Sinoroc KB

sinoroc

Dec 11, 2023

CONTENTS

Part I

Foreword

Loosely structured bits of knowledge

Selected chapters

Author

- sinoroc.gitlab.io[1]
- sinoroc.github.io[2]

[1] https://sinoroc.gitlab.io
[2] https://sinoroc.github.io

Part II

Python

CHAPTER

ONE

PYTHON PACKAGE DATA

Further down is a minimal example showing how to achieve both:

- packaging a data file `file.src` in `sdist` only;
- and packaging another data file `file.bin` in `bdist` only;
- additionally it shows how `file.all` is packaged in both distribution packages and `file.not` in none of them.

> **`file.bin` and built files**
>
> Files such as `file.bin` are not in the original source code of the project (i.e. not in the *git* source code repository for example) but should still be installed. Typically these files are created during a build step such as `./setup.py build` for example (think `gettext *.mo` messages catalogs).

The gist of it is:

- first and foremost, always thoroughly clean up the working directory between two packaging attempts while tweaking these packaging options (in particular empty the `src/Thing.egg-info` directory containing the `SOURCES.txt` file as well as the `build`, and `dist` directories) or the results will be inconsistent;
- set the `include_package_data` option to `True`;
- `file.all` and files that belong in both `sdist` and `bdist` are specified in `MANIFEST.in`;
- `file.bin` and files that belong in `bdist` only are specified in `package_data`;
- `file.src` and files that belong in `sdist` only are specified in both `MANIFEST.in` and `exclude_package_data`;
- `file.not` and files that do not belong in any distribution package are not specified anywhere.

The directory structure for our example:

```
.
├ MANIFEST.in
├ setup.py
└ src
    └ thing
        ├ __init__.py
        └ data
            ├ file.all
            ├ file.bin
            ├ file.not
            └ file.src
```

In `MANIFEST.in`:

```
recursive-include src/thing *.all
recursive-include src/thing *.src
```

In `setup.py`:

