Less known packaging features and tricks

Who

Ionel Cristian Mărieș

„ionel” is read like *„yonel”*, `@ionelmc`, blog.ionelmc.ro

- Did PyPI releases of 40-something distinct packages, since 2007
- Working on a project with ~125 python package dependencies
- Working on a rewrite of virtualenv (still not merged, but quite usable)

But first ...

Trigger warning: There's going to be lots of talking about `setup.py`

What's going on in a `setup.py`?

- It's a really nasty archiver
  - There are a bunch of options from `distutils`
  - And some more from `setuptools`.
- `setuptools` adds very useful improvements (detailed later on). There's no reason to not use it. Even `pip` depends on it now-days.
- Boils down to having a file `setup.py` with:

```
from setuptools import setup
setup(name="mypackage", packages=["mypackage"], **lots_of_kwargs)
```

- And running `python setup.py sdist bdist_wheel`.

Setting the record straight

Mandatory clarifications: **packages** vs **distributions**

- *importable* **packages**:

```
├── package1
│   ├── __init__.py
│   ├── module.py
│   └── subpackage
│       └── __init__.py
└── package2
    ├── __init__.py
    ├── module.py
    └── subpackage
        └── __init__.py
```

- **distribution** *packages*:

```
lazy-object-proxy-1.2.0.tar.gz
lazy_object_proxy-1.2.0-cp27-none-win32.whl
lazy_object_proxy-1.2.0-cp27-none-win_amd64.whl
lazy_object_proxy-1.2.0-cp34-none-win32.whl
lazy_object_proxy-1.2.0-cp34-none-win_amd64.whl
```

Types of archives

- They are actually called *distributions*.
- packaging.python.org calls them *distribution packages* to avoid some of the confusion.
- Two kinds:
  - Source distributions (`sdist`)
  - Binary/built distributions (`bdist`, `bdist_wheel`, `bdist_egg` etc).

  They have different rules for gathering the files because they generally have different files.

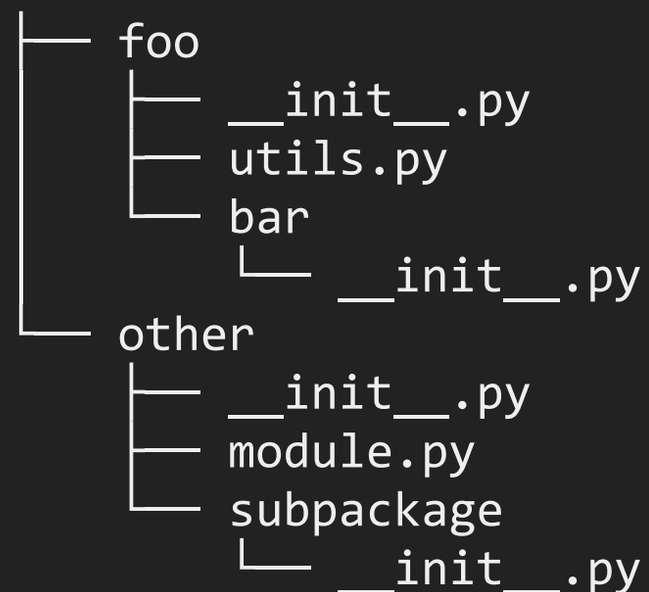
File gathering: `sdist`

- Files included in `sdist` by default:
  - `README`, `README.txt`, `setup.py`, `test/test*.py`
  - all files that match `packages` and `py_modules`
  - all C sources from `ext_modules` (no headers 😕)
  - all files from `package_data`
  - all files from `data_files`
  - all files from `scripts`
- Whatever you manage to specify in `MANIFEST.in`

File gathering: `bdist*`

- Files included in `bdist*`:
  - all files that match `packages` and `py_modules`
  - all `.so`/`.pyd` built by `ext_modules`
  - all files from `package_data`
  - all files from `data_files`
  - all files from `scripts`
- If `include_package_data=True` is used (`setuptools` only) then files from `MANIFEST.in` will get included, if they are inside a package.

