Testing your package

DEVELOPING PYTHON PACKAGES

James Fulton
Climate informatics researcher

The art and discipline of testing

Imagine you are working on this function

```
def get_ends(x):
    """Get the first and last element in a list"""
    return x[0], x[-1]
```

You might test it to make sure it works

```
# Check the function
get_ends([1,1,5,39,0])
```

```
(1, 0)
```

The art and discipline of testing

Good packages brag about how many tests they have

- 91% of the pandas package code has tests

Writing tests

```
def get_ends(x):
    """Get the first and last element in a list"""
    return x[0], x[-1]
```

```
def test_get_ends():
    assert get_ends([1,5,39,0]) == (1,0)
```

```
test_get_ends()
```

Writing tests

```
def get_ends(x):
    """Get the first and last element in a list"""
    return x[0], x[1]
```

```
def test_get_ends():
    assert get_ends([1,5,39,0]) == (1,0)
```

```
test_get_ends()
```

```
AssertionError:
...
```

Writing tests

```
def get_ends(x):
    """Get the first and last element in a list"""
    return x[0], x[-1]
```

```
def test_get_ends():
    assert get_ends([1,5,39,0]) == (1,0)
    assert get_ends(['n','e','r','d']) == ('n','d')
```

Organizing tests inside your package

```
mysklearn/
|-- mysklearn   <-- package
|-- tests       <-- tests directory
|-- setup.py
|-- LICENSE
|-- MANIFEST.in
```

Organizing tests inside your package

Test directory layout

```
mysklearn/tests/
|-- __init__.py
|-- preprocessing
|   |-- __init__.py
|   |-- test_normalize.py
|   |-- test_standardize.py
|-- regression
|   |-- __init__.py
|   |-- test_regression.py
|-- test_utils.py
```

Code directory layout

```
mysklearn/mysklearn/
|-- __init__.py
|-- preprocessing
|   |-- __init__.py
|   |-- normalize.py
|   |-- standardize.py
|-- regression
|   |-- __init__.py
|   |-- regression.py
|-- utils.py
```

Organizing a test module

Inside `test_normalize.py`

```
from mysklearn.preprocessing.normalize import (
    find_max, find_min, normalize_data
)

def test_find_max(x):
    assert find_max([1,4,7,1])==7

def test_find_min(x):
    assert ...

def test_normalize_data(x):
    assert ...
```

Inside `normalize.py`

```
def find_max(x):
    ...
    return x_max

def find_min(x):
    ...
    return x_min

def normalize_data(x):
    ...
    return x_norm
```

DataCamp: Unit testing for data science

Running tests with pytest

```
pytest
```

- `pytest` looks inside the `test` directory
- It looks for modules like `test_modulename.py`
- It looks for functions like `test_functionname()`
- It runs these functions and shows output

```
mysklearn/ <-- navigate to here
|-- mysklearn
|-- tests
|-- setup.py
|-- LICENSE
|-- MANIFEST.in
```

Running tests with pytest

```
pytest
```

```
========================= test session starts ==========================
platform linux -- Python 3.7.9, pytest-6.1.2, py-1.9.0, pluggy-0.13.1
rootdir: /home/workspace/mypackages/mysklearn
collected 6 items

tests/preprocessing/test_normalize.py ...                          [ 50%]
tests/preprocessing/test_standardize.py ...                        [100%]

========================== 6 passed in 0.23s ===========================
```

Running tests with pytest

```
pytest
```

```
========================= test session starts ==========================
platform linux -- Python 3.7.9, pytest-6.1.2, py-1.9.0, pluggy-0.13.1
rootdir: /home/workspace/mypackages/mysklearn     <--    ran in this directory
collected 6 items                                 <--    found 6 test functions

tests/preprocessing/test_normalize.py ...                          [ 50%]
tests/preprocessing/test_standardize.py ...                        [100%]

========================== 6 passed in 0.23s ===========================
```

Running tests with pytest

```
pytest
```

```
========================= test session starts ==========================
platform linux -- Python 3.7.9, pytest-6.1.2, py-1.9.0, pluggy-0.13.1
rootdir: /home/workspace/mypackages/mysklearn
collected 6 items

tests/preprocessing/test_normalize.py ...                        [ 50%]  <--
tests/preprocessing/test_standardize.py ...                      [100%]  <--

========================== 6 passed in 0.23s ===========================
```

