



# EFFICIENCY

Powerful tests and reproducible benchmarks  
with pytest-cases

2021-07-28 – Sylvain Marié

Senior Group Expert, *IoT & Digital Offers > Analytics & AI*

# 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

```
def test_foo():
```

- A test parameter is a function argument

```
@pytest.mark.parametrize("param1", [0, 1])  
def test_foo(param1):
```

- Tests can reuse shared tools, objects, features with setup/teardown mechanisms: « fixtures »

```
def test_foo(tmpdir):
```

Creates a temporary directory before test runs, and deletes it afterwards. The argument contains the path.

> (+ Many other features + Great IDE integration) = Making tests becomes easy and fun !

# pytest basics: misc

- **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

<https://smarie.github.io/python-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

pytest-cases

Separate test code from test cases in `pytest`.

python 2.7 | 3.5 | 3.6 | 3.7 | 3.8 | pytest 2 | 3 | 4 | 5 | 6 | Build passing | tests 1304 | coverage 84.43%

codecov 81% | flake8 0 C, 0 W, 47 I

doc latest | pypi v3.6.3 | downloads 669k | downloads/week 10k | stars 165 | DOI 10.5281/zenodo.3937829

# 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

```
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 !

```
# case 1: non-sorted uniformly sampled timeseries with 2 holes
case1 = pd.DataFrame({"datetime": pd.date_range(start='2017/1/1', periods=20,
                                                freq='-1d', tz='UTC'),
                    "data1": np.arange(0, 20),
                    "data2": np.arange(1, 21),
                    "data3": np.arange(1, 21)})
case1.drop([3, 12], inplace=True)
```

# Proposed solution: case functions



Iterable of parameter values

```
@pytest.mark.parametrize("name", ["EuroPython 2021", "World"])
def test_greet_basic_param(name):
    """A parametrized version of the test for `greet`."""
    result = greet(name)
    print(result)
    assert result == "Hello, %s!" % name
```

pytest-cases

Collection\* of « case functions » returning the values

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

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

@parametrize_with_cases("name", cases=".")
def test_greet_basic_param(name):
    ...
```

\*Collection per se is described later

# Case functions – ids and marks



```
@pytest.mark.parametrize("name",
    (
        "EuroPython 2021",
        pytest.param("World", marks=pytest.mark.skip)
    ),
    ids=("europython", "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

`ids=("europython", "world")`

`ids = iterable or callable`

pytest-cases

default id extracted from the function name

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

custom id

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

The mark decorator can be used

# Case functions – parametrization and fixtures

In `pytest`, parameters can not be parametrized, and parameters can not use fixtures.

This is possible with `pytest-cases`.

```
@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}"
```

parametrized case

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

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.

# Cases Collection – default

**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).

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)
```

Parametrized by

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

```
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)
```

module or module name

```
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
```

class

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

```
@parametrize_with_cases("name", cases=[".", Foo, case_world])
```

# Cases Collection – prefixes, filters and tags

The `case_prefix` can be changed to collect several « kind » of cases independently:

```
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

```
@case(tags="whitelisted")  
def case_tagged():  
    return "w"  
  
@parametrize_with_cases("p", cases=".", has_tag="whitelisted")  
def test_tags(p):  
    assert p == "w"
```

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

```
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):

```
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)

```
@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 0x00000222D3E518C8>, 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.


# One of the key tools of pytest-cases: `makefun`

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.

