Reproducible Science - Python Packaging

Jul 15, 2020

Contents:

Tutorial on creating a reproducible python package.

CHAPTER 1

Introduction

This tutorial will teach you how to create a reproducible python package for anyone to install.

1.1 Module Learning Objectives

This module will be fully interactive. Participants are **strongly encouraged** to follow along on the command line. After completing this module, participants should be able to:

- Create a python package hosted on GitHub
- Specify package dependencies
- Create tests to validate package
- Understand the importance of random seeds and deterministic testing
- Install package with pip from GitHub

1.2 Why is this important?

Python is often thought as a scripting language, and used in a similar manner to bash. While this is true, many python scripts require third-party packages, and there is often no way to know if the script actually functions as expected on your system.

If you plan on sharing your script with others, we recommend transforming it into a proper package with specified dependencies and validation tests. This precaution will improve reproducibility and help you avoid the "works on my system" issues you encounter while supporting your community.

1.3 Requirements

- Accounts

  - GitHub
- Software
  - Python 3
  - git
  - python pip

CHAPTER 2

Creating the package repository

The python package we create in this tutorial will be hosted from GitHub.

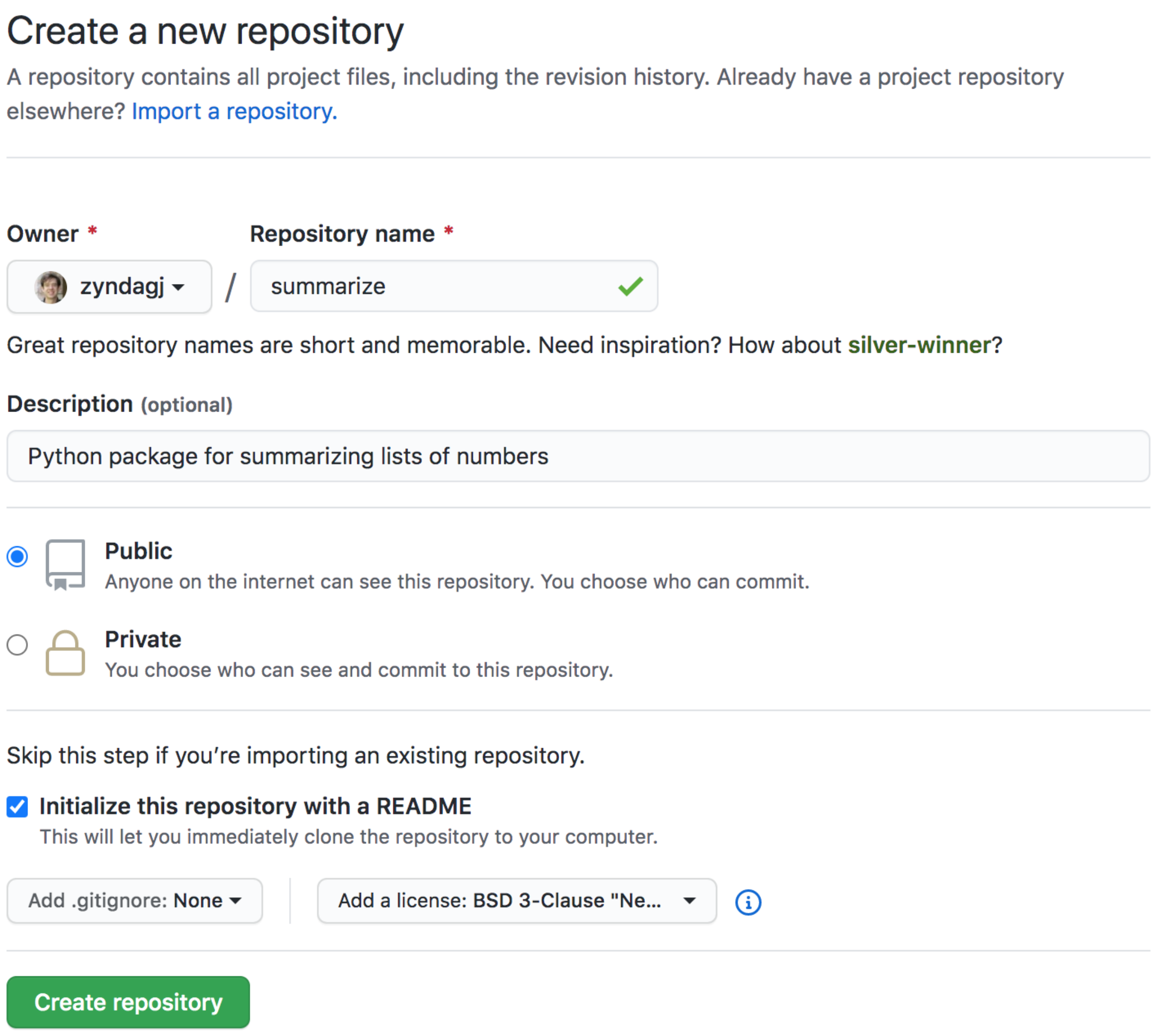
> *We won't be targeting pypi with this tutorial because it is a proper archive, where projects are not meant to be deleted*

