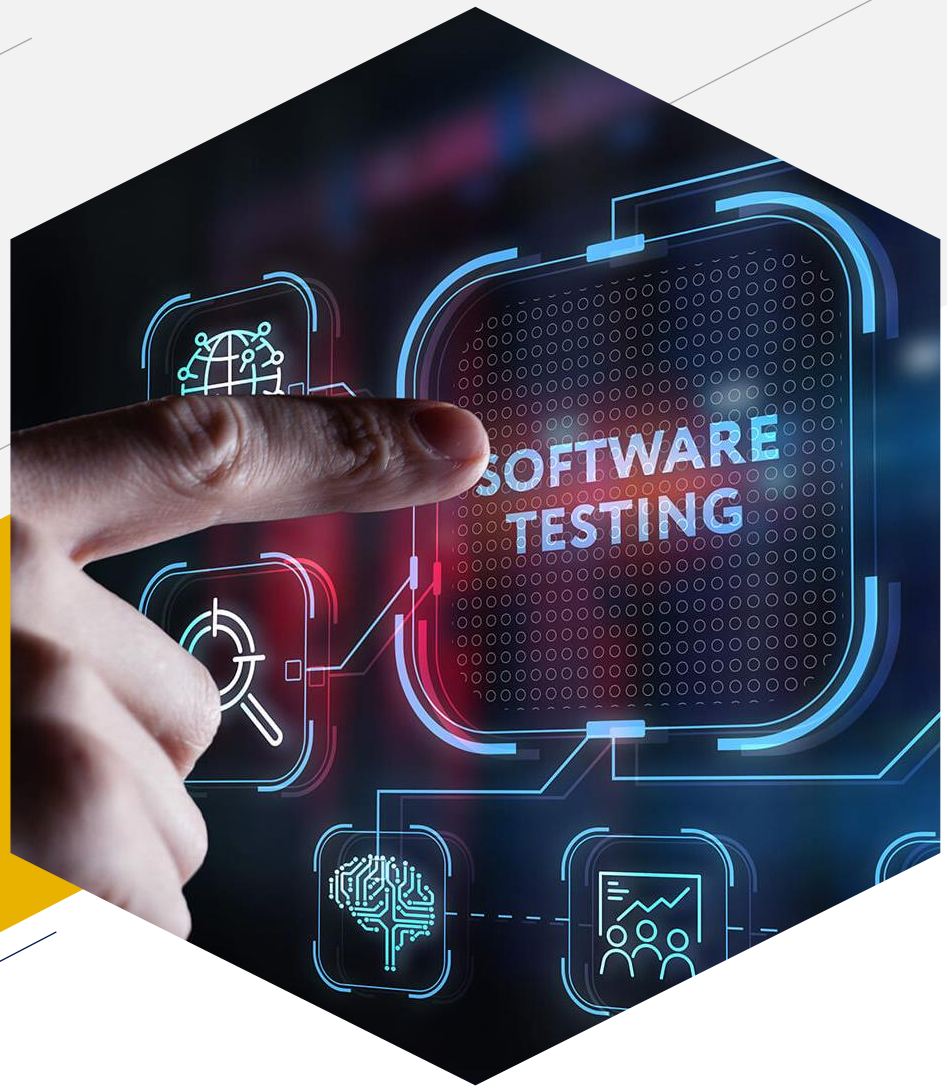


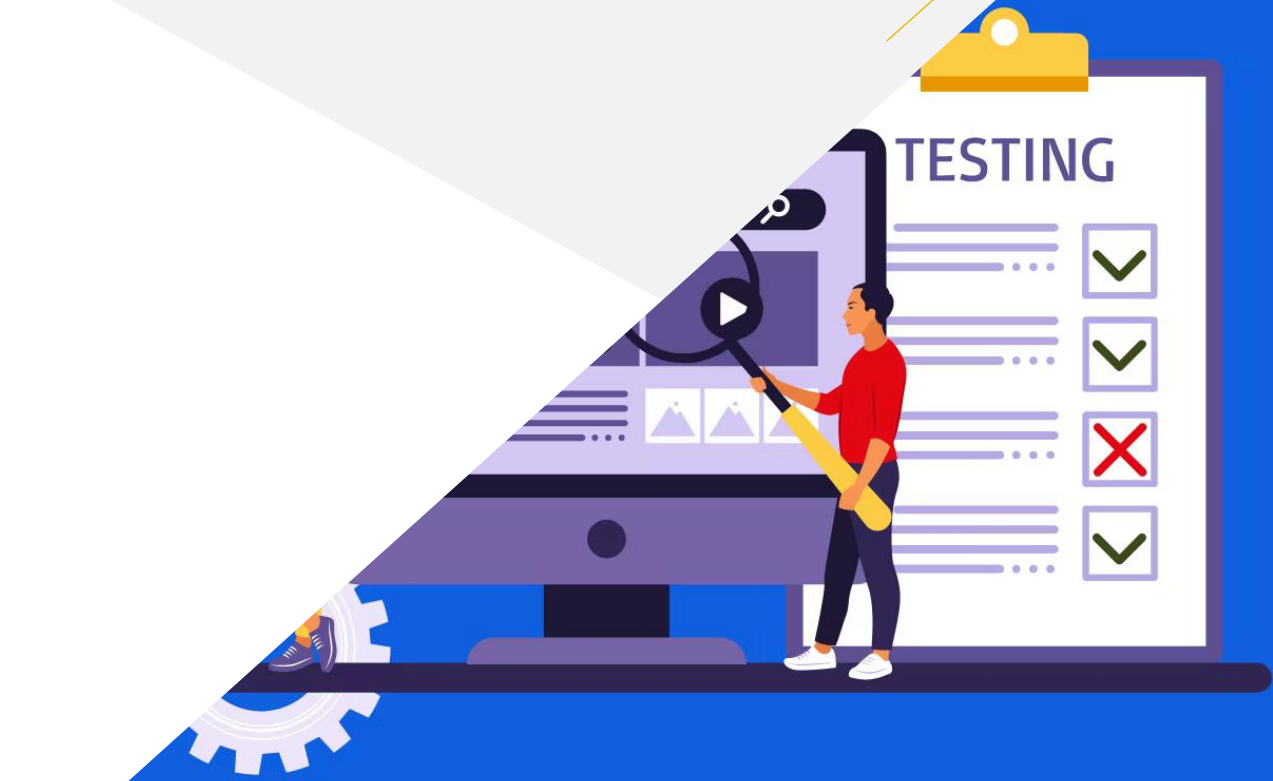
Executar Teste e Implantação de Aplicativos Computacionais

SENAC PE

28 de Setembro de 2024

Testes com Pytest

- Introdução



Introdução

- Pytest é uma das estruturas e ferramentas de teste mais populares para Python. Embora Pytest possa ajudar com cenários de teste altamente complexos, ele não força seus recursos ao criar testes. Você pode escrever testes simples e ainda se beneficiar do executor de teste rápido e com recursos e relatórios úteis.
- Um aspecto crucial do Pytest é que facilita a escrita de testes. Você pode escrever uma função de teste sem dependências nem configurações e executar o teste imediatamente.

Testes com Pytest

- Noções básicas de Pytest

Noções básicas – Convenções

- Antes de mergulhar na escrita de testes, precisamos tratar de algumas das convenções de teste nas quais Pytest se baseia.
- Não há regras rígidas sobre arquivos de teste, diretórios de teste ou layouts de teste em geral no Python. Conhecendo essas regras, você pode aproveitar a detecção e a execução de testes automáticas sem necessidade de nenhuma configuração extra.

Noções básicas – diretório e arquivos

- O diretório principal para os testes é o diretório ***tests***. Você pode posicionar esse diretório no nível raiz do projeto, mas também não é incomum vê-lo ao lado dos módulos de código.
- Como fica a raiz de um pequeno projeto em Python chamado jformat:

- O diretório ***tests*** está na raiz do projeto com um só arquivo de teste. Nesse caso, o arquivo de teste é chamado test_main.py. Este exemplo demonstra duas convenções críticas:
  - Use um diretório *testes* para posicionar arquivos de teste e diretórios de teste aninhados.
  - Usar o prefixo *test* nos arquivos de teste. O prefixo indica que o arquivo contém código de teste.
- [CUIDADO] - Evite usar **test** (forma singular) como nome do diretório. O nome test é um módulo de Python, portanto, criar um diretório com o mesmo nome o substituiria. Em vez disso, use sempre o plural tests.

