type `0x02` when this format was introduced.)

pyembed\x03 Format

Version 3 of the packed resources data format.

This version introduces field type values `0x1b` to `0x1e`.

These fields provide the ability for a resource to identify itself as an arbitrary filename and for the arbitrary file data to be embedded within the data structure or referenced via a relative path.

Unlike previous fields that use OS-native encoding of filesystem paths (`[u8]` on POSIX and `[u16]` on Windows), the paths for these new fields use UTF-8. This can't represent all valid paths on all platforms. But it is portable and works for most paths encountered in the wild.

Design Considerations

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

Performance is a significant consideration. We want everything to be as fast as possible. Possible dimensions influencing performance include parse time, payload size, and I/O access patterns.

The payload is designed such that the *index* data is at the beginning so a reader only has to read a contiguous slice of data to fully understand the data within. This is in opposition to jumping around the entire data structure to extract metadata of the data within. This means that we only need to page in a fraction of the total backing data structure in order to initialize our custom importer. In addition, the index data is read sequentially. Sequential I/O should always be faster than random access I/O.

x86 is little endian, so we use little endian integers so we don't need to waste cycles on endian transformation.

We store all data for the same field next to each other in the data structure. This is in opposition to say packing all of resource A's data then resource B's, etc. We do this to help maximize locality for similar data. This can help with performance because often the same field for multiple resources is accessed together. e.g. an importer will access a

bunch of module bytecode entries at the same time. This locality helps minimize the number of pages that must be read. Locality can also help yield higher compression ratios.

Everything is designed to facilitate a reader leveraging 0-copy. If a reader has the data structure in memory, we don't want to require it to copy memory in order to reference entries. In Rust speak, we should be able to hold `&[u8]` references everywhere.

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

Potential Future Features

This data structure is robust enough to be used by PyOxidizer to power importing of every Python module used by a Python interpreter. However, there are various aspects that could be improved.

Compression

A potential area for optimization is use of general compression. Various fields should compress well - either in streaming mode or by utilizing compression dictionaries. Compression would undermine 0-copy, of course. But in environments where we want to optimize for size, it could be desirable.

Project History

Changelog

0.6.0

Released on June 6, 2022.

- Added missing API docs for *OxidizedDistribution*.
- `OxidizedDistribution.metadata` now returns an `importlib.metadata._adapters.Message` instance on Python 3.10+.
- `OxidizedDistribution.entry_points` now calls `importlib.metadata.EntryPoints._from_text_for` on Python 3.10+. Previously, the implementation of this method didn't work properly on 3.10+.
- Added `name` property to *OxidizedDistribution*.
- Added `_normalized_name` property to *OxidizedDistribution*.
- PyO3 Rust crate upgraded to 0.16.5. This gets us better compatibility with Python 3.10.

1.3 pyembed

A Rust library crate to control embedded Python interpreters in Rust applications. The `pyembed` crate enhances the functionality of embedded Python interpreters by implementing additional features such as integration with *oxidized_importer*, easy configuration of alternate memory allocators, automatic terminfo database resolution, and more.

`pyembed` is usable as a standalone Rust crate and can be used by any Rust project embedding Python to abstract over some of the complexities with embedding a Python interpreter.

1.3.1 The `pyembed` Rust Crate

The `pyembed` Rust crate facilitates the control of an embedded Python interpreter.

The crate provides an API for instantiating and controlling an embedded Python interpreter. It also defines a custom *meta path importer* that can be used to import Python resources (such as module bytecode) from memory.

The crate is developed alongside the PyOxidizer project. However, it is a generic crate and can be used outside the context of PyOxidizer.

The `pyembed` crate is published to crates.io and its Rust documentation is available at https://docs.rs/pyembed.

Building

A design goal of `pyembed` is for it to exist like normal Rust crates. However, because `pyembed` needs to link against Python, there are some special requirements.

Configuring PyO3

`pyembed` pulls in a Python library link dependency via the `pyo3` crate. At `cargo build` time, `pyo3` (technically `pyo3-build-config`) will attempt to locate a `libpython` to link against. This behavior is documented at https://pyo3.rs/v0.15.0/building_and_distribution.html.

Generally speaking, all the caveats documented by `pyo3` apply to `pyembed` as well, since this project is a glorified, value-adding wrapper around `pyo3`.

The short version of the PyO3 documentation is as follows:

- By default the build script will look for an executable `python` on `PATH` and attempt to derive its build configuration from it.

- You can point it at a specific Python executable by setting the `PYO3_PYTHON` environment variable.

- For more advanced use cases (including cross-compiling), you can create a custom config file to configure the `pyo3-build-config` crate and point to it via the `PYO3_CONFIG_FILE` environment variable.

Generally speaking, if you are able to build the `pyo3` crate in isolation, you should be able to build the `pyembed` crate. To customize how the `pyembed` crate links against Python, use `pyo3`'s mechanisms for doing that.

Controlling Python from Rust Code

Initializing a Python Interpreter

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

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

Using a Python Interpreter

Once you've constructed a `pyembed::MainPythonInterpreter` instance, you can obtain a `pyo3::Python` instance via `.with_gil()` and then use it:

```rust
fn do_it(interpreter: &MainPythonInterpreter) -> {
    interpreter.with_gil(|py| {
        match py.eval("print('hello, world')") {
            Ok(_) => print("python code executed successfully"),
            Err(e) => print("python error: {:?}", e),
        }
    });

}
```

Since CPython's API relies on static variables (sadly), if you really wanted to, you could call out to CPython C APIs directly (probably via the bindings in the `pyo3` 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.

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.

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!

Adding Extension Modules At Run-Time

A Python extension module is effectively a callable function defined in a library somewhere.

The `pyembed` crate supports registering Python extension modules multiple ways.

Statically Linked Extension Modules

You can inform the `pyembed` crate about the existence of additional Python extension modules which are statically linked into the binary.

To do this, you will need to populate the `extra_extension_modules` field of the `OxidizedPythonInterpreterConfig` Rust struct used to construct the Python interpreter. Simply add an entry defining the extension module's `import` name and a pointer to its C initialization function (often named `PyInit_<name>`. e.g. if you are defining the extension module `foo`, the initialization function would be `PyInit_foo` by convention.

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

Dynamically Linked Extension Modules

If you have an extension module provided as a shared library (this is typically how Python extension modules work), it will be possible to load this extension module provided that the Python interpreter supports loading dynamically linked Python extension modules.

There is not yet an explicit Rust API for loading additional dynamically linked extension modules. It is theoretically possible to add an entry to the parsed embedded resources data structure. The path of least resistance is likely to enable the standard filesystem importer and put your shared library extension module somewhere on Python's `sys.path`. (This is how extension modules are typically loaded.)

Python Interpreter Configuration Data Structures

This document describes the data structures for configuring the behavior of a Python interpreter. The data structures are consumed by the `pyembed` Rust crate. All type names should correspond to public symbols in the `pyembed` crate.

This documentation is auto-generated from the inline documentation in Rust source files. Some formatting has been lost as part of the conversion. See https://docs.rs/pyembed/ for the native Rust API documentation

Structs:

- *OxidizedPythonInterpreterConfig*
- *PythonInterpreterConfig*

Enums:

- *MemoryAllocatorBackend*
- *PythonInterpreterProfile*
- *Allocator*
- *BytecodeOptimizationLevel*
- *BytesWarning*
- *CheckHashPycsMode*
- *CoerceCLocale*
- *MultiprocessingStartMethod*
- *TerminfoResolution*

`OxidizedPythonInterpreterConfig` Struct

Configuration for a Python interpreter.

This type is used to create a `crate::MainPythonInterpreter`, which manages a Python interpreter running in the current process.

This type wraps a `PythonInterpreterConfig`, which is an abstraction over the low-level C structs (`PyPreConfig` and `PyConfig`) used as part of Python's C initialization API. In addition to this data structure, the fields on this type facilitate control of additional features provided by this crate.

The `PythonInterpreterConfig` has a single non-optional field: `PythonInterpreterConfig::profile`. This defines the defaults for various fields of the `PyPreConfig` and `PyConfig` C structs. See https://docs.python.org/3/c-api/init_config.html#isolated-configuration for more.

When this type is converted to `PyPreConfig` and `PyConfig`, instances of these C structs are created from the specified profile. e.g. by calling `PyPreConfig_InitPythonConfig()`, `PyPreConfig_InitIsolatedConfig`, `PyConfig_InitPythonConfig`, and `PyConfig_InitIsolatedConfig`. Then for each field in `PyPreConfig` and `PyConfig`, if a corresponding field on `PythonInterpreterConfig` is `Some`, then the `PyPreConfig` or `PyConfig` field will be updated accordingly.

During interpreter initialization, `Self::resolve()` is called to resolve/finalize any missing values and convert the instance into a `ResolvedOxidizedPythonInterpreterConfig`. It is this type that is used to produce a `PyPreConfig` and `PyConfig`, which are used to initialize the Python interpreter.

Some fields on this type are redundant or conflict with those on `PythonInterpreterConfig`. Read the documentation of each field to understand how they interact. Since `PythonInterpreterConfig` is defined in a different crate, its docs are not aware of the existence of this crate/type.

This struct implements `Deserialize` and `Serialize` and therefore can be serialized to any format supported by the `serde` crate. This feature is used by `pyoxy` to allow YAML-based configuration of Python interpreters.

exe Field

The path of the currently executing executable.

This value will always be `Some` on `ResolvedOxidizedPythonInterpreterConfig` instances.

Default value: `None`.

`Self::resolve()` behavior: sets to `std::env::current_exe()` if not set. Will canonicalize the final path, which may entail filesystem I/O.

Type: `Option<PathBuf>`

origin Field

The filesystem path from which relative paths will be interpreted.

This value will always be `Some` on `ResolvedOxidizedPythonInterpreterConfig` instances.

Default value: `None`.

`Self::resolve()` behavior: sets to `Self::exe.parent()` if not set.

Type: `Option<PathBuf>`

interpreter_config Field

Low-level configuration of Python interpreter.

Default value: `PythonInterpreterConfig::default()` with `PythonInterpreterConfig::profile` always set to `PythonInterpreterProfile::Python`.

`Self::resolve()` behavior: most fields are copied verbatim. `PythonInterpreterConfig::module_search_paths` entries have the special token `$ORIGIN` expanded to the resolved value of `Self::origin`.

Type: `PythonInterpreterConfig`

`allocator_backend` Field

Memory allocator backend to use.

Default value: `MemoryAllocatorBackend::Default`.

Interpreter initialization behavior: after `Py_PreInitialize()` is called, `crate::pyalloc::PythonMemoryAllocator::from_back` is called. If this resolves to a `crate::pyalloc::PythonMemoryAllocator`, that allocator will be installed as per `Self::allocator_raw`, `Self::allocator_mem`, `Self::allocator_obj`, and `Self::allocator_pymalloc_arena`. If a custom allocator backend is defined but all the `allocator_*` flags are `false`, the allocator won't be used.

Type: `MemoryAllocatorBackend`

`allocator_raw` Field

Whether to install the custom allocator for the `raw` memory domain.

See https://docs.python.org/3/c-api/memory.html for documentation on how Python memory allocator domains work.

Default value: `true`

Interpreter initialization behavior: controls whether `Self::allocator_backend` is used for the `raw` memory domain.

Has no effect if `Self::allocator_backend` is `MemoryAllocatorBackend::Default`.

Type: `bool`

`allocator_mem` Field

Whether to install the custom allocator for the `mem` memory domain.

See https://docs.python.org/3/c-api/memory.html for documentation on how Python memory allocator domains work.

Default value: `false`

Interpreter initialization behavior: controls whether `Self::allocator_backend` is used for the `mem` memory domain.

Has no effect if `Self::allocator_backend` is `MemoryAllocatorBackend::Default`.

Type: `bool`

`allocator_obj` Field

Whether to install the custom allocator for the `obj` memory domain.

See https://docs.python.org/3/c-api/memory.html for documentation on how Python memory allocator domains work.

Default value: `false`

Interpreter initialization behavior: controls whether `Self::allocator_backend` is used for the `obj` memory domain.

Has no effect if `Self::allocator_backend` is `MemoryAllocatorBackend::Default`.

Type: `bool`

`allocator_pymalloc_arena` Field

Whether to install the custom allocator for the `pymalloc` arena allocator.

See https://docs.python.org/3/c-api/memory.html for documentation on how Python memory allocation works.

Default value: `false`

Interpreter initialization behavior: controls whether `Self::allocator_backend` is used for the `pymalloc` arena allocator.

This setting requires the `pymalloc` allocator to be used for the `mem` or `obj` domains (`allocator_mem = false` and `allocator_obj = false` - this is the default behavior) and for `Self::allocator_backend` to not be `MemoryAllocatorBackend::Default`.

Type: `bool`

`allocator_debug` Field

Whether to set up Python allocator debug hooks to detect memory bugs.

Default value: `false`

Interpreter initialization behavior: triggers the calling of `PyMem_SetupDebugHooks()` after custom allocators are installed.

This setting can be used with or without custom memory allocators (see other `allocator_*` fields).

Type: `bool`

`set_missing_path_configuration` Field

Whether to automatically set missing "path configuration" fields.

If `true`, various path configuration (https://docs.python.org/3/c-api/init_config.html#path-configuration) fields will be set automatically if their corresponding `.interpreter_config` fields are `None`. For example, `program_name` will be set to the current executable and `home` will be set to the executable's directory.

If this is `false`, the default path configuration built into libpython is used.

Setting this to `false` likely enables isolated interpreters to be used with "external" Python installs. If this is `true`, the default isolated configuration expects files like the Python standard library to be installed relative to the current executable. You will need to either ensure these files are present, define `packed_resources`, and/or set `.interpreter_config.module_search_paths` to ensure the interpreter can find the Python standard library, otherwise the interpreter will fail to start.

Without this set or corresponding `.interpreter_config` fields set, you may also get run-time errors like `Could not find platform independent libraries <prefix>` or `Consider setting $PYTHONHOME to <prefix>[:<exec_prefix>]`. If you see these errors, it means the automatic path config resolutions built into libpython didn't work because the run-time layout didn't match the build-time configuration.

Default value: `true`

Type: `bool`

`oxidized_importer` Field

Whether to install `oxidized_importer` during interpreter initialization.

If `true`, `oxidized_importer` will be imported during interpreter initialization and an instance of `oxidized_importer.OxidizedFinder` will be installed on `sys.meta_path` as the first element.

If `Self::packed_resources` are defined, they will be loaded into the `OxidizedFinder`.

If `Self::filesystem_importer` is `true`, its *path hook* will be registered on `sys.path_hooks` so `PathFinder` (the standard filesystem based importer) and `pkgutil` can use it.

Default value: `false`

Interpreter initialization behavior: See above.

Type: `bool`

`filesystem_importer` Field

Whether to install the path-based finder.

Controls whether to install the Python standard library `PathFinder` meta path finder (this is the meta path finder that loads Python modules and resources from the filesystem).

Also controls whether to add `OxidizedFinder`'s path hook to `sys.path_hooks`.

Due to lack of control over low-level Python interpreter initialization, the standard library `PathFinder` will be registered on `sys.meta_path` and `sys.path_hooks` for a brief moment when the interpreter is initialized. If `sys.path` contains valid entries that would be serviced by this finder and `oxidized_importer` isn't able to service imports, it is possible for the path-based finder to be used to import some Python modules needed to initialize the Python interpreter. In many cases, this behavior is harmless. In all cases, the path-based importer is removed after Python interpreter initialization, so future imports won't be serviced by this path-based importer if it is disabled by this flag.

Default value: `true`

Interpreter initialization behavior: If false, path-based finders are removed from `sys.meta_path` and `sys. path_hooks` is cleared.

Type: `bool`

`packed_resources` Field

References to packed resources data.

The format of the data is defined by the `python-packed-resources` crate. The data will be parsed as part of initializing the custom meta path importer during interpreter initialization when `oxidized_importer=true`. If `oxidized_importer=false`, this field is ignored.

If paths are relative, that will be evaluated relative to the process's current working directory following the operating system's standard path expansion behavior.

Default value: `vec![]`

`Self::resolve()` behavior: `PackedResourcesSource::MemoryMappedPath` members have the special string `$ORIGIN` expanded to the string value that `Self::origin` resolves to.

This field is ignored during serialization.

Type: `Vec<PackedResourcesSource>`

`extra_extension_modules` Field

Extra extension modules to make available to the interpreter.

The values will effectively be passed to `PyImport_ExtendInitTab()`.

Default value: `None`

Interpreter initialization behavior: `PyImport_Inittab` will be extended with entries from this list. This makes the extensions available as built-in extension modules.

This field is ignored during serialization.

Type: `Option<Vec<ExtensionModule>>`

`argv` Field

Command line arguments to initialize `sys.argv` with.

Default value: `None`

`Self::resolve()` behavior: `Some` value is used if set. Otherwise `PythonInterpreterConfig::argv` is used if set. Otherwise `std::env::args_os()` is called.

Interpreter initialization behavior: the resolved `Some` value is used to populate `PyConfig.argv`.

Type: `Option<Vec<OsString>>`

`argvb` Field

Whether to set `sys.argvb` with bytes versions of process arguments.

On Windows, bytes will be UTF-16. On POSIX, bytes will be raw char* values passed to `int main()`.

Enabling this feature will give Python applications access to the raw `bytes` values of raw argument data passed into the executable. The single or double width bytes nature of the data is preserved.

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

Default value: `false`

Interpreter initialization behavior: `sys.argvb` will be set to a `list[bytes]`. `sys.argv` and `sys.argvb` should have the same number of elements.

Type: `bool`

`multiprocessing_auto_dispatch` Field

Automatically detect and run in `multiprocessing` mode.

If set, `crate::MainPythonInterpreter::run()` will detect when the invoked interpreter looks like it is supposed to be a `multiprocessing` worker and will automatically call into the `multiprocessing` module instead of running the configured code.

Enabling this has the same effect as calling `multiprocessing.freeze_support()` in your application code's `__main__` and replaces the need to do so.

Default value: `true`

Type: `bool`

`multiprocessing_start_method` Field

Controls how to call `multiprocessing.set_start_method()`.

Default value: `MultiprocessingStartMethod::Auto`

Interpreter initialization behavior: if `Self::oxidized_importer` is `true`, the `OxidizedImporter` will be taught to call `multiprocessing.set_start_method()` when `multiprocessing` is imported. If `false`, this value has no effect.

Type: `MultiprocessingStartMethod`

`sys_frozen` Field

Whether to set sys.frozen=True.

Setting this will enable Python to emulate "frozen" binaries, such as those used by PyInstaller.

Default value: `false`

Interpreter initialization behavior: If `true`, `sys.frozen = True`. If `false`, `sys.frozen` is not defined.

Type: `bool`

`sys_meipass` Field

Whether to set sys._MEIPASS to the directory of the executable.

Setting this will enable Python to emulate PyInstaller's behavior of setting this attribute. This could potentially help with self-contained application compatibility by masquerading as PyInstaller and causing code to activate *PyInstaller mode*.

Default value: `false`

Interpreter initialization behavior: If `true`, `sys._MEIPASS` will be set to a `str` holding the value of `Self::origin`. If `false`, `sys._MEIPASS` will not be defined.

Type: `bool`

`terminfo_resolution` Field

How to resolve the `terminfo` database.

Default value: `TerminfoResolution::Dynamic`

Interpreter initialization behavior: the `TERMINFO_DIRS` environment variable may be set for this process depending on what `TerminfoResolution` instructs to do.

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

Type: `TerminfoResolution`

`tcl_library` Field

Path to use to define the `TCL_LIBRARY` environment variable.

This directory should contain an `init.tcl` file. It is commonly a directory named `tclX.Y`. e.g. `tcl8.6`.

Default value: `None`

`Self::resolve()` behavior: the token `$ORIGIN` is expanded to the resolved value of `Self::origin`.

Interpreter initialization behavior: if set, the `TCL_LIBRARY` environment variable will be set for the current process.

Type: `Option<PathBuf>`

`write_modules_directory_env` Field

Environment variable holding the directory to write a loaded modules file.

If this value is set and the environment it refers to is set, on interpreter shutdown, we will write a `modules-<random>` file to the directory specified containing a `\n` delimited list of modules loaded in `sys.modules`.

This setting is useful to record which modules are loaded during the execution of a Python interpreter.

Default value: `None`

Type: `Option<String>`

`PythonInterpreterConfig` Struct

Holds configuration of a Python interpreter.

This struct holds fields that are exposed by `PyPreConfig` and `PyConfig` in the CPython API.

Other than the profile (which is used to initialize instances of `PyPreConfig` and `PyConfig`), all fields are optional. Only fields with `Some(T)` will be updated from the defaults.

`profile` Field

Profile to use to initialize pre-config and config state of interpreter.

Type: `PythonInterpreterProfile`

`allocator` Field

Name of the memory allocator.

See https://docs.python.org/3/c-api/init_config.html#c.PyPreConfig.allocator.

Type: `Option<Allocator>`

configure_locale Field

Whether to set the LC_CTYPE locale to the user preferred locale.

See https://docs.python.org/3/c-api/init_config.html#c.PyPreConfig.configure_locale.

Type: `Option<bool>`

coerce_c_locale Field

How to coerce the locale settings.

See https://docs.python.org/3/c-api/init_config.html#c.PyPreConfig.coerce_c_locale.

Type: `Option<CoerceCLocale>`

coerce_c_locale_warn Field

Whether to emit a warning if the C locale is coerced.

See https://docs.python.org/3/c-api/init_config.html#c.PyPreConfig.coerce_c_locale_warn.

Type: `Option<bool>`

development_mode Field

Whether to enable Python development mode.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.dev_mode.

Type: `Option<bool>`

isolated Field

Isolated mode.

See https://docs.python.org/3/c-api/init_config.html#c.PyPreConfig.isolated.

Type: `Option<bool>`

legacy_windows_fs_encoding Field

Whether to use legacy filesystem encodings on Windows.

See https://docs.python.org/3/c-api/init_config.html#c.PyPreConfig.legacy_windows_fs_encoding.

Type: `Option<bool>`

`parse_argv` **Field**

Whether argv should be parsed the way `python` parses them.

See https://docs.python.org/3/c-api/init_config.html#c.PyPreConfig.parse_argv.

Type: `Option<bool>`

`use_environment` **Field**

Whether environment variables are read to control the interpreter configuration.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.use_environment.

Type: `Option<bool>`

`utf8_mode` **Field**

Controls Python UTF-8 mode.

See https://docs.python.org/3/c-api/init_config.html#c.PyPreConfig.utf8_mode.

Type: `Option<bool>`

`argv` **Field**

Command line arguments.

These will become `sys.argv`.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.argv.

Type: `Option<Vec<OsString>>`

`base_exec_prefix` **Field**

Controls `sys.base_exec_prefix`.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.base_exec_prefix.

Type: `Option<PathBuf>`

`base_executable` **Field**

Controls `sys._base_executable`.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.base_executable.

Type: `Option<PathBuf>`

`base_prefix` Field

Controls `sys.base_prefix`.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.base_prefix.

Type: `Option<PathBuf>`

`buffered_stdio` Field

Controls buffering on `stdout` and `stderr`.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.buffered_stdio.

Type: `Option<bool>`

`bytes_warning` Field

Controls warnings/errors for some bytes type coercions.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.bytes_warning.

Type: `Option<BytesWarning>`

`check_hash_pycs_mode` Field

Validation mode for `.pyc` files.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.check_hash_pycs_mode.

Type: `Option<CheckHashPycsMode>`

`configure_c_stdio` Field

Controls binary mode and buffering on C standard streams.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.configure_c_stdio.

Type: `Option<bool>`

`dump_refs` Field

Dump Python references.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.dump_refs.

Type: `Option<bool>`

exec_prefix Field

Controls `sys.exec_prefix`.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.exec_prefix.

Type: `Option<PathBuf>`

executable Field

Controls `sys.executable`.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.executable.

Type: `Option<PathBuf>`

fault_handler Field

Enable `faulthandler`.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.faulthandler.

Type: `Option<bool>`

filesystem_encoding Field

Controls the encoding to use for filesystems/paths.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.filesystem_encoding.

Type: `Option<String>`

filesystem_errors Field

Filesystem encoding error handler.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.filesystem_errors.

Type: `Option<String>`

hash_seed Field

Randomized hash function seed.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.hash_seed.

Type: `Option<c_ulong>`

home **Field**

Python home directory.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.home.

Type: `Option<PathBuf>`

import_time **Field**

Whether to profile `import` time.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.import_time.

Type: `Option<bool>`

inspect **Field**

Enter interactive mode after executing a script or a command.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.inspect.

Type: `Option<bool>`

install_signal_handlers **Field**

Whether to install Python signal handlers.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.install_signal_handlers.

Type: `Option<bool>`

interactive **Field**

Whether to enable the interactive REPL mode.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.interactive.

Type: `Option<bool>`

legacy_windows_stdio **Field**

Controls legacy stdio behavior on Windows.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.legacy_windows_stdio.

Type: `Option<bool>`

`malloc_stats` Field

Whether to dump statistics from the `pymalloc` allocator on exit.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.malloc_stats.

Type: `Option<bool>`

`module_search_paths` Field

Defines `sys.path`.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.module_search_paths.

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

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

Type: `Option<Vec<PathBuf>>`

`optimization_level` Field

Bytecode optimization level.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.optimization_level.

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

Type: `Option<BytecodeOptimizationLevel>`

`parser_debug` Field

Parser debug mode.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.parser_debug.

Type: `Option<bool>`

`pathconfig_warnings` Field

Whether calculating the Python path configuration can emit warnings.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.pathconfig_warnings.

Type: `Option<bool>`

`prefix` Field

Defines `sys.prefix`.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.prefix.

Type: `Option<PathBuf>`

`program_name` Field

Program named used to initialize state during path configuration.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.program_name.

Type: `Option<PathBuf>`

`pycache_prefix` Field

Directory where `.pyc` files are written.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.pycache_prefix.

Type: `Option<PathBuf>`

`python_path_env` Field

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.pythonpath_env.

Type: `Option<String>`

`quiet` Field

Quiet mode.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.quiet.

Type: `Option<bool>`

`run_command` Field

Value of the `-c` command line option.

Effectively defines Python code to evaluate in `Py_RunMain()`.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.run_command.

Type: `Option<String>`

run_filename Field

Filename passed on the command line.

Effectively defines the Python file to run in `Py_RunMain()`.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.run_filename.

Type: `Option<PathBuf>`

run_module Field

Value of the `-m` command line option.

Effectively defines the Python module to run as `__main__` in `Py_RunMain()`.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.run_module.

Type: `Option<String>`

show_ref_count Field

Whether to show the total reference count at exit.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.show_ref_count.

Type: `Option<bool>`

site_import Field

Whether to import the `site` module at startup.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.site_import.

The `site` module is typically not needed for standalone applications and disabling it can reduce application startup time.

Type: `Option<bool>`

skip_first_source_line Field

Whether to skip the first line of `Self::run_filename`.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.skip_source_first_line.

Type: `Option<bool>`

`stdio_encoding` Field

Encoding of `sys.stdout`, `sys.stderr`, and `sys.stdin`.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.stdio_encoding.

Type: `Option<String>`

`stdio_errors` Field

Encoding error handler for `sys.stdout` and `sys.stdin`.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.stdio_errors.

Type: `Option<String>`

`tracemalloc` Field

Whether to enable `tracemalloc`.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.tracemalloc.

Type: `Option<bool>`

`user_site_directory` Field

Whether to add the user site directory to `sys.path`.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.user_site_directory.

Type: `Option<bool>`

`verbose` Field

Verbose mode.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.verbose.

Type: `Option<bool>`

`warn_options` Field

Options of the `warning` module to control behavior.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.warnoptions.

Type: `Option<Vec<String>>`

`write_bytecode` Field

Controls `sys.dont_write_bytecode`.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.write_bytecode.

Type: `Option<bool>`

`x_options` Field

Values of the `-X` command line options / `sys._xoptions`.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.xoptions.

Type: `Option<Vec<String>>`

`MemoryAllocatorBackend` Enum

Defines a backend for a memory allocator.

This says which memory allocator API / library to configure the Python interpreter to use.

Not all allocators are available in all program builds.

Serialization type: `string`

`Default` Variant
> The default allocator as configured by Python.
>
> This likely utilizes the system default allocator, normally the `malloc()`, `free()`, etc functions from the libc implementation being linked against.
>
> Serialized value: `default`

`Jemalloc` Variant
> Use the jemalloc allocator.
>
> Requires the binary to be built with jemalloc support.
>
> Never available on Windows.
>
> Serialized value: `jemalloc`

`Mimalloc` Variant
> Use the mimalloc allocator (https://github.com/microsoft/mimalloc).
>
> Requires the binary to be built with mimalloc support.
>
> Serialized value: `mimalloc`

`Snmalloc` Variant
> Use the snmalloc allocator (https://github.com/microsoft/snmalloc).
>
> Not always available.
>
> Serialized value: `snmalloc`

`Rust` Variant
> Use Rust's global allocator.
>
> The Rust allocator is less efficient than other allocators because of overhead tracking allocations. For optimal performance, use the default allocator. Or if Rust is using a custom global allocator, use the enum variant corresponding to that allocator.

Serialized value: `rust`

`PythonInterpreterProfile` Enum

Defines the profile to use to configure a Python interpreter.

This effectively provides a template for seeding the initial values of `PyPreConfig` and `PyConfig` C structs.

Serialization type: `string`.

`Isolated` Variant
>   Python is isolated from the system.
>
>   See https://docs.python.org/3/c-api/init_config.html#isolated-configuration.
>
>   Serialized value: `isolated`

`Python` Variant
>   Python interpreter behaves like `python`.
>
>   See https://docs.python.org/3/c-api/init_config.html#python-configuration.
>
>   Serialized value: `python`

`Allocator` Enum

Name of the Python memory allocators.

See https://docs.python.org/3/c-api/init_config.html#c.PyPreConfig.allocator.

Serialization type: `string`

`NotSet` Variant
>   Don't change memory allocators (use defaults).
>
>   Serialized value: `not-set`

`Default` Variant
>   Default memory allocators.
>
>   Serialized value: `default`

`Debug` Variant
>   Default memory allocators with debug hooks.
>
>   Serialized value: `debug`

`Malloc` Variant
>   Use `malloc()` from the C library.
>
>   Serialized value: `malloc`

`MallocDebug` Variant
>   Force usage of `malloc()` with debug hooks.
>
>   Serialized value: `malloc-debug`

`PyMalloc` Variant
>   Python `pymalloc` allocator.
>
>   Serialized value: `py-malloc`

PyMallocDebug Variant
> Python `pymalloc` allocator with debug hooks.
>
> Serialized value: `py-malloc-debug`

`BytecodeOptimizationLevel` Enum

An optimization level for Python bytecode.

Serialization type: `int`

Zero Variant
> Optimization level 0.
>
> Serialized value: `0`

One Variant
> Optimization level 1.
>
> Serialized value: `1`

Two Variant
> Optimization level 2.
>
> Serialized value: `2`

`BytesWarning` Enum

Defines what to do when comparing `bytes` or `bytesarray` with `str` or comparing `bytes` with `int`.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.bytes_warning.

Serialization type: `string`

None Variant
> Do nothing.
>
> Serialization value: `none`

Warn Variant
> Issue a warning.
>
> Serialization value: `warn`

Raise Variant
> Raise a `BytesWarning`.
>
> Serialization value: `raise`

`CheckHashPycsMode` Enum

Control the validation behavior of hash-based .pyc files.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.check_hash_pycs_mode.

Serialization type: `string`

Always Variant
> Hash the source file for invalidation regardless of value of the `check_source` flag.
>
> Serialized value: `always`

Never Variant
> Assume that hash-based pycs always are valid.

> Serialized value: `never`

Default Variant
> The `check_source` flag in hash-based pycs determines invalidation.

> Serialized value: `default`

CoerceCLocale Enum

Holds values for `coerce_c_locale`.

See https://docs.python.org/3/c-api/init_config.html#c.PyPreConfig.coerce_c_locale.

Serialization type: `string`

LCCtype Variant
> Read the LC_CTYPE locale to decide if it should be coerced.

> Serialized value: `LC_CTYPE`

C Variant
> Coerce the C locale.

> Serialized value: `C`

MultiprocessingStartMethod Enum

Defines how to call `multiprocessing.set_start_method()` when `multiprocessing` is imported.

When set to a value that is not `none`, when `oxidized_importer.OxidizedFinder` services an import of the `multiprocessing` module, it will automatically call `multiprocessing.set_start_method()` to configure how worker processes are created.

If the `multiprocessing` module is not imported by `oxidized_importer.OxidizedFinder`, this setting has no effect.

Serialization type: `string`

None Variant
> Do not call `multiprocessing.set_start_method()`.

> This mode is what Python programs do by default.

> Serialized value: `none`

Fork Variant
> Call with value `fork`.

> Serialized value: `fork`

ForkServer Variant
> Call with value `forkserver`

> Serialized value: `forkserver`

Spawn Variant
> Call with value `spawn`

> Serialized value: `spawn`

Auto Variant

Call with a valid appropriate for the given environment.

This likely maps to `spawn` on Windows and `fork` on non-Windows.

Serialized value: `auto`

`TerminfoResolution` Enum

Defines `terminfo` database resolution semantics.

Python links against libraries like `readline`, `libedit`, and `ncurses` which need to utilize a `terminfo` database (a set of files defining terminals and their capabilities) in order to work properly.

The absolute path to the terminfo database is typically compiled into these libraries at build time. If the compiled path on the building machine doesn't match the path on the runtime machine, these libraries cannot find the terminfo database and terminal interactions won't work correctly because these libraries don't know how to resolve terminal features. This can result in quirks like the backspace key not working in prompts.

The `pyembed` Rust crate is able to point libraries at a terminfo database at runtime, overriding the compiled-in default path. This enum is used to control that behavior.

Serialization type: `string`.

Dynamic Variant

Resolve `terminfo` database using appropriate behavior for current OS.

We will look for the terminfo database in paths that are common for the current OS / distribution. The terminfo database is present in most systems (except the most barebones containers or sandboxes) and this method is usually successfully in locating the terminfo database.

Serialized value: `dynamic`

None Variant

Do not attempt to resolve the `terminfo` database. Basically a no-op.

This is what should be used for applications that don't interact with the terminal. Using this option will prevent some I/O syscalls that would be incurred by `dynamic`.

Serialized value: `none`

Static Variant

Use a specified string as the `TERMINFO_DIRS` value.

Serialized value: `static:<path>`

e.g. `static:/usr/share/terminfo`.

1.4 PyOxidizer

PyOxidizer is a [Rust] application for streamlining the creation of distributable Python applications.

PyOxidizer is often used to generate binaries embedding a Python interpreter and a custom Python application. However, its configuration files support additional functionality, such as the ability to produce Windows MSI installers, macOS application bundles, and more.

PyOxidizer is primarily made available as the `pyoxidizer` command line tool. However, it is also usable as a Rust library crate.

1.4.1 PyOxidizer

PyOxidizer is a utility for streamlining the creation and distribution of Python applications. See *Overview* for more. Or click through to a documentation section via the index below.

Overview

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

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

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

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`.

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.

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.

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.

How It Works

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

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

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

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

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

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

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

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

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

Getting Started

Python Requirements

PyOxidizer currently targets Python 3.8, 3.9, and 3.10. Your Python application will need to already be compatible with 1 of these versions for it to work with PyOxidizer. See *Why is Python 3.8 Required?* for more on the minimum Python requirement.

Operating System Requirements

PyOxidizer is officially supported on the following operating systems:

- Windows x86 (32-bit)
- Windows x86_64/amd64 (64-bit)
- macOS x86_64 (Intel processors)
- macOS aarch64 (ARM/Apple processors)
- Linux i686 (32-bit)
- Linux x86_64 (64-bit)

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

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

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

Other System Dependencies

You will need a working C compiler/toolchain in order to build binaries. If a C compiler cannot be found, you should see an error message with instructions on how to install one.

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

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

While PyOxidizer is implemented in Rust and invokes the Rust compiler and build tooling to build binaries, PyOxidizer *manages a Rust installation for you*. This means Rust is not an explicit install dependency for PyOxidizer unless you are building PyOxidizer from source code.

Installing

Pre-Built Installers and Executables

PyOxidizer provides pre-built installers and executables as part of its release process. The following should be made available:

- Linux x86-64 statically linked binary.

- macOS universal binary.

- Windows x86 (32-bit) MSI installer.

- Windows amd64 (64-bit) MSI installer.

- Windows universal (x86+amd64) EXE installer.

- Python wheels.

These installers can generally be found at https://github.com/indygreg/PyOxidizer/releases/latest.

If this URL does not redirect to a PyOxidizer release, go to https://github.com/indygreg/PyOxidizer/releases and look for a release with PyOxidizer release artifacts. You should see giant text that reads `PyOxidizer <version>` that looks different from other entries in the list. You may have to click through multiple *next* links at the bottom of the release list until you find a PyOxidizer release.

If pre-built artifacts are not available for your machine, you will need to compile PyOxidizer from source code.

Python Wheels

PyOxidizer is made available as a binary Python wheel (`.whl`) and releases are published on PyPI. So you can install PyOxidizer like any other Python package:

```
$ python3 -m pip install pyoxidizer

# To upgrade an existing install
$ python3 -m pip install --upgrade pyoxidizer
```

Installing PyOxidizer from Source

Installing Rust

PyOxidizer is a Rust application and requires Rust (1.60 or newer) to be installed in order to build PyOxidizer.

You can verify your installed version of Rust by running:

```
$ rustc --version
rustc 1.61.0 (fe5b13d68 2022-05-18)
```

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.

Installing PyOxidizer

Once Rust is installed, PyOxidizer can be installed from its latest published crate on Rust's official/default package repository:

```
$ cargo install pyoxidizer
```

From PyOxidizer's canonical 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 canonical Git repository and building the project locally:

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

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

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

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

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

```
$ pyoxidizer
PyOxidizer 0.14.0-pre
Gregory Szorc <gregory.szorc@gmail.com>
Build and distribute Python applications

USAGE:
    pyoxidizer [FLAGS] [SUBCOMMAND]


...
```

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

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.

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.

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.

Customizing Python and Packaging Behavior

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

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

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

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

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

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

It works!

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

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

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

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

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

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

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

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

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

You've just produced an executable for `pyflakes`!

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

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

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`.

Settings

Cache Directory

`pyoxidizer` may need to download resources such as Python distributions and Rust toolchains from the Internet. These resources are cached in a per-user directory.

PyOxidizer chooses the first available directory from the following list to use as the cache:

- The value of the environment variable `PYOXIDIZER_CACHE_DIR`.

- `$XDG_CACHE_HOME/pyoxidizer` on Linux if `XDG_CACHE_HOME` is set.

- `$HOME/.cache/pyoxidizer` on Linux if `HOME` is set.

- `$HOME/Library/Caches/pyoxidizer` on macOS if `HOME` is set.

- `{FOLDERID_LocalAppData}/pyoxidizer` on Windows.

- `~/.pyoxidizer/cache`

The `pyoxidizer cache-clear` command can be used to delete the contents of the cache.

Managed Rust Toolchain

PyOxidizer leverages the Rust programming language and its tooling for building binaries embedding Python.

By default, PyOxidizer will automatically download and use Rust toolchains (the Rust compiler, standard library, and Cargo) when their functionality is needed. PyOxidizer will store these Rust toolchains in the configured *cache*.

If you already have Rust installed on your machine and want PyOxidizer to use the existing Rust installation, either pass the `--system-rust` flag to `pyoxidizer` invocations or define the `PYOXIDIZER_SYSTEM_RUST` environment variable to any value. When the *system* Rust is being used, `pyoxidizer` will automatically use the `cargo` executable found on the current search path (typically the `PATH` environment variable).

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.

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

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.

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
```

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.

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.

Debugging Resource Scanning and Identification with `find-resources`

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

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

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

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

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

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

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

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

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

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

Defining Extra Variables in Starlark Environment

Various `pyoxidizer` commands (like `build` and `run`) accept arguments to define extra variables in the Starlark environment in the `VARS` global dict. This feature can be used to parameterize and conditionalize the evaluation of configuration files.

Note: While we could inject global variables into the Starlark environment, since it is illegal to access an undefined symbol (there's not even a way to test if a symbol is defined) and since we have no hook point to inject variables after the symbol has been defined, we resort to populating a global `VARS` dict with variables.

For example, let's make the name of the built executable dynamic:

```python
DEFAULT_APP_NAME = "default"


def make_exe(dist):
    dist = default_python_distribution()
    return dist.to_python_executable(name = VARS.get("app_name", DEFAULT_APP_NAME))


register_target("exe", make_exe)

resolve_targets()
```

Then let's build it:

```
# Uses `default` as the application name.
$ pyoxidizer build

# Uses `my_app` as the application name.
$ pyoxidizer build --var app_name my_app

# Uses `env_name` as the application name via an environment variable.
$ APP_NAME=env_name pyoxidizer build --var-env app_name APP_NAME
```

Configuration Files

PyOxidizer uses Starlark files to configure run-time behavior.

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

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

Automatic File Location Strategy

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

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

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

If no configuration file is found, an error occurs.

Concepts

Processing

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

Since Starlark is effectively a subset of Python, executing a `PyOxidizer` configuration file is effectively running a sandboxed Python script. It is conceptually similar to running `python setup.py` to build a Python package. As functions within the Starlark environment are called, `PyOxidizer` will perform actions as described by those functions.

Targets

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

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

register_target("dist", get_python_distribution)
```

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

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

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

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

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

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

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

Python Distributions Provide Python

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

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

Instances of *PythonDistribution* are typically constructed from *default_python_distribution()*.

Python Executables Run Python

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

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

Python Resources

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

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

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

Specifying Resource Locations

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

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

in-memory
> Load the resource from memory.

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

Resource Attributes Influencing Adding

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

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

add_include

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

add_location

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

It can be set to the following values:

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

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

filesystem-relative:<prefix>

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

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

add_location_fallback

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

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

add_source

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

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

add_bytecode_optimization_level_zero

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

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

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

add_bytecode_optimization_level_one

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

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

add_bytecode_optimization_level_two

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

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

Global Symbols

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

The Starlark environment contains symbols from the following:

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

In addition, extra global variables can be injected into the execution environment on a per-invocation basis. This is commonly encountered with use of the `--var` and *–var-env*` arguments to various `pyoxidizer` sub-commands.

Global Types

PyOxidizer's Starlark dialect defines the following custom types:

`File`
> Represents a filesystem path and content.

`starlark_tugger.FileContent`
> Represents the content of a file on the filesystem.
>
> (Unlike `File`, this does not track the filename internally.)

`starlark_tugger.FileManifest`
> Represents a mapping of filenames to file content.

`PythonDistribution`
> Represents an implementation of Python.
>
> Used for embedding into binaries and running Python code.

`PythonEmbeddedResources`
> Represents resources made available to a Python interpreter.