While this means your source code will be public, you should always expect the source code of your python packages and scripts to be visible since it is only compiled at runtime.

2.1 Create the repository

Log in to GitHub and create a new repository. The package we create in this tutorial will generate and summarize a list of numbers, so we'll call both the project and package "summarize."

Make sure you initialize the repository with a README file, and choose an appropriate license.



Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? Import a repository.

Owner * zyndagj / Repository name * summarize

Great repository names are short and memorable. Need inspiration? How about silver-winner?

Description (optional)

Python package for summarizing lists of numbers

[✓ Public] Anyone on the internet can see this repository. You choose who can commit.

Private You choose who can see and commit to this repository.

Skip this step if you're importing an existing repository.

[✓] Initialize this repository with a README

This will let you immediately clone the repository to your computer.

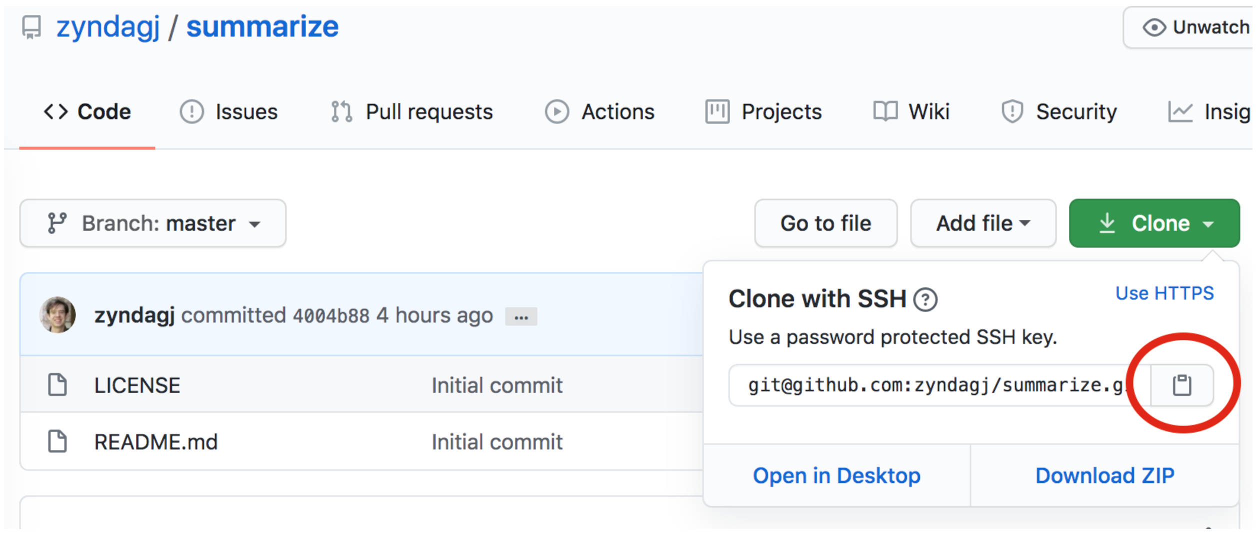
Add .gitignore: None

Add a license: BSD 3-Clause "Ne...

Then click **"Create repository"** to create your repository.