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

import setuptools

setuptools.setup(
    exclude_package_data={'thing': ['data/*.src']},
    include_package_data=True,
    package_data={'thing': ['data/*.bin']},
    #
    name='Thing',
    version='1.0.0',
    #
    package_dir={'': 'src'},
    packages=setuptools.find_packages(where='src'),
)
```

This has been tested with:

- Python 3.6.7
- setuptools 39.0.1
- wheel 0.33.1

CHAPTER

TWO

PYTHON PACKAGING

2.1 Introduction

About proper packaging of Python projects…

2.2 Terminology

2.2.1 Module

Commonly a Python file (`mymodule.py`). Multiple Python modules are usually gathered in a Python package.

2.2.2 Package

Confusion #1: *Import package* vs. *distribution package*

One of the biggest confusion in the Python packaging terminology is around the meaning of the term *package*. Sometimes the terms *import package* and *distribution package* are used to clarify this.

It is sometimes named *import package*, as opposed to *distribution package* (see below).

A Python package is a directory containing at least one Python module `__init__.py` (the package initializer) and zero or more additional Python modules. The package initializer can be completely empty, but it has to be there.

It is possible for a package to contain other sub-packages in a tree-like structure. The outermost package is then called the *top-level package*.

2.2.3 Project

A Python project is usually a collection of code (and sometimes also data) that is intended to be distributed as a single unit. Typically a Python project is a library, an application, a plugin, a framework, or a toolkit. In most cases this corresponds to a single source code repository (for example a *git*, *SVN*, or *CVS* repository).

Multiple *top-level* packages and modules

For example *setuptools* (version `46.1.2` as of this writing) has two *top-level packages* `setuptools` and `pkg_resources`. It additionally seems to have one *top-level module* `easy_install`.

It is not often the case, but a Python project can contain multiple top-level packages. So of course the name of a top-level package is not always the same as the name of the project itself. It would be otherwise impossible to have more than one top-level package per project.

Some Python projects are only made of one or more Python modules directly at the root without tree-like package structure.

2.2.4 Distribution package

Confusion #2: *Distribution package* of a Python project vs. *Python distribution*

The term *Python distribution* is used to describe a specific implementation or build of a Python interpreter. *CPython* is probably the most famous one, but there are plenty of others such as *ActiveState Python* and *Anaconda*. Further examples: https://wiki.python.org/moin/PythonDistributions

Not to be confused with *import package* (see above) or *Python distribution* (see aside).

A distribution package contains a specific release of a project. A release being a snapshot of the Python project at a certain point in time. A distribution package is always labelled with the name of the project and the version string for the snapshot.

There are two common types of distribution formats: *source distribution* and *built distribution*.

Source distribution

A source distribution, sometimes abbreviated as *sdist*, is a distribution format.

A source distribution is meant to be installable on all Python interpreters and platforms that the project supports. It is not tied to a specific Python interpreter implementation, Python interpreter version, operating system, CPU architecture, CPU bitness. A source distribution can be used to build all the built distributions for all targets the project supports.

Source distributions are *gzip*'ed *tar* files with the `.tar.gz.` extension.

Attention: It is strongly recommended to always offer at least the *sdist* of a Python project (for example on PyPI). The reason is that it is always possible to use the *sdist* on any platform. On the other hand it is most likely impossible to use a *bdist* targetted for another platform.

So if no *bdist* of the project is available for the target platform, the *sdist* can still be used and eventually a target specific *bdist* can be built locally.

Built distribution

A built distribution, sometimes abbreviated as *bdist*, is a distribution format. It is designed so that the installation step is as straightforward as possible. In short: files only need to be extracted from the built distribution archive and copied to the right locations on disk. It does not require any kind of build step, as all files in a built distribution are already built for the specific target environment. Build distributions can be platform-specific.

Nowadays the only kind of built distributions one should know about is the *wheel*. The *egg* is an older kind of built distribution that should not be used anymore (use *wheel* instead).

Wheel

Wheel is a *built distribution* format. It is the preferred format of *distribution package*. It is defined by a standard specification[3]. A *wheel* is a file with the `.whl` extension.

2.2.5 Python package index

The *Python package index*, commonly called *PyPI*, is the main repository of Python project distributions packages.

It can be found at following URL:

- https://pypi.org/

2.3 References

- David Beazley "*Modules and Packages: Live and Let Die!*"
  - http://www.dabeaz.com/modulepackage/ModulePackage.pdf
- Glossary — Python Packaging User Guide
  - https://packaging.python.org/en/latest/glossary/

[3] https://packaging.python.org/en/latest/specifications/binary-distribution-format/

CHAPTER

THREE

PACKAGING TOOLS COMPARISONS

3.1 Use cases

	Install Python	Install packages	Build distributions	Upload distributions	Manage virtual environments	Lock files
build	no	no	yes	no	no	no
Flit	no	yes	yes	yes	yes	yes
Hatch	no	yes	yes	yes	yes	no
PDM	no	yes	yes	yes	yes	yes
pip	no	yes	yes	no	no	yes
pip-tools	no	yes	no	no	no	yes
Pipenv	no	yes	no	no	yes	yes
pipx	no	yes	no	no	no	no
Poetry	no	yes	yes	yes	yes	yes
pyenv	yes	no	no	no	no	no
Pyflow	yes	yes	yes	yes	yes	yes
setuptools	no	yes	yes	no	no	no
twine	no	no	no	yes	no	no
venv	no	no	no	no	yes	no
virtualenv	no	no	no	no	yes	no
virtualenvwrapper	no	no	no	no	yes	no
wheel	no	no	yes	no	no	no

Build back-ends are not listed here, but they are in a dedicated section below.

3.2 Comparisons

3.2.1 Development workflow tools

	[build-system] (PEP-517)	Builc	Up-load	Manage virtual en-vironments	Interchangeable build back-end	Plu-gins	Lock file
Flit	yes	yes	yes	no	no	no	no
Hatch	yes	yes	yes	yes	yes	yes	no
PDM	yes	yes	yes	yes	yes	yes	yes
Po-etry	yes	yes	yes	yes	no	yes	yes
Pyflov	no	yes	yes	yes	no	no	yes

See also build back-end features in dedicated section.

There is no standard for lock files.

3.2.2 Install Python interpreters

	Install Python interpreters
pyenv	yes
Pyflow	yes

3.2.3 Install packages

	Dependency resolution	Editable
pip	yes	yes
pip-tools	yes	yes
Pipenv	yes	yes
pipx	yes	no

`pipx` is intended to be used to install standalone applications rather than to install packages in a virtual environment.

3.2.4 Build distributions

These tools are also called "*build front-ends*".

	[build-system] (PEP-517)	sdist	wheel
build	yes	yes	yes
pip	yes	no	yes
wheel	no	no	yes
dev workflow tools (Hatch, Flit, PDM, Poetry, etc.)	yes	yes	yes

3.2.5 Build back-ends

	[build-system] (PEP-517)	[project] (PEP-621)	Editable installation (PEP-660)	Extensions configuration
enscons	yes	yes	yes	*SCONS*
flit-core	yes	yes	yes	no
hatchling	yes	yes	yes	via plug-ins
maturin	yes	yes	yes	*Cargo* (*Rust*)
meson-python	yes	yes	yes	*Meson*
pdm-backend	yes	yes	yes	no
poetry-core	yes	no	yes	build.py[4]
pymsbuild	yes	no	no	_msbuild.py
scikit-build-(	yes	yes	no	*CMake*
setuptools	yes	yes	yes	setup.py
trampolim	yes	yes	no	no
whey	yes	yes	yes	no

3.2.6 Upload distributions

	Upload
Flit	yes
Hatch	yes
PDM	yes
Poetry	yes
twine	yes

3.2.7 Manage virtual environments

	For any Python interpreter	Description in file
Hatch	yes	yes[5]
nox	yes	yes[6]
PDM	yes	no
Pipenv	yes	no
Poetry	yes	no
tox	yes	yes[7]
venv	no	no
virtualenv	yes	no
virtualenvwrapper	yes	no

Unlike the other tools presented in this section, `venv` is part of Python's own standard library, it should be always available without having to be installed separately. But note that some Linux distributions (e.g. Debian, Ubuntu, and derivatives) made the decision to package `venv` separately from the rest of the Python distribution and consequently it might be necessary to install `venv` explicitly (typically with a command such as `apt install python3-venv`, consult the documentation of the Linux distribution for exact details).

[4] Poetry has an undocumented feature allowing the customization of the build process via a `build.py` file, which indirectly allows the handling of C extensions (this is comparable to `setuptools` own *setup.py*).

[5] `[tool.hatch.envs]` section of `pyproject.toml`

[6] `noxfile.py`

[7] `tox.ini`

3.2.8 Lock files

There is no PyPA standard for the concept of "*lock files*". There is some kind of a *de facto* convention around *pip*'s `requirements.txt` file format but it can not be considered a good enough *lock file* format.

	Format
pip	`requirements.txt`
pip-tools	`requirements.txt`
Pipenv	`Pipfile.lock`
poetry	`poetry.lock`
PDM	`pdm.lock`

CHAPTER

FOUR

PYTHON PROJECT VERSION SINGLE-SOURCING

4.1 Problem

It is not entirely straightforward where the version string should be written within a Python project.

A couple of things are sure:

- the version must be written in a `__version__` attribute as a string (see PEP 396[8])
- the version string must be available from the setup script
- the version string should be in the changelog

It is annoying to have to keep the version string up to date in these three locations. A solution for single-sourcing the project version would fix that.

4.2 Solution

This solution shows how to keep the Python project version string in just one place. The suggested location is in the change log:

Listing 1: CHANGELOG.rst

