# Reversing Python Packaging

anatoly techtonik <techtonik@gmail.com>

# Reversing Python Packaging

anatoly techtonik <techtonik@gmail.com>

This book is for sale at http://leanpub.com/reversingpythonpackaging

This version was published on 2015-11-03


This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

*Dedicated to people who hacked on Python packaging challenges through all these years.*

# Contents

# Intro

For many years Python had ability to install modules and packages, and for many years it was impossible to do uninstalls. Now this ability is provided by external tool called `pip`, but why is it needed and what happens behind the scenes?

At the end of this book, you will likely to get familiar with established practice in packaging internals, might get a better understanding what a Python package is, what types of packages exist, and maybe even how to distribute Python code for various operating systems.

The book is written in parallel with "researching" problems behind packaging in my free time. If you find this stuff useful, you may support the activity through https://gratipay.com/techtonik/ In future, we may add an ability to add specific targets (goals or causes) to Gratipay so that you can specifically "thank" for the existence particular book.

# An important note about `package` confusion

In general, **a package** is software packed in a special way for some purpose. In Python `a package` is **a directory** that groups multiple Python files together under one importable namespace. Such directories have `__init__.py` file inside to be recognized by Python interpreter as **importable package**.

Archives with code and binaries are called **distributions** in Python world, because historically the `package` word was already taken. But people from Linux world are used to refer to such archives as `packages`. That increases chances to make any explanations involving both `package` concepts extremely confusing, so keep note on the **context** where the word `package` is used to not get lost.

Funny that even in Python word people rarely use the word `distribution` to refer to packaged archive. Even official distribution site - **PyPI** is an abbreviation of `the Python Package Index`. I can guess that that's because Python interpreter itself could be distributed in different ways on different operating systems, and that is what would be called `a distribution`.

Official Python docs mention all these definitions, but that's didn't get too far, because we can't rely that people read the docs these days. This book may repeat some details from that docs, but adds more details on what's going on in 2015. Still, to make the concepts more clear, the book tries to follow this terminology:

- **Python package** - directory that Python interpreter can import
- **package** - archive
- **source package** - archive with Python source code
- **binary package** - archive with compiled extensions
- **extension** - .dll or .pyd module from non-Python sources

There were more attempts to improve packaging and that gave birth to new package formats called `eggs` and `wheels` (more on them in later chapters).

- **egg** - package format used by setuptools tools
- **wheel** - package format to install by unpacking

Both **wheel** and **egg** formats may carry binary stuff.

# Where Python looks for modules?

When you invoke `import something`, Python looks for either `something.py` or `something/__init__.py`,

If these files are not shipped with Python (in that case they are assumed to be a part of `Python standard library` sometimes called `stdlib`), then these are *user modules*, and a place where Python expects to find them is called `site-packages`. This location of site-packages varies for each operating system. For example, on Windows it would be something like:

```
C:\Python34\Lib\site-packages
```

To get this location programatically, you may use site module[1] shipped with Python.

```
python -c "import site; print(site.USER_SITE)"
```

On Ubuntu it shows:

```
/home/techtonik/.local/lib/python2.7/site-packages
```

On my machine that directory doesn't even exist. How come? The reason is that there is **global site-packages** dir and **user level** one. `site.USER_SITE` was added in Python 2.6 and points to the latter. Python packaging tools prefer to install everything globally, which is not always a good idea. There is rarely a need for me to install module for someone else, so even if it seems like nobody is using `site.USER_SITE`, it may worth to use it. Just for the reference, on Windows it will be:

```
C:\Users\techtonik\AppData\Roaming\Python\Python27\site-packages
```

Browsing site module[2] docs will give you more paths to look for, but `site.USER_SITE` is probably the only one that is writable.

Now, if you installed your Python package, egg, wheel or whatever you at least know where to look for your files.

So let's see how it works, reinvent the packaging from scratch and look where things are different and why they went wrong.

