Publishing (Perfect) Python Packages On PyPI

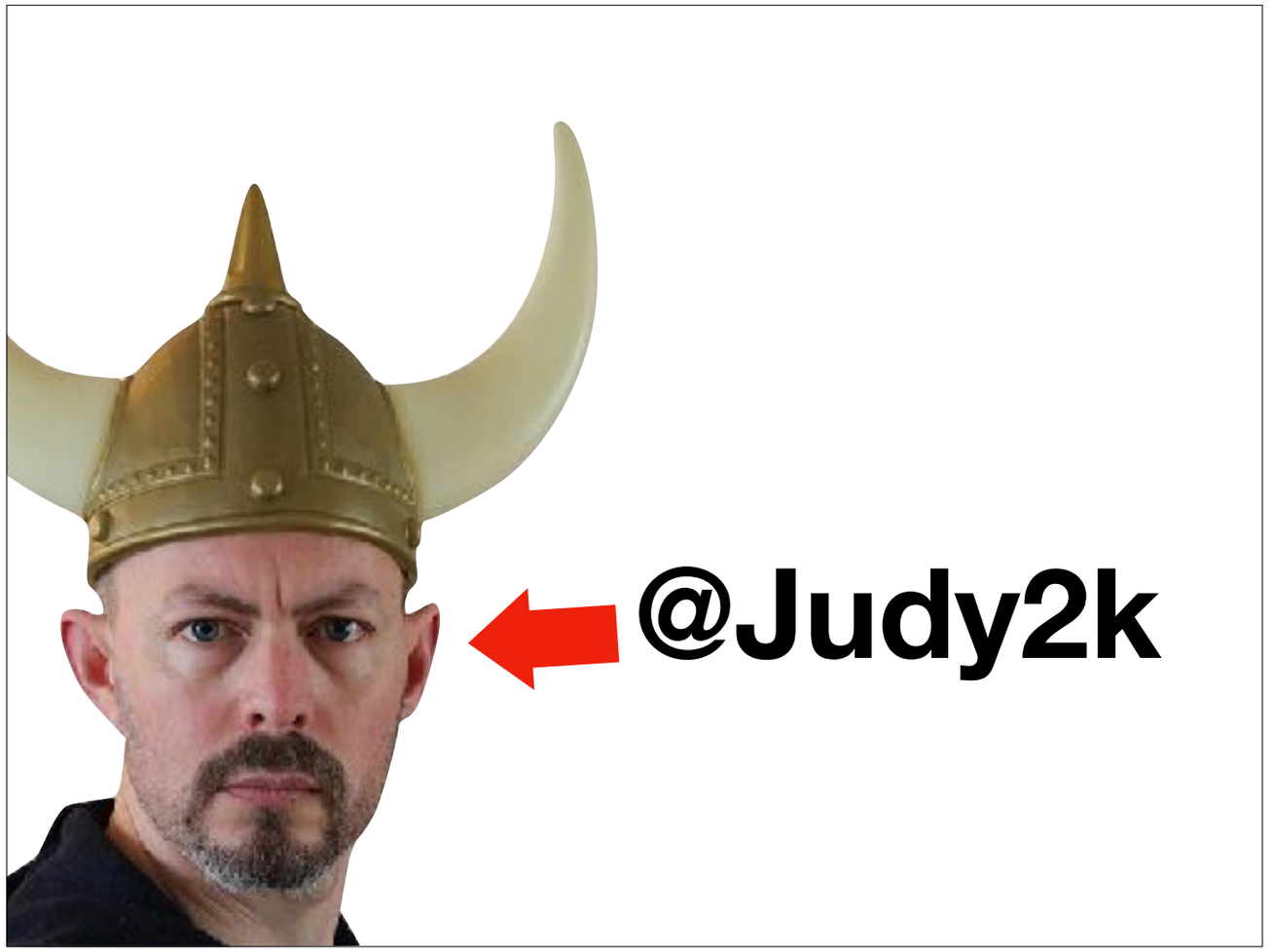
Mark Smith
Nexmo

Publishing (Perfect) Python Packages On PyPI

Mark Smith
Nexmo

Changed talk title. But first, let me introduce myself.

I am Judy2k on Twitter, GitHub and pretty much everything else.



I am Judy2k on Twitter, GitHub and pretty much everything else.

Real name is Mark Smith, and I'm a Developer Advocate for Nexmo.

I stole this beautiful slide from my colleague Aaron.
Nexmo provides **web-based APIs** that allow web developers to write code that **sends text messages and makes phone calls, or send messages via different instant messaging platforms**.
If any of that sounds interesting, please come and talk to me afterwards, and I may be able to give you some free credit to try it out.
That's enough about me.

