

# Documentation pytest

O **pytest** é uma lib super flexível, escalável, compatível com vários plugins, estável com suporte ao PyPy

## Executando o Pytest

O mínimo para para executar o pytest são que as funções de testes sejam nomeadas com o sufixo "test\_"

```
def test_any_function():
    pass
```

Dentro do pytest o que em outros lugares como o unittest de python onde temos vários tipos de assert, como `assertEqual`, `assertNotEqual`, isso e subi

```
def test_anu_function():
    assert True
```

Quando vamos escrever testes, por praxe utilizamos nomes grandes que para dar o entendimento do que aquela função faz

Ainda no Assunto de teste é importante entender que os testes é dividido em três partes, e que provem já de técnicas de BDD, e são:

- Given: Dado
- when: Quando
- then: então

ou seja, **dado** esse valor