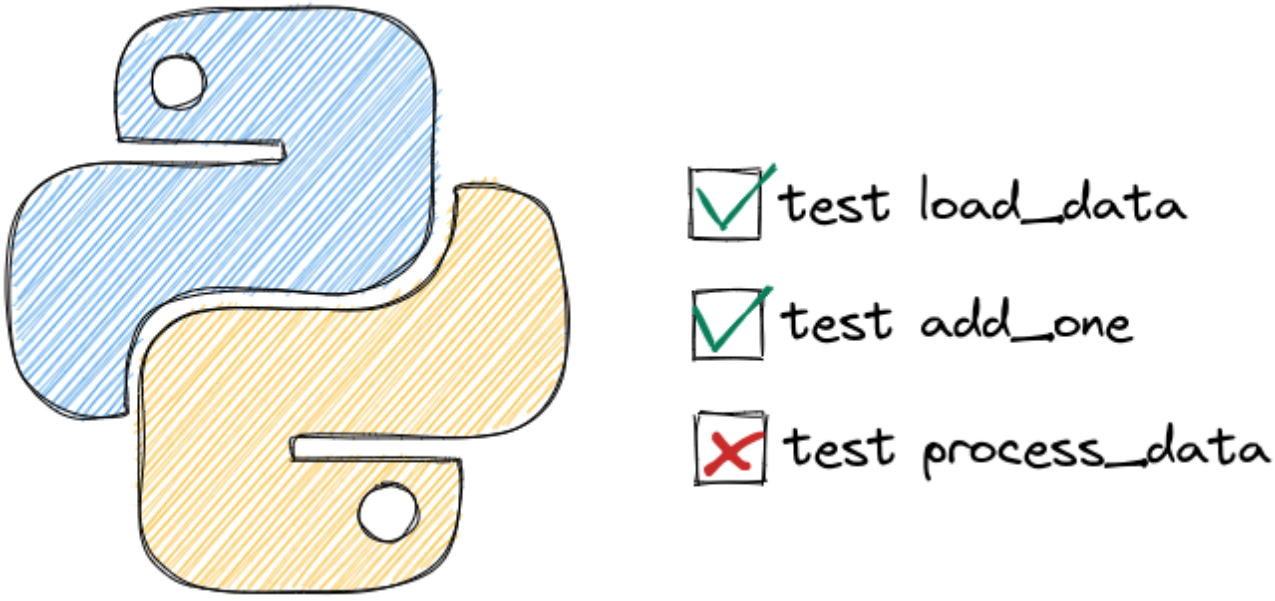
Efficient Python Tricks and Tools for Data Scientists

Testing - By Khuyen Tran



pytest benchmark: A Pytest Fixture to Benchmark Your Code

```
$ pip install pytest-benchmark
```

If you want to benchmark your code while testing with pytest, try pytest-benchmark.

To use pytest-benchmark works, add `benchmark` to the test function that you want to benchmark.

```
# pytest_benchmark_example.py
def list_comprehension(len_list=5):
    return [i for i in range(len_list)]


def test_concat(benchmark):
    res = benchmark(list_comprehension)
    assert res == [0, 1, 2, 3, 4]
```

On your terminal, type:

```
$ pytest pytest_benchmark_example.py
```

You should see the statistics of the time it takes to execute the test functions on your terminal:

```
pytest_benchmark_example.py .                                      [100%]

------------------------------------------------- benchmark: 1 tests -------------------------------------------
-----------
Name (time in ns)          Min         Max      Mean    StdDev    Median     IQR   Outliers  OPS (Mops/s)  Rounds
Iterations
----------------------------------------------------------------------------------------------------------------
-----------
test_concat           286.4501  4,745.5498  309.3872  106.6583  297.5001  5.3500  2686;5843        3.2322  162101
20
----------------------------------------------------------------------------------------------------------------
-----------
```

Link to pytest-benchmark.

pytest.mark.parametrize: Test Your Functions with Multiple Inputs

If you want to test your function with different examples, use `pytest.mark.parametrize` decorator.