Inside `setup.py`: `packages` (1)



Packages for that:

- `foo`
- `foo.bar`
- `other`
- `other.subpackage`

Inside `setup.py`: `packages` (2)

Don't hard-code the list of packages, use `setuptools.find_packages()`

Don't:

```
setup(
    ...
    # everything is fine and dandy until one day someone
    # converts foo/utils.py to a package
    # and forgets to add `foo.utils`
    packages=['foo', 'foo.bar', 'other', 'other.subpackage']
)
```

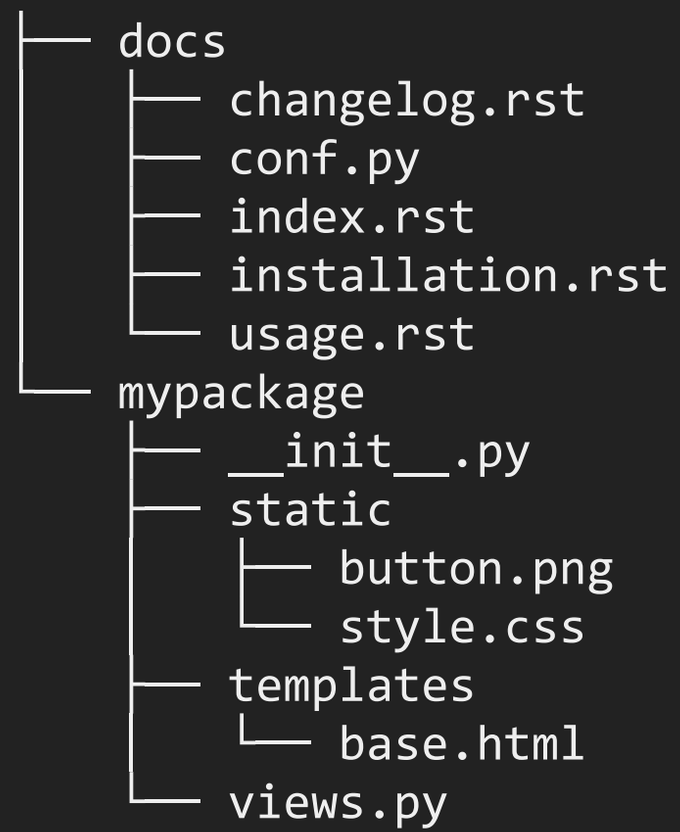
Do:

```
setup(
    ...
    packages=find_packages()
)
```

The `MANIFEST.in`

- It's a template with lots of commands: `include`, `exclude`, `recursive-include`, `recursive-exclude`, `global-include`, `global-exclude`, `prune` and `graft`.
- A too fine-grained `MANIFEST.in` is a frequent cause for issues: incomplete `sdist` when you forget to add the extension for your new file. **And you will forget.**
- Most projects can just use a `graft`, `global-exclude` and few `include`.

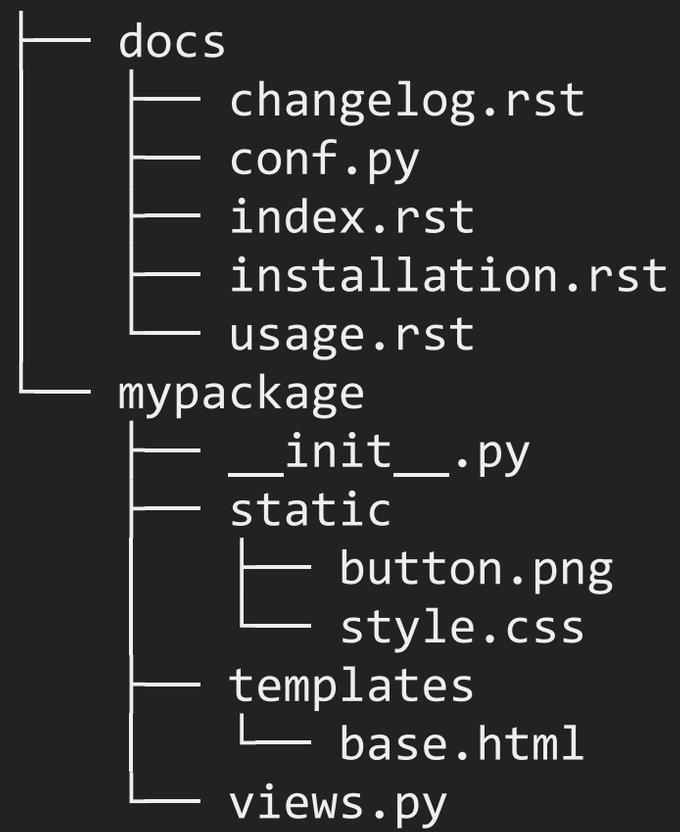
Bad MANIFEST.in



Too fine grained, missing files:

```
recursive-include mypackage *.html *.css *.png *.xml *.py
include docs/changelog.rst
```