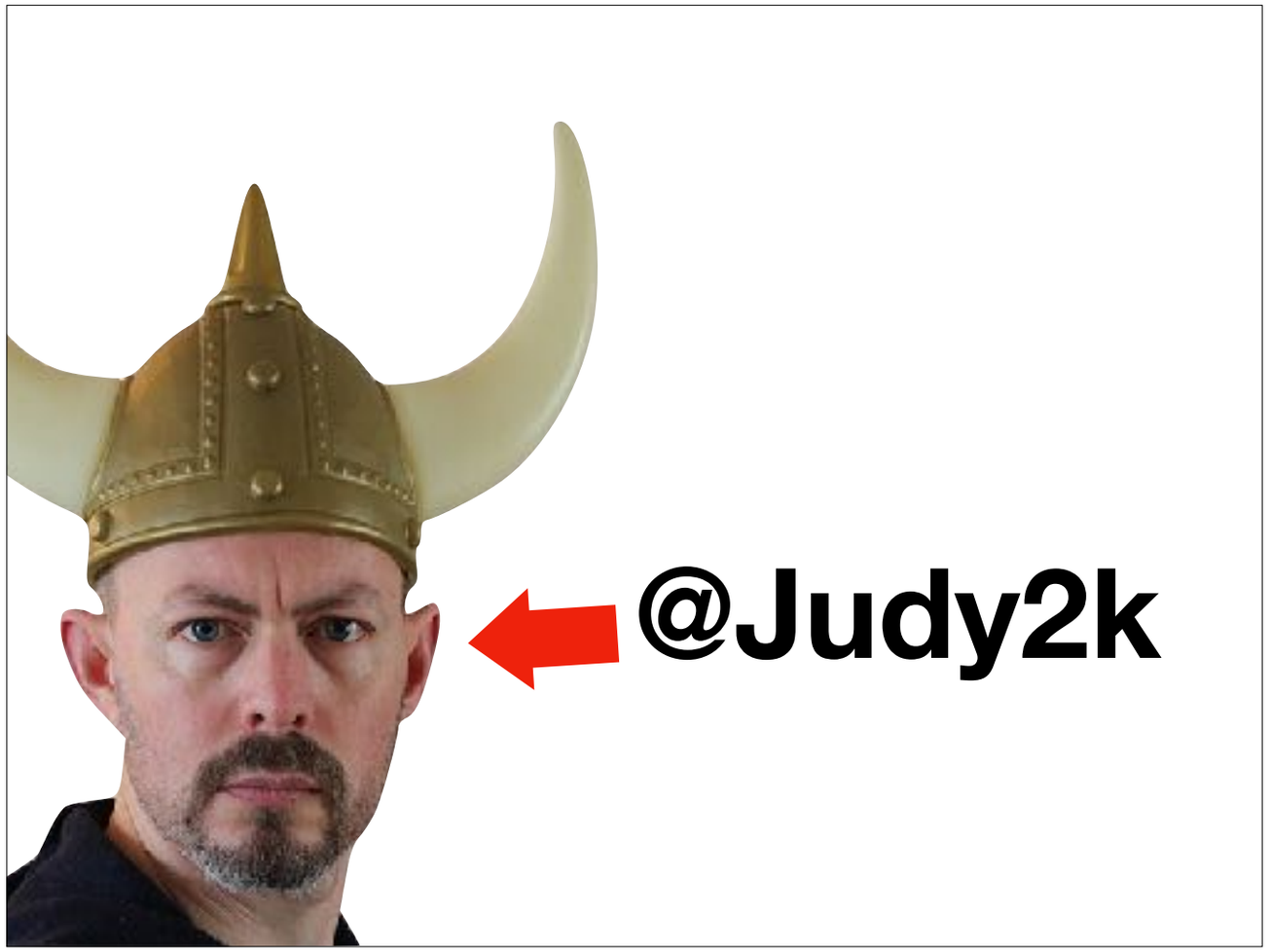
THE left-pad PROBLEM

In March 2016 a developer removed a library called left-pad from npm, the NodeJS equivalent of PyPI, a big web service containing packages. It broke lots and lots of libraries that depended on it. Left Pad was downloaded **2,486,696** times in the month before it was removed. It was just **one function**, and **11 lines of code**. That padded a string to a certain length by adding characters to the start.

Lots of people thought this made the Javascript community look stupid. Why would anyone publish a library consisting of just 11 lines of code? Why would anyone use it? Now the fact that it could be removed like this is a problem but,

I think it made the Javascript community (esp. NPM) look **amazing**. (If you disagree, you can fight me on Twitter.

Refs: https://www.theregister.co.uk/2016/03/23/npm_left_pad_chaos/



If you disagree, you can fight me on Twitter.

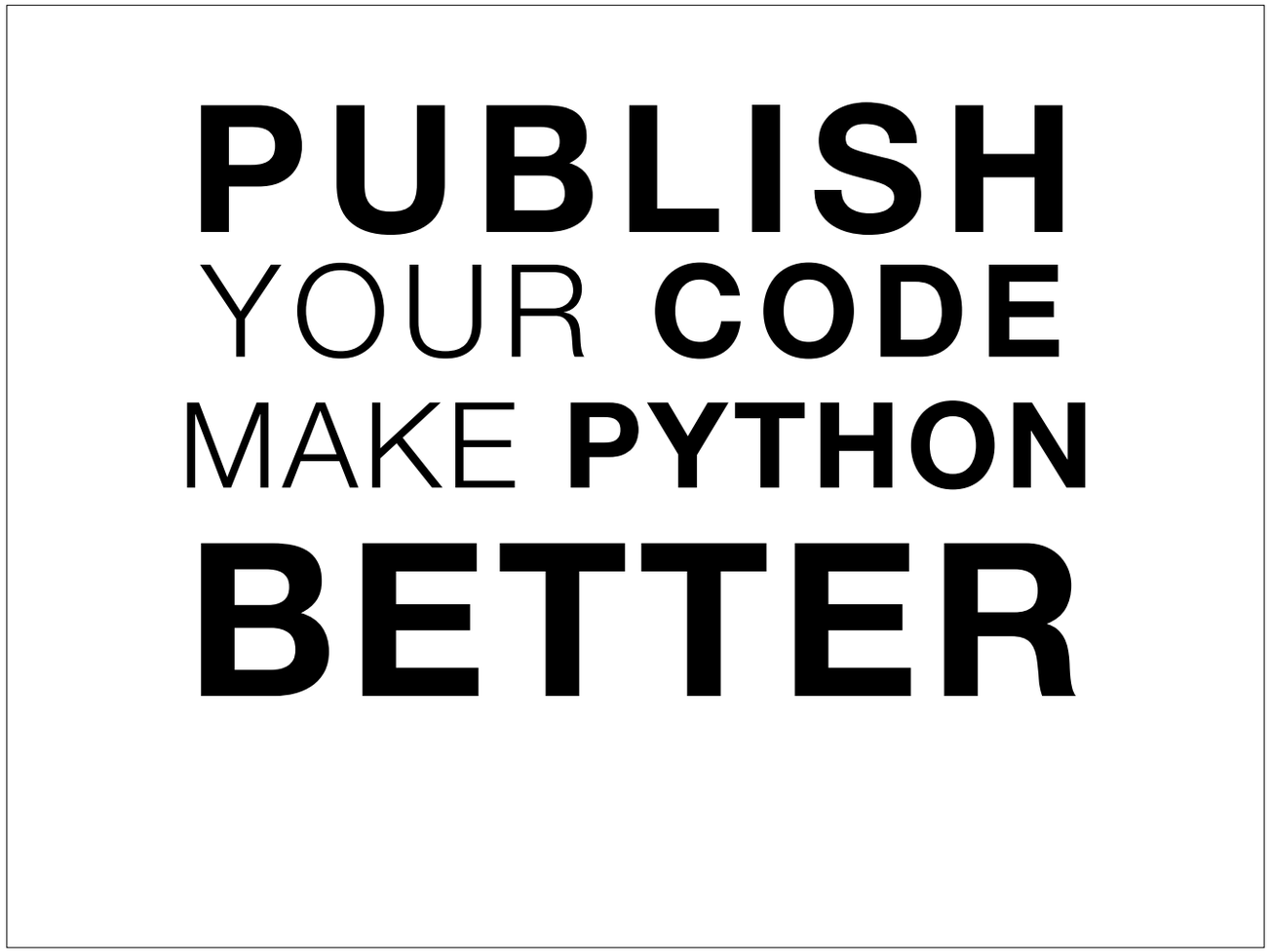
THE left-pad PROBLEM

In my opinion, Left-pad was not a problem. [click]

left-pad was the **SOLUTION**

Left-pad was a solution to a problem. Because the Javascript standard library doesn’t contain a solution for adding whitespace to the start of a string, the developer of left-pad solved the problem himself, and published the solution.

