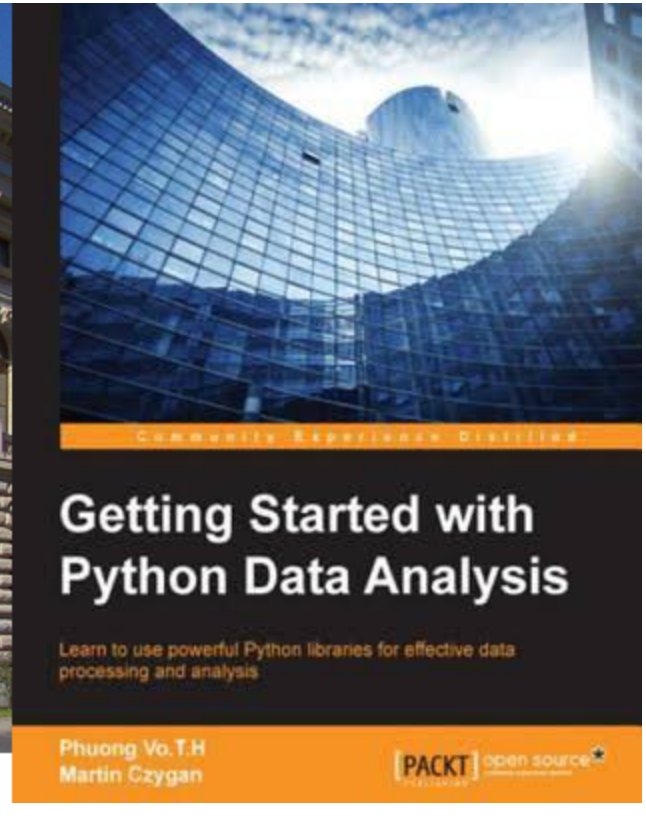
Packaging Python Applications

PyCon Balkan, Belgrade, 2018-11-17 16:00-16:25

Martin Czygan / github.com/miku / @cvvfj

About me

- Software developer at Leipzig University Library
- Co-author *Getting Started with Python Data Analysis* (2015)
- Maintainer of a few niche open source tools



Interests

- interest in (build) automation: code, writing, data
- anecdata: ant, ephemeral VMs, tried to ease adoption of Python at workplace

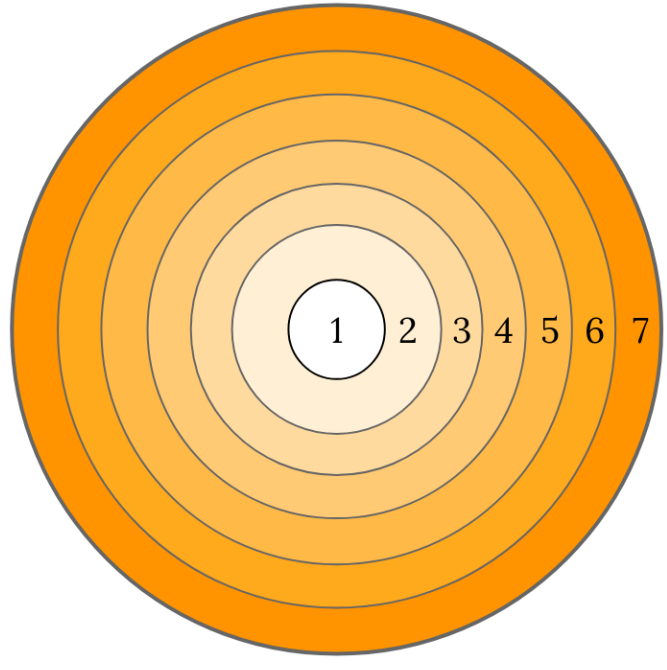
There is some satisfaction in being able to just run *one command*. There are also some trade-offs.

Packaging

- loosely defined as *approaches* and *tools* to create usable (installable, shippable) software
- There is this thing called: The packaging gradient

Packaging for Python **applications**

1. **PEX** - libraries included
2. **anaconda** - Python ecosystem
3. **freezers** - Python included
4. **images** - system libraries included
5. **containers** - sandboxed images
6. **virtual machines** - kernel included
7. **hardware** - plug and play



Packaging is moderately exiting

- not directly related to features
- many options

> ... plethora of packaging options ...
> (https://packaging.python.org/overview/)

- less formalized, project-dependent

So why care about packaging?

