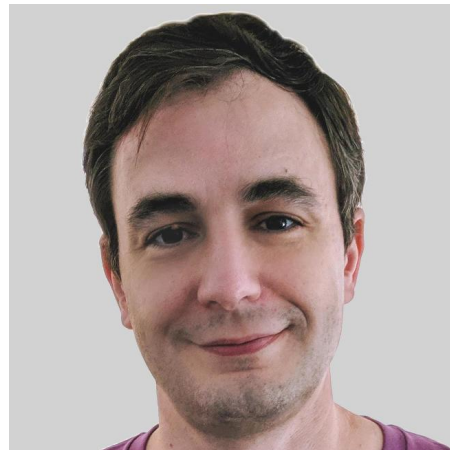# Testing in Python, from 0 to 100

Julio Martínez

About me

- Developing software since 2001

- Worked in retail, adtech, web, industrial sw

- Find me on Internet: @liopic

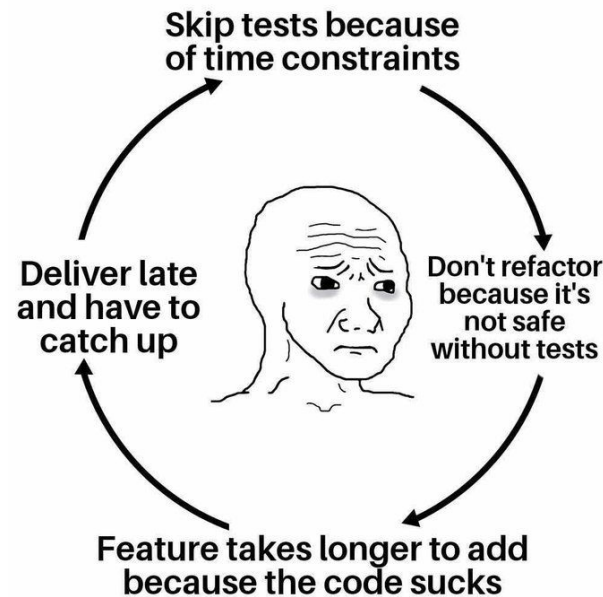- Focus on engineering good practices

Pragmatic approach to Testing

- Do we need tests?

- It depends!

  - **YES** : Team-work, company product, quality work

  - **NO** : PoC, web freelance, one run, pet project?

Do we need tests?

- **Key point: Maintenance!**

- Tests give lots of benefits to our development

  - (In theory) guarantee code quality

  - Allow **other team members** to **safely** work on the code

Skip tests because of time constraints

Don't refactor because it's not safe without tests

Feature takes longer to add because the code sucks

Deliver late and have to catch up

Worst team member is yourself in the future

Pragmatic approach to Testing

- What's testing?

- What's a good test?

```python
class Something:
    def is_testing_do_a_print(self):
        return "NO"


if __name__ == "__main__":
    print(Something().is_testing_do_a_print())
```

What's testing?

- In theory "Write tests to validate the functionality of the code"

- **Key: AUTOMATIC!**

    - Continuous Integration: github actions, gitlab, circleCI, travis, jenkins

- Tests also are documentation

    - Showing : examples of code use, happy path, variations, edge cases, pitfalls

- Test Driven Development

    - vs Coding+Testing

What's a good test?

- One that helps developers/maintainers

- Features:

    - Easy to understand

    - Test only ONE thing

    - Minimal/concise

        - Given - When - Then

WHAT ABOUT ME?

Tests in Python

- Included:

    - unittest (unittest.TestCase)

    - doctest (X)

- Most used package:

    - pytest

Code Example

```python
from datetime import datetime, timedelta
from math import ceil



class MoonLocator:
    moon_on_south = datetime(year=2022, month=11, day=15, hour=6, minute=1, second=12)
    moon_interval = timedelta(days=1, hours=0, minutes=50, seconds=28)

    def next_moon_on_south_from_date(self, from_date: datetime) -> datetime:
        since_moon_on_south = from_date - self.moon_on_south
        cycle = ceil(since_moon_on_south / self.moon_interval)
        return self.moon_on_south + self.moon_interval * cycle
```

Test Example

```python
from datetime import datetime, timedelta

from moon import MoonLocator


def test_date_on_south_a_second_before_should_return_that_date():
    sut = MoonLocator()
    some_moon_on_south = datetime(year=2022, month=11, day=15, hour=6, minute=1, second=12)
    a_moment_before = some_moon_on_south - timedelta(seconds=1)

    next_moon = sut.next_moon_on_south_from_date(a_moment_before)

    assert next_moon == some_moon_on_south
```

Another Test Example: randomness

```python
def test_next_moon_should_be_between_a_date_and_one_day_one_hour_later():
    sut = MoonLocator()
    some_date = datetime(2022, 12, 14, 0, 0, 0) # This could be random!
    one_day_one_hour_later = some_date + timedelta(days=1, hours=1)

    next_moon = sut.next_moon_on_south_from_date(some_date)

    assert next_moon >= some_date
    assert one_day_one_hour_later >= next_moon
```

Snapshot testing? No flexibility

```python
def test_next_moon_on_south():
    sut = MoonLocator()
    some_date = datetime(2022, 12, 14, 0, 0, 0)

    next_moon = sut.next_moon_on_south_from_date(some_date)

    assert next_moon == datetime(2022, 12, 14, 5, 34, 16)
```