Running tests with pytest

```
pytest
```

```
========================= test session starts ==========================
platform linux -- Python 3.7.9, pytest-6.1.2, py-1.9.0, pluggy-0.13.1
rootdir: /home/workspace/mypackages/mysklearn
collected 6 items

tests/preprocessing/test_normalize.py ...                          [ 50%]
tests/preprocessing/test_standardize.py ...                        [100%]

========================== 6 passed in 0.23s ===========================
```

Running tests with pytest

```
pytest
```

```
================================== test session starts ==================================
...
tests/preprocessing/test_normalize.py .F.                                          [ 50%]
tests/preprocessing/test_standardize.py ...                                        [100%]

======================================== FAILURES ========================================
_____________________________________ test_mymax _____________________________________
...

tests/preprocessing/test_normalize.py:10: AssertionError
================================ short test summary info ================================
FAILED tests/preprocessing/test_normalize.py::test_mymax - assert -100 == 100      <-- test_mymax
================================ 1 failed, 5 passed in 0.17s ============================
```

Let's practice!

DEVELOPING PYTHON PACKAGES

Testing your package with different environments

DEVELOPING PYTHON PACKAGES

James Fulton
Climate informatics researcher

Testing multiple versions of Python

This `setup.py` allows any version of Python from version 2.7 upwards.

```
from setuptools import setup, find_packages

setup(
    ...
    python_requires='>=2.7',
)
```

To test these Python versions you must:

- Install all these Python versions
- Install your package and all dependencies into each Python
- Run `pytest`

Testing multiple versions of Python

This `setup.py` allows any version of Python from version 2.7 upwards.

```
from setuptools import setup, find_packages

setup(
    ...
    python_requires='>=2.7',
)
```

To test these Python versions you must:

- Install all these Python versions
- Run `tox`

What is tox?

- Designed to run tests with multiple versions of Python

Configure tox

Configuration file - `tox.ini`

```
mysklearn/
|-- mysklearn
|   |-- ...
|-- tests
|   |-- ...
|-- setup.py
|-- LICENSE
|-- MANIFEST.in
|-- tox.ini   <--- configuration file
```

Configure tox

Configuration file - `tox.ini`

```
[tox]
envlist = py27, py35, py36, py37

[testenv]
deps = pytest
commands =
    pytest
    echo "run more commands"
    ...
```

- Headings are surrounded by square brackets `[...]`.
- To test Python version X.Y add `pyXY` to `envlist`.
- The versions of Python you test need to be installed already.
- The `commands` parameter lists the terminal commands `tox` will run.
- The `commands` list can be any commands which will run from the terminal, like `ls`, `cd`, `echo` etc.

Running tox

```
tox
```

```
mysklearn/    <-- navigate to here
|-- mysklearn
|   |-- ...
|-- tests
|   |-- ...
|-- setup.py
|-- LICENSE
|-- MANIFEST.in
|-- tox.ini
```

tox output

```
py27 create: /mypackages/mysklearn/.tox/py27
py27 installdeps: pytest
py27 inst: /mypackages/mysklearn/.tox/.tmp/package/1/mysklearn-0.1.0.zip
py27 installed: mysklearn==0.1.0,numpy==1.16.6,pandas==0.24.2,pytest==4.6.11,...
py27 run-test-pre: PYTHONHASHSEED='2837498672'
...
```

tox output

```
py27 run-test: commands[0] | pytest
========================== test session starts ==========================
platform linux2 -- Python 2.7.17, ...
rootdir: /home/workspace/mypackages/mysklearn
collected 6 items

tests/preprocessing/test_normalize.py ...                          [ 50%]
tests/preprocessing/test_standardize.py ...                        [100%]

============================ 6 passed in 0.23s ============================
```

tox output

```
...
_________________________ summary ___________________________
  py27: commands succeeded
  py35: commands succeeded
  py36: commands succeeded
  py37: commands succeeded
```

tox output

```
...
_________________________ summary ___________________________
  py27: commands succeeded
  py35: commands succeeded
  py36: commands succeeded
ERROR:   py37: commands failed
```

Let's practice!