- as individual or team
- as company

Individuals or teams

- to share code
- collaborate or invite contributions

> It might seem strange to think about packaging before writing code, but this process does wonders for avoiding future headaches. (https://packaging.python.org/overview/)

As a company

- Aug 9, 2000: The Joel Test: 12 Steps to Better Code

There are two items related to packaging:

- #2 Can you make a build in one step?
- #3 Do you make daily builds?

Obviously, there are many posts with title "Joel test in 2019" and such.

On item #2

> By this I mean: how many steps does it take to make a shipping build from the latest source snapshot? On good teams, there's a single script you can run that does a full checkout from scratch, rebuilds every line of code, makes the EXEs, in all their various versions, languages, and #ifdef combinations, creates the installation package, and creates the final media — CDROM layout, download website, whatever.

On item #2

> If the process takes any more than one step, it is prone to errors. And when you get closer to shipping, you want to have a very fast cycle of fixing the "last" bug, making the final EXEs, etc. If it takes 20 steps to compile the code, run the installation builder, etc., you're going to go crazy and you're going to make silly mistakes.

Packaging is only a part of a larger story

- the decade of devops (2009-)
- code--build--test--package--release--configure--monitor

There slides reflect only a few small pieces of the puzzle.

The Packaging Gradient

There is an nice talk called *The Packaging Gradient* by Mahmoud Hashemi at PyBay 2017 (yt: 601), BayPiggies2017 (yt: 82) - blog post.

> One lesson threaded throughout Enterprise Software with Python is that deployment is not the last step of development.

What will we look at?

- "just do nothing"
- modules, packages, distributions, PyPI
- single file deployments (PEP 441)
- reusing linux package managers
- building images

A single Python file (module)

- with a large standard library, it is possible to write useful things in Python and stdlib only
- deployment cannot get simpler that `scp` or `curl`

```
# scp script.py ...
```

Requirements:

- ssh
- python on target machine (matching version)
- script should have no dependencies

Beautiful, if possible (*simple is better than complex*).

Module, package, distribution

- a module is as single, importable python file
- a package is a directory (containing an `__init__.py`)
- a distribution is a way to bundle zero or more packages (source and built distribution)

A minimal setup.py

Writing a setup.py file can be simple.

> The smallest python project is two files. A setup.py file which describes the metadata about your project, and a file containing Python code to implement the functionality of your project.

However, there are only three required fields: name, version, and packages.

> The name field must be unique if you wish to publish your package on the Python Package Index (PyPI). The version field keeps track of different releases of the project. The packages field describes where you've put the Python source code within your project.

A minimal setup.py

```
$ tree
.
├── hellopkg
│   ├── __init__.py
│   └── hello.py
└── setup.py
```

```
$ cat setup.py
from setuptools import setup

setup(name='hellopkg',
      version='0.1.0',
      packages=['hellopkg'])
```

Additional fields in setup.py

Usually, you project will have dependencies and it might come with command line programs:

```
from setuptools import setup

setup(name='hellopkg',
      version='0.1.0',
      packages=['hellopkg'],
      install_requires=['requests'],
      entry_points={
          'console_scripts': [
                'hellopkg-cli=hellopkg.hello:hello'
          ],
      })
```

Creating a source distribution (sdist)

- includes source files, might adjust what gets included in a MANIFEST.in

```
$ python setup.py sdist
...

$ tree .
.
├── dist
│   └── hellopkg-0.1.0.tar.gz
├── hellopkg
│   ├── __init__.py
│   └── hello.py
└── setup.py
```

Upload to [Test]PyPI

```
$ twine upload dist/hellopkg-0.1.0.tar.gz
```

- https://pypi.org/project/twine/

> The biggest reason to use twine is that it securely authenticates you to PyPI over HTTPS using a verified connection regardless of the underlying Python version, while whether or not python setup.py upload will work correctly and securely depends on your build system, your Python version and the underlying operating system.