Because it’s really easy to share code with NPM, people do - with really small libraries - the type of thing you'd normally find on StackOverflow or in a Gist. But Copy & Paste is not how you should share code!
I feel people are afraid of setup.py. The docs and best practice have been a bit tricky to put together, but they’re getting better all the time, and hopefully this talk will help.



Every time you publish code, you make the python ecosystem stronger, you make someone’s life easier. Where would we be if Django, Numpy, Pandas had never been published?

Make A **Package**

```python
def say_hello(name=None):
    if name is None:
        return "Hello, World!"
    else:
        return f"Hello, {name}!"
```

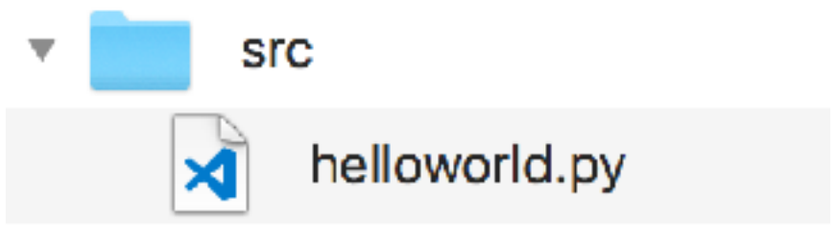
! You’ve written some code that you’re proud of, and you’d like to share it! [click]
Here extract it into a file called helloworld.py.

helloworld.py

```
def say_hello(name=None):
    if name is None:
        return "Hello, World!"
    else:
        return f"Hello, {name}!"
```

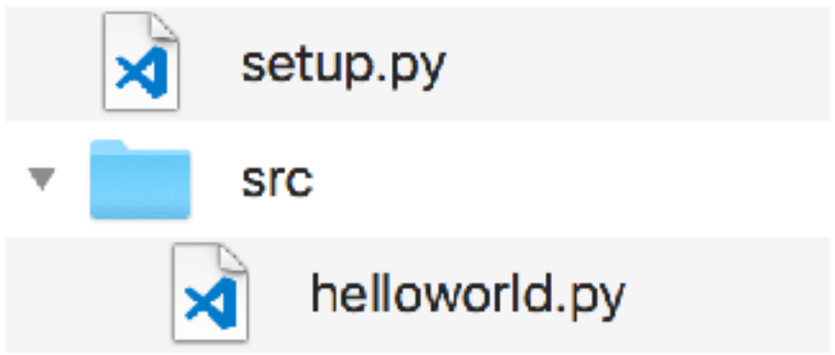
! You’ve written some code that you’re proud of, and you’d like to share it! [click]
Here extract it into a file called helloworld.py.

helloworld



We’re going to put our python code in a src directory. I’ll explain why later.
[click]
Now we’re going to write a setup.py file.

helloworld



We’re going to put our python code in a src directory. I’ll explain why later.
[click]
Now we’re going to write a setup.py file.

setup.py

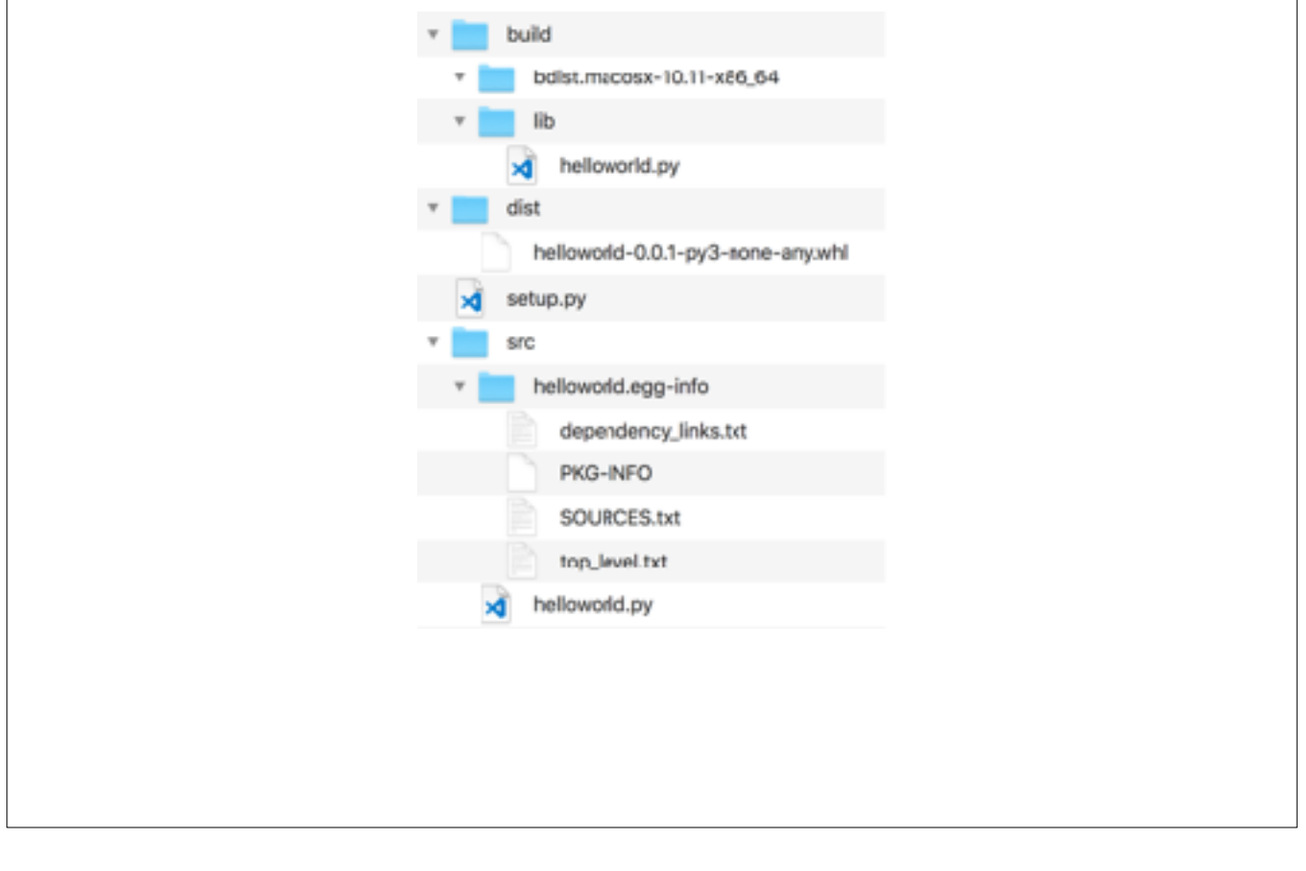
```
from setuptools import setup

setup(
    name='helloworld',
    version='0.0.1',
    description='Say hello!',
    py_modules=["helloworld"],
    package_dir={'': 'src'},
)
```

Note first, we're importing setuptools. Setuptools is a 3rd-party package, but it's also part of pip, so you already have it.
The rest is basically configuration.
name & the py_module will usually be the same. The first is what you pip install. The second is what you import. Version will usually be 0.0.1 for your first release. We need the last line because we've put our python code in a src directory.