Noções básicas – Testar funções

- Um dos argumentos fortes para usar o Pytest é que ele permite gravar funções de teste. De modo semelhante aos arquivos de teste, as funções de teste devem ser prefixadas com **test_**. O prefixo **test_** garante que o Pytest colete o teste e o execute.

```
def test_main():
    assert "a string value" == "a string value"
```

Noções básicas – Classes e métodos

- De modo semelhante às convenções para arquivos e funções, as classes de teste e os métodos de teste usam as seguintes convenções:
  - As classes de teste têm o prefixo **Test**
  - Os métodos de teste têm o prefixo test_
- O exemplo a seguir usa esses prefixos e outras convenções de nomenclatura do Python para classes e métodos. Ele demonstra uma pequena classe de teste que verifica nomes de usuário em um aplicativo.

```
class TestUser:

    def test_username(self):
        assert default() == "default username"
```

Noções básicas – Executar testes

- Pytest é uma estrutura de teste e um executor de testes. O executor de testes é um executável na linha de comando que, em alto nível, pode:
  - Realize a coleção de testes encontrando todos os arquivos de teste, classes de teste e funções de teste para uma execução de teste.
  - Inicie uma execução de teste executando todos os testes.
  - Acompanhe as falhas, os erros e os testes aprovados.
  - Forneça relatórios avançados ao final de uma execução de teste.

Noções básicas – Executar testes

- Considerando esse conteúdo em um arquivo *test_main.py*, podemos ver como Pytest se comporta executando os testes:

```
# contents of test_main.py file

def test_main():
    assert True
```

- Na linha de comando, no mesmo caminho em que existe o arquivo *test_main.py*, podemos executar o executável **pytest**:

```
$ pytest
============================= test session starts ==============================
platform -- Python 3.10.1, pytest-6.2.5, py-1.11.0, pluggy-1.0.0
rootdir: /private/tmp/project
collected 1 item

test_main.py .                                                           [100%]

============================== 1 passed in 0.00s ===============================
```

- Nos bastidores, o Pytest coleta o teste de exemplo no arquivo de teste sem precisar de nenhuma configuração.

Noções básicas – instrução assert

- Os exemplos de teste estão todos usando a chamada **assert** simples. Normalmente, no Python, a instrução **assert** não é usada para testes porque não há um relato adequado quando a declaração falha.
- O Pytest, entretanto, não tem essa limitação. Nos bastidores, Pytest está habilitando a instrução a executar comparações avançadas sem forçar o usuário a escrever mais código ou configurar algo.
- Usando a instrução **assert** sem formatação, você pode usar os operadores de Python. Por exemplo >, <, !=, >= ou <=. Todos os operadores de Python são válidos. Esta funcionalidade pode ser o recurso mais crucial de Pytest: você não precisa aprender uma nova sintaxe para escrever declarações.

Noções básicas – instrução assert

- Vamos ver como isso se traduz ao lidar com comparações comuns com objetos de Python. Neste caso, vamos examinar o relatório de falhas ao comparar cadeia de caracteres longas:

```
================================ FAILURES ================================
____________________________ test_long_strings _____________________________

    def test_long_strings():
        left = "this is a very long strings to be compared with another long string"
        right = "This is a very long string to be compared with another long string"
>       assert left == right
E       AssertionError: assert 'this is a ve...r long string' == 'This is a ve...r long string'
E         - This is a very long string to be compared with another long string
E         ? ^
E         + this is a very long strings to be compared with another long string
E         ? ^                          +

test_main.py:4: AssertionError
```

- Pytest mostra um contexto útil em torno da falha. O uso incorreto de maiúsculas e minúsculas no início da cadeia de caracteres e um caractere extra em uma palavra.