Use testpypi for testing

- https://packaging.python.org/guides/using-testpypi/

> TestPyPI is a separate instance of the Python Package Index (PyPI) that allows you to try out the distribution tools and process without worrying about affecting the real index. TestPyPI is hosted at test.pypi.org

Minimal viable distribution

1. write code
2. add setup.py
3. `$ python setup.py sdist && twine upload dist/*`

Tools to create a Python Package

- cookiecutter
- e.g. https://github.com/audreyr/cookiecutter-pypackage

```
$ pip install -U cookiecutter
$ cookiecutter https://github.com/audreyr/cookie...
```

Creating a built distribution (bdist)

- wheel packaging standard, PEP 427

```
$ python setup.py bdist_wheel
.
├── dist
│   └── hellopkg-0.1.0-py3-none-any.whl
├── hellopkg
│   ├── __init__.py
│   └── hello.py
└── setup.py

$ file dist/hellopkg-0.1.0-py3-none-any.whl
dist/hellopkg-0.1.0-py3-none-any.whl: Zip archive data
```

Wheel types

- universal, pure python (2 or 3), platform (ext)

The wheel filename follows PEP 425.

```
{distribution}-{version}(-{build tag})?- \
    {python tag}-{abi tag}-{platform tag}.whl
```

> For example, the tag py27-none-any indicates compatible with Python 2.7 with no abi requirement, on any platform. (cf. PEP 3149).

```
hellopkg-0.1.0-cp36-cp36m-macosx_10_12_x86_64.whl
```

Creating a built distribution (bdist)

```
$ python setup.py bdist_wheel
```

> This will build any C extensions in the project and then package those and the pure Python code into a .whl file in the dist directory.

-- https://wheel.readthedocs.io/en/stable/user_guide.html

Crossplatform wheels

> Python wheels are great. Building them across Mac, Linux, Windows, on multiple versions of Python, is not.

- https://github.com/joerick/cibuildwheel (Travis CI, Appveyor, and CircleCI)

The manylinux tag

- https://github.com/pypa/manylinux, demo

> The goal of the manylinux project is to provide a convenient way to distribute binary Python extensions as wheels on Linux.

From PEP 513:

> ... For Linux, the situation is much more delicate. In general, compiled Python extension modules built on one Linux distribution will not work on other Linux distributions, [...]
>
> The two key causes are dependencies on shared libraries which are not present on users' systems, and dependencies on particular versions of certain core libraries like glibc.

Additional setup.cfg

> If your project contains no C extensions and is expected to work on both Python 2 and 3, you will want to tell wheel to produce universal wheels by adding this to your setup.cfg file.

```
$ python setup.py bdist_wheel --universal
```

or

```
$ cat setup.cfg

[bdist_wheel]
universal = 1
```

Wheel benefits

> Wheels are unbelievably critical in that they allow super easy caching of pre-built packages on any given host. If you've noticed these days you can type `pip install numpy` at will and it seems to usually run in less than two seconds rather than 5 minutes, thank wheels. This is particularly a big deal if you work lot with CI.

-- zzzeek on Aug 14, 2016

- 291/360 popular packages available as wheels (https://pythonwheels.com/)

Now to something completely different

Python can run zip files

Via PEP-441:

> Python has had the ability to execute directories or ZIP-format archives as scripts since version 2.6. When invoked with a zip file or directory as its first argument the interpreter adds that directory to sys.path and executes the main module.

And:

> These archives provide a great way to publish software that needs to be distributed as a single file script but is complex enough to need to be written as a collection of modules.

Moderately popular feature

PEP-441 cont.:

> This feature is not as popular as it should be mainly because it was not promoted as part of Python 2.6.

Works with packages

See: Package.

Tooling for zip files: PEX

- https://github.com/pantsbuild/pex

> pex is a library for generating .pex (Python EXecutable) files which are executable Python environments in the spirit of virtualenvs. pex is an expansion upon the ideas outlined in PEP 441 and makes the deployment of Python applications as simple as cp.

> PEX files have been used by Twitter to deploy Python applications to production since 2011.

How it works

> PEX files rely on a feature in the Python importer that considers the presence of a `__main__.py` within the module as a signal to treat that module as an executable.

Python import subsystem is flexible to handle code from disk or from within a zip file.

> Adding #!/usr/bin/env python to the top of a .zip file containing a main.py and marking it executable will turn it into an executable Python program.

