shiv

Apr 27, 2021

Contents

Shiv is a command line utility for building fully self-contained Python zipapps as outlined in PEP 441 but with all their dependencies included!

Shiv's primary goal is making distributing Python applications fast & easy.

CHAPTER 1

How it works

Shiv includes two major components: a *builder* and a small *bootstrap* runtime.

1.1 Building

In order to build self-contained single-artifact executables, shiv leverages `pip` to stage your project's dependencies and then shiv uses the features described in PEP 441 to create a "zipapp".

The feature of PEP 441 we are using is Python's ability to implicitly execute a *__main__.py* file inside of a zip archive. Shiv packs your dependencies into a zip and then adds a special *__main__.py* file that instructs the Python interpreter to unpack those dependencies to a known location, add them to your interpreter's search path, and that's it!

Note: "Conventional" zipapps don't include any dependencies, which is what sets shiv apart from the stdlib zipapp module.

shiv accepts only a few command line parameters of its own, described here, and any unprocessed parameters are delegated to `pip install`. This allows users to fully leverage all the functionality that pip provides.

For example, if you wanted to create an executable for `flake8`, you'd specify the required dependencies (in this case, simply `flake8`), the callable (either via `-e` for a setuptools-style entry point or `-c` for a bare console_script name), and the output file:

```
$ shiv -c flake8 -o ~/bin/flake8 flake8
```

This creates an executable (`~/bin/flake8`) containing all the dependencies required by `flake8` that invokes the console_script `flake8` when executed!

You can optionally omit the entry point specification, which will drop you into an interpreter that is bootstrapped with the dependencies you specify. This can be useful for creating a single-artifact executable Python environment:

```
$ shiv httpx -o httpx.pyz --quiet
$ ./httpx.pyz
Python 3.7.7 (default, Mar 10 2020, 16:11:21)
[Clang 11.0.0 (clang-1100.0.33.12)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>> import httpx
>>> httpx.get("https://shiv.readthedocs.io")
<Response [200 OK]>
```

This is particularly useful for running scripts without needing to contaminate your Python environment, since the `pyz` files can be used as a shebang!

```
$ cat << EOF > tryme.py
> #!/usr/bin/env httpx.pyz
>
> import httpx
> url = "https://shiv.readthedocs.io"
> response = httpx.get(url)
> print(f"Got {response.status_code} from {url}!")
>
> EOF
$ chmod +x tryme.py
$ ./tryme.py
Got 200 from https://shiv.readthedocs.io!
```

1.2 Bootstrapping

As mentioned above, when you run an executable created with shiv, a special bootstrap function is called. This function unpacks dependencies into a uniquely named subdirectory of `~/.shiv` and then runs your entry point (or interactive interpreter) with those dependencies added to your interpreter's search path (`sys.path`). To improve performance, once the dependencies have been extracted to disk, any further invocations will re-use the 'cached' site-packages unless they are deleted or moved.

Note: Dependencies are extracted (rather than loaded into memory from the zipapp itself) because of limitations of binary dependencies. Just as an example, shared objects loaded via the dlopen syscall require a regular filesystem. Many libraries also expect a filesystem in order to do things like building paths via `__file__` (which doesn't work when a module is imported from a zip), etc. To learn more, check out this resource about the setuptools "zip_safe" flag.

CHAPTER 2

Influencing Runtime

Whenever you are creating a zipapp with `shiv`, you can specify a few flags that influence the runtime. For example, the `-c/--console-script` and `-e/--entry-point` options already mentioned in this doc. To see the full list of command line options, see this page.

In addition to options that are settable during zipapp creation, there are a number of environment variables you can specify to influence a zipapp created with shiv at run time.

2.1 SHIV_ROOT

This should be populated with a full path, it overrides `~/.shiv` as the default base dir for shiv's extraction cache.

This is useful if you want to collect the contents of a zipapp to inspect them, or if you want to make a quick edit to a source file, but don't want to taint the extraction cache.

2.2 SHIV_INTERPRETER

This is a boolean that bypasses and console_script or entry point baked into your pyz. Useful for dropping into an interactive session in the environment of a built cli utility.

2.3 SHIV_ENTRY_POINT

Note: Same functionality as `-e/--entry-point` at build time

This should be populated with a setuptools-style callable, e.g. "module.main:main". This will execute the pyz with whatever callable entry point you supply. Useful for sharing a single pyz across many callable 'scripts'.

2.4 SHIV_FORCE_EXTRACT

This forces re-extraction of dependencies even if they've already been extracted. If you make hotfixes/modifications to the 'cached' dependencies, this will overwrite them.

2.5 SHIV_EXTEND_PYTHONPATH

Note: Same functionality as `-E/--extend-pythonpath` at build time.