To use pytest.mark.parametrize, add `@pytest.mark.parametrize` to the test function that you want to experiment with.

```
# pytest_parametrize.py
import pytest

def text_contain_word(word: str, text: str):
    '''Find whether the text contains a
particular word'''

    return word in text

test = [
    ('There is a duck in this text',True),
    ('There is nothing here', False)
    ]

@pytest.mark.parametrize('sample, expected',
test)
```

```
def test_text_contain_word(sample, expected):

    word = 'duck'

    assert text_contain_word(word, sample) ==
expected
```

In the code above, I expect the first sentence to contain the word "duck" and expect the second sentence not to contain that word. Let's see if my expectations are correct by running:

```
$ pytest pytest_parametrize.py
```

```
pytest_parametrize.py ..                                                [100%]

============================== 2 passed in 0.01s ===============================
```

Sweet! 2 tests passed when running pytest.

Link to my article about pytest.

pytest parametrize twice: Test All Possible Combinations of Two Sets of Parameters

If you want to test the combinations of two sets of parameters, writing all possible combinations can be time-consuming and is difficult to read.

```
import pytest

def average(n1, n2):
    return (n1 + n2) / 2

def perc_difference(n1, n2):
    return (n2 - n1)/n1 * 100

# Test the combinations of operations and
inputs
@pytest.mark.parametrize("operation, n1, n2",
[(average, 1, 2), (average, 2, 3),
(perc_difference, 1, 2), (perc_difference, 2,
3)])
def test_is_float(operation, n1, n2):
    assert isinstance(operation(n1, n2),
float)
```

You can save your time by using `pytest.mark.parametrize` twice instead.

```
# pytest_combination.py
import pytest

def average(n1, n2):
    return (n1 + n2) / 2

def perc_difference(n1, n2):
    return (n2 - n1)/n1 * 100

# Test the combinations of operations and
inputs
@pytest.mark.parametrize("operation",
[average, perc_difference])
@pytest.mark.parametrize("n1, n2", [(1, 2),
(2, 3)])
def test_is_float(operation, n1, n2):
    assert isinstance(operation(n1, n2),
float)
```

On your terminal, run:

```
$ pytest -v pytest_combination.py
```

```
pytest_combination.py::test_is_float[1-2-average] PASSED                [ 25%]
pytest_combination.py::test_is_float[1-2-perc_difference] PASSED        [ 50%]
pytest_combination.py::test_is_float[2-3-average] PASSED                [ 75%]
pytest_combination.py::test_is_float[2-3-perc_difference] PASSED        [100%]

============================== 4 passed in 0.27s ===============================
```

From the output above, we can see that all possible combinations of the given operations and inputs are tested.

Pytest Fixtures: Use The Same Data for Different Tests

If you want to use the same data to test different functions, use pytest fixtures.

To use pytest fixtures, add the decorator `@pytest.fixture` to the function that creates the data you want to reuse.

```
# pytest_fixture.py
import pytest
from textblob import TextBlob

def extract_sentiment(text: str):
    """Extract sentimetn using textblob.
Polarity is within range [-1, 1]"""

    text = TextBlob(text)
    return text.sentiment.polarity

@pytest.fixture
def example_data():
    return 'Today I found a duck and I am
happy'

def test_extract_sentiment(example_data):
```

```
    sentiment =
extract_sentiment(example_data)
    assert sentiment > 0
```

On your terminal, type:

```
$ pytest pytest_fixture.py
```

Output:

```
pytest_fixture.py .                                                  [100%]

============================== 1 passed in 0.53s ===============================
```

Pytest repeat

```
$ pip install pytest-repeat
```

It is a good practice to test your functions to make sure they work as expected, but sometimes you need to test 100 times until you found the rare cases when the test fails. That is when pytest-repeat comes in handy.