---

[1] https://docs.python.org/2/library/site.html

[2] https://docs.python.org/2/library/site.html

# Installing Python module by hand

The procedure is simple:

1. Find `site-packages`
2. Copy your module there
3. Check that it works

First ensure that your code is not importable. Let's use non-existing module called `tester.py`:

```
> python -c "import tester"
Traceback (most recent call last):
  File "<string>", line 1, in <module>
ImportError: No module named tester
```

Now with code stolen from previous chapter, figure out the location of `site-packages` and move `tester.py` there. I'll be using Linux:

```
$ python -c "import site; print(site.USER_SITE)"
/home/techtonik/.local/lib/python2.7/site-packages
$ echo "print('imported successfully')" > tester.py
$ mv .py /home/techtonik/.local/lib/python2.7/site-packages
mv: cannot move ‘tester.py’ to ‘/home/techtonik/.local/lib/python2.7/site-packag\
es’: No such file or directory
```

Right, Python installation on Linux doesn't create `site-packages` dir on Linux, because Linux people assume that everything should be installed by root to ensure maximum security, but that is inconvenient! So, let's create dir for user installed files:

```
$ mkdir -p /home/techtonik/.local/lib/python2.7/site-packages
$ mv tester.py /home/techtonik/.local/lib/python2.7/site-packages
$ python -c "import tester"
imported successfully
```

Just in case you wonder, `mkdir -p` creates all parent dirs if they are missing. That's it. The only thing that is left is to rewrite the code above as a Python script using site module[3], shutil[4] and os[5].

[3] https://docs.python.org/2/library/site.html
[4] https://docs.python.org/2/library/shutil.html
[5] https://docs.python.org/2/library/os.html

# What is a Python source package?

Trying to reverse that. An informal official definition:

> Python source package is an archive in OS system specific archive format (.tar.gz or .zip) produced by **setup.py sdist** command.

That definition is not sufficient to understand what exactly the package contains and how it can be used. The only thing that is evident is that nothing except 'sdist' command can produce such package.

A more useful non-official definition could be:

> Packed Python sources that can be installed on target system and/or compiled into binary package format.

Or alternative:

> Archive with packed pure Python code that can be installed by pip from PyPI.

So if the can be installed and contains the full source of the Python application/module, that is a **source package** (if it looks like a duck and quacks like a duck, then it is a duck).

## Reversing the Python source distribution format

Using alternative definition from above, it is archive with packed pure Python code. What is in the archive exactly?

I took the existing package named hexdump-3.2.zip and modified it to see what is possible and what is not. It appeared that the following could be installed using pip. This test was done with pip 1.5.4 in April 2015. Thing may break in future if PyPA team decides to "fix the issue".

- [x] .zip with setup.py at root (hexdump-3.2.zip)
- [x] .zip with setup.py at root (renamed to blabla.zip)
- [x] .zip with setup.py in arbitrary directory inside of archive (must contain exactly one directory at root level and nothing else)

So, the setup.py file looks important, let's see how it works. Hopefully, this won't require learning any Python code at all. That's the point of whole point of reversing - enjoy the fun of learning stuff in non-standard way.

## Researching setup.py behaviour

I upgraded pip to 6.1.1 (latest released version as of April 2015) to avoid good old bugs (and deal only with new bad ones).

Removing setup.py from archive and trying to install this .zip produced this error.

```
$ pip install blabla.zip
Processing c:\leanbook\manuscript\blabla.zip
    Complete output from command python setup.py egg_info:
    Traceback (most recent call last):
      File "<string>", line 18, in <module>
      File "C:\Python34\lib\tokenize.py", line 437, in open
        buffer = builtins.open(filename, 'rb')
    FileNotFoundError: [Errno 2] No such file or directory:
'C:\\Temp\\pip-7rc60spl-build\\setup.py'

    ----------------------------------------
    Command "python setup.py egg_info" failed with error code 1
in C:\Temp\pip-7rc60spl-build
```

`pip` tried to execute it with `egg_info` command. This is strange, because setup.py is described in official Python docs, but there is no `egg_info`, so it must be some internal `pip` hack.