Good MANIFEST.in



Just take whatever you have on the filesystem:

```
graft mypackage docs
global-exclude *.py[cod] __pycache__ *.so
```

Advices for `MANIFEST.in`

- When choosing the `MANIFEST.in` commands consider that dirty releases are better than unusable releases.

  A couple harmless stray files less bad than missing required files.

- Use Git/Mercurial, don't release with untracked files.
- Use check-manifest (integrate it in your CI or test suite).
- Consider using setuptools_scm extension instead of `MANIFEST.in` (it takes all the files from Git/Mercurial)

Inside `setup.py`: `package_data`

- With `distutils` you'd have to use `package_data` to include data files in packages.
- However, `setuptools` add the `include_package_data` option:

  If `True` then files from `MANIFEST.in` will get included, if they are inside a package.

Do not use `package_data`:

Don't use both `MANIFEST.in` and `package_data`. Use the easiest (`MANIFEST.in` + `include_package_data=True`)

Inside `setup.py`: `data_files`

```
data_files=[('config', ['cfg/data.cfg']),
            ('/etc/init.d', ['init-script'])]
```

Avoid like the plague. Too inconsistent to be of any general use:

- For relative paths:
  - `setup.py install` with `setuptools` put them inside the egg zip/dir
  - `setup.py install` with `distutils` and `pip install` put them in `sys.prefix` or `sys.exec_prefix`
- For absolute paths `sys.prefix` is not used. Installing in a virtualenv fails without `sudo`.

A quick interlude: applications

- If you really need `data_files` you are probably trying to package an application.
- Use a specialized package like:
  - `deb` (dh-virtualenv, py2deb)
  - `rpm`
  - pynsist (Windows)
  - or your own CustomThing™ (NSIS, makeself etc)

A quick interlude: dependencies

```
setup(
    ...
    install_requires=[
        'Jinja2',
        ...
    ]
)
```

- Dependencies install *alongside*, unlike with `npm` (node.js)
- If you want to bundle/vendor deps - *pex <https://github.com/pantsbuild/pex>*:
  - Bundles up your code and the deps in a self-extracting executable
  - Installs them in a virtualenv automatically

Importing code in `setup.py`

Don't do this:

```
from setuptools import setup
from mypackage import __version__
setup(
    name='mypackage',
    version=__version__,
    ...
)
```

- Users won't be able to install your package if you import dependencies in your `__init__.py` (they might not be available).
- There are many other ways to get the version on packaging.python.org
- Note that setuptools_scm handles version for you.

The `__main__` module

Supported since Python 2.7 (`python -m mypackage` to run):

```
mypackage
├── __init__.py
├── cli.py
└── __main__.py
```

In `__main__.py` you'd have something like:

```
from mypackage.cli import main

if __name__ == "__main__":
    main()
```

You should never import anything from __main__ because `python -m mypackage` will run it as a script (thus creating double execution issues).

Inside `setup.py`: `entry_points`

Then in `setup.py`:

```
setup(
    ...
    entry_points={
        'console_scripts': [
            'mytool = mypackage.cli:main',
        ]
    }
)
```

Advantages over using `setup(scripts=['mytool'])`:

- Nice .exe wrappers on Windows
- Proper shebang (`#!/path/to/python`)

Optional dependencies

```
setup(
    ...
    extras_require={
        'pdf': ['reportlab'],
    },
)
```

- Then you `pip install "mypackage[pdf]"` to get support for pdf output.
- Some people abuse this feature for development/test dependencies.
  It works but you entangle your `setup.py` with development concerns.
  Tox is a good solution for development environments.