`PythonExecutable`
> Represents an executable file containing a Python interpreter.

`PythonExtensionModule`
> Represents a compiled Python extension module.

`PythonInterpreterConfig`
> Represents the configuration of a Python interpreter.

`PythonPackageDistributionResource`
> Represents a file containing Python package distribution metadata.

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

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

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

Global Constants

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

BUILD_TARGET_TRIPLE

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

CONFIG_PATH

The string path to the configuration file currently being evaluated.

CONTEXT

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

CWD

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

Global Functions

PyOxidizer's Starlark dialect defines the following global functions:

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

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

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

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

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

Types with Target Behavior

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

The following types have special behavior registered:

starlark_tugger.FileManifest
> Build behavior is to materialize all files in the file manifest.
>
> Run behavior is to run the last added *PythonExecutable* if available, falling back to an executable file installed by the manifest if there is exactly 1 executable file.

PythonEmbeddedResources
> Build behavior is to write out files this type represents.
>
> There is no run behavior.

PythonExecutable
> Build behavior is to build the executable file.
>
> Run behavior is to run that built executable.

Functions for Manipulating Global State

starlark_pyoxidizer.**set_build_path**(*path: str*)
> 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.
>
> ---

Functions for Managing Targets

register_target()

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

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

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

Arguments:

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

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

depends

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

default

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

default_build_script

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

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

`resolve_target()`

Triggers resolution of a requested build target.

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

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

`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.

Extensions to Tugger's Starlark Dialect

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

`FileManifest.add_python_resource()`

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

Arguments:

prefix

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

value

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

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

`FileManifest.add_python_resources()`

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

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

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

File

class `starlark_pyoxidizer.`**`File`**

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

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

path

(string)

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

is_executable

(bool)

Whether the file is executable.

is_*

(various)

See *Resource Attributes Influencing Adding*.

PythonDistribution

class `starlark_pyoxidizer.`**`PythonDistribution`**

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

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

`__init__`(*sha256: str, local_path: Optional[string] = None, url: Optional[string], flavor: Optional[string] = None*) → *PythonDistribution*

Construct an instance from arguments.

The following arguments are accepted:

sha256

The SHA-256 of the distribution archive file.

local_path
> Local filesystem path to the distribution archive.

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

flavor
> The distribution flavor. Must be `standalone`.

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

A distribution is defined by a location and a hash.

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

Examples:

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

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

python_resources() → list[Union[*PythonModuleSource*, *PythonExtensionModule*, *PythonPackageResource*]]

> Returns objects representing Python resources in this distribution. Returned values can be `PythonModuleSource`, `PythonExtensionModule`, `PythonPackageResource`, etc.
>
> There may be multiple `PythonExtensionModule` with the same name.

make_python_interpreter_config() → *PythonInterpreterConfig*

> Obtain a `PythonInterpreterConfig` derived from the distribution.
>
> The interpreter configuration automatically uses settings appropriate for the distribution.

make_python_packaging_policy() → *PythonPackagingPolicy*

> Obtain a `PythonPackagingPolicy` derived from the distribution.
>
> The policy automatically uses settings globally appropriate for the distribution.

to_python_executable(*name: str*, *packaging_policy:* PythonPackagingPolicy, *config:* PythonInterpreterConfig) → *PythonExecutable*

> This method constructs a `PythonExecutable` instance. It essentially says *build an executable embedding Python from this distribution*.
>
> The accepted arguments are:

name
> The name of the application being built. This will be used to construct the default filename of the executable.

packaging_policy
> The packaging policy to apply to the executable builder.

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

config

The default configuration of the embedded Python interpreter.

Default is what `make_python_interpreter_config()` returns.

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

---

default_python_distribution()

starlark_pyoxidizer.**default_python_distribution**(*flavor: str = 'standalone'*, *build_target: str =*
BUILD_TARGET, *python_version: str = '3.10'*) →
PythonDistribution

Resolves the default *PythonDistribution*.

The following named arguments are accepted:

flavor

Denotes the *distribution* flavor. See the section below on allowed values.

build_target

Denotes the machine target triple that we're building for.

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

python_version

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

Supported values are `3.8`, `3.9`, and `3.10`.

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

standalone

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

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

standalone_dynamic

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

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

standalone_static

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

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

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

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

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

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

PythonEmbeddedResources

class starlark_pyoxidizer.**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.

PythonExecutable

class starlark_pyoxidizer.**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()*.

licenses_filename

(`str`)

The filename to use / write for an auto-generated report of software component licensing relevant to the built executable.

The file will contain a bill-of-materials of all the known software components in the built binary. This includes information about the Python distribution, extension modules and libraries used by the Python distribution, 3rd party Python packages, and Rust crates.

Each component is annotated with licensing information, including the license text, if available.

All content in the file is best effort.

If `None`, no file will be written.

Default: `COPYING.txt`

packed_resources_load_mode

> (str)
>
> Defines how the *packed Python resources data* (see *Python Packed Resources*) is written and loaded at run-time by the embedded Python interpreter.
>
> The following values/patterns can be defined:
>
> **none**
>> No resources data will be serialized or loaded at run-time. (Use this if you are using Python's filesystem based module importer and don't want to use PyOxidizer's custom importer.)
>
> **embedded:<filename>**
>> The packed resources data will be embedded in the binary and loaded from a memory address at run-time.
>>
>> `filename` denotes the path of the on-disk file used at build time. This file is written to the *artifacts* directory that PyOxidizer writes required build files to.
>
> **binary-relative-memory-mapped:<filename>**
>> The packed resources data will be written to a file relative to the built binary and loaded from there at run-time using memory mapped I/O.
>
> The default is `embedded:packed-resources`.

tcl_files_path

> (Optional[str])
>
> Defines a directory relative to that of the built executable in which to install tcl/tk files.
>
> If set to a value, tcl/tk files present in the Python distribution being used will be installed next to the build executable and the embedded Python interpreter will automatically set the `TCL_LIBRARY` environment variable to load tcl files from this directory.
>
> If `None` (the default), no tcl/tk files will be installed.

windows_runtime_dlls_mode

> (str)
>
> Controls how Windows runtime DLLs should be managed when building the binary.
>
> Windows binaries often have a dependency on various runtime DLLs, such as `vcruntime140.dll`. The built executable will need access to these DLLs or it won't work.
>
> This setting controls whether to install required Windows runtime DLLs next to the built binary at build time. For example, if you are producing a `myapp.exe`, this setting can automatically install a `vcruntime140.dll` next to that binary.
>
> The following values are recognized:
>
> **never**
>> Never install Windows runtime DLLs.
>
> **when-present**
>> Install Windows runtime DLLs when they can be located. Do nothing if they can't be found.
>
> **always**
>> Install Windows runtime DLLs and fail if they can't be located.

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

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

windows_subsystem

> (`str`)
>
> Controls the value to use for the Rust `#![windows_subsystem = "..."]` attribute added to the autogenerated Rust program to build the executable.
>
> This attribute only has meaning on Windows. It effectively controls the value passed to the linker's `/SUBSYSTEM` flag.
>
> Rust only supports certain values but PyOxidizer does not impose limitations on what values are used. Common values include:
>
> **console**
> > Win32 character-mode application. A console window will be opened when the application runs.
> >
> > This value is suitable for command-line executables.
>
> **windows**
> > Application does not require a console and may provide its own windows.
> >
> > This value is suitable for GUI applications that do not wish to launch a console window on start.
>
> Default is `console`.

make_python_module_source(*name: str*, *source: str*, *is_package: bool*) → *PythonModuleSource*

> This method creates a *PythonModuleSource* instance suitable for use with the executable being built.
>
> Arguments are as follows:
>
> **name**
> > The name of the Python module. This is the fully qualified module name. e.g. `foo` or `foo.bar`.
>
> **source**
> > Python source code comprising the module.
>
> **is_package**
> > Whether the Python module is also a package. (e.g. the equivalent of a `__init__.py` file or a module without a `.` in its name.

pip_download(*args: list[str]*) → list[Any]

> This method runs `pip download <args>` with settings appropriate to target the executable being built.
>
> This always uses `--only-binary=:all:`, forcing pip to only download wheel based packages.
>
> This method accepts the following arguments:
>
> **args**
> > (`list` of `str`) Command line arguments to pass to `pip download`. Arguments will be added after default arguments added internally.
>
> Returns a `list` of objects representing Python resources collected from wheels obtained via `pip download`.

pip_install(*args: list[str]*, *extra_envs: Optional[dict[str, str]]*) → list[Any]

> This method runs `pip install <args>` with settings appropriate to target the executable being built.
>
> **args**
> > List of strings defining raw process arguments to pass to `pip install`.

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

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

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

read_package_root(*path: str*, *packages: list[str]*) → list[Any]
> 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**
>> The filesystem path to the directory to scan.

> **packages**
>> List of package names to include.

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

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

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

read_virtualenv(*path: str*) → list[Any]
> 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**
>> The filesystem path to the root of the virtualenv.

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

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

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

setup_py_install(*package_path: str*, *extra_envs: dict[str, str] = {}*, *extra_global_arguments: dict[str, str] = {}*) → list[Any]

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

It accepts the following arguments:

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

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

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

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

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

add_python_resource(*resource: Union[*PythonModuleSource, PythonPackageResource, PythonExtensionModule*]*)

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

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

The following arguments are accepted:

resource
  The resource to add to the embedded Python environment.

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

add_python_resources(*resources: list[Union[PythonModuleSource, PythonPackageResource, PythonExtensionModule]]*)

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

add_cargo_manifest_licensing(*manifest_path:* str, *all_features:* bool = *False*, *features=None*)

Register software component licensing for a package defined in a `Cargo.toml` manifest.

This method accepts the following arguments:

manifest_path
  Filesystem path of `Cargo.toml` to process.

all_features
  Whether to activate all crate features when determining licensing info.

features
  List of strings denoting explicit features to enable.

  Ignored if `all_features` is enabled.

filter_resources_from_files(*files:* list[str], *glob_files:* list[str])

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
> 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
> 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.

to_embedded_resources()

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

See the *PythonEmbeddedResources* type documentation for more.

to_file_manifest(*prefix: str*) → *starlark_tugger.FileManifest*

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

This method accepts the following arguments:

prefix
> The directory prefix of files in the *starlark_tugger.FileManifest*. Use . to denote no prefix.

to_wix_bundle_builder(*id_prefix: str*, *product_name: str*, *product_version: str*, *product_manufacturer: str*, *msi_builder_callback: Callable*) → *starlark_tugger.WiXBundleBuilder*

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

This method accepts the following arguments:

id_prefix
> See *starlark_tugger.WiXMSIBuilder.__init__()* for usage.

product_name
> See *starlark_tugger.WiXMSIBuilder.__init__()* for usage.

product_version
> See *starlark_tugger.WiXMSIBuilder.__init__()* for usage.

product_manufacturer
> See *starlark_tugger.WiXMSIBuilder.__init__()* for usage.

msi_builder_callback
> (function) A callable function that can be used to modify the *starlark_tugger.WiXMSIBuilder* constructed for the application.
>
> The function will receive the *starlark_tugger.WiXMSIBuilder* as its single argument. The return value is ignored.

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

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

---

to_wix_msi_builder(*id_prefix:* *str*, *product_name:* *str*, *product_version:* *str*, *product_manufacturer:* *str*)
→ *starlark_tugger.WiXMSIBuilder*

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

This method accepts the following arguments:

id_prefix
> See *starlark_tugger.WiXMSIBuilder.__init__()* for usage.

product_name
> See *starlark_tugger.WiXMSIBuilder.__init__()* for usage.

product_version
> See *starlark_tugger.WiXMSIBuilder.__init__()* for usage.

product_manufacturer
> See *starlark_tugger.WiXMSIBuilder.__init__()* for usage.

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

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

build(*target:* *str*) → *starlark_tugger.ResolvedTarget*

Produces a binary executable embedding Python using the settings configured on this instance.

> **target**
> > The name of the target being built.

Under the covers, this will generate a temporary Rust project and invoke `cargo`, Rust's build tool, for generating an executable. The end result of this process is a single executable embedding a Python interpreter.

Upon successful generation of a binary, the produced binary will be assessed for code signing with the `python-executable-creation` *action*.

write_licenses(*path*)

Writes software component licensing info to the file specified via `path`.

The file will contain a bill of materials of all software components included in the resulting binary and licensing information related to them. This includes license texts, when available.

Licensing info is best effort.

PythonExtensionModule

class starlark_pyoxidizer.**PythonExtensionModule**

> This type represents a compiled Python extension module.

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

---

is_stdlib

>   (bool)

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

add_*

>   (various)

>   See *Resource Attributes Influencing Adding*.

PythonInterpreterConfig

class starlark_pyoxidizer.**PythonInterpreterConfig**

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

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

>   ---

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

>   ---

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

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

>   Instance state is managed via attributes.

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

>   The following attributes control features provided by the pyembed Rust crate, which manages the embedded Python interpreter in generated executables. These attributes provide features and level of control over embedded Python interpreters beyond what is possible with Python's initialization C API.

>   - *allocator_backend*
>   - *allocator_raw*
>   - *allocator_mem*
>   - *allocator_obj*
>   - *allocator_pymalloc_arena*
>   - *allocator_debug*
>   - *oxidized_importer*
>   - *filesystem_importer*
>   - *argvb*
>   - *multiprocessing_auto_dispatch*

- *multiprocessing_start_method*
- *sys_frozen*
- *sys_meipass*
- *terminfo_resolution*
- *write_modules_directory_env*

The following attributes correspond to fields of the PyPreConfig C struct used to initialize the Python interpreter.

- *config_profile*
- *allocator*
- *configure_locale*
- *coerce_c_locale*
- *coerce_c_locale_warn*
- *development_mode*
- *isolated*
- *legacy_windows_fs_encoding*
- *parse_argv*
- *use_environment*
- *utf8_mode*

The following attributes correspond to fields of the PyConfig C struct used to initialize the Python interpreter.

- *base_exec_prefix*
- *base_executable*
- *base_prefix*
- *buffered_stdio*
- *bytes_warning*
- *check_hash_pycs_mode*
- *configure_c_stdio*
- *dump_refs*
- *exec_prefix*
- *executable*
- *fault_handler*
- *filesystem_encoding*
- *hash_seed*
- *home*
- *import_time*
- *inspect*
- *install_signal_handlers*
- *interactive*

- *legacy_windows_stdio*
- *malloc_stats*
- *module_search_paths*
- *optimization_level*
- *parser_debug*
- *pathconfig_warnings*
- *prefix*
- *program_name*
- *pycache_prefix*
- *python_path_env*
- *quiet*
- *run_command*
- *run_filename*
- *run_module*
- *show_ref_count*
- *site_import*
- *skip_first_source_line*
- *stdio_encoding*
- *stdio_errors*
- *tracemalloc*
- *user_site_directory*
- *verbose*
- *warn_options*
- *write_bytecode*
- *x_options*

allocator_backend

(string)

See *allocator_backend Field*.

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

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

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

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

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

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

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

allocator_raw

> (bool)
>
> See *allocator_raw Field*.
>
> Defaults to `True`.

allocator_mem

> (bool)
>
> See *allocator_mem Field*.
>
> Defaults to `False`.

allocator_obj

> (bool)
>
> See *allocator_obj Field*.
>
> Defaults to `False`.

allocator_pymalloc_arena

> (bool)
>
> See *allocator_pymalloc_arena Field*.
>
> Defaults to `False`.

allocator_debug

> (bool)
>
> See *allocator_debug Field*.
>
> Defaults to `False`.

oxidized_importer

> (bool)
>
> See *oxidized_importer Field*.
>
> Defaults to `True`.

filesystem_importer

> (bool)
>
> See *filesystem_importer Field*.

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

argvb

> (bool)
>
> See *argvb Field*.

multiprocessing_auto_dispatch

> (bool)
>
> See *multiprocessing_auto_dispatch Field*.
>
> Default value is `True`.
>
> See *Automatic Detection and Dispatch of multiprocessing Processes* for more.

multiprocessing_start_method

> (str)
>
> See *multiprocessing_start_method Field*.

sys_frozen

> (bool)
>
> See *sys_frozen Field*.
>
> Default is `True`.

sys_meipass

> (bool)
>
> See *sys_meipass Field*.
>
> Default is `False`.

terminfo_resolution

> (string)
>
> See *terminfo_resolution Field*.
>
> See *Terminfo Database* for more about terminal databases.

write_modules_directory_env

> (string or None)
>
> See *write_modules_directory_env Field*.

config_profile

> (string)
>
> See *profile Field*.

allocator

> (string or None)
>
> See *allocator Field*.

configure_locale

> (bool or None)
>
> See *configure_locale Field*.

coerce_c_locale

> (string or None)
>
> See *coerce_c_locale Field*.

coerce_c_locale_warn

> (bool or None)
>
> See *coerce_c_locale_warn Field*.

development_mode

> (bool or None)
>
> See *development_mode Field*.

isolated

> (bool or None)
>
> See *isolated Field*.

legacy_windows_fs_encoding

> (bool or None)
>
> See *legacy_windows_fs_encoding Field*.

parse_argv

> (bool or None)
>
> See *parse_argv Field*.

use_environment

> (bool or None)
>
> See *use_environment Field*.

utf8_mode

> (bool or None)
>
> See *utf8_mode Field*.

base_exec_prefix

> (string or None)
>
> See *base_exec_prefix Field*.

base_executable

> (string or None)
>
> See *base_executable Field*.

base_prefix

> (string or None)
>
> See *base_prefix Field*.

buffered_stdio

> (bool or None)
>
> See *buffered_stdio Field*.

bytes_warning

>   (string or None)

>   See *bytes_warning Field*.

check_hash_pycs_mode

>   (string or None)

>   See *check_hash_pycs_mode Field*.

configure_c_stdio

>   (bool or None)

>   See *configure_c_stdio Field*.

dump_refs

>   (bool or None)

>   See *dump_refs Field*.

exec_prefix

>   (string or None)

>   See *exec_prefix Field*.

executable

>   (string or None)

>   See *executable Field*.

fault_handler

>   (bool or None)

>   See *fault_handler Field*.

filesystem_encoding

>   (string or None)

>   See *filesystem_encoding Field*.

filesystem_errors

>   (string or None)

>   See *filesystem_errors Field*.

hash_seed

>   (int or None)

>   See *hash_seed Field*.

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

home

>   (string or None)

>   See *home Field*.

import_time

>   See *import_time Field*.

inspect

> (bool or None)
>
> See *inspect Field*.

install_signal_handlers

> (bool or None)
>
> See *install_signal_handlers Field*.

interactive

> (bool or None)
>
> See *interactive Field*.

legacy_windows_stdio

> (bool or None)
>
> See *legacy_windows_stdio Field*.

malloc_stats

> (bool or None)
>
> See *malloc_stats Field*.

module_search_paths

> (list[string] or None)
>
> See *module_search_paths Field*.
>
> Setting this to a non-empty value also has the side-effect of setting `filesystem_importer = True`

optimization_level

> (int or None)
>
> See *optimization_level Field*.

parser_debug

> (bool or None)
>
> See *parser_debug Field*.

pathconfig_warnings

> (bool or None)
>
> See *pathconfig_warnings Field*.

prefix

> (string or None)
>
> See *prefix Field*.

program_name

> (string or None)
>
> See *program_name Field*.

pycache_prefix

> (string or None)
>
> See *pycache_prefix Field*.

python_path_env

(string or None)

See *python_path_env Field*.

quiet

(bool or None)

See *quiet Field*.

run_command

(string or None)

See *run_command Field*.

run_filename

(string or None)

See *run_filename Field*.

run_module

(string or None)

See *run_module Field*.

show_ref_count

(bool or None)

See *show_ref_count Field*.

site_import

(bool or None)

See *site_import Field*.

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

skip_first_source_line

(bool or None)

See *skip_first_source_line Field*.

stdio_encoding

(string or None)

See *stdio_encoding Field*.

stdio_errors

(string or None)

See *stdio_errors Field*.

tracemalloc

(bool or None)

See *tracemalloc Field*.

user_site_directory

(bool or None)

See *user_site_directory Field*.

verbose

> (bool or None)
>
> See *verbose Field*.

warn_options

> (list[string] or None)
>
> See *warn_options Field*.

write_bytecode

> (bool or None)
>
> See *write_bytecode Field*.

x_options

> (list[string] or None)
>
> See *x_options Field*.

Starlark Caveats

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

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

For example:

```
config = dist.make_python_interpreter_config()

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

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

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

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

PythonModuleSource

class starlark_pyoxidizer.**PythonModuleSource**

> This type represents Python source modules, agnostic of location.
>
> Instances can be constructed via *PythonExecutable.make_python_module_source()* or by calling methods that emit Python resources.
>
> **name**
>
> > (string)
> >
> > Fully qualified name of the module. e.g. `foo.bar`.

---

source

>   (string)
>
>   The Python source code for this module.

is_package

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

is_stdlib

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

add_*

>   (various)
>
>   See *Resource Attributes Influencing Adding*.

PythonPackageResource

class starlark_pyoxidizer.**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.

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

>   **name**
>
>   >   (string)
>   >
>   >   Name of this resource.

>   **is_stdlib**
>
>   >   (bool)
>   >
>   >   Whether this module is part of the Python standard library (part of the Python distribution).

>   **add_***
>
>   >   (various)
>   >
>   >   See *Resource Attributes Influencing Adding*.

PythonPackageDistributionResource

class starlark_pyoxidizer.**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.

package

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

name

> (`string`)
>
> Name of this resource.

is_stdlib

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

add_*

> (various)
>
> See *Resource Attributes Influencing Adding*.

PythonPackagingPolicy

class `starlark_pyoxidizer.`**PythonPackagingPolicy**

> When building a Python binary, there are various settings that control which Python resources are added, where they are imported from, and other various settings. This collection of settings is referred to as a *Python Packaging Policy*. These settings are represented by the `PythonPackagingPolicy` type.
>
> **allow_files**
>
> > (`bool`)
> >
> > Whether to allow the collection of generic *file* resources.
> >
> > If false, all collected/packaged resources must be instances of concrete resource types (`PythonModuleSource`, `PythonPackageResource`, etc).
> >
> > If true, `File` instances can be added to resource collectors.
>
> **allow_in_memory_shared_library_loading**
>
> > (`bool`)
> >
> > Whether to allow loading of Python extension modules and shared libraries from memory at run-time.
> >
> > Some platforms (notably Windows) allow opening shared libraries from a memory address. This mode of opening shared libraries allows libraries to be embedded in binaries without having to statically link them. However, not every library works correctly when loaded this way.
> >
> > This flag defines whether to enable this feature where supported. Its true value can be ignored if the target platform doesn't support loading shared library from memory.
>
> **bytecode_optimize_level_zero**
>
> > (`bool`)
> >
> > Whether to add Python bytecode at optimization level 0 (the default optimization level the Python interpreter compiles bytecode for).
>
> **bytecode_optimize_level_one**
>
> > (`bool`)
> >
> > Whether to add Python bytecode at optimization level 1.

bytecode_optimize_level_two

>   (bool)
>
>   Whether to add Python bytecode at optimization level 2.

extension_module_filter

>   (string)
>
>   The filter to apply to determine which extension modules to add. The following values are recognized:
>
>   **all**
>   Every named extension module will be included.
>
>   **minimal**
>   Return only extension modules that are required to initialize a Python interpreter. This is a very small set and various functionality from the Python standard library will not work with this value.
>
>   **no-libraries**
>   Return only extension modules that don't require any additional libraries.
>
>   Most common Python extension modules are included. Extension modules like `_ssl` (links against OpenSSL) and `zlib` are not included.
>
>   **no-copyleft**
>   Return only extension modules that do not link against *copyleft* licensed libraries.
>
>   Not all Python distributions may annotate license info for all extensions or the libraries they link against. If license info is missing, the extension is not included because it *could* be *copyleft* licensed. Similarly, the mechanism for determining whether a license is *copyleft* is based on the SPDX license annotations, which could be wrong or out of date.
>
>   Default is `all`.

file_scanner_classify_files

>   (bool)
>
>   Whether file scanning should attempt to classify files and emit typed resources corresponding to the detected file type.
>
>   If `True`, operations that emit resource objects (such as *PythonExecutable.pip_install()*) will emit specific types for each resource flavor. e.g. *PythonModuleSource*, *PythonExtensionModule*, etc.
>
>   If `False`, the file scanner does not attempt to classify the type of a file and this rich resource types are not emitted.
>
>   Can be used in conjunction with *PythonPackagingPolicy.file_scanner_emit_files*. If both are `True`, there will be a *File* and an optional non-file resource for each source file.
>
>   Default is `True`.

file_scanner_emit_files

>   (bool)
>
>   Whether file scanning should emit file resources for each seen file.
>
>   If `True`, operations that emit resource objects (such as *PythonExecutable.pip_install()*) will emit *File* instances for each encountered file.
>
>   If `False`, *File* instances will not be emitted.
>
>   Can be used in conjunction with *PythonPackagingPolicy.file_scanner_classify_files*.
>
>   Default is `False`.

include_classified_resources

> (bool)
>
> Whether strongly typed, classified non-`File` resources have their `add_include` attribute set to `True` by default.
>
> Default is `True`.

include_distribution_sources

> (bool)
>
> Whether to add source code for Python modules in the Python distribution.
>
> Default is `True`.

include_distribution_resources

> (bool)
>
> Whether to add Python package resources for Python packages in the Python distribution.
>
> Default is `False`.

include_file_resources

> (bool)
>
> Whether `File` resources have their `add_include` attribute set to `True` by default.
>
> Default is `False`.

include_non_distribution_sources

> (bool)
>
> Whether to add source code for Python modules not in the Python distribution.

include_test

> (bool)
>
> Whether to add Python resources related to tests.
>
> Not all files associated with tests may be properly flagged as such. This is a best effort setting.
>
> Default is `False`.

resources_location

> (string)
>
> The location that resources should be added to by default.
>
> Default is `in-memory`.

resources_location_fallback

> (string or `None`)
>
> The fallback location that resources should be added to if `resources_location` fails.
>
> Default is `None`.

preferred_extension_module_variants

> (dict<string, string>) (readonly)
>
> Mapping of extension module name to variant name.
>
> This mapping defines which preferred named variant of an extension module to use. Some Python distributions offer multiple variants of the same extension module. This mapping allows defining which variant of which extension to use when choosing among them.

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

register_resource_callback(*f: Callable*)

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

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

set_preferred_extension_module_variant(*extension: str*, *variant: str*)

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

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

set_resource_handling_mode(*mode: str*)

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

classify

Files are classified as typed resources and handled as such.

Only classified resources can be added by default.

files

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

Only files can be added by default.

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

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

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

Packaging User Guide

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

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

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.

Packaging Primitives in `pyoxidizer.bzl` Files

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

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

Targets Define Actions

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

Defining an Executable Embedding Python

In this example, we create an executable embedding Python:

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

    return dist.to_python_executable("myapp")

register_target("exe", make_exe)
resolve_targets()
```

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

Configuring the Python Interpreter Run-Time Behavior

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

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

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

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

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

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

register_target("exe", make_exe)
resolve_targets()
```

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

- Control of low-level Python interpreter settings, such as whether environment variables (like PYTHONPATH) should influence run-time behavior, whether stdio should be buffered, and the filesystem encoding to use.

- Whether to enable the importing of Python modules from the filesystem and what the initial value of sys.path should be.

- The memory allocator that the Python interpreter should use.

- What Python code to run when the interpreter is started.

- How the terminfo database should be located.

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

Adding Python Packages to Executables

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

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

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

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

Install Manifests Copy Files Next to Your Application

The *starlark_tugger.FileManifest* Starlark type represents a collection of files and their content. When *starlark_tugger.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.

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

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

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

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

    return dist.to_python_executable("myapp")

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

<div align="right">(continues on next page)</div>

```
    m.add_python_resource(".", exe)

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

    return m

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

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

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

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

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

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

Understanding Python Distributions

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

Available Python Distributions

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

Here are the built-in Python distributions:

Source	Version	Flavor	Build Target
CPython	3.9.13	standalone_dynamic	aarch64-unknown-linux-gnu
CPython	3.10.4	standalone_dynamic	aarch64-unknown-linux-gnu
CPython	3.8.13	standalone_dynamic	x86_64-unknown-linux-gnu
CPython	3.9.13	standalone_dynamic	x86_64-unknown-linux-gnu
CPython	3.10.4	standalone_dynamic	x86_64-unknown-linux-gnu
CPython	3.8.13	standalone_static	x86_64-unknown-linux-musl
CPython	3.9.13	standalone_static	x86_64-unknown-linux-musl
CPython	3.10.4	standalone_static	x86_64-unknown-linux-musl
CPython	3.8.13	standalone_dynamic	i686-pc-windows-msvc
CPython	3.9.13	standalone_dynamic	i686-pc-windows-msvc
CPython	3.10.4	standalone_dynamic	i686-pc-windows-msvc
CPython	3.8.13	standalone_static	i686-pc-windows-msvc
CPython	3.9.13	standalone_static	i686-pc-windows-msvc
CPython	3.10.4	standalone_static	i686-pc-windows-msvc
CPython	3.8.13	standalone_dynamic	x86_64-pc-windows-msvc
CPython	3.9.13	standalone_dynamic	x86_64-pc-windows-msvc
CPython	3.10.4	standalone_dynamic	x86_64-pc-windows-msvc
CPython	3.8.13	standalone_static	x86_64-pc-windows-msvc
CPython	3.9.13	standalone_static	x86_64-pc-windows-msvc
CPython	3.10.4	standalone_static	x86_64-pc-windows-msvc
CPython	3.8.13	standalone_dynamic	aarch64-apple-darwin
CPython	3.9.13	standalone_dynamic	aarch64-apple-darwin
CPython	3.10.4	standalone_dynamic	aarch64-apple-darwin
CPython	3.8.13	standalone_dynamic	x86_64-apple-darwin
CPython	3.9.13	standalone_dynamic	x86_64-apple-darwin
CPython	3.10.4	standalone_dynamic	x86_64-apple-darwin

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

Here is what those target triple values translate to:

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

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

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

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

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

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

Python Version Compatibility

PyOxidizer is capable of working with Python 3.8 and 3.9.

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

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

Choosing a Python Distribution

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

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

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

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

Binary Portability of Distributions

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

- All 3rd party shared libraries are part of the distribution (e.g. `libssl` and `libsqlite3`) and don't need to be provided by the run-time environment.

- Some distributions are statically linked and have no dependencies on any external shared libraries.

- On the glibc linked Linux distributions, they use an old glibc version for symbol versions, enabling them to run on Linux distributions created years ago. (The current version is 2.19, which was released in 2014.)

- Any shared libraries not provided by the distribution are available in base operating system installs. On Linux, example shared libraries include `libc.so.6` and `linux-vdso.so.1`, which are part of the Linux Standard Base Core Configuration and should be present on all conforming Linux distros. On macOS, referenced dylibs include `libSystem`, which is part of the macOS core install.

- For Linux, see *Distribution Considerations for Linux* for portability considerations.

- For macOS, see *Distribution Considerations for macOS* for portability considerations.

- For Windows, see *Distribution Considerations for Windows* for portability considerations.

Known Issues with Distributions

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

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

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

Managing How Resources are Added

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

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

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

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

Classified Resources Versus Files

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

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

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

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

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

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

Packaging Policies and Adding Resources

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

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

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

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

Resource Types

The following Starlark types represent individual resources:

`PythonModuleSource`
> Source code for a Python module. Roughly equivalent to a `.py` file.
>
> This type can also be converted to Python bytecode (roughly equivalent to a `.pyc`) when added to a resource collector.

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

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

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

`File`
> Represents a filesystem path and its content.

`starlark_tugger.FileContent`
> Represents the content of a filesystem file.
>
> This is different from `File` in that it only represents file content and doesn't have an associated path. (It is likely these 2 types will be merged someday.)

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

`starlark_tugger.FileManifest`
> Holds a mapping of relative filesystem paths to `starlark_tugger.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.)

Resource Locations

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

In-Memory

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

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

Filesystem-Relative

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

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

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

Customizing Python Packaging Policies

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

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

The primary mechanisms for doing this are:

1. Modifying the `PythonPackagingPolicy` instance's internal state. See `PythonPackagingPolicy` for the full list of object attributes and methods that can be set or called.

2. Registering a function that will be called whenever a resource is created. This enables custom Starlark code to perform arbitrarily complex logic to influence settings and enables application developers to devise packaging strategies more advanced than what PyOxidizer provides out-of-the-box.

The following sections give examples of customized packaging policies.

Changing the Resource Handling Mode

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

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

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

    policy = dist.make_python_packaging_policy()

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

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

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

Customizing Default Resource Locations

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

The accepted values are:

in-memory
> Load resources from memory.

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

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

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

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

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

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

```
        packaging_policy = policy,
    )

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

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

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

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

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

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

    return exe
```

Using Callbacks to Influence Resource Attributes

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

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

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

Functions can be leveraged to unify all resource packaging logic in a single place, making your Starlark configuration files easier to reason about.

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

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

```
        else:
            resource.add_location = "in-memory"

def make_exe():
    dist = default_python_distribution()

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

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

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

`PythonExtensionModule` Location Compatibility

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

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

- A module that is part of a Python *distribution* and is compiled into `libpython` (a *builtin* extension module).

- A module that is part of a Python *distribution* that is compiled as a standalone shared library (e.g. a `.so` or `.pyd` file).

- A non-*distribution* module that is compiled as a standalone shared library.

- A non-*distribution* module that is compiled as a static library.

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

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

- A *builtin* extension module that's part of a Python *distribution* will always be statically linked into `libpython`.

- A Windows Python distribution with a statically linked `libpython` (e.g. the `standalone_static` *distribution flavor*) is not capable of loading extension modules defined as shared libraries and only supports loading *builtin* extension modules statically linked into the binary.

- A Windows Python distribution with a dynamically linked `libpython` (e.g. the `standalone_dynamic` *distribution flavor*) is capable of loading shared library backed extension modules from the *in-memory* location. Other operating systems do not support the *in-memory* location for loading shared library extension modules.

- If the current build configuration targets Linux MUSL-libc, shared library extension modules are not supported and all extensions must be statically linked into the binary.

- If the object files for the extension module are available, the extension module may be statically linked into the produced binary.

- If loading extension modules from in-memory import is supported, the extension module will have its dynamic library embedded in the binary.

- The extension module will be materialized as a file next to the produced binary and will be loaded from the filesystem. (This is how Python extension modules typically work.)

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

Packaging Python Files

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

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

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

`PythonExecutable` Python Resources Methods

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

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

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

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

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

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

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

The following sections demonstrate common use cases.

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 *PythonExecutable.add_python_resources()* method which adds an iterable of objects representing Python resources to the set of embedded resources.

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

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

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

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

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

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

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

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

The new `make_exe()` function should look something like the following (with comments removed for brevity):

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

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

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

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

    return exe
```

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

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

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

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

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

Packaging an Application from an Existing Virtualenv

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

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

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

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

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

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

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

    return exe
```

Of course, you need a populated virtualenv!:

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

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

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

> **Warning:** When consuming a pre-populated virtualenv, there may be compatibility differences between the Python distribution used to populate the virtualenv and the Python distributed used by PyOxidizer at build and application run time.
>
> For best results, it is recommended to use a packaging method like `pip_install(...)` or `setup_py_install(...)` to use PyOxidizer's Python distribution to invoke Python's packaging tools.

Packaging an Application from a Local Python Package

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

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

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

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

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

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

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

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

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

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

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

> **Note:** In this mode, all Python resources must already be in place in their final installation layout for things to work correctly. Many `setup.py` files perform additional actions such as compiling Python extension modules, installing additional files, dynamically generating some files, or changing the final installation layout.
>
> For best results, use a packaging method that invokes a Python packaging tool (like `pip_install(...)` or `setup_py_install(...)`).

Choosing Which Packaging Method to Call

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

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

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

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

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

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

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

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

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

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

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

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

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

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

---

will build those extension modules in such a way that they aren't compatible with other machines you want to run them on.

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

Packaging Files Instead of In-Memory Resources

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

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

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

Examples of Packaging Failures

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

black

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

We start by creating a new project:

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

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

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

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

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

    return exe
```

Then let's attempt to build the application:

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

Looking good so far!

Now let's try to run it:

```
$ pyoxidizer run --path black
Traceback (most recent call last):
  File "black", line 46, in <module>
  File "blib2to3.pygram", line 15, in <module>
NameError: name '__file__' is not defined
SystemError
```

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

As the error message shows, the `blib2to3.pygram` module is trying to access `__file__`, which is not defined. As explained by *__file__ and __cached__ Module Attributes*, `PyOxidizer` doesn't set `__file__` for modules loaded from memory. This is perfectly legal as Python doesn't mandate that `__file__` be defined. But `black` (and many other Python modules) assume `__file__` always exists. So it is a problem we have to deal with.

NumPy

Let's attempt to package [NumPy](#), a popular Python package used by the scientific computing crowd.

> $ pyoxidizer init-config-file numpy

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

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

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

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

    return exe
```

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

Then let's attempt to build the application:

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

```
resolving Python distribution...
...
```

Looking good so far!

Now let's try to run it:

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

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

That's not good! What happened?

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

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

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

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

Installing Classified Resources on the Filesystem

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

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

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

This will require configuration file changes.

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

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

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

    python_config = dist.make_python_interpreter_config()
    python_config.run_module = "black"

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

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

    return exe
```

There are a few changes here.

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

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

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

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

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

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

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

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

Installing Unclassified Files on the Filesystem

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

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

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

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

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

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

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

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

    return exe
```

There are a few key lines here.

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

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

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

Anyway, let's see if this actually works:

```
$ pyoxidizer run --path numpy
...
adding extra file lib/numpy.libs/libgfortran-2e0d59d6.so.5.0.0 to .
```

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

It works!

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

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

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

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

Working with Python Extension Modules

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

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

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

Extension Module Flavors

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

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

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

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

Extension Module Restrictions

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

Known Incompatibility with Cython

PyOxidizer currently has a known incompatibility with Cython. Specifically, PyOxidizer fails to find object files that Cython builds. This can lead to missing symbols and build/link time errors.

This is tracked by https://github.com/indygreg/PyOxidizer/issues/567.

musl libc Linux Distributions Only Support Built-in Extension Modules

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

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

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

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

Windows Static Distributions Only Support Built-in Extension Modules

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

See the above section for implications on this.

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

Loading Extension Modules from `in-memory` Location

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

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

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

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

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

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

Extension Module Library Dependencies

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

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

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

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

Building with a Custom Distutils

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

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

When PyOxidizer invokes `pip` or `setup.py` to build a package, it installs a modified version of `distutils` into the invoked Python's `sys.path`. This modified `distutils` changes the behavior of some key build steps (notably how C extensions are compiled) such that the build emits artifacts that PyOxidizer can statically link into a custom binary.

For example, on Linux, PyOxidizer copies the intermediate object files produced by the build and links them into the binary containing the generated `libpython`. PyOxidizer completely ignores the shared library that is or would typically be produced.

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

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

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

Managing *Packed* Resources Data

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

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

Available Resource Data Load Modes

Embedded

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

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

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

Binary Relative Memory Mapped File

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

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

Here's an example:

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

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

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

    return exe
```

None / Disabled

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

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

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

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.

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.

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 ± ):     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 ± ):     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 ± ):     217.6 ms ±   6.4 ms    [User: 200.4 ms, System: 13.7 ms]
  Range (min ... max):   207.9 ms ... 243.1 ms    50 runs


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

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

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

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

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

The results are quite clear:

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

These results help us isolate specific areas of speedups:

- *oxidized* over *traditional* is a rough proxy for the benefits of `python -S` over `python`. Although there are other factors at play that may be influencing the numbers.

- *filesystem* over *oxidized* isolates the benefits of using PyOxidizer's importer instead of Python's default importer. The performance wins here are due to a) avoiding excessive I/O system calls to locate the paths to resources and b) functionality being implemented in Rust instead of Python.

- *in-memory* over *filesystem* isolates the benefits of avoiding explicit filesystem I/O to load Python resources. The Rust code backing these 2 variants is very similar. The only meaningful difference is that *in-memory* constructs a Python object from a memory address and *filesystem* must open and read a file using standard OS mechanisms before doing so.

From this data, one could draw a few conclusions:

- Processing of the `site` module during Python interpreter initialization can add substantial overhead.

- Maintaining an index of Python resources such that you can avoid discovery via filesystem I/O provides a meaningful speedup.

- Loading Python resources from an in-memory data structure is faster than incurring explicit filesystem I/O to do so.

Ignoring `site`

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

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

```
$ hyperfine -m 500 -- '/usr/local/bin/python3.7 -c 1' '/usr/local/bin/python3.7 -S -c 1'
Benchmark #1: /usr/local/bin/python3.7 -c 1
  Time (mean ± ):      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 ± ):      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!

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.

C and Other Native Extension Modules

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

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

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:

```python
import sys

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

Incorrect Resource Identification

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

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

Important: Please file a bug to report problems!

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

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.

PythonInterpreterConfig.sys_frozen controls whether `sys.frozen = True` is set. This can allow `PyOxidizer` to advertise itself as a *frozen* application.

In addition, the *PythonInterpreterConfig.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.

Standalone / Single File Applications with Static Linking

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

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

Building Fully Statically Linked Binaries on Linux

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

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

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

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

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

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

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

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

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

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

Rust 1.37 or newer is required for the modern musl version compatibility. And newer versions of Rust may change which version of musl they use, introducing failures similar to above. If you run into problems with a modern version of Rust, consider reporting an issue against PyOxidizer!

Once Rust's musl target is installed, you can build away:

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

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

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

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

Building Statically Linked Binaries on Windows

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

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

```
dist = default_python_distribution(flavor = "standalone_static")
```

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

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

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

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*.

Licensing Considerations

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

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

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

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

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

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

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

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

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

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.

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 `PythonInterpreterConfig.terminfo_resolution` config setting.

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

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

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

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

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

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

Using the `multiprocessing` Python Module

The `multiprocessing` Python module has special behavior and interactions with PyOxidizer.

In general, `multiprocessing` *just works* with PyOxidizer if the default settings are used: you do not need to call any functions in `multiprocessing` to enable `multiprocessing` to work with your executable.

Worker Process Spawn Method

The `multiprocessing` module works by spawning work in additional processes. It has multiple mechanisms for spawning processes and the default mechanism can be specified by calling `multiprocessing.set_start_method()`.

PyOxidizer has support for automatically calling `multiprocessing.set_start_method()` when the `multiprocessing` module is imported by `oxidized_importer.OxidizedFinder`. This behavior is configured via `PythonInterpreterConfig.multiprocessing_start_method`.

The default value is `auto`, which means that if the `multiprocessing` module is serviced by PyOxidizer's custom importer (as opposed to Python's default filesystem importer), your application **does not** need to call `multiprocessing.set_start_method()` early in its *__main__* routine, as the Python documentation says to do.