2.2 Clone the repository locally

Click the "clone" button to get the URL to clone your repository locally.



```
git@github.com:zyndagj/summarize.g
```

The URL that was copied can then be used to "git clone". After cloning the repository, enter the directory.

```
$ git clone <paste>
$ cd summarize
```

CHAPTER 3

Writing the python code

The “summarize” package will not flex your knowledge of python, it is just a means to learn how to package code. The source code itself will consist of the single `__init__.py` file, which will contain the following 3 functions:

main()

- The default function that is run when the package is invoked from the command line
- Uses argparse to accept the `-N` parameter, which specifies the number of values that are generated, and defaults to 5.

gen_numbers(n_numbers)

- A function that uses numpy.random.randint to generate `n_numbers` random numbers ranging from 0 to 99

summarize(numbers)

- Returns the mean of the `numpy.ndarray`, `numbers`.

3.1 Create the package directory

The current directory is the repository, create another `summarize` directory to serve as the package.

```
$ mkdir summarize
$ cd summarize
```

When the package is installed on a system, you can imagine this directory and its contents being copied into the `site-packages` path.

3.2 Creating the `__init__.py` file

While the __init__.py file was designed to allow for a directory to be imported as a module with sub-modules, it is a good convention to always follow when creating packages.

In your preferred text editor, create a file called `__init__.py` that contains the following sections:

3.2.1 The header section

```
1 #!/usr/bin/env python
2
3 import numpy as np
4 import argparse
```

Line 1 tells your shell how to run this file when executed, and lines 3 and 4 import the packages necessary to run this python program. While argparse is built in to all python distributions, Numpy is an external dependency that must be installed to be imported and run.

To test whether you have numpy already installed on your system, you can use `pip list` to list out all installed packages.

```
$ pip list | grep numpy
```

If you don't see a line for numpy, please install it using your preferred method.

```
$ pip install --user numpy
```

3.2.2 The `main()` function

```
1  def main():
2      # CLI arguments
3      parser = argparse.ArgumentParser(description='A simple tool for computing the mean of a random list')
4      parser.add_argument('-N', metavar='INT', type=int, help='Number of random integers [%(default)s]', default=5)
5      args = parser.parse_args()
6      # Generate the random numbers
7      numbers = gen_numbers(args.N)
8      # Calculate the mean
9      mean = summarize(numbers)
10     # Print the mean
11     print(mean)
```

Transforming your python script into a tool usable on the CLI through argparse can be done in as few as 3 lines. The first instruction (line 3) constructs the ArgumentParser object and also describes the the tool itself. Line 4 adds the first and only argument, which restricts values to integers and includes a description which states the default of 5 numbers. Line 5 parses the input and generates the `args` object. The value passed in through the `-N` parameter can then be accessed through `args.N`.

After setting up the CLI arguments, the random numbers are generated in line 7 and the mean is calculated in line 9. Finally, the calculated mean is printed before exiting.

3.2.3 The `gen_numbers()` function

After the `main()` function, add the `gen_numbers()` function to generate a variable length array of random integers.

```python
def gen_numbers(n_numbers):
    '''
    Generates n_numbers integers ranging from 0 to 99
    '''
    return np.random.randint(100, size=n_numbers)
```

3.2.4 The `summarize()` function

Even though it may seem redundant, create a `summarize()` function to compute the mean of the input array. Creating a specialized function for this will help with testing later.

```python
def summarize(numbers):
    '''
    Computes the mean of the numbers ndarray
    '''
    return np.mean(numbers)
```

3.2.5 Epilogue

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

These two lines tell python what to run when the script is invoked. In our case, the `main()` function is run. This section should always exist at the end of a file so all functions and global-scoped variables have already been initialized before running anything.

3.3 Current structure

At this point, you should have a directory called `summarize` containing the file `__init__.py`.

```
$ cd ..
$ tree summarize
summarize/
└── __init__.py

0 directories, 1 file
```

Assuming you have numpy already installed, running `__init__.py` with the `-h` argument should present you with its help text.

```
$ python summarize/__init__.py -h
usage: __init__.py [-h] [-N INT]

A simple tool for computing the mean of a random list

optional arguments:
  -h, --help   show this help message and exit
  -N INT       Number of random integers [5]
```

Running it without an argument should also return a number.

```
$ python summarize/__init__.py
52.2
```

CHAPTER 4

Creating the package files

At this point, the package directory is complete. However, the metadata that describes the package is still missing.

This means you can

```
$ python
>>> import summarize
>>> summarize.main()
35.0
```

but `pip install .` will not work yet. To enable this, the setup.py file needs to be created to describe the package and any requirements.