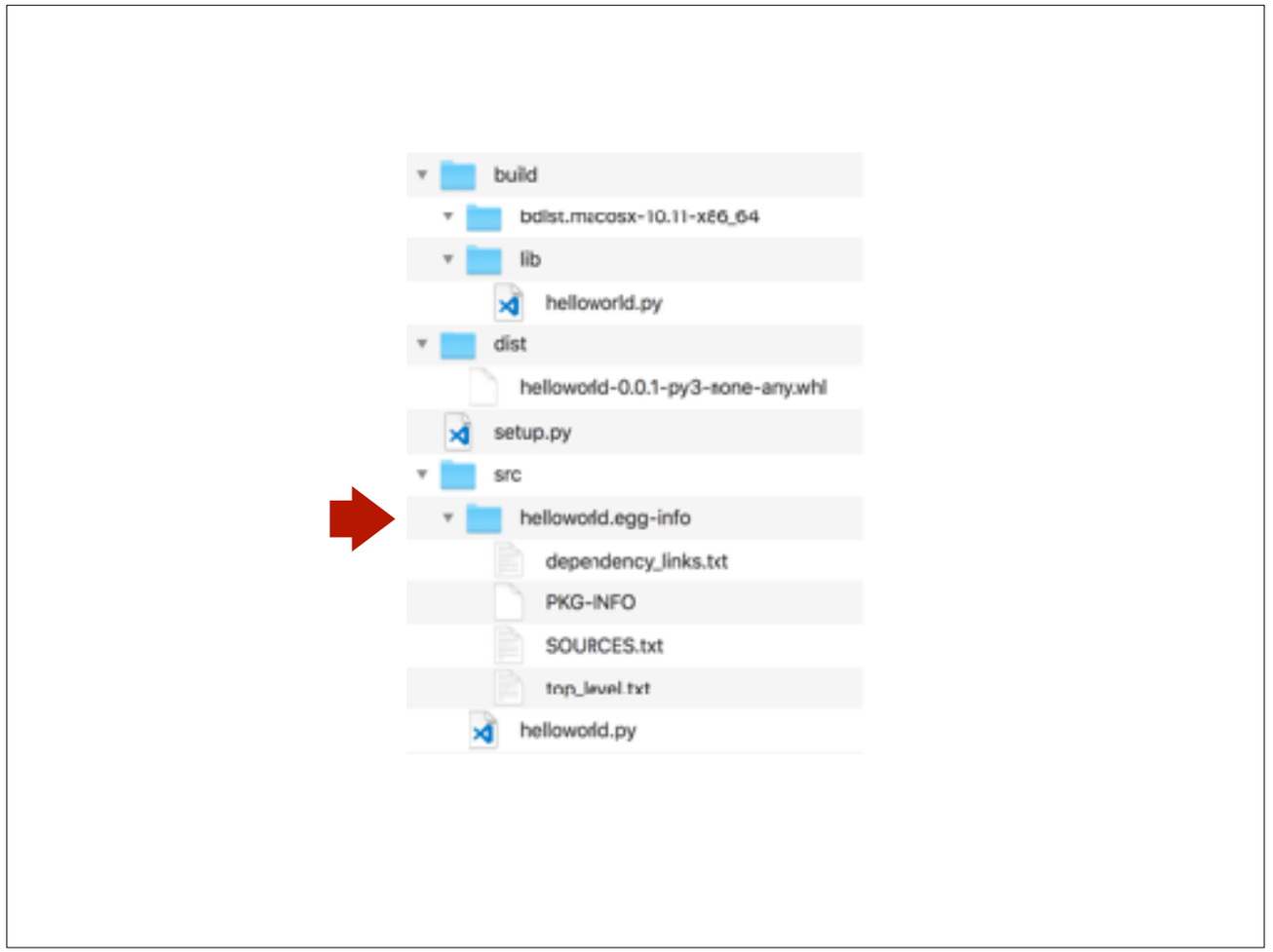
Let’s Test It!

```
$ python setup.py bdist_wheel
running bdist_wheel
…
copying src/helloworld.py -> build/lib
…
creating ‘/path/to/helloworld/dist/
helloworld-0.0.1-py3-none-any.whl’ and adding
'.' to it
removing build/bdist.macosx-10.11-x86_64/
wheel
```

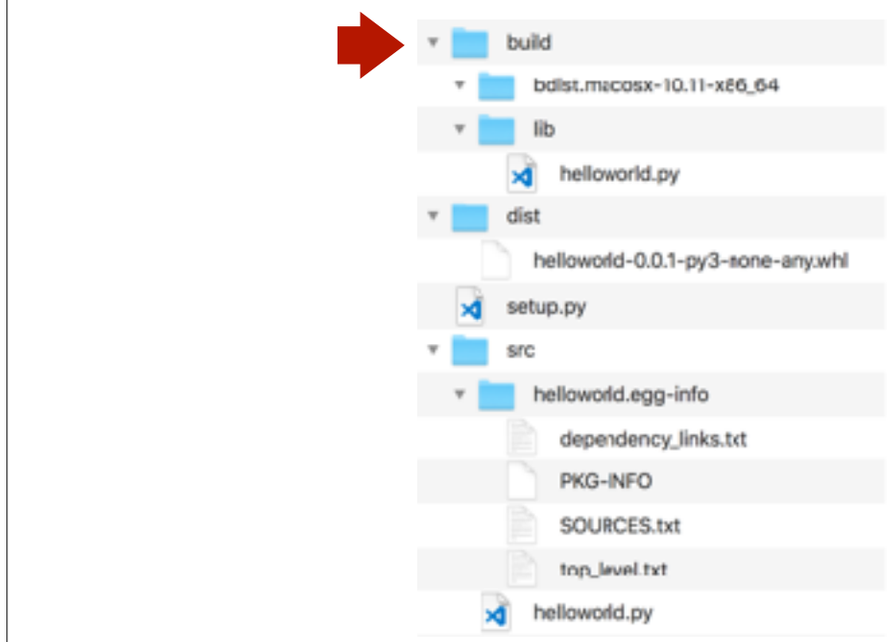
We’ve barely done anything, but already we have something we *could* publish!
In our terminal, if we run the setup file with bdist_wheel, which is short for “build me a wheel binary distribution file” - which is the standard file format for pip.
We get lots of output, but the line I’ve highlighted is the important one!