To use pytest-repeat, add the decorator `@pytest.mark.repeat(N)` to the test function you want to repeat `N` times

```
# pytest_repeat_example.py
import pytest
import random

def generate_numbers():
    return random.randint(1, 100)

@pytest.mark.repeat(100)
def test_generate_numbers():
    assert generate_numbers() > 1 and
generate_numbers() < 100
```

```python
# pytest_repeat_example.py
import pytest
import random

def generate_numbers():
    return random.randint(1, 100)

@pytest.mark.repeat(100)
def test_generate_numbers():
    assert generate_numbers() > 1 and
generate_numbers() < 100
```

On your terminal, type:

```
pytest pytest_repeat_example.py
```

We can see that 100 experiments are executed and passed:

```
pytest_repeat_example.py ..................................................... [ 47%]
.....................................................                          [100%]

============================== 100 passed in 0.07s ==============================
```

Link to pytest-repeat

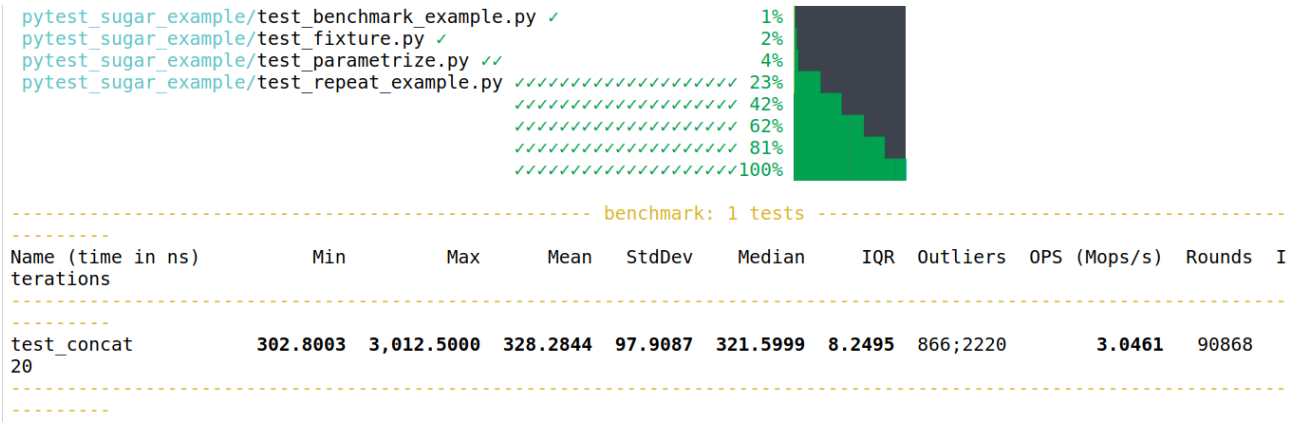
pytest-sugar: Show the Failures and Errors Instantly With a Progress Bar

```
$ pip install pytest-sugar
```

It can be frustrating to wait for a lot of tests to run before knowing the status of the tests. If you want to see the failures and errors instantly with a progress bar, use pytest-sugar.

pytest-sugar is a plugin for pytest. The code below shows how the outputs will look like when running pytest.

```
$ pytest
```



Link to pytest-sugar.

Pandera: a Python Library to Validate Your Pandas DataFrame

```
$ pip install pandera
```

The outputs of your pandas DataFrame might not be like what you expected either due to the error in your code or the change in the data format. Using data that is different from what you expected can cause errors or lead to decrease performance.

Thus, it is important to validate your data before using it. A good tool to validate pandas DataFrame is pandera. Pandera is easy to read and use.

```
import pandera as pa
from pandera import check_input
import pandas as pd

df = pd.DataFrame({"col1": [5.0, 8.0, 10.0],
"col2": ["text_1", "text_2", "text_3"]})
schema = pa.DataFrameSchema(
    {
        "col1": pa.Column(float,
pa.Check(lambda minute: 5 <= minute)),
        "col2": pa.Column(str,
pa.Check.str_startswith("text_")),
    }
)
validated_df = schema(df)
validated_df
```

	col1	col2
0	5.0	text_1
1	8.0	text_2
2	10.0	text_3

You can also use the Pandera's decorator `check_input` to validates input pandas DataFrame before entering the function.

```
@check_input(schema)
def plus_three(df):
    df["col1_plus_3"] = df["col1"] + 3
    return df


plus_three(df)
```

DeepDiff Find Deep Differences of Python Objects

```
$ pip install deepdiff
```

When testing the outputs of your functions, it can be frustrated to see your tests fail because of something you don't care too much about such as:

- order of items in a list
- different ways to specify the same thing such as abbreviation
- exact value up to the last decimal point, etc

