
Testes
Luiza
<CODE>
Luizalabs
MAGALU

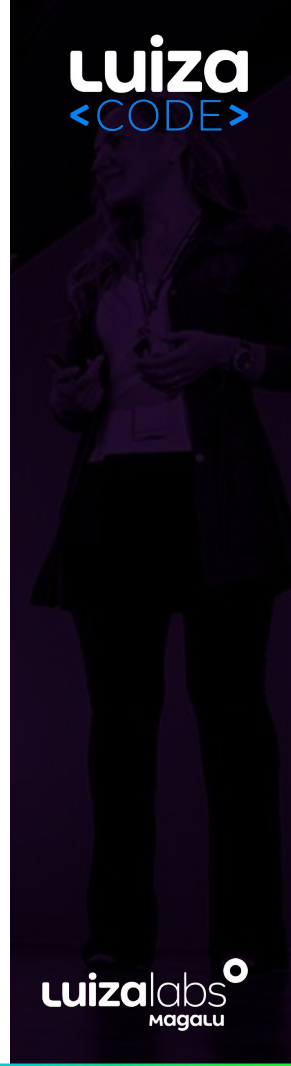
Ementa:

1. Testes automatizados vs manuais
2. Testes unitários vs testes de integração
3. Frameworks de testes
4. Pytest
5. Testes unitários
6. Mock

1. Testes automatizados vs testes manuais

O teste exploratório é uma forma de teste que é feita sem um plano. Por exemplo, quando você executou seu aplicativo e o usou pela primeira vez, também verificou os recursos e experimentou usá-los. Isso é conhecido como teste exploratório e é uma forma de teste manual.

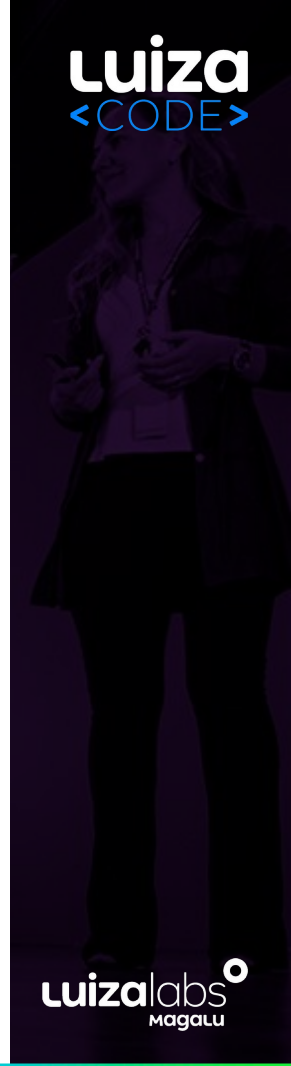
Para ter um conjunto completo de testes manuais, tudo o que você precisa fazer é uma lista de todos os recursos que seu aplicativo possui, os diferentes tipos de entrada que ele pode aceitar e os resultados esperados. E toda vez que fizer uma alteração no seu código, é preciso passar por todos os itens dessa lista e verificá-los.



1. Testes automatizados vs testes manuais

O teste automatizado é a execução do seu plano de teste (as partes do seu aplicativo que você deseja testar, a ordem em que deseja testá-las e as respostas esperadas) por um script em vez de um humano .

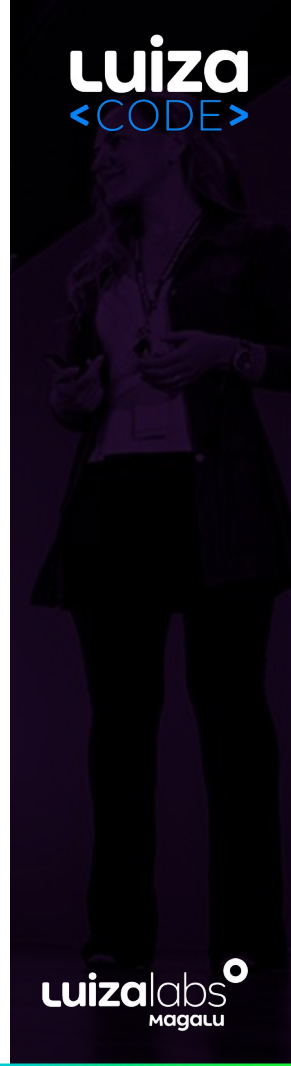
O Python já vem com um conjunto de ferramentas e bibliotecas para ajudá-lo a criar testes automatizados para seu aplicativo. Vamos explorar essas ferramentas e bibliotecas.