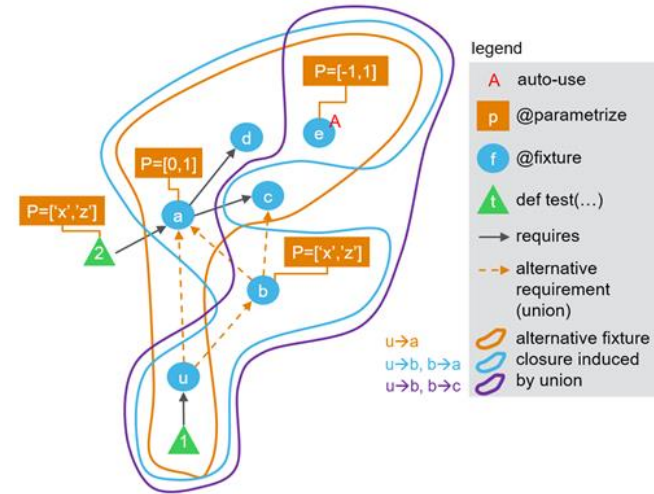
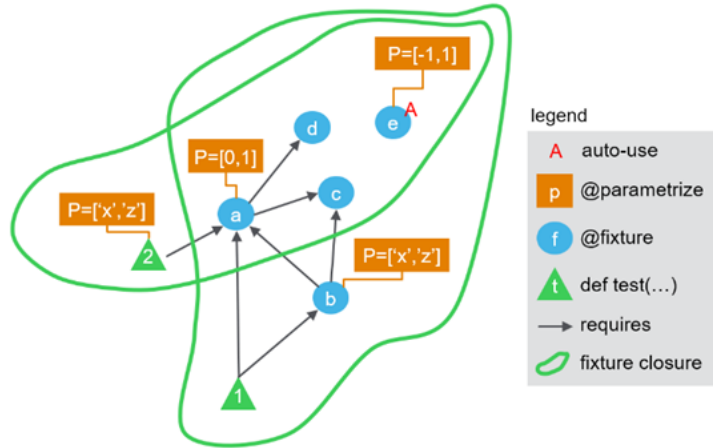
Since `functools.wraps` was not capable of this, I wrote `makefun`.



The screenshot shows the PyPI page for the `makefun` package. The title is "makefun" and the description is "Dynamically create python functions with a proper signature." Below the description are several badges: "python" with versions 2.7, 3.5, 3.6, 3.7, and 3.8; "Build" with a "passing" status; "tests" with a count of 77/77; "codecov" with a coverage of 71%; "doc" with a "latest" version; "pypi" with version v1.11.3; "downloads" with a count of 1M; "downloads/week" with a count of 29k; and "stars" with a count of 41.

# fixture\_union

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

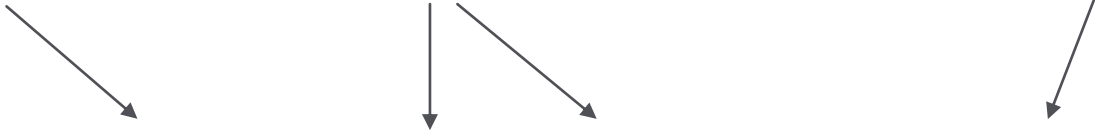
- (reminder) `pytest` basics
- `pytest-cases`
- Under the hood
- **Benchmarking**