Is there a way that you can exclude certain parts of the object from the comparison? That is when DeepDiff comes in handy.

```
from deepdiff import DeepDiff
```

DeepDiff can output a meaningful comparison like below:

```
price1 = {'apple': 2, 'orange': 3, 'banana':
[3, 2]}
price2 = {'apple': 2, 'orange': 3, 'banana':
[2, 3]}

DeepDiff(price1, price2)
```

```
{'values_changed': {"root['banana'][0]":
{'new_value': 2, 'old_value': 3},
  "root['banana'][1]": {'new_value': 3,
'old_value': 2}}}
```

With DeepDiff, you also have full control of which characteristics of the Python object DeepDiff should ignore. In the example below, since the order is ignored `[3, 2]` is equivalent to `[2, 3]`.

```
# Ignore orders

DeepDiff(price1, price2, ignore_order=True)
```

```
{}
```

We can also exclude certain part of our object from the comparison. In the code below, we ignore `ml` and `machine learning` since `ml` is a abbreviation of `machine learning`.

```
experience1 = {"machine learning": 2,
"python": 3}
experience2 = {"ml": 2, "python": 3}

DeepDiff(
    experience1,
    experience2,
    exclude_paths={"root['ml']",
"root['machine learning']"},
)
```

```
{}
```

Compare 2 numbers up to a specific decimal point:

```
num1 = 0.258
num2 = 0.259

DeepDiff(num1, num2, significant_digits=2)
```

```
{}
```

Link to DeepDiff.

hypothesis: Property-based Testing in Python

```
$ pip install hypothesis
```

If you want to test some properties or assumptions, it can be cumbersome to write a wide range of scenarios. To automatically run your tests against a wide range of scenarios and find edge cases in your code that you would otherwise have missed, use hypothesis.

In the code below, I test if the addition of two floats is commutative. The test fails when either `x` or `y` is `NaN`.

```
# test_hypothesis.py

from hypothesis import given
from hypothesis.strategies import floats

@given(floats(), floats())
def test_floats_are_commutative(x, y):
    assert x + y == y + x
```

```
$ pytest test_hypothesis.py
```

```
_____________________________ test_floats_are_commutative _____________________________

    @given(floats(), floats())
>   def test_floats_are_commutative(x, y):

test_hypothesis.py:7:
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

x = 0.0, y = nan

    @given(floats(), floats())
    def test_floats_are_commutative(x, y):
>       assert x + y == y + x
E       assert (0.0 + nan) == (nan + 0.0)

test_hypothesis.py:8: AssertionError
---------------------------------- Hypothesis ----------------------------------
Falsifying example: test_floats_are_commutative(
    x=0.0, y=nan,  # Saw 1 signaling NaN
)

 test_hypothesis.py ×                                                      100% ██████████
=========================== short test summary info ============================
FAILED test_hypothesis.py::test_floats_are_commutative - assert (0.0 + nan) =...

Results (0.38s):
       1 failed
         - test_hypothesis.py:6 test_floats_are_commutative
```

Now I can rewrite my code to make it more robust against these edge cases.

Link to hypothesis.

Deepchecks: Check Category Mismatch Between Train and Test Set

```
$ pip install deepchecks
```

Sometimes, it is important to know if your test set contains the same categories in the train set. If you want to check the category mismatch between the train and test set, use Deepchecks.

In the example below, the result shows that there are 2 new categories in the test set. They are 'd' and 'e'.

```
from deepchecks.checks.integrity.new_category
import CategoryMismatchTrainTest
from deepchecks.base import Dataset
import pandas as pd
```

```
train = pd.DataFrame({"col1": ["a", "b",
"c"]})
test = pd.DataFrame({"col1": ["c", "d", "e"]})

train_ds = Dataset(train, cat_features=
["col1"])
test_ds = Dataset(test, cat_features=["col1"])
```

```
CategoryMismatchTrainTest().run(train_ds, test_ds)
```

Category Mismatch Train Test

Find new categories in the test set.

Additional Outputs

Column	Number of new categories	Percent of new categories in sample	New categories examples
col1	2	66.67%	['d', 'e']

Link to Deepchecks