To make the embedded Python interpreter behave as `python` would, set *PythonInterpreterConfig.* *multiprocessing_start_method* to `none` in your configuration file. This will disable the automatic calling of `multiprocessing.set_start_method()`.

If `multiprocessing.set_start_method()` is called twice, it will raise `RuntimeError("context has already been set")`. This error can be suppressed by passing the `force=True` keyword argument to the function.

Buggy `fork` When Using Framework Python on macOS

The `multiprocessing` spawn methods of `fork` and `forkserver` are known to be buggy when Python is built as a *framework*.

Python by default will use the `spawn` method because of this bug.

Since PyOxidizer does not use *framework* builds of Python, `auto` mode will use `fork` on macOS, since it is more efficient than `spawn`.

`spawn` Only Works on Windows with PyOxidizer

The `spawn` start method is known to be buggy with PyOxidizer except on Windows. It is recommended to only use `fork` or `forkserver` on non-Windows platforms.

Important: If *oxidized_importer.OxidizedFinder* doesn't service the `multiprocessing` import, the default start method on macOS will be `spawn`, and this won't work correctly.

In this scenario, your application code should call `multiprocessing.set_start_method("fork", force=True)` before `multiprocessing` functionality is used.

Automatic Detection and Dispatch of `multiprocessing` Processes

When the `spawn` start method is used, `multiprocessing` effectively launches a new `sys.executable` process with arguments `--multiprocessing-fork [key=value] ....`

Executables built with PyOxidizer using the default settings recognize when processes are invoked this way and will automatically call into `multiprocessing.spawn.spawn_main()`, just as `multiprocessing.freeze_support()` would.

When `multiprocessing.spawn.spawn_main()` is called automatically, this replaces any other run-time settings for that process. i.e. your custom code will not run in this process, as this is a *multiprocessing process*.

This behavior means that `multiprocessing` should *just work* and your application code doesn't need to call into the `multiprocessing` module in order for `multiprocessing` to work.

If you want your code to be compatible with non-PyOxidizer running methods, you should still call `multiprocessing.` `freeze_support()` early in `__main__`, per the `multiprocessing` documentation. This function should no-op unless the process is supposed to be a *multiprocessing process*.

If you want to disable the automatic detection and dispatching into `multiprocessing.spawn.spawn_method()`, set *PythonInterpreterConfig.multiprocessing_auto_dispatch* to `False`.

Dependence on `sys.frozen`

`multiprocessing` changes its behavior based on whether `sys.frozen` is set.

In order for `multiprocessing` to *just work* with PyOxidizer, `sys.frozen` needs to be set to `True` (or some other truthy value). This is the default behavior. However, this setting is configurable via *PythonInterpreterConfig. sys_frozen* and via the Rust struct that configures the Python interpreter, so `sys.frozen` may not always be set, causing `multiprocessing` to not work.

Sensitivity to `sys.executable`

When in `spawn` mode, `multiprocessing` will execute new `sys.executable` processes to create a worker process.

If `sys.frozen == True`, the first argument to the new process will be `--multiprocessing-fork`. Otherwise, the arguments are `python` arguments to define code to execute.

This means that `sys.executable` must be capable of responding to process arguments to dispatch to `multiprocessing` upon process start.

In the default configuration, `sys.executable` should be the PyOxidizer built executable, `sys.frozen == True`, and everything should *just work*.

However, if `sys.executable` isn't the PyOxidizer built executable, this could cause `multiprocessing` to break.

If you want `sys.executable` to be an executable that is separate from the one that `multiprocessing` invokes, call `multiprocessing.set_executable()` from your application code to explicitly install an executable that responds to `multiprocessing`'s process arguments.

Debugging `multiprocessing` Problems

If you run into problems with `multiprocessing` in a PyOxidizer application, here's what you should do.

1. Verify you are running a modern PyOxidizer. Only versions 0.17 and newer have `multiprocessing` support that *just works*.

2. Verify the *start method*. Call `multiprocessing.get_start_method()` from your application / executable. On Windows, the value should be `spawn`. On non-Windows, `fork`. Other values are known to cause issues. See the documentation above.

3. Verify `sys.frozen` is set. If missing or set to a non-truthy value, `multiprocessing` may not work correctly.

4. When using `spawn` mode (default on Windows), verify `multiprocessing.spawn.get_executable()` returns an executable that exists and is capable of handling `--multiprocessing-fork` as its first argument. In most cases, the returned path should be the path of the PyOxidizer built executable and should also be the same value as `sys.executable`.

SSL Certificate Loading

If using the `ssl` Python module (e.g. as part of making connections to `https://` URLs), Python in its default configuration will want to obtain a list of *trusted* X.509 / SSL certificates for verifying connections.

If a list of trusted certificates cannot be found, you may encounter errors like `ssl.SSLCertVerificationError: [SSL: CERTIFICATE_VERIFY_FAILED] certificate verify failed:  unable to get local issuer certificate`.

How Python Looks for Certificates

By default, Python will likely call `ssl.SSLContext.load_default_certs()` to load the *default certificates*.

On Windows, Python automatically loads certificates from the Windows certificate store. This should *just work* with PyOxidizer.

On all platforms, Python attempts to load certificates from the default locations compiled into the OpenSSL library that is being used. With PyOxidizer, the OpenSSL (or LibreSSL) library is part of the Python distribution used to produce a binary.

The OpenSSL library hard codes default certificate search paths. For PyOxidizer's Python distributions, the paths are:

- (Windows) `C:\Program Files\Common Files\SSL\cert.pem` (file) and `C:\Program Files\Common Files\SSL\certs` (directory).

- (non-Windows) `/etc/ssl/cert.pem` (file) and `/etc/ssl/certs` (directory).

In addition, OpenSSL (but not LibreSSL) will look for path overrides in the `SSL_CERT_FILE` and `SSL_CERT_DIR` environment variables.

You can verify all of this behavior by calling `ssl.get_default_verify_paths()`:

```
$ python3.9
Python 3.9.5 (default, Apr 16 2021, 08:56:35)
[GCC 10.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import ssl
>>> ssl.get_default_verify_paths()
DefaultVerifyPaths(cafile=None, capath='/etc/ssl/certs', openssl_cafile_env='SSL_CERT_
→FILE', openssl_cafile='/etc/ssl/cert.pem', openssl_capath_env='SSL_CERT_DIR', openssl_
→capath='/etc/ssl/certs')
```

On macOS, `/etc/ssl` *should* exist, as it is part of the standard macOS install. So OpenSSL / Python should find certificates automatically.

On Windows, the default certificate path won't exist unless something that isn't PyOxidizer materializes the afore-mentioned files/directories. However, since Python loads certificates from the Windows certificate store automatically, OpenSSL / Python should be able to load certificates from PyOxidizer applications without issue.

On Linux, things are more complicated. The `/etc/ssl` directory is common, but not ubiquitous. This directory likely exists on all Debian based distributions, like Ubuntu. If the directory does not exist, OpenSSL / Python will likely fail to find certificates and summarily fail to verify connections against them.

Using Alternative Certificate Paths

PyOxidizer doesn't yet have a built-in mechanism for automatically registering additional certificates or certificate paths at run-time. Therefore, if OpenSSL / Python is unable to locate certificates, you will need to add custom logic to your application to have it look for additional certificates.

Certifi

The certifi Python package provides access to a copy of Mozilla's trusted certificates list. Using `certifi` enables you to have access to a known trusted certificates list without dependence on certificates present in the run-time environment / operating system.

Because `certifi` and its certificate list is distributed with your application, it is guaranteed to be present and certificate loading should *just work*.

To use `certifi` with PyOxidizer, you can install it as an additional package. From your Starlark configuration file:

```python
def make_exe():
    dist = default_python_distribution()
    exe = dist.to_python_executable(name="myapp")

    # Check for newer versions at https://pypi.org/project/certifi/.
    exe.add_python_resources(exe.pip_install(["certifi==2020.12.5"]))

    return exe
```

Then from your application's Python code:

```python
import certifi
import ssl

# Obtain a default ssl.SSLContext but with certifi's certificate data loaded.
ctx = ssl.create_default_context(cadata=certifi.contents())

# Or if you already have an ssl.SSLContext instance and want to load
# certifi's data in it:
ctx.load_verify_locations(cadata=certifi.contents())

# Various APIs that create connections also accept a `cadata` argument.
# Under the hood they pass this argument to construct the ssl.SSLContext.
# e.g. urllib.request.urlopen().
import urllib.request
urllib.request.urlopen(url, cadata=certifi.contents())
```

Manually Specifying Paths to Certificates

If you know the paths to certificates to use, you can specify those paths via various `ssl` APIs, often through the `cafile` and `capath` arguments. e.g.

```python
import ssl

ctx = ssl.create_default_context(capath="/path/to/ssl/certs")

import urllib.request
urllib.request.urlopen(url, capath="/path/to/ssl/certs")
```

Using Environment Variables

OpenSSL (but not LibreSSL) will look for the `SSL_CERT_FILE` and `SSL_CERT_DIR` environment variables to automatically set the CA file and directory, respectively.

You can set these within your process to point to alternative paths. e.g.

```python
import os

os.environ["SSL_CERT_DIR"] = "/path/to/ssl/certs"
```

Using the `tkinter` Python Module

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

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

Installing tcl Files

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

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

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

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

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

    return exe
```

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

tcl Files Prevent Self-Contained Executables

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

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

Building an Executable that Behaves Like `python`

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

To start, initialize a new config file:

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

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

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

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

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

    python_config = dist.make_python_interpreter_config()

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

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

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

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

    return exe

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

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

    return files

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

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

resolve_targets()
```

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

From there, build/run from the config:

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

Resource Loading Caveats

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

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

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

Binary Portability

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

Distributing User Guide

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

Overview

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

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

Using Tugger Starlark

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

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

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

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

Portability of Binaries Built with PyOxidizer

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

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

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

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

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

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

Using `pyoxidizer analyze` For Assessing Binary Portability

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

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

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

Building Windows Installers with the WiX Toolset

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

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

Extensions to Tugger Starlark Dialect

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

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

FileManifest.add_python_resource()
>   Adds a Python resource type to Tugger's `starlark_tugger.FileManifest`.

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

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

PythonExecutable.to_wix_bundle_builder()
>   Converts a `PythonExecutable` to a `starlark_tugger.WiXBundleBuilder`.
>
>   This method will produce a `starlark_tugger.WiXBundleBuilder`. that is pre-configured with appropriate settings and state for a Python application. The produced `.exe` installer should *just work*.

PythonExecutable.to_wix_msi_builder()
>   Converts a `PythonExecutable` to a `starlark_tugger.WiXMSIBuilder`.
>
>   This method will produce a `starlark_tugger.WiXMSIBuilder` that is pre-configured to install a Python application and all its support files. The MSI will install all files composing the Python application, excluding system-level dependencies.

Choosing an Installer Creation Method

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

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

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

When using *PythonExecutable.to_wix_msi_builder()* or *PythonExecutable.to_wix_bundle_builder()*, PyOxidizer will automatically add the Visual C++ Redistributable to the installer if it is required. However, the method

varies. For bundle installers, the installer will contain the official `VC_Redist*.exe` installer and this installer will be executed as part of running your application's installer. For MSI installers, Tugger will attempt to locate the `vcruntimeXXX.dll` files on your system (this requires an installation of Visual Studio) and copy these files next to your built/installed executable.s

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

Distribution Considerations for Linux

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

Exception for musl libc Binaries

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

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

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

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

Python Distribution Dependencies

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

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

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

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

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

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

Built Application Dependencies

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

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

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

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

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

Managing Binary Portability on Linux

Linux is a difficult platform to tackle for binary portability.

The best way to produce a portable Linux binary is to produce a fully statically-linked binary. There are no shared libraries to worry about and generally speaking these binaries *just work*. See *Building Fully Statically Linked Binaries on Linux* for more.

If you produce a dynamic binary with library dependencies, things are complicated.

Nearly every binary built on Linux will require linking against `libc` and will require a symbol provided by `glibc`. `glibc` versions it symbols. And when the linker resolves those symbols at link time, it usually uses the version of `glibc` being linked against. For example, if you link on a machine with `glibc` 2.19, the symbol versions in the produced binary will be against version 2.19 and the binary will load against `glibc` versions >=2.19. But if you link on a machine with `glibc` 2.29, symbol versions are against version 2.29 and you can only load against versions >= 2.29.

This means that to ensure maximum portability, you want to link against old `glibc` symbol versions. While it is possible to use old symbol versions when a more modern `glibc` is present, the path of least resistance is to build in an environment that has an older `glibc`.

A similar story plays out with a dependency on `libgcc_s.so.1`.

The default Python distributions use Debian 8 (Jessie) as their build environment. So a Debian 8 build environment is a good candidate to build on. Ubuntu 14.04, OpenSUSE 13.2, OpenSUSE 42.1, RHEL/CentOS 7, and Fedora 21 (`glibc` 2.20) are also good candidates for build environments.

Of course, if you are producing distribution-specific binaries and/or control installation (so e.g. dependencies are installed automatically), this matters less to you.

The `pyoxidizer analyze` command can be very useful for inspecting binaries for portability and alerting you to any potential issues.

Distribution Considerations for macOS

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

Operating System and Architecture Requirements

PyOxidizer has support for targeting x86_64 (Intel) and aarch64 (ARM) Apple devices. The default *Python distributions* target macOS 10.9+ for x86_64 and 11.0+ for aarch64.

Build Machine Requirements

PyOxidizer needs to link new binaries containing Python. Due to the way linking works on Apple platforms, you **must** use an Apple SDK no older than the one used to build the Python distributions or linker errors (likely undefined symbols) can occur.

PyOxidizer will automatically attempt to locate, validate, and use an appropriate Apple SDK given requirements specified by the Python distribution in use. If you have Xcode or the Xcode Commandline Tools installed, PyOxidizer should be able to locate Apple SDKs automatically. When building, PyOxidizer will print information about Apple SDK discovery. More details are printed when running `pyoxidizer --verbose`.

PyOxidizer will automatically look for SDKs in the directory specified by `xcode-select --print-path`. This path is often `/Applications/Xcode.app/Contents/Developer`. You can specify an alternative directory by setting the `DEVELOPER_DIR` environment variable. e.g.:

```
DEVELOPER_DIR=/Applications/Xcode-beta.app/Contents/Developer pyoxidizer build
```

You can override PyOxidizer's automatic SDK discovery by setting `SDKROOT` to the base directory of an Apple SDK you want to use. (If you find yourself doing this to work around SDK discovery *bugs*, please consider creating a GitHub issue to track the problem.) e.g.:

```
SDKROOT=/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/
→SDKs/MacOSX.sdk pyoxidizer build
```

Python Distribution Dependencies

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

The python-build-standalone project has gone to great lengths to ensure that the Python distributions only link against external libraries and symbols that are present on a default macOS installation.

The default Python distributions are built to target macOS 10.9 on x86_64 and 11.0 on aarch64. So they should *just work* on those and any newer versions of macOS.

Single Architecture Binaries

PyOxidizer currently only emits single architecture binaries.

Multiple architecture binaries (often referred to as *universal* or *fat* binaries) can not (yet) be emitted natively by PyOxidizer.

This means that if you distribute a binary produced by PyOxidizer and want it to run on both Intel and ARM machines, you will need to maintain separate artifacts for Intel and ARM machines or you will need to produce a *fat* binary outside of PyOxidizer.

https://github.com/indygreg/PyOxidizer/issues/372 tracks implementing support for emitting *fat* binaries from PyOxidizer. Please engage there if this feature is important to you.

Managing Portability of Built Applications

Like Linux, the macOS build environment can *leak* into the built application and introduce additional dependencies and degrade the portability of the default Python distributions.

It is common for built binaries to pull in modern macOS SDK features. A common way to prevent this is to set the `MACOSX_DEPLOYMENT_TARGET` environment variable during the build to the oldest version of macOS you want to support.

The default *Python distributions* target macOS 10.9 on x86_64 and 11.0 on aarch64.

Important: PyOxidizer will automatically set the deployment target to match what the Python distribution was built with, so in many cases you don't need to worry about version targeting.

If you wish to override the default deployment targets, set an alternative value using the appropriate environment variable.:

```
$ MACOSX_DEPLOYMENT_TARGET=10.15 pyoxidizer build
```

Apple's Xcode documentation has various guides useful for further consideration.

Distribution Considerations for Windows

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

Important: The restrictions in this document regard the run-time / target environment that a binary will run on: they do not describe the environment used to build that binary. In many cases, a binary built on Windows 10 or Windows Server 2019 will work fine on earlier operating system versions.

Readers may also find the Microsoft documentation on deployment considerations for Windows binaries a useful resource to supplement this document with more generic considerations.

Operating System Requirements

The default *Python distributions* used by PyOxidizer require Windows 8 or Windows 2012 or newer.

The official Python 3.8 Windows distributions available on www.python.org support Windows 7. PyOxidizer has chosen to drop support for Windows 7 to simplify support.

In addition to the restrictions imposed by the Python distribution in use, Rust may impose its own restrictions. However, Rust has historically produced binaries that work on Windows 8 and Windows 2012, so this likely is not an issue.

General Runtime / DLL Dependencies

The default *Python distributions* used by PyOxidizer require the Microsoft Visual C++ Redistributable and Universal CRT (UCRT).

The `standalone_dynamic` distributions (the default distribution flavor) have a run-time dependency on various 3rd party DLLs used by extensions (OpenSSL, SQLite3, etc). However, these 3rd party DLLs are part of the Python distribution and PyOxidizer should automatically install them if they are required.

All other DLL dependencies required by the default Python distributions should be core Windows operating system components and always available, even in a freshly installed Windows machine.

Application Specific Dependencies

When adding custom behavior to your application, PyOxidizer makes some effort to ensure additional dependencies (beyond the operating system, Python distribution, and Microsoft runtimes) are met. However, there are limitations to this.

When installing custom Python packages, PyOxidizer attempts to identify and install compiled Python extensions and `.dll` dependencies distributed with that package. See *Packaging Files Instead of In-Memory Resources* for more. However, there are corner cases and occasional bugs that may prevent this from working correctly.

To ensure are DLL dependencies are properly captured, it is recommend to inspect your binaries for references to missing DLLs before distributing them. The Dependency Walker tool can be used for this. `pyoxidizer analyze` may also provide useful information.

In many cases, installing a missing DLL is a matter of installing the DLL next to your application/binary by treating the DLL as an *additional file* from the Starlark configuration. See *Packaging Files Instead of In-Memory Resources* for more.

When possible, it is recommended to test your application in a freshly installed Windows environment to ensure it works. Please note that many Windows virtual machines already contain additional software and may not reflect real world deployment targets.

Managing the Visual C++ Redistributable Requirement

Binaries built with PyOxidizer often have a run-time dependency on the Microsoft Visual C++ Redistributable. These are DLLs with filenames like `vcruntime140.dll` and `vcruntime140_1.dll`.

Important: The Visual C++ Redistributable is **not** a core Windows operating system component and any distributed Windows application **must take measures to ensure the Visual C++ Redistributable is available on the remote machine** or the application may fail to run with a missing DLL error.

See Microsoft's Redistributing Visual C++ Files documentation for the canonical source on distribution requirements.

PyOxidizer has built-in features to make satisfying these requirements turnkey. Read the sections below for details of each.

Installing the Visual C++ Redistributable as Part of Your Application Installer

PyOxidizer can produce Windows `.exe` application installers that embed a copy of the Microsoft Visual C++ Redistributable installer (files named `vc_redist<arch>.exe`) and automatically run this installer during application install.

The way this works is PyOxidizer contains a reference to the URL and SHA-256 of these `vc_redist<arch>.exe` installers. When your application installer is built, these files are downloaded from Microsoft's servers and embedded in the new meta-installer. At install time, these embedded installers are executed automatically (if they need to be) and the Visual C++ files are installed at the system level, where they are available to any application.

If a newer version of the Visual C++ Redistributable files are already present, the installer should no-op instead of downgrading what's already installed.

The following Starlark functionality can be used to bundle the Visual C++ Redistributable installer as part of your application installer:

- *PythonExecutable.to_wix_bundle_builder()*
- *starlark_tugger.WiXBundleBuilder.add_vc_redistributable()*

Installing the Visual C++ Redistributable Files Locally Next to Your Binary

Another method of installing the Visual C++ Redistributable files is to distribute copies of the DLLs next to the binary that loads them. e.g. if you produce a `myapp.exe`, there will be a `vcruntime140[_1].dll` in the same directory as `myapp.exe`. Since Windows attempts to load DLLs next to the executable, if the DLLs are present, this should *just work*.

PyOxidizer supports automatically finding and copying the required DLLs in this manner. The Starlark setting controlling this behavior is *PythonExecutable.windows_runtime_dlls_mode*.

This setting effectively instructs the *PythonExecutable* building code to materialize extra files next to the binary. The Visual C++ files are treated just like any other supplementary files (like Python resources). This means that Visual C++ files will be materialized on the filesystem when running `pyoxidizer build`, `pyoxidizer run`. The files will also be present in file lists when using Starlark methods like *PythonExecutable.to_file_manifest()* or *PythonExecutable.to_wix_msi_builder()*.

This *local files* mode relies on locating DLLs on the local system. It does so using `vswhere.exe` to locate a Visual Studio installation containing the `Microsoft.VisualCPP.Redist.<version>.Latest` component (`<version>` is `14` for `vcruntime140.dll`). This should *just work* if you have Visual Studio 2017 or 2019 installed with support for building C/C++ applications. If the files cannot be found, run the Visual Studio Installer, `Modify` your installation, go to `Individual Components`, search for `redistributable`, and make sure all items are checked.

Important: It is possible to include a copy of the Visual C++ Redistributable in both your application installer and as files local to the built binary. This behavior is redundant and will likely result in the local files being used.

When including the Visual C++ Redistributable installer as part of your deployment solution, it is recommended to set *PythonExecutable.windows_runtime_dlls_mode* to `"never"` to prevent them from being redundantly installed.

Managing the Universal CRT (UCRT) Requirement

Binaries built with PyOxidizer may have a run-time dependency on the Universal C Runtime (UCRT).

The UCRT is a Windows operating system component and is always present in installations of Windows 10, Windows Server 2016, and newer. Combined with PyOxidizer's Windows version requirements, this means you don't need to worry about the UCRT unless you are targeting Windows 8 or Windows Server 2012.

PyOxidizer does not currently support automatically materializing the UCRT. See https://docs.microsoft.com/en-us/cpp/windows/universal-crt-deployment for instructions on deploying the UCRT with your application.

We are receptive to adding a feature to support more turnkey UCRT management if there is interest in it.

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.

Using Cargo with PyOxidizer Source Checkouts

PyOxidizer's source repository consists of multiple Rust projects/crates. At the root of the repository is a `Cargo.toml` defining a workspace consisting of all these crates.

Important: Building various Rust crates from source can be extremely brittle and a top-level `cargo build` will likely encounter multiple build failures.

If you want to run `cargo` from a PyOxidizer source checkout, you will likely want to limit the invocation to a single crate at a time to ensure things can build.

The following sections detail how to build various crates inside a source checkout.

`pyoxidizer` Crate

Building the `pyoxidizer` crate in isolation (e.g. `cargo build -p pyoxidizer`) should *just work*, as it is a pretty typical Rust crate.

Perhaps the only special property of this crate is that it defines both a library and an executable. So you may want to limit operations to a specific binary. e.g. `cargo build --bin pyoxidizer` or `cargo test --bin pyoxidizer`.

`python-packed-resources` Crate

This is a standard Rust crate and should always build without issue. e.g. `cargo build -p python-packed-resources`.

`python-packaging` Crate

This is a standard Rust crate and should always build without issue. e.g. `cargo build -p python-packaging` or `cargo test -p python-packaging`.

`pyembed` Crate

The `pyembed` crate provides the bulk of the run-time functionality for binaries embedding Python interpreters. Because the crate needs to consult with a Python interpreter at build time and link against it, its build configuration can be fragile.

Important: Almost all workspace build failures are somehow related to the `pyembed` crate.

In its default configuration, a Python 3.9 executable needs to be found on `PATH`. If said executable can't be found, you'll get a `No python interpreter found of version 3.*` error at build time.

To work around this, add a `python3.9` or `python3` executable to `PATH` or run `cargo build` with the `PYO3_PYTHON` environment variable pointing to a specific Python 3 executable. e.g.

> $ PYO3_PYTHON=/path/to/python3.9 cargo build -p pyembed

`python-oxidized-importer` Crate

This crate defines a Python extension module defining a Python meta path importer. See *oxidized_importer Python Extension*.

This crate needs to link against a Python interpreter and the same caveats for the `pyembed` crate apply to it as well.

Generic Python Embedding in Rust Applications

PyOxidizer can be used to produce artifacts facilitating the embedding of Python in a Rust application. This enables Rust developers to leverage PyOxidizer's technology for linking an embedded Python and managing the Python interpreter at run-time without a build-time dependency on PyOxidizer. This can greatly simplify development workflows at the cost of not being able to utilize the full power of PyOxidizer during builds. If you would like to use PyOxidizer as a build dependency, see *PyOxidizer Rust Projects* instead.

Producing Embedding Artifacts

The `pyoxidizer generate-python-embedding-artifacts` command can be used to write Python embedding artifacts into an output directory. e.g.:

```
$ pyoxidizer generate-python-embedding-artifacts artifacts
$ ls artifacts
default_python_config.rs  libpython3.a  packed-resources  pyo3-build-config-file.txt
→stdlib tcl
```

This command essentially runs `pyoxidizer run-build-script` with a default configuration file that produces artifacts suitable for generic Python embedding scenarios.

The Written Artifacts

`pyoxidizer generate-python-embedding-artifacts` will write the following files.

A Linkable Python Library

On UNIX platforms, this will likely be named `libpython3.a`. On Windows, `python3.dll` and a `pythonXY.dll` (where `XY` is the major-minor Python version, e.g. 39).

The library can be linked to provide an embedded Python interpreter.

A Rust Source File Containing a Python Interpreter Config

The `default_python_config.rs` file contains the definition of a `pyembed::OxidizedPythonInterpreterConfig` Rust struct for defining an embedded Python interpreter. The config should *just work* with the other files produced.

You can `include!(...)` this file in your Rust program if you want. Or you can ignore it and write your own configuration.

Packed Resources for the Standard Library

A file containing the *Python Packed Resources* for the Python standard library will be written. This file can be used by *oxidized_importer Python Extension* to import the standard library efficiently.

PyO3 Build Configuration

A `pyo3-build-config-file.txt` file will be written defining a configuration for the `pyo3-build-config` crate which will link the `libpython` produced by this command.

To use this configuration, set the `PYO3_CONFIG_FILE` environment variable to its **absolute** path and Python should get linked the way PyOxidizer would link it.

Python Standard Library

The `stdlib` directory will contain a copy of the Python standard library as it existed in the source distribution.

Note: `.pyc` files are often not present and PyOxidizer doesn't yet provide a turnkey way to produce these files.

Tcl/tk Support Files

The `tcl` directory will contain tcl/tk support files to support the `tkinter` Python module.

Example Workflows

Embed Python With `pyo3`

In this example, we will produce a Rust executable that uses the `pyo3` crate for interfacing with an embedded Python interpreter. We will not use PyOxidizer's `pyembed` crate or the `oxidized_importer` extension module for enhancing functionality of Python.

First, create a new Rust project:

```
$ cargo init --bin pyapp
```

Then edit its `Cargo.toml` to add the `pyo3` dependency. e.g.

```
[package]
name = "pyapp"
version = "0.1.0"
edition = "2021"

[dependencies]
pyo3 = "0.14"
```

And define a `src/main.rs`:

```
use pyo3::prelude::*;

fn main() -> PyResult<()> {
    unsafe {
        pyo3::with_embedded_python_interpreter(|py| {
            py.run("print('hello, world')", None, None)
        })
    }
}
```

Now use `pyoxidizer` to generate the Python embedding artifacts:

```
$ pyoxidizer generate-python-embedding-artifacts pyembedded
```

And finally build the Rust project using the PyO3 configuration file to tell PyO3 how to link the Python library we just generated:

```
$ PYO3_CONFIG_FILE=$(pwd)/pyembedded/pyo3-build-config-file.txt cargo run
```

If you are doing this on a UNIX-like platform like Linux or macOS, chances are this fails with an error similar to the following:

```
Could not find platform independent libraries <prefix>
Could not find platform dependent libraries <exec_prefix>
Consider setting $PYTHONHOME to <prefix>[:<exec_prefix>]
Python path configuration:
  PYTHONHOME = (not set)
  PYTHONPATH = (not set)
  program name = 'python3'
  isolated = 0
  environment = 1
  user site = 1
  import site = 1
  sys._base_executable = '/usr/bin/python3'
  sys.base_prefix = '/install'
  sys.base_exec_prefix = '/install'
  sys.platlibdir = 'lib'
  sys.executable = '/usr/bin/python3'
  sys.prefix = '/install'
  sys.exec_prefix = '/install'
  sys.path = [
    '/install/lib/python39.zip',
    '/install/lib/python3.9',
    '/install/lib/lib-dynload',
  ]
Fatal Python error: init_fs_encoding: failed to get the Python codec of the filesystem␣
↪encoding
Python runtime state: core initialized
ModuleNotFoundError: No module named 'encodings'

Current thread 0x00007ffa5abd9c80 (most recent call first):
<no Python frame>
```

This is because the embedded Python library doesn't know how to locate the Python standard library. Essentially, the compiled Python library has some hard-coded defaults for where the Python standard library is located and its default logic is to search in those paths. The references to /install are referring to the build environment for the Python distributions.

The quick fix for this is to define the PYTHONPATH environment variable to the location of the Python standard library. e.g.:

```
$ PYO3_CONFIG_FILE=$(pwd)/pyembedded/pyo3-build-config-file.txt PYTHONPATH=pyembedded/
↪stdlib cargo run
Could not find platform independent libraries <prefix>
Could not find platform dependent libraries <exec_prefix>
Consider setting $PYTHONHOME to <prefix>[:<exec_prefix>]
hello, world
```

We still get some warnings. But our embedded Python interpreter does work!

To make these config changes more permanent and to silence the remaining warnings, you'll need to customize the initialization of the Python interpreter using C APIs like the Python Initialization Configuration APIs. This requires a

fair bit of `unsafe` code.

Abstracting away the complexities of initializing the embedded Python interpreter is one of the reasons the *pyembed* Rust crate exists. So if you want a simpler approach, consider using `pyembed` for controlling the Python interpreter.

Embed Python with `pyembed`

In this example we'll use the *pyembed* crate (part of the PyOxidizer project) for managing the embedded Python interpreter.

First, create a new Rust project:

```
$ cargo init --bin pyapp
```

Then edit its `Cargo.toml` to add the `pyembed` dependency. e.g.

```
[package]
name = "pyapp"
version = "0.1.0"
edition = "2021"

[dependencies]
# Check for the latest version in case these docs are out of date.
pyembed = "0.18"
```

And define a `src/main.rs`:

```
include!("../pyembedded/default_python_config.rs");

fn main() {
    // Get config from default_python_config.rs.
    let config = default_python_config();

    let interp = pyembed::MainPythonInterpreter::new(config).unwrap();

    // `py` is a `pyo3::Python` instance.
    interp.with_gil(|py| {
        py.run("print('hello, world')", None, None).unwrap();
    });

}
```

Now use `pyoxidizer` to generate the Python embedding artifacts:

```
$ pyoxidizer generate-python-embedding-artifacts pyembedded
```

And finally build the Rust project using the PyO3 configuration file to tell PyO3 how to link the Python library we just generated:

```
$ PYO3_CONFIG_FILE=$(pwd)/pyembedded/pyo3-build-config-file.txt cargo run
...
 Finished dev [unoptimized + debuginfo] target(s) in 3.87s
  Running `target/debug/pyapp`
hello, world
```

If all goes as expected, this should *just work*!

PyOxidizer Rust Projects

PyOxidizer uses Rust projects to build binaries embedding Python. This documentation describes how they work. If you are only interested in embedding Python in a Rust application without using PyOxidizer as part of the regular development workflow, see *Generic Python Embedding in Rust Applications* for instructions.

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.

Layout

Generated Rust projects all have a similar layout:

```
$ find pyapp -type f | grep -v .git
.cargo/config
Cargo.toml
Cargo.lock
build.rs
pyapp.exe.manifest
pyapp-manifest.rc
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 running an embedded Python interpreter.

In addition, the `build = "build.rs"` helps to dynamically configure the crate.

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*.

Crate Features

The auto-generated Rust project defines a number of features to control behavior. These are documented in the sections below.

build-mode-standalone

This is the default build mode. It is enabled by default.

This build mode uses default Python linking behavior and feature detection as implemented by the `pyo3`. It will attempt to find a `python` in `PATH` or from the `PYO3_PYTHON` environment variable and link against it.

This is the default mode for convenience, as it enables the `pyembed` crate to build in the most environments. However, the built binaries will have a dependency against a foreign `libpython` and likely aren't suitable for distribution.

This mode does 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 `pyo3` crate 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.

`pyembed` has a dependency on Python 3.8+. If an older Python is detected, it can result in build errors, including unresolved symbol errors.

build-mode-pyoxidizer-exe

A `pyoxidizer` executable will be run to generate build artifacts.

The path to this executable can be defined via the `PYOXIDIZER_EXE` environment variable. Otherwise `PATH` will be used.

At build time, `pyoxidizer run-build-script` will be run. A PyOxidizer configuration file will be discovered using PyOxidizer's heuristics for doing so. `OUT_DIR` will be set if running from `cargo`, so a `pyoxidizer.bzl` next to the main Rust project being built should be found and used.

`pyoxidizer run-build-script` will resolve the default build script target by default. To override which target should be resolved, specify the target name via the `PYOXIDIZER_BUILD_TARGET` environment variable. e.g.:

```
$ PYOXIDIZER_BUILD_TARGET=build-artifacts cargo build
```

build-mode-prebuilt-artifacts

This mode tells the build script to reuse artifacts that were already built. (Perhaps you called `pyoxidizer build` or `pyoxidizer run-build-script` outside the context of a normal `cargo build`.)

In this mode, the build script will look for artifacts in the directory specified by `PYOXIDIZER_ARTIFACT_DIR` if set, falling back to `OUT_DIR`.

`global-allocator-jemalloc`

This feature will configure the Rust global allocator to use `jemalloc`.

`global-allocator-mimalloc`

This feature will configure the Rust global allocator to use `mimalloc`.

`global-allocator-snmalloc`

This feature will configure the Rust global allocator to use `snmalloc`.

`allocator-jemalloc`

This configures the `pyembed` crate with support for having the Python interpreter use the `jemalloc` allocator.

`allocator-mimalloc`

This configures the `pyembed` crate with support for having the Python interpreter use the `mimalloc` allocator.

`allocator-snmalloc`

This configures the `pyembed` crate with support for having the Python interpreter use the `snmalloc` allocator.

Using Cargo With Generated Rust Projects

Building a PyOxidizer-enabled Rust project with `cargo` is not as turn-key as it is with `pyoxidizer`. That's because PyOxidizer has to do some non-conventional things to get Rust projects to build in very specific ways. Commands like `pyoxidizer build` abstract away all of this complexity for you.

If you do want to use `cargo` directly, the following sections will give you some tips.

Linking Against Python

Autogenerated Rust projects need to link against Python. The link settings are ultimately derived from the `pyo3-build-config` crate via the dependency on `pyo3` in the `pyembed` crate. (`pyembed` is part of the PyOxidizer project.)

See *Building* for documentation on how to configure the Python linking settings of the `pyembed` crate.

Important: If you don't set environment variables to point `pyembed/pyo3` at a custom Python, Python won't be linked into your binary the way that `pyoxidizer build` would link it.

For best results, you'll want to use a Python library built the same way that PyOxidizer builds it. The `pyoxidizer generate-python-embedding-artifacts` command can be used to produce such a library along with a PyO3 configuration file for linking it. See *Generic Python Embedding in Rust Applications* for details.

Cargo Configuration

Linking a custom libpython into the final Rust binary can be finicky, especially when statically linking on Windows.

The auto-generated `.cargo/config` file defines some custom compiler settings to enable things to work. However, this only works for some configurations. The file contains some commented out settings that may need to be set for some configurations (e.g. the `standalone_static` Windows distributions).

Please consult this file if running into build errors when not building through `pyoxidizer`.

Also consider porting these linker settings to your own crate.

Building with Cargo and PyOxidizer

It is possible to use `cargo` to drive builds but still invoke `pyoxidizer` as part of the build. This is an advanced workflow that hasn't been optimized for ergonomics and it requires setting many environment variables to get things to play together nicely.

This is essentially a 2 step process:

1. Generate build artifacts consumed by the `pyembed` and `pyo3` crates.

2. Build with `cargo`.

Starting from a project freshly created with `pyoxidizer init-rust-project sample`, you'll first need to generate required build artifacts:

```
$ CARGO_MANIFEST_DIR=. \
  TARGET=x86_64-unknown-linux-gnu \
  PROFILE=debug \
  OUT_DIR=target/out \
  pyoxidizer run-build-script build.rs
```

This command will evaluate your PyOxidizer configuration file and write output files. The environment variables simulate the Cargo environment from which this command is usually called.

If all works correctly, build artifacts will be written to `target/out`.

Then you can run `cargo` to build your crate, consuming the built artifacts:

```
$ PYOXIDIZER_ARTIFACT_DIR=$(pwd)/target/out \
  PYO3_CONFIG_FILE=$(pwd)/target/out/pyo3-build-config-file.txt \
  cargo build \
    --no-default-features \
    --features "build-mode-prebuilt-artifacts global-allocator-jemalloc allocator-
↪jemalloc"
```

After building, you should find an executable in `target/debug/`.

Note: On Windows, you should remove the features referencing `jemalloc`, as this feature isn't available on Windows.

Important: When building through `cargo`, additional files are not copied into place next to the built crate. This can include required shared libraries, extension modules, and even the Python standard library. This can result in the embedded Python interpreter not working correctly.

You may need to manually copy additional files for the built binary to work as expected. The easiest way to do this is to build your project with `pyoxidizer build` and copy the files from its output.

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.

Interacting with the `pyembed` Crate

When writing Rust code to interact with a Python interpreter, your primary area of contact will be with the `pyembed` crate.

The `pyembed` crate is a standalone crate maintained as part of the PyOxidizer project. This crate provides the core run-time functionality for PyOxidizer, such as the implementation of *PyOxidizer's custom importer*. It also exposes a high-level API for initializing a Python interpreter and running code in it.

See *The pyembed Rust Crate* for full documentation on the `pyembed` crate. *Controlling Python from Rust Code* in particular describes how to interface with the embedded Python interpreter.

The following documentation will be unique to PyOxidizer's use of the `pyembed` crate.

Using the Default `OxidizedPythonInterpreterConfig`

When using a PyOxidizer-generated Rust project and that project is configured to use PyOxidizer to build (the default), that project/crate's build script will call into PyOxidizer to emit various build artifacts. This will process the PyOxidizer configuration file and write some files somewhere.

One of the files generated is a Rust source file containing a `fn default_python_config() -> pyembed::OxidizedPythonInterpreterConfig` which emits a `pyembed::OxidizedPythonInterpreterConfig` using the configuration from the PyOxidizer configuration file. This configuration is based off the *PythonInterpreterConfig* defined in the PyOxidizer Starlark configuration file.

The crate's build script will set the `DEFAULT_PYTHON_CONFIG_RS` environment variable to the path to this file, exposing it to Rust code.

This all means that to use the auto-generated `pyembed::OxidizedPythonInterpreterConfig` instance with your Rust application, you simply need to do something like the following:

```rust
include!(env!("DEFAULT_PYTHON_CONFIG_RS"));

fn create_interpreter() -> Result<pyembed::MainPythonInterpreter> {
    // Calls function from include!()'d file.
    let config: pyembed::OxidizedPythonInterpreterConfig = default_python_config();

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

Using a Custom `OxidizedPythonInterpreterConfig`

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

First, if you use an explicit `OxidizedPythonInterpreterConfig`, the *PythonInterpreterConfig* Starlark type defined in your PyOxidizer configuration file doesn't matter that much. The primary purpose of this Starlark type is to derive the default `OxidizedPythonInterpreterConfig` Rust struct. And if you are using your own custom `OxidizedPythonInterpreterConfig` instance, you can ignore most of the arguments when creating the `PythonInterpreterConfig` instance.

An exception to this is the `raw_allocator` argument/field. If you are using a custom allocator (like jemalloc, mimalloc, or snmalloc), you will need to enable a Cargo feature when building the `pyembed` crate or else you will get a run-time error that the specified allocator is not available.

`pyembed::OxidizedPythonInterpreterConfig::default()` can be used to construct a new instance, pre-populated with default values for each field. The defaults should match what the *PythonInterpreterConfig* Starlark type would yield.

The main catch to constructing the instance manually is that the custom *meta path importer* won't be able to service Python `import` requests unless you populate a few fields. In fact, if you just use the defaults, things will blow up pretty hard at run-time:

```
$ myapp
Fatal Python error: initfsencoding: Unable to get the locale encoding
ModuleNotFoundError: No module named 'encodings'

Current thread 0x00007fa0e2cbe9c0 (most recent call first):
Aborted (core dumped)
```

What's happening here is that Python interpreter initialization hits a fatal error because it can't `import encodings` (because it can't locate the Python standard library) and Python's C code is exiting the process. Rust doesn't even get the chance to handle the error, which is why we're seeing a segfault.

The reason we can't `import encodings` is twofold:

1. The default filesystem importer is disabled by default.

2. No Python resources are being registered with the `OxidizedPythonInterpreterConfig` instance.

This error can be addressed by working around either.

To enable the default filesystem importer:

```
let mut config = pyembed::OxidizedPythonInterpreterConfig::default();
config.filesystem_importer = true;
config.sys_paths.push("/path/to/python/standard/library");
```

As long as the default filesystem importer is enabled and `sys.path` can find the Python standard library, you should be able to start a Python interpreter.

Hint: The `sys_paths` field will expand the special token `$ORIGIN` to the directory of the running executable. So if the Python standard library is in e.g. the `lib` directory next to the executable, you can do something like `config.sys_paths.push("$ORIGIN/lib")`.

If you want to use the custom *PyOxidizer Importer* to import Python resources, you will need to update a handful of fields:

```
let mut config = pyembed::OxidizedPythonInterpreterConfig::default();
config.packed_resources = ...;
config.oxidized_importer = true;
```

The `packed_resources` field defines a reference to *packed resources data* (a `PackedResourcesSource` enum. This is a custom serialization format for expressing *resources* to make available to a Python interpreter. See *Python Packed Resources* for more. The easiest way to obtain this data blob is by using PyOxidizer and consuming the `packed-resources` build artifact/file, likely though `include_bytes!`. *OxidizedFinder Meta Path Finder* can also be used to produce these data structures.

Finally, setting `oxidized_importer = true` is necessary to enable `oxidized_importer.OxidizedFinder`.