Launching an interpreter with dependencies installed

```
$ pex requests
Python 3.6.4 (default, Feb 14 2018, 14:01:23)
[GCC 4.2.1 Compatible Apple LLVM 9.0.0 ...
Type "help", "copyright", "credits" or ...
(InteractiveConsole)

>>> import requests
>>> requests.get("https://pyconbalkan.com")
<Response [200]>
```

Build a single executable

Use `-c` for script name and `-o` for an output file.

```
$ pex sphinx -c sphinx-quickstart -o sphinx.pex

$ ./sphinx
Welcome to the Sphinx 1.8.2 quickstart utility.
...
```

Limitations

- Editable distributions not supported
- Matching Python versions and platform-specific

Another tool: shiv

From LinkedIn.

> 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.

Easier to handle single files.

> 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.

Pex differences

- shiv only works with Python 3.6+
- shiv will decompress bundle transparently
- tries to mirror environment more closely (site-packages instead of wheels)

> [...] executables created with shiv can outperform ones created with PEX by almost 2x.

Reusing Linux package managers

- Linux package managers have solved a hard problem already
- Allows to add additional files (e.g. service definitions, configuration)
- Upgrade and downgrade path

Debian

- dh-virtualenv from Spotify

> The idea behind dh-virtualenv is to be able to combine the power of Debian packaging with the sandboxed nature of virtualenvs. In addition to this, using virtualenv enables installing requirements via Python Package Index instead of relying on the operating system provided Python packages. The only limiting factor is that you have to run the same Python interpreter as the operating system.

Example: dh-virtualenv

Example: Debian.

Docker and Python

Add a level of isolation.

- grocker (2016-07-20)
- build because debian packaging took 2 days (in 2015)
- docker pull and run

Building images with grocker

```
$ pip install grocker
```

Basic usage:

```
$ grocker build ipython==7.1.1 --entrypoint ipython
```

- multi-stage build (compile with build deps and runner)
- root, compiler and runner image
- alpine base image
- requires packaged applications
- https://asciinema.org/a/Jwg7rY0ARcX4TKzLUrrOA8qYZ

Running IPython in a container

```
$ docker images
REPOSITORY TAG   IMAGE ID     CREATED         SIZE
ipython    7.1.1 7ca7552ef4bf 36 seconds ago 76MB
```

Then run:

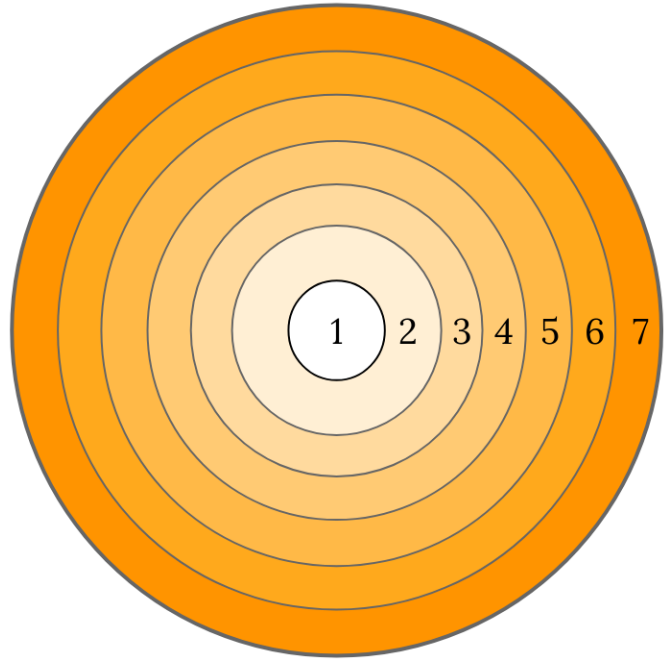
```
$ docker run --rm -ti ipython:7.1.1
Python 3.6.6 (default, Aug 24 2018, 05:04:18)
...
In [1]: import os

In [2]: os.getpid()
Out[2]: 1

In [3]:
```

Wrap up

- many options available
- basics are simple
- depends on your target platform

Packaging for Python **applications**

1. **PEX** - libraries included
2. **anaconda** - Python ecosystem
3. **freezers** - Python included
4. **images** - system libraries included
5. **containers** - sandboxed images
6. **virtual machines** - kernel included
7. **hardware** - plug and play



Resources

- Packaging Gradient blog post
- Share your code! (PyCon 2017)
- Python Packaging User Guide
- Python Packaging from The Architecture of Open Source Applications
- pex
- shiv
- dh-virtualenv
- grocker
- Slides and examples at https://github.com/miku/packpy