Test with fixture

```python
@pytest.fixture(scope="module")
def moon_locator():
    return MoonLocator()


@pytest.fixture
def moon_on_south():
    yield datetime(year=2022, month=11, day=15, hour=6, minute=1, second=12)


def test_date_on_south_a_second_before_should_return_that_date(moon_locator, moon_on_south):
    a_moment_before = moon_on_south - timedelta(seconds=1)

    next_moon = moon_locator.next_moon_on_south_from_date(a_moment_before)

    assert next_moon == moon_on_south
```

Example with dependency

```python
def reference_date_for_year(year) -> datetime:
    if year >= 2022:
        return datetime(2022, 11, 15, 6, 1, 12)
    else:
        return datetime(2010, 1, 2, 2, 32, 4)


class MoonLocator:
    moon_interval = timedelta(days=1, hours=0, minutes=50, seconds=28)

    def next_moon_on_south_from_date(self, from_date: datetime) -> datetime:
        moon_on_south = reference_date_for_year(from_date.year)
        since_moon_on_south = from_date - moon_on_south
        cycle = ceil(since_moon_on_south / self.moon_interval)
        return moon_on_south + self.moon_interval * cycle
```

Test with mock

```python
from unittest import mock

from moon_locator import MoonLocator

namespace = "moon_locator"
@mock.patch(f"{namespace}.reference_date_for_year", return_value=datetime(2020, 2, 1))
def test_date_on_south_a_second_before_should_return_that_date(mock_reference, moon_locator):
    some_moon = datetime(year=2020, month=2, day=1)
    a_moment_before = some_moon - timedelta(seconds=1)

    next_moon = moon_locator.next_moon_on_south_from_date(a_moment_before)

    assert next_moon == some_moon
    mock_reference.assert_any_call(2020)
```

Advanced testing

- Quality of tests

    - Test coverage (pytest-cov)

    - Mutant testing (mutmut)

- Other ways to do tests

    - Property-based testing (hypothesis)

    - BDD testing (behave)

Test coverage

- Do your tests cover all the code?

    - Lines of code

    - Paths of execution

- pytest-cov gives you statement-based reports

```
count = 0
if some_thing:
    count += 1
if other_thing:
    count += 10
```

Coverage for **moon.py**: 91%

```python
1  from datetime import datetime, timedelta
2  from math import ceil
3
4
5  class MoonLocator:
6      moon_on_south = datetime(year=2022, month=11, day=15, hour=6, minute=1, second=12)
7      moon_interval = timedelta(days=1, hours=0, minutes=50, seconds=28)
8
9      def next_moon_on_south_from_date(self, from_date: datetime) -> datetime:
10         since_moon_on_south = from_date - self.moon_on_south
11         cycle = ceil(since_moon_on_south / self.moon_interval)
12         return self.moon_on_south + self.moon_interval * cycle
13
14     def other_untested_function() -> int:
15         0
```

Mutant testing (mutmut)

- Test your tests

- Mutate the original code...

  - Add 1 to numbers

  - Change < to <=

  - ... other changes

- ... and execute the tests

Mutant testing (mutmut)

🎉 Killed mutants.   The goal is for everything to end up in this bucket.

⏰ Timeout.   Test suite took 10 times as long as the baseline so were killed.

🤔 Suspicious.   Tests took a long time, but not long enough to be fatal.

🙁 Survived.   This means your tests needs to be expanded.

🚫 Skipped.   Skipped.

1. Running tests without mutations

∴ Running...Done

2. Checking mutants

∴ 136/136 🎉 90 ⏰ 0 🤔 0 🙁 46 🚫 0

Mutant testing (mutmut)

- Good vs Bad

    - + Captures edge cases

    - + "Clever coverage"

    - - Make tests not flexible

    - - Slow

- Recommendation: just pieces of code with complexity

Take home key idea

- Create one test

- Execute your test, **automatically (CI)**

- Iterate from this base

THANK YOU!

Extra testing methods...

Property-based testing (hypothesis)

```python
imfrom hypothesis import given, settings
import hypothesis.strategies as st

from moon_locator import MoonLocator

datetimes = st.datetimes(max_value=datetime(2100, 1, 1))

@given(datetimes)
@settings(max_examples=5000)
def test_next_moon_should_be_after_some_given_date(date):
    moon_locator = MoonLocator()
    assert date <= moon_locator.next_moon_on_south_from_date(date)
```

Property-based testing (hypothesis)

- Good vs Bad

  - \+ Finds unexpected cases

  - \+ hypothesis can find the minimum error

  - \- Force the developer to stop and fix unrelated code

- Recommendation: Build random stubs (Faker)

BDD testing (behave)

- An interface with product/stakeholders

```
Feature: next moon calculation

  Scenario: next moon in December
    Given we set last moon to "2022-11-15 19:00:00"
     When we ask for the next moon on South from today
     Then the next moon on South should be in the future
```

BDD testing (behave)

```python
@given('we set last moon on South to "{date}"')
def step_impl(context, date):
    last_south = datetime.strptime(date, '%Y-%m-%d %H:%M:%S')
    context.moon_locator = MoonLocator()
    context.moon_locator.default_moon = last_south

@when('we ask for the next moon on South from today')
def step_impl(context):
    context.today = datetime.now()
    context.next_moon = context.moon_locator.next_moon_on_south_from_date(context.today)

@then('the next moon on South should be in the future')
def step_impl(context):
    assert context.today <= context.next_moon
```

BDD testing (behave)

- Good vs Bad

  - + Text is highly configurable

  - + High level testing

  - - You need a diligent product team

- Recommendation: try it with easy-to-implement features