This is a boolean that adds the modules bundled into the zipapp into the `PYTHONPATH` environment variable. It is not needed for most applications, but if an application calls Python as a subprocess, expecting to be able to import the modules bundled in the zipapp, this will allow it to do so successfully.

2.6 Preamble

As an application packager, you may want to run some sanity checks or clean up tasks when users execute a pyz. For such a use case, `shiv` provides a `--preamble` argument. Any executable script provided will be packed into the resulting zipapp and executed during bootstrapping.

If the preamble file is written in Python (e.g. ends in `.py`) then shiv will inject two variables into the runtime that may be useful to preamble authors:

- `archive`: path to the current PYZ file, equivalent to `sys.argv[0]`
- `env`: an instance of the Environment object.
- `site_packages`: a pathlib.Path of the directory where the current PYZ's site_packages were extracted to during bootsrap.

For an example, a preamble file that cleans up prior extracted `~/.shiv` directories might look like:

```
#!/usr/bin/env python3

import shutil

from pathlib import Path

# variable injected from shiv.bootstrap
site_packages: Path

current = site_packages.parent
cache_path = current.parent
name, build_id = current.name.split('_')

if __name__ == "__main__":
    for path in cache_path.iterdir():
        if path.name.startswith(f"{name}_") and not path.name.endswith(build_id):
            shutil.rmtree(path)
```

2.7 Reproducibility

`shiv` supports the ability to create reproducible artifacts. By using the `--reproducible` command line option or by setting the `SOURCE_DATE_EPOCH` environment variable during zipapp creation. When this option is selected, if the inputs do not change, the output should be idempotent.

For more information, see https://reproducible-builds.org/.

CHAPTER 3

Table of Contents

3.1 Complete CLI Reference

This is a full reference of the project's command line tools, with the same information as you get from using the `-h` option. It is generated from source code and thus always up to date.

3.1.1 Available Commands

shiv

Shiv is a command line utility for building fully self-contained Python zipapps as outlined in PEP 441, but with all their dependencies included!

```
shiv [OPTIONS] [PIP_ARGS]...
```

Options

--version
Show the version and exit.

-e, --entry-point `<entry_point>`
The entry point to invoke (takes precedence over –console-script).

-c, --console-script `<console_script>`
The console_script to invoke.

-o, --output-file <output_file>
The path to the output file for shiv to create.

-p, --python <python>
The python interpreter to set as the shebang (such as '/usr/bin/env python3')

--site-packages <site_packages>
The path to an existing site-packages directory to copy into the zipapp.

--compressed, --uncompressed
Whether or not to compress your zip.

--compile-pyc
Whether or not to compile pyc files during initial bootstrap.

-E, --extend-pythonpath
Add the contents of the zipapp to PYTHONPATH (for subprocesses).

--reproducible
Generate a reproducible zipapp by overwriting all files timestamps to a default value. Timestamp can be overwritten by SOURCE_DATE_EPOCH env variable. Note: If SOURCE_DATE_EPOCH is set, this option will be implicitly set to true.

--no-modify
If specified, this modifies the runtime of the zipapp to raise a RuntimeException if the source files (in ~/.shiv or SHIV_ROOT) have been modified. It's recommended to use Python's "–check-hash-based-pycs always" option with this feature.

--preamble <preamble>
Provide a path to a preamble script that is invoked by shiv's runtime after bootstrapping the environment, but before invoking your entry point.

--root <root>
Override the 'root' path (default is ~/.shiv).

Arguments

PIP_ARGS
Optional argument(s)

shiv-info

A simple utility to print debugging information about PYZ files created with `shiv`

```
shiv-info [OPTIONS] PYZ
```

Options

-j, --json
output as plain json

Arguments

PYZ
Required argument

3.1.2 Additional Hints

Choosing a Python Interpreter Path

A good overall interpreter path as passed into `--python` is `/usr/bin/env python3`. If you want to make sure your code runs on the Python version you tested it on, include the minor version (e.g. `... python3.6`) – use what fits your circumstances best.

On Windows, the Python launcher `py` knows how to handle shebangs using `env`, so it's overall the best choice if you target multiple platforms with a pure Python zipapp.

Also note that you can always fix the shebang during installation of a zipapp using this:

```
python3 -m zipapp -p '/usr/bin/env python3.7' -o ~/bin/foo foo.pyz
```

3.2 Motivation & Comparisons

3.2.1 Why?

At LinkedIn we ship hundreds of command line utilities to every machine in our data-centers and all of our employees workstations. The vast majority of these utilties are written in Python. In addition to these utilities we also have many internal libraries that are uprev'd daily.

