

Effective Programming Practices for Economists

Software engineering

Writing simple (py)tests

Janoś Gabler and Hans-Martin von Gaudecker

Reminder of the example

```
>>> raw = pd.read_csv("survey.csv")
>>> raw
```

Q001	Q002	Q003
0 strongly disagree	agree	python
1 strongly agree	strongly agree	Python
2 -77	disagree	R
3 agree	-77	Python
4 -99	-99	Python
5 NaN	strongly agree	Python
6 neutral	strongly agree	Python
7 disagree	agree	python
8 strongly agree	-99	PYTHON
9 agree	-99	Ypthon

From the metadata you know

- Q001: I am a coding genius
- Q001: I learned a lot
- Q003: What is your favourite language
- -77 not readable
- -99 no reply

First function in clean_data.py

```
def _clean_agreement_scale(sr):  
    sr = sr.replace({"-77": pd.NA, "-99": pd.NA})  
    categories = ["strongly disagree", "disagree", "neutral", "agree", "strongly agree"]  
    dtype = pd.CategoricalDtype(categories=categories, ordered=True)  
    return sr.astype(dtype)
```

A function in test_clean_data.py

Function's properties:

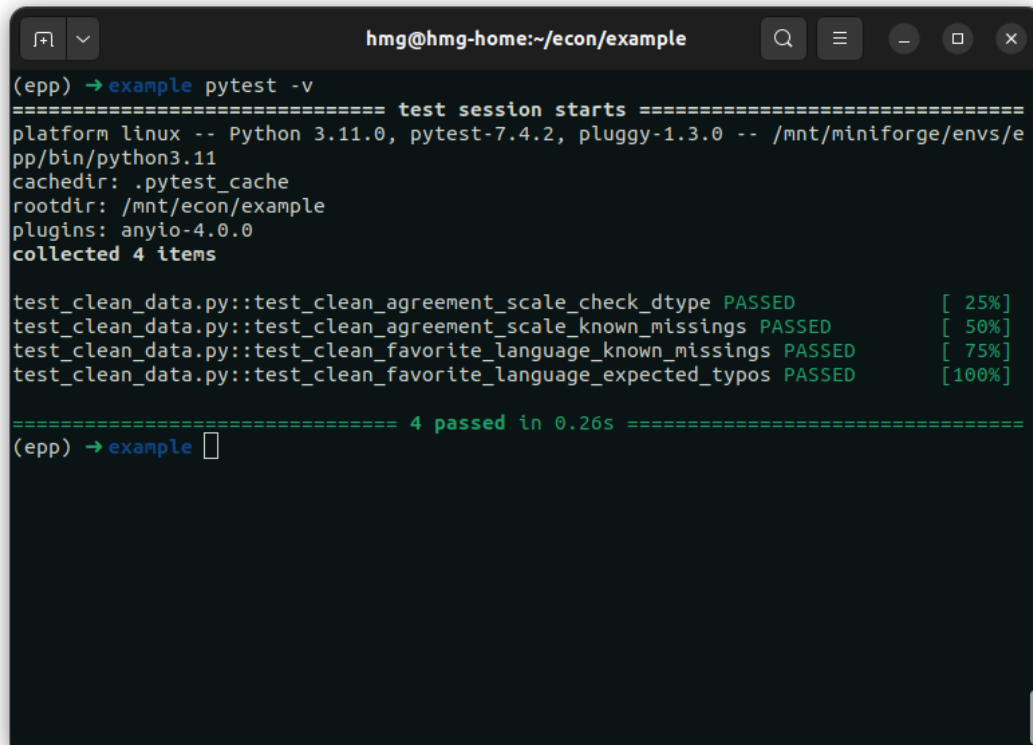
- starts with `test_`
- name explains what it does
- defines what we expect
- calls the function to be tested to calculate actual result
- asserts that actual and expected results coincide

```
def test_clean_agreement_scale_check_dtype():  
    expected = pd.CategoricalDtype(  
        categories=[  
            "strongly disagree",  
            "disagree",  
            "neutral",  
            "agree",  
            "strongly agree",  
        ],  
        ordered=True,  
    )  
    actual = _clean_agreement_scale(pd.Series([])).dtype  
    assert expected == actual
```

Another function in test_clean_data.py

```
def test_clean_agreement_scale_known_missings():  
    result = _clean_agreement_scale(pd.Series(["-77", "-99"]))  
    expected = pd.Series([pd.NA, pd.NA], dtype=result.dtype)  
    pd.testing.assert_series_equal(result, expected)
```

Run pytest

A terminal window with a dark background and light text. The title bar shows the user 'hmg' at 'hmg-home' in the directory '~/econ/example'. The terminal content shows the execution of 'pytest -v' in a JupyterLab environment (epp). The output includes session details like platform (linux), Python version (3.11.0), and pytest version (7.4.2). It lists four collected items, all of which passed. A progress bar shows 100% completion. The final summary states '4 passed in 0.26s'.

```
(epp) → example pytest -v
===== test session starts =====
platform linux -- Python 3.11.0, pytest-7.4.2, pluggy-1.3.0 -- /mnt/miniforge/envs/epp/bin/python3.11
cachedir: .pytest_cache
rootdir: /mnt/econ/example
plugins: anyio-4.0.0
collected 4 items

test_clean_data.py::test_clean_agreement_scale_check_dtype PASSED [ 25%]
test_clean_data.py::test_clean_agreement_scale_known_missings PASSED [ 50%]
test_clean_data.py::test_clean_favorite_language_known_missings PASSED [ 75%]
test_clean_data.py::test_clean_favorite_language_expected_typos PASSED [100%]

===== 4 passed in 0.26s =====
(epp) → example
```

Basic rules

- Put tests in modules called `test_XXX.py` , with functions `test_YYY_ZZZ` , ...
 - `XXX` is the name of the module to be tested
 - `YYY` is the name of the function to be tested
 - `ZZZ` is a description of the behaviour being tested
- Inside these functions, keep structure clear:
 - Define expected result
 - Calculate actual result
 - Assert that they coincide
- Usually one assert statement per test function