Noções básicas – instrução assert

- Pytest pode ajudar com outros objetos e estruturas de dados. Por exemplo, veja como ele se comporta com listas:

```
_____________________________ test_lists ______________________________

    def test_lists():
        left = ["sugar", "wheat", "coffee", "salt", "water", "milk"]
        right = ["sugar", "coffee", "wheat", "salt", "water", "milk"]
>       assert left == right
E       AssertionError: assert ['sugar', 'wh...ater', 'milk'] == ['sugar', 'co...ater', 'milk']
E         At index 1 diff: 'wheat' != 'coffee'
E         Full diff:
E         - ['sugar', 'coffee', 'wheat', 'salt', 'water', 'milk']
E         ?                     ---------
E         + ['sugar', 'wheat', 'coffee', 'salt', 'water', 'milk']
E         ?           ++++++++

test_main.py:9: AssertionError
```

- Este relatório identifica que o índice 1 (segundo item na lista) é diferente. Ele não apenas identifica o número do índice, mas também fornece uma representação da falha.

Noções básicas – instrução assert

- Além das comparações entre itens, ele também pode relatar se itens estão ausentes e fornecer informações que podem informar exatamente qual pode ser o item. No caso a seguir, isso será "**milk**":

```
_____________________________ test_lists ______________________________

    def test_lists():
        left = ["sugar", "wheat", "coffee", "salt", "water", "milk"]
        right = ["sugar", "wheat", "salt", "water", "milk"]
>       assert left == right
E       AssertionError: assert ['sugar', 'wh...ater', 'milk'] == ['sugar', 'wh...ater', 'milk']
E         At index 2 diff: 'coffee' != 'salt'
E         Left contains one more item: 'milk'
E         Full diff:
E         - ['sugar', 'wheat', 'salt', 'water', 'milk']
E         + ['sugar', 'wheat', 'coffee', 'salt', 'water', 'milk']
E         ?                    ++++++++++

test_main.py:9: AssertionError
```

Noções básicas – instrução assert

- Por fim, vamos ver como ele se comporta com dicionários. A comparação de dois dicionários grandes pode ser esmagadora se existirem falhas, mas o Pytest faz um excelente trabalho ao fornecer o contexto e identificar a falha:

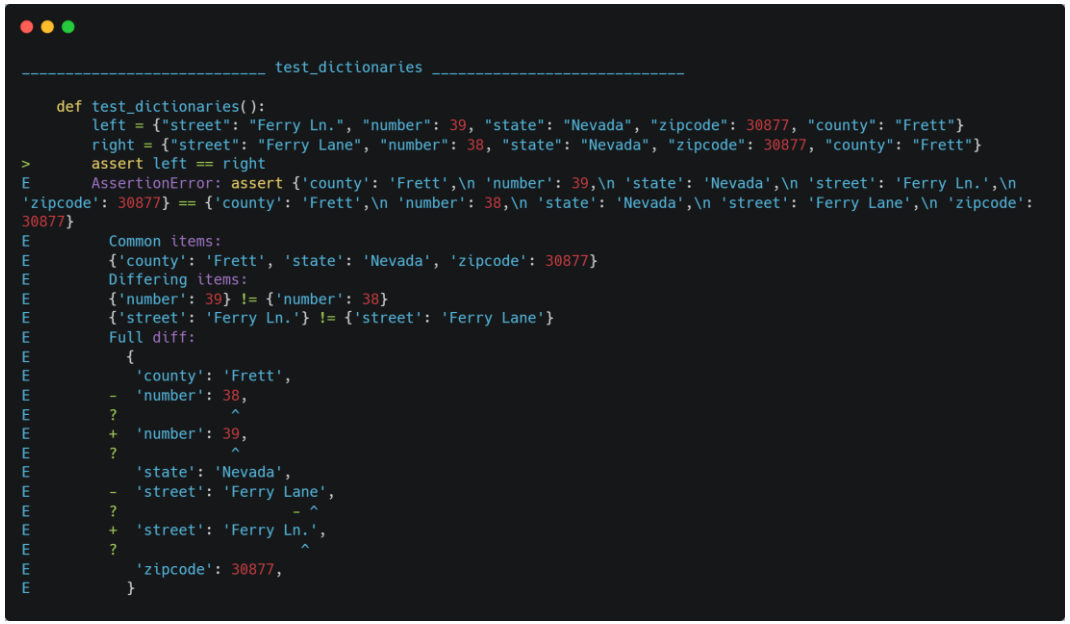
```
_____________________________ test_dictionaries _____________________________

    def test_dictionaries():
        left = {"street": "Ferry Ln.", "number": 39, "state": "Nevada", "zipcode": 30877, "county": "Frett"}
        right = {"street": "Ferry Lane", "number": 38, "state": "Nevada", "zipcode": 30877, "county": "Frett"}
>       assert left == right
E       AssertionError: assert {'county': 'F...rry Ln.', ...} == {'county': 'F...ry Lane', ...}
E         Omitting 3 identical items, use -vv to show
E         Differing items:
E         {'street': 'Ferry Ln.'} != {'street': 'Ferry Lane'}
E         {'number': 39} != {'number': 38}
E         Full diff:
E           {
E            'county': 'Frett',...
E
E         ...Full output truncated (12 lines hidden), use '-vv' to show
```