```
1.2.3
=====

* More bugs fixed

1.2.2
=====

* Bugs fixed
```

The current version string should always be on the same line and on its own so that the setup script can easily find it and extract it:

Listing 2: setup.py

```
import os
import setuptools

with open(os.path.join(HERE, 'CHANGELOG.rst')) as file_:
    changelog = file_.read()

setuptools.setup(
```

(continues on next page)

[8] https://www.python.org/dev/peps/pep-0396/

(continued from previous page)

```
    name='Example',
    version=changelog.splitlines()[0],
    # ...
)
```

From the actual code of the project the version number should be accessed via `importlib.metadata`. Knowing the name of the project it is easy to get the version string:

Listing 3: src/example/__init__.py

```
import importlib.metadata

__version__ = importlib.metadata.version('Example')
```

The `importlib.metadata` package is part of the standard library starting with *Python 3.8*. For earlier versions use importlib-metadata[9] instead.

As a positive side effect, changing the version number forces the project maintainer to modify the change log and thus they always get at least one chance to keep it up to date.

[9] https://pypi.org/project/importlib-metadata/

CHAPTER FIVE

PYTHON PROJECT NAME

5.1 Problem

How to get the name of the project containing the current module (or package)?

- https://stackoverflow.com/a/60363617
- https://stackoverflow.com/a/60351412
- https://stackoverflow.com/a/60975978
- https://stackoverflow.com/a/63849982

5.2 Solution

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

import importlib.util
import pathlib

import importlib_metadata

def get_distribution(file_name):
    result = None
    for distribution in importlib_metadata.distributions():
        try:
            relative = (
                pathlib.Path(file_name)
                .relative_to(distribution.locate_file(''))
            )
        except ValueError:
            pass
        else:
            if relative in distribution.files:
                result = distribution
    return result

def _alpha():
    file_name = importlib.util.find_spec('alpha').origin
    distribution = get_distribution(file_name)
    print("alpha", distribution.metadata['Name'])

def _bravo():
    import bravo
    file_name = bravo.__file__
```