Porting a Python Application to Rust

PyOxidizer can be used to gradually port a Python application to Rust. What we mean by this is that Python code in an application would slowly be rewritten in Rust.

Overview

When porting a Python application to Rust, the goal is to port Python code - and possibly Python C extension code - to Rust. Parts of the Rust code will presumably need to call into Python code and vice-versa.

When porting code to Rust, there are essentially two *flavors* of Rust code that will be written and executed:

1. *Vanilla* Rust code
2. *Python-flavored* Rust code

Vanilla Rust code is standard Rust code. It is what you would write if authoring a Rust-only project.

Python-flavored Rust code is Rust code that interacts with the Python C API. It is regular Rust code, of course, but it is littered with references to *PyObject* and function calls into the Python C API (although these function calls may be abstracted so you don't have to use `unsafe`).

These different *flavors* of Rust code dictate different approaches to porting. Both *flavors*/approaches can be used simultaneously when porting an application to Rust.

Vanilla Rust code will supplement the boilerplate Rust code that PyOxidizer uses to define and build a standalone executable embedded Python. See *Extending Rust Projects* for more.

Python-flavored Rust code typically involves writing Python extension modules in Rust. In this approach, you create a Python extension modules implemented in Rust and then make them available to the Python interpreter, which is managed by a Rust project.

Extending Rust Projects

When building an application from a standalone `pyoxidizer.bzl` file, PyOxidizer creates and builds a temporary, boilerplate Rust project behind the scenes. This Rust project has just enough code to initialize and run an embedded Python interpreter. That's the extent of the Rust code.

PyOxidizer also supports persistent Rust projects. In this mode, you have full control over the Rust project and can add custom Rust code to it as you desire. In this mode, you can run Rust code independent of the Python interpreter.

Supplementing the Rust code contained in your executable gives you the power to run arbitrary Rust code however you see fit. Here are some common scenarios this can enable:

- Implementing argument parsing in Rust instead of Python. This could allow you to parse out the sub-command being invoked and dispatch to pure Rust code paths if possible, falling back to running Python code only if necessary.

- Running a *forking* server, which doesn't start a Python interpreter until an event occurs.

- Starting a thread with a high-performance application component implemented in Rust. For example, you could run a thread servicing a high-performance logging subsystem or HTTP server implemented in Rust and have that thread interact with a Python interpreter via a pipe or some other handle.

Getting Started

To extend a Rust project with custom Rust code, you'll first want to materialize the boilerplate Rust project used by PyOxidizer:

```
$ pyoxidizer init-rust-project myapp
```

See *PyOxidizer Rust Projects* for details on the files materialized by this command.

If you are using version control, now would be a good time to add the created files to version control. e.g.:

```
$ git add myapp
$ git commit -m 'create boilerplate PyOxidizer project'
```

From here, your next steps are to modify the Rust project to do something new and different.

The auto-generated `src/main.rs` file contains the `main()` function used as the entrypoint for the Rust executable. The default file will simply instantiate a Python interpreter from a configuration, run that interpreter, then exit the process.

To extend your application with custom Rust code, simply add custom code to `main()`. e.g.

```rust
fn main() {
    println!("hello from Rust!")

    // Code auto-generated by ``pyoxidizer init-rust-project`` goes here.
    // ...
}
```

That is literally all there is to it!

To build your custom Rust project, `pyoxidizer build` is the most robust way to do that. But it is also possible to use `cargo build`.

What Can Go Wrong

`pyoxidizer` Not Found or Rust Code Version Mismatch

When using `cargo build`, the `pyoxidizer` executable will be invoked behind the scenes. This requires that executable to be on `PATH` and for the version to be compatible with the Rust code you are trying to build. (The Rust APIs do change from time to time.)

If the `pyoxidizer` executable is not on `PATH` or its version doesn't match the Rust code, you can forcefully tell the Rust build system which `pyoxidizer` executable to use:

```
$ PYOXIDIZER_EXE=/path/to/pyoxidizer cargo build
```

```
thread 'main' panicked at 'jemalloc is not available in this build configuration'
```

If you see this error, the problem is that the Python interpreter configuration says to use *jemalloc* as the memory allocator but the Rust project was built without *jemalloc* support. This is likely because the default Rust project features in `Cargo.toml` don't include *jemalloc* by default.

You can resolve this issue by either disabling jemalloc in the Python configuration or by enabling jemalloc in Rust.

To disable jemalloc, open your `pyoxidizer.bzl` file and find the definition of `allocator_backend`. You can set it to `raw_allocator="default"` so Python uses the system memory allocator instead of jemalloc.

To enable jemalloc, you have a few options.

First, you could build the Rust project with jemalloc support:

```
$ cargo build --features allocator-jemalloc
```

Or, you modify `Cargo.toml` so the *allocator-jemalloc* feature is enabled by default:

```
.. code-block:: toml
```

> [features] default = ["build-mode-pyoxidizer-exe", "allocator-jemalloc"]

jemalloc is typically a faster allocator than the system allocator. So if you care about performance, you may want to use it.

Implementing Python Extension Modules in Rust

If you want to port a Python application to Rust, chances are that you will need to have Rust and Python code interact with each other. A common way to do this is to implement Python extensions in Rust so that Rust code will be invoked as a Python interpreter is running.

There are two ways Rust-implemented Python extension modules can be consumed by PyOxidizer:

1. Define them via Python packaging tools (e.g. via a `setup.py` file for your Python package).

2. Define them in Rust code and register them as a *built-in* extension module.

Python Built Rust Extension Modules

If you've defined a Rust Python extension module via a Python package build tool (e.g. inside a `setup.py`), PyOxidizer should automatically detect said extension module as part of packaging the corresponding Python package: there is no need to take special action to tell PyOxidizer it is a Rust extension, as this is all handled by Python packaging tools invoked as part of processing your `pyoxidizer.bzl` file.

See *Packaging User Guide* for more.

The topic of authoring Python extension modules implemented in Rust is arguably outside the scope of this documentation. A search engine search for `Rust Python extension` should set you on the right track.

Built-in Rust Extension Modules

A Python extension module is defined as a `PyInit__<name>` function which is called to initialize an extension module. Typically, Python extension modules are compiled as standalone shared libraries, which are then loaded into a process, after which their `PyInit__<name>` function is called.

But Python has an additional mechanism for defining extension modules: *built-ins*. A *built-in* extension module is simply an extension module whose `PyInit__<name>` function is already present in the process address space. Typically, these are extensions that are part of the Python distribution itself and are compiled directly into `libpython`.

When you instantiate a Python interpreter, you give it a list of the available *built-in* Python extension modules. And PyOxidizer's `pyembed` crate allows you to supplement the default list with custom extensions.

To use *built-in* extension modules implemented in Rust, you'll need to implement said extension module in Rust, either as part of your application's Rust crate or as part of a different crate. Either way, you'll need to extend the boilerplate Rust project code (see *Extending Rust Projects*) and tell it about additional *built-in* extension modules. See *Adding Extension Modules At Run-Time* for instructions on how to do this.

The tricky part here is implementing your Rust extension module.

You probably want to use the cpython or PyO3 Rust crates for interfacing with the CPython API, as these provide an interface that is more ergonomic and doesn't require use of `unsafe { }`. Use of these crates is beyond the scope of the PyOxidizer documentation.

If you attempt to use the `cpython` or `PyO3` macros for defining a Python extension module, you'll likely run into problems because these assume that extension modules are standalone shared libraries, which isn't the case for *built-in* extension modules!

If you attempt to use a separate Rust crate to define your extension module, you may run into Python symbol issues at link time because the build system for the `cpython` and `PyO3` crates will use their own logic for locating a Python interpreter and that interpreter may not have a configuration that is compatible with the one embedded in your PyOxidizer binary!

At the end of the day, all you need to register a *built-in* extension module with PyOxidizer is an `extern "C" fn ()` `-> *mut python3_sys::PyObject`. Here is the boilerplate for defining a Python extension module in Rust (this uses the `cpython` crate).

```rust
use python3_sys as pyffi;
use cpython::{PyErr, PyModule, PyObject};

static mut MODULE_DEF: pyffi::PyModuleDef = pyffi::PyModuleDef {
    m_base: pyffi::PyModuleDef_HEAD_INIT,
    m_name: std::ptr::null(),
    m_doc: std::ptr::null(),
    m_size: std::mem::size_of::<ModuleState>() as isize,
    m_methods: 0 as *mut _,
    m_slots: 0 as *mut _,
    m_traverse: None,
    m_clear: None,
    m_free: None,
};

#[allow(non_snake_case)]
pub extern "C" fn PyInit_my_module() -> *mut pyffi::PyObject {
    let py = unsafe { cpython::Python::assume_gil_acquired() };

    unsafe {
        if MODULE_DEF.m_name.is_null() {
```

(continues on next page)

```
            MODULE_DEF.m_name = "my_module".as_ptr() as *const _;
            MODULE_DEF.m_doc = "usage docs".as_ptr() as *const _;
        }
    }

    let module = unsafe { pyffi::PyModule_Create(&mut MODULE_DEF) };

    if module.is_null() {
        return module;
    }

    let module = match unsafe { pyffi::from_owned_ptr(py, module).cast_into::<PyModule>
→(py) } {
        Ok(m) => m,
        Err(e) => {
            PyErr::from(e).restore(py);
            return std::ptr::null_mut();
        }
    };

    match module_init(py, &module) {
        Ok(()) => module.into_object().steal_ptr(),
        Err(e) => {
            e.restore(py);
            std::ptr::null_mut()
        }
    }
}
```

If you want a concrete example of what this looks like and how to do things like define Python types and have Python functions implemented in Rust, do a search for `PyInit_oxidized_importer` in the source code of the `pyembed` crate (which is part of the PyOxidizer repository) and go from there.

The documentation for authoring Python extension modules and using the Python C API is well beyond the scope of this document. A good place to start is the official documentation.

Frequently Asked Questions

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

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

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.

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.

Why is Python 3.8 Required?

Python 3.8 contains a new C API for controlling how embedded Python interpreters are started. This makes the run-time code that native binaries execute much, much simpler.

PyOxidizer versions up to 0.7 supported Python 3.7. But a decision was made to require Python 3.8 because the run-time code to manage the Python interpreter was vastly simpler and less prone to bugs. Given that Python 3.8 is mostly backwards compatible with Python 3.7, this wasn't perceived as a significant annoyance.

`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 `PYO3_PYTHON` environment variable to the path of a Python 3.8+ executable and try again. e.g.:

```
# UNIX
$ export PYO3_PYTHON=/usr/bin/python3.9
# Windows
$ SET PYO3_PYTHON=c:\python39\python.exe
```

Note: The `pyoxidizer` tool should take care of setting `PYO3_PYTHON` and prevent this error. If you see this error and you are building with `pyoxidizer`, it is a bug that should be reported.

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.

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.

Can Applications Import Python Modules from the Filesystem?

Yes!

While PyOxidizer supports importing Python resources from in-memory, it also supports filesystem-based import like traditional Python applications.

This can be achieved by adding Python resources to a non *in-memory* resource location (see *Managing How Resources are Added*) or by enabling Python's standard filesystem-based importer by enabling `filesystem_importer=True` (see *PythonInterpreterConfig*).

`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.

`vcruntime140.dll was not found` Error on Windows

Binaries built with PyOxidizer often have a dependency on the Visual C++ Redistributable Runtime, or `vcruntime140.dll`. If this file is not present on your system or in a path where the built binary can find it, you'll get an error about this missing file when attempting to run/load the binary.

PyOxidizer has some support for managing this file for you. See *Managing the Visual C++ Redistributable Requirement* for more.

If PyOxidizer is not materializing this file next your built binary, either you've disabled this functionality via your configuration file (see *PythonExecutable.windows_runtime_dlls_mode*) or PyOxidizer could not find the Visual Studio component providing this file.

The quick fix for this is to install the Visual C++ Redistributable runtime globally on your system. Simply go to https://support.microsoft.com/en-us/topic/the-latest-supported-visual-c-downloads-2647da03-1eea-4433-9aff-95f26a218cc0 and download and install the appropriate platform installer for `Visual Studio 2015, 2017 and 2019`.

If you want PyOxidizer to materialize the DLL(s) next to your built binary, you'll need to install Visual Studio with the `Microsoft.VisualCPP.Redist.14.Latest` component (you will typically get this component if installing support for building C/C++ applications).

`ld: unsupported tapi file type '!tapi-tbd' in YAML file` on macOS When Building

If you see this error when building on macOS, it means that the linker (likely Clang) being used is not able to read the `.tbd` files from a more modern Apple SDK.

PyOxidizer requires using an Apple SDK no older than the one used to build the Python distributions being embedded (see *Build Machine Requirements*). So the only recourse to this problem is to use a more modern linker.

On Apple platforms, it is common to use the clang/linker from an Xcode or Xcode Commandline Tools install. So the problem can usually be fixed by upgrading Xcode or the Xcode Commandline Tools.

Project Status

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

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.

Major Missing Features

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.

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*.

Incomplete `pyoxidizer` Commands

`pyoxidizer analyze` aren't fully implemented.

There is no `pyoxidizer upgrade` command.

Work on all of these is planned.

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.

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.

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.

Lesser Missing Features

Python Version Support

Python 3.8, 3.9, and 3.10 are currently supported. Older versions of PyOxidizer (through version 0.7) supported Python 3.7. See *Why is Python 3.8 Required?* for why we require these Python versions.

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.

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).

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.

Configuration Files

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

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.

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.

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.

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).

Module Import Performance Tracing

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

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

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.

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.

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.

Comparisons to Other Tools

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

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

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

PyInstaller

PyInstaller is a tool to convert regular python scripts to *standalone* executables. The standard packaging produces a tiny executable and a custom directory structure to host dynamic libraries and Python code (zipped compiled bytecode).

`PyInstaller` can produce a self-contained executable file containing your application, however, at run-time, PyInstaller will extract binary files and a custom ZlibArchive to a temporary directory then import modules from the filesystem.

`PyOxidizer` often skips this step and loads modules directly from memory using zero-copy. This makes `PyOxidizer` executables significantly faster to start when this feature is employed.

When `PyOxidizer` is running in single-file mode, it needs to build all binary dependencies from source to facilitate static linking. Although this behavior is optional and `PyOxidizer` can also work with pre-built binary Python packages.

A current difference between the tools is that `PyInstaller` generally has better support for binary dependencies. `PyInstaller` knows how to find runtime dependencies and allows a lot of not-easy-to-build packages like PyQT to work out of the box. With `PyOxidizer`, you could need to add sufficient complexity to its configuration files to get things to work.

py2exe

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

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

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

`py2exe` and `PyOxidizer` both employ a clever trick on Windows that allows loading DLLs from memory. This enables DLLs to be embedded in an executable so you can ship a single `.exe` and not have to worry about bundling DLLs as separate files. (`PyOxidizer` is using the same in-memory DLL loading library as `py2exe`.)

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

---

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

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.

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.

Shiv

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

Shiv requires the target system to have a Python executable and for the target to support shebangs in executable files. This is acceptable for controlled environments where Python is installed and Python shebangs work. It isn't acceptable for environments where you can't guarantee an appropriate Python executable is installed/available.

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

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.

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.

Docker / Running a Container

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

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

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

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

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`.

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.

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.

Bazel

Bazel has Python rules for building Python binaries and libraries. From a high level, it works similarly to how PyOxidizer's Starlark config files allow you to perform much of the same actions.

The executables produced by `py_binary` are significantly different from what PyOxidizer does, however.

An executable produced by `py_binary` is a glorified self-executing zip file. At run time, it extracts Python resources to a temporary directory and then runs a Python interpreter against them. The approach is similar in nature to what Shiv and PEX do.

PyOxidizer, by contrast, produces a specialized binary containing the Python interpreter and allows you to embed Python resources inside that binary, enabling Python modules to be imported without the overhead of writing a temporary directory and extracting a zip file.

Contributing to PyOxidizer

This page documents how to contribute to PyOxidizer.

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.

As a Developer

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

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

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

The most important command for contributors to know how to run is `cargo run --bin pyoxidizer`. This will compile the `pyoxidizer` executable program and run it. Use it like `cargo run --bin pyoxidizer -- init ~/tmp/myapp` to run `pyoxidizer init ~/tmp/myapp` for example.

The `Cargo.toml` in the root of the repository defines a Cargo workspace containing many crates.

Some crates have build/link dependencies against a Python interpreter. If you run into build errors, try targeting just the crate you care about by adding the `-p` argument. e.g. `cargo build -p pyoxidizer`. Or exclude troublesome crates like `cargo build --workspace --exclude python-oxidized-importer --exclude pyembed`.

Financial Contributions

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

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

Project History

Work on PyOxidizer started in November 2018 by Gregory Szorc.

Blog Posts

- Announcing the 0.9 Release of PyOxidizer (2020-10-18)
- Announcing the 0.8 Release of PyOxidizer (2020-10-12)
- Using Rust to Power Python Importing with oxidized_importer (2020-05-10)
- PyOxidizer 0.7 (2020-04-09)
- C Extension Support in PyOxidizer (2019-06-30)
- Building Standalone Python Applications with PyOxidizer (2019-06-24)
- PyOxidizer Support for Windows (2019-01-06)
- Faster In-Memory Python Module Importing (2018-12-28)
- Distributing Standalone Python Applications (2018-12-18)

Version History

0.21.0

Released June 4, 2022.

Backwards Compatibility Notes

- The minimum Rust version has been changed from 1.56 to 1.60 to facilitate use of features required by some Rust crates.
- The default Python version is 3.10 (instead of 3.9).

Bug Fixes

- Fixed regression in the behavior of various `pyoxidizer` commands which prevented them from working without specifying any arguments. This regression was introduced in 0.20 with the upgrade of the `clap` crate to version 3.1. (#523)

- PyO3 Rust crates upgraded from 0.16.4 to 0.16.5. The upgrade fixes compatibility issues with Python 3.10 that could lead to runtime crashes or incorrect behavior in many configurations.

- Fixed a runtime panic when incorrect attribute assignments were attempted on the `PythonExtensionModule`, `PythonPackageDistributionResource`, and `PythonPackageResource` Starlark types. (#561)

- We no longer panic when encountering invalid UTF-8 when reading process output of various `python` invocations. Previously, invocations of `pip`, `setup.py`, and other processes could result in a panic if invalid UTF-8 was emitted. (#579)

New Features

- Default CPython distributions upgraded from 3.8.12, 3.9.10, and 3.10.2 to 3.8.13, 3.9.13, and 3.10.4, respectively. See additional changes in these distributions at https://github.com/indygreg/python-build-standalone/releases/tag/20220318, https://github.com/indygreg/python-build-standalone/releases/tag/20220501, and https://github.com/indygreg/python-build-standalone/releases/tag/20220528.

- The default Python version is now 3.10 (instead of 3.9).

- The mechanism for handling software licenses has been overhauled.

  - The formatting of licenses during building has changed significantly.

  - Rust licensing information is now dynamically derived at build time rather than derived from a static list. The Rust components with annotated licensing should be more accurate as a result.

  - *PythonExecutable* Starlark types now write out a file containing licensing information for software components within the binary. This restores a feature that was dropped in version 0.5. The name of the file (or disabling of the feature) can be controlled via the *PythonExecutable.licenses_filename* attribute.

  - A new `pyoxidizer rust-project-licensing` command for printing licensing information for Rust projects. This can be used to help debug Rust licensing issues or to generate licensing content for any Rust project.

  - A *PythonExecutable.add_cargo_manifest_licensing()* Starlark method for registering the licensing information for a `Cargo.toml` Rust project. This can be used by Rust projects wishing to have their licensing information captured.

- Initial support for `aarch64-unknown-linux-gnu` Python distributions. The distributions are now defined and PyOxidizer should use them when appropriate. However, the distributions aren't yet well tested. So feedback on their operation via GitHub issues would be appreciated!

- `aarch64-apple-darwin` (Apple M1) now has a default Python 3.8 distribution. This distribution is not very well tested and use of a newer distribution is strongly preferred.

Other Relevant Changes

- Managed Rust toolchain upgraded from 1.56.1 to 1.61.0.

- Starlark *PythonInterpreterConfig* documentation has been changed to refer to *Python Interpreter Configuration Data Structures*. The latter is automatically derived from the canonical Rust source code. So the change effectively results in a single, consistent set of documentation for interpreter configuration.

- The mechanism for locating the Apple SDK now uses the apple-sdk Rust crate. The new crate work similarly to how our custom logic was working before. But there may be subtle changes in behavior. If you see new build errors related to Apple SDKs in this release, don't hesitate to create an issue. One notable change is that we will now look for SDKs in all `/Applications/Xcode*.app` directories. In environments like GitHub Actions, this will result in finding and using the newest installed SDK.

- The jemalloc allocator in built binaries has been upgraded to version 5.3.

- The auto-generated Rust project created during binary building is now explicitly licensed to the public domain.

- Derivation of a custom `libpython` library archive now sometimes uses pure Rust code instead of calling external processes. There should be no meaningful change in behavior except for build output being more concise.

- Auto-generated Rust projects now contain an empty `[workspace]` table in their `Cargo.toml`. This enables auto-generated projects to be nested under an existing workspace directory without Cargo complaining. This approach is more robust in the common case where the Rust project isn't part of a larger workspace.

0.20.0

Released March 6, 2022.

Bug Fixes

- The pyembed crate will now properly call `multiprocessing.spawn.spawn_main()` when the `multiprocessing` auto dispatch function as configured by *PythonInterpreterConfig.multiprocessing_start_method* is set to `spawn`. This resolves a `TypeError: spawn_main()` `missing 1 required positional argument: 'pipe_handle'` run-time error that would occur in this configuration. (#483)

- The pyembed crate better handles errors during interpreter initialization. This fixes a regression to the error handling introduced by the port to PyO3 in version 0.18.0. (#481)

- The `pyembed::MainPythonInterpreter` type is now more resilient against calling into a finalized Python interpreter. Before, calling `py_runmain()` (possibly via `run()`) could result in a segfault in the type's `Drop` implementation.

- `oxidized_importer.OxidizedFinder.find_distributions()` now properly normalizes names when performing comparisons. Previously, the specified `name` was properly normalized but it was compared against un-normalized strings. Both the search and candidate names are now normalized when performing a comparison. This should fix cases where case and other special character differences could result in a distribution not being found. (#488)

- A potential crash when importing extension modules from memory on Windows was fixed. The crash could occur due to discrepancy in Python reference counting when multi-phase initialization was used. (#490)

- Our patched `distutils` only sets `Py_BUILD_CORE_BUILTIN` on Windows. This fixes errors building at least the `grpcio` package outside of Windows.

- When using a modified `distutils` to install Python packages, the SETUPTOOLS_USE_DISTUTILS=stdlib environment variable is now set. This prevents `setuptools` from using its vendored copy of `distutils` and ignoring our modifications. Before this change, packages with extension modules may not have built correctly, resulting in build or run-time errors.

Backwards Compatibility Notes

- The `pyembed::MainPythonInterpreter` Rust API for controlling embedded Python interpreters has been refactored. Various methods now take `&self` instead of `&mut self`. `acquire_gil()` and `release_gil()` have been removed (use `with_gil()` instead). `MainPythonInterpreter` instances now release the GIL after initialization. Before, the GIL would be perpetually held by the instance. Consumers that were calling `PyEval_SaveThread()` to release the GIL to work around this should delete calls to that function, as the GIL is now released automatically. APIs on `MainPythonInterpreter` will acquire the GIL as necessary. (#500)

New Features

- Support for Python 3.10 on all previously supported platforms. Python 3.9 is still the default Python version. Target Python 3.10 by passing `python_version = "3.10"` to the *default_python_distribution()* Starlark function.

- Default Python distributions upgraded from 3.9.7 to 3.9.10. Various library dependencies have also been upgraded. See https://github.com/indygreg/python-build-standalone/releases/tag/20211017 and https://github.com/indygreg/python-build-standalone/releases/tag/20220222 for the full list of changes.

- The `pyembed::MainPythonInterpreter` Rust struct has gained a `with_gil()` function for executing a function with the Python GIL held.

Other Relevant Changes

- PyO3 Rust crate upgraded from version `0.14` to `0.16`.
- Managed Rust toolchain upgraded from 1.56.0 to 1.56.1.

0.19.0

Released October 28, 2021.

Changes

- `p12` Rust crate updated to avoid dependency on version yanked from crates.io (version 0.18.0 could not be installed via `cargo` in some configurations because of this).

- PyOxidizer's documentation is now more isolated from the rest of the projects in the same repository.

0.18.0

Released October 24, 2021.

Bug Fixes

- The `unable to identify deployment target environment variable for macosx (please report this bug)` error message seen when attempting to use a too-old macOS SDK has been replaced to automatically assume the use of `MACOSX_DEPLOYMENT_TARGET`. A warning message that this will possibly lead to build failures is printed. (#414)

- Invocation of `signtool.exe` on Windows now always passes `/fd SHA256` by default. Previously, we did not specify `/fd` unless a signing algorithm was explicitly requested. Newer versions of `signtool.exe` appear to insist that `/fd` be specified.

- Default Python distributions now properly advertise system library dependencies on Linux and macOS. The older distributions failed to annotate some library dependencies, which could lead to missing symbol errors in some build environments.

- Linux default Python distributions no longer utilize the `pthread_yield()` function, enabling them to be linked against glibc 2.34+, which deprecated the symbol. (#463)

- Python `.whl` resources parsing now ignores directories. Previously, directories may have been emitted as 0-sized resources.

- In some `pyoxidizer.bzl` configurations, an error would occur due to attempting to write a built executable to a directory that doesn't exist. This should no longer occur. (#447)

Backwards Compatibility Notes

- The minimum Rust version has been changed from 1.52 to 1.56 to facilitate use of the newest versions of some Rust crates, Rust 2021 edition, and some Cargo features to enhance linker control.

- The run-time Rust code for interfacing with the Python interpreter now uses the PyO3 crate instead of cpython. The code port was quite extensive and while we believe all important run-time functionality is backwards compatible, there are possibly subtle differences in behavior. Please file GitHub issues to report any undesired changes in behavior.

- Development workflows relying on specifying the `PYTHON_SYS_EXECUTABLE` environment variable have changed to use `PYO3_PYTHON`, as the environment variable has changed between the `cpython` and `pyo3` crates.

- The `pyembed` crate no longer has `cpython-link-unresolved-static` and `cpython-link-default` Cargo features. Autogenerated Rust projects also no longer have `cpython-link-unresolved-static` and `cpython-link-default` features (which existed as proxies to these features in the `pyembed` crate).

- The `pyoxidizer add` command has been removed because it didn't work as advertised.

- The `pyembed` crate no longer has `build-mode-*` features and its build script no longer attempts to integrate with PyOxidizer or its build artifacts.

- The `pyembed` crate no longer annotates a `links` entry.

- The mechanism by which auto-generated Rust projects integrate with the `pyembed` crate has changed substantially. If you had created a standalone Rust project via `pyoxidizer init-rust-project`, you may wish to create a fresh project and reconcile differences in the auto-generated files to ensure things now build as expected.

- Default Python distributions on macOS aarch64 are now built with macOS SDK 11.3. macOS x86_64 are now built with macOS SDK 11.1.

New Features

- Default Python distributions upgraded from 3.8.11 and 3.9.6 to 3.8.12 and 3.9.7. Various library dependencies have also been upgraded. See https://github.com/indygreg/python-build-standalone/releases/tag/20211012 and https://github.com/indygreg/python-build-standalone/releases/tag/20211017 for the full list of changes.

- When in verbose mode, messages will be printed displaying the actual result of every request to add a resource. Before, the Starlark code would emit a message like `adding extension module foo` before requesting the addition and the operation could no-op. This behavior was misleading and hard to debug since it often implied a resource was added when in fact it wasn't! The new behavior is for the resource collector to tell its callers exactly what actions it took and for the actual results to be displayed to the end-user. This should hopefully make it easier to debug issues with how resources are added to binaries.

- A new `pyoxidizer generate-python-embedding-artifacts` command that writes out file artifacts useful for embedding Python in another project. The command essentially enables other projects to use PyOxidizer's embeddable Python distributions without using PyOxidizer to build them. See *Generic Python Embedding in Rust Applications* for documentation.

Other Relevant Changes

- When Apple SDKs are found via the `SDKROOT` environment variables, a hard error now occurs if that SDK does not support the target platform or deployment target. Previously, we would allow the use of the SDK, only to likely encounter a hard-to-debug compile error. If the SDKs version does not meet the desired minimum version, a warning message is printed but the build proceeds. (#431)

- The `pyembed` crate (which built binaries use to interface with an embedded Python interpreter) now uses the `pyo3` crate instead of `cpython` to interface with Python APIs.

- Nightly Cargo features are no longer required on Windows. (Courtesy of PyO3 giving us complete control over how Python is linked.)

- The mechanism by which built binaries link against `libpython` has been significantly refactored. Before, the `cpython` crate would link against a partial `libpython` in many configurations and the `pyembed` crate would *complete* the linking with a library defined by PyOxidizer. With the PyO3 crate supporting a configuration file to configure all attributes of linking, the PyO3 crate now fully links against `libpython` and `pyembed` doesn't care about linking `libpython` at all.

- The `pyembed` crate is now generic and no longer attempts to integrate with `pyoxidizer` or its build artifacts. The crate can now be used by any Rust application wishing to embed a Python interpreter.

- The `oxidized_importer` Python extension has been extracted from the `pyembed` crate and is now defined in the `python-oxidized-importer` crate. The `pyembed` crate now depends on this crate to provide the custom importer functionality.

- Previous versions of PyOxidizer would not build on Rust 1.56+ due to incompatibilities with an older version of the `starlark` crate. The crate was upgraded to version 0.3.2 to fix this issue.

- Managed Rust toolchain upgraded from 1.54.0 to 1.56.0.

0.17.0

Released August 8, 2021.

Backwards Compatibility Notes

- The minimum Rust version has been changed from 1.46 to 1.52 to facilitate use of const generics and some other stabilized APIs.

- *starlark_tugger.PythonWheelBuilder.write_to_directory()* now interprets relative paths as relative to the currently configured build path, not relative to the process's current working directory.

- Various Starlark types now ensure that they cannot get out of sync when cloned. Previously, various Starlark types would clone their underlying Rust struct when the Starlark value was cloned. This could cause Starlark value instances to become out of sync with each other if one value was mutated. Now, all mutable Starlark types should hold a reference to a shared resource, ensuring that cloned Starlark values all refer to the same instance. This change could result in Starlark configuration files behaving differently. For example, before you could mutate a value in a function call and that mutation wouldn't be reflected in the caller's Starlark value. Now, it would be.

- *oxidized_importer.OxidizedFinder* will now automatically call `multiprocessing.set_start_method()` when it imports the `multiprocessing` module. Applications that explicitly call `multiprocessing.set_start_method()` may fail with `RuntimeError("context has already been set")` as a result of this change. See *Using the multiprocessing Python Module* for workarounds.

- *PythonInterpreterConfig.sys_frozen* now defaults to `True` instead of `False`.

- *starlark_tugger.WiXInstaller*, *starlark_tugger.WiXMSIBuilder*, and *starlark_tugger.WiXBundleBuilder* instances now always default to building an installer for the `x64` WiX architecture. Previously, the default architecture would be derived from the architecture of the running binary.

- *starlark_tugger.WiXMSIBuilder* instances no longer have a `target_triple` attribute.

Bug Fixes

- The default target triple is now derived from the target triple of the running binary, not the environment the running binary was built in. In many cases these would be identical. However, they would diverge if the binary was cross-compiled.

- The default Python packaging policy now disables bytecode generation for various modules in the Python standard library in the `lib2to3.tests` and `test` packages that contain invalid Python 3 source code and would fail to compile to bytecode. This should enable Python resources to compile without error when setting *PythonPackagingPolicy.include_test* to `True`, without requiring a custom resource handling callback to disable bytecode generation. (#147)

- Applications with hyphens (-) in their name now build properly on Windows. Previously, there would be a cryptic build failure when running `rc.exe`. (#402)

- The ELF (read: Linux) binaries in the default Python distributions have changed how they perform dynamic library loading so they should always pick up the libpython from the distribution. Before, `LD_LIBRARY_PATH` environment variables could result in the wrong libpython being loaded and errors like `ModuleNotFoundError: No module named '_posixsubprocess'` being encountered. (#406)

New Features

- Default Python distributions upgraded from 3.8.10 and 3.9.5 to 3.8.11 and 3.9.6. Various library dependencies have also been upgraded. See https://github.com/indygreg/python-build-standalone/releases/tag/20210724 for the full list of changes.

- `oxidized_importer.OxidizedFinder` now calls `multiprocessing.set_start_method()` when the `multiprocessing` module is imported. The behavior of this feature can be controlled via the new `PythonInterpreterConfig.multiprocessing_start_method` attribute. On macOS, the default start method is effectively switched from `spawn` to `fork`, as PyOxidizer supports this mode. The main execution routine of built executables also now recognizes the *signatures* of processes spawned for `multiprocessing` use and will automatically function accordingly. This behavior can be disabled via `PythonInterpreterConfig.multiprocessing_auto_dispatch`. These changes mean that `multiprocessing` should *just work* when default settings are used. See *Using the multiprocessing Python Module* for full documentation of `multiprocessing` interactions with PyOxidizer.

- The `oxidized_importer.OxidizedFinder.pkg_resources_import_auto_register` now exposes whether the `oxidized_importer.OxidizedFinder` instance will automatically register itself with `pkg_resources`.

- `starlark_tugger.AppleUniversalBinary` has gained the `starlark_tugger.AppleUniversalBinary.write_to_directory()` method.

- `starlark_tugger.FileContent` has gained the `starlark_tugger.FileContent.write_to_directory()` method.

- `starlark_tugger.MacOsApplicationBundleBuilder` has gained the `starlark_tugger.MacOsApplicationBundleBuilder.write_to_directory()` method.

- `starlark_tugger.WiXInstaller` has gained the `starlark_tugger.WiXInstaller.to_file_content()` and `starlark_tugger.WiXInstaller.write_to_directory()` methods.

- `starlark_tugger.WiXMSIBuilder` has gained the `starlark_tugger.WiXMSIBuilder.to_file_content()` and `starlark_tugger.WiXMSIBuilder.write_to_directory()` methods.

- `starlark_tugger.WiXBundleBuilder` has gained the `starlark_tugger.WiXBundleBuilder.to_file_content()` and `starlark_tugger.WiXBundleBuilder.write_to_directory()` methods.

- `starlark_tugger.WiXInstaller` has gained the `starlark_tugger.WiXInstaller.arch` attribute to retrieve and modify the architecture of the WiX installer being built.

- The constructors for `starlark_tugger.WiXInstaller`, `starlark_tugger.WiXMSIBuilder`, and `starlark_tugger.WiXBundleBuilder` now accept an `arch` argument to control the WiX architecture of the installer.

- `starlark_tugger.WiXMSIBuilder` has gained the `starlark_tugger.WiXMSIBuilder.arch` attribute to define the architecture of the WiX installer being built.

Other Relevant Changes

- Managed Rust toolchain upgraded from 1.52.0 to 1.54.0.

- Visual C++ Redistributable installers upgraded from version 14.28.29910 to 14.29.30040.

0.16.0

Released May 9, 2021.

Bug Fixes

- The Rust build environment now always sets RUSTC to the path to the Rust compiler that we've detected. This should hopefully prevent `could not execute process `rustc...` errors in environments where Rust is not otherwise installed.

- Pre-release `pyoxidizer` binaries built in CI should now generate `Cargo.lock` files in Rust projects that work with `cargo build --frozen`.

- Managed Rust toolchains now properly install the Rust stdlib for cross-compiles. Previously, the logs said it was installing them but didn't actually, leading to build failures due to an incomplete Rust toolchain.

- The file modified times in files extracted from Python distributions are now set to the current time. Previously, we preserved the mtime in the tar archive and the Windows archives had an mtime of the UNIX epoch. This could lead to runtime errors in `pip` due to pip attempting to create a zip file of itself and Python's zip file code not supporting times older than 1980. If you see a `ValueError:  ZIP does not support timestamps before 1980` error when running `pip` as part of running PyOxidizer, you are hitting this bug. You will need *modernize* the mtimes in the extracted Python distributions. The easiest way to do this is to clear PyOxidizer's Python distribution cache via `pyoxidizer cache-clear`.

- MSI installers built with *starlark_tugger.WiXMSIBuilder* should now properly update the `PATH` environment variable if that installation option is active. This affects PyOxidizer's own MSI installers.

New Features

- The new *starlark_tugger.PythonWheelBuilder* type can be used to create Python wheel (`.whl`) files. It is currently rather low-level and doesn't have any integrations with other Starlark Python types. But it does allow you to create Python wheels from file content. PyOxidizer uses the type for building its own wheels (previously it was using `maturin`).

Other Relevant Changes

- When building for Apple platforms, we now check for a compatible Apple SDK earlier during binary building (when compiling a custom `config.c` for a custom `libpython`). This should surface missing dependencies sooner in the build and potentially replace cryptic compiler error messages with an actionable one about the Apple SDK. Related to this, we now target a specific Apple SDK when compiling the aforementioned source file to ensure that the same, validated SDK is consistently used.

0.15.0

Released May 6, 2021.

Backwards Compatibility Notes

- The order of the `content` and `path` arguments to `starlark_tugger.MacOsApplicationBundleBuilder.add_macos_file()` and `starlark_tugger.MacOsApplicationBundleBuilder.add_resources_file()` has been reversed and `path` now defaults to `None`. While technically a backwards incompatible change, the old methods weren't usable in prior versions of PyOxidizer because the `starlark_tugger.FileContent` Starlark type couldn't be instantiated!

- `starlark_tugger.FileManifest` now performs path normalization and checking on every insertion. Before, there were a few code paths that may have skipped this step, causing *bad* paths to be inserted.

- Tracked paths in `starlark_tugger.FileManifest` should now have Windows-style directory separators (\) normalized to UNIX style (/).

Bug Fixes

- Apple code signatures using a time-stamp server now validate Apple's code signature checks. Previously, they failed validation due the time-stamped data being incorrect.

- The WiX XML IDs and GUIDs in autogenerated `.wxs` files corresponding to *install files* were sometimes internally inconsistent or duplicated, leading to malformed `.wxs` files being generated. Autogenerated `.wxs` files should now hopefully be well-formed.

- Release artifacts should now reference the `pyembed` crate from the package registry instead of a Git URL. Previously, auto-generated Rust projects might insist the `pyembed` crate was available at a Git URL. This would disagree with the auto-generated `Cargo.lock` file and result in a build failure due to building with `cargo build --frozen`.

New Features

- Default Python distributions upgraded from 3.8.9 and 3.9.4 to 3.8.10 and 3.9.5.

- PyOxidizer releases are now published as pre-built binary wheels to PyPI and can be installed via `pip install pyoxidizer`.

- Apple code signatures now include a time-stamp token issued by Apple's time-stamp server by default. Presence of the time-stamp token in code signatures is a requirement to notarize applications.

- It is now possible to add code signatures to Mach-O binaries that don't have an existing signature. Previously, it was only possible to sign binaries that had an existing signature.

- The `starlark_tugger.FileContent` Starlark type can now be constructed from filesystem paths or string content via `starlark_tugger.FileContent.__init__()`. The type also exposes mutable attributes `starlark_tugger.FileContent.executable` and `starlark_tugger.FileContent.filename` to view and change instance state.

- The new `starlark_tugger.FileManifest.add_file()` method can be used to add a `starlark_tugger.FileContent` to a `starlark_tugger.FileManifest`. The method allows controlling the destination path within the `starlark_tugger.FileManifest`. Combined with the introduction of `starlark_tugger.FileContent.__init__()`, it is now possible to add arbitrary file-based or string-based files to a `starlark_tugger.FileManifest`.

- The new `starlark_tugger.FileManifest.paths()` method can be used to retrieve the paths currently tracked by a `starlark_tugger.FileManifest`.

- The new `starlark_tugger.FileManifest.get_file()` method can be used to retrieve a `starlark_tugger.FileContent` from a path in `starlark_tugger.FileManifest`. The new

`starlark_tugger.FileManifest.remove()` method can be used to remove a tracked path from a `starlark_tugger.FileManifest`. The new methods unlock the ability to mutate the contents of `starlark_tugger.FileManifest` instances.

- Starlark now has a `starlark_tugger.AppleUniversalBinary` type that can be used to construct *universal/fat/multi-architecture* Mach-O binaries, the binary executable format used by Apple operating systems. Starlark primitives like `PythonExecutable` can today only yield a single architecture binary. However, with the new type, it is possible to take multiple source binaries and combine them into a *universal* binary, all from Starlark.

- The `starlark_tugger.WiXInstaller` Starlark type now exposes mutable attributes `starlark_tugger.WiXInstaller.install_files_root_directory_id` and `starlark_tugger.WiXInstaller.install_files_wxs_path` to control the autogenerated `.wxs` file containing fragment for *install files*. See the type's documentation for more.

Other Relevant Changes

- `starlark_tugger.WiXInstaller.build()` now automatically materializes and builds a `.wxs` file containing fragments for files registered for installation. Before, this Starlark type was not very usable without this file, as WiX wouldn't pick up files that had been registered for install.

- Rust 1.52.0 is now used as the default Rust toolchain (from version 1.51.0).

- The musl libc linked default Python distributions no longer use the `reallocarray()` symbol, which was introduced in musl libc 1.2.2. This should enable musl libc builds to work with musl 1.2.1 and possibly older versions.

0.14.1

Released April 30, 2021.

Bug Fixes

- Fixed a bug in the 0.14.0 release where newly created projects won't build due to `Cargo.lock` issues.

0.14.0

Released April 30, 2021.

Backwards Compatibility Notes

- PyOxidizer no longer uses the system's installed Rust toolchain when building projects. By default, it will download and use a specific version of the Rust toolchain. See *Managed Rust Toolchain* for instructions on disabling this behavior.

- The `pyembed` crate now always canonicalizes the path to the current executable. Previously, if `OxidizedPythonInterpreterConfig.exe` were set, it would not be canonicalized. It is possible this could break use cases where the current executable is deleted after the executable starts. In this case, the Python interpreter will fail to initialize. If this functionality is important to you, file a feature request.

- The `pyembed` crate will now remove entries from `sys.path_hooks` related to filesystem importers if filesystem importing is disabled. Previously, only `sys.meta_path` would have its filesystem importers removed.

- The `pyembed` crate now always registers the `oxidized_importer.OxidizedFinder` path hook on `sys.path_hooks` when an instance is being installed on `sys.meta_path`. This ensures that consumers of `sys.path_hooks` outside the module importing mechanism (such as `pkgutil` and `pkg_resources`) can use the path hook.

- The `pyembed` crate now registers the `oxidized_importer.OxidizedFinder` path hook as the 1st entry on `sys.path_hooks`, not the last.