- Neste teste, há duas falhas no dicionário. Uma é que o valor de "**street**" é diferente, e outra é que "**number**" não corresponde.

Noções básicas – instrução assert

- O Pytest está detectando com precisão essas diferenças (mesmo que seja uma falha em um único teste). Como os dicionários contêm muitos itens, Pytest omite as partes idênticas e mostra apenas o conteúdo relevante. Vamos ver o que acontece quando usamos o sinalizador **-vv** sugerido para aumentar o detalhamento na saída:



```
_____________________________ test_dictionaries _____________________________

    def test_dictionaries():
        left = {"street": "Ferry Ln.", "number": 39, "state": "Nevada", "zipcode": 30877, "county": "Frett"}
        right = {"street": "Ferry Lane", "number": 38, "state": "Nevada", "zipcode": 30877, "county": "Frett"}
>       assert left == right
E       AssertionError: assert {'county': 'Frett',\n 'number': 39,\n 'state': 'Nevada',\n 'street': 'Ferry Ln.',\n
'zipcode': 30877} == {'county': 'Frett',\n 'number': 38,\n 'state': 'Nevada',\n 'street': 'Ferry Lane',\n 'zipcode':
30877}
E         Common items:
E         {'county': 'Frett', 'state': 'Nevada', 'zipcode': 30877}
E         Differing items:
E         {'number': 39} != {'number': 38}
E         {'street': 'Ferry Ln.'} != {'street': 'Ferry Lane'}
E         Full diff:
E           {
E            'county': 'Frett',
E         -  'number': 38,
E         ?            ^
E         +  'number': 39,
E         ?            ^
E            'state': 'Nevada',
E         -  'street': 'Ferry Lane',
E         ?                  - ^
E         +  'street': 'Ferry Ln.',
E         ?                    ^
E            'zipcode': 30877,
E           }
```

- Ao executar **pytest -vv**, o relatório aumenta a quantidade de detalhes e fornece uma comparação granular.

Testes com Pytest

- Métodos e classes de teste

Métodos e classes

- Além de escrever funções de teste, o Pytest permite que você use classes. Como já mencionado, não há necessidade de herança e as classes de teste seguem algumas regras simples. O uso de classes oferece mais flexibilidade e reutilização. Como vendo a seugir, o Pytest não atrapalha e evita obrigar você a escrever testes de uma maneira específica.
- Assim como funções, você também pode escrever declarações usando a instrução **assert**.

Métodos e classes – criar uma classe

- A função a seguir verifica se um determinado arquivo contém "sim" em seu conteúdo. Em caso afirmativo, ele retornará **True**. Se o arquivo não existir ou se ele contiver "não" em seu conteúdo, ele retornará **False**. Esse cenário é comum em tarefas assíncronas que usam o sistema de arquivos para indicar a conclusão.
- A aparência da função:

```
import os

def is_done(path):
    if not os.path.exists(path):
        return False
    with open(path) as _f:
        contents = _f.read()
    if "yes" in contents.lower():
        return True
    elif "no" in contents.lower():
        return False
```