(continues on next page)

(continued from previous page)

```
    distribution = get_distribution(file_name)
    print("bravo", distribution.metadata['Name'])

if __name__ == '__main__':
    _alpha()
    _bravo()
```

5.2.1 Update February 2021

Looks like this could be solved in a simpler way thanks to the newly added *packages_distributions()* function in *importlib_metadata*:

- https://importlib-metadata.readthedocs.io/en/stable/using.html#package-distributions
- https://github.com/python/importlib_metadata/pull/287/files

CHAPTER

SIX

PYTHON IMPORTS

1. Identify clearly what you want your top level modules and packages to be.
2. Make all imports absolute.
3. Either:
    - make your project a real installable project, so that those top level modules and packages are installed in the environment's `site-packages` directory;
    - or make sure that the current working directory is the one containing the top level modules and packages.
4. Make sure to call your code via the *executable module* method instead of the *script* method:
    - DO
        - `path/to/pythonX.Y -m toplevelpackage.module`
        - `path/to/pythonX.Y -m toplevelmodule`
        - `path/to/pythonX.Y -m toplevelpackage.subpackage` (assuming there is a `toplevelpackage/subpackage/__main__.py` file)
    - DON'T
        - `path/to/pythonX.Y toplevelpackage/module.py`
        - `path/to/pythonX.Y toplevelmodule.py`
5. Later on, once it all works well and everything is under control, you might decide to change some or all imports to relative. (If things are done right, I believe it could be possible to make it so that it is possible to call the executable modules from any level within the directory structure as the current working directory.)

References:

- Old reference, possibly outdated, but assuming I interpreted it right, it says that running *scripts* that live in a package is an anti pattern, and one should use `python -m package.module` instead:
    - https://mail.python.org/pipermail/python-3000/2007-April/006793.html
    - https://www.python.org/dev/peps/pep-3122/

CHAPTER SEVEN

PYTEST

7.1 Introduction

Python test runner

http://pytest.org/

7.2 pycodestyle and pylint

Use the plugins pytest-pep8[10] and pytest-pylint[11].

pep8 vs. pycodestyle

The Python project `pep8` has been renamed[12] to `pycodestyle`. But there is no `pytest-pycodestyle` project yet.

https://bitbucket.org/pytest-dev/pytest-pep8/issues/15

With these plugins the linting operations are completely integrated within the test workflow. The results of the tests and linting operations are rendered in a consistent format.

[10] https://pypi.python.org/pypi/pytest-pep8
[11] https://pypi.python.org/pypi/pytest-pylint
[12] https://github.com/PyCQA/pycodestyle/issues/466

7.2.1 pep8 only

Run only the pep8 linting.

Listing 1: shell console

```
$ pytest --pep8 -m pep8
```

7.2.2 pylint only

Run only the `pylint` linting.

Listing 2: shell console

```
$ pytest --pylint -m pylint
```

7.2.3 Both pep8 and pylint

Run both linting tools but not the tests themselves.

Listing 3: shell console

```
$ pytest --pep8 --pylint -m 'pep8 or pylint'
```

Run all the tests including the linting tools.

Listing 4: shell console

```
$ pytest
```

CHAPTER

EIGHT

TOX

8.1 Introduction

The tox tool allows to easily create multiple Python virtual environments while specifying a list of Python dependencies to install in each environment as well as a list of commands to run in each environment.

The original purpose of the tool is to test the source distribution (sdist) of a Python project against multiple combinations of Python interpreters and Python dependencies.

- https://tox.readthedocs.io/

8.2 Defaults

Listing 1: tox.ini

```
[tox]
envlist =
    py37
    py38
isolated_build = True
# ...

[testenv]
commands =
    python3 -m pytest
extras =
    dev_test
# ...
```

8.3 Development environment

It is a good idea to setup an environment for interactive use. The purpose of this environment is to be actually activated from the interactive shell in order to do the actual development.

The `commands` configuration setting should be relatively neutral. It can also be left empty. There is no need to trigger any test suite or linting, since those should be triggered manually once the environment is active.

The environment should contain the dependencies for all use cases: test, build, distribute, and then eventually some more to develop.