- The `oxidized_importer.OxidizedFinder` path hook is now more strict about the path values it will respond to. Previously, it would accept `str`, `bytes`, `pathlib.Path`, or any other path-like type. Now, it only responds to `str` values. Furthermore, it will only respond to values that exactly match `oxidized_importer.OxidizedFinder.path_hook_base_str` or a well-formed virtual sub-directory thereof. Previously, it would attempt to canonicalize path strings, taking into account the current working directory, filesystem links, and other factors affecting path normalization. The new implementation is simpler and by being stricter should be less brittle at run-time. See *Paths Hooks Compatibility* for documentation on the path hooks behavior.

- The `pyembed` crate has prefixed all its allocator features (`jemalloc`, `mimalloc`, and `snmalloc`) with `allocator-`. This makes the names consistent with the features in auto-generated Rust projects.

Bug Fixes

- Rust projects created with `pyoxidizer init-rust-project` no longer fail to build due to a cryptic `writing packed resources` error.

- When materializing Python package distribution resources (i.e. files in `.dist-info` and `.egg-info` directories) to the filesystem, package names are now normalized to lowercase with hyphens replaced with underscores. The new behavior matches expectations of official Python resource handling APIs like `importlib.metadata`. Before, APIs like `importlib.metadata` would fail to find files materialized by PyOxidizer for package names containing a hyphen or capital latter. (#394)

New Features

- PyOxidizer now automatically downloads and uses a Rust toolchain at run time. This means there is no longer an install requirement of having Rust already available on your system (unless you install PyOxidizer from source). See *Managed Rust Toolchain* for details of the new feature, including directions on how to disable the feature and have PyOxidizer use an already installed Rust.

- `oxidized_importer.OxidizedFinder` now supports `pkg_resources` integration. Most of the `pkg_resources` APIs are implemented, enabling most `pkg_resources` functionality to work. `pkg_resources` integration is automatically enabled upon import of the `pkg_resources` module, so `pkg_resources` integration should *just work* for many applications. See *Support for pkg_resources* for the full documentation, including which features aren't implemented.

- `oxidized_importer.OxidizedFinder` now exposes the properties `oxidized_importer.OxidizedFinder.path_hook_base_str` and `oxidized_importer.OxidizedFinder.origin`.

- Starlark configuration files can now produce macOS Application Bundles. See :py:class`starlark_tugger.MacOsApplicationBundleBuilder` for the API documentation.

- `pyoxidizer` commands that evaluate Starlark files now accept the arguments `--var` and `--var-env` to define extra variables to define in the evaluated Starlark file. This enables Starlark files to be parameterized based on explicit strings provided via `--var` or through the content of environment variables via `--var-env`.

- PyOxidizer can now automatically add cryptographic code signatures when running. This feature is extensively documented at *Code Signing*. From a high-level, you instantiate and activate a `starlark_tugger.CodeSigner` in your Starlark configuration to define your code signing certificate. As files are processed as part of evaluating

your Starlark configuration file, they are examined for the ability to be signed and code signing is automatically attempted. We support signing Windows files using Microsoft's official `signtool.exe` application and Apple Mach-O and bundle files using a pure Rust reimplementation of Apple's code signing functionality. This functionality is still in its early stages of development and is lacking some power user features to exert low-level control over code signing. Please file feature requests as you encounter limitations with the functionality!

- The new Starlark functions `starlark_tugger.prompt_confirm()`, `starlark_tugger.prompt_input()`, `starlark_tugger.prompt_password()`, and `starlark_tugger.can_prompt()` can be used to allow configuration files to perform interaction with the user via the terminal. The functions all allow a default value to be provided, enabling them to be used in scenarios when stdin isn't connected to a TTY and can't be prompted.

Other Relevant Changes

- The Python API for the `oxidized_importer` Python extension module providing our custom importer logic is now centrally documented instead of spread out over multiple documentation pages. See *API Reference* for the new docs. Various type references throughout the generated documentation should now link to the new API docs.

- The Starlark dialect is now documented as native Python classes and functions using Sphinx's support for doing so. The documentation should now look more familiar to Python developers familiar with Sphinx for Python API documentation.

- PyOxidizer now stores persistent artifacts (like Rust toolchains) and downloaded Python distributions) in a per-user *cache* directory. See *Cache Directory* for more.

- The `pyoxidizer` CLI now accepts `--verbose` as a sub-command argument. Previously, it was only accepted as an argument before the sub-command name.

- Generated Rust projects (which can be temporary as part of building binaries) now contain a `Cargo.lock` file and are built with `cargo build --locked`. The template of the `Cargo.lock` is static and under version control. The presence of the `Cargo.lock` coupled with `cargo build --locked` should ensure that Rust crate versions used by Rust projects exactly match those used by the build of PyOxidizer that produced the project. This should result in more deterministic builds and higher reliability of build success.

0.13.2

Released April 15, 2021.

Bug Fixes

- Fixes a build failure on Windows.

0.13.1

Released April 15, 2021.

Bug Fixes

- The 0.13.0 release contained improper crate paths in `Cargo.toml` files due to a bug in our automated release mechanism. This release should fix those issues.

0.13.0

Released April 15, 2021.

Bug Fixes

- `WiXSimpleMsiBuilder` now properly writes XML when a license file is provided.

- `WixBundleInstallerBuilder` now handles the *already installed* exit code from the VC++ Redistributable installer as a success condition. Previously, installs would abort.

- `WixBundleInstallerBuilder` no longer errors on a missing build directory when attempting to download the Visual C++ Redistributable runtime files.

New Features

- Per-platform Windows MSI and multi-platform Windows exe installers for PyOxidizer are now available. The installers are built with PyOxidizer, using its built-in support for producing Windows installers.

Other Relevant Changes

- Default CPython distributions upgraded from 3.9.3 to 3.9.4.
- Default Python distributions upgraded setuptools from 54.2.0 to 56.0.0.

0.12.0

Released April 14, 2021.

> **Danger:** The 0.12.0 release uses CPython 3.9.3, which inadvertently shipped an ABI incompatible change, causing some extension modules to not work or crash. Please avoid this release if you use pre-built Python extension modules.

Backwards Compatibility Notes

- The minimum Rust version has been changed from 1.45 to 1.46 to facilitate use of *const fn*.

- On Apple platforms, PyOxidizer now validates that the Apple SDK being used is compatible with the Python distribution being used and will abort the build if not. Previously, PyOxidizer would blindly use whatever SDK was the default and this could lead to cryptic error messages when building (likely undefined symbol errors when linking). The current default Python distributions impose a requirement of the macosx10.15+ SDK for Python 3.8 and macosx11.0+ for Python 3.9. See issue #373 for a comprehensive discussion of this topic.

- On Apple platforms, binaries built with PyOxidizer now automatically target the OS version that the Python distribution was built to target. Previously, binaries would likely target the OS version of the building machine unless explicit action was taken. The practical effect of this change is binaries targeting x86_64 should now work on macOS 10.9 without any end-user action required.

- Documentation URLs for PyOxidizer now all consistently begin with `pyoxidizer_`. Many old documentation URLs no longer work.

Bug Fixes

- The autogenerated `pyoxidizer.bzl` correctly references the `no-copyleft` extension module filter instead of the old `no-gpl` value.

- Linux binaries using the `libedit` variant of the `readline` Python extension (occurs when using the `no-copyleft` extension module filter) no longer encounter an undefined symbol error when linking. (#376)

- The *ctypes* extension was previously compiled incorrectly, leading to run-time errors on various platforms. These issues should be fixed.

New Features

- On Apple platforms, PyOxidizer now automatically locates, validates, and uses an appropriate SDK given the settings of the Python distribution being used. PyOxidizer will reject building with an SDK older than the one used to produce the Python distribution. PyOxidizer will automatically use the newest installed SDK compatible with the target configuration. The SDK and targeting information is printed during builds. See *Build Machine Requirements* for details on how to override default behavior.

- `OxidizedFinder` now implements `path_hook()` and a path hook is automatically registered on `sys.path_hooks` during interpreter initialization when an `OxidizedFinder` is being used. Feature contributed by William Schwartz in #343.

Other Relevant Changes

- The `snmalloc` allocator now uses the C API directly and avoids going through an allocation tracking layer, improving the performance of this allocator. Improvement contributed by Ryan Clanton.

- Python distributions updated to latest versions. Changes include: macOS Python 3.8 is now built against the 10.15 SDK instead of 11.1; musl libc upgraded to 1.2.2; setuptools upgraded to 54.2.0; LibreSSL upgraded to 3.2.5; OpenSSL upgraded to 1.1.1k; SQLite upgraded to 3.35.4.

0.11.0

Released March 4, 2021.

Backwards Compatibility Notes

- The default Python distribution is now CPython 3.9 instead of 3.8. To use 3.8, pass the `python_version="3.8"` argument to *default_python_distribution()* in your configuration file. We don't anticipate dropping support for 3.8 any time soon. However, this may be necessary in order to more easily support new Python features.

- The Python 3.8 distributions no longer support Windows 7 and require Windows 8, Windows 2012, or newer. The Python 3.9 distributions already required these Windows versions.

- The minimum Rust version has been changed from 1.41 to 1.45 to facilitate the use of procedural macros.

- The `pyembed::MainPythonInterpreter::run_as_main()` method has been renamed to `py_runmain()` to reflect that it always calls `Py_RunMain()`.

- The `py-module-names` file is no longer written as part of the files comprising an embedded Python interpreter.

- `OxidizedFinder.__init__()` no longer accepts `resources_data` and `resources_file` argument to specify the resources to load. Instead, call one of the new `index_*` methods on constructed instances.

- `OxidizedFinder.__init__()` no longer automatically indexes builtin extension modules and frozen modules. Instead, you must now call one of the `index_*` methods to index these resources.

- The `pyembed::OxidizedPythonInterpreterConfig.packed_resources` field is now a `Vec<pyembed::PackedResourcesSource>` instead of `Vec<&[u8]>`. The new enum allows specifying files as alternative resources sources.

- The `no-gpl` value of `PythonPackagingPolicy.extension_module_filter` has been changed to `no-copyleft` and it operates on the SPDX license annotations instead of a list we maintained.

- `show_alloc_count` has been removed from types representing Python interpreter configuration because support for this feature was removed in Python 3.9.

- `pyembed::MainPythonInterpreter.acquire_gil()`'s signature has changed, now returning a `Python` value directly without wrapping it in a `Result`.

- `pyembed::OxidizedPythonInterpreterConfig` had its memory allocator fields refactored to support new features and to help prevent bad configs (like defining multiple custom memory allocators).

- The Starlark `PythonInterpreterConfig.raw_allocator` field has been renamed to `allocator_backend`. The `system` value has been renamed to `default`.

- The `pyembed` crate now canonicalizes the current executable's path and uses this canonicalized path when resolving values with `$ORIGIN` in them. Previously, the path passed into the program was used without resolving symlinks, etc. If that path were a symlink or hardlink, unexpected results could ensue.

- `OxidizedFinder.find_distributions()` now returns an iterator of `OxidizedDistribution` instead of a `list`. Code in the standard library of older versions of CPython expected an iterator to be returned and the new behavior is more compatible. This change enables `importlib.metadata.metadata()` to work with `OxidizedFinder`.

Bug Fixes

- Escaping of string and path values when emitting Rust code for the embedded Python interpreter configuration should now be more robust. Previously, special characters (like \) were not escaped properly. (#321)

- The `load()` Starlark function should now work. (#328)

- `pyembed::OxidizedPythonInterpreterConfig.argv` is now always used when set, even if `self.interpreter_config.argv` is also set.

- `OxidizedFinder` now normalizes trailing `.__init__` in module names to be equivalent to the parent package to partially emulate CPython's behavior. See *Support for __init__ in Module Names* for more. (#317)

- The lifetime of `pyembed::MainPythonInterpreter.acquire_gil()`'s return value has been adjusted so the Rust compiler will refuse to compile code that could crash due to attempting to use a finalized interpreter. (#345)

- `pyembed::MainPythonInterpreter.py_runmain()`'s signature has changed, now consuming ownership of the receiver. Subsequent borrows of `self` now fail to compile rather than causing runtime errors.

- The optional `rust` memory allocator is now thread-safe. Previously, it wasn't and releasing of the GIL could lead to memory corruption and crashes.

- `OxidizedResourceCollector.oxidize()` should now properly clean up the temporary directory it uses during execution. Before, premature Python interpreter termination (such as during failing tests) could cause the temporary directory to not be removed. Closes #346. Fix contributed by William Schwartz in #347.

- `OxidizedFinder.find_distributions()` now properly handles the default/empty `Context` instance (specifically instances where `.name = None`). Previously, `name = None` would filter as if `.name = "None"`. This means that all distributions should now be returned with the default/empty `Context` instance.

- `OxidizedFinder.find_distributions()` now properly filters when the passed `Context`'s `name` attribute is set to a string. Previously, the `name` and `path` attributes had their order swapped in a function call, leading to incorrect filtering.

- The Windows `standalone_static` distributions should now work again. They had been broken for a few releases and likely never worked with Python 3.9. Test coverage of this build configuration has been added to help prevent future regressions. (#360)

New Features

- Support added for `aarch64-apple-darwin` (Apple M1 machines). Only Python 3.9 is supported on this architecture. Because we do not have CI coverage for this architecture (due to GitHub Actions not yet having M1 machines), support is considered beta quality at this time.

- The `FileManifest` Starlark type now exposes an `add_path()` to add a single file to the manifest.

- The `PythonExecutable` Starlark type now exposes a `to_file_manifest()` to convert the instance to a `FileManifest`.

- The `PythonExecutable` Starlark type now exposes a `to_wix_msi_builder()` method to obtain a `WiXMSIBuilder`, which can be used to generate an MSI installer for the application.

- The `PythonExecutable` Starlark type now exposes a `to_wix_bundle_builder()` method to obtain a `WiXBundleBuilder`, which can be used to generate an `.exe` installer for the application.

- The `pyembed` crate and `OxidizedFinder` importer now support indexing multiple resources sources. You can have multiple in-memory data blobs, multiple file-based resources, or a mix of all of the above.

- The `OxidizedFinder` Python type now exposed various `index_*` methods to facilitate loading/indexing of resource data in arbitrary byte buffers or files. You can call these methods multiple times to chain multiple resources blobs together.

- The `PythonExecutable` Starlark type now exposes a `packed_resources_load_mode` attribute allowing control over where *packed resources data* is written and how it is loaded at run-time. This attribute facilitates disabling the embedding of packed resources data completely (enabling you to produce an executable that behaves very similarly to `python`) and allows writing and loading resources data to a standalone file installed next to the binary (enabling multiple binaries to share the same resources file). See *Managing Packed Resources Data* for more on this feature.

- PyOxidizer now scans for licenses of Python packages processed during building and prints a report about what it finds when writing build artifacts. This feature is best effort and relies on packages properly advertising their license metadata.

- Support for configuring Python's memory allocators has been expanded. The Starlark `PythonInterpreterConfig.allocator_debug` field has been added and allows enabling Python memory allocator debug hooks. The Starlark `PythonInterpreterConfig.allocator_mem`, `PythonInterpreterConfig.allocator_obj`, and `PythonInterpreterConfig.allocator_pymalloc_arena` fields have been added to control whether to install a custom allocator for the *mem* and *obj* domains as well as `pymalloc`'s arena allocator.

- The *mimalloc* and *snmalloc* memory allocators can now be used as Python's memory allocators. See documentation for `PythonInterpreterConfig.allocator_backend`. Code contributed by Ryan Clanton in #358.

- The *mimalloc* and *snmalloc* memory allocators will now automatically be used as Rust's global allocator when configured to be used by Python.

- The `@classmethod OxidizedDistribution.find_name()` and `OxidizedDistribution.discover()` are now implemented, filling in a feature gap in `importlib.metadata` functionality.

- There is a new `PythonExecutable.windows_runtime_dlls_mode` attribute to control how required Windows runtime DLL files should be materialized during application building. By default, if a built binary requires the Visual C++ Redistributable Runtime (e.g. `vcruntime140.dll`), PyOxidizer will attempt to locate and copy those files next to the built binary. See *Managing the Visual C++ Redistributable Requirement* for more.

- Documentation around portability of binaries produced with PyOxidizer has been reorganized and overhauled. See *Portability of Binaries Built with PyOxidizer* for the new documentation.

Other Relevant Changes

- Python distributions upgraded to CPython 3.8.8 and 3.9.2 (from 3.8.6 and 3.9.0). See https://github.com/indygreg/python-build-standalone/releases/tag/20210103 and https://github.com/indygreg/python-build-standalone/releases/tag/20210227 for a full list of changes in these distributions.

- CI has been moved from Azure Pipelines to GitHub Actions.

- Low level code in the `pyembed` crate for loading and indexing resources has been significantly refactored. This code has historically been a bit brittle, as it needs to do *unsafe* things. We think the new code is much more robust. But there's a chance that crashes could occur.

- When using the `no-copyleft` (formerly `no-gpl`) extension module filter, some system library dependencies are now allowed, enabling various extension modules to be present in this mode.

- The `pyembed` and `oxidized-importer` crates had their SPDX license expression changed from `Python-2.0 AND MPL-2.0` to `Python-2.0 OR MPL-2.0`. The author misunderstood what `AND` did and after realizing his mistake, corrected it to `OR` so the crates can one license or the other.

- When using dynamically linked Python distributions on Windows, the `python3.dll` file is automatically installed if it is present. (#336)

- `libclang_rt.osx.a` is now linked into Python binaries on macOS. This was necessary to avoid undefined symbols errors from symbols which Python 3.9.1+ relies on.

- The `oxidized_importer` Python module now exports the `OxidizedDistribution` symbol, which is the custom `importlib.metadata` *distribution* type used by `OxidizedFinder`.

- When building with Windows `standalone_static` distributions, `pyoxidizer` now sets RUSTFLAGS=-C `target-feature=+crt-static -C link-args=/FORCE:MULTIPLE` to force static CRT linkage and ignore duplicate symbol errors. Previously, the Python distribution would be using static CRT linkage and the Rust application would use dynamic/DLL CRT linkage. Furthermore, many `standalone_static` distributions have build configurations that lead to duplicate symbols and this would lead to a linker error. Suppressing the duplicate symbol error is not ideal, but it restores building with `standalone_static` until a more appropriate workaround can be devised.

0.10.3

Released November 10, 2020.

Bug Fixes

- The `run_as_main()` function on embedded Python interpreters now always calls `Py_RunMain()`. This fixes a regression in previous 0.10 releases that prevented a REPL from running when no explicit `run_*` attribute was set on the Python interpreter configuration.

0.10.2

Released November 10, 2020.

Bug Fixes

- Fixes a version mismatch between the `pyoxidizer` and `pyembed` crates that could cause builds to fail.

0.10.1

Released November 9, 2020.

Danger: The 0.10.1 release has a serious bug where the version of the `pyembed` crate needed to build binaries may not be correct, preventing the build from working. Please use a newer release.

Bug Fixes

0.10.0

Released November 8, 2020.

Danger: The 0.10.0 release has a serious Starlark bug preventing PyOxidizer from working correctly in many scenarios. Please use a newer release.

Backwards Compatibility Notes

- A lot of unused Rust functions for running Python code have been removed from the `pyembed` crate. The deleted code has not been used since the `PyConfig` data structure was adopted for running code during interpreter initialization. The deleted code was reimplementing functionality in CPython and much of it was of questionable quality.

- The built-in Python distributions have been updated to use version 6 of the standalone distribution format. PyOxidizer only recognizes version 6 distributions.

- The `pyembed::OxidizedPythonInterpreterConfig` Rust struct now contains a `tcl_library` field to control the value of the *TCL_LIBRARY* environment variable.

- The `pyembed::OxidizedPythonInterpreterConfig` Rust struct no longer has a `run_mode` field.

- The `PythoninterpreterConfig` Starlark type no longer has a `run_mode` attribute. To define what code to run at interpreter startup, populate a `run_*` attribute or leave all `None` with `.parse_argv = True` (the default for `profile = "python"`) to start a REPL.

- Minimum Rust version changed from 1.40 to 1.41 to facilitate using a new crate which requires 1.41.

- The default Cargo features of the `pyembed` crate now use the default Python interpreter detection and linking configuration as determined by the `cpython` crate. This enables the `cargo build` or `cargo test` to *just work* without having to explicitly specify features.

- The `python-distributions-extract` command now receives the path to an existing distribution archive via the `--archive-path` argument instead of an unnamed argument.

Bug Fixes

- Fixed a broken documentation example for `glob()`. (#300)

- Fixed a bug where generated Rust code for *Option<PathBuf>* interpreter configuration fields was not being generated correctly.

- Fixed serialization of string config options to Rust code that was preventing the following attributes of the `PythonInterpreterConfig` Starlark type from working: `filesystem_encoding`, `filesystem_errors`, `python_path_env`, `run_command`, `run_module`, `stdio_encoding`, `stdio_errors`, `warn_options`, and `x_options`. (#309)

New Features

- The `PythonExecutable` Starlark type now exposes a `windows_subsystem` attribute to control the value of Rust's `#![windows_subsystem = "..."]` attribute. Setting this to `windows` prevents Windows executables from opening a console window when run. (#216)

- The `PythonExecutable` Starlark type now exposes a `tcl_files_path` attribute to define a directory to install tcl/tk support files into. Setting this attribute enables the use of the `tkinter` Python module with compatible Python distributions. (#25)

- The `python-distribution-extract` CLI command now accepts a `--download-default` flag to download the default distribution for the current platform.

Other Relevant Changes

- The Starlark types with special *build* or *run* behavior are now explicitly documented.

- The list of glibc and GCC versions used by popular Linux distributions has been updated.

- The built-in Linux and macOS Python distributions are now compiled with LLVM/Clang 11 (as opposed to 10).

- The built-in Python distributions now use pip 20.2.4 and setuptools 50.3.2.

- The Starlark primitives for defining build system targets have been extracted into a new `starlark-dialect-build-targets` crate.

- The code for resolving how to reference PyOxidizer's Git repository has been rewritten. The resolution is now performed at build time in the pyoxidizer crate's `build.rs`. There now exist environment variables that can be specified at crate build time that influence how PyOxidizer constructs these references.

0.9.0

Released October 18, 2020.

Backwards Compatibility Notes

- The `pyembed::OxidizedPythonInterpreterConfig` Rust struct now contains an `argv` field that can be used to control the population of `sys.argv`.

- The `pyembed::OxidizedPythonInterpreterConfig` Rust struct now contains a `set_missing_path_configuration` field that can be used to control the automatic run-time population of missing *path configuration* fields.

- The `configure_locale` interpreter configuration setting is enabled by default. (#294)

- The `pyembed::OxidizedPythonInterpreterConfig` Rust struct now contains an `exe` field holding the path of the currently running executable.

- At run-time, the `program_name` and `home` fields of the embedded Python interpreter's path configuration are now always set to the currently running executable and its directory, respectively, unless explicit values have been provided.

- The packed resource data version has changed from 2 to 3 in order to support storing arbitrary file data. Support for reading and writing version 2 has been removed. Packed resources blobs will need to be regenerated in order to be compatible with new versions of PyOxidizer.

- The `pyembed::OxidizedPythonInterpreterConfig` Rust struct had its `packed_resources` field changed from `Option<&'a [u8]>` to `Vec<&'a [u8]>` so multiple resource inputs can be specified.

- The `PythonDistribution` Starlark type no longer has `extension_modules()`, `package_resources()` and `source_modules()` methods. Use `PythonDistribution.python_resources()` instead.

New Features

- A `print(*args)` function is now exposed to Starlark. This function is documented as a Starlark built-in but isn't provided by the Rust Starlark implementation by default. So we've implemented it ourselves. (#292)

- The new `pyoxidizer find-resources` command can be used to invoke PyOxidizer's code for scanning files for resources. This command can be used to debug and triage bugs related to PyOxidizer's custom code for finding and handling resources.

- Executables built on Windows now embed an application manifest that enables long paths support. (#197)

- The Starlark `PythonPackagingPolicy` type now exposes an `allow_files` attribute controlling whether files can be added as resources.

- The Starlark `PythonPackagingPolicy` type now exposes `file_scanner_classify_files` and `file_scanner_emit_files` attributes controlling whether file scanning attempts to classify files and whether generic file instances are emitted, respectively.

- The Starlark `PythonPackagingPolicy` type now exposes `include_classified_resources` and `include_file_resources` attributes to control whether certain classes of resources have their `add_include` attribute set by default.

- The Starlark `PythonPackagingPolicy` type now has a `set_resources_handling_mode()` method to quickly apply a mode for resource handling.

- The Starlark `PythonDistribution` type now has a `python_resources()` method for obtaining all Python resources associated with the distribution.

- Starlark `File` instances can now be added to resource collections via `PythonExecutable.add_python_resource()` and `PythonExecutable.add_python_resources()`.

Bug Fixes

- Fix some documentation references to outdated Starlark configuration syntax (#291).

- Emit only the `PythonExtensionModule` built with our patched distutils instead of emitting 2 `PythonExtensionModule` for the same named module. This should result in compiled Python extension modules being usable as built-in extensions instead of being recognized as only shared libraries.

- Fix typo preventing the Starlark method `PythonExecutable.read_virtualenv()` from being defined. (#297)

- The default value of the Starlark `PythonInterpreterConfig.configure_locale` field is `True` instead of `None` (effectively `False` since the default `.profile` value is `isolated`). This results in Python's encodings being more reasonable by default, which helps ensure non-ASCII arguments are interpreted properly. (#294)

- Properly serialize `module_search_paths` to Rust code. Before, attempting to set `PythonInterpreterConfig.module_search_paths` in Starlark would result in malformed Rust code being generated. (#298)

Other Relevant Changes

- The `pyembed` Rust crate now calls `PyConfig_SetBytesArgv` or `PyConfig_SetArgv()` to initialize argv instead of `PySys_SetObject()`. The encoding of string values should also behave more similarly to what `python` does.

- The `pyembed` tests exercising Python interpreters now run in separate processes. Before, Rust would instantiate multiple interpreters in the same process. However, CPython uses global variables and APIs (like `setlocale()`) that also make use of globals and process reuse resulted in tests not having pristine execution environments. All tests now run in isolated processes and should be much more resilient.

- When PyOxidizer invokes a subprocess and logs its output, stderr is now redirected to stdout and logged as a unified stream. Previously, stdout was logged and stderr went to the parent process stderr.

- There now exists *documentation* on how to create an executable that behaves like `python`.

- The documentation on binary portability has been overhauled to go in much greater detail.

- The list of standard library test packages is now obtained from the Python distribution metadata instead of a hardcoded list in PyOxidizer's source code.

0.8.0

Released October 12, 2020.

Backwards Compatibility Notes

- The default Python distributions have been upgraded to CPython 3.8.6 (from 3.7.7) and support for Python 3.7 has been removed.

- On Windows, the `default_python_distribution()` Starlark function now defaults to returning a `standalone_dynamic` distribution variant, meaning that it picks a distribution that can load standalone `.pyd` Python extension modules by default.

- The *standalone* Python distributions are now validated to be at least version 5 of the distribution format. If you are using the default Python distributions, this change should not affect you.

- Support for packaging the official Windows embeddable Python distributions has been removed. This support was experimental. The official Windows embeddable distributions are missing critical support files that make them difficult to integrate with PyOxidizer.

- The `pyembed` crate now defines a new `OxidizedPythonInterpreterConfig` type to configure Python interpreters. The legacy `PythonConfig` type has been removed.

- Various `run_*` functions on `pyembed::MainPythonInterpreter` have been moved to standalone functions in the `pyembed` crate. The `run_as_main()` function (which is called by the default Rust program that is generated) will always call `Py_RunMain()` and finalize the interpreter. See the extensive crate docs for move.

- Python resources data in the `pyembed` crate is no longer annotated with the `'static` lifetime. Instances of `PythonConfig` and `OxidizedPythonInterpreterConfig` must now be annotated with a lifetime for the resources data they hold such that Rust lifetimes can be enforced.

- The type of the custom Python importer has been renamed from `PyOxidizerFinder` to `OxidizedFinder`.

- The name of the module providing our custom importer has been renamed from `_pyoxidizer_importer` to `oxidized_importer`.

- Minimum Rust version changed from 1.36 to 1.40 to allow for upgrading various dependencies to modern versions.

- Windows static extension building is possibly broken due to changes to `distutils`. However, since we changed the default configuration to not use this build mode, we've deemed this potential regression acceptable for the 0.8 release. If it exists, it will hopefully be fixed in the 0.9 release.

- The `pip_install()`, `read_package_root()`, `read_virtualenv()` and `setup_py_install()` methods of the `PythonDistribution` Starlark type have been moved to the `PythonExecutable` type. Existing Starlark config files will need to change references accordingly (often by replacing `dist.` with `exe.`).

- The `PythonDistribution.extension_modules()` Starlark function no longer accepts arguments `filter` and `preferred_variants`. The function now returns every extension in the distribution. The reasons for this change were to make code simpler and the justification for removing it was rather weak. Please file an issue if this feature loss affects you.

- The `PythonInterpreterConfig` Starlark type now internally has most of its fields defined to `None` by default instead of their default values.

- The following Starlark methods have been renamed: `PythonExecutable.add_module_source()` -> `PythonExecutable.add_python_module_source()`; `PythonExecutable.add_module_bytecode()` -> `PythonExecutable.add_python_module_bytecode()`; `PythonExecutable.add_package_resource()` -> `PythonExecutable.add_python_package_resource()`; `PythonExecutable.add_package_distribution_resource()` -> `PythonExecutable.add_python_package_distribution_resource()`; `PythonExecutable.add_extension_module()` -> `PythonExecutable.add_python_extension_module()`.

- The location-specific Starlark methods for adding Python resources have been removed. The functionality can be duplicated by modifying the `add_location` and `add_location_fallback` attributes on Python resource types. The following methods were removed: `PythonExecutable.add_in_memory_module_source()`; `PythonExecutable.add_filesystem_relative_module_source()`, `PythonExecutable.add_in_memory_module_bytecode()`; `PythonExecutable.add_filesystem_relative_module_bytecode()`; `PythonExecutable.add_in_memory_package_resource()`; `PythonExecutable.add_filesystem_relative_package_resource()`; `PythonExecutable.add_in_memory_package_distribution_resource()` `PythonExecutable.add_filesystem_relative_package_distribution_resource()`; `PythonExecutable.add_in_memory_extension_module()`; `PythonExecutable.add_filesystem_relative_extension_module()`; `PythonExecutable.add_in_memory_python_resource()`; `PythonExecutable.add_filesystem_relative_python_resource()`; `PythonExecutable.add_in_memory_python_resources()`; `PythonExecutable.add_filesystem_relative_python_resources()`.

- The Starlark `PythonDistribution.to_python_executable()` method no longer accepts the arguments `extension_module_filter`, `preferred_extension_module_variants`, `include_sources`, `include_resources`, and `include_test`. All of this functionality has been replaced by the optional `packaging_policy`, which accepts a `PythonPackagingPolicy` instance. The new type represents all settings influencing executable building and control over resources added to the executable.

- The Starlark type `PythonBytecodeModule` has been removed. Previously, this type was internally a request to convert Python module source into bytecode. The introduction of `PythonPackagingPolicy` and underlying abilities to derive bytecode from a Python source module instance when adding that resource type rendered this Starlark type redundant. There may still be the need for a Starlark type to represent actual Python module bytecode (not derived from source code at build/packaging time). However, this functionality did not exist before so the loss of this type is not a loss in functionality.

- The Starlark methods `PythonExecutable.add_python_resource()` and `PythonExecutable.add_python_resources()` no longer accept the arguments `add_source_module`, `add_bytecode_module`, and `optimize_level`. Instead, set various `add_*` attributes on resource instances being passed into the methods to influence what happens when they are added.

- The Starlark methods `PythonExecutable.add_python_module_source()`, `PythonExecutable.add_python_module_bytecode()`, `PythonExecutable.add_python_package_resource()`,

> PythonExecutable.add_python_package_distribution_resource(), and PythonExecutable.
> add_python_extension_module() have been removed. The remaining PythonExecutable.
> add_python_resource() and PythonExecutable.add_python_resources() methods are capable
> of handling all resource types and should be used. Previous functionality available via argument passing on
> these methods can be accomplished by setting add_* attributes on individual Python resource objects.

- The Starlark type PythonSourceModule has been renamed to PythonModuleSource.

- Serialized Python resources no longer rely on the flavor field to influence how they are loaded at run-time. Instead, the new is_* fields expressing individual type affinity are used. The flavor attributes from the OxidizedResource Python type has been removed since it does nothing.

- The packed resources data format version has been changed from 1 to 2. The parser has dropped support for reading version 1 files. Packed resources blobs will need to be written and read by the same version of the Rust crate to be compatible.

- The autogenerated Rust file containing the Python interpreter configuration now emits a pyembed::OxidizedPythonInterpreterConfig instance instead of pyembed::PythonConfig. The new type is more powerful and what is actually used to initialize an embedded Python interpreter.

- The concept of a *resources policy* in Starlark has now largely been replaced by attributes denoting valid locations for resources.

- **oxidized_importer.OxidizedResourceCollector.__init__() now**
    accepts an allowed_locations argument instead of policy.

- The PythonInterpreterConfig() constructor has been removed. Instances of this Starlark type are now created via PythonDistribution.make_python_interpreter_config(). In addition, instances are mutated by setting attributes rather than passing perhaps dozens of arguments to a constructor function.

- The default build configuration for Windows no longer forces extension modules to be loaded from memory and materializes some extension modules as standalone files. This was done because some some extension modules weren't working when loaded from memory and the configuration caused lots of problems in the wild. The new default should be much more user friendly. To use the old settings, construct a custom PythonPackagingPolicy and set allow_in_memory_shared_library_loading = True and resources_location_fallback = None.

New Features

- Python distributions upgraded to CPython 3.8.6.

- CPython 3.9 distributions are now supported by passing python_version="3.9" to the default_python_distribution() Starlark function. CPython 3.8 is the default distribution version.

- Embedded Python interpreters are now managed via the new apis defined by PEP-587. This gives us much more control over the configuration of interpreters.

- A FileManifest Starlark instance will now have its default pyoxidizer run executable set to the last added Python executable. Previously, it would only have a run target if there was a single executable file in the FileManifest. If there were multiple executables or executable files (such as Python extension modules) a run target would not be available and pyoxidizer run would do nothing.

- Default Python distributions upgraded to version 5 of the standalone distribution format. This new format advertises much more metadata about the distribution, enabling PyOxidizer to take fewer guesses about how the distribution works and will help enable more features over time.

- The pyembed crate now exposes a new OxidizedPythonInterpreterConfig type (and associated types) allowing configuration of every field supported by Python's interpreter configuration API.

- Resources data loaded by the `pyembed` crate can now have a non-`'static` lifetime. This means that resources data can be more dynamically obtained (e.g. by reading a file). PyOxidizer does not yet support such mechanisms, however.

- `OxidizedFinder` instances can now be *constructed from Python code*. This means that a Python application can instantiate and install its own oxidized module importer.

- The resources indexed by `OxidizedFinder` instances are now representable to Python code as `OxidizedResource` instances. These types can be created, queried, and mutated by Python code. See *OxidizedResource* for the API.

- `OxidizedFinder` instances can now have custom `OxidizedResource` instances registered against them. This means Python code can collect its own Python modules and register them with the importer. See *oxidized_importer.OxidizedFinder.add_resource()* for more.

- `OxidizedFinder` instances can now serialize indexed resources out to a `bytes`. The serialized data can be loaded into a separate `OxidizedFinder` instance, perhaps in a different process. This facility enables the creation and reuse of packed resources data structures without having to use `pyoxidizer` to collect Python resources data.

- The types returned by `OxidizedFinder.find_distributions()` now implement `entry_points`, allowing *entry points* to be discovered.

- The types returned by `OxidizedFinder.find_distributions()` now implement `requires`, allowing package requirements to be discovered.

- `OxidizedFinder` is now able to load Python modules when only source code is provided. Previously, it required that bytecode be available.

- `OxidizedFinder` now implements `iter_modules()`. This enables `pkgutil.iter_modules()` to return modules serviced by `OxidizedFinder`.

- The `PythonModuleSource` Starlark type now exposes module source code via the `source` attribute.

- The `PythonExecutable` Starlark type now has a `make_python_module_source()` method to allow construction of `PythonModuleSource` instances.

- The `PythonModuleSource` Starlark type now has attributes `add_include`, `add_location`, `add_location_fallback`, `add_source`, `add_bytecode_optimization_level_zero`, `add_bytecode_optimization_level_one`, and `add_bytecode_optimization_level_two` to influence what happens when instances are added to to binaries.

- The Starlark methods for adding Python resources now accept an optional `location` argument for controlling the load location of the resource. This functionality replaces the prior functionality provided by location-specific APIs such as `PythonExecutable.add_in_memory_python_resource()`. The following methods gained this argument: `PythonExecutable.add_python_module_source()`; `PythonExecutable.add_python_module_bytecode()`; `PythonExecutable.add_python_package_resource()`; `PythonExecutable.add_python_package_distribution_resource()`; `PythonExecutable.add_python_extension_module()`; `PythonExecutable.add_python_resource()`; `PythonExecutable.add_python_resources()`.

- Starlark now has a `PythonPackagingPolicy` type to represent the collection of settings influencing how Python resources are packaged into binaries.

- The `PythonDistribution` Starlark type has gained a `make_packaging_policy()` method for obtaining the default `PythonPackagingPolicy` for that distribution.

- The `PythonPackagingPolicy.register_resource_callback()` method can be used to register a Starlark function that will be called whenever resources are created. The callback allows a single function to inspect and manipulate resources as they are created.

- Starlark types representing Python resources now expose an `is_stdlib` attribute denoting whether they came from the Python distribution.

- The new `PythonExecutable.pip_download()` method will run `pip download` to obtain Python wheels for the requested package(s). Those wheels will then be parsed for Python resources, which can be added to the executable.

- The Starlark function `default_python_distribution()` now accepts a `python_version` argument to control the *X.Y* version of Python to use.

- The `PythonPackagingPolicy` Starlark type now exposes a flag to control whether shared libraries can be loaded from memory.

- The `PythonDistribution` Starlark type now has a `make_python_interpreter_config()` method to obtain instances of `PythonInterpreterConfig` that are appropriate for that distribution.

- `PythonInterpreterConfig` Starlark types now expose attributes to query and mutate state. Nearly every setting exposed by Python's initialization API can be set.

Bug Fixes

- Fixed potential process crash due to illegal memory access when loading Python bytecode modules from the filesystem.

- Detection of Python bytecode files based on registered suffixes and cache tags is now more robust. Before, it was possible for modules to get picked up having the cache tag (e.g. `cpython-38`) in the module name.

- In the custom Python importer, `read_text()` of distributions returned from `find_distributions()` now returns `None` on unknown file instead of raising `IOError`. This matches the behavior of `importlib.metadata`.

- The `pyembed` Rust project build script now reruns when the source Starlark file changes.

- Some Python resource types were improperly installed in the wrong relative directory. The buggy behavior has been fixed.

- Python extension modules and their shared library dependencies loaded from the filesystem should no longer have the library file suffix stripped when materialized on the filesystem.

- On Windows, the `sqlite` module can now be imported. Before, the system for serializing resources thought that `sqlite` was a shared library and not a Python module.

- The build script of the pyoxidizer crate now uses the `git2` crate to try to resolve the Git commit instead of relying on the `git` command. This should result in fewer cases where the commit was being identified as `unknown`.

- `$ORIGIN` is properly expanded in `sys.path`. (This was a regression during the development of version 0.8 and is not a regression from the 0.7 release.)

Other Relevant Changes

- The registration of the custom Python importer during interpreter initialization no longer relies on running custom frozen bytecode for the `importlib._bootstrap_external` Python module. This simplifies packaging and interpreter configuration a bit.

- Packaging documentation now gives more examples on how to use available Starlark packaging methods.

- The modified `distutils` files used when building statically linked extensions have been upgraded to those based on Python 3.8.3.

- The default `pyoxidizer.bzl` now has comments for the `packaging_policy` argument to `PythonDistribution.to_python_executable()`.

- The default `pyoxidizer.bzl` now uses `add_python_resources()` instead of `add_in_memory_python_resources()`.

- The Rust Starlark crate was upgraded from version 0.2 to 0.3. There were numerous changes as part of this upgrade. While we think behavior should be mostly backwards compatible, there may be some slight changes in behavior. Please file issues if any odd behavior or regressions are observed.

- The configuration documentation was reorganized. The unified document for the complete API document (which was the largest single document) has been split into multiple documents.

- The serialized data structure for representing Python resources metadata and its data now allows resources to identify as multiple types. For example, a single resource can contain both Python module source/bytecode and a shared library.

- `pyoxidizer --version` now prints verbose information about where PyOxidizer was installed, what Git commit was used, and how the `pyembed` crate will be referenced. This should make it easier to help debug installation issues.

- The autogenerated/default Starlark configuration file now uses the `install` target as the default build/run target. This allows extra files required by generated binaries to be available and for built binaries to be usable.

0.7.0

Released April 9, 2020.

Backwards Compatibility Notes

- Packages imported from memory using PyOxidizer now set `__path__` with a value formed by joining the current executable's path with the package name. This mimics the behavior of `zipimport`.

- Resolved Python resource names have changed behavior. See the note in the bug fixes section below.

- The `PythonDistribution.to_python_executable()` Starlark method has added a `packaging_policy` named argument as its 2nd argument / 1st named argument. If you were affected by this, you should add argument names to all arguments passed to this method.

- The default Rust project for built executables now builds executables such that dynamic symbols are exported from the executable. This change is necessary in order to support executables loading Python extension modules, which are shared libraries which need access to Python symbols defined in executables.

- The `PythonResourceData` Starlark type has been renamed to `PythonPackageResource`.

- The `PythonDistribution.resources_data()` Starlark method has been renamed to `PythonDistribution.package_resources()`.

- The `PythonExecutable.to_embedded_data()` Starlark method has been renamed to `PythonExecutable.to_embedded_resources()`.

- The `PythonEmbeddedData` Starlark type has been renamed to `PythonEmbeddedResources`.

- The format of Python resource data embedded in binaries has been completely rewritten. The separate modules and resource data structures have been merged into a single data structure. Embedded resources data can now express more primitives such as package distribution metadata and different bytecode optimization levels.

- The *pyembed* crate now has a *dev* dependency on the *pyoxidizer* crate in order to run tests.

Bug Fixes

- PyOxidizer's importer now always sets `__path__` on imported packages in accordance with Python's stated behavior (#51).

- The mechanism for resolving Python resource files from the filesystem has been rewritten. Before, it was possible for files like `package/resources/foo.txt` to be normalized to a (package, resource_name) tuple of *(package, resources.foo.txt)*, which was weird and not compatible with Python's resource loading mechanism. Resources in sub-directories should no longer encounter munging of directory separators to `.`. In the above example, the resource path will now be expressed as (`package, resources/foo.txt`).