A quick interlude: Tox

```
# content of: tox.ini , put in same dir as setup.py
[tox]
envlist = py26,py27

[testenv]
deps=pytest       # install pytest in the venvs
commands=py.test  # or 'nosetests' or ...
```

- Reproducible environments, for each python version:
  - Installs dependencies you've specified
  - Installs your project (runs `setup.py install` or `develop`)
  - Runs your test commands

Other ways to manage environments

- There are other solutions for virtuelenv management:
  - vex and pew are notable.
- pyenv is another interesting solution but manages complete interpreters.

Environment markers - PEP-426

- An underused feature. Declarative conditional dependencies:

```
setup(
    ...
    extras_require={
        ':python_version=="2.6"': ['argparse'],
        ':sys_platform=="win32"': ['colorama'],
    },
)
```

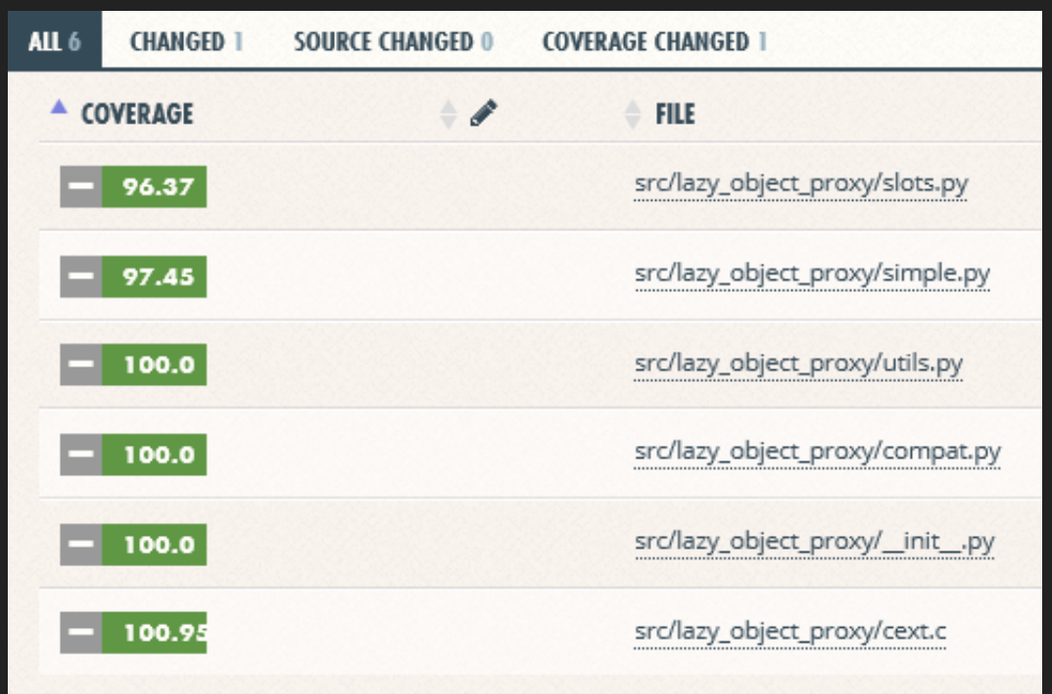
- **Why:** you can build universal wheels that have conditional dependencies.
- Environment markers are supported since *setuptools 0.7*
- More reading: wheel docs, PEP-426.

Coverage for C extensions

Easy to do on Linux:

```
export CFLAGS=-coverage
python setup.py clean --all build_ext --force --inplace
# run tests
```

Example on Coveralls:



Uploading

- Twine - secure upload to PyPI:
  ```
  twine upload dist/*
  ```
- Interesting recent change: PyPI doesn't allow reuploading distributions anymore. You can only delete.

Versioning

- Version normalization (`PEP-440`) active since *setuptools 8.0* and *pip 6.0*:
  - `1.2.3-4` becomes `1.2.3.post4`
  - `1.2.3-dev4` becomes `1.2.3.dev4`
  - `1.2.3alpha4` becomes `1.2.3a4`
  - etc
- semver.org not compatible with `PEP-440` on just two clauses:
  - #9 - prereleases, eg: `1.2.3-alpha`
  - #10 - build info, eg: `1.2.3+da4109fcf9`

Ending

There's a cookiecutter template that *bakes in* a lots of the ideas presented here:

cookiecutter-pylibrary

Thank you!