DEVELOPING PYTHON PACKAGES

Keeping your package stylish

DEVELOPING PYTHON PACKAGES

James Fulton
Climate informatics researcher

datacamp

Introducing flake8

- Standard Python style is described in **PEP8**
- A style guide dictates how code should be laid out
- `pytest` is used to find bugs
- `flake8` is used to find styling mistakes

Running flake8

Static code checker - reads code but doesn't run

```
flake8 features.py
```

```
features.py:2:1: F401 'math' imported but unused
..
```

```
<filename>:<line number>:<charcter number>:<error code> <desciption>
```

Using the output for quality code

```
1. import numpy as np
2. import math
3.
4. def mean(x):
5.     """Calculate the mean"""
6.     return np.mean(x)
7. def std(x):
8.     """Calculate the standard deviation"""
9.     mean_x = mean(x)
10.    std = mean((x-mean(x))**2)
11.    return std
12.
```

```
flake8 features.py
```

```
2:1: F401 'math' imported but unused
4:1: E302 expected 2 blank lines, found 1
7:1: E302 expected 2 blank lines, found 0
5:4: E111 indentation is not a multiple
     of four
6:4: E111 indentation is not a multiple
     of four
9:5: F841 local variable 'mean_x' is
     assigned to but never used
```

Using the output for quality code

```
1. import numpy as np
2.
3.
4. def mean(x):
5.     """Calculate the mean"""
6.     return np.mean(x)
7.
8.
9. def std(x):
10.     """Calculate the standard deviation"""
11.     mean_x = mean(x)
12.     std = mean((x - mean_x)**2)
13.     return std
14.
```

```
flake8 features.py
```

Breaking the rules on purpose

`quadratic.py`

```
4. ...
5. quadratic_1 = 6 * x**2 + 2 * x + 4;
6. quadratic_2 = 12 * x**2 + 2 * x + 8
7. ...
```

Breaking the rules on purpose

`quadratic.py`

```
4. ...
5. quadratic_1 =  6 * x**2 + 2 * x + 4;
6. quadratic_2 = 12 * x**2 + 2 * x + 8
7. ...
```

```
flake8 quadratic.py
```

```
quadratic.py:5:14: E222 multiple spaces after operator
quadratic.py:5:35: E703 statement ends with a semicolon
```

Breaking the rules on purpose

`quadratic.py`

```
4. ...
5. quadratic_1 =  6 * x**2 + 2 * x + 4; # noqa
6. quadratic_2 = 12 * x**2 + 2 * x + 8
7. ...
```

```
flake8 quadratic.py
```

Breaking the rules on purpose

`quadratic.py`

```
4. ...
5. quadratic_1 =  6 * x**2 + 2 * x + 4; # noqa: E222
6. quadratic_2 = 12 * x**2 + 2 * x + 8
7. ...
```

```
flake8 quadratic.py
```

```
quadratic.py:5:35: E703 statement ends with a semicolon
```

flake8 settings

Ignoring style violations without using comments

```
flake8 --ignore E222 quadratic.py
```

```
quadratic.py:5:35: E703 statement ends with a semicolon
```

```
flake8 --select F401,F841 features.py
```

```
2:1: F401 'math' imported but unused
9:5: F841 local variable 'mean_x' is assigned
     to but never used
```

Choosing package settings using setup.cfg

Create a `setup.cfg` to store settings

Package file tree

```
.
|-- example_package
|   |-- __init__.py
|   `-- example_package.py
|-- tests
|   |-- __init__.py
|   `-- test_example_package.py
|-- README.rst
|-- LICENSE
|-- MANIFEST.in
`-- setup.py
```

Choosing package settings using setup.cfg

Create a `setup.cfg` to store settings

```
[flake8]

ignore = E302
exclude = setup.py

per-file-ignores =
  example_package/example_package.py: E222
```

Package file tree

```
.
|-- example_package
|   |-- __init__.py
|   `-- example_package.py
|-- tests
|   |-- __init__.py
|   `-- test_example_package.py
|-- README.rst
|-- LICENSE
|-- MANIFEST.in
|-- setup.py
`-- setup.cfg
```

The whole package

```
$ flake8
```

```
```

Package file tree

```
.
|-- example_package
|   |-- __init__.py
|   `-- example_package.py
|-- tests
|   |-- __init__.py
|   `-- test_example_package.py
|-- README.rst
|-- LICENSE
|-- MANIFEST.in
|-- setup.py
`-- setup.cfg
```

Use the least filtering possible

Least filtering

1. `# noqa : <code>`
2. `# noqa`
3. `setup.py` → `per-file-ignores`
4. `setup.py` → `exclude`, `ignore`

Most filtering

Let's practice!

DEVELOPING PYTHON PACKAGES