4.1 The `setup.py` file

Open your favorite text editor and create your `setup.py` containing the following fields:

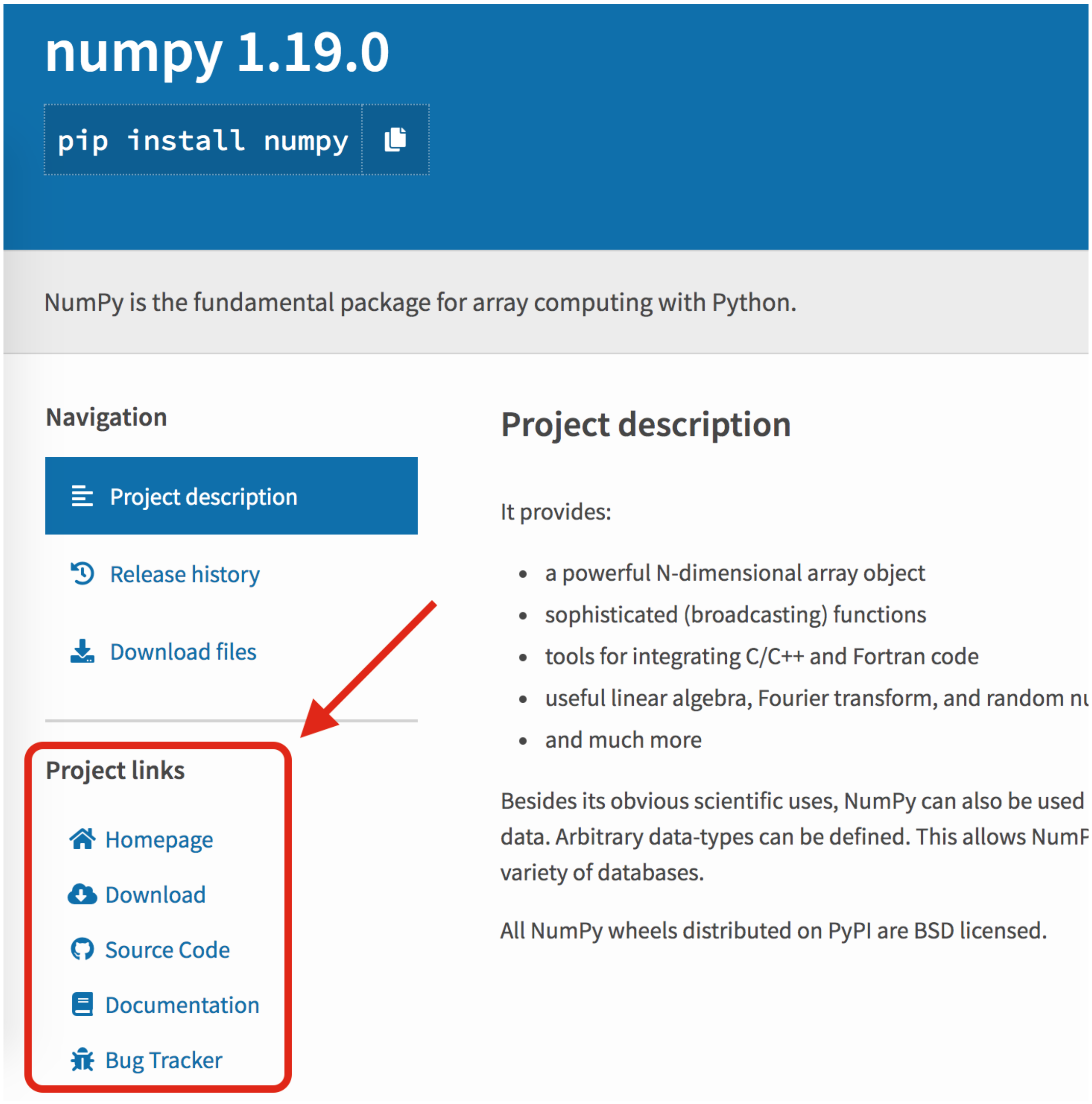
```
#!/usr/bin/env python

from setuptools import setup

setup(
    name='summarize',            # package name
    version='0.0.1',             # package version
    description='A simple tool for computing the mean of a random list', # short␣
↪description
    url='GitHub repo URL',       # URL for the project (optional)
    packages=["summarize"],      # Package directory
    python_requires='>=3.6,<4', # Requires a recent python3
    install_requires=[           # Runtime dependencies
        'numpy>=1,<2'
    ],
    extras_require={ #   $ pip install sampleproject[dev]
```