Métodos e classes – criar uma classe

- Agora, veja como fica uma classe com dois testes (um para cada condição) em um arquivo chamado *test_files.py*:

```
class TestIsDone:

    def test_yes(self):
        with open("/tmp/test_file", "w") as _f:
            _f.write("yes")
        assert is_done("/tmp/test_file") is True

    def test_no(self):
        with open("/tmp/test_file", "w") as _f:
            _f.write("no")
        assert is_done("/tmp/test_file") is False
```

⊗ Cuidado

Os métodos de teste estão usando o caminho */tmp* para um arquivo de teste temporário porque é mais fácil de usar para o exemplo. No entanto, se você precisa usar arquivos temporários, considere usar uma biblioteca como `tempfile` que pode criá-los (e removê-los) com segurança para você. Nem todo sistema tem um diretório */tmp*, e esse local pode não ser temporário dependendo do sistema operacional.

Métodos e classes – criar uma classe

- Executar os testes com o sinalizador **-v** para aumentar o detalhamento mostra os testes aprovados:

```
pytest -v test_files.py
============================= test session starts ==============================
Python 3.9.6, pytest-6.2.5, py-1.11.0, pluggy-1.0.0
cachedir: .pytest_cache
rootdir: /private/
collected 2 items

test_files.py::TestIsDone::test_yes PASSED                               [ 50%]
test_files.py::TestIsDone::test_no PASSED                                [100%]

============================== 2 passed in 0.00s ===============================
```

- Embora os testes sejam aprovados, eles parecem repetitivos e também estão deixando os arquivos após a conclusão do teste. Antes de vermos como aprimorá-los, abordaremos os métodos auxiliares na próxima seção.

Métodos e classes – Métodos auxiliares

- Em uma classe de teste, você pode usar alguns métodos para configurar e desinstalar a execução do teste. O Pytest os executará automaticamente se forem definidos. Para usar esses métodos, você precisa saber que eles têm uma ordem e um comportamento específicos.
  - **setup**: é executado uma vez antes de cada teste em uma classe
  - **teardown**: é executado uma vez após cada teste em uma classe
  - **setup_class**: é executado uma vez antes de todos os testes em uma classe
  - **teardown_class**: é executado uma vez após todos os testes em uma classe
- Quando os testes exigem recursos semelhantes (ou idênticos) para funcionar, é útil escrever métodos de instalação. O ideal é que um teste não deixe recursos para trás após sua conclusão, portanto, métodos de desinstalação podem ajudar na limpeza do teste nessas situações.

Métodos e classes – Limpeza

- Veja uma classe de teste atualizada que limpa os arquivos após cada teste:

```
class TestIsDone:

    def teardown(self):
        if os.path.exists("/tmp/test_file"):
            os.remove("/tmp/test_file")

    def test_yes(self):
        with open("/tmp/test_file", "w") as _f:
            _f.write("yes")
        assert is_done("/tmp/test_file") is True

    def test_no(self):
        with open("/tmp/test_file", "w") as _f:
            _f.write("no")
        assert is_done("/tmp/test_file") is False
```

- Como usamos o método **teardown()**, essa classe de teste não deixa mais um /tmp/test_file para trás.

Testes com Pytest

- Exercício

Exercício: Form. Cadastro de Usuário

- Você está responsável por testar o sistema de cadastro de usuários de um site. O formulário de cadastro contém os seguintes campos:
  - 1. Nome: Deve ser um string até 30 caracteres
  - 2. Nome: Não deve haver caracteres especiais
  - 3. Idade: Deve ser um número inteiro entre 18 e 65 anos
- Tarefas:
  - Particionamento em Classes de Equivalência:
    - Identifique classes válidas e inválidas para cada campo do formulário
  - Análise do Valor Limite:
    - Defina os valores exatos para testar os limites de cada campo.
- Busque um código de Cadastro de Sistema de Usuário no github para realizar os testes.
- Utilize o framework Pytest para automatizar os testes.

Senac Pernambuco
Educação Profissional Recife

Thiago Dias Nogueira

Instrutor Técnico

(81) 9 9627-0419

thiago.nogueira@pe.senac.br