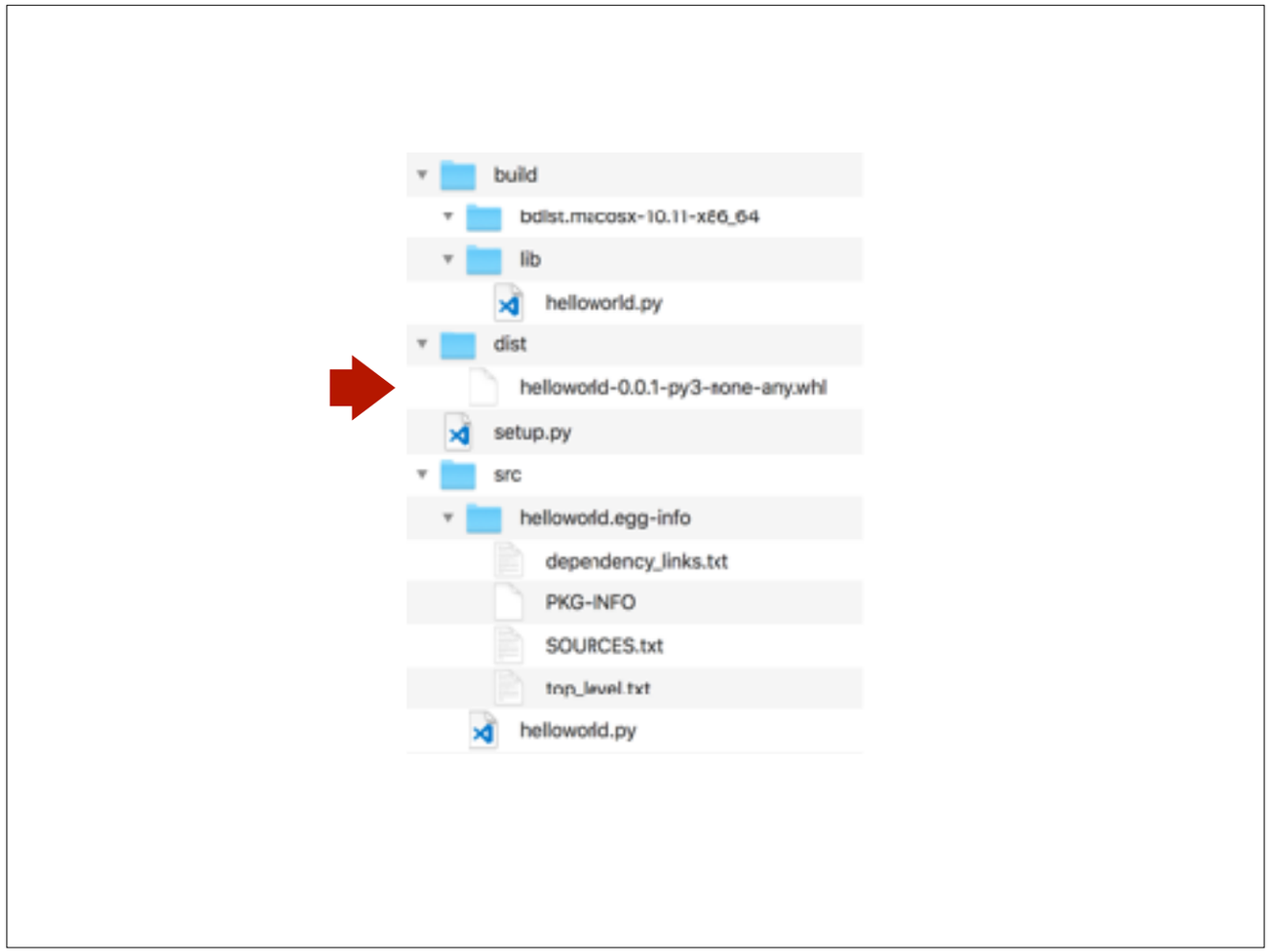
Look at all the new stuff! [click]
First, it's created an egg-info folder inside our source folder and filled it with some files. [click]
Then it's created a build directory and it's copied our code into it (that's good) [click]
Then it's zipped up some of this stuff into a wheel file, which is what we wanted! So we know it builds! We don't know if everything we need is in there. So let's install it.



Look at all the new stuff! [click]
First, it's created an egg-info folder inside our source folder and filled it with some files. [click]
Then it's created a build directory and it's copied our code into it (that's good) [click]
Then it's zipped up some of this stuff into a wheel file, which is what we wanted! So we know it builds! We don't know if everything we need is in there. So let's install it.



Look at all the new stuff! [click]
First, it's created an egg-info folder inside our source folder and filled it with some files. [click]
Then it's created a build directory and it's copied our code into it (that's good) [click]
Then it's zipped up some of this stuff into a wheel file, which is what we wanted! So we know it builds! We don't know if everything we need is in there. So let's install it.



Look at all the new stuff! [click]
First, it's created an egg-info folder inside our source folder and filled it with some files. [click]
Then it's created a build directory and it's copied our code into it (that's good) [click]
Then it's zipped up some of this stuff into a wheel file, which is what we wanted! So we know it builds! We don't know if everything we need is in there. So let's install it.

Install it locally

```
$ pip install -e .
Obtaining file:///path/to/helloworld
Installing collected packages: helloworld
  Running setup.py develop for helloworld
Successfully installed helloworld
```

You run this command in your **virtual environment**, in the folder that has your **setup file**.
[click]
You may be wondering what the -e dot means. -e means “install this as code I’m editing” which means your code won’t be **copied** into your virtualenv, it will be linked. So if you edit your code, when you run python it will pick up your changes. And the dot means to install the current directory. So it will install from the setup.py file in your current directory.
The end result here is that your hello world library will now be importable in your virtual environment, even though your code is inside the source directory!

Install it locally

```
$ pip install -e .
Obtaining file:///path/to/helloworld
Installing collected packages: helloworld
  Running setup.py develop for helloworld
Successfully installed helloworld
```

You run this command in your **virtual environment**, in the folder that has your **setup file**.
[click]
You may be wondering what the -e dot means. -e means “install this as code I’m editing” which means your code won’t be **copied** into your virtualenv, it will be linked. So if you edit your code, when you run python it will pick up your changes. And the dot means to install the current directory. So it will install from the setup.py file in your current directory.
The end result here is that your hello world library will now be importable in your virtual environment, even though your code is inside the source directory!