(continues on next page)

(continued from previous page)

```
16             'dev': ['pytest','tox'],
17             'test': ['pytest','tox']
18         },
19         entry_points={              # Creates CLI scripts for accessing modules and␣
   ↪functions
20             'console_scripts': [
21                 'summarize=summarize:main'
22             ],
23         }
24 )
```

While the `url` field on line 9 is optional for publishing, fill in the URL to your GitHub repository to get used to providing this information. If you eventually publish a package on pypi, this is how the project links are populated.


numpy 1.19.0

```
pip install numpy
```

NumPy is the fundamental package for array computing with Python.

Navigation

- Project description
- Release history
- Download files

Project links

- Homepage
- Download
- Source Code
- Documentation
- Bug Tracker

Project description

It provides:

- a powerful N-dimensional array object
- sophisticated (broadcasting) functions
- tools for integrating C/C++ and Fortran code
- useful linear algebra, Fourier transform, and random nu
- and much more

Besides its obvious scientific uses, NumPy can also be used data. Arbitrary data-types can be defined. This allows NumF variety of databases.

All NumPy wheels distributed on PyPI are BSD licensed.

There are many other optional fields to describe the package, which you can see here. Feel free to take some time to add any additional fields.

One of the main reasons to transition from a simple script to a full package is to not only support an intuitive installation process, but to also specify fine-grained dependencies.

python_requires This field on line 11 restricts the version of python that can install the summarize package. In this case, Python must be at least version 3.6, and we do not assume summarize will work with the hypothetical Python 4.

install_requires This field on line 12 is a list of packages and potentially versions required for the package to run. We know that summarize imports and uses Numpy for generating and summarizing random numbers. The current version of Numpy is 1.19. Instead of just saying "this requires numpy", the version was restricted to a release between 1.0 and 2.0. While your software may not install when Numpy eventually transitions to v2, it would force you to determine whether it was still compatible, which is usually done through testing.

The `extra_requires` is not a required field, but the next two sections in this tutorial will depend on both `pytest` and `tox`, so they were included now to streamline their usage later.

4.2 Installing your package

At this point, you should be able to install your package with `pip`. Installing summarize with

```
$ pip install --user .
$ pip list | grep summarize
```

installs it, where `.` targets the package in your current working directory (cwd), as a static package to your local site-packages. You can also enable active development by installing it with

```
$ pip uninstall -y summarize
$ pip install --user -e .
$ pip list | grep summarize
summarize                    0.0.1              /Users/gzynda/Documents/
↪reproducible_python/docs/assets
```

If the installation location is already on your path, you'll be able to run the `summarize` CLI script created by `setup.py`.

```
$ summarize -h
usage: summarize [-h] [-N INT]

A simple tool for computing the mean of a random list

optional arguments:
  -h, --help  show this help message and exit
  -N INT      Number of random integers [5]
```

If this does not work, you'll have to add the install location to your path

```
# Linux + macOS
export PATH=~/.local/bin:${PATH}
# Windows
# ????
```

4.3 Current structure

```
$ tree
.
├── setup.py
└── summarize
    └── __init__.py
```

Note: You may see additional files in this tree if you did run `__init__.py` or `import summarize`.

4.4 Additional Information

- https://packaging.python.org/guides/distributing-packages-using-setuptools/
- https://github.com/pypa/sampleproject

CHAPTER 5

Creating tests with pytest

```
$ mkdir tests
```

In your favorite editor, create the file `tests/test_summarize.py` and add the follow 4 lines to load the summarize package and numpy.

```python
#!/usr/bin/env python

import numpy as np
import summarize
```

Make sure everything is in order by running

```
$ pip install --user pytest
$ python -m pytest tests/
```

to install pytest and make sure it works on our simple test.

If summarize is installed, you can also just run

```
$ pytest
```

5.1 Your first test

```python
def test_true():
        assert True == True
```

While not very useful for your package, this is a simple test to ensure `True` is equal to `True` with the standard assert statement, and *should* always succeed.

If you have any errors running `pytest` at this point, something is wrong with your configuration.