# Benchmarking - Does this look familiar ?

datasets

algorithms

evaluation protocol



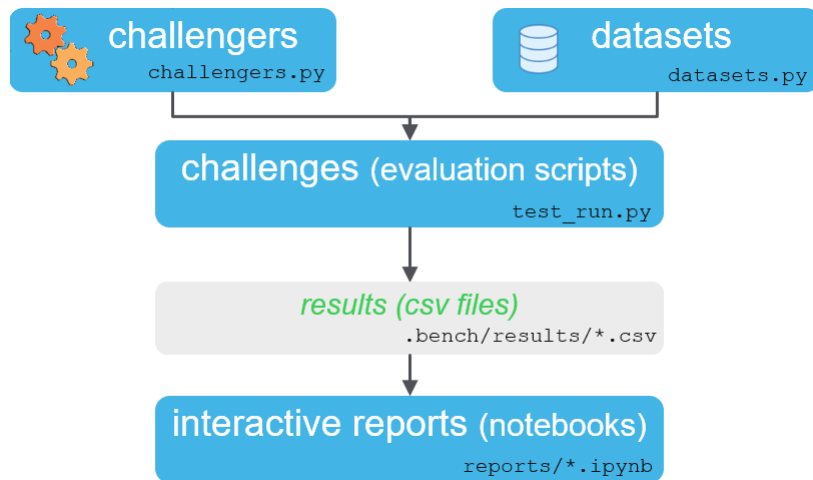
dataset	challenger	degree	status	duration_ms	cvmse
Anscombe's quartet 1	polyfit(degree=1)	1	passed	1,9999	14,91%
Anscombe's quartet 1	polyfit(degree=2)	2	passed	1,0002	14,44%
Anscombe's quartet 2	polyfit(degree=1)	1	passed	0,9999	14,92%
Anscombe's quartet 2	polyfit(degree=2)	2	passed	0,9999	0,02%
Anscombe's quartet 3	polyfit(degree=1)	1	passed	0,0000	14,91%
Anscombe's quartet 3	polyfit(degree=2)	2	passed	0,9999	14,49%
Anscombe's quartet 4	polyfit(degree=1)	1	passed	0,0000	14,90%
Anscombe's quartet 4	polyfit(degree=2)	2	passed	0,9999	14,90%
Data file 'contant-1.csv'	polyfit(degree=1)	1	passed	0,0000	0,00%
Data file 'contant-1.csv'	polyfit(degree=2)	2	passed	0,9999	0,00%
Data file 'v-shape.csv'	polyfit(degree=1)	1	passed	0,0000	63,19%
Data file 'v-shape.csv'	polyfit(degree=2)	2	passed	0,0000	21,23%

# 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



## case functions

```
def case_europython():  
    """The name of the europython conference."""  
    return "EuroPython 2021"  
  
@case(id="earth")  
@pytest.mark.skip  
def case_world():  
    """A very common name for such a demo."""  
    return "World"
```

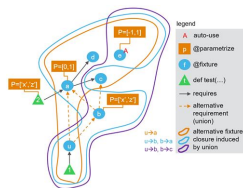
## pytest-cases

## cases collection

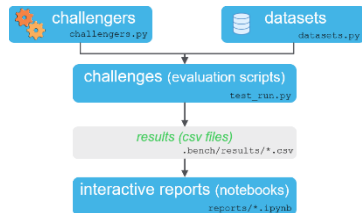
```
@parametrize_with_cases("p", cases=".", prefix="algo_")  
def test_algo(p):  
    assert p == "foo"
```

## pytest-cases under the hood

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



## Benchmarking



dataset	challenge	degree	status	duration_op	costme
Anscombe's quartet 1	polyfit(degree=1)	1	passed	1.9999	14.91%
Anscombe's quartet 1	polyfit(degree=2)	2	passed	1.0002	14.44%
Anscombe's quartet 2	polyfit(degree=1)	1	passed	0.9999	14.92%
Anscombe's quartet 2	polyfit(degree=2)	2	passed	0.9999	0.00%
Anscombe's quartet 3	polyfit(degree=1)	1	passed	0.0000	14.91%
Anscombe's quartet 3	polyfit(degree=2)	2	passed	0.9999	14.49%
Anscombe's quartet 4	polyfit(degree=1)	1	passed	0.0000	14.90%
Anscombe's quartet 4	polyfit(degree=2)	2	passed	0.9999	14.90%
Data file 'constant-1.csv'	polyfit(degree=1)	1	passed	0.0000	0.00%
Data file 'constant-1.csv'	polyfit(degree=2)	2	passed	0.9999	0.00%
Data file 'v-shape.csv'	polyfit(degree=1)	1	passed	0.0000	63.19%
Data file 'v-shape.csv'	polyfit(degree=2)	2	passed	0.0000	21.23%

## 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](#))

Life Is On



**Schneider**  
Electric