Let's create an empty *setup.py* to see how it behaves:

```
$ pip install blabla.zip
No files/directories in C:\Temp\pip-82yc9aep-build\pip-egg-info
(from PKG-INFO)
```

This is mystic, but gives a hint that some required PKG-INFO. Looking into former hexdump-3.2.zip before modification, there is a PKG-INFO file with a lot of content. Placing empty PKG-INFO together with empty setup.py gives the same error as above. Actually, the PKG-INFO doesn't matter at all - even complete file still gives the misleading error.

So, getting back to the setup.py, after some experiments, this is the minimal magic incantation that should be present to get something installed:

```
from distutils.core import setup
setup(
    py_modules=['hexdump'],
)
```

With such file pip will install hexdump as UNKNOWN-0.0.0 package. The optimal content looks like this:

```
from distutils.core import setup

setup(
    name='hexdump',
    version='3.2',
    author='anatoly techtonik <techtonik@gmail.com>',
    url='https://bitbucket.org/techtonik/hexdump/',

    description="view/edit your binary with any text editor",
    license="Public Domain",

    py_modules=['hexdump'],
)
```

This is weird. There is no algorithm to pack a Python module into package, but a magic file content. I am not even sure how to convert that into Debian or Fedora package, or how to analyse the license field from all PyPI packages without running all setup.py (which is not secure) on all of them.

This setup.py looks like an awkward and [non-intuitive] (https://stackoverflow.com/questions/1471994/what-is-setup-py) legacy from the old times, but there is no alternative to setup.py from what I know.

## Uploading to PyPI - required PKG-INFO

The second part of alternative definition from above says that archive should be installable from PyPI. The problem arised when I tried to upload the .zip without PKG-INFO. It failed with `invalid distribution file` error.

The validation made at https://bitbucket.org/pypa/pypi/src/8efd5f92/verify_filetype.py?at=default#cl-44 and requires that PKG-INFO is present in either root of archive or in subdirectory (os.path.split returns 2 element array even for filename without path). Uploading .zip file with empty PKG-INFO worked.

# So, what is a source package?

In terminology chapter two primary things were described:

- importable Python package - a directory with **init**.py
- archive with Python source or compiled extensions

Python 2.7.9 docs are also distinct about **pure Python package** or **source package** and **binary distribution**. This is clearly inspired by Linux where you have this distinction - software in source form and software in compiled form all need to be packed for distribution.

There is somewhat vague definition of **source package** as archive produced by `setup.py sdist`. A more appropriate definition is that **source package is** a .tar.gz or .zip file with subdirectory named after the archive and `setup.py` inside of it. It is not clear why this extra directory is necessary, becuase `pip` is able to install archive even without it.

# Anatomy of a wheel

**Python wheel** is a .zip file that promises to contain all files "in a way that is very close to the on-disk format".

The structure of a wheel archive is described at https://www.python.org/dev/peps/pep-0427/#installing-a-wheel-distribution-1-0-py32-none-any-whl

But it is not clear enough. In summary, the obligatory contents of a wheel:

```
distribution-1.0.dist-info/    - package description directory
  METADATA                     - package content description
  RECORD                       - file list
  WHEEL                        - additional package description
```

Creating .whl without METADATA gives a fatal error while installing with pip 7.1.2. With empty METADATA it exits with message `distribution is in an unsupported or invalid wheel`. So METADATA alone is not enough, you need to create additional file WHEEL where pip expects to find at least:

```
Wheel-Version: 0.1
```

But that is not enough either. The RECORD file is obligatory as well. So an archive named something-0.5.0-py2-none-any.whl with empty METADATA and RECORD and a single line WHEEL file is a valid **wheel** file that can be successfully installed and uninstalled with **pip**.

