Testing

snoop : Smart Print to Debug Your Python Function

```
!pip install snoop
```

If you want to figure out what is happening in your code without adding many print statements, try snoop.

To use snoop, simply add the `@snoop` decorator to a function you want to understand.

```
import snoop

@snoop
def factorial(x: int):
    if x == 1:
        return 1
    else:
        return (x * factorial(x-1))
```

```
num = 2
print(f'The factorial of {num} is {factorial(num)}')
```

```
12:46:29.30 >>> Call to factorial in File
"/tmp/ipykernel_72663/20705592.py", line 4
12:46:29.30 ...... x = 2
12:46:29.30    4 | def factorial(x: int):
12:46:29.30    5 |     if x == 1:
12:46:29.30    8 |         return (x * factorial(x-1))
    12:46:29.30 >>> Call to factorial in File
"/tmp/ipykernel_72663/20705592.py", line 4
    12:46:29.30 ...... x = 1
    12:46:29.30    4 | def factorial(x: int):
    12:46:29.31    5 |     if x == 1:
    12:46:29.31    6 |         return 1
    12:46:29.31 <<< Return value from factorial: 1
12:46:29.31    8 |         return (x * factorial(x-1))
12:46:29.31 <<< Return value from factorial: 2
```

```
The factorial of 2 is 2
```

Link to my article about snoop.

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
```

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

```
------------------------------------ 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
-----------------------------------------------------------------
-----------------------------------------

Legend:
  Outliers: 1 Standard Deviation from Mean; 1.5 IQR
(InterQuartile Range) from 1st Quartile and 3rd Quartile.
  OPS: Operations Per Second, computed as 1 / Mean
============================== 1 passed in 2.47s
==============================
```

Link to pytest-benchmark.

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

```
!pip install pytest
```

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 Fixtures: Use The Same Data for Different Tests

```
!pip install pytest
```

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
```

On your terminal, type:

```
pytest pytest_repeat_example.py
```

We can see that 100 experiments are executed and passed:

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

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

Link to pytest-repeat

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)
```

	col1	col2	col1_plus_3
0	5.0	text_1	8.0
1	8.0	text_2	11.0
2	10.0	text_3	13.0

Link to Pandera