2. Testes unitários vs testes de integração

Como você pode testar as luzes de um carro? Você acenderia as luzes (etapa de teste) e sairia do carro para verificar se as luzes estão acesas (afirmação de teste). O teste de vários componentes é conhecido como teste de integração.

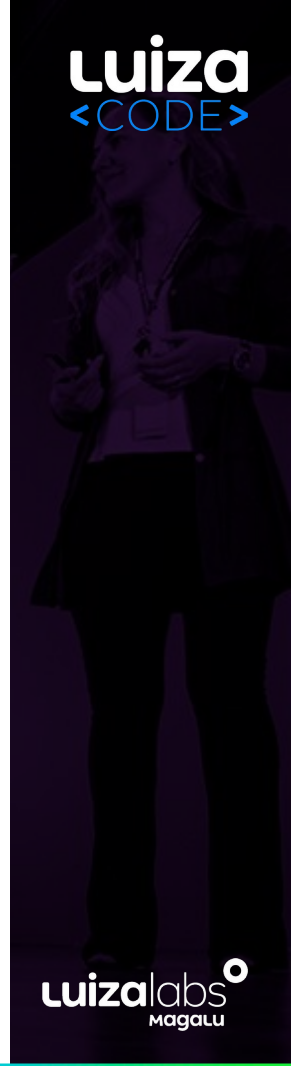
E todas as coisas que precisam funcionar corretamente para que uma tarefa simples dê o resultado certo, são como as partes do seu aplicativo, todas essas classes, funções e módulos que você escreveu.



2. Testes unitários vs testes de integração

Alguns modelos de carros, lhe dirão quando suas lâmpadas se apagarem, isso é feito usando uma forma de teste de unidade.

Um teste de unidade é um teste menor, que verifica se um único componente funciona da maneira correta. E também ajuda a isolar o que está com problema em seu aplicativo e corrigi-lo mais rapidamente.



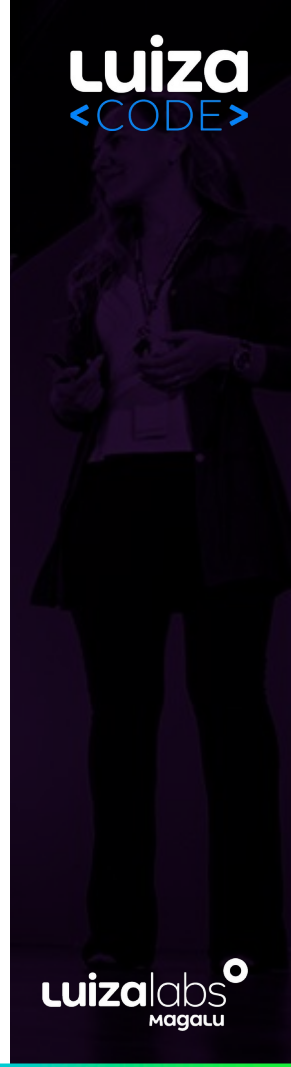
3. Frameworks de testes

4. Pytest

4.1 Instalação

Para instalar o framework que iremos utilizar, execute o comando abaixo no seu ambiente virtual:

```
pip install pytest
```



4. Pytest

4.2 Funcionamento pytest

Quando se executa o comando pytest dentro do ambiente virtual, ele faz um scan nos diretórios e subdiretórios do repositório procurando por arquivos no formato de nomenclatura **test_*.py** ou ***_test.py**. É recomentado usar o padrão **__init__.py** em cada diretório para o pytest reconhecê-los como módulos.

Exemplo de estrutura para organização do código:

```
tests/
    __init__.py
    user/
        __init__.py
        test_new_user.py
    store/
        __init__.py
        test_new_store.py
        test_delete_store.py
        some_helper.py
```

Rodar os testes:
`pytest`
Rodar somente os testes do diretório store:
`pytest tests/store`
Rodar somente um arquivo de teste:
`pytest tests/store/test_new_store.py`
Rodar somente um método de teste:
`pytest tests/store/test_new_store.py::test_metodo_de_teste`

5. Testes unitários

5.1 Executando o primeiro teste

Exemplo de um método que retorna a soma de dois número.

```python
def sum(x, y):
    return x + y

def test_sum():
    assert 5 == sum(2, 3)
```

```
(venv) michelly@michelly-Latitude-3490:~/Documentos/Luiza Code/Códigos$ pytest test_example.py
============================================================================== test session starts
platform linux -- Python 3.10.4, pytest-7.1.3, pluggy-1.0.0
rootdir: /home/michelly/Documentos/Luiza Code/Códigos
collected 1 item

test_example.py .

============================================================================== 1 passed in 0.01s
```