- Certain packaging actions are only performed once during building instead of twice. The user-visible impact of this change is that some duplicate log messages no longer appear.

- Added a missing *)* for *add_python_resources()* in auto-generated *pyoxidizer.bzl* files.

New Features

- Python resource scanning now recognizes `*.dist-info` and `*.egg-info` directories as package distribution metadata. Files within these directories are exposed to Starlark as *PythonPackageDistributionResource* instances. These resources can be added to the embedded resources payload and made available for loading from memory or the filesystem, just like any other resource. The custom Python importer implements `get_distributions()` and returns objects that expose package distribution files. However, functionality of the returned *distribution* objects is not yet complete. See *importlib.metadata Compatibility* for details.

- The custom Python importer now implements `get_data(path)`, allowing loading of resources from filesystem paths (#139).

- The `PythonDistribution.to_python_executable()` Starlark method now accepts a `packaging_policy` argument to control a policy and default behavior for resources on the produced executable. Using this argument, it is possible to control how resources should be materialized. For example, you can specify that resources should be loaded from memory if supported and from the filesystem if not. The argument can also be used to materialize the Python standard library on the filesystem, like how Python distributions typically work.

- Python resources can now be installed next to built binaries using the new Starlark functions `PythonExecutable.add_filesystem_relative_module_source()`, `PythonExecutable.add_filesystem_relative_module_bytecode()`, `PythonExecutable.add_filesystem_relative_package_resource()`, `PythonExecutable.add_filesystem_relative_extension_module()`, `PythonExecutable.add_filesystem_relative_python_resource()`, `PythonExecutable.add_filesystem_relative_package_distribution_resource()`, and `PythonExecutable.add_filesystem_relative_python_resources()`. Unlike adding Python resources to `FileManifest` instances, Python resources added this way have their metadata serialized into the built executable. This allows the special Python module importer present in built binaries to service the `import` request without going through Python's default filesystem-based importer. Because metadata for the file-based Python resources is *frozen* into the application, Python has to do far less work at run-time to load resources, making operations faster. Resources loaded from the filesystem in this manner have attributes like `__file__`, `__cached__`, and `__path__` set, emulating behavior of the default Python importer. The custom import now also implements the `importlib.abc.ExecutionLoader` interface.

- Windows binaries can now import extension modules defined as shared libraries (e.g. `.pyd` files) from memory. PyOxidizer will detect `.pyd` files during packaging and embed them into the binary as resources. When the module is imported, the extension module/shared library is loaded from memory and initialized. This feature enables PyOxidizer to package pre-built extension modules (e.g. from Windows binary wheels published on PyPI) while still maintaining the property of a (mostly) self-contained executable.

- Multiple bytecode optimization levels can now be embedded in binaries. Previously, it was only possible to embed bytecode for a given module at a single optimization level.

- The `default_python_distribution()` Starlark function now accepts values `standalone_static` and `standalone_dynamic` to specify a *standalone* distribution that is either statically or dynamically linked.

- Support for parsing version 4 of the `PYTHON.json` distribution descriptor present in standalone Python distribution archives.

- Default Python distributions upgraded to CPython 3.7.7.

Other Relevant Changes

- The directory for downloaded Python distributions in the build directory now uses a truncated SHA-256 hash instead of the full hash to help avoid path length limit issues (#224).

- The documentation for the `pyembed` crate has been moved out of the Sphinx documentation and into the Rust crate itself. Rendered docs can be seen by following the *Documentation* link at https://crates.io/crates/pyembed or by running `cargo doc` from a source checkout.

0.6.0

Released February 12, 2020.

Backwards Compatibility Notes

- The `default_python_distribution()` Starlark function now accepts a `flavor` argument denoting the distribution flavor.

- The `pyembed` crate no longer includes the auto-generated default configuration file. Instead, it is consumed by the application that instantiates a Python interpreter.

- Rust projects for the main executable now utilize and require a Cargo build script so metadata can be passed from `pyembed` to the project that is consuming it.

- The `pyembed` crate is no longer added to created Rust projects. Instead, the generated `Cargo.toml` will reference a version of the `pyembed` crate identical to the `PyOxidizer` version currently running. Or if `pyoxidizer` is running from a Git checkout of the canonical `PyOxidizer` Git repository, a local filesystem path will be used.

- The fields of `EmbeddedPythonConfig` and `pyembed::PythonConfig` have been renamed and reordered to align with Python 3.8's config API naming. This was done for the Starlark type in version 0.5. We have made similar changes to 0.6 so naming is consistent across the various types.

Bug Fixes

- Module names without a `.` are now properly recognized when scanning the filesystem for Python resources and a package allow list is used (#223). Previously, if filtering scanned resources through an explicit list of allowed packages, the top-level module/package without a dot in its full name would not be passed through the filter.

New Features

- The `PythonDistribution()` Starlark function now accepts a `flavor` argument to denote the distribution type. This allows construction of alternate distribution types.

- The `default_python_distribution()` Starlark function now accepts a `flavor` argument which can be set to `windows_embeddable` to return a distribution based on the zip file distributions published by the official CPython project.

- The `pyembed` crate and generated Rust projects now have various `build-mode-*` feature flags to control how build artifacts are built. See *PyOxidizer Rust Projects* for more.

- The `pyembed` crate can now be built standalone, without being bound to a specific `PyOxidizer` configuration.

- The `register_target()` Starlark function now accepts an optional `default_build_script` argument to define the default target when evaluating in *Rust build script* mode.

- The `pyembed` crate now builds against published `cpython` and `python3-sys` crates instead of a a specific Git commit.

- Embedded Python interpreters can now be configured to run a file specified by a filename. See the `run_file` argument of *PythonInterpreterConfig*.

Other Relevant Changes

- Rust internals have been overhauled to use traits to represent various types, namely Python distributions. The goal is to allow different Python distribution flavors to implement different logic for building binaries.

- The `pyembed` crate's `build.rs` has been tweaked so it can support calling out to `pyoxidizer`. It also no longer has a build dependency on `pyoxidizer`.

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.)

0.5.0

Released January 26, 2020.

General Notes

This release of PyOxidizer is significant rewrite of the previous version. The impetus for the rewrite is to transition from TOML to Starlark configuration files. The new configuration file format should allow vastly greater flexibility for building applications and will unlock a world of new possibilities.

The transition to Starlark configuration files represented a shift from static configuration to something more dynamic. This required refactoring a ton of code.

As part of refactoring code, we took the opportunity to shore up lots of the code base. PyOxidizer was the project author's first real Rust project and a lot of bad practices (such as use of *.unwrap()*/panics) were prevalent. The code mostly now has proper error handling. Another new addition to the code is unit tests. While coverage still isn't great, we now have tests performing meaningful packaging activities. So regressions should hopefully be less common going forward.

Because of the scale of the rewritten code in this release, it is expected that there are tons of bugs of regressions. This will likely be a transitional release with a more robust release to follow.

Backwards Compatibility Notes

- Support for building distributions/installers has been temporarily dropped.

- Support for installing license files has been temporarily dropped.

- Python interpreter configuration setting names have been changed to reflect names from Python 3.8's interpreter initialization API.

- `.egg-info` directories are now ignored when scanning for Python resources on the filesystem (matching the behavior for `.dist-info` directories).

- The `pyoxidizer init` sub-command has been renamed to `init-rust-project`.

- The `pyoxidizer app-path` sub-command has been removed.

- Support for building distributions has been removed.

- The minimum Rust version to build has been increased from 1.31 to 1.36. This is mainly due to requirements from the `starlark` crate. We could potentially reduce the minimum version requirements again with minimal changes to 3rd party crates.

- PyOxidizer configuration files are now Starlark instead of TOML files. The default file name is `pyoxidizer.bzl` instead of `pyoxidizer.toml`. All existing configuration files will need to be ported to the new format.

Bug Fixes

- The `repl` run mode now properly exits with a non-zero exit code if an error occurs.

- Compiled C extensions now properly honor the `ext_package` argument passed to `setup()`, resulting in extensions which properly have the package name in their extension name (#26).

New Features

- A *glob()*` function has been added to config files to allow referencing existing files on the filesystem.

- The in-memory `MetaPathFinder` now implements `find_module()`.

- A `pyoxidizer init-config-file` command has been implemented to create just a `pyoxidizer.bzl` configuration file.

- A `pyoxidizer python-distribution-info` command has been implemented to print information about a Python distribution archive.

- The `EmbeddedPythonConfig()` config function now accepts a `legacy_windows_stdio` argument to control the value of `Py_LegacyWindowsStdioFlag` (#190).

- The `EmbeddedPythonConfig()` config function now accepts a `legacy_windows_fs_encoding` argument to control the value of `Py_LegacyWindowsFSEncodingFlag`.

- The `EmbeddedPythonConfig()` config function now accepts an `isolated` argument to control the value of `Py_IsolatedFlag`.

- The `EmbeddedPythonConfig()` config function now accepts a `use_hash_seed` argument to control the value of `Py_HashRandomizationFlag`.

- The `EmbeddedPythonConfig()` config function now accepts an `inspect` argument to control the value of `Py_InspectFlag`.

- The `EmbeddedPythonConfig()` config function now accepts an `interactive` argument to control the value of `Py_InteractiveFlag`.

- The `EmbeddedPythonConfig()` config function now accepts a `quiet` argument to control the value of `Py_QuietFlag`.

- The `EmbeddedPythonConfig()` config function now accepts a `verbose` argument to control the value of `Py_VerboseFlag`.

- The `EmbeddedPythonConfig()` config function now accepts a `parser_debug` argument to control the value of `Py_DebugFlag`.

- The `EmbeddedPythonConfig()` config function now accepts a `bytes_warning` argument to control the value of `Py_BytesWarningFlag`.

- The `Stdlib()` packaging rule now now accepts an optional `excludes` list of modules to ignore. This is useful for removing unnecessary Python packages such as `distutils`, `pip`, and `ensurepip`.

- The `PipRequirementsFile()` and `PipInstallSimple()` packaging rules now accept an optional `extra_env` dict of extra environment variables to set when invoking `pip install`.

- The `PipRequirementsFile()` packaging rule now accepts an optional `extra_args` list of extra command line arguments to pass to `pip install`.

Other Relevant Changes

- PyOxidizer no longer requires a forked version of the `rust-cpython` project (the `python3-sys` and `cpython` crates. All changes required by PyOxidizer are now present in the official project.

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.

0.3.0

Released on August 16, 2019.

Backwards Compatibility Notes

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

Bug Fixes

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

New Features

- The default Python distributions have been updated. Archives are generally about half the size from before. Tcl/tk is included in the Linux and macOS distributions (but PyOxidizer doesn't yet package the Tcl files).

- Extra extension modules can now be registered with `PythonConfig` instances. This can be useful for having the application embedding Python provide its own extension modules without having to go through Python build mechanisms to integrate those extension modules into the Python executable parts.

- Built applications now have the ability to detect and use `terminfo` databases on the execution machine. This allows applications to interact with terminals properly. (e.g. the backspace key will now work in interactive `pdb` sessions). By default, applications on non-Windows platforms will look for `terminfo` databases at well-known locations and attempt to load them.

- Default Python distributions now use CPython 3.7.4 instead of 3.7.3.

- A warning is now emitted when a Python source file contains `__file__`. This should help trace down modules using `__file__`.

- Added 32-bit Windows distribution.

- New `pyoxidizer distribution` command for producing distributable artifacts of applications. Currently supports building tar archives and `.msi` and `.exe` installers using the WiX Toolset.

- Libraries required by C extensions are now passed into the linker as library dependencies. This should allow C extensions linked against libraries to be embedded into produced executables.

- `pyoxidizer --verbose` will now pass verbose to invoked `pip` and `setup.py` scripts. This can help debug what Python packaging tools are doing.

All Other Relevant Changes

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

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.

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)

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.

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)

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.

Technical Notes

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)``
      ``_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")``
```

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`.

`sys.modules` After Interpreter Init

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

Modules Imported by `site.py`

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

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.

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.

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.

`__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.

`importlib.metadata` Documentation Deficiencies

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

`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.

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.

Ability to Install Meta Path Importers Before `Py_Initialize()`

`Py_Initialize()` will import some standard library modules during its execution. It does so using the default meta path importers available to the distribution. This means that standard library modules must come from the filesystem (`PathImporter`), frozen modules, built-in extension modules, or zip files (via `PathImporter`).

This restriction prevents importing the entirety of the standard library from the binary containing Python, in effect preventing the use of self-contained executables. PyOxidizer works around this by patching the `importlib._bootstrap` and `importlib._bootstrap_external` source code, compiling that to bytecode, and making said bytecode available as a frozen module. The patched code (which runs as part of `Py_Initialize()`) installs a `sys.meta_path` importer which imports modules from memory. This solution is extremely hacky, but is necessary to achieve single file executables with all imports serviced from memory.

In order for this to work, PyOxidizer needs a copy of these `importlib` modules so it can patch them and compile them to bytecode. This is problematic in some cases because e.g. the Windows embeddable Python distributions ship only the bytecode of these modules in a `pythonXY.zip` file. So PyOxidizer needs to find the source code from another location when consuming these distributions.

But patching the `importlib` bootstrap modules is hacky itself. It would be better if PyOxidizer didn't need to do this at all. This could be achieved by splitting up the interpreter initialization APIs to give embedding applications the opportunity to muck with `sys.meta_path` before any `import` is performed. It could also be achieved by introducing an initialization config option to somehow inject code at the right point during startup to register the `sys.meta_path` importer. This could be done by importing a named module (presumably serviced by the frozen or built-in importer)

and having that module run code to modify `sys.meta_path` as a side-effect of module evaluation at import time. A variation would be to define a callable in said module to call after the module is importer. Whatever the solution, there needs to be a way to somehow inject a `sys.meta_path` importer before any `import` not serviced by the frozen or built-in importers is performed.

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.)

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.

Extension Module / Shared Library Filename Ambiguity

On some platforms, Python extension modules and shared libraries have the same filename extension. e.g. on Linux, both are named `foo.so`.

PyOxidizer's packaging functionality needs to classify files as specific resource types (source modules, bytecode modules, resource files, extension modules, shared libraries, etc). Because certain file patterns (like `.so`) are ambiguous, PyOxidizer cannot perform this classification trivially.

It would be much preferred if there were unique file extensions that distinguished Python extension modules from regular shared libraries.

On Windows, this is already the case with the `.pyd` extension. However, POSIX architectures aren't so fortunate.

Ambiguous File Classification

This is somewhat related to the previous section but is more generic.

Python's default path-based importer dynamically looks for presence of various files on the filesystem and loads the first type variant (extension module, bytecode, source, etc) discovered.

PyOxidizer's importer indexes resources during packaging and its import-time resource resolution is static: the type of resource is baked into the definition of the resource.

These approaches are somewhat at odds with each other. The path-based importer is dynamic in nature: it defers answering questions until a specific resource is requested. PyOxidizer's importer is static / pre-compiled: it must classify a resource based on its filename/path so it can bake that knowledge into an immutable data structure. It does not have knowledge of what names will be requested at run-time.

Bridging this divide has revealed various ambiguities and corner cases in the filenames of Python resources.

The Python extension module or shared library ambiguity is described above.

There is also an ambiguity with extra files that aren't part of a known Python package. If you attempt to classify every file in a `sys.path` directory, it is tempting to classify a file as a Python module (`.py`, `.pyc`, or extension module), package resource (`importlib.resources`), or package metadata (e.g. `.dist-info` files accessed via `importlib.metadata`). However, there exists the possibility that a file is not obviously classified as one of these.

For example, a file `foo/libfoo.so` without the presence of a `foo/__init__.py` file is ambiguous. We could say this is an extension module (`foo.libfoo`) due to the extension module shared library ambiguity. We could also consider this a package resource `foo:libfoo.so` or `"":foo/libfoo.so`. Although the latter case of using an empty string for the package name doesn't make much sense. And we arguably shouldn't consider it a resource of `foo` because no obvious `foo` Python package exists!

This is relevant in the real world because various Python packages rely on installing arbitrary files in `sys.path` directories. For example, `numpy` installs files like `numpy.libs/libz-eb09ad1d.so.1.2.3`, where the `numpy.libs` directory only contains file extensions `*.so[.*]`. Note that this example is particularly confusing because the directory names in `sys.path` directories are typically split on `.` and correspond to Python [sub-]packages.

Because there is no unambiguous way to classify all files in a `sys.path` directory and because Python packaging tools allow the presence of files not contained within a known Python package (identified by the presence of an `__init__` file/module), this externalizes the requirement to introduce an *other* classification of files. And because a specific file can't easily be classified as a specific type, this effectively prevents the use of *resource* loading techniques not involving explicit filesystem I/O without significant smarts. I.e. because PyOxidizer cannot easily unambiguously identify file X as a specific type, it is forced to materialize that file at a similar location on the run-time system. However, if runtimes like PyOxidizer were able to identify the type of a file by its file extension and/or presence of other files, it would know exactly how to load/treat the file at run-time without having to resort to heuristics.

This ambiguity effectively means that PyOxidizer needs to:

- Determine if a file is a shared library or not (because shared libraries are treated specially and we can't unambiguously identify a shared library from its file extension).

- Examine symbols within shared libraries to see if a Python extension module is present (via presence of `PyInit_*` symbols).

- Preserve *extra* files not present in a Python package. (In the case of numpy, there are no *obvious* links to the shared libraries in the `numpy.libs` directory: this relative path is encoded within the extension module shared library via e.g. `DT_NEEDED`.)

The most robust mitigation to this ambiguity is for all files associated with an installable Python package/distribution to be annotated with their type and for Python package installers to refuse to process files that aren't identified. This could be achieved by having a `.dist-info/` file annotating the *role* of each file.

Push Harder for Wheels

Wheels are superior for Python packaging distribution because they are more *static* and follow a finite set of rules for how they should be installed. In theory, one could write code to install a wheel in any programming language. Non-wheel distributions, however, are a different matter entirely. A `.tar.gz` source distribution often relies on running a `setup.py` file, which requires a Python interpreter.

In the ideal world, PyOxidizer doesn't care about how a package is built: just the files that comprise the installed package. So wheels are a more desirable distribution format. In fact, PyOxidizer has Rust code for extracting wheels and repackaging their contents: no Python necessary. This means PyOxidizer can do things like download wheels targeting non-native architectures and it *just works*.

As good as wheels are, they are universal in Python land. There are tons of packages that don't have wheel distributions and continue to offer the older `.tar.gz` distribution format.

We would like to see a concerted effort to push harder for the presence of wheels. For example, PyPI could encourage/nag package maintainers to upload wheels.

No Way to Hook `open()`

`oxidized_importer` wants to load Python modules and resource data from memory, without using files.

There is a convention of using virtual paths to express paths within some other entity. e.g. the zip importer uses `/path/to/archive.zip/foo.py` refers to the path `foo.py` within the `/path/to/archive.zip` zip file. It is also common to use the current executable's path to refer to entities within the current executable. e.g. `/path/to/myapp/foo.py` would refer to a `foo.py` somehow embedded in the `/path/to/myapp` executable.

These virtual paths are a great idea. You can even implement `pathlib.Path` around these paths and have a custom `Path.open()` that does custom I/O. However, it is really easy for these paths to *leak* and to get fed in to `io.open()` or similar APIs for operating on filesystem paths. For example, someone does `open(foo.__path__, "rb")` instead of `foo.__path__.open("rb")`. If this happens, you'll likely get an I/O error since virtual paths aren't real filesystem paths.

It would be really nice if Python had some abstraction around filesystem I/O that allowed custom paths to be registered. This is what schemes in URIs and URLs are for. e.g. `file:///path/to/file`. However, schemes aren't paths per se. So if we want to preserve compatibility with a path based API and allow `io.open()` to work with virtual paths, we need a mechanism to register a hook to intercept `io.open()` (and possibly other I/O operations like `stat()`) so we can plumb in a custom I/O implementation.

PEP 578 almost does this with `PyFile_SetOpenCodeHook()` and the `io.open_code()` mechanism. But `io.open_code()` is only for a limited use case and isn't generally usable.

1.5 PyOxy

PyOxy is an application providing an alternative Python runner. Think of it as an alternative implementation and re-imagination of the ubiquitous `python` command.

PyOxy enables access to some of the technology built for `pyoxidizer` (notably *oxidized_importer* and *pyembed*) without having to use `pyoxidizer`.

PyOxy is distributed as a standalone application.

1.5.1 The `PyOxy` Python Runner

PyOxy is:

- A single file Python distribution (no temporary files or virtual filesystems like SquashFS: everything is imported directly from data inside the executable).

- An alternative implementation and re-imagination of the ubiquitous `python` command, enabling you to have nearly full control over how to run a Python interpreter.

- Written in Rust, using reusable components initially built for PyOxidizer.

- Part of the PyOxidizer umbrella project.

The official home of PyOxy is https://github.com/indygreg/PyOxidizer/. Read the (stable | latest) docs online.

Releases can be found at https://github.com/indygreg/PyOxidizer/releases.

Overview

The pyoxy Executable

PyOxy is distributed as a `pyoxy` compiled executable. This executable links against a Python implementation/distribution (i.e. `libpython`).

The Python implementation/distribution and any resources defined in its standard library *may* be compiled statically into the `pyoxy` executable. This enables `pyoxy` to function as a single file Python distribution. (This is how official builds of `pyoxy` are distributed.)

`pyoxy`'s `int main()` is implemented in Rust. It simply parses the process arguments and executes a sub-command.

Full Python Interpreter Control

Commands like `pyoxy run-yaml` (see *Running YAML Based Applications*) give you very low-level control over the behavior of the Python interpreter: much lower than what is possible with `python` command arguments or environment variables.

This control can be useful for iterating/testing on different Python embedding configurations (such as how you would need to configure PyOxidizer). The control can also be useful for use in automated testing where you may want to simulate an embedded Python configuration but don't want to produce your own executable for each configuration variation. With commands like `pyoxy run-yaml`, you can simply define a YAML file defining the interpreter configuration and use a single executable for driving the Python interpreter N ways.

Additional Python Features

pyoxy supplements the built-in features of `python` with its own.

With `pyoxy`, you can:

- Dynamically choose from the system, jemalloc, mimalloc, or snmalloc memory allocators.

- Easily leverage the `oxidized_importer` extension module for importing Python modules and loading file-based resources faster than the official importers in the Python standard library.

- Automatically discover the location of the `terminfo` database at runtime, helping to ensure terminal functionality works as intended.

- Automatically write a file containing a list of imported modules when the Python interpreter finalizes.

- And more.

pyoxy aims to expose all the value-added features implemented in the `pyembed` Rust crate via the CLI so Python developers can harness these features without having to use something more heavyweight, like PyOxidizer.

Masquerading as `python`

The `pyoxy run-python` command can be used to make the executable behave like `python` would. e.g. `pyoxy run-python -- -c "print('hello, world')"`.

In addition, if the `pyoxy` executable's file name begins with `python` (e.g. `python`, `python3`, `python3.9`, `python.exe`), its custom argument parsing is short-circuited and the executable will behave as if it is actually `python`. This theoretically enables `pyoxy` to be used as a drop-in replacement for `python`.

```
$ mv pyoxy python
$ ./python
Python 3.9.5 (default, May 11 2021, 08:20:37)
[GCC 10.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Installing

It is **highly recommended** to obtain and use one of the official pre-built binaries for PyOxy. These can be obtained from GitHub releases. Go to https://github.com/indygreg/PyOxidizer/releases and scroll until you find the latest release for PyOxy.

System Requirements

The requirements in this section only apply to the official pre-built binaries. Binaries built by others may not have the same requirements.

Linux

The x86_64-unknown-linux-gnu binaries should work on any Linux having glibc 2.18+ and GCC 4.2+ and conform to the Linux Standard Base Core Specification. This should be ~every Linux distribution released since 2014-2015.

Note: Modern versions of Fedora / CentOS / RHEL have a bug and don't conform to the LSB Core Specification unless you install the `libxcrypt-compat` package. If you see an error about missing `libcrypt.so.1`, your distribution is buggy.

---

macOS

The x86_64-apple-darwin binaries target macOS 10.9+.

The aarch64-apple-darwin binaries target macOS 11.0+.

The binaries should work on a fresh install of macOS.

Running YAML Based Applications

The `pyoxy run-yaml` command enables you to run a Python interpreter given a configuration defined in a YAML file.

Usage

Run `pyoxy help run-yaml` to see full documentation.

The high-level operation is:

```
pyoxy run-yaml [FILE] [-- <args>...]
```

You give the command the path to a YAML file to parse and optional additional arguments following a `--`. e.g.:

```
pyoxy run-yaml myapp.yaml
pyoxy run-yaml myapp.yaml -- --arg true
```

File Parsing

We use a customized mechanism for parsing the specified content for a YAML config. The rules are as follows:

- The file MUST be UTF-8. (YAML allows encodings other than UTF-8. We do not yet support alternative encodings, such as UTF-16.)

- The content of the file up to a line beginning with `---` is ignored.

- Parsing stops when a line beginning with `...` is encountered.

- All content between the initial line beginning with `---` and either a) the first line beginning with `...` or b) the end of the file is parsed as YAML.

YAML Configuration

The YAML document attempts to deserialize to a `pyembed::OxidizedPythonInterpreterConfig` Rust struct. This type and its fields are extensively documented at *OxidizedPythonInterpreterConfig Struct*.

Some of the most important fields in the configuration data structure define what to run when the interpreter starts. e.g.

```
---
interpreter_config:
  run_command: 'print("hello, world")'
...
```

```
---
interpreter_config:
  run_module: 'mypackage.__main__'
...
```

Portable Invocation Using a Shell Shebang

On UNIX-like platforms, files containing an embedded YAML config can be made to execute with `pyoxy run-yaml` by using a specially crafted shebang (leading `#!` line) and making the file executable.

For example, say you distribute the `pyoxy` binary in the same directory as your executable `myapp` file. Here's what `myapp` would look like:

```
#!/bin/sh
"exec" "`dirname $0`/pyoxy" run-yaml "$0" -- "$@"
---
# YAML configuration.
...
```

This file defines a shell script which simply calls `exec` to invoke `pyoxy run-yaml`, giving it the path to the current file and additional arguments passed to the original invocation. Because our custom YAML parsing ignores content up to the first line beginning with `---`, the shebang and shell script content is ignored and the file evaluates as if those initial lines did not exist.

Python Interpreter Configuration Data Structures

This document describes the data structures for configuring the behavior of a Python interpreter. The data structures are consumed by the `pyembed` Rust crate. All type names should correspond to public symbols in the `pyembed` crate.

This documentation is auto-generated from the inline documentation in Rust source files. Some formatting has been lost as part of the conversion. See https://docs.rs/pyembed/ for the native Rust API documentation

Structs:

- *OxidizedPythonInterpreterConfig*
- *PythonInterpreterConfig*

Enums:

- *MemoryAllocatorBackend*
- *PythonInterpreterProfile*
- *Allocator*

- *BytecodeOptimizationLevel*

- *BytesWarning*

- *CheckHashPycsMode*

- *CoerceCLocale*

- *MultiprocessingStartMethod*

- *TerminfoResolution*

`OxidizedPythonInterpreterConfig` Struct

Configuration for a Python interpreter.

This type is used to create a `crate::MainPythonInterpreter`, which manages a Python interpreter running in the current process.

This type wraps a `PythonInterpreterConfig`, which is an abstraction over the low-level C structs (`PyPreConfig` and `PyConfig`) used as part of Python's C initialization API. In addition to this data structure, the fields on this type facilitate control of additional features provided by this crate.

The `PythonInterpreterConfig` has a single non-optional field: `PythonInterpreterConfig::profile`. This defines the defaults for various fields of the `PyPreConfig` and `PyConfig` C structs. See https://docs.python.org/3/c-api/init_config.html#isolated-configuration for more.

When this type is converted to `PyPreConfig` and `PyConfig`, instances of these C structs are created from the specified profile. e.g. by calling `PyPreConfig_InitPythonConfig()`, `PyPreConfig_InitIsolatedConfig`, `PyConfig_InitPythonConfig`, and `PyConfig_InitIsolatedConfig`. Then for each field in `PyPreConfig` and `PyConfig`, if a corresponding field on `PythonInterpreterConfig` is `Some`, then the `PyPreConfig` or `PyConfig` field will be updated accordingly.

During interpreter initialization, `Self::resolve()` is called to resolve/finalize any missing values and convert the instance into a `ResolvedOxidizedPythonInterpreterConfig`. It is this type that is used to produce a `PyPreConfig` and `PyConfig`, which are used to initialize the Python interpreter.

Some fields on this type are redundant or conflict with those on `PythonInterpreterConfig`. Read the documentation of each field to understand how they interact. Since `PythonInterpreterConfig` is defined in a different crate, its docs are not aware of the existence of this crate/type.

This struct implements `Deserialize` and `Serialize` and therefore can be serialized to any format supported by the `serde` crate. This feature is used by `pyoxy` to allow YAML-based configuration of Python interpreters.

`exe` Field

The path of the currently executing executable.

This value will always be `Some` on `ResolvedOxidizedPythonInterpreterConfig` instances.

Default value: `None`.

`Self::resolve()` behavior: sets to `std::env::current_exe()` if not set. Will canonicalize the final path, which may entail filesystem I/O.

Type: `Option<PathBuf>`

`origin` Field

The filesystem path from which relative paths will be interpreted.

This value will always be `Some` on `ResolvedOxidizedPythonInterpreterConfig` instances.

Default value: `None`.

`Self::resolve()` behavior: sets to `Self::exe.parent()` if not set.

Type: `Option<PathBuf>`

`interpreter_config` Field

Low-level configuration of Python interpreter.

Default value: `PythonInterpreterConfig::default()` with `PythonInterpreterConfig::profile` always set to `PythonInterpreterProfile::Python`.

`Self::resolve()` behavior: most fields are copied verbatim. `PythonInterpreterConfig::module_search_paths` entries have the special token `$ORIGIN` expanded to the resolved value of `Self::origin`.

Type: `PythonInterpreterConfig`

`allocator_backend` Field

Memory allocator backend to use.

Default value: `MemoryAllocatorBackend::Default`.

Interpreter initialization behavior: after `Py_PreInitialize()` is called, `crate::pyalloc::PythonMemoryAllocator::from_back` is called. If this resolves to a `crate::pyalloc::PythonMemoryAllocator`, that allocator will be installed as per `Self::allocator_raw`, `Self::allocator_mem`, `Self::allocator_obj`, and `Self::allocator_pymalloc_arena`. If a custom allocator backend is defined but all the `allocator_*` flags are `false`, the allocator won't be used.

Type: `MemoryAllocatorBackend`

`allocator_raw` Field

Whether to install the custom allocator for the `raw` memory domain.

See https://docs.python.org/3/c-api/memory.html for documentation on how Python memory allocator domains work.

Default value: `true`

Interpreter initialization behavior: controls whether `Self::allocator_backend` is used for the `raw` memory domain.

Has no effect if `Self::allocator_backend` is `MemoryAllocatorBackend::Default`.

Type: `bool`

allocator_mem Field

Whether to install the custom allocator for the `mem` memory domain.

See https://docs.python.org/3/c-api/memory.html for documentation on how Python memory allocator domains work.

Default value: `false`

Interpreter initialization behavior: controls whether `Self::allocator_backend` is used for the `mem` memory domain.

Has no effect if `Self::allocator_backend` is `MemoryAllocatorBackend::Default`.

Type: `bool`

allocator_obj Field

Whether to install the custom allocator for the `obj` memory domain.

See https://docs.python.org/3/c-api/memory.html for documentation on how Python memory allocator domains work.

Default value: `false`

Interpreter initialization behavior: controls whether `Self::allocator_backend` is used for the `obj` memory domain.

Has no effect if `Self::allocator_backend` is `MemoryAllocatorBackend::Default`.

Type: `bool`

allocator_pymalloc_arena Field

Whether to install the custom allocator for the `pymalloc` arena allocator.

See https://docs.python.org/3/c-api/memory.html for documentation on how Python memory allocation works.

Default value: `false`

Interpreter initialization behavior: controls whether `Self::allocator_backend` is used for the `pymalloc` arena allocator.

This setting requires the `pymalloc` allocator to be used for the `mem` or `obj` domains (`allocator_mem = false` and `allocator_obj = false` - this is the default behavior) and for `Self::allocator_backend` to not be `MemoryAllocatorBackend::Default`.

Type: `bool`

allocator_debug Field

Whether to set up Python allocator debug hooks to detect memory bugs.

Default value: `false`

Interpreter initialization behavior: triggers the calling of `PyMem_SetupDebugHooks()` after custom allocators are installed.

This setting can be used with or without custom memory allocators (see other `allocator_*` fields).

Type: `bool`

`set_missing_path_configuration` Field

Whether to automatically set missing "path configuration" fields.

If `true`, various path configuration (https://docs.python.org/3/c-api/init_config.html#path-configuration) fields will be set automatically if their corresponding `.interpreter_config` fields are `None`. For example, `program_name` will be set to the current executable and `home` will be set to the executable's directory.

If this is `false`, the default path configuration built into libpython is used.

Setting this to `false` likely enables isolated interpreters to be used with "external" Python installs. If this is `true`, the default isolated configuration expects files like the Python standard library to be installed relative to the current executable. You will need to either ensure these files are present, define `packed_resources`, and/or set `.interpreter_config.module_search_paths` to ensure the interpreter can find the Python standard library, otherwise the interpreter will fail to start.

Without this set or corresponding `.interpreter_config` fields set, you may also get run-time errors like `Could not find platform independent libraries <prefix>` or `Consider setting $PYTHONHOME to <prefix>[:<exec_prefix>]`. If you see these errors, it means the automatic path config resolutions built into libpython didn't work because the run-time layout didn't match the build-time configuration.

Default value: `true`

Type: `bool`

`oxidized_importer` Field

Whether to install `oxidized_importer` during interpreter initialization.

If `true`, `oxidized_importer` will be imported during interpreter initialization and an instance of `oxidized_importer.OxidizedFinder` will be installed on `sys.meta_path` as the first element.

If `Self::packed_resources` are defined, they will be loaded into the `OxidizedFinder`.

If `Self::filesystem_importer` is `true`, its *path hook* will be registered on `sys.path_hooks` so `PathFinder` (the standard filesystem based importer) and `pkgutil` can use it.

Default value: `false`

Interpreter initialization behavior: See above.

Type: `bool`

`filesystem_importer` Field

Whether to install the path-based finder.

Controls whether to install the Python standard library `PathFinder` meta path finder (this is the meta path finder that loads Python modules and resources from the filesystem).

Also controls whether to add `OxidizedFinder`'s path hook to `sys.path_hooks`.

Due to lack of control over low-level Python interpreter initialization, the standard library `PathFinder` will be registered on `sys.meta_path` and `sys.path_hooks` for a brief moment when the interpreter is initialized. If `sys.path` contains valid entries that would be serviced by this finder and `oxidized_importer` isn't able to service imports, it is possible for the path-based finder to be used to import some Python modules needed to initialize the Python interpreter. In many cases, this behavior is harmless. In all cases, the path-based importer is removed after Python interpreter initialization, so future imports won't be serviced by this path-based importer if it is disabled by this flag.

Default value: `true`

Interpreter initialization behavior: If false, path-based finders are removed from `sys.meta_path` and `sys.path_hooks` is cleared.

Type: `bool`

`packed_resources` Field

References to packed resources data.

The format of the data is defined by the `python-packed-resources` crate. The data will be parsed as part of initializing the custom meta path importer during interpreter initialization when `oxidized_importer=true`. If `oxidized_importer=false`, this field is ignored.

If paths are relative, that will be evaluated relative to the process's current working directory following the operating system's standard path expansion behavior.

Default value: `vec![]`

`Self::resolve()` behavior: `PackedResourcesSource::MemoryMappedPath` members have the special string `$ORIGIN` expanded to the string value that `Self::origin` resolves to.

This field is ignored during serialization.

Type: `Vec<PackedResourcesSource>`

`extra_extension_modules` Field

Extra extension modules to make available to the interpreter.

The values will effectively be passed to `PyImport_ExtendInitTab()`.

Default value: `None`

Interpreter initialization behavior: `PyImport_Inittab` will be extended with entries from this list. This makes the extensions available as built-in extension modules.

This field is ignored during serialization.

Type: `Option<Vec<ExtensionModule>>`

`argv` Field

Command line arguments to initialize `sys.argv` with.

Default value: `None`

`Self::resolve()` behavior: Some value is used if set. Otherwise `PythonInterpreterConfig::argv` is used if set. Otherwise `std::env::args_os()` is called.

Interpreter initialization behavior: the resolved Some value is used to populate `PyConfig.argv`.

Type: `Option<Vec<OsString>>`

`argvb` Field

Whether to set `sys.argvb` with bytes versions of process arguments.

On Windows, bytes will be UTF-16. On POSIX, bytes will be raw char* values passed to `int main()`.

Enabling this feature will give Python applications access to the raw `bytes` values of raw argument data passed into the executable. The single or double width bytes nature of the data is preserved.

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

Default value: `false`

Interpreter initialization behavior: `sys.argvb` will be set to a `list[bytes]`. `sys.argv` and `sys.argvb` should have the same number of elements.

Type: `bool`

`multiprocessing_auto_dispatch` Field

Automatically detect and run in `multiprocessing` mode.

If set, `crate::MainPythonInterpreter::run()` will detect when the invoked interpreter looks like it is supposed to be a `multiprocessing` worker and will automatically call into the `multiprocessing` module instead of running the configured code.

Enabling this has the same effect as calling `multiprocessing.freeze_support()` in your application code's `__main__` and replaces the need to do so.

Default value: `true`

Type: `bool`

`multiprocessing_start_method` Field

Controls how to call `multiprocessing.set_start_method()`.

Default value: `MultiprocessingStartMethod::Auto`

Interpreter initialization behavior: if `Self::oxidized_importer` is `true`, the `OxidizedImporter` will be taught to call `multiprocessing.set_start_method()` when `multiprocessing` is imported. If `false`, this value has no effect.

Type: `MultiprocessingStartMethod`

`sys_frozen` Field

Whether to set sys.frozen=True.

Setting this will enable Python to emulate "frozen" binaries, such as those used by PyInstaller.

Default value: `false`

Interpreter initialization behavior: If `true`, `sys.frozen = True`. If `false`, `sys.frozen` is not defined.

Type: `bool`

`sys_meipass` Field

Whether to set sys._MEIPASS to the directory of the executable.

Setting this will enable Python to emulate PyInstaller's behavior of setting this attribute. This could potentially help with self-contained application compatibility by masquerading as PyInstaller and causing code to activate *PyInstaller mode*.

Default value: `false`

Interpreter initialization behavior: If `true`, `sys._MEIPASS` will be set to a `str` holding the value of `Self::origin`. If `false`, `sys._MEIPASS` will not be defined.

Type: `bool`

`terminfo_resolution` Field

How to resolve the `terminfo` database.

Default value: `TerminfoResolution::Dynamic`

Interpreter initialization behavior: the `TERMINFO_DIRS` environment variable may be set for this process depending on what `TerminfoResolution` instructs to do.

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

Type: `TerminfoResolution`

`tcl_library` Field

Path to use to define the `TCL_LIBRARY` environment variable.

This directory should contain an `init.tcl` file. It is commonly a directory named `tclX.Y`. e.g. `tcl8.6`.

Default value: `None`

`Self::resolve()` behavior: the token `$ORIGIN` is expanded to the resolved value of `Self::origin`.

Interpreter initialization behavior: if set, the `TCL_LIBRARY` environment variable will be set for the current process.

Type: `Option<PathBuf>`

`write_modules_directory_env` Field

Environment variable holding the directory to write a loaded modules file.

If this value is set and the environment it refers to is set, on interpreter shutdown, we will write a `modules-<random>` file to the directory specified containing a `\n` delimited list of modules loaded in `sys.modules`.

This setting is useful to record which modules are loaded during the execution of a Python interpreter.

Default value: `None`

Type: `Option<String>`

`PythonInterpreterConfig` Struct

Holds configuration of a Python interpreter.

This struct holds fields that are exposed by `PyPreConfig` and `PyConfig` in the CPython API.

Other than the profile (which is used to initialize instances of `PyPreConfig` and `PyConfig`), all fields are optional. Only fields with `Some(T)` will be updated from the defaults.

`profile` Field

Profile to use to initialize pre-config and config state of interpreter.

Type: `PythonInterpreterProfile`

`allocator` Field

Name of the memory allocator.

See https://docs.python.org/3/c-api/init_config.html#c.PyPreConfig.allocator.

Type: `Option<Allocator>`

`configure_locale` Field

Whether to set the LC_CTYPE locale to the user preferred locale.

See https://docs.python.org/3/c-api/init_config.html#c.PyPreConfig.configure_locale.

Type: `Option<bool>`

`coerce_c_locale` Field

How to coerce the locale settings.

See https://docs.python.org/3/c-api/init_config.html#c.PyPreConfig.coerce_c_locale.

Type: `Option<CoerceCLocale>`

`coerce_c_locale_warn` Field

Whether to emit a warning if the C locale is coerced.

See https://docs.python.org/3/c-api/init_config.html#c.PyPreConfig.coerce_c_locale_warn.

Type: `Option<bool>`

`development_mode` Field

Whether to enable Python development mode.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.dev_mode.

Type: `Option<bool>`

`isolated` Field

Isolated mode.

See https://docs.python.org/3/c-api/init_config.html#c.PyPreConfig.isolated.

Type: `Option<bool>`

`legacy_windows_fs_encoding` Field

Whether to use legacy filesystem encodings on Windows.

See https://docs.python.org/3/c-api/init_config.html#c.PyPreConfig.legacy_windows_fs_encoding.

Type: `Option<bool>`

`parse_argv` Field

Whether argv should be parsed the way `python` parses them.

See https://docs.python.org/3/c-api/init_config.html#c.PyPreConfig.parse_argv.

Type: `Option<bool>`

`use_environment` Field

Whether environment variables are read to control the interpreter configuration.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.use_environment.

Type: `Option<bool>`

`utf8_mode` Field

Controls Python UTF-8 mode.

See https://docs.python.org/3/c-api/init_config.html#c.PyPreConfig.utf8_mode.

Type: `Option<bool>`

argv Field

Command line arguments.

These will become `sys.argv`.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.argv.

Type: `Option<Vec<OsString>>`

base_exec_prefix Field

Controls `sys.base_exec_prefix`.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.base_exec_prefix.

Type: `Option<PathBuf>`

base_executable Field

Controls `sys._base_executable`.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.base_executable.

Type: `Option<PathBuf>`

base_prefix Field

Controls `sys.base_prefix`.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.base_prefix.

Type: `Option<PathBuf>`

buffered_stdio Field

Controls buffering on `stdout` and `stderr`.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.buffered_stdio.

Type: `Option<bool>`

bytes_warning Field

Controls warnings/errors for some bytes type coercions.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.bytes_warning.

Type: `Option<BytesWarning>`

check_hash_pycs_mode Field

Validation mode for `.pyc` files.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.check_hash_pycs_mode.

