Testing your package

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]
```

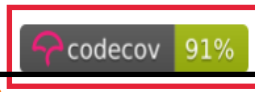
You might test it to make sure it works

```
# 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

4. Writing tests
Each function in your package should have a test function. Here the test function for get-ends is defined. It runs the get-ends function on a list and makes sure get-ends returns the correct answer. If get-ends returns the correct answer then the test function passes. If something is wrong with get-ends, then this test function raises an assertion error.

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
```

8. Organizing tests inside your package
The best way to lay out the tests directory is to copy the structure of the code directory. The test directory has an empty init file and the preprocessing subdirectory. Inside this subdirectory is the test-normalize module. The test-normalize module should contain all the tests for functions in the normalize module. Every other module in the code directory should have its own test module in the test directory.

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
```

9. Organizing a test module
Inside the test module, there should be a test function for each function defined in the source module. Remember that you will need to import the functions you are testing from the main package. This should be done using an absolute import. In this course we are only going to use these simple assert statements for testing, but if your package has more complex functions take UNIT TESTING Course.

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
```

10. Running tests with pytest
Once you have written these tests you can run them all at once using pytest. From the terminal all you need to do is navigate to the top of your directory, and then run the pytest command. Pytest will look inside the test directory, and search for all modules that start with test-underscore. Inside those modules it will look for all functions that start with test-underscore, and it will run these functions.

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 =======================
```

12. Running tests with pytest
Pytest was run in the mysklearn directory and it found 6 test functions.

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 ==============================
```

13. Running tests with pytest
The test functions cover the normalize and standardize modules with 50 percent and 100% coverage of the code.

Testing your package with different environments

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`
- 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.

6. Configure tox
Inside the file you need to create the tox heading like this. You will then specify the versions of Python you want to test using the envlist parameter. Here we are testing python 2.7, 3.5, 3.6 and 3.7. You need to have these versions of Python already installed on your computer. Tox won't install new python versions. Using each of these versions of Python, tox will install your package and its dependencies. Under the testenv heading you need to tell tox what you want it to do once it has installed your package. You use the commands parameter to tell tox to run pytest. However, pytest isn't installed by your package dependencies, so you need to specify pytest as a tox dependency. You could actually add any commands you like to the commands list and tox will run them all. These need to be shell commands, which you could run from the terminal.

Running tox

```
tox
```

Keeping your package stylish

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. ...
```

```
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: 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

```
[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