5. Testes unitários

5.2 Forçando um erro no teste

O assert foi modificado para um resultado incorreto, resultado 6.

```python
def sum(x, y):
    return x + y

def test_sum():
    assert 6 == sum(2, 3)
```

```
(venv) michelly@michelly-Latitude-3490:~/Documentos/Luiza Code/Códigos$ pytest test_example.py
============================= test session starts ==============================
platform linux -- Python 3.10.4, pytest-7.1.3, pluggy-1.0.0
rootdir: /home/michelly/Documentos/Luiza Code/Códigos
collected 1 item

test_example.py F

=================================== FAILURES ===================================
_________________________________ test_sum _________________________________

    def test_sum():
>       assert 6 == sum(2, 3)
E       assert 6 == 5
E        +  where 5 = sum(2, 3)

test_example.py:5: AssertionError
=========================== short test summary info ============================
FAILED test_example.py::test_sum - assert 6 == 5
============================== 1 failed in 0.02s ===============================
```

5. Testes unitários

5.3 Executando vários testes

Agora o arquivo test_example.py está com três métodos e três testes referentes aos métodos.

```python
def sum(x, y):
    return x + y

def multiplies(x, y):
    return x * y

def divide(x, y):
    return x / y

def test_sum():
    assert 5 == sum(2, 3)

def test_multiplies():
    assert 6 == multiplies(2, 3)

def test_divide():
    assert 1.5 == divide(3, 2)
```

```
(venv) michelly@michelly-Latitude-3490:~/Documentos/Luiza Code/Códigos$ pytest test_example.py
============================================================================== test session starts
platform linux -- Python 3.10.4, pytest-7.1.3, pluggy-1.0.0
rootdir: /home/michelly/Documentos/Luiza Code/Códigos
collected 3 items

test_example.py ...

============================================================================== 3 passed in 0.01s =
```

5. Testes unitários

5.4 Mock dependências

Mockar é uma forma que ajuda a escrever casos de testes independentes e rápidos, sem depender de resultados externos.

Exemplo de um método que retorna o período do dia atual.

```python
from datetime import datetime

def get_time_of_day():
    time = datetime.now()

    if 0 <= time.hour <6:
        return "Night"
    if 6 <= time.hour < 12:
        return "Morning"
    if 12 <= time.hour <18:
        return "Afternoon"
    return "Evening"
```

5. Testes unitários

5.4 Mock dependências

Instalação: pip install pytest-mock

Estamos fazendo um mock de data para forçar o retorno do período **Afternoon**.

```python
import pytest
from datetime import datetime
from src.example import get_time_of_day

def test_get_time_of_day_afternoon(mocker):
    mock_now = mocker.patch("src.example.datetime")
    mock_now.now.return_value = datetime(2022, 9, 10, 14, 10, 0)

    assert get_time_of_day() == "Afternoon"
```

5. Testes unitários

5.5 Mock dependências e parametrização

Agora estamos parametrizando vários mocks de data para forçar todos os tipos de retornos.

```python
@pytest.mark.parametrize(
    "datetime_obj, expect",
    [
        (datetime(2016, 5, 20, 0, 0, 0), "Night"),
        (datetime(2016, 5, 20, 1, 10, 0), "Night"),
        (datetime(2016, 5, 20, 6, 10, 0), "Morning"),
        (datetime(2016, 5, 20, 12, 0, 0), "Afternoon"),
        (datetime(2016, 5, 20, 14, 10, 0), "Afternoon"),
        (datetime(2016, 5, 20, 18, 0, 0), "Evening"),
        (datetime(2016, 5, 20, 19, 10, 0), "Evening"),
    ],
)

def test_get_time_of_day(datetime_obj, expect, mocker):
    mock_now = mocker.patch("src.example.datetime")
    mock_now.now.return_value = datetime_obj

    assert get_time_of_day() == expect
```

5. Testes unitários

5.6 Mock env

Alguns testes exigem configuração do ambiente, pode ser uma conexão de banco de dados, acesso à rede ou configuração de variável de ambiente. O monkeypatch ajuda a definir/excluir um atributo, item de dicionário ou variável de ambiente.