Type: `Option<CheckHashPycsMode>`

configure_c_stdio Field

Controls binary mode and buffering on C standard streams.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.configure_c_stdio.

Type: `Option<bool>`

dump_refs Field

Dump Python references.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.dump_refs.

Type: `Option<bool>`

exec_prefix Field

Controls `sys.exec_prefix`.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.exec_prefix.

Type: `Option<PathBuf>`

executable Field

Controls `sys.executable`.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.executable.

Type: `Option<PathBuf>`

fault_handler Field

Enable `faulthandler`.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.faulthandler.

Type: `Option<bool>`

`filesystem_encoding` Field

Controls the encoding to use for filesystems/paths.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.filesystem_encoding.

Type: `Option<String>`

`filesystem_errors` Field

Filesystem encoding error handler.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.filesystem_errors.

Type: `Option<String>`

`hash_seed` Field

Randomized hash function seed.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.hash_seed.

Type: `Option<c_ulong>`

`home` Field

Python home directory.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.home.

Type: `Option<PathBuf>`

`import_time` Field

Whether to profile `import` time.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.import_time.

Type: `Option<bool>`

`inspect` Field

Enter interactive mode after executing a script or a command.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.inspect.

Type: `Option<bool>`

`install_signal_handlers` Field

Whether to install Python signal handlers.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.install_signal_handlers.

Type: `Option<bool>`

`interactive` Field

Whether to enable the interactive REPL mode.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.interactive.

Type: `Option<bool>`

`legacy_windows_stdio` Field

Controls legacy stdio behavior on Windows.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.legacy_windows_stdio.

Type: `Option<bool>`

`malloc_stats` Field

Whether to dump statistics from the `pymalloc` allocator on exit.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.malloc_stats.

Type: `Option<bool>`

`module_search_paths` Field

Defines `sys.path`.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.module_search_paths.

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

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

Type: `Option<Vec<PathBuf>>`

`optimization_level` Field

Bytecode optimization level.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.optimization_level.

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

Type: `Option<BytecodeOptimizationLevel>`

`parser_debug` Field

Parser debug mode.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.parser_debug.

Type: `Option<bool>`

`pathconfig_warnings` Field

Whether calculating the Python path configuration can emit warnings.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.pathconfig_warnings.

Type: `Option<bool>`

`prefix` Field

Defines `sys.prefix`.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.prefix.

Type: `Option<PathBuf>`

`program_name` Field

Program named used to initialize state during path configuration.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.program_name.

Type: `Option<PathBuf>`

`pycache_prefix` Field

Directory where `.pyc` files are written.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.pycache_prefix.

Type: `Option<PathBuf>`

python_path_env Field

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.pythonpath_env.

Type: `Option<String>`

quiet Field

Quiet mode.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.quiet.

Type: `Option<bool>`

run_command Field

Value of the `-c` command line option.

Effectively defines Python code to evaluate in `Py_RunMain()`.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.run_command.

Type: `Option<String>`

run_filename Field

Filename passed on the command line.

Effectively defines the Python file to run in `Py_RunMain()`.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.run_filename.

Type: `Option<PathBuf>`

run_module Field

Value of the `-m` command line option.

Effectively defines the Python module to run as `__main__` in `Py_RunMain()`.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.run_module.

Type: `Option<String>`

show_ref_count Field

Whether to show the total reference count at exit.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.show_ref_count.

Type: `Option<bool>`

`site_import` Field

Whether to import the `site` module at startup.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.site_import.

The `site` module is typically not needed for standalone applications and disabling it can reduce application startup time.

Type: `Option<bool>`

`skip_first_source_line` Field

Whether to skip the first line of `Self::run_filename`.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.skip_source_first_line.

Type: `Option<bool>`

`stdio_encoding` Field

Encoding of `sys.stdout`, `sys.stderr`, and `sys.stdin`.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.stdio_encoding.

Type: `Option<String>`

`stdio_errors` Field

Encoding error handler for `sys.stdout` and `sys.stdin`.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.stdio_errors.

Type: `Option<String>`

`tracemalloc` Field

Whether to enable `tracemalloc`.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.tracemalloc.

Type: `Option<bool>`

`user_site_directory` Field

Whether to add the user site directory to `sys.path`.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.user_site_directory.

Type: `Option<bool>`

verbose Field

Verbose mode.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.verbose.

Type: `Option<bool>`

warn_options Field

Options of the `warning` module to control behavior.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.warnoptions.

Type: `Option<Vec<String>>`

write_bytecode Field

Controls `sys.dont_write_bytecode`.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.write_bytecode.

Type: `Option<bool>`

x_options Field

Values of the `-X` command line options / `sys._xoptions`.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.xoptions.

Type: `Option<Vec<String>>`

MemoryAllocatorBackend Enum

Defines a backend for a memory allocator.

This says which memory allocator API / library to configure the Python interpreter to use.

Not all allocators are available in all program builds.

Serialization type: `string`

Default Variant
> The default allocator as configured by Python.
>
> This likely utilizes the system default allocator, normally the `malloc()`, `free()`, etc functions from the libc implementation being linked against.
>
> Serialized value: `default`

Jemalloc Variant
> Use the jemalloc allocator.
>
> Requires the binary to be built with jemalloc support.
>
> Never available on Windows.
>
> Serialized value: `jemalloc`

Mimalloc Variant

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

Requires the binary to be built with mimalloc support.

Serialized value: `mimalloc`

Snmalloc Variant

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

Not always available.

Serialized value: `snmalloc`

Rust Variant

Use Rust's global allocator.

The Rust allocator is less efficient than other allocators because of overhead tracking allocations. For optimal performance, use the default allocator. Or if Rust is using a custom global allocator, use the enum variant corresponding to that allocator.

Serialized value: `rust`

`PythonInterpreterProfile` Enum

Defines the profile to use to configure a Python interpreter.

This effectively provides a template for seeding the initial values of `PyPreConfig` and `PyConfig` C structs.

Serialization type: `string`.

Isolated Variant

Python is isolated from the system.

See https://docs.python.org/3/c-api/init_config.html#isolated-configuration.

Serialized value: `isolated`

Python Variant

Python interpreter behaves like `python`.

See https://docs.python.org/3/c-api/init_config.html#python-configuration.

Serialized value: `python`

`Allocator` Enum

Name of the Python memory allocators.

See https://docs.python.org/3/c-api/init_config.html#c.PyPreConfig.allocator.

Serialization type: `string`

NotSet Variant

Don't change memory allocators (use defaults).

Serialized value: `not-set`

Default Variant

Default memory allocators.

Serialized value: `default`

Debug Variant
> Default memory allocators with debug hooks.

> Serialized value: `debug`

Malloc Variant
> Use `malloc()` from the C library.

> Serialized value: `malloc`

MallocDebug Variant
> Force usage of `malloc()` with debug hooks.

> Serialized value: `malloc-debug`

PyMalloc Variant
> Python `pymalloc` allocator.

> Serialized value: `py-malloc`

PyMallocDebug Variant
> Python `pymalloc` allocator with debug hooks.

> Serialized value: `py-malloc-debug`

`BytecodeOptimizationLevel` Enum

An optimization level for Python bytecode.

Serialization type: `int`

Zero Variant
> Optimization level 0.

> Serialized value: `0`

One Variant
> Optimization level 1.

> Serialized value: `1`

Two Variant
> Optimization level 2.

> Serialized value: `2`

`BytesWarning` Enum

Defines what to do when comparing `bytes` or `bytesarray` with `str` or comparing `bytes` with `int`.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.bytes_warning.

Serialization type: `string`

None Variant
> Do nothing.

> Serialization value: `none`

Warn Variant
> Issue a warning.

> Serialization value: `warn`

Raise Variant
> Raise a `BytesWarning`.
>
> Serialization value: `raise`

CheckHashPycsMode Enum

Control the validation behavior of hash-based .pyc files.

See https://docs.python.org/3/c-api/init_config.html#c.PyConfig.check_hash_pycs_mode.

Serialization type: `string`

Always Variant
> Hash the source file for invalidation regardless of value of the `check_source` flag.
>
> Serialized value: `always`

Never Variant
> Assume that hash-based pycs always are valid.
>
> Serialized value: `never`

Default Variant
> The `check_source` flag in hash-based pycs determines invalidation.
>
> Serialized value: `default`

CoerceCLocale Enum

Holds values for `coerce_c_locale`.

See https://docs.python.org/3/c-api/init_config.html#c.PyPreConfig.coerce_c_locale.

Serialization type: `string`

LCCtype Variant
> Read the LC_CTYPE locale to decide if it should be coerced.
>
> Serialized value: `LC_CTYPE`

C Variant
> Coerce the C locale.
>
> Serialized value: `C`

MultiprocessingStartMethod Enum

Defines how to call `multiprocessing.set_start_method()` when `multiprocessing` is imported.

When set to a value that is not `none`, when `oxidized_importer.OxidizedFinder` services an import of the `multiprocessing` module, it will automatically call `multiprocessing.set_start_method()` to configure how worker processes are created.

If the `multiprocessing` module is not imported by `oxidized_importer.OxidizedFinder`, this setting has no effect.

Serialization type: `string`

None Variant
> Do not call `multiprocessing.set_start_method()`.
>
> This mode is what Python programs do by default.
>
> Serialized value: `none`

Fork Variant
> Call with value `fork`.
>
> Serialized value: `fork`

ForkServer Variant
> Call with value `forkserver`
>
> Serialized value: `forkserver`

Spawn Variant
> Call with value `spawn`
>
> Serialized value: `spawn`

Auto Variant
> Call with a valid appropriate for the given environment.
>
> This likely maps to `spawn` on Windows and `fork` on non-Windows.
>
> Serialized value: `auto`

`TerminfoResolution` Enum

Defines `terminfo` database resolution semantics.

Python links against libraries like `readline`, `libedit`, and `ncurses` which need to utilize a `terminfo` database (a set of files defining terminals and their capabilities) in order to work properly.

The absolute path to the terminfo database is typically compiled into these libraries at build time. If the compiled path on the building machine doesn't match the path on the runtime machine, these libraries cannot find the terminfo database and terminal interactions won't work correctly because these libraries don't know how to resolve terminal features. This can result in quirks like the backspace key not working in prompts.

The `pyembed` Rust crate is able to point libraries at a terminfo database at runtime, overriding the compiled-in default path. This enum is used to control that behavior.

Serialization type: `string`.

Dynamic Variant
> Resolve `terminfo` database using appropriate behavior for current OS.
>
> We will look for the terminfo database in paths that are common for the current OS / distribution. The terminfo database is present in most systems (except the most barebones containers or sandboxes) and this method is usually successfully in locating the terminfo database.
>
> Serialized value: `dynamic`

None Variant
> Do not attempt to resolve the `terminfo` database. Basically a no-op.
>
> This is what should be used for applications that don't interact with the terminal. Using this option will prevent some I/O syscalls that would be incurred by `dynamic`.
>
> Serialized value: `none`

Static Variant

Use a specified string as the `TERMINFO_DIRS` value.

Serialized value: `static:<path>`

e.g. `static:/usr/share/terminfo`.

Development Guide

Development of PyOxy strives to look like a normal Rust crate as much as possible. This means that normal workflows such as `cargo build` and `cargo test` should work.

Please note that if you are working from the root of the PyOxidizer Git repository, you may want to limit the package being operated on via e.g. `cargo build -p pyoxy` or `cargo test -p pyoxy`.

The `pyoxy` crate depends on `pyembed`, which depends on `pyo3`, which insists on finding a runnable and linkable Python install in `PATH`. You can set the `PYO3_PYTHON` environment variable to point at an explicit Python interpreter rather than utilizing the default search logic.

Building `pyoxy` With Embedded Python

If you just `cargo build`, it is likely that `pyo3` will pick up a non-portable, dynamically linked `libpython`. Furthermore, it will load Python resources from the filesystem, from the path configured in `libpython`. Such a configuration is not portable across machines!

To produce a portable, single file executable embedding `libpython` and its resources, we need to perform a little extra work. This essentially entails asking PyOxidizer to produce a static `libpython`, a Python packed resources file containing the standard library, and a PyO3 configuration file. Then, we build with a reference to that PyO3 configuration file to link the appropriate `libpython` and pick up the packed resources file.

```
cd pyoxy
pyoxidizer build --release
PYO3_CONFIG_FILE=$(pwd)/build/x86_64-unknown-linux-gnu/release/resources/pyo3-build-
→config-file.txt cargo build --release
```

On Linux, to ensure portability of the produced binary, you'll need to link against a sufficiently old (and therefore widely available) glibc version. This often entails running the build/link in an older Linux distribution, such as Debian Jessie.

On macOS, you'll probably want to define `MACOSX_DEPLOYMENT_TARGET` to the minimum version of macOS you want to target or else the produced binary won't be portable.

Project History

Blog Posts

- Announcing the PyOxy Python Runner (2022-05-10)

Changelog

0.3.0

Not yet released.

0.2.0

Released on 2022-05-30.

- Official release artifacts now contain Python 3.8 and 3.10 variants. Previously, only Python 3.9 was provided.

- The Sphinx documentation now contains extensive documentation for the Python interpreter configuration structs and enums. The content is derived from the canonical Rust source code. This should make it easier to understand the fields in YAML configurations without having to consult Rust crate docs.

- Release artifacts are now `.tar.gz` files and contain a `COPYING` file with licensing annotations.

- The release mechanism for PyOxy is now streamlined, hopefully enabling faster releases going forward.

0.1.0

Released 2022-05-10.

1.6 Tugger

Tugger is an umbrella project for implementing generic application packaging and distribution functionality. It is comprised as several Rust crates, each providing domain-specific functionality including:

- Debian packaging formats

- Software licensing

- Snapcraft packaging

- Apple code signing

- Rust toolchain installation

- Windows installer generation

- And much more

Tugger defines Starlark primitives for scripting common application packaging and distribution actions.

Tugger is used by PyOxidizer for performing functionality that isn't specific to Python.

There are aspirations to make Tugger a standalone tool someday. But for now, it is only available as a series of Rust crates.

1.6.1 Shipping Applications with `tugger`

The Tugger project aims to make it easy to ship applications. It does so by implementing generic functionality related to application distribution in a myriad (fleet?) of individual, domain-specific crates. See *Modular Crate Architecture* for more. Tugger supports generating distributable artifacts in common formats such as Windows `.msi` installers, Debian `.deb` files, and Snapcraft `.snap` files.

Tugger's Rust crates can be consumed as regular Rust library crates by any project and are explicitly designed for this use case. Tugger also defines a Starlark dialect (Starlark is a Python-like configuration language), enabling applications to define packaging functionality in configuration files, which Tugger can execute. The Starlark dialect is effectively a scriptable interface to Tugger's Rust internals.

Tugger is part of the PyOxidizer Project and is developed inside the PyOxidizer repository at https://github.com/indygreg/PyOxidizer. However, Tugger is designed to be a standalone project and doesn't require PyOxidizer.

Overview

Tugger aims to be a generic tool to help application maintainers ship their applications to end-users.

Tugger can be thought of a specialized build system for distributable artifacts (Windows MSI installers, Debian packages, RPMs, etc). However, Tugger itself is generally not concerned with details of how a particular file is built: Tugger's role is to consume existing files and *package* them into artifacts that are distributed/installed on other machines.

Designed to Be Platform Agnostic

An explicit goal of Tugger is to be platform agnostic and to have as much functionality implemented in-process. For example, it should be possible to produce a Linux `.deb` from Windows, a Windows MSI installer from macOS, or a macOS DMG from Linux without any out-of-process dependencies.

Tugger attempts to implement packaging functionality in Rust with minimal dependence on external tools. For example, RPMs and Debian packages are built by constructing the raw archive files using Rust code rather than calling out to tools like `rpmbuild` or `debuild`. This enables Tugger to build artifacts that don't target the current architecture or operating system.

While Tugger may not achieve this goal for all distributable formats and architectures, it is something that Tugger strives to do.

File Centric View

Tugger attempts to take a file-centric view towards packaging. This helps achieve platform independent and *cross-compiling*. What this means in practice is many of Tugger's packaging facilities operate by taking an input set of files and assembling them into some other distributable format. Contrast this with specialized tools for each distributable format, which generally invoke a custom build system and have domain-specific configuration files.

A side-effect of this decision is that Tugger is often not aware of build systems: it is often up to you to script Tugger to produce the files you wish to distribute.

Modular Crate Architecture

Tugger is composed of a series - a *fleet* if you will - of Rust crates. Each Rust crate provides domain-specific functionality. While the Rust crates are part of the Tugger project, an attempt is made to implement them such that they can be used outside of Tugger.

The following crates compose Tugger's crate *fleet*:

`tugger-binary-analysis`
    Analyze platform native binaries. Finds library dependencies. Identifies Linux distribution compatibility. Etc.

`tugger-common`
    Shared functionality required by multiple crates. This entails things like downloading files, shared test code, etc.

`tugger-debian`
    Debian packaging primitives. Parsing and serializing control files. Writing `.deb` files.

`tugger-file-manifest`
    A virtual manifest of a collection of files. Virtual file manifests are used throughout Tugger to represent a collection of files, their content, and file metadata.

`tugger-rpm`
    RPM packaging primitives.

`tugger-snapcraft`
    Snapcraft packaging. Represent `snapcraft.yaml` files. Invoke `snapcraft` to produce `.snap` files.

`tugger-windows`
    Windows-specific functionality. Finding the Microsoft SDK and Visual C++ Redistributable files. Signing Windows binaries.

`tugger-wix`
    Interface to the WiX Toolset (produces Windows `.msi` and `.exe` installers). Can build Windows installers with little-to-no knowledge about how the WiX Toolset works.

`tugger`
    The primary crate. Implements Starlark dialect and driver code for running it. This crate has minimal use as a library, as most library functionality is within the domain-specific crates.

Tugger Starlark Dialect

Tugger uses Starlark files to configure run-time behavior.

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

Tugger defines its own *dialect* of Starlark - types and functions - specific to Tugger.

Global Symbols

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

The Starlark environment contains symbols from the following:

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

Global Types

Tugger's Starlark dialect defines the following custom types:

AppleUniversalBinary
    Represents a multi-architecture *universal* binary for Apple platforms.

CodeSigner
    An entity capable of performing code signing.

CodeSigningRequest
    Holds settings to influence code signing on a single entity.

FileContent
    Represents the content of a file on the filesystem.

FileManifest
    Represents a mapping of filenames to file content.

MacOsApplicationBundleBuilder
    Used to create macOS Application Bundles (i.e. `.app` directories).

PythonWheelBuilder
    Create Python wheels (*.whl* files) from settings and file content.

SnapApp
    Represents an application inside a `snapcraft.yaml` file.

SnapPart
    Represents a part inside a `snapcraft.yaml` file.

Snap
    Represents a `snapcraft.yaml` file.

SnapcraftBuilder
    Manages the environment and invocations of the `snapcraft` command.

WiXBundleBuilder
    Produce a Windows exe installer containing multiple installers using WiX.

WiXInstaller
    Produce a Windows installer using WiX.

WiXMSIBuilder
    Produce a Windows MSI installer with common installer features using WiX.

Global Functions

Tugger's Starlark dialect defines the following global functions:

glob()
    Collect files from the filesystem.

Functions for Interacting with the Filesystem

starlark_tugger.**glob**(*include=List[str]*, *exclude=Optional[List[str]]*, *strip_prefix=Optional[str]*) →
                   FileManifest

> The glob() function resolves file patterns to a `starlark_tugger.FileManifest`.
>
> This function accepts the following arguments:
>
> **include**
> > Defines file patterns that will be matched using the glob Rust crate. If patterns begin with / or look like
> > a filesystem absolute path, they are absolute. Otherwise they are evaluated relative to the directory of the
> > current config file.
>
> **exclude**
> > File patterns used to exclude files from the result. All patterns in `include` are evaluated before `exclude`.
>
> **strip_prefix**
> > Prefix to strip from the beginning of matched files. `strip_prefix` is stripped after `include` and `exclude`
> > are processed.

Functions for Interacting with the Terminal

starlark_tugger.**can_prompt**() → bool

> Returns whether we are capable of prompting for user input.
>
> If this returns `False`, functions like *prompt_input()* and *prompt_password()* will be unable to collect input
> from the user and will error unless a default value is provided.

starlark_tugger.**prompt_confirm**(*prompt: str*, *default: Optional[bool] = None*) → bool

> Prompt the user to confirm something.
>
> This will print the provided prompt and wait for user input to confirm it.
>
> If `y` or `n` is pressed, this evaluates to `True` or `False`, respectively. If the escape key is pressed, an error is raised.
>
> If stdin is not interactive (e.g. it is connected to /dev/null), this will return `default` if provided or raise an
> error otherwise.

starlark_tugger.**prompt_input**(*prompt: str*, *default: Optional[str] = None*) → str

> Prompt the user for input on the terminal.
>
> This will print a prompt with the given `prompt` text to stderr. If `default` is provided, the default value will
> printed and used if no input is provided.
>
> The string constituting the raw input (without a trailing newline) is returned.
>
> If stdin is not interactive (e.g. it is connected to /dev/null), this will return `default` if provided or raise an
> error otherwise.

starlark_tugger.**prompt_password**(*prompt: str*, *confirm: bool = False*, *default: Optional[str] = None*) → str

> Prompt the user for a password input on the terminal.
>
> This will print a prompt with the given `prompt` text to stderr.
>
> When the user inputs their password, its content will not be printed back to the terminal.
>
> If `confirm` is `True`, the user will be prompted to confirm the hidden value they just entered and subsequent
> prompts will be attempted until values agree.

If stdin is not interactive (e.g. it is connected to /dev/null), this will return default if provided or raise an error if not.

The password value is stored in plain text in memory and treated like any other string value.

AppleUniversalBinary

class starlark_tugger.**AppleUniversalBinary**

Represents a *universal/fat/multi-architecture* Mach-O binary - the executable file format used by Apple operating systems.

Instances exist to facilitate the creation of universal binaries from source Mach-O binaries. This type provides similar functionality to the lipo tool, which is Apple's tool for interfacing with universal binaries.

__init__(*filename: str*) → *AppleUniversalBinary*

Construct a new instance representing an empty binary having the given filename.

add_path(*path: str*)

Add a binary from a given filesystem path to this instance.

This effectively marks the binary for inclusion when we go to produce a new *universal* binary.

The file can be a single architecture Mach-O or *universal* Mach-O. If *universal*, all architectures within that file will be added.

add_file(*content: FileContent*)

Add a binary from the given `FileContent` instance to this instance.

This is like `AppleUniversalBinary.add_path()` except the content of the binary comes from a `FileContent` instance instead of the filesystem.

to_file_content() → *FileContent*

Convert this instance to a `FileContent`.

The content of the returned object will be a just-in-time produced *universal* Mach-O binary.

write_to_directory(*path: str*) → str

Write a file containing this *universal* Mach-O binary into the directory specified.

Absolute paths are accepted as-is. Relative paths are relative to the currently configured *build* path.

Returns the absolute path of the written file.

CodeSigner

class starlark_tugger.**CodeSigner**

Instances of `CodeSigner` are used to digitally sign code or content.

When instances are registered in your Starlark configuration file, they will automatically be used to sign entities.

See *Code Signing* for details on what code signing is supported.

activate()

Registers this instance with Tugger so that it is consulted when code signing events occur.

Once this method is called, subsequent mutations to the instance may or may not be reflected with the instance that is registered to handle events.

Failure to call this method will mean this instance won't be queried to handle code signing events as Tugger runs.

chain_issuer_certificates_pem_file(*path: str*)

> Register PEM encoded X.509 certificates located in a file to the certificate chain.
>
> The file should have content like `-----BEGIN CERTIFICATE-----`. Multiple certificates can exist in a single file.
>
> See *Understanding Code Signing Certificates* for the meaning of the certificate chain.

chain_issuer_certificates_macos_keychain()

> Register the issuer certificate chain by looking for certificates in the macOS keychain.
>
> This function only works on macOS and will raise errors when called on other platforms.
>
> See *Understanding Code Signing Certificates* for the meaning of the certificate chain.

set_time_stamp_server(*path: str*)

> Set the URL of a Time-Stamp Protocol server to use.
>
> Calling this is not necessary when signing Apple primitives, as Apple's server will be used automatically.
>
> Calling this will force the use of a particular time-stamp protocol server.

set_signing_callback(*f: Callable*)

> Defines a function that will be invoked when Tugger has encountered a signable entity that this instance is capable of signing.
>
> The function's signature is: `def callback(request:  CodeSigningRequest) -> Union[bool, dict, None]`.
>
> The function receives as its arguments:
>
> **request**
> > The *CodeSigningRequest* that is about to be signed.
>
> The *CodeSigningRequest* passed in is unique to this *CodeSigner* instance and can be used to inspect the imminent code signing operation or influence how it is performed - even preventing it entirely. See *CodeSigningRequest* for the full API documentation.

Constructor Functions

`starlark_tugger.`**code_signer_from_pfx_file**(*path: str*, *password: str*) → *CodeSigner*

> Construct a *CodeSigner* by specifying the path to a PFX file.
>
> PFX files are commonly used to hold code a code signing key and its corresponding x509 certificate. These files typically have the extension `.pfx` or `.p12`.
>
> PFX files require a password to read. It is possible for the password to be the empty string (`""`). If you did not supply a password when exporting the code signing certificate, the password is likely the empty string.
>
> The password can be collected interactively via the *prompt_password()* function.

`starlark_tugger.`**code_signer_from_windows_store_sha1_thumbprint**(*thumbprint: str*, *store: str = 'my'*)
> → *CodeSigner*

> Construct a *CodeSigner* that uses a certificate in the Windows certificate store having the specified SHA-1 thumbprint.
>
> This is the most reliable way to specify a certificate in the Windows certificate store, as SHA-1 thumbprints should uniquely identify a certificate.
>
> `store` denotes the Windows certificate store to use. Possible values are `my`, `root`, `trust`, `ca`, and `userds` (all case-insensitive). The meaning of these values is described in Microsoft's documentation.

starlark_tugger.**code_signer_from_windows_store_subject**(*subject: str*, *store: str = 'my'*) →
CodeSigner

> Construct a *CodeSigner* using a code signing certificate in a Windows certificate store.
>
> subject defines a string value that is used to locate the certificate in the store. The string value is matched against the subject field of the certificate (who the certificate was issued to). Its value is often the name of someone or something.
>
> See *code_signer_from_windows_store_sha1_thumbprint()* for accepted values for the store argument.

starlark_tugger.**code_signer_from_windows_store_auto**() → *CodeSigner*

> Construct a *CodeSigner* that automatically chooses a code signing certificate from the Windows certificate store.
>
> This will choose the *best available* found certificate. The heuristics are not well-defined and may change over time. For reliable results, use a different method.

CodeSigningRequest

class starlark_tugger.**CodeSigningRequest**

> This type represents the invocation of and settings for a single code signing operation.
>
> When *CodeSigner* instances are registered with Tugger, they can optionally register a callback function via *CodeSigner.set_signing_callback()* to influence the imminent code signing operation. This type is used to convey information about the code signing operation and to influence its settings.
>
> Instances are constructed internally by Tugger and cannot be constructed via Starlark.
>
> **action**
>
> > (read-only str)
> >
> > The named action that triggered this code signing request.
>
> **filename**
>
> > (read-only str)
> >
> > The filename this request is associated with. This is only the filename: not a full filesystem path.
>
> **path**
>
> > (read-only Union[str, None])
> >
> > The filesystem path this request is associated with. May be None. The path may be a *virtual* path, such as one tracked in a *FileManifest* instance.
>
> **defer**
>
> > (write-only bool)
> >
> > Whether to defer processing of this request to another signer.
> >
> > Normally, the first *CodeSigner* that is capable of signing something attempts to sign it and *CodeSigner* traversal is stopped. Setting this to True will enable additional *CodeSigner* (or callback functions on the same signer) to encounter this request.
>
> **prevent_signing**
>
> > (write-only bool)
> >
> > If set to True, the resource will not be signed and the signing attempt will be aborted.

FileContent

class starlark_tugger.**FileContent**

> This type represents the content of a single file.
>
> Instances essentially track the following:
>
> > • The content of a file (either a reference to a filesystem path or in-memory data).
> >
> > • Whether the file is executable.
> >
> > • The filename associated with the content. This is just the file name: directory components are not allowed.
>
> Unfortunately, since Starlark doesn't expose a `bytes` type, we are unable to expose the raw content tracked by instances of this type.
>
> **executable**
>
> > (`bool`)
> >
> > Whether a materialized file should be marked as executable.
>
> **filename**
>
> > (`str`)
> >
> > The filename associated with this instance.
> >
> > This is just the filename.
>
> **__init__**(*path: Optional[str] = None, filename: Optional[str] = None, content: Optional[str] = None, executable: Optional[bool] = None*) → *FileContent*
>
> > Construct a new instance given an existing filesystem `path` or string `content`.
> >
> > 1 of `path` or `content` must be provided to define the content tracked by this instance.
> >
> > If `content` is provided, `filename` must also be provided.
> >
> > `filename` must be just a file name: no directory components are allowed.
> >
> > If `path` is provided, it must refer to an existing filesystem path or an error will occur. Relative paths are interpreted as relative to the global `CWD` variable. Absolute paths are used as-is.
> >
> > If `path` is provided, by default `filename` and `executable` will be resolved from the given path. However, if the `filename` or `executable` arguments are not `None`, their values will be override those derived from `path`.
> >
> > If `content` is provided and `executable` is not, `executable` defaults to `False`.
>
> **write_to_directory**(*path: str*) → str
>
> > Materialize this instance as a file in a directory.
> >
> > Absolute paths are treated as is. Relative paths are relative to the currently configured build directory.
> >
> > Returns the absolute path of the file that was written.

FileManifest

class starlark_tugger.**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.
>
> **add_manifest**(*manifest:* FileManifest)
>
>> This method overlays another :py:class`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.
>
> **add_file**(*content:* FileContent, *path: Optional[str] = None*, *directory: Optional[str] = None*)
>
>> Add a *FileContent* instance to this manifest, optionally controlling its path within the manifest.
>>
>> If neither path nor directory are specified, the file will be materialized in the root directory of the manifest with the filename given by *FileContent.filename*.
>>
>> If path is provided, it defines the exact path within the manifest to use.
>>
>> If directory is provided, the manifest path is effectively computed the same as os.path. join(directory, content.filename).
>>
>> An error occurs if both path and directory are non-None.
>
> **add_path**(*path: str*, *strip_prefix: str*, *force_read: bool = False*)
>
>> This method adds a file on the filesystem to the manifest.
>>
>> The following arguments are accepted:
>>
>> **path**
>>> The filesystem path to add.
>>
>> **strip_prefix**
>>> The string prefix to strip from the path. The remaining path will be stored in the manifest.
>>
>> **force_read**
>>> Whether to read the file data into memory now.
>>>
>>> This can be set when reading temporary files.
>
> **get_file**(*path: str*) → Optional[*FileContent*]
>
>> Obtain a *FileContent* at a given path in the manifest, or None if no such path exists in the manifest.
>>
>> A copy of the content in the *FileManifest* is returned and mutations on the returned object will not be reflected in the *FileManifest*.
>
> **install**(*path: str*, *replace: bool = 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*.
>>
>> Upon successful materialization of all files in the manifest, all written files will be assessed for code signing with the file-manifest-install *action*.
>
> **paths**() → list[str]
>
>> Obtain all paths currently tracked by this instance.

remove(*path: str*) → Optional[*FileContent*]

> Remove the entry in this manifest at `path`, returning a `FileContent` representing the removed entry if there was one or `None` if the path isn't tracked by the manifest.

MacOsApplicationBundleBuilder

class starlark_tugger.**MacOsApplicationBundleBuilder**

> The `MacOsApplicationBundleBuilder` type allows creating *macOS Application Bundles* (typically `.app` directories) providing applications on macOS.
>
> For reference, see Apple's bundle format documentation for the structure of application bundles.
>
> **__init__**(*bundle_name: str*) → *MacOsApplicationBundleBuilder*
>
> > Construct new instances. It accepts the following arguments:
> >
> > **bundle_name**
> >
> > > The name of the application bundle.
> > >
> > > This will become the value for `CFBundleName` and form the name of the generated bundle directory.
>
> **add_icon**(*path: str*)
>
> > Accepts a `string` argument defining the path to a file that will become the `<bundle_name>.icns` file for the bundle.
>
> **add_manifest**(*manifest:* FileManifest)
>
> > Adds file data to the bundle via a `FileManifest` instance. All files in the manifest will be materialized in the `Contents/` directory of the bundle.
> >
> > Accepts the following arguments:
> >
> > **manifest**
> >
> > > Collection of files to materialize.
> >
> > Bundles have a well-defined structure and files should only be materialized in certain locations. This method will allow you to materialize files in locations resulting in a malformed bundle. Use with caution.
>
> **add_macos_file**(*content:* FileContent, *path: Optional[str] = None*)
>
> > Adds a single file to be installed in the `Contents/MacOS` directory in the bundle.
> >
> > Accepts the following arguments:
> >
> > **content**
> >
> > > Object representing file content to materialize.
> >
> > **path**
> >
> > > Relative path of file under `Contents/MacOS`. If not defined, the file will be installed into the equivalent of `os.path.join("Contents/MacOS", content.filename)`.
>
> **add_macos_manifest**(*manifest: FileManifest*))
>
> > Adds a `FileManifest` of content to be materialized in the `Contents/MacOS` directory.
> >
> > Accepts the following arguments:
> >
> > **manifest**
> >
> > > Collection of files to materialize.
>
> **add_resources_file**(*content:* FileContent, *path: Optional[str]*)
>
> > Adds a single file to be installed in the `Contents/Resources` directory in the bundle.
> >
> > Accepts the following arguments:

content
> Object representing file content to materialize.

path
> Relative path of file under `Contents/Resources`. If not defined, the file will be installed into the equivalent of `os.path.join("Contents/Resources", content.filename)`.

add_resources_manifest(*manifest:* FileManifest)

> Adds a `FileManifest` of content to be materialized in the `Contents/Resources` directory.

> Accepts the following arguments:

manifest
> Collection of files to materialize.

set_info_plist_key(*key: str*, *value: Union[bool, int, str]*)

> Sets the value of a key in the `Contents/Info.plist` file.

> Accepts the following arguments:

key
> Key in the `Info.plist` file to set.

value
> Value to set. Can be a `bool`, `int`, or `string`.

set_info_plist_required_keys(*display_name: str*, *identifier: str*, *version: str*, *signature: str*, *executable: str*)

> This method defines required keys in the `Contents/Info.plist` file.

> The following named arguments are accepted and must all be provided:

display_name
> Sets the bundle display name (`CFBundleDisplayName`).

> This is the name of the application as displayed to users.

identifier
> Sets the bundle identifier (`CFBundleIdentifer`).

> This is a reverse DNS type identifier. e.g. `com.example.my_program`.

version
> Sets the bundle version string (`CFBundleVersion`)

signature
> Sets the bundle creator OS type code (`CFBundleSignature`).

> The value must be exactly 4 characters.

executable
> Sets the name of the main executable file (`CFBundleExecutable`).

> This is typically the same name as the bundle.

build(*target: str*)

> This method will materialize the `.app` bundle/directory given the settings specified.

> This method accepts the following arguments:

target
> The name of the target being built.

Upon successful bundle directory creation, the entire bundle is considered for code signing with the signing action `macos-application-bundle-creation`. All signable Mach-O files and nested bundles should be signed.

write_to_directory(*path: str*)

This method will materialize the `.app` bundle/directory to the specified directory.

Absolute paths are treated as-is. Relative paths are relative to the currently configured build path.

Upon successful bundle directory creation, the entire bundle is considered for code signing with the signing action `macos-application-bundle-creation`. All signable Mach-O files and nested bundles should be signed.

PythonWheelBuilder

class `starlark_tugger.`**PythonWheelBuilder**

The `PythonWheelBuilder` type facilitates creating Python wheels (`.whl` files) from settings and file content.

Python wheels are zip files with some well-defined files describing the wheel and the entity that is packaged. See PEP 427 for more on the wheel format and how it works.

By default, new instances target the *compatibility tag* `py3-none-any`. This is suitable for a wheel containing pure Python code (`.py` files) and no binary files. If your wheel contains binary files or is limited in the Python compatibility in any way, you should modify the *compatibility* tag by setting instance attributes accordingly.

By default, the `.dist-info/WHEEL`, `.dist-info/METADATA`, and `.dist-info/RECORD` files will be derived automatically from settings upon wheel creation. It is possible to provide your own custom file content for the `.dist-info/WHEEL` and `.dist-info/METADATA` files by calling *PythonWheelBuilder.add_file_dist_info()*. A custom `.dist-info/RECORD` file, if provided, will be ignored.

__init__(*distribution: str*, *version: str*) → *PythonWheelBuilder*

Construct a new instance to produce a wheel for a given `distribution` (read: Python package) and `version` of that *distribution*.

build_tag

(`Optional[str]`)

The *build tag* for this wheel. This constitutes an extra component in the wheel's filename and metadata.

Build tags are typically not set on released versions: only for in-development, pre-release versions.

tag

(`str`)

The *compatibility tag* for this wheel.

This is equivalent to {`python_tag`}-{`abi_tag`}-{`platform_tag`}.

python_tag

(`str`)

The *Python tag* component of the wheel's *compatibility tag*. This should be a value like `py3` or `py39`.

abi_tag

(`str`)

The *ABI tag* component of the wheel's *compatibility tag*. This should be a value like `none`, `abi3`, or `cp39`.

platform_tag

> (str)
>
> The *platform tag* component of the wheel's *compatibility tag*. This should be a value like `any`, `linux_x86_64`, `manylinux2010_x86_64`, `macosx_10_9_x86_64`, etc.

generator

> (str)
>
> Describes the thing that constructed the wheel. This value is added to the default `.dist-info/WHEEL` file produced for this instance.

root_is_purelib

> (bool)
>
> The value for the `Root-Is-Purelib` setting for the wheel.
>
> If `True`, the wheel is extracted to Python's `purelib` directory. If `False`, to `platlib`.
>
> This should be set to `True` if the wheel contains pure Python files (no binary files).

modified_time

> (int)
>
> The file modification time for files in wheel zip archives in seconds since UNIX epoch.
>
> Default value is the time this instance was created.

wheel_file_name

> (read-only str)
>
> The file name the wheel should be materialized as.
>
> Wheel filenames are derived from the distribution, version, build tag, and *compatibility tag*.

add_file_dist_info(*file:* FileContent, *path: Optional[str] = None, directory: Optional[str] = None*)

> Add a *FileContent* to the wheel in the `.dist-info/` directory for the distribution being packaged.
>
> If neither `path` nor `directory` are specified, the file will be materialized in the `.dist-info/` directory with the filename given by *FileContent.filename*.
>
> If `path` is provided, it defines the exact path under `.dist-info/` to use.
>
> If `directory` is provided, the path is effectively `os.path.join(directory, file.filename)`.

add_file_data(*destination:* str, *file:* FileContent, *path: Optional[str] = None, directory: Optional[str] = None*)

> Add a *FileContent* to the wheel in a `.data/<destination>/` directory.
>
> `destination` represents a known Python installation directory. Recognized values include `purelib`, `platlib`, `headers`, `scripts`, `data`. `destination` effectively maps different file types to appropriate installation paths on wheel installation.
>
> If neither `path` nor `directory` are specified, the file will be materialized in the `.data/<destination>` directory with the filename given by *FileContent.filename*.
>
> If `path` is provided, it defines the exact path under `.data/<destination>` to use.
>
> If `directory` is provided, the path is effectively `os.path.join(directory, file.filename)`.

add_file(*file:* FileContent, *path: Optional[str] = None, directory: Optional[str] = None*)

>   Add a *FileContent* to the wheel.

>   If neither `path` nor `directory` are specified, the file will be materialized in the root directory with the filename given by *FileContent.filename*.

>   If `path` is provided, it defines the exact path in the wheel.

>   If `directory` is provided, the path is effectively `os.path.join(directory, file.filename)`.

to_file_content() → *FileContent*

>   Obtain a *FileContent* representing the built wheel.

>   The returned instance will have its *FileContent.filename* set to the appropriate name for this wheel given current settings. The data in the file should be a zip archive containing a well-formed Python wheel.

write_to_directory(*path: str*) → str

>   Write a `.whl` file to the given directory (specified by `path`) with the current state in this builder instance.

>   Returns the path of the written file.

build(*target: str*) → *ResolvedTarget*

>   Build the instance.

>   This is equivalent to *PythonWheelBuilder.write_to_directory()*, writing out the wheel to the build directory for the named target.

ResolvedTarget

class `starlark_tugger.`**ResolvedTarget**

>   Represents a *build* target that has been resolved.

SnapApp

class `starlark_tugger.`**SnapApp**

>   The `SnapApp` type represents an application entry in a `snapcraft.yaml` file. Specifically, this type represents the values of `apps.<app-name>` keys.

>   See https://snapcraft.io/docs/snapcraft-yaml-reference for more documentation.

>   Instances of `SnapApp` expose attributes that map to the keys within `apps.<app-name>` entries in `snapcraft.yaml` configuration files.

>   Currently the attributes are write only.

>   Setting an attribute value to `None` has the side-effect of removing that attribute from the serialized `snapcraft.yaml` file.

>   See https://snapcraft.io/docs/snapcraft-yaml-reference for detailed documentation about what each attribute means.

__init__() → *SnapApp*

>   SnapApp() creates an empty instance. It accepts no arguments.

adapter

>   (`Optional[str]`)

autostart
> (Optional[str])

command_chain
> (Optional[list[str]])

command
> (Optional[str])

common_id
> (Optional[str])

daemon
> (Optional[str])

desktop
> (Optional[str])

environment
> (Optional[list[str]])

extensions
> (Optional[list[str]])

listen_stream
> (Optional[str])

passthrough
> (Optional[dict[str, str]])

plugs
> (Optional[list[str]])

post_stop_command
> (Optional[str])

restart_condition
> (Optional[str])

slots
> (Optional[list[str]])

stop_command
> (Optional[str])

stop_timeout
> (Optional[str])

timer
> (Optional[str])

socket_mode
> (Optional[int])

socket
> (Optional[dict[str]])

SnapPart

class starlark_tugger.**SnapPart**

    The SnapPart type represents a part entry in a `snapcraft.yaml` file. Specifically, this type represents the values of `parts.<part-name>` keys.

    See https://snapcraft.io/docs/snapcraft-yaml-reference for more documentation.

    Instances of SnapPart expose attributes that map to the keys within `parts.<part-name>` entries in `snapcraft.yaml` configuration files.

    Currently the attributes are write only.

    Setting an attribute value to `None` has the side-effect of removing that attribute from the serialized `snapcraft.yaml` file.

    See https://snapcraft.io/docs/snapcraft-yaml-reference for detailed documentation about what each attribute means.