Listing 2: tox.ini

```
# ...
[testenv:develop]
commands =
deps =
    dev_doc
    dev_lint
    dev_package
    dev_test
usedevelop = True
# ...
```

8.4 Notes

8.4.1 GitLab CI

Automatically set the `TOXENV` environment variable based on the job name:

Listing 3: .gitlab-ci.yml

```
'.review':
  script:
    - 'export TOXENV="${CI_JOB_NAME##review}"'
    - 'python3 -m pip install tox'
    - 'python3 -m tox'

'review py37':
  extends: '.review'
  image: 'python:3.7'

'review py38':
  extends: '.review'
  image: 'python:3.8'
```

CHAPTER

NINE

PEX

9.1 Introduction

In a couple of words: `pex` helps create *self-contained executable Python virtual environments*.

https://pex.readthedocs.io/

https://www.youtube.com/watch?v=NmpnGhRwsu0

9.2 Bootstrap

Bootstrap `pex` with these steps:

- create a short lived Python virtual environment;
- install `pex` in this environment;
- use the newly installed `pex` to create a `pex` file:
  - containing the `pex` project as well as the dependencies; *and*
  - having the `pex` console script as its entry point.

With Python 3 and the `~/bin` directory on the `PATH` this could look like this:

Listing 1: shell console

```
$ python3 -m venv pexenv
$ . pexenv/bin/activate
(pexenv) $ pip install pex
(pexenv) $ pex \
> 'pex[requests,cachecontrol]' \
> --console-script=pex \
> --output-file=~/bin/pex
(pexenv) $ deactivate
```

(continues on next page)

(continued from previous page)

```
$ rm --force --recursive pexenv
$ which pex
$ pex --version
```

The `pexenv` Python virtual environment can be deleted immediately afterwards. `pex` can be used directly since it is self contained in its own Python virtual environment within the `~/bin/pex` file.

9.3 Overview

Per default pex starts the Python interpreter in a dynamically created empty virtual environment.

Listing 2: shell console

```
$ pex
Python 2.7.12 (default, Nov 19 2016, 06:48:10)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>> exit()
```

It is possible to select which Python interpreter should be used.

Listing 3: shell console

```
$ pex --python=python3
Python 3.5.2 (default, Nov 17 2016, 17:05:23)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>> exit()
```

`pex` allows to specify which Python projects should be installed in the virtual environment.

Listing 4: shell console

```
$ pex 'requests<2.0.0' 'setuptools==30'
Python 3.5.2 (default, Nov 17 2016, 17:05:23)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>> import requests
>>> requests.__version__
'1.2.3'
>>> import setuptools
>>> setuptools.__version__
'30.0.0'
>>> exit()
```

The dependencies can be specified via a `pip` `requirements.txt` file.

Listing 5: shell console

```
$ pex --requirement=requirements.txt
```

`pex` also allows to specify an entry point that should be executed from within the virtual environment.

Listing 6: shell console

```
$ pex 'httpie==0.9.6' --console-script=http -- --version
0.9.6
$ pex --python=python3 --entry-point=http.server
Serving HTTP on 0.0.0.0 port 8000 ...
```

Finally pex allows to write this self-contained executable virtual environment into a single file.

Listing 7: shell console

```
$ pex --python=python3 --entry-point=http.server --output-file=server.pex
$ ./server.pex
Serving HTTP on 0.0.0.0 port 8000 ...
```

9.4 Inspect

Since pex files are ZIP archives, inspecting their content is very straighforward.

Listing 8: shell console

```
$ python -m zipfile -l example.pex
$ unzip -l example.pex
```

It is a good idea to check that only the required and necessary dependencies are included. Nothing more and nothing less should be found in the .deps directory.

9.5 setuptools

To easily build a pex executable with setuptools use the bdist_pex command. bdist_pex will use the console_scripts entry point bearing the exact name of the Python project itself.

Listing 9: setup.py

```python
import setuptools

NAME = 'Example'

setuptools.setup(
    name=NAME,
    entry_points={
        'console_scripts': [
            '{}=example.app:run'.format(NAME),
        ],
    },
    # ...
)
```

9.5.1 Requirements

For a stricter control over the dependencies added to the `pex` file, a `requirements.txt` file can be specified via the `--pex-args` option.