Exemplo de um método que verifica se a env var **CONTRACT**_CLASS é igual algum dos valores e retorna uma string, se não retorna uma .

```python
import os

def use_env_var():
    contract_class = os.environ['CONTRACT_CLASS']

    if contract_class == 'en_cloud':
        return "This is en_cloud"
    if contract_class == 'en_onprem':
        return "This is en_onprem"
    raise ValueError(f"Contract class {contract_class} not found!")
```

5. Testes unitários

5.6 Mock env

Agora temos um teste que está mockando a env var **CONTRACT_CLASS** com os dois valores possíveis.

```python
import pytest
from src.example import use_env_var


@pytest.mark.parametrize(
    "mock_contract_class,expect", [("en_cloud", "This is en_cloud"), ("en_onprem", "This is en_onprem")]
)
def test_mock_env_var(mock_contract_class, expect, monkeypatch):
    monkeypatch.setenv("CONTRACT_CLASS", mock_contract_class)
    assert use_env_var() == expect
```

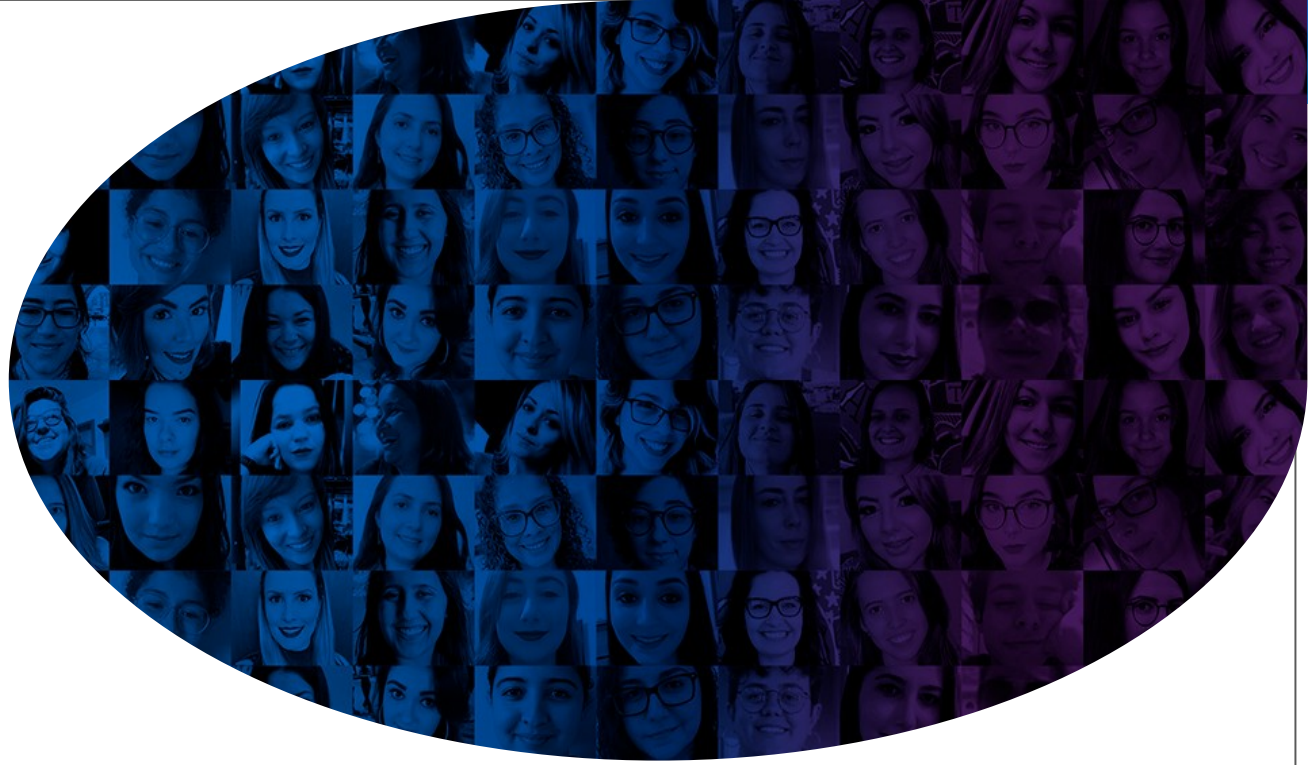
5. Testes unitários

5.7 Mock Exception

E por fim, temos um teste que está mockando a env var **CONTRACT_CLASS** com um valor que retorna a exception.

```python
import pytest
from src.example import use_env_var


def test_exception(monkeypatch):
    monkeypatch.setenv("CONTRACT_CLASS", "Something not existed")
    with pytest.raises(ValueError, match=r"Contract class Something not existed not found!"):
        use_env_var()
```



Perguntas?

#VemSerFeliz