Because of differences in iteration rate and the inherent problems present when dealing with such a huge dependency graph, we need to package the executables discretely. Initially we took advantage of the great open source tool PEX. PEX elegantly solved the isolated packaging requirement we had by including all of a tool's dependencies inside of a single binary file that we could then distribute!

However, as our tools matured and picked up additional dependencies, we became acutely aware of the performance issues being imposed on us by `pkg_resources`'s Issue 510. Since PEX leans heavily on `pkg_resources` to bootstrap its environment, we found ourselves at an impass: lose out on the ability to neatly package our tools in favor of invocation speed, or impose a few second performance penalty for the benefit of easy packaging.

After spending some time investigating extricating pkg_resources from PEX, we decided to start from a clean slate and thus `shiv` was created.

3.2.2 How?

Shiv exploits the same features of Python as PEX, packing `__main__.py` into a zipfile with a shebang prepended (akin to zipapps, as defined by PEP 441, extracting a dependency directory and injecting said dependencies at runtime. We have to credit the great work by @wickman, @kwlzn, @jsirois and the other PEX contributors for laying the groundwork!

The primary differences between PEX and shiv are:

- `shiv` completely avoids the use of `pkg_resources`. If it is included by a transitive dependency, the performance implications are mitigated by limiting the length of `sys.path`. Internally, at LinkedIn, we always include the -s and -E Python interpreter flags by specifying `--python "/path/to/python -sE"`, which ensures a clean environment.
- Instead of shipping our binary with downloaded wheels inside, we package an entire site-packages directory, as installed by `pip`. We then bootstrap that directory post-extraction via the stdlib's `site.addsitedir` function. That way, everything works out of the box: namespace packages, real filesystem access, etc.

Because we optimize for a shorter `sys.path` and don't include `pkg_resources` in the critical path, executables created with `shiv` can outperform ones created with PEX by almost 2x. In most cases the executables created with `shiv` are even faster than running a script from within a virtualenv!

3.3 Shiv API

3.3.1 cli

`shiv.cli.`**`console_script_exists`**(*site_packages_dirs: List[pathlib.Path], console_script: str*) → bool

Return true if the console script with provided name exists in one of the site-packages directories.

Console script is expected to be in the 'bin' directory of site packages.

Parameters

- **`site_packages_dirs`** – Paths to site-packages directories on disk.
- **`console_script`** – A console script name.

`shiv.cli.`**`copytree`**(*src: pathlib.Path*, *dst: pathlib.Path*) → None

A utility function for syncing directories.

This function is based on shutil.copytree. In Python versions that are older than 3.8, shutil.copytree would raise FileExistsError if the "dst" directory already existed.

`shiv.cli.`**`find_entry_point`**(*site_packages_dirs: List[pathlib.Path], console_script: str*) → str

Find a console_script in a site-packages directory.

Console script metadata is stored in entry_points.txt per setuptools convention. This function searches all entry_points.txt files and returns the import string for a given console_script argument.

Parameters

- **`site_packages_dirs`** – Paths to site-packages directories on disk.
- **`console_script`** – A console_script string.

`shiv.cli.`**`get_interpreter_path`**(*append_version: bool = False*) → str

A function to return the path to the current Python interpreter.

Even when inside a venv, this will return the interpreter the venv was created with.

constants —

This module contains various error messages.

3.3.2 builder

This module is a modified implementation of Python's "zipapp" module.

We've copied a lot of zipapp's code here in order to backport support for compression. https://docs.python.org/3.7/library/zipapp.html#cmdoption-zipapp-c

`shiv.builder.`**`create_archive`**(*sources: List[pathlib.Path], target: pathlib.Path, interpreter: str, main: str, env: shiv.bootstrap.environment.Environment, compressed: bool = True*) → None

Create an application archive from SOURCE.

This function is a heavily modified version of stdlib's zipapp.create_archive

shiv.builder.**write_file_prefix**(*f: IO[Any], interpreter: str*) → None
Write a shebang line.

Parameters

- **f** – An open file handle.
- **interpreter** – A path to a python interpreter.

shiv.builder.**write_to_zipapp**(*archive: zipfile.ZipFile, arcname: str, data: bytes, date_time: Tuple[int, int, int, int, int, int], compression: int, stat: Optional[os.stat_result] = None*) → None
Write a file or a bytestring to a ZipFile as a separate entry and update contents_hash as a side effect.

3.3.3 pip

shiv.pip.**clean_pip_env**() → Generator[[None, None], None]
A context manager for temporarily removing 'PIP_REQUIRE_VIRTUALENV' from the environment.

Since shiv installs via *–target*, we need to ignore venv requirements if they exist.

shiv.pip.**install**(*args: List[str]*) → None
pip install as a function.