Listing 10: shell console

```
$ python setup.py bdist_pex --pex-args='--requirement=requirements.txt'
```

CHAPTER

TEN

PYTHON TASK RUNNERS

- https://pypi.org/project/chuy/
- https://pypi.org/project/doit/
- https://pypi.org/project/invoke/
- https://pypi.org/project/poethepoet/
- https://pypi.org/project/taskipy/
- https://pypi.org/project/thx/

CHAPTER

ELEVEN

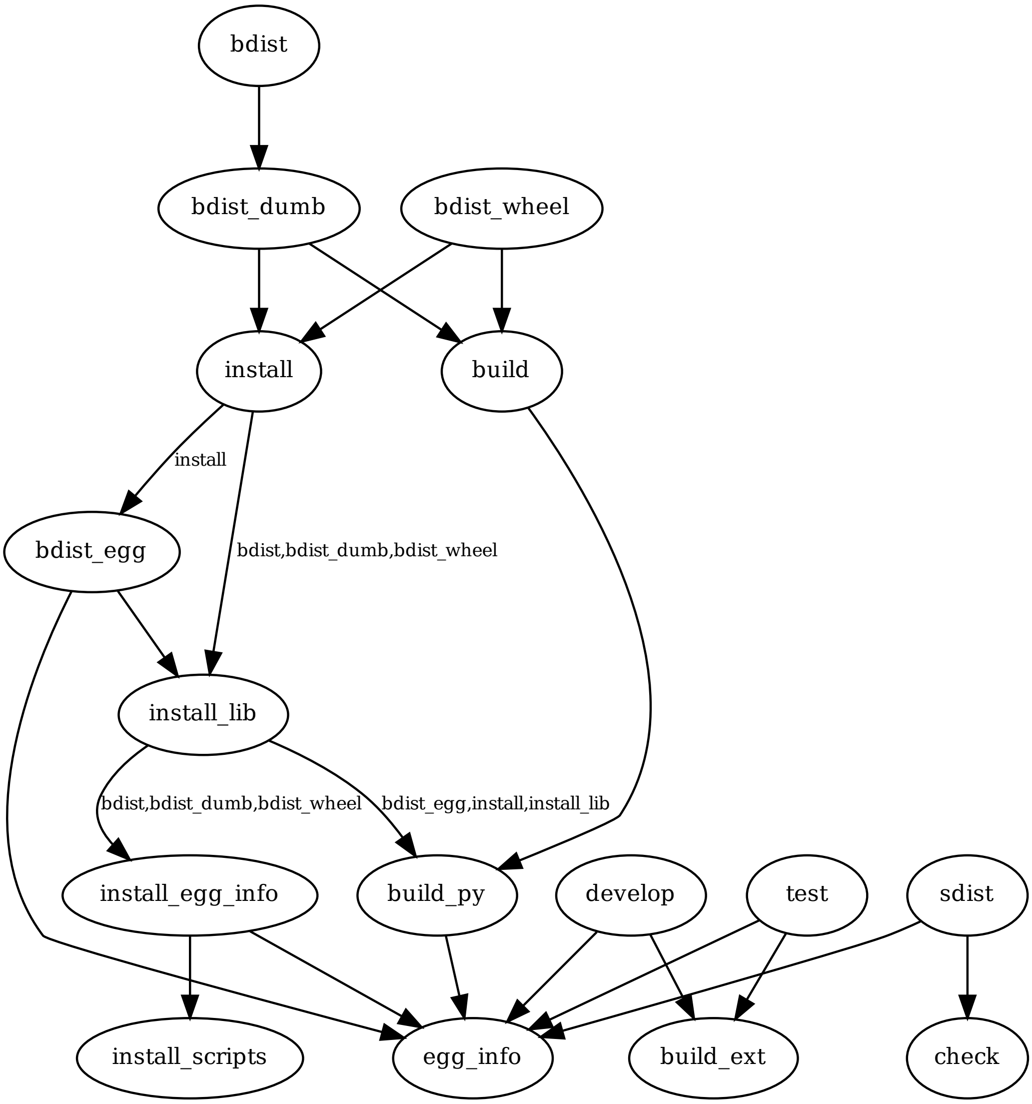
SETUPTOOLS

11.1 Tests

Place the tests in the `test` directory. Per default `setuptools` adds the `test` directory to the source distribution `sdist`. This can be disabled in the `MANIFEST.in`.

11.2 Commands dependencies

Graph showing the dependencies between the common `setuptools` commands:



11.3 Extend install command

> **Warning:** This is a work in progress that needs to be improved on.