Let’s Test It!

```
$ python

>>> from helloworld import say_hello

>>> say_hello()
'Hello, World!'

>>> say_hello("Everybody")
'Hello, Everybody!'
```

We run python in the virtual env we just installed into. And you can see that we can import helloworld as if it was in our current directory (it’s not), and run the code inside! This is going to be a bit tiresome if we have to do this every time to make sure our setup file is correct, so we’ll sort that out in a minute

Perfect It

In theory we can upload this to PyPI, but I feel there are 3 things we need to do first:

* Documentation
* Tests

… but first a bit of housekeeping

When we build our wheel it creates a bunch of files we don't want to check into git.
There's a great site called gitignore.io. Type Python in the box, hit **create** and it'll generate you a file you can paste into a .gitignore file.

choosealicense.com

setup.py

```
setup(
    ...
    classifiers=[
         "Programming Language :: Python :: 3",
         "Programming Language :: Python :: 3.6",
         "Programming Language :: Python :: 3.7",
         "License :: OSI Approved :: GNU General
Public License v2 or later (GPLv2+)",
         "Operating System :: OS Independent",
     ],
)
```

https://pypi.org/classifiers/

ReStructured Text

- Pythonic
- Powerful
- Can use Sphinx

Markdown

- More Widespread
- Simpler
- Can use MkDocs

Pick a format for documentation - you can choose ReStructured Text or Markdown. There are similar tools to build your documentation for both, and both can be published to Read The Docs. I’m not going to show you how to do that, but you should write comprehensive documentation, and you should publish to Read The Docs!

README.md

```
# Hello World

This is an example project demonstrating
how to publish a python module to PyPI.
```

A readme should contain a title and a short description of what the module does.

README.md

````
…

## Installation

Run the following to install:

```python
pip install helloworld
```
````

… you should also write some installation instructions …

README.md

````
…
## Usage

```python
from helloworld import say_hello

# Generate "Hello, World!"
say_hello()

# Generate "Hello, Everybody!"
say_hello("Everybody")
```
````

… and a usage section.

In some projects a readme may be all you need. In others you'll need to write more than one page of documentation and I recommend you use Sphinx or MkDocs and publish to ReadTheDocs.

Add to setup.py

```
from setuptools import setup

with open("README.md", "r") as fh:
    long_description = fh.read()

setup(
    …
    long_description=long_description,
    long_description_content_type="text/
markdown",
)
```

Need to add the README info to PyPI, so we do this.

Test with pytest?

You should use pytest for testing - it's great!
Now we need to install pytest. Which means other contributors will need to install pytest. So we need a requirements.txt file or a Pipfile. Let’s use Pipenv and Pipfile because that’s new and fun!

Test with pytest?

First we need a Pipfile

You should use pytest for testing - it's great!
Now we need to install pytest. Which means other contributors will need to install pytest. So we need a requirements.txt file or a Pipfile. Let’s use Pipenv and Pipfile because that’s new and fun!

Create a Pipfile

```
$ pipenv install -e .

$ pipenv install --dev 'pytest>=3.7'

$ pipenv shell
```

Pipenv manages both our dependencies and our virtualenv.

This creates ...

Pipfile

```
...

[packages]
"e1839a8" = {path = ".", editable = true}

[dev-packages]
pytest = ">=3.7"
...
```

Pipfile.lock

```
  "pytest": {
    "hashes": [

"sha256:3459a12...d9541834a164666aa40395b02",

"sha256:96bfd45dbe863...31475d2bccc4f305118"
    ],
    "index": "pypi",
    "version": "==3.7.2"
  },
```

Check both of these into git. Now when someone else runs Pipenv install they get *exactly* the same versions as you, so they will always get 3.7.2 unless the lock file is updated.

Setup vs Pipfile

- **setup.py**
  - is for production dependencies (Flask, Click, Numpy, Pandas)
  - versions should be as relaxed as possible (>3.0,<4.0)
- **Pipfile**
  - is for development requirements: (Pytest, Mock, Coverage.py)
  - versions should be as relaxed as possible, but are **fixed** in the lock file

Our setup file doesn't have any dependencies, but if it did...

test_helloworld.py

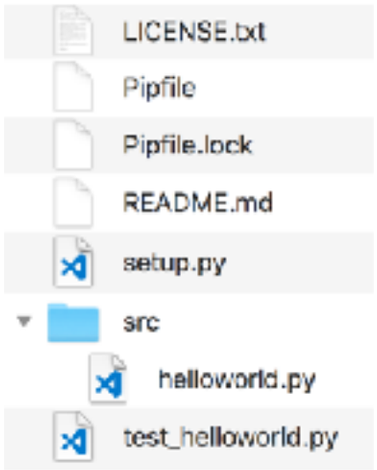
```
from helloworld import say_hello


def test_helloworld_no_params():
    assert say_hello() == "Hello, World!"


def test_helloworld_with_param():
    assert say_hello("Everyone") == "Hello, Everyone!"
```

Now that we have pytest installed, we can write some tests!

Running tests

```
$ pytest
============== test session starts ==============
platform darwin -- Python 3.6.3, pytest-3.7.2,
py-1.5.4, pluggy-0.7.1
rootdir: /path/to/helloworld, inifile:
collected 2 items

test_helloworld.py ..                          [100%]
============ 2 passed in 0.02 seconds ===========
```

helloworld



Just a quick pause to show you what this all looks like.

Source Distribution

```
$ python setup.py sdist
running sdist
...
warning: check: missing required meta-data: url
warning: check: missing meta-data: either (author
and author_email) or (maintainer and
maintainer_email) must be supplied
...
Creating tar archive
removing 'helloworld-0.0.1' (and everything under
it)
```

A source distribution should be published in parallel to each binary dist. It should be an offline version of your git repo, generally.

setup.py

```
setup(
    ...
    url="https://github.com/judy2k/helloworld",
    author="Mark Smith",
    author_email="mark.smith@vonage.com",
)
```

Test It?

```
$ tar tzf dist/helloworld-0.0.1.tar.gz
helloworld-0.0.1/
helloworld-0.0.1/PKG-INFO
helloworld-0.0.1/README.md
helloworld-0.0.1/setup.py
helloworld-0.0.1/setup.cfg
helloworld-0.0.1/src/
helloworld-0.0.1/src/helloworld.egg-info/
helloworld-0.0.1/src/helloworld.egg-info/PKG-INFO
helloworld-0.0.1/src/helloworld.egg-info/SOURCES.txt
helloworld-0.0.1/src/helloworld.egg-info/top_level.txt
helloworld-0.0.1/src/helloworld.egg-info/
dependency_links.txt
helloworld-0.0.1/src/helloworld.py
```

LICENSE.txt?
Pipfile
Pipfile.lock
test_helloworld.py

So that command created a source tarball.

