

FACULDADE SANTA TEREZINHA - CEST

SISTEMAS DE INFORMAÇÃO – 5º PERÍODO NOTURNO

QUALIDADE DE SOFTWARE

FERRAMENTA DE TESTES - PYTEST

THIAGO DIAS DA SILVA

SÃO LUÍS- MA

2022

Pytest

Pytest é uma estrutura de teste [Python](https://en.wikipedia.org/wiki/Python_(programming_language)) que se originou do projeto [PyPy](https://en.wikipedia.org/wiki/PyPy" \o "PyPy) . Ele pode ser usado para escrever vários tipos de testes de software, incluindo testes de [unidade, testes](https://en.wikipedia.org/wiki/Unit_tests) de [integração, testes](https://en.wikipedia.org/wiki/Integration_tests) de [ponta a ponta](https://en.wikipedia.org/wiki/System_testing) e [testes funcionais](https://en.wikipedia.org/wiki/Functional_tests).Seus recursos incluem testes [parametrizados ,](https://en.wikipedia.org/wiki/Parameter_(computer_programming))[fixtures](https://en.wikipedia.org/wiki/Test_fixture) e reescrita de [assert .](https://en.wikipedia.org/wiki/Assertion_(software_development)" \o "Asserção (desenvolvimento de software))

As fixtures do Pytest fornecem os contextos para testes passando nomes de parâmetros em casos de teste; sua parametrização elimina código duplicado para testar vários conjuntos de entrada e saída; [e suas declarações assert](https://en.wikipedia.org/wiki/Assertion_(software_development)) reescritas fornecem saída detalhada para causas de falhas. O Pytest foi desenvolvido como parte de um esforço de [pacotes de terceiros](https://en.wikipedia.org/wiki/Python_Package_Index) para resolver as deficiências do [módulo](https://en.wikipedia.org/wiki/Python_(programming_language)#Libraries) unittest do Python. Originou-se como parte do PyPy, uma implementação alternativa do Python ao [CPython](https://en.wikipedia.org/wiki/CPython" \o "CPython) padrão . Desde a sua criação no início de 2003, o PyPy teve uma forte ênfase em [testes](https://en.wikipedia.org/wiki/Software_testing) . O PyPy tinha testes de unidade para código recém-escrito, testes de regressão para bugs e testes de integração usando o conjunto de testes do CPython.

Em meados de 2004, uma estrutura de teste chamada utest surgiu e os colaboradores do PyPy começaram a converter [casos de teste](https://en.wikipedia.org/wiki/Test_case) existentes para utest. Enquanto isso, na EuroPython 2004, uma [biblioteca padrão](https://en.wikipedia.org/wiki/Standard_library) complementar para testes, chamada std, foi inventada. Este pacote estabeleceu os princípios, como a reescrita de assert, do que mais tarde se tornaria pytest. No final de 2004, o projeto std foi renomeado para py, std.utest tornou-se py.test e a [biblioteca](https://en.wikipedia.org/wiki/Library_(computing)) py foi separada de PyPy. Em novembro de 2010, o pytest 2.0.0 foi lançado como um pacote separado do py. Ele ainda era chamado de py.test até agosto de 2016, mas após o lançamento do pytest 3.0.0, o [ponto de entrada de](https://en.wikipedia.org/wiki/Entry_point)[linha de comando](https://en.wikipedia.org/wiki/Command-line_interface) recomendado tornou-se pytest.

O Pytest foi classificado pela plataforma de segurança do desenvolvedor Snyk como um dos principais projetos de ecossistema em Python devido à sua popularidade. Alguns projetos conhecidos que mudaram para pytest de unittest e nose (outro pacote de teste) incluem os de [Mozilla](https://en.wikipedia.org/wiki/Mozilla) e [Dropbox](https://en.wikipedia.org/wiki/Dropbox" \o "Dropbox) É um padrão comum em teste de software enviar valores por meio de [funções](https://en.wikipedia.org/wiki/Subroutine) de teste e verificar a saída correta. Em muitos casos, para [testar completamente as funcionalidades](https://en.wikipedia.org/wiki/Functional_testing), é necessário testar vários conjuntos de entrada/saída, e escrever esses casos separadamente causaria [código duplicado,](https://en.wikipedia.org/wiki/Duplicate_code) pois a maioria das ações permaneceria a mesma, diferindo apenas nos valores de entrada/saída. O recurso de teste parametrizado do Pytest elimina esse código duplicado combinando diferentes iterações em um caso de teste, depois executa essas iterações e exibe o resultado de cada teste separadamente.