5.2 Testing the size of the array

The next test will ensure that when the `summarize.gen_numbers` function is given a 5, it returns an array with 5 values. Running the code manually would look something like

```
>>> import summarize
>>> summarize.gen_numbers(5)
array([20, 19, 38, 79, 50])
```

The test should then assert that the returned array has a length of 5.

```
def test_gen_numbers_5():
        assert len(summarize.gen_numbers(5)) == 5
```

Try creating a new test called test_gen_numbers_10 that ensures it works with an argument of 10 too.

5.3 Running a test across multiple values

For times like this where you want to run a test across multiple values, you can import pytest and utilize the pytest.mark.parametrize to sweep across a list of values.

> *Note: The decorator function is spelled parametrize, not parameterize. Your brain may unconsciously auto-correct that.*

```
import pytest

@pytest.mark.parametrize("n", [5, 10])
def test_gen_numbers_len(n):
        assert len(summarize.gen_numbers(n)) == n
```

Much cleaner, and easier to scale. Try modifying fixture this to also test for 20.

5.4 Testing the returned type

Each of the values in the array is the type `np.int64` and we can test for that.

```
def test_gen_numbers_5_type():
        for n in summarize.gen_numbers(5):
                assert isinstance(n, np.int64)
```

5.5 Testing the returned values

If you call `summarize.gen_numbers(5)` multiple times, you'll notice that you get different numbers each time.

```
>>> import summarize
>>> summarize.gen_numbers(5)
array([20, 19, 38, 79, 50])
>>> summarize.gen_numbers(5)
array([95, 94, 80, 68,  4])
```

You can make this random process deterministic for your tests be setting the random seed used to initialize the random number generator.

```
>>> import summarize
>>> import numpy as np
>>> np.random.seed(5)
>>> summarize.gen_numbers(5)
array([99, 78, 61, 16, 73])
>>> summarize.gen_numbers(5)
array([ 8, 62, 27, 30, 80])
>>> np.random.seed(5)
>>> summarize.gen_numbers(5)
array([99, 78, 61, 16, 73])
>>> np.random.seed(5)
>>> summarize.gen_numbers(5)
array([99, 78, 61, 16, 73])
```

You can apply this to testing as well for deterministic output, even with random calls.

```
def test_gen_numbers_5_vals_seed():
        np.random.seed(5)
        assert np.all(summarize.gen_numbers(5) == [99, 78, 61, 16, 73])
```

You can also test the returned values of the summary function. First, with a hardcoded input

```
def test_summarize_custom():
        assert summarize.summarize([1,1,1,1,1]) == 1
```

Then, with the seeded input

```
def test_summarize_seed():
        np.random.seed(5)
        numbers = summarize.gen_numbers(5)
        assert summarize.summarize(numbers) == np.mean([99,78,61,16,73])
```

5.6 Conclusions

After writing all these tests, you should see something like

```
===================================================== test session starts␣
↪=====================================================
platform darwin -- Python 3.8.3, pytest-5.4.3, py-1.9.0, pluggy-0.13.1
rootdir: /Users/gzynda/Documents/reproducible_python/docs/assets
collected 9 items

tests/test_summarize.py .........                                          ␣
↪                                   [100%]

====================================================== 9 passed in 0.14s␣
↪======================================================
```

and just **feel good**.

This may seem like plenty of tests for just a few simple functions, but you want at least one test per function, and this is a minimum. Ideally, you would want multiple tests per function to handle all kinds of edge cases. For instance, even after writing all of these tests, `summary.summary` still has the opportunity to fail and throw an error when given `-N` is less than 1. An error like this may be confusing to the user, and a human readable error statement should be returned before exiting. A test can even be written to test for the exit code.

This guide is not meant to teach you mastery over pytest and testing itself. This is only meant to be a gentle introduction to show you that writing tests is fairly simple and rewarding. There is still much to learn, like

- Error codes
- Consistent setup/teardown

CHAPTER 6

Sandboxing tests with tox

If you thought it was odd that we ran our pytests in our active python environment, you were correct. Ideally, the package should be installed in a clean environment that only contains the dependencies of the package being tested. After it is confirmed that the package can be installed, any tests should then be run.

Tox aims to not only support this usecase, but also run builds and tests on an array of platforms and python versions.