    __init__() → *SnapPart*

        SnapPart() creates an empty instance. It accepts no arguments.

    after

        (Optional[list[str]])

    build_attributes

        (Optional[list[str]])

    build_environment

        (Optional[list[dict[str, str]]])

    build_packages

        (Optional[list[str]])

    build_snaps

        (Optional[list[str]])

    filesets

        (Optional[dict[str, list[str]]])

    organize

        (Optional[dict[str, str]])

    override_build

        (Optional[str])

    override_prime

        (Optional[str])

    override_pull

        (Optional[str])

    override_stage

        (Optional[str])

    parse_info

        (Optional[str])

plugin

>   (Optional[str])

prime

>   (Optional[list[str]])

source_branch

>   (Optional[str])

source_checksum

>   (Optional[str])

source_commit

>   (Optional[str])

source_depth

>   (Optional[int])

source_subdir

>   (Optional[str])

source_tag

>   (Optional[str])

source_type

>   (Optional[str])

source

>   (Optional[str])

stage_packages

>   (Optional[list[str]])

stage_snaps

>   (Optional[list[str]])

stage

>   (Optional[list[str]])

Snap

class starlark_tugger.**Snap**

>   The Snap type represents an entire `snapcraft.yaml` file.
>
>   See https://snapcraft.io/docs/snapcraft-yaml-reference for more documentation.
>
>   Instances of `Snap` expose attributes that map to the keys within `snapcraft.yaml` files.
>
>   Currently the attributes are write only.
>
>   Setting an attribute value to `None` has the side-effect of removing that attribute from the serialized `snapcraft.yaml` file.
>
>   See https://snapcraft.io/docs/snapcraft-yaml-reference for detailed documentation about what each attribute means.

__init__(*name: str*, *version: str*, *summary: str*, *description: str*)

Creates an instance initialized with required parameters. It accepts the following arguments:

`name version summary description`

adopt_info

(Optional[str])

apps

(Optional[dict[str, SnapApp]])

architectures

(Optional[dict["build_on" | "run_on", str]])

assumes

(Optional[list[str]])

base

(Optional[str])

confinement

(Optional[str])

description

(str)

grade

(Optional[str])

icon

(Optional[str])

license

(Optional[str])

name

(str)

passthrough

(Optional[dict[str, str]])

parts

(Optional[dict[str, SnapPart]])

plugs

(Optional[dict[str, list[str]]])

slots

(Optional[dict[str, list[str]]])

summary

(str)

title

(Optional[str])

type

(Optional[str])

> **version**
>
> > (str)
>
> **to_builder**() → *SnapcraftBuilder*
>
> > Converts this instance into a *SnapcraftBuilder*.
> >
> > This method accepts no arguments and is equivalent to calling SnapcraftBuilder(self).

SnapcraftBuilder

class starlark_tugger.**SnapcraftBuilder**

> The SnapcraftBuilder type coordinates the invocation of the snapcraft command.
>
> **__init__**(*snap:* Snap) → *SnapcraftBuilder*
>
> > SnapcraftBuilder() constructs a new instance from a *Snap*.
> >
> > It accepts the following arguments:
> >
> > **snap**
> >
> > > The *Snap* defining the configuration to be used.
>
> **add_invocation**(*args: List[str]*, *purge_build: Optional[bool]*)
>
> > This method registers an invocation of snapcraft with the builder. When this instance is built, all registered invocations will be run sequentially.
> >
> > The following arguments are accepted:
> >
> > **args**
> >
> > > Arguments to pass to snapcraft executable.
> >
> > **purge_build**
> >
> > > Whether to purge the build directory before running this invocation.
> > >
> > > If not specified, the build directory is purged for the first registered invocation and not purged for all subsequent invocations.
>
> **add_file_manifest**(*manifest:* FileManifest)
>
> > This method registers the content of a *FileManifest* with the build environment for this builder.
> >
> > When this instance is built, the content of the passed manifest will be materialized in a directory next to the snapcraft.yaml file this instance is building.
> >
> > The following arguments are accepted:
> >
> > **manifest**
> >
> > > Defines files to install in the build environment.
>
> **build**(*target: str*) → *ResolvedTarget*
>
> > This method invokes the builder and runs snapcraft.
> >
> > The following arguments are accepted:
> >
> > **target**
> >
> > > The name of the build target.
> >
> > This method returns a ResolvedTarget. That target is not runnable.

WiXBundleBuilder

class starlark_tugger.**WiXBundleBuilder**

>The WiXBundleBuilder type allows building simple *bundle* installers with the WiX Toolset.
>
>WiXBundleBuilder instances allow you to create .exe installers that are composed of a chain of actions. At execution time, each action in the chain is evaluated. See the WiX Toolset documentation for more.
>
>**__init__**(*id_prefix: str*, *name: str*, *version: str*, *manufacturer: str*, *arch: str = 'x64'*) → *WiXBundleBuilder*
>
>>WiXBundleBuilder() is called to construct new instances. It accepts the following arguments:
>>
>>**id_prefix**
>>>The string prefix to add to auto-generated IDs in the .wxs XML.
>>>
>>>The value must be alphanumeric and – cannot be used.
>>>
>>>The value should reflect the application whose installer is being defined.
>>
>>**name**
>>>The name of the application being installed.
>>
>>**version**
>>>The version of the application being installed.
>>>
>>>This is a string like X.Y.Z, where each component is an integer.
>>
>>**manufacturer**
>>>The author of the application.
>>
>>**arch**
>>>The WiX architecture of the installer being built.
>
>**add_condition**(*condition: str*, *message: str*)
>
>>Defines a <bal:Condition> that must be satisfied to run this installer.
>>
>>See the WiX Toolkit documentation for more.
>>
>>This method accepts the following arguments:
>>
>>**condition**
>>>The condition expression that must be satisfied.
>>
>>**message**
>>>The message that will be displayed if the condition is not met.
>
>**add_vc_redistributable**(*platform: str*)
>
>>This method registers the Visual C++ Redistributable to be installed.
>>
>>This method accepts the following arguments:
>>
>>**platform**
>>>The architecture to install for. Valid values are x86, x64, and arm64.
>>
>>The bundle can contain Visual C++ Redistributables for multiple runtime architectures. The bundle installer will only install the Redistributable when running on a machine of that architecture. This allows a single bundle installer to target multiple architectures.
>
>**add_wix_msi_builder**(*builder: WiXMSIBuilder*, *display_internal_ui: Optional[bool] = False*, *install_condition: Optional[str] = None*)
>
>>This method adds a *WiXMSIBuilder* to be installed by the produced installer.
>>
>>This method accepts the following arguments:

builder
> The *WiXMSIBuilder* representing an MSI to install.

display_internal_ui
> Whether to display the UI of the MSI.

install_condition
> An expression that must be true for this MSI to be installed.

This method effectively coerces the *WiXMSIBuilder* instance to an <MsiPackage> element and adds it to the <Chain> in the bundle XML. See the WiX Toolset documentation for more.

build(*target: str*) → *ResolvedTarget*

> This method will build an exe using the WiX Toolset.

> This method accepts the following arguments:

> **target**
> > The name of the target being built.

> Upon successful generation of an installer, the produced installer will be assessed for code signing with the windows-installer-creation *action*.

to_file_content() → *FileContent*

> Build an exe installer using the WiX Toolset and return a *FileContent* representing the built installer.

> Upon successful generation of an installer, the produced installer will be assessed for code signing with the windows-installer-creation *action*.

write_to_directory(*path: str*) → str

> Build an exe installer using the WiX Toolset and write the built installer to the directory specified, returning the absolute path of the written file.

> Absolute paths are treated as-is. Relative paths are relative to the current build path.

> Upon successful generation of an installer, the produced installer will be assessed for code signing with the windows-installer-creation *action*.

WiXInstaller

class starlark_tugger.**WiXInstaller**

> The WiXInstaller type represents a Windows installer built with the WiX Toolset.

> WiXInstaller instances allow you to collect .wxs files for processing and to turn these into an installer using the light.exe tool in the WiX Toolset.

> Files constituting your application's install layout can be registered via methods like *WiXInstaller.add_install_file()* and *WiXInstaller.add_install_files()*. When the installer is generated, we take the registered *install files* and dynamically produce a .wxs file named *WiXInstaller.install_files_wxs_path* containing <Fragment>, <Directory>, etc entries for the *install files*. This produced .wxs file is automatically built. The root <DirectoryRef> in this autogenerated file refers to a <Directory> in some external .wxs file where the files should be materialized. The exact Id value to use is defined by *WiXInstaller.install_files_root_directory_id*. Its default value is APPLICATIONFOLDER.

> What all this means is that this type takes care of materializing files registered for installation such that WiX can find them and register them for installation. All you have to do is define a .wxs defining an installer and ensure *WiXInstaller.install_files_root_directory_id* points to a valid <Directory Id= value.

arch

    (str)

    The WiX architecture of the installer being built.

    Valid values include `x64`, `x86`, and `arm64`.

    No validation of the value or its appropriateness for the installer's content is performed. So invalid architecture values or values that don't match the content in the installer can result in run-time errors or bad/buggy installers.

install_files_root_directory_id

    (str)

    Defines the Id value for the `<Directory>` where *install files* (files added via methods like *WiXInstaller.add_install_file()*) should be materialized.

install_files_wxs_path

    (str)

    Defines the filename/path that the auto-generated `.wxs` file containing fragments for *install files* should be written to.

__init__(*id: str, filename: str, arch: str = 'x64'*) → *WiXInstaller*

    `WiXInstaller()` is called to construct a new instance. It accepts the following arguments:

    id
        The name of the installer being built.

        This value is used in `Id` attributes in WiX XML files and must conform to limitations imposed by WiX. Notably, this must be alphanumeric and `-` cannot be used.

        This value is also used to derive GUIDs for the installer.

        This value should reflect the name of the entity being installed and should be unique to prevent collisions with other installers.

    filename
        The name of the file that will be built.

        WiX supports generating multiple installer file types depending on the content of the `.wxs` files. You will have to provide a filename that is appropriate for the installer type.

        File extensions of `.msi` and `.exe` are common. If using `add_simple_installer()`, you will want to provide an `.msi` filename.

    arch
        The WiX architecture of the installer being built.

        This effectively sets the default value for the `.arch` attribute.

add_build_files(*manifest: FileManifest*)

    This method registers additional files to make available to the build environment. Files will be materialized next to `.wxs` files that will be processed as part of building the installer.

    Accepted arguments are:

    manifest
        The file manifest defining additional files to install.

add_build_file(*build_path: str, filesystem_path: str, force_read: Optional[bool] = False*)

    This method registers a single additional file to make available to the build environment.

    Accepted arguments are:

build_path
> The relative path to materialize inside the build environment

filesystem_path
> The filesystem path of the file to copy into the build environment.

force_read
> Whether to read the content of this file into memory when this function is called.

add_install_file(*install_path: str*, *filesystem_path: str*, *force_read: Optional[bool] = False*)

> Add a file from the filesystem to be installed by the installer.
>
> This methods accepts the following arguments:
>
> **install_path**
> > The relative path to materialize inside the installation directory.
>
> **filesystem_path**
> > The filesystem path of the file to install.
>
> **force_read**
> > Whether to read the content of this file into memory when this function is called.
>
> As a file is added, it is checked for code signing compatibility with the action `windows-installer-file-added`.

add_install_files(*manifest:* FileManifest)

> Add files defined in a `FileManifest` to be installed by the installer.
>
> This method accepts the following arguments:
>
> **manifest**
> > Defines files to materialize in the installation directory. All these files will be installed by the installer.
>
> As files are added, they are checked for code signing compatibility with the action `windows-installer-file-added`.

add_msi_builder(*builder:* WiXMSIBuilder)

> This method adds a `WiXMSIBuilder` instance to this instance, marking it for processing/building.

add_simple_installer(*id_prefix: str*, *product_name: str*, *product_version: str*, *product_manufacturer: str*, *program_files:* FileManifest)

> This method will populate the installer configuration with a pre-defined and simple/basic configuration suitable for simple applications. This method effectively derives a `.wxs` which will produce an MSI that materializes files in the `Program Files` directory.
>
> Accepted arguments are:
>
> **id_prefix**
> > String prefix for generated WiX identifiers.
>
> **product_name**
> > The name of the installed product. This becomes the value of the `<Product Name="...">` attribute in the generated `.wxs` file.
>
> **product_version**
> > The version string of the installed product. This becomes the value of the `<Product Version="...">` attribute in the generated `.wxs` file.
>
> **product_manufacturer**
> > The author of the product. This becomes the value of the `<Product Manufacturer="...">` attribute in the generated `.wxs` file.

> **program_files**
>> Files to materialize in the `Program Files/<product_name>` directory upon install.

add_wxs_file(*path: str*, *preprocessor_parameters: Optional[dict[str, str]]*)

> Adds an existing `.wxs` file to be processed as part of building this installer.
>
> Accepted arguments are:
>
> **path**
>> The filesystem path to the `.wxs` file to add. The file will be copied into a temporary directory as part of building the installer and the destination filename will be the same as the file's name.
>
> **preprocessor_parameters**
>> Preprocessor parameters to define when invoking `candle.exe` for this `.wxs` file. These effectively constitute `-p` arguments to `candle.exe`.

set_variable(*key: str*, *value: Optional[str]*)

> Defines a variable to be passed to `light.exe` as `-d` arguments.
>
> Accepted arguments are:
>
> **key**
>> The name of the variable.
>
> **value**
>> The value of the variable. If `None` is used, the variable has no value and is simply defined.

build(*target: str*) → *ResolvedTarget*

> This method will build the installer using the WiX Toolset.
>
> This method accepts the following arguments:
>
> **target**
>> The name of the target being built.
>
> Upon successful generation of an installer, the produced installer will be assessed for code signing with the `windows-installer-creation` *action*.

to_file_content() → *FileContent*

> This method will build the installer using the WiX Toolset and convert the built installer into a *FileContent* instance representing the built installer.
>
> Upon successful generation of an installer, the produced installer will be assessed for code signing with the `windows-installer-creation` *action*.

write_to_directory(*path: str*) → str

> Builds the installer using the WiX Toolset and writes the installer file to the directory specified, returning the absolute path to that installer.
>
> If the path is absolute, it is treated as-is. If it is relative, it is relative to the current build path.
>
> Upon successful generation of an installer, the produced installer will be assessed for code signing with the `windows-installer-creation` *action*.

WiXMSIBuilder

class starlark_tugger.**WiXMSIBuilder**

>The WiXMSIBuilder type allows building simple MSI installers using the [WiX Toolset](#).

>WiXMSIBuilder instances allow you to create and build a .wxs file with common features. A goal of this type is to allow simple applications - without complex installer needs - to generate MSI installers without having to author your own .wxs files.

>Instances have multiple attributes, which are write-only.

>**__init__**(*id_prefix: str, product_name: str, product_version: str, product_manufacturer: str, arch: str = 'x64'*) → *WiXMSIBuilder*

>>WiXMSIBuilder() is called to construct new instances. It accepts the following arguments:

>>**id_prefix**
>>>The string prefix to add to auto-generated IDs in the .wxs XML.

>>>The value must be alphanumeric and – cannot be used.

>>>The value should reflect the application whose installer is being defined.

>>**product_name**
>>>The name of the application being installed.

>>**product_version**
>>>The version of the application being installed.

>>>This is a string like X.Y.Z, where each component is an integer.

>>**product_manufacturer**
>>>The author of the application.

>>**arch**
>>>The WiX architecture of the installer.

>**arch**

>>(str)

>>The WiX architecture of the installer.

>>No validation is performed that the value is a valid WiX architecture or that the content of the installer matches the provided architecture.

>**banner_bmp_path**

>>(str)

>>The path to a 493 x 58 pixel BMP file providing the banner to display in the installer.

>**dialog_bmp_path**

>>(str)

>>The path to a 493 x 312 pixel BMP file providing an image to be displayed in the installer.

>**eula_rtf_path**

>>(str)

>>The path to a RTF file containing the EULA that will be shown to users during installation.

>**help_url**

>>(str)

>>A URL that will be presented to provide users with help.

license_path

>   (str)

>   Path to a file containing the license for the application being installed.

msi_filename

>   (str)

>   The filename to use for the built MSI.

>   If not set, the default is <product_name>-<product_version>.msi.

package_description

>   (str)

>   A description of the application being installed.

package_keywords

>   (str)

>   Keywords for the application being installed.

product_icon_path

>   (str)

>   Path to a file providing the icon for the installed application.

upgrade_code

>   (str)

>   A GUID defining the upgrade code for the application.

>   If not provided, a stable GUID derived from the application name will be derived automatically.

add_program_files_manifest(*manifest:* FileManifest)

>   This method registers the content of a `FileManifest` to be installed in the *Program Files* directory for this application.

>   This method accepts the following arguments:

>   **manifest**
>   >   Files to register for installation.

>   As files are added, they are checked for code signing compatibility with the action `windows-installer-file-added`.

add_visual_cpp_redistributable(*redist_version: str*, *platform: str*)

>   This method will locate and add the Visual C++ Redistributable runtime DLL files (e.g. `vcruntime140.dll`) to the *Program Files* manifest in the builder, effectively materializing these files in the installed file layout.

>   This method accepts the following arguments:

>   **redist_version**
>   >   The version of the Visual C++ Redistributable to search for and add. 14 is the version used for Visual Studio 2015, 2017, and 2019.

>   **platform**
>   >   Identifies the Windows run-time architecture. Must be one of the values `x86`, `x64`, or `arm64`.

>   This method uses `vswhere.exe` to locate the `vcruntimeXXX.dll` files inside a Visual Studio installation. This should *just work* if a modern version of Visual Studio is installed. However, it may fail due to system variance.

build(*target: str*) → *ResolvedTarget*

> This method will build an MSI using the WiX Toolset.
>
> This method accepts the following arguments:
>
> **target**
> > The name of the target being built.
>
> Upon successful generation of an installer, the produced installer will be assessed for code signing with the `windows-installer-creation` *action*.

to_file_content() → *FileContent*

> Builds the MSI using the WiX Toolset and returns a `FileContent` representing the built MSI.
>
> Upon successful generation of an installer, the produced installer will be assessed for code signing with the `windows-installer-creation` *action*.

write_to_directory(*path: str*) → str

> Builds the MSI using the WiX Toolset and writes that installer to the specified directory, returning the absolute path of the written file.
>
> Absolute paths are treated as-is. Relative paths are relative to the current build path.
>
> Upon successful generation of an installer, the produced installer will be assessed for code signing with the `windows-installer-creation` *action*.

Working with Files

Tugger's Starlark dialect exposes various types and functions for working with files. The most important primitives are:

FileContent
> Represents an individual file - it's filename, content, and an executable bit.

FileManifest
> Represents a collection of files. This is a glorified mapping from an install path to *FileContent*.

glob()
> Read files from the filesystem by performing a *glob* filename pattern search.

If a primitive in Tugger is tracking a logical collection of files (e.g. a *WiXInstaller* tracking files that an installer should materialize), chances are that it is using a *FileManifest* for doing so.

Copying Files

Say you want to collect and then materialize a collection of files. Here's how you would do that in Starlark.

```
# Create a new empty file manifest.
m = FileManifest()

# Add individual files to the manifest.
m.add_file(FileContent(path = "file0.txt"))
m.add_file(FileContent(path = "file1.txt"))

# Then copy/materialize them somewhere.
m.install("output/directory")
```

If you wanted, you could even rename files as part of this:

```
m = FileManifest()

f = FileContent(path = "file0.txt")
f.filename = "renamed.txt"
m.add_file(f)
```

Or more concisely:

```
m = FileManifest()
m.add_file(f, path="renamed.txt")
```

Code Signing

Tugger has support for automatically performing code signing when evaluating Starlark configuration files.

Various platforms and distribution channels enforce requirements that binaries and other artifacts are cryptographically signed by a trusted certificate.

For example:

- On Windows, executables and installers must be signed by a trusted certificate to avoid warnings about running untrusted applications.

- On macOS, executables, pkg installers, and more need to be signed by a trusted certificate or Gatekeeper (read: the OS) may refuse to run them.

Tugger's support for automatic signing enables you to meet these requirements with hpoefully minimal effort.

Code Signing Support

Tugger supports signing the following signable entities:

- PE binaries. This is the file executable format in use on Windows platforms.

- MSI installers. This is a common file-based installer format on Windows.

- Mach-O binaries. This is the file executable format in use on Apple platforms.

- Apple application bundles. e.g. `My Program.app` directories. Bundles are a common application *packaging* format on Apple platforms.

Signing on Windows currently uses Microsoft's `signtool.exe` to perform the signing. So signing Windows entities requires access to this tool. (We have plans to implement equivalent functionality in Rust to avoid this dependency.)

Signing Apple formats uses a pure Rust implementation of the code signing functionality and works on any machine. Apple's `codesign` tool or access to Apple hardware is not required to sign Apple entities.

Code signing requires the use of a *code signing certificate*. See *Understanding Code Signing Certificates* for more.

Tugger supports using *code signing certificates* in the following locations:

- From a PFX / PKCS #12 file. (e.g. `.pfx` or `.p12` files.)

- Certificates available in the *Windows certificate store*. Via the *Windows certificate store*, certificates stored in hardware devices (such as HSMs and hardware tokens such as YubiKeys) can also be used.

Configuring Code Signing in Starlark

Code signing needs to be explicitly enabled and configured in your Starlark configuration file.

From a high level, here's how it works:

1. Your Starlark configuration instantiates, configures, and enables a *CodeSigner*, which is the entity that performs code signing.

2. As your configuration file is evaluated, actions that produce or encounter signable entities (such as creating Windows MSI installers) interact with registered *CodeSigner* instances and attempt code signing.

Tugger abstracts away a lot of the complexity around code signing, such as figuring out which files need to be signed (it looks at the content of files and determines if a file is signable). So in many cases, all you need to do is tell Tugger where your code signing certificate is and it can do the rest!

Continuing reading for details on how to customize code signing. Or just straight into *Code Signing Examples*.

Instantiating `CodeSigner` to Perform Code Signing

To perform code signing, first instantiate a *CodeSigner* via one of its available constructor functions:

- *code_signer_from_pfx_file()*
- *code_signer_from_windows_store_sha1_thumbprint()*
- *code_signer_from_windows_store_subject()*
- *code_signer_from_windows_store_auto()*

code_signer_from_pfx_file() is the most versatile method, as it gives Tugger full access to the signing certificate and private key. However, this method is arguably the least secure, as it requires the private key to exist in a file and Tugger holds the decrypted private key in memory during signing. Both of these make the private key much more susceptible to being accessed by unwanted parties. If you are paranoid about security, you should only use this method on machines that you trust.

The `code_signer_from_windows_` functions reference code signing keys stored in the Windows certificate store. Signature requests are processed through the Windows APIs and the private key never leaves the control of the Windows certificate store, helping to keep the private key secure.

Important: Constructed *CodeSigner* instances must be *activated* in order to automatically perform code signing. See *Activating Automatic Code Signing* for more.

Configuring `CodeSigner` Instances

Once you've obtained a *CodeSigner*, you may need to register additional settings to influence signing.

Registering the Issuing Certificate Chain

Produced signatures should often contain details about the *chain* of certificates that issued the code signing certificate. See *Understanding Code Signing Certificates* for more on this topic.

You may need to tell `CodeSigner` about the existence of these certificates.

- When using a code signing certificate backed by the Windows certificate store, you do not need to register the certificate's signing chain.

- When using a code signing certificate backed by a PFX file, you need to register the certificate chain, even if those X.509 certificates are in the PFX file (we don't yet support reading these from the PFX file). `CodeSigner.chain_issuer_certificates_pem_file()` is the most versatile method to register issuer certificates, as it works on all platforms and PEM is a very widespread format for storing X.509 certificates.

- On macOS, `CodeSigner.chain_issuer_certificates_macos_keychain()` can be called to attempt to resolve the certificate chain by speaking directly to the macOS keychain APIs. This requires that the signing certificate be accessible in the current user's keychain and its entire issuing chain to be present in that keychain.

Influencing Signing Operations

`CodeSigner` instances have the opportunity to influence individual signing operations. This gives you significant control over how signing is performed.

`CodeSigner.set_signing_callback()` registers a function that will be invoked on each attempted signing operation. This callback function receives an argument - a `CodeSigningRequest` instance - that describes the entity capable of being signed. This type exposes functionality for influencing the signing operation. For example:

- Setting `CodeSigningRequest.defer` to `True` will opt this `CodeSigner` out of signing this particular entity.

- Setting `CodeSigningRequest.prevent_signing` to `True` will prevent this and other `CodeSigner` from signing this entity.

See the `CodeSigningRequest` API documentation for all available functionality on this type.

Leveraging custom callback functions enables configuration files to employ arbitrarily complex logic for influencing code signing. Your main constraint are the settings exposed on `CodeSigningRequest`. If you find yourself needing a setting that doesn't exist, please file a feature request!

Activating Automatic Code Signing

A `CodeSigner` needs to be *activated* for automatic use by Tugger. i.e. your signable files won't be signed as your Starlark configuration file is evaluated unless a `CodeSigner` is *activated*.

To activate your `CodeSigner`, simply call `CodeSigner.activate()`.

Code Signing Actions

Various activities within the evaluation of your Starlark configuration file trigger the assessment of - and possible performing of - code signing.

Each unique activity has its own string *action* name describing it. This name is accessible via `CodeSigningRequest.action`, enabling callback functions to key off of it. For example, you may want to not sign during certain operations.

The following named actions are defined by Tugger:

file-manifest-install

> Used when a *FileManifest* is materialized on the filesystem through an action like *FileManifest.install()*.

macos-application-bundle-creation

> When a macOS Application Bundle is created by Tugger.
>
> This will be triggered by *MacOsApplicationBundleBuilder.build()*.

windows-installer-creation

> When a Windows installer file is created by Tugger.
>
> Methods like *WiXMSIBuilder.build()* and *WiXBundleBuilder.build()* will trigger this action.

windows-installer-file-added

> When a file that will be installed is added to a Windows installer.
>
> Triggered by *WiXMSIBuilder.add_program_files_manifest()*, *WiXInstaller.add_install_file()*, and *WiXInstaller.add_install_files()*.

Other applications extending Tugger's core functionality may define their own actions.

Duplicate Events

It is possible for the same logical file to trigger multiple signing events as it is processed. For example, *MacOsApplicationBundleBuilder.build()* may trigger an event for macOS Application Bundle generation then a later action loads the bundle files into a *FileManifest* and materializes them somewhere else via *FileManifest.install()*, which would trigger an additional signability check.

As a result, the same file or entity may be signed multiple times.

If this behavior is undesirable, the use of a custom callback function can be used to choose which signing requests to respond to.

Unfortunately, we do not yet expose metadata on *CodeSigningRequest* indicating if a file is signed or not. This would likely be the obvious attribute to filter against. This feature is tracked at https://github.com/indygreg/PyOxidizer/issues/400.

Code Signing Examples

Automatically Sign all Signable Content with a Specific Certificate in the Windows Store

Say you have a code signing certificate in the Windows certificate store with the SHA-1 thumbprint `deadbeefdeadbeefdeadbeefdeadbeefdeadbeef` and you want Tugger to sign all signable files as it runs. Here's what you'll need to do in your Starlark configuration file:

```
signer = code_signer_from_windows_store_sha1_thumbprint(
    "deadbeefdeadbeefdeadbeefdeadbeefdeadbeef")
signer.activate()
```

As Tugger encounters `.exe`, `.dll`, `.msi` files and any file that it identifies as signable, it will attempt to automatically sign them!

Choosing a Code Signing Certificate Dynamically

Say you have multiple code signing certificates but want to parameterize which one to use. We can do that through the use of the `VARS` global dict, which holds settings passed in via the command line.

```python
PFX_PATH = VARS.get("PFX_PATH")
PFX_PASSWORD = VARS.get("PFX_PASSWORD", "")

# This needs to be in its own function because Starlark doesn't allow `if`
# at the file/module scope.
def make_code_signers():
    if PFX_PATH:
        signer = code_signer_from_pfx_file(PFX_PATH, PFX_PASSWORD)
        signer.activate()


# Don't forget to call the function!
make_code_signers()
```

Then when running the configuration file, specify an extra variable. e.g.:

```
$ pyoxidizer --var PFX_PATH /path/to/certificate.pfx --var PFX_PASSWORD hunter2
```

Or you could use functions like *prompt_confirm()*, *prompt_input()*, and *prompt_password()* to ask the user which certificate to use.

```python
def make_code_signers():
    if prompt_confirm("enable code signing?", default=False):
        pfx_path = prompt_input("enter path to PFX file:")
        pfx_password = prompt_password("enter path to PFX password:", confirm=True)

        signer = code_signer_from_pfx_file(pfx_path, pfx_password)
        signer.activate()

make_code_signers()
```

Selectively Ignoring Files to Sign

It is common to want to ignore certain files from signing. For example, you may ship a pre-built binary that already has a valid code signature. Here's how you can do that.

```python
# Define a function that will be called for every signing request that
# can influence operation.
def code_signer_callback(request):
    # Match a known filename that doesn't need signed and set
    # `prevent_signing = True` to prevent it from being signed.
    if request.filename == "vcruntime140.dll":
        request.prevent_signing = True


signer = code_signer_from_windows_store_sha1_thumbprint(
    "deadbeefdeadbeefdeadbeefdeadbeefdeadbeef")
```

```
signer.set_signing_callback(code_signer_callback)
signer.activate()
```

You could even use the *prompt_confirm()* function to prompt whether to sign each file:

```python
def code_signer_callback(request):
    request.prevent_signing = not prompt_confirm("sign %s?" % request.filename)

signer = code_signer_from_...()
signer.set_signing_callback(code_signer_callback)
signer.activate()
```

Understanding Code Signing Certificates

A *code signing certificate* consists of a secure, private *key* and a public *certificate* that describes itself to others. These components are strictly separate but are often represented and stored together.

The public certificate is an X.509 certificate, much like those used in HTTP to identify web sites. The main difference is that the certificate's subject describes a person or organization (instead of a website) and the certificate contains attributes that denote it for use by code signing.

Like web site X.509 certificates, code signing certificates are *signed* by another X.509 certificate. This is called the *issuing* certificate. There is often a *chain* of certificates - the *certificate chain* - leading to a *self-signed* certificate (a certificate whose issuer was itself), which is referred to as the *root* certificate.

Typically, the *certificate chain* is included in code signatures. This enables readers of the signature to have full access to all relevant certificates, without an implicit dependency on them being present on the reading machine. This enables validation to be conducted more robustly.

Code Signing Certificate Storage

Code signing certificates can be stored in a number of formats. Here are the popular ones:

- As standalone `.pfx` or `.p12` files. These are files containing data as defined by the PFX and PKCS #12 specifications. Most tools that support saving code signing certificates to files support this format if not use it by default.

- In your operating system's certificate store. Windows, macOS, and other operating systems have built-in functionality for storing and accessing certificates. On Windows, the `certmgr.msc` tool can be used to view certificates. On macOS, `Keychain Access` is the official GUI application.

In addition, the public X.509 certificates and the certificates in the *certificate chain* are often represented as PEM. This is a human-readable text format with content like `-----BEGIN CERTIFICATE-----`. PEM is actually base64 encoded BER/DER encoding of ASN.1 data structures, but that's not important. What is important is public certificates are often stored in files having this `-----BEGIN CERTIFICATE-----` content. These files often have the extension `.pem` or `.crt`.

The *certificate chain* is constant for the lifetime of a code signing certificate. So it is possible to export these certificates to a persisted file and reference this file when you need to access the issuer certificates chain.

Securing Your Code Signing Certificate

Your code signing certificate's private key attests that its owner was in possession of that certificate and has vouched for the integrity of whatever it signed.

Important: Code signing certificates can be very attractive theft targets for hackers, as possession of a code signing certificate enables you to sign software that can run on other machines and appears to be trusted. Therefore, it is often important to try to secure your code signing certificates!

The most secure way to store code signing certificates is in dedicated hardware devices, such as HSMs or personal hardware tokens (such as YubiKeys). Often, the private key component of the certificate is generated directly in said hardware and it is impossible to export the private key and obtain its raw value. Instead, operations like signing are issued to the hardware and the hardware gives you the rest.

Tugger doesn't yet support interfacing directly with hardware devices. However, we do have support for interfacing with the operating system's certificate stores:

- On Windows, a certificate in the Windows certificate store can be referenced by its SHA-1 fingerprint. (This is the preferred mechanism to reference a certificate on Windows.)

- On Windows, a certificate in the Windows certificate store can be referenced by specifying a string to match against in the certificate's *subject* field. (This is less precise than specifying a certificate's SHA-1 fingerprint.)

- On Windows, you can tell the signing tool to automatically find the most appropriate certificate to use. It will look for a certificate in known certificate stores. (This is the least precise of all options available on Windows.)

Note: Your operating system's certificate store can often interface with hardware devices holding code signing certificates. So Tugger's support for interfacing with the operating system store is often just as effective as interfacing directly with hardware devices.

For example, on Windows, certificates stored in a YubiKey will be available if you have the YubiKey Smart Card Minidriver installed.

If Tugger doesn't support using a remote certificate, you will need to export a certificate to a file and have Tugger use that. If you export your certificate to a file, you should take care to secure that file as best you can.

File-based code signing certificates often exist in `.pfx` or `.p12` files. These are often protected with a password. **You should use a strong and unique password to secure this file.**

Important: If someone else gains access to the file containing your code signing certificate, they will be able to perform an offline attack using as many compute resources as possible to guess your password and gain access to the code signing certificate.

You should take the following precautions to protect file-based code signing certificates:

- Choose a strong, unique password for protecting the file content.

- Limit the time the files exist. If you can create the file only when needed, this is better than having the file linger on the filesystem.

- Limit the number of copies of the file. Every copy of the file is an opportunity for the file to be obtained by someone else.

Exporting a Code Signing Certificate from macOS Keychain

Apple platforms require a code signing certificate issued by Apple to sign distributed files.

If you have an Apple-issued code signing certificate, it is likely registered in a *keychain* on your machine. Tugger doesn't currently support interfacing directly with the macOS keychain and you will need to export your signing certificate to a PFX / `.p12` file so Tugger can use it. Here's how to do that.

1. Press `command + spacebar` and search for and open the `Keychain Access` application.

2. Make sure the correct keychain is selected. The keychain code signing certificates are typically located in is the `login` keychain under the `Default Keychains` list.

3. From the horizontal list of filters above the main pane, select `Certificates` (it is probably the last item).

4. Find the certificate you want to export. It likely has a name like `Developer ID Application:  <your name (some ID)>`

5. Do a double finger tap, right click, or `File -> Export Items ...` to bring up the export dialog.

6. For the file format, make sure `Personal Information Exchange (.p12)` is selected.

7. Navigate to a folder where you want to save the file, choose an appropriate name, and click `Save`.

8. You will be asked for a *password which will be used to protect the exported items*. Enter one. This password will need to be provided to Tugger later to unlock the content in the file.

9. You may be prompted to enter the password to the keychain to allow the key export. If so, enter that password.

10. You may be prompted multiple times. Just keep entering your keychain password(s) until it is done.

11. You are done! There should be a `.p12` file wherever you told `Keychain Access` to save it.

Important: Please see *Securing Your Code Signing Certificate* for important information on keeping your file-based code signing certificate secure.

Finding the Code Signing SHA-1 Thumbprint on Windows

On Windows, it is recommended to use code signing certificates in the Windows certificate store and to specify those certificates via their SHA-1 thumbprint, which should uniquely identify a certificate.

The Windows certificate store supports interfacing with hardware certificate stores (such as YubiKeys and other hardware devices). So this method should work with connected hardware certificate stores as well.

1. Press `Windows Key + r` to open the `Run` panel. Type in `certmgr.msc` and run that program.

2. Code signing certificates are likely under `Personal -> Certificates`. Find that item in the tree and look for a certificate in the main pane.

3. Find the certificate you want to use and double click on it to view its details.

4. Open the `Details` tab.

5. In the table of fields, find and select `Thumbprint`.

6. Copy the 40 character hexadecimal value that is printed.

The SHA-1 thumbprint can be fed into `code_signer_from_windows_store_sha1_thumbprint()` to construct a `CodeSigner` that uses the specified certificate.

If the certificate is protected by a password or requires key to unlock, you should see prompts to do that as Tugger attempts to sign things.

Exporting a Code Signing Certificate from Windows Certificate Store

Code signing certificates on Windows are often stored in the Windows certificate store.

Important: Tugger has support for using certificates directly in the Windows certificate store. Exporting certificates to files will likely result in a net loss of security.

Here is how you can export a certificate to a PFX file.

1. Press `Windows Key + r` to open the `Run` panel. Type in `certmgr.msc` and run that program.

2. Code signing certificates are likely under `Personal -> Certificates`. Find that item in the tree and look for a certificate in the main pane.

3. Double click on the certificate you want to export, open its `Details` table, and click the `Copy to File...` button. This should open the *Certificate Export Wizard*.

4. Click `Next`.

5. Make sure `Yes, export the private key` is selected and click `Next`.

6. For the format, make sure the selected value is `Personal Information Exchange PKCS #12 (PFX)`. For the checkboxes, check `Include all certificates in the certificate path, if possible`. Then click `Next`.

7. You should be prompted for a password. Enter a secure, unique password. In the `Encryption` drop-down, ensure `TripleDES-SHA1` is selected (we don't yet support `AES256-SHA256`). Then click `Next`.

8. Select a filename and click `Next`.

9. Click `Finish` to close the wizard.

Important: Please see *Securing Your Code Signing Certificate* for important information on keeping your file-based code signing certificate secure.

Using the WiX Toolset to Produce Windows Installers

The WiX Toolset is an open source collection of tools used for building Windows installers (`.msi` files, `.exe`, etc). The WiX Toolset is incredibly powerful and enables building anything from simple to complex installers.

Tugger defines interfaces to the WiX Toolset via Rust APIs and exposes much of this functionality to Starlark.

Concepts

With the WiX Toolset, you define your installer through `.wxs` XML files. You use the `candle.exe` program to *compile* these files into `.wixobj` files. These *compiled* files are then *linked* together using `light.exe` to produce an installer (`.msi`, `.exe`, etc).

The goal of Tugger's Rust API is to expose the low-level control over WiX Toolset that the most demanding applications will need while also providing high-level and simpler interfaces for performing common tasks (such as producing a simple `.msi` installer that simply materializes files into the `Program Files` directory).

Tugger's WiX APIs

Tugger implements various interfaces for interacting with WiX. This section attempts to document them at a high level and talks about when to use which.

WxsBuilder

> The `WxsBuilder` Rust struct is used to build a single `.wxs` file. You provide the path of the `.wxs` and build settings and it knows how to invoke `candle.exe` for this file.

WiXInstallerBuilder

> The `WiXInstallerBuilder` Rust struct and *WiXInstaller* Starlark type are used to manage the end-to-end building and linking of `.wxs` files. This type knows how to register multiple `WxsBuilder` instances and build them as a collection. This type holds all the logic for invoking `candle.exe` and `light.exe`.

WiXSimpleMSIBuilder

> The `WiXSimpleMSIBuilder` Rust struct and *WiXMSIBuilder* Starlark type provide a high-level interface for generating an MSI based installer with common features. It enables you to generate a `.wxs` file by providing a few parameters, without having to know WiX XML.
>
> A `WiXSimpleMSIBuilder` ultimately is converted to a `WiXInstallerBuilder`.

WiXBundleInstallerBuilder

> The `WiXBundleInstallerBuilder` Rust struct and *WiXBundleBuilder* Starlark type provide a high-level interface for generating an `.exe` based installed with common features.
>
> A `WiXBundleInstallerBuilder` ultimately is converted to a `WiXInstallerBuilder`.

If your application only needs the limited functionality exposed by the high-level `WiXSimpleMSIBuilder` and `WiXBundleInstallerBuilder` interfaces, you are encouraged to use these for building your installer, as you won't need to concern yourself with the low-level WiX XML details.

If your application needs what you think is simple or common functionality not provided by the aforementioned high-level builders, consider filing a feature request to request the missing functionality.

Complex applications that have outgrown the limited capabilities of the high-level *builder* interfaces will need to use the lower level `WiXInstallerBuilder` / *WiXInstaller* interface. This interface allows you to provide your own `.wxs` files. This means you can still use Tugger for invoking WiX, even if all of your `.wxs` files are maintained outside of Tugger, enabling Tugger to grow with your needs. Note that it is possible to use one of the higher-level interfaces for automatically generating a `.wxs` file and then supplement this automatically-generated file with other `.wxs` files that you maintain.

Note: Ideally no WiX installer should be too complicated to be handled by Tugger. If Tugger's functionality is not sufficient, consider creating an issue to request a feature to close the feature gap.

How Tugger Invokes WiX

Tugger's Rust APIs collects which `.wxs` files to compile and their compilation settings. It also collects additional files needed to compile `.wxs` files.

When you *build* your installer, Tugger copies all the registered `.wxs` files plus other registered files into a common directory. It then invokes `candle.exe` on each `.wxs` file followed by `light.exe` to link them together. This is different from a traditional environment, where `.wxs` files are often processed in place: Tugger always makes copies to try to ensure results are reproducible and the full build environment is captured.

Automatic <Fragment> Generation for Files

Tugger supports automatically generating a `.wxs` file with <Fragment>'s describing a set of files. Given a set of input files, it will produce a deterministic `.wxs` file with <DirectoryRef> holding <Component> and <File> of every file therein as well as <ComponentGroup> for each distinct directory tree.

This functionality is similar to what WiX Toolset's `heat.exe` tool can do. However, Tugger uses a deterministic mechanism to derive GUIDs and IDs for each item. This enables the produced elements to be referenced in other `.wxs` files more easily. And the generated file doesn't need to be saved or manually updated, as it does with the use of `heat.exe`.

You simply give Tugger a manifest of files to index and the prefix for `Id` attributes in XML, and it will emit a deterministic `.wxs` file!

Project History

Version History

0.4.0

Not yet released.

0.3.0

Released March 4, 2021.

New Features

- The `FileManifest` Starlark type now exposes an `add_path()` method.
- The Starlark dialect now exposes `SnapApp`, `Snappart`, and `Snap` types representing Snapcraft configuration files.
- The Starlark dialect now has a `SnapcraftBuilder` type that serves as an interface to invoking `snapcraft`.
- The Starlark dialect now exposes `WiXBundleBuilder`, `WiXInstaller`, and `WiXMSIBuilder` types for defining Windows installers using the WiX Toolset.

0.2.0

Version 0.2 was released November 8, 2020.

Version 0.2 marked the beginning of a complete rewrite of Tugger. The canonical source code repository was moved to the PyOxidizer repository.

Not all features from version 0.1 were ported to version 0.2.

0.1.0

Version 0.1 was released on August 25, 2019.

Version 0.1 was mostly a proof of concept to demonstrate the viability of Starlark configuration files. But Tugger was usable in this release.

Symbols

A

X