This shows how to add a subcommand to the `install` command. This also shows how the subcommand can add to the list of files to be installed (packaged in a `bdist`).

```
class install_something(setuptools.Command):
    user_options = [
        ('install-dir=', 'd', "directory to install to"),
    ]
    def initialize_options(self):
        self.install_dir = None
```

(continues on next page)

(continued from previous page)

```
    def finalize_options(self):
        self.outputs = []
        self.set_undefined_options(
            'install',
            ('install_lib', 'install_dir'),
        )
    def run(self):
        self.outputs.append('package/something.bin')
        self.mkpath(self.install_dir + 'package')
        self.copy_file(
            'src/package/something.bin',
            self.install_dir + 'package/something.bin',
        )
    def get_outputs(self):
        return self.outputs


class install(distutils.command.install.install):
    _sub_command = (
        'install_something',
        None,
    )
    _sub_commands = distutils.command.install.install.sub_commands
    sub_commands = [_sub_command] + _sub_commands
```

CHAPTER

TWELVE

CHAMELEON

12.1 Introduction

- https://pypi.org/project/Chameleon/
- https://chameleon.readthedocs.io/

12.2 Macros

12.2.1 Omit tag

Tags from the namespace `tal` and `metal` are omitted. But no specific tag name is required. So use something like this

```
<metal: metal:something="whatever">...</metal:>
<tal: tal:something="whatever">...</tal:>
```

12.2.2 Same file

Use macro from the same template (same file).

The macros are available under `template.macros` or directly under `macros`.

```
<metal: metal:define-macro="ping">pong</metal:>

<metal: metal:use-macro="template.macros['ping']"></metal:>
<metal: metal:use-macro="macros['ping']"></metal:>
```

12.3 I18N

12.3.1 Babel

According to its documentation `chameleon` should provide a message extractor for `Babel`, but it is not actually the case.

https://github.com/malthe/chameleon/issues/12

Use `lingua` instead. It has a message extractor for `chameleon`.

12.3.2 lingua

Even though `lingua` claims in its documentation to always extract messages that do not have a domain, it is not the case for the `chameleon` extractor.

Make sure to always specify a `domain` in the `.pt` file. Otherwise the messages won't be extracted by `pot-create`.

```
<tal: i18n:domain="MyDomain">
    <!-- ... -->
    <span i18n:translate="">message</span>
    <!-- ... -->
</tal:>
```

CHAPTER

THIRTEEN

WORKING WITH PYTHON

13.1 No *pip*

Do not install a global system-wide version of *pip* at all.

There is almost never a good reason to install global system-wide packages via *pip* to begin with. Especially on Linux where the default version of Python is part of the system and used by the system. So mixing this with Python projects that the user install install themselves via *pip* is very likely to cause conflicts sooner rather than later.

13.2 Use isolation

If Python tools are needed to be always available from the command line, then isolate them with *zapp*, *shiv*, or *pex*.

- *zapp* https://pypi.org/project/zapp/
- *shiv* https://pypi.org/project/shiv/
- *pex* https://pypi.org/project/pex/

Those are all *zipapp* single-file Python executables.

- https://www.python.org/dev/peps/pep-0441/
- https://docs.python.org/3/library/zipapp.html

shiv and *pex* applications are self extractable. *zapp* does not need to be extracted. The code is executed directly from within the zip-compressed archive.

pex applications are executed from their own virtual environment. *zapp* applications are not executed in a virtual environment. Not sure about *shiv*.

shiv applications show up somehow in the current environment. Whereas *zapp* applications do not, so they are perfect for tools such as *deptree*, and *pipdeptree*.

13.3 Use *toolmaker*

To automate the creation of single file Python applications with *zapp*, *shiv*, or *pex*, one can use *toolmaker*.

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

13.4 Use *venv*

Python 3 has the module *venv* in its standard library since version 3.3.

- https://docs.python.org/3/library/venv.html

So the need for the third party library *virtualenv* is much less pressing.

```
$ python3 -m venv .venv
$ . .venv/bin/activate
```

13.5 Do not activate virtual environments

The scripts that are installed in a virtual environment (with *setuptools* at least) get a shebang with the full path to the Python interpeter from the virtual environment. So there is no need to activate the virtual environment to call such scripts.

```
$ .venv/bin/myscript
$ .venv/bin/python3 -m mymodule
```

13.6 Interactive debug

- https://docs.python.org/3/library/functions.html#breakpoint

```
breakpoint()
```

- https://docs.python.org/3/using/cmdline.html#cmdoption-i

```
python -i main.py

python -i -m something
```

- https://stackoverflow.com/a/1396386/11138259

```
import pdb; pdb.set_trace()
```

Then:

- https://docs.python.org/3/library/pdb.html#pdbcommand-interact

```
(Pdb) interact
*interactive*
>>>
```

Or:

- https://docs.python.org/3/library/code.html#code.interact

```
import code; code.interact(local=locals())
```

CHAPTER

FOURTEEN

FIZZ BUZZ

Toy implementation of the *Fizz buzz* game.