Accepts a list of pip arguments.

```
>>> install(['numpy', '--target', 'site-packages'])
Collecting numpy
Downloading numpy-1.13.3-cp35-cp35m-manylinux1_x86_64.whl (16.9MB)
    100% || 16.9MB 53kB/s
Installing collected packages: numpy
Successfully installed numpy-1.13.3
```

3.3.4 bootstrap

shiv.bootstrap.**bootstrap**()
Actually bootstrap our shiv environment.

shiv.bootstrap.**cache_path**(*archive*, *root_dir*, *build_id*)
Returns a ~/.shiv cache directory for unzipping site-packages during bootstrap.

Parameters

- **archive** (`ZipFile`) – The zipfile object we are bootstrapping from.
- **root_dir** (`str`) – Optional, either a path or environment variable pointing to a SHIV_ROOT.
- **build_id** (`str`) – The build id generated at zip creation.

shiv.bootstrap.**current_zipfile**()
A function to vend the current zipfile, if any

shiv.bootstrap.**ensure_no_modify**(*site_packages*, *hashes*)
Compare the sha256 hash of the unpacked source files to the files when they were added to the pyz.

shiv.bootstrap.**extend_python_path**(*environ*, *additional_paths*)
Create or extend a PYTHONPATH variable with the frozen environment we are bootstrapping with.

`shiv.bootstrap.`**`extract_site_packages`**(*archive*, *target_path*, *compile_pyc=False*, *compile_workers=0*, *force=False*)

Extract everything in site-packages to a specified path.

Parameters

- **`archive`** (*`ZipFile`*) – The zipfile object we are bootstrapping from.
- **`target_path`** (*`Path`*) – The path to extract our zip to.
- **`compile_pyc`** (*`bool`*) – A boolean to dictate whether we pre-compile pyc.
- **`compile_workers`** (*`int`*) – An int representing the number of pyc compiler workers.
- **`force`** (*`bool`*) – A boolean to dictate whether or not we force extraction.

`shiv.bootstrap.`**`import_string`**(*import_name*)

Returns a callable for a given setuptools style import string

Parameters **`import_name`** (*`str`*) – A console_scripts style import string

`shiv.bootstrap.`**`run`**(*module*)

Run a module in a scrubbed environment.

If a single pyz has multiple callers, we want to remove these vars as we no longer need them and they can cause subprocesses to fail with a ModuleNotFoundError.

Parameters **`module`** (*`Callable`*) – The entry point to invoke the pyz with.

3.3.5 bootstrap.environment

This module contains the `Environment` object, which combines settings decided at build time with overrides defined at runtime (via environment variables).

3.3.6 bootstrap.interpreter

The code in this module is adapted from https://github.com/pantsbuild/pex/blob/master/pex/pex.py

It is used to enter an interactive interpreter session from an executable created with `shiv`.

3.4 Deploying django apps

Because of how shiv works, you can ship entire django apps with shiv, even including the database if you want!

3.4.1 Defining an entrypoint

First, we will need an entrypoint.

We'll call it `main.py`, and store it at `<project_name>/<project_name>/main.py` (alongside `wsgi.py`)

```
import os
import sys

import django

# setup django
```

(continues on next page)

(continued from previous page)

```
os.environ.setdefault("DJANGO_SETTINGS_MODULE", "<project_name>.settings")
django.setup()

try:
    production = sys.argv[1] == "production"
except IndexError:
    production = False

if production:
    import gunicorn.app.wsgiapp as wsgi

    # This is just a simple way to supply args to gunicorn
    sys.argv = [".", "<project_name>.wsgi", "--bind=0.0.0.0:80"]

    wsgi.run()
else:
    from django.core.management import call_command

    call_command("runserver")
```

This is meant as an example. While it's fully production-ready, you might want to tweak it according to your project's needs.

3.4.2 Build script

Next, we'll create a simple bash script that will build a zipapp for us.

Save it as `build.sh` (next to manage.py)

```
#!/usr/bin/env bash

# clean old build
rm -r dist <project_name>.pyz

# include the dependencies from `pip freeze`
pip install -r <(pip freeze) --target dist/

# or, if you're using pipenv
# pip install -r  <(pipenv lock -r) --target dist/

# specify which files to be included in the build
# You probably want to specify what goes here
cp -r \
-t dist \
<app1> <app2> manage.py db.sqlite3

# finally, build!
shiv --site-packages dist --compressed -p '/usr/bin/env python3' -o <project_name>.
↪pyz -e <project_name>.main
```

And then, you can just do the following

```
$ ./build.sh

$ ./<project_name>.pyz
```

(continues on next page)

(continued from previous page)

```
# In production -

$ ./<project_name>.pyz production
```

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

S

Index

Symbols

B

C

E

F

G

I

P

R

S