Contents

1 How it works 3
  1.1 Building ................................................................. 3
  1.2 Bootstrapping ......................................................... 4

2 Influencing Runtime 5
  2.1 SHIV_ROOT ............................................................... 5
  2.2 SHIV_INTERPRETER ................................................... 5
  2.3 SHIV_ENTRY_POINT .................................................... 5
  2.4 SHIV_FORCE_EXTRACT ................................................ 5
  2.5 SHIV_EXTEND_PYTHONPATH ......................................... 6

3 Table of Contents 7
  3.1 Complete CLI Reference ............................................... 7
  3.2 Motivation & Comparisons .......................................... 9
  3.3 Shiv API ................................................................. 10
  3.4 Deploying django apps ............................................... 12

4 Indices and tables 15

Python Module Index 17

Index 19
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 bootstrap module.

1.1 Building

In order to build self-contained single-artifact executables, shiv leverages pip and stdlib’s zipapp module.

Note: Unlike “conventional” zipapps, shiv packs a site-packages style directory of your tool’s dependencies into the resulting binary, and then at bootstrap time extracts it into a ~/.shiv cache directory. More on this in the Bootstrapping section.

shiv accepts only a few command line parameters of its own, and any unprocessed parameters are delegated to pip install.

For example, if you wanted to create an executable for Pipenv, you’d specify the required dependencies (pipenv and pew), the callable (either -e for a setuptools-style entry point or -c for a bare console_script name), and the output file.

```
$ shiv -c pipenv -o ~/bin/pipenv pipenv pew
```

This creates an executable (~/bin/pipenv) containing all the dependencies required by pipenv and pew that invokes the console_script pipenv 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.

```
$ shiv requests -o requests.pyz --quiet
$ ./requests.pyz
```

(continues on next page)
This is particularly useful for running scripts without needing to contaminate your Python environment, since the .pyz files can be used as a shebang!

1.2 Bootstrapping

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 sys.path. 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. 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__, etc.
Influencing Runtime

There are a number of environment variables you can specify to influence a pyz file created with shiv.

2.1 SHIV_ROOT

This should be populated with a full path, it effectively overrides ~/.shiv as the default base dir for shiv’s 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

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

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.
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-info`

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.

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

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

-p, --python <python>
The path to a python interpreter to use.

--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, --no-compile-pyc
Whether or not to compile pyc files during initial bootstrap.

-E, --extend-pythonpath, --no-extend-pythonpath
Add the contents of the zipapp to PYTHONPATH (for subprocesses).

--reproducible, --not-reproducible
Generate a reproducible zipapp by overwriting all files timestamps to a default value. Timestamp can be overwritten by SOURCE_DATE_EPOCH env variable. If SOURCE_DATE_EPOCH is set, this option will be implicitly set to true too.

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, @kwizn, @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) \(\rightarrow\) 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 find_entry_point (site_packages_dirs: List[pathlib.Path], console_script: str) \(\rightarrow\) 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.

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](https://docs.python.org/3.7/library/zipapp.html#cmdoption-zipapp-c)


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) \(\rightarrow\) None

Write a shebang line.

Parameters

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

3.3.3 pip

shiv.pip clean_pip_env () \(\rightarrow\) 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.pipinstall(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.
- `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.extract_site_packages(archive, target_path, compile_pyc, 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.

shiv.bootstrap.import_string(import_name)

Returns a callable for a given setuptools style import string

**Parameters**

- `import_name` – 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`)

```python
import os
import sys
import django

# setup django
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)

```bash
#!/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

# In production -

$ ./<project_name>.pyz production
CHAPTER 4

Indices and tables

• genindex
• modindex
• search
S

shiv, 10
shiv.bootstrap, 11
shiv.bootstrap.environment, 11
shiv.bootstrap.interpreter, 12
shiv.builder, 10
shiv.cli, 10
shiv.constants, 10
shiv.pip, 10
Symbols
-compile-pyc, -no-compile-pyc
  shiv command line option, 8
-compressed, -uncompressed
  shiv command line option, 8
-reproducible, -not-reproducible
  shiv command line option, 8
-site-packages <site_packages>
  shiv command line option, 8
-version
  shiv command line option, 7
-E, -extend-pythonpath,
  -no-extend-pythonpath
  shiv command line option, 8
-c, -console-script <console_script>
  shiv command line option, 7
-e, -entry-point <entry_point>
  shiv command line option, 7
-j, -json
  shiv-info command line option, 8
-o, -output-file <output_file>
  shiv command line option, 7
-p, -python <python>
  shiv command line option, 8

B
bootstrap() (in module shiv.bootstrap), 11

C
cache_path() (in module shiv.bootstrap), 11
clean_pip_env() (in module shiv.pip), 10
console_script_exists() (in module shiv.cli), 10
create_archive() (in module shiv.builder), 10
current_zipfile() (in module shiv.bootstrap), 11

E
extract_site_packages() (in module shiv.bootstrap), 11

F
find_entry_point() (in module shiv.cli), 10

I
import_string() (in module shiv.bootstrap), 11
install() (in module shiv.pip), 10

P
PIP_ARGS
  shiv command line option, 8
PYZ
  shiv-info command line option, 8

R
run() (in module shiv.bootstrap), 11

S
shiv (module), 10
shiv command line option
  -compile-pyc, -no-compile-pyc, 8
  -compressed, -uncompressed, 8
  -reproducible, -not-reproducible, 8
  -site-packages <site_packages>, 8
  -version, 7
-E, -extend-pythonpath,
  -no-extend-pythonpath, 8
-c, -console-script
  <console_script>, 7
-e, -entry-point <entry_point>, 7
-o, -output-file <output_file>, 7
-p, -python <python>, 8
PIP_ARGS, 8
shiv-info command line option
  -j, -json, 8
PYZ, 8
shiv.bootstrap (module), 11
shiv.bootstrap.environment (module), 11
shiv.bootstrap.interpreter (module), 12
shiv.builder (module), 10
shiv.cli (module), 10
shiv.constants (module), 10
shiv.pip (module), 10

W
write_file_prefix() (in module shiv.builder), 10