shiv

Nov 14, 2018

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 *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 it's 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
Python 3.6.1 (default, Apr 19 2017, 15:02:08)
[GCC 4.2.1 Compatible Apple LLVM 7.3.0 (clang-703.0.29)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
```

(continues on next page)

(continued from previous page)

```
>>> import requests
>>> requests.get('http://shiv.readthedocs.io/')
<Response [200]>
```

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.

CHAPTER 2

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.

CHAPTER 3

Table of Contents

3.1 Motivation & Comparisons

3.1.1 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 it's 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.

3.1.2 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!

3.2 Shiv API

3.2.1 cli

`shiv.cli.`**`copy_bootstrap`**(*bootstrap_target: pathlib.Path*) → None
Copy bootstrap code from shiv into the pyz.

This function is excluded from type checking due to the conditional import.

Parameters **`bootstrap_target`** – The temporary directory where we are staging pyz contents.

`shiv.cli.`**`find_entry_point`**(*site_packages: 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`** – A path to a site-packages directory on disk.
- **`console_script`** – A console_script string.

constants —

This module contains various error messages.

3.2.2 builder

This module is a slightly 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`**(*source: pathlib.Path*, *target: pathlib.Path*, *interpreter: str*, *main: str*, *compressed: bool = True*) → None
Create an application archive from SOURCE.

A slightly 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.

3.2.3 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
```

3.2.4 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.
- **buidl_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*)
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

3.2.5 bootstrap.environment

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

3.2.6 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`.

3.3 Deploying django apps

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

3.3.1 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
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.

3.3.2 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
```

(continues on next page)

(continued from previous page)

```
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

Python Module Index

S

Index