Strange that package requires both METADATA and WHEEL to describe it. One of those files should perish, as for me. Looks like METADATA is not extensive enough to include extra fields. From the other side all its info could be present in a WHEEL file. Anyway.. Let's see if that wheel will be accepted by PyPI.

## Uploading wheel to PyPI

First, PyPI can't just take .whl, parse it and merge metadata from there. That's a pity. So you will need to create package `something` manually by submitting the form with required fields online. After that you can upload that crippled wheel without any problem.

## Adding contents

Handling empty wheels is so much fun, but let's make it better. Just add some files into .whl and they will get magically unpacked into Python's `site-packages` directory on installing. That's it.

Clearly the best packaging format.

## Platform specific code

Sometimes you need to ship compiled C extensions. These come as DLLs in Windows or SO files in Linux. Both can be 32 or 64 bits. For that, wheels require to specify target platform in filename.

```
somethings-1.0.1-py2-none-win32.whl
      ^^^^                          - name
            ^^^^                    - version
                 ^^^                - Python version (can be py2.py3)
                     ^^^^           - mystic ABI
                           ^^^^     - platform
```

The ABI is relevant to C extension writers. Platform can be `win32`, `linux_i386` and `linux_x86_64`.

So compile your stuff, create yourstuff-version.dist-info/ directory with three files, and add stuff to the root of the archive. Name it appropriately for your platform / Python version and you're all set. No need for complicated `setup.py` dances.

## Bonus

For the test, I added `emptywheel-1.0.1-py2-none-any.whl` to the data/ directory in the source code of this book. Try to install it and locate `find me` in your `site-packages`.

I also include `testdata-wheel-reversing.zip` with the wheels I used to test `pip` behaviour.

And for the last thing, I include `create-record.py` script that can be used to create RECORD from an existing directory heirarchy. It is useless for the time being, but may become handy in future if `pip` gets more strict checks for unpacking the wheels.

# Reasons why Python packaging suxx

Python docs are good at explaining the magic rituals, but apparently they are insufficient to explain *why* the rituals were born on the first place. Like with many solutions that were born in evolutionary ways there are some atavisms and things preserved for backward compatibility that make the whole system look awkward.

Here are some reason why this happens and what to do about that.

1. Technical

That one is obvious. People write more code, cover more user stories, complexity increases, developers hit Norris' Number. Natural process of evolution.

Complexity of texts describing the system and code can be reduced with diagrams, visualizations and new media material that is more suitable for new people.

1. Social

People think differently and use computers differently, but not everybody is aware of the details. Something that "works for me" for those who have commit rights maybe a major showstopper for other, less experienced people.

So, there is a gap between people who already know how the stuff works, and people who don't know about it. The stuff is even worse because of `curse of knowledge` where people with experience don't realise how hard it is to be without that system knowledge.

And without that experience it is hard to get through the complexity of details to fix things, even with a fair amount of work being made to document the stuff (including this book). Making more experienced people aware of the `curse of knowledge` concept will make people consider spending more time on building more accessible materials.

1. Economical

New generation of people is raised to consume information in small chunks in visual way. They prefer the way of engaging experiments rather than taking their time to read through long texts. Producing visual media requires a lot of time that no company will pay for, and a lot of thinking that no company will wait for.

There are a lot of designers, people interested to play with visualization and making Python a better tool, but only few of a thousands are capable of hacking the economy to free themselves for doing

this. Why? Because current economy discourages to spend time on stuff that is not profitable. And the less free resource we have, the less money is generated to pay bills, loans and buy out time for doing good things. Making Python better is a good thing, but in consumption economy there is no laws to compensate time for people who share what they know for free. There are more adaptable economic models to support that, but they require initial seed of investments to build up a protection scheme so that bigger economy won't consume the new one.

In the meanwhile, what could be done? Supporing the mission behind Gratipay and thinking about how to compensate the open source value in real world.