Check Manifest

```
$ pip install check-manifest

$ check-manifest --create

$ git add MANIFEST.in
```

Check manifest compares what's in git to what's in your MANIFEST.in and tries to ensure they match.

Test It?

```
  helloworld-0.0.1/LICENSE.txt
  helloworld-0.0.1/MANIFEST.in
  helloworld-0.0.1/PKG-INFO
✅ helloworld-0.0.1/Pipfile
✅ helloworld-0.0.1/Pipfile.lock
✅ helloworld-0.0.1/README.md
  helloworld-0.0.1/setup.cfg
  helloworld-0.0.1/setup.py
  helloworld-0.0.1/src/
  helloworld-0.0.1/src/helloworld.egg-info/
  helloworld-0.0.1/src/helloworld.egg-info/PKG-INFO
  helloworld-0.0.1/src/helloworld.egg-info/SOURCES.txt
  helloworld-0.0.1/src/helloworld.egg-info/dependency_links.txt
  helloworld-0.0.1/src/helloworld.egg-info/top_level.txt
  helloworld-0.0.1/src/helloworld.py
✅ helloworld-0.0.1/test_helloworld.py
```

Publish It!

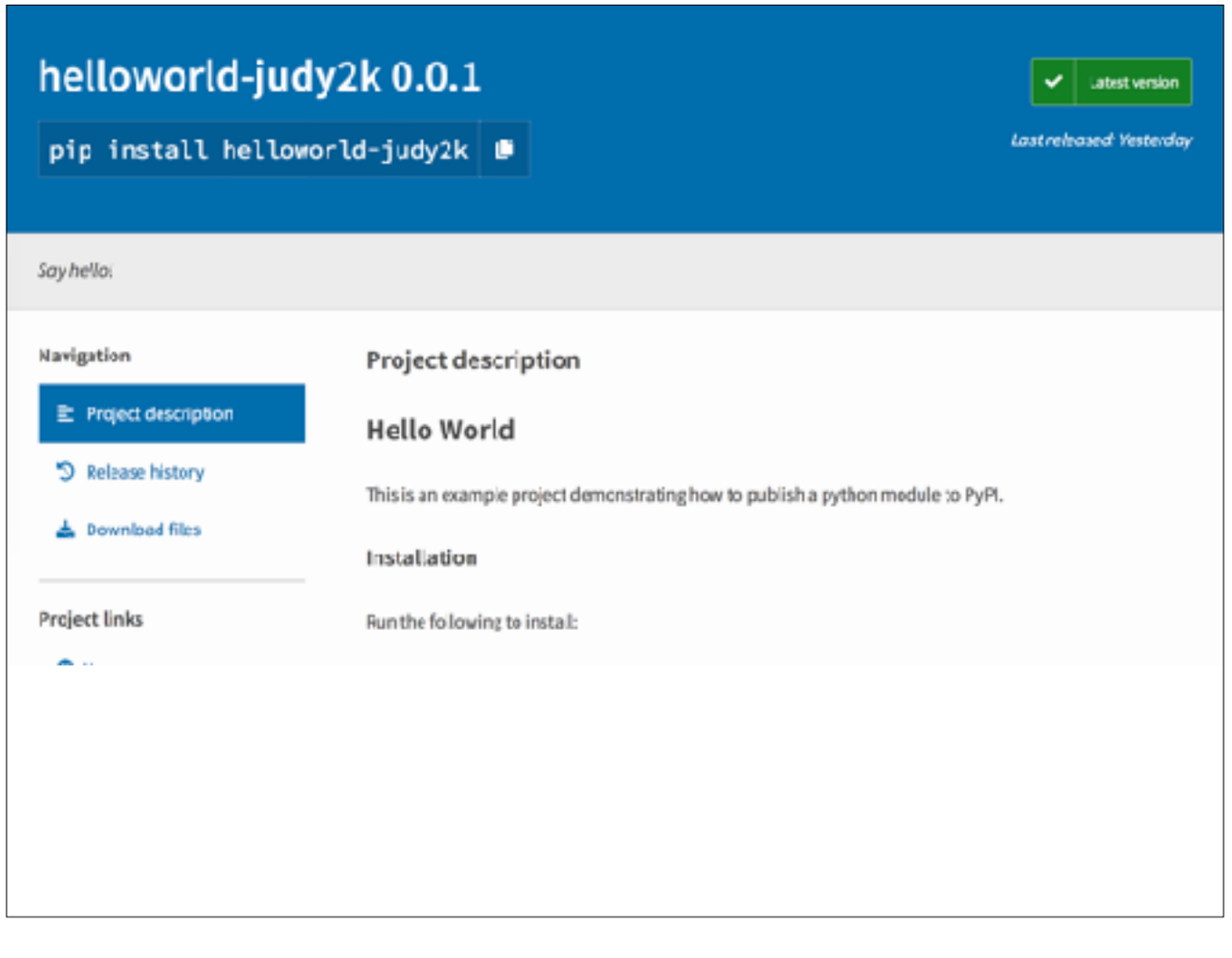
Now we're really at the point where we can publish, so let's get this thing on PyPI.

Build It

```
$ python setup.py bdist_wheel sdist

$ ls dist/
helloworld-0.0.1-py3-none-any.whl
helloworld-0.0.1.tar.gz
```

Push To PyPI

```
$ pipenv install --dev twine

$ twine upload dist/*
username: USER
password:
Uploading distributions to https://
upload.pypi.org/legacy/
Uploading helloworld-0.0.1-py3-none-any.whl
Uploading helloworld-0.0.1.tar.gz
```



Formatted our markdown. A link to GitHub repo, that you can't see...

Productionize? It

So those are really the bare essentials for publishing a package. We've got some tests, some documentation, and we've published the package. I'd recommend doing another couple of things, but maybe you can leave these until *after* you've first published.
We should test against different versions of Python - and there's a tool called Tox that does exactly that...

tox.ini

```
[tox]
envlist = py36,py37

[testenv]
deps = pytest
commands = pytest
```

The first thing to do is to write a config file for tox. It can be as simple as this, but can be a lot more complicated if you want.

Testing With Tox

```
$ pip install tox
$ tox
...
================ test session starts ================
platform darwin -- Python 3.6.3, pytest-3.7.2,
py-1.5.4, pluggy-0.7.1
rootdir: /path/to/helloworld, inifile:
collected 2 items

test_helloworld.py ..
[100%]

============== 2 passed in 0.02 seconds ==============
...
```

Tox creates a virtual env for each version of Python, installs your package into it, and then runs your tests under each one.

