Powerful tests and reproducible benchmarks with pytest-cases
Agenda

• (reminder) pytest basics
• pytest-cases
• Under the hood
• Benchmarking
Agenda

• (reminder) pytest basics
• pytest-cases
• Under the hood
• Benchmarking
pytest basics

• « the » test framework for python, now far more widely used than unittest & nose

• Philosophy: you can reduce boilerplate and copy-paste code in tests to zero
  • A test is a plain old python function
    ```python
def test_foo():
```
  • A test parameter is a function argument
    ```python
@pytest.mark.parametrize("param1",[0, 1])
def test_foo(param1):
```
  • Tests can reuse shared tools, objects, features with setup/teardown mechanisms: « fixtures »
    ```python
def test_foo(tmpdir):
```

> (+ Many other features + Great IDE integration) = Making tests becomes easy and fun!
• **Test configuration files** named `conftest.py` can be placed in any test folder to declare common mechanisms (fixtures, custom hooks)

• A very wide range of **customization hooks** is available to plugins, so that they can modify almost everything. This is the reason why the ecosystem of plugin is so great! ([https://docs.pytest.org/en/latest/reference/plugin_list.html](https://docs.pytest.org/en/latest/reference/plugin_list.html)). For example distributed testing, etc.
Agenda

• (reminder) *pytest basics*
• *pytest-cases*
• Under the hood
• Benchmarking
pytest-cases

- A plugin for pytest 😊

- Main goal: make pytest easier to use in the context of data science (test data from files and simulation, benchmarks, etc.)

- Secondary goal: make pytest easier to use in general (propose innovative ideas that may end up in pytest in the future)

- A mature project, with active users:
  - 648 commits since June 2018
  - 103 version tags
  - Very low bugfix latency

https://smarie.github.io/python-pytest-cases/
Initial use-case: testing Analytics & AI libraries

Tested code: analytics libraries
  e.g. Machine Learning: Predictive Maintenance, Energy Cons/Prod Forecasting…

Typical inputs:
  data frames (= 2-dimensional tables, $n$ rows x $m$ columns) > "complex" python objects

Test data used to create these inputs:
  simulated (nominal, edge cases) + files/other storage (realistic nominal/customer feedback)

Extra need: benchmarking
  compare several algorithms/versions on a reference collection of datasets
Initial use-case

How to integrate complex test parameters in our tests?

Simple test parameters:
e.g. 2 tuples

```python
import pytest
from example import foo

@pytest.mark.parametrize("a, b", [(1, 2), (-1, -2)])
def test⏱foo(a, b):
    # check that foo runs correctly and that the result is a tuple.
    assert isinstance(foo(a, b), tuple)
```

Complex test parameters: e.g. 20+ DataFrames, each

- Requiring **a few lines of code** to be generated
- Requiring a detailed documentation for maintenance purposes
- Possibly **parametrized** (several simulation variants, several data files…)
- Possibly **requiring resources** (fixtures, but used by a parameter not by a test)
- Requiring a **user-friendly readable test node id**!
Proposed solution: case functions

Iterable of parameter values

```python
@pytest.mark.parametrize("name", ["EuroPython 2021", "World"])
def test_greet_basic_param(name):
    
    result = greet(name)
    print(result)

    assert result == "Hello, %s!" % name
```

`Collection` of « case functions » returning the values

```python
def case_europython():
    
    return "EuroPython 2021"

def case_world():
    
    return "World"
```

@parametrize_with_cases("name", cases=".")
def test_greet_basic_param(name):
    
    ...
Case functions – ids and marks

```python
@pytest.mark.parametrize("name",
    ["EuroPython 2021",
     pytest.param("World", marks=pytest.mark.skip)]
),
ids=["europepython", "world"]

def test_greet_complete_param(name):
    """A parametrized version of the test for 'greet'."""
    result = greet(name)
    print(result)
    assert result == "Hello, %s!" % name
```

Marks and ids can be set using `pytest.param` around each param value.

```python
def case_europepython():
    """The name of the europython conference."""
    return "EuroPython 2021"

def case_world():
    """A very common name for such a demo."""
    return "World"
```

The mark decorator can be used as follows:

```python
@case(id="earth")
@pytest.mark.skip
```
Case functions – parametrization and fixtures

In pytest, parameters can not be parametrized, and parameters can not use fixtures. This is possible with pytest-cases.