Testes parametrizados em pytest são marcados pelo @pytest.mark.parametrize(argnames, argvalues) [decorador](https://en.wikipedia.org/wiki/Python_syntax_and_semantics#Decorators) , onde o primeiro [parâmetro](https://en.wikipedia.org/wiki/Parameter_(computer_programming)) , argnames, é uma string de nomes separados por vírgula e argvaluesé uma lista de valores a serem passados ​​para argnames. Quando houver vários nomes em argnames, argvaluesseria uma lista de tuplas em que os valores em cada tupla correspondem aos nomes argnamespor índice. Os nomes argnamessão então passados ​​para a função de teste marcada pelo decorador como parâmetros. Quando pytest executa esses testes decorados, cada par de argnamese argvaluesconstituiria uma execução separada com sua própria saída de teste e identificador exclusivo. O identificador pode então ser usado para executar pares de dados individuais,

Quando você executa o comando pytest dentro do seu ambiente virtual python, ele vai fazer um scan nos diretórios e subdiretórios do seu repositório procurando por arquivos que respeitem o formato de nomenclatura test\_\*.py ou \*\_test.py.Como o processo de descoberta é amplo e existe mais de uma forma de chamar o *pytest* para rodar (tanto comando direto pytest mencionado anteriormente, quanto com o comando python -m pytest), é altamente recomendado usar o padrão*\_\_init\_\_.py* em cada diretório para o *pytest* reconhecê-los como módulos e evitar colisão de nome nos arquivo de teste. Também se aconselha o uso dos padrões de organização de código-fonte caso seus testes estejam no mesmo repositório do código testado. A documentação oficial do *pytest* dá algumas sugestões de como organizar seus módulos de teste e módulos da aplicação:

setup.py  
src/  
 mypkg/  
 \_\_init\_\_.py  
 app.py  
 view.py  
tests/  
 \_\_init\_\_.py  
 foo/  
 \_\_init\_\_.py  
 test\_view.py  
 bar/  
 \_\_init\_\_.py  
 test\_view.py

Criando uma classe de testes dentro do arquivo “media.py”, o **pytest** procurará por qualquer classe iniciada com “Test” e executará todos os métodos dela que começarem com “test”.

class TestMedia:

def test\_media\_ok(self):

assert media\_aritmetica([1, 1]) == 1.0

O teste executará a função *media\_aritmetica()* para um conjunto de valores cujo resultado é comparado com o resultado esperado. A verificação será feita através do comando [assert](https://docs.python.org/3/reference/simple_stmts.html" \l "assert" \t "_blank) que verificará a expressão e provocará uma [exceção](https://giovannireisnunes.wordpress.com/2018/06/22/excecoes-em-python-parte-1/) de AssertionError caso o resultado não seja verdadeiro.

Anúncios

Daí é executar a rotina de testes.

$ pytest -v medias.py

**======================== test session starts =========================**

...

**collected 1 item**

medias.py::TestMedias::test\_media\_ok **PASSED** [100%]

====================== 1 passed in 0.01 seconds ======================

Mas a função tem uma falha, ela não verifica se a lista de números contém elementos antes de tentar efetuar o cálculo e o resultado é divisão por zero e consequentemente a interrupção do programa.

$ python ./programa.py

Traceback (most recent call last):

File "./programa.py", line 5, in

print(media\_aritmetica([]))

...

ZeroDivisionError: division by zero

A solução é verificar a quantidade de itens antes de efetivamente efetuar o cálculo.

def media\_aritmetica(valores):

quantidade = len(valores)

if quantidade:

return sum(valores) / quantidade

else:

return None

Daí é executar novamente a rotina de testes.

$ pytest -v medias.py

**======================== test session starts =========================**

...

collected 2 items

medias.py::TestMedias::test\_media\_com\_lista\_numerica **PASSED** [ 50%]

medias.py::TestMedias::test\_media\_com\_lista\_vazia **PASSED** [100%]

====================== 2 passed in 0.01 seconds ======================

Digamos que acidentalmente o retorno da função seja alterado de None para False. Em tese isto não afetaria o funcionamento do programa mas em **Python** o False pode ser interpretado como zero¹ criando um resultado errôneo.

O programa continuaria rodando normalmente, exceto que apresentando um comportamento diferente do esperado. Como no outro programa de exemplo, o “alunos.py”.

|  |  |
| --- | --- |
| **Retornando None:**  $ alunos.py  Adão Nogueira 8.258  Bruno Tavares sem notas  João da Silva 6.412  Jose Queiroz 8.511 | **Retornando False:**  $ alunos.py  Adão Nogueira 8.258  Bruno Tavares 0.000  João da Silva 6.412  Jose Queiroz 8.511 |

Porém, a rotina de testes detectaria esta mudança na função…

$ pytest -v medias.py

**======================== test session starts =========================**

...

collected 2 items

medias.py::TestMedias::test\_media\_com\_lista\_numerica **PASSED** [ 50%]

medias.py::TestMedias::test\_media\_com\_lista\_vazia **FAILED** [100%]

============================== FAILURES ==============================

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ TestMedias.test\_media\_com\_lista\_vazia \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

...

**def test\_media\_com\_lista\_vazia(self):**

**> assert media\_aritmetica([]) == False**

**E assert False == None**

**E + where None = media\_aritmetica([])**

medias.py:20: AssertionError

================= 1 failed, 1 passed in 0.03 seconds =================

E notificaria o erro imediatamente.