Testing With Tox

```
...
_____________________________ summary
_____________________________
  py36: commands succeeded
  py37: commands succeeded
  congratulations :)
```

... and we get a smiley face to say that everything went okay!

Testing With Tox

```
...
_____________________________ summary
_____________________________
  py36: commands succeeded
  py37: commands succeeded
  congratulations :)
```

... and we get a smiley face to say that everything went okay!

Testing With Tox

```
============= test session starts =============
platform darwin -- Python 3.6.3, pytest-3.7.2,
py-1.5.4, pluggy-0.7.1
rootdir: /path/to/helloworld, inifile:
collected 2 items

test_helloworld.py ..
[100%]

=========== 2 passed in 0.02 seconds ===========
```

This is why we use a src dir. Our CWD is our project dir, which is in our Python Path! If our code was here, even if we've installed into our testing virtualenvs, we would be _using_ the code in this directory, so we're not testing our package!

.travis.yml

```
language: python
sudo: false

python:
  - "3.6"
  - "3.7-dev"

install:
  - pip install tox

script:
  - tox -v -e py
```

So far everything run on a developer's machine. We don't know if it will work on someone else's computer! So we should set up CI, and I like Travis for that.

Extra Credit

- Badges!
  - Code Coverage (Coveralls, codecov.io)
  - Quality Metrics (Code Climate, Landscape.io)
- Manage versioning with bumpversion
- Test on OSX & Windows
- More Documentation
  - Contributors Section
  - Code of Conduct

We can add status badges to our readme, with coverage, quality. We can use bumpversion to manage our version numbers. Test on different platforms ... add documentation...
Code Coverage: coveralls.io, codecov.io
Code Quality: codeclimate.com, landscape.io

Don't Do This!

All this stuff took me a couple of days. It's boring, time-consuming and error-prone!
And you know what's really good for boring, time consuming and error prone work? Automation!

Cookiecutter is a tool that generates a folder full of files following a given template. There are a bunch of good templates out there, and they're easy to copy and customise.
So let's pretend we didn't spend all that time doing all of this by hand...

Use Cookiecutter

Ionel Cristian Mărieș has written a template for python projects that closely matches this advice. Whereas this talk has been quite opinionated, Ionel's template has a bunch of options, so it will ask you **lots** of questions.

Use Cookiecutter

```
$ pip install cookiecutter
```

Ionel Cristian Mărieș has written a template for python projects that closely matches this advice. Whereas this talk has been quite opinionated, Ionel's template has a bunch of options, so it will ask you **lots** of questions.

Use Cookiecutter

```
$ pip install cookiecutter

$ cookiecutter gh:ionelmc/cookiecutter-
pylibrary
```

Ionel Cristian Mărieș has written a template for python projects that closely matches this advice. Whereas this talk has been quite opinionated, Ionel's template has a bunch of options, so it will ask you **lots** of questions.

Use Cookiecutter

```
$ pip install cookiecutter

$ cookiecutter gh:ionelmc/cookiecutter-
pylibrary
... Lots of questions ...
```

Ionel Cristian Mărieș has written a template for python projects that closely matches this advice. Whereas this talk has been quite opinionated, Ionel's template has a bunch of options, so it will ask you **lots** of questions.

Use Cookiecutter

```
$ pip install cookiecutter

$ cookiecutter gh:ionelmc/cookiecutter-
pylibrary
... Lots of questions ...
... Copy in your code and tests ...
```

Ionel Cristian Mărieș has written a template for python projects that closely matches this advice. Whereas this talk has been quite opinionated, Ionel's template has a bunch of options, so it will ask you **lots** of questions.

Use Cookiecutter

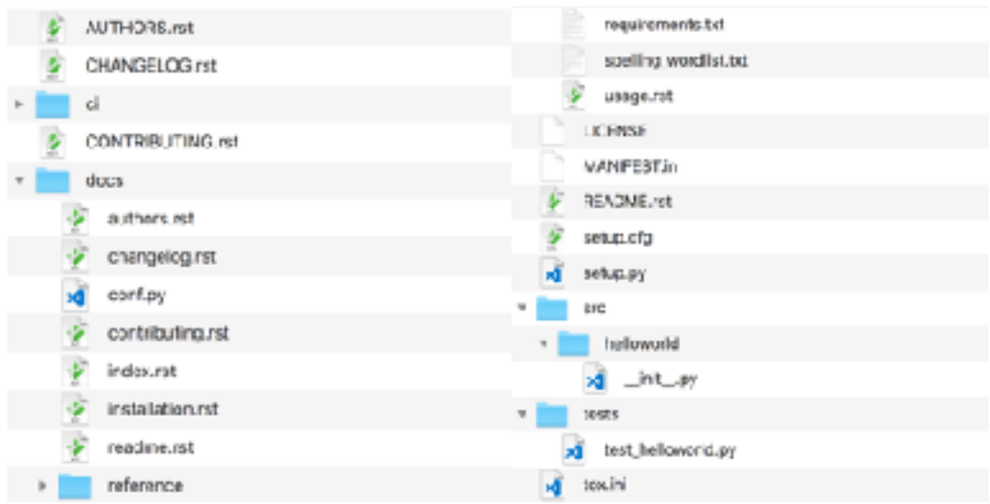
```
$ pip install cookiecutter

$ cookiecutter gh:ionelmc/cookiecutter-
pylibrary
... Lots of questions ...
... Copy in your code and tests ...
... Some minor file tweaks ...
```

Ionel Cristian Mărieș has written a template for python projects that closely matches this advice. Whereas this talk has been quite opinionated, Ionel's template has a bunch of options, so it will ask you **lots** of questions.

Use Cookiecutter

```
$ pip install cookiecutter

$ cookiecutter gh:ionelmc/cookiecutter-
pylibrary
... Lots of questions ...
... Copy in your code and tests ...
... Some minor file tweaks ...
... DONE!
```

Ionel Cristian Mărieș has written a template for python projects that closely matches this advice. Whereas this talk has been quite opinionated, Ionel's template has a bunch of options, so it will ask you **lots** of questions.



So ... that took me about **5 minutes**. I could have cut this talk down to the last 2 slides, if I'd wanted, instead of **wasting all your time**. [click] But hopefully this gives you an overview of good packaging practice and encourages you to publish your own packages on PyPI!

So ... that took me about **5 minutes**. I could have cut this talk down to the last 2 slides, if I'd wanted, instead of **wasting all your time**. [click] But hopefully this gives you an overview of good packaging practice and encourages you to publish your own packages on PyPI!

Slides & Code:
bit.ly/ppppopypi

Follow Me On Twitter: **@Judy2k**!