```python
@pytest.mark.parametrize("suffix", range(2))
def case_parametrized(suffix):
    """Let's add a suffix so that the name is less common."""
    return f"World{suffix}"

def case_needing_fixture(tmpdir):
    """This case returns the name of the tmp dir created"""
    return tmpdir.basename
```

parametrized case  case requiring a fixture

Note: you can parametrize cases with cases (recursion)
Case functions – properties

• Laziness
  • All case functions are lazily called during the test node setup phase before the test is run
  • Case functions that require fixtures are transformed into fixtures, which behave also this way.

• Caching
  • if a case function result is needed several time during a single test node run (for example by plugins),
    the case function is guaranteed to be called only once.
**Initial idea:** associate each test module with a separate cases module following a default naming convention. This is still the default behaviour (two naming schemes are supported).

```python
test_foo.py
from example import foo
from pytest_cases import parametrize_with_cases

@parametrize_with_cases("a,b")
def test_foo(a, b):
    # check that foo runs correctly and that the result is a tuple.
    assert isinstance(foo(a, b), tuple)
```

```python
test_foo_cases.py
OR
cases_foo.py

def case_two_positive_ints():
    """ Inputs are two positive integers """
    return 1, 2

def case_two_negative_ints():
    """ Inputs are two negative integers """
    return -1, -2
```
Cases Collection – explicit reference

Easier for daily use: simply reference some container explicitly and rely on

```python
from pytest_cases import parametrize_with_cases

def case_one_positive_int():
    return 1

def case_one_negative_int():
    return -1

@parametrize_with_cases("i", cases=":.")
def test_with_this_module(i):
    assert i == int(i)

from pytest_cases import parametrize_with_cases

class Foo:
    def case_a_positive_int(self):
        return 1

    def case_another_positive_int(self):
        return 2

@parametrize_with_cases("a", cases=Foo)
def test_foo(a):
    assert a > 0
```

You can also use a list of modules, classes, functions:

```python
@parametrize_with_cases("name", cases=":.", Foo, case_world)
```
The `case_` prefix can be changed to collect several « kind » of cases independently:

```python
def algo_foo():
    return "foo"

def case_bar():
    return "bar"

@parametrize_with_cases("p", cases=".", prefix="algo_")
def test_algo(p):
    assert p == "foo"
```

Cases can be tagged

```python
@Case(tags="whitelisted")
def case_tagged():
    return "w"

@parametrize_with_cases("p", cases=".")
def test_tags(p):
    assert p == "w"
```

Advanced filtering using a callable filter (a few useful filters are provided built-in)

```python
from pytest_cases.filters import id_match_regex

@parametrize_with_cases("p", cases=".", filter=id_match_regex("t.*d"))
def test_filters(p):
    assert p == "w"
```
Additional features

• @parametrize_with_cases can be used on fixtures too!
  thanks to @fixture (see next part):

```python
from pytest_cases import fixture, parametrize_with_cases

@fixture
@parametrize_with_cases("a,b")
def c(a, b):
    return a + b
```

• Built-in current_cases fixture
  for debugging/reasoning on the current case (id, function)

```python
@parametrize_with_cases("name", cases=".")
def test_greet_basic_param2(name, current_cases):
    print(current_cases['name'])
    # e.g. Case(id='world', func=<function case_world at 0x00000222D3E51B8>, params={})
```
Agenda

- (reminder) pytest basics
- pytest-cases
- Under the hood
- Benchmarking
Under the hood: pytest goodies

- `pytest-cases` features required a few major improvements in pytest mechanisms
- These are available independently of case functions: documentation here
- In particular
  - `fixture_union` creates a fixture that will alternate between several fixtures
  - `@parametrize` = `@pytest.mark.parametrize`+
    - Supports an alternate way to define argname/argvalue (keyword-way). If you use it, the ids will also be auto-generated in a more user-friendly way 😊
    - Supports lazy parameter values (=functions), supports using a fixture as a parameter value
  - `@fixture` = `@pytest.fixture`+
    - Allows fixtures to be parametrized using `@pytest.mark.parametrize` (or `@parametrize`)
    - Correctly supports fixture unions.
Design choices at the beginning of pytest-cases:

- Do not try to do this as a pytest PR (too big, too blurry initially)
- Keep usage of pytest hooks to the bare minimum (too risky for compatibility with others)

Idea: dynamically create functions that wrap the user-written test and fixture functions, possibly adding (injecting) or removing arguments. Since `functools.wraps` was not capable of this, I wrote `makefun`.