To test summarize with tox, you'll need to

- Install tox via pip
- Create the `pyproject.toml` file
- Create the `tox.ini` file

6.1 Installing tox

```
$ pip install --user tox
```

6.2 Creating `pyproject.toml`

Similar to `setup.py`, this is a standardized file name that tox looks for, and cannot change.

```
[build-system]
requires = [
    "setuptools >= 35.0.2",
    "setuptools_scm >= 2.0.0, <3"
]
build-backend = "setuptools.build_meta"
```

While a `pyproject.toml` can be fairly complex, our just tells tox that a modern version of setuptools is required to build our package.

6.3 Creating `tox.ini`

Once again, this is a standardized file name and cannot change.

```
[tox]
envlist =
    {py36,py37,py38}
isolated_build = true
skip_missing_interpreters = true

[testenv]
deps = pytest
commands =
    pytest {posargs}
```

The first section describes the python test environments. In our case, we want to test summarize using python 3.6, 3.7, and 3.8. If any of these python versions are not available on your system, those tests will be skipped and not throw an error. Each of these test environments will also be isolated from your system, and fresh dependencies will be installed from scratch. This will help ensure that your package will work for others as dependencies are updated over time.

The second section describes the test environment. The "deps" section lists out any dependencies required to build and test (not run) the package. In the case of summarize, pytest is the only external test dependency, and since tox creates a clean environment, pytest will not be available if not specified here. Lastly, you specify how the tests are run with the "commands" field.

6.4 Running tox

Running tox is extremely simple

```
$ tox
```

It will create a clean virtual environment for every python version you want to test again, install your package and all dependencies, and then run your test commands. This is meant to make continuous integration and delivery easy and simple.

CHAPTER 7

Documentation

All good projects have documentation - even if your program has help text. At a minimum, you should detail any requirements and write instructions for installing, testing, and using a python package. GitHub will always render the `README.md` file on the landing page for your repository. We suggest creating subsections for each of these topics in the file.

7.1 Requirements

```
## Requirements

Summarize runs on Python >= 3.6 and requires Numpy.
```

7.2 Installing

Pip can install from more places than just pypi and a local directory. If you `git add`, `git commit`, and `git push` all the files we created, you'll be able to install summarize without a separate checkout command.

These directions show how users can install your package directly from your GitHub URL.

```
## Installing

Install with pip from master

```
$ pip install --user git+https://github.com/zyndagj/summarize.git
```

or a specific release

```
```

(continues on next page)
```

(continued from previous page)

~~~
$ pip install --user git+https://github.com/zyndagj/summarize.git@release
```
~~~

7.3 Testing

It's always a good idea to demonstrate how to both setup the test environment and run tests

~~~
## Testing

Install test dependencies and run tests

```
$ git clone https://github.com/zyndagj/summarize.git
$ cd summarize
$ pip install --user .[dev]

# Run tests with tox
$ tox

# Run tests with pytest
$ pip install .
$ pytest
```
~~~

7.4 Usage

Lastly, include an explanation on usage along with expected output.

~~~
## Usage

```
usage: summarize [-h] [-N INT]

A simple tool for computing the mean of a random list

optional arguments:
  -h, --help  show this help message and exit
  -N INT      Number of random integers [5]
```

```
$ summarize -N 5
48.8
```
~~~

CHAPTER 8

Conclusions and Extras

8.1 Conclusions

Congratulations! You have successfully created a python package that is reproducible on other systems!

Your package is better prepared to survive the test of time than most of the software repositories that exist on GitHub because it contains

- helpful documentation
- specific requirements
- tests for each method
- standard installation procedure

Remember that if you decide to continue development on this or any other package, you can increment the version in `setup.py` and pip will be able to upgrade the package and any dependencies too.

8.2 Extras

Additional challenges to explore in your free time

8.2.1 Add the version flag

Using the 'version' action and the following code in setup.py

```
# Create version
VERSION = "0.0.1"
with open(os.path.join(pkg_dir,'version.py'), 'w') as VF:
    cnt = """
# THIS FILE IS GENERATED FROM SETUP.PY
version = '%s'
```

(continues on next page)

(continued from previous page)

```
"""
    VF.write(cnt%(VERSION))
```

8.2.2 Add additional functions and create tests for them

- Create a new function called `my_median()` to compute median
- Add a new `--median` flag to use this new function instead of `summarize()`
- Create a new pytest for `my_median()`

8.2.3 Create a GitHub action that tests your project on push

https://help.github.com/en/actions/language-and-framework-guides/using-python-with-github-actions

Allow the installation on python2 and use tox to test both python 2 and 3 environments