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

class Injector:

    def __init__(self, multiple, word):
        self._multiple = multiple
        self._output = '{}!'.format(word)

    def __call__(self, value):
        result = None
        if value % self._multiple == 0:
            result = self._output
        return result

def fizz_buzz(start, end):
    injectors = [
        Injector(3, 'Fizz'),
        Injector(5, 'Buzz'),
    ]
    #
    for i in range(start, end + 1):
        items = []
        output = None
        #
        for injector in injectors:
            item = injector(i)
            if item:
                items.append(item)
        #
        if items:
            output = ' '.join(items)
        else:
            output = str(i)
        #
        print(output)

def main():
    fizz_buzz(1, 50)

if __name__ == '__main__':
    main()

# EOF
```

Part III

Docker

CHAPTER

FIFTEEN

PRESENTATION

First public presentation of Docker, *The future of Linux Containers*: https://www.youtube.com/watch?v=wW9CAH9nSLs

Official website: https://www.docker.com/

CHAPTER

SIXTEEN

TIPS

16.1 Playground

Play with Docker in the web browser: https://labs.play-with-docker.com/

Part IV

Miscellaneous

CHAPTER

SEVENTEEN

HTML5

17.1 Sectioning

```
<!DOCTYPE html>
<html lang="en">
 <head>
  <title>Title</title>
 </head>
 <body>
  <main>
   <h1>Title</h1>
   <article>
    <h2>Section</h2>
    <section>
     <h3>Subsection</h3>
     <p>Content</p>
    </section>
   </article>
  </main>
 </body>
</html>
```

Use following link to validate: https://validator.w3.org/nu/?showoutline=yes

17.2 Minimal document

Shortest valid HTML5 document:

```
<!DOCTYPE html><title>x</title>
```

CHAPTER

EIGHTEEN

MAKEFILE

18.1 Links

- https://www.gnu.org/software/make/manual/make.html
- http://clarkgrubb.com/makefile-style-guide
- http://gromnitsky.users.sourceforge.net/articles/notes-for-new-make-users/

18.2 Example

```
input_dir := input
output_dir := output

input_files := $(wildcard $(input_dir)/*.in)
output_files := $(patsubst $(input_dir)/%.in,$(output_dir)/%.out,$(input_files))

vpath %.in $(input_dir)

.DEFAULT_GOAL := all

.PHONY: all
all: $(output_files)

$(output_dir)/%.out: %.in | $(output_dir)
	cp $< $@

$(output_dir):
	mkdir --parent $@

.PHONY: clean
clean:
	$(RM) $(output_files)

# Disable default rules and suffixes
# (improve speed and avoid unexpected behaviour)
MAKEFLAGS := --no-builtin-rules
.SUFFIXES:
```

CHAPTER

NINETEEN

NPM

19.1 Packages in home directory

This will let `npm` use a custom directory for globally installed package.

Listing 1: ~/.profile

```
# ...
export NPM_PACKAGES="${HOME}/.npm_packages"
PATH="${NPM_PACKAGES}/bin:${PATH}"
NODE_PATH="${NPM_PACKAGES}/lib/node_modules:${PATH}"
# ...
```

Listing 2: ~/.npmrc

```
# ...
prefix = "${NPM_PACKAGES}"
# ...
```

Listing 3: shell interactive console

```
$ . ~/.profile
$ npm install --global npm
```

CHAPTER

TWENTY

SHELL

Create a temporary directory and change to it:

```
$ cd ($mktemp --directory)
$ cd ($mktemp -d)
```

List directories by disk usage:

```
$ du --human-readable | sort --human-numeric-sort --reverse | less
$ du -h | sort -hr | less
```

```
$ sudo du --all --human-readable --max-depth=1 / 2>/dev/null | sort --human-numeric-
↪sort --reverse
$ sudo du -a -d 1 -h / 2>/dev/null | sort -hr
```

Part V

Appendix

CHAPTER

TWENTYONE

ABOUT

21.1 Introduction

Written in reStructuredText[13] and built with Sphinx[14].

21.1.1 Mirrors

- https://sinoroc.gitlab.io/kb/
- https://sinoroc.github.io/kb/

21.2 Hacking

21.2.1 Repositories

- https://gitlab.com/sinoroc/kb
- https://github.com/sinoroc/kb

21.2.2 Style guide

Use the following for section headings:

- # with overline, for parts
- * with overline, for chapters
- =, for sections
- -, for subsections
- ^, for subsubsections
- ", for paragraphs

Suggestion taken from Python Developer's Guide[15].

[13] http://docutils.sourceforge.net/rst.html
[14] http://www.sphinx-doc.org/en/stable/index.html
[15] https://devguide.python.org/documentation/markup/#sections

CHAPTER

TWENTYTWO

LICENSE