```makefun
Dynamically create python functions with a proper signature.
```

![PyPI Badge](https://img.shields.io/pypi/py_version/tz) ![Build Badge](https://img.shields.io/build badge) ![Coverage Badge](https://img.shields.io/coverage) ![Tests Badge](https://img.shields.io/testsBadge) ![Codecov Badge](https://img.shields.io/codecov) ![Life Is On](https://img.shields.io/lifeon)

EuroPython 2021 - Sylvain Marié, Schneider Electric | Page 22
A glimpse on the challenge that was solved (detailed explanation on the documentation page)

No union: one test function = one fixture closure

With unions: one test function = several alternative fixture closures, unioned
Agenda

• (reminder) pytest basics
• pytest-cases
• Under the hood
• Benchmarking
Benchmarking - Does this look familiar?

<table>
<thead>
<tr>
<th>dataset</th>
<th>challenger</th>
<th>degree</th>
<th>status</th>
<th>duration_ms</th>
<th>cvrmse</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anscombe's quartet 1</td>
<td>polyfit(degree=1)</td>
<td>1</td>
<td>passed</td>
<td>1,9999</td>
<td>14,91%</td>
</tr>
<tr>
<td>Anscombe's quartet 1</td>
<td>polyfit(degree=2)</td>
<td>2</td>
<td>passed</td>
<td>1,0002</td>
<td>14,44%</td>
</tr>
<tr>
<td>Anscombe's quartet 2</td>
<td>polyfit(degree=1)</td>
<td>1</td>
<td>passed</td>
<td>0,9999</td>
<td>14,92%</td>
</tr>
<tr>
<td>Anscombe's quartet 2</td>
<td>polyfit(degree=2)</td>
<td>2</td>
<td>passed</td>
<td>0,9999</td>
<td>0,02%</td>
</tr>
<tr>
<td>Anscombe's quartet 3</td>
<td>polyfit(degree=1)</td>
<td>1</td>
<td>passed</td>
<td>0,0000</td>
<td>14,91%</td>
</tr>
<tr>
<td>Anscombe's quartet 3</td>
<td>polyfit(degree=2)</td>
<td>2</td>
<td>passed</td>
<td>0,9999</td>
<td>14,49%</td>
</tr>
<tr>
<td>Anscombe's quartet 4</td>
<td>polyfit(degree=1)</td>
<td>1</td>
<td>passed</td>
<td>0,0000</td>
<td>14,90%</td>
</tr>
<tr>
<td>Anscombe's quartet 4</td>
<td>polyfit(degree=2)</td>
<td>2</td>
<td>passed</td>
<td>0,9999</td>
<td>14,90%</td>
</tr>
<tr>
<td>Data file 'contant-1.csv'</td>
<td>polyfit(degree=1)</td>
<td>1</td>
<td>passed</td>
<td>0,0000</td>
<td>0,00%</td>
</tr>
<tr>
<td>Data file 'contant-1.csv'</td>
<td>polyfit(degree=2)</td>
<td>2</td>
<td>passed</td>
<td>0,9999</td>
<td>0,00%</td>
</tr>
<tr>
<td>Data file 'v-shape.csv'</td>
<td>polyfit(degree=1)</td>
<td>1</td>
<td>passed</td>
<td>0,0000</td>
<td>63,19%</td>
</tr>
<tr>
<td>Data file 'v-shape.csv'</td>
<td>polyfit(degree=2)</td>
<td>2</td>
<td>passed</td>
<td>0,0000</td>
<td>21,23%</td>
</tr>
</tbody>
</table>
How to get there with pytest-cases

Two kind of case functions: **challengers** (prefix `algo_`) and **datasets** (prefix `data_`)

One test representing the **evaluation protocol** (typically doing train/score/evaluate)

Results can be dumped to file by each test node, or collected in a **result_bag** using `pytest-harvest`
How to get there – multiple python environments

Sketch of design:

• @pytest.mark each « challenger » case function with a python virtual environment name
• For each virtual environment, only run the tests where the appropriate mark is set
• Combine results for all environments.

Useful tools:

• pytest-pilot to create a custom pytest mark and an easy to use CLI option to filter on it
• doit-api for nox-like orchestration of tasks using doit. (Unfortunately nox does not yet have all the features required for this, see nox issue tracker)
Thanks for watching! Questions?

**pytest basics**

```python
def case_europython():
    """The name of the europython conference."""
    return 'EuroPython 2021'
```

```python
@case(id='earth')
@pytest.mark.skip
def case_world():
    """A very common name for such a demo."""
    return 'World'
```

---

**pytest-cases under the hood**

```python
@parametrize
@fixture
@fixture_ref
@lazy_value
@fixture_union
...```

---

**cases collection**

```python
@parametrize_with_cases("p", cases="", prefix="algo_")
def test_algo(p):
    assert p == "foo"
```

---

**pytest-cases**

Before leaving, if I may ask for a favor:
- if you like pytest-cases please give the github repo a star!
- if you use it for research, you may now cite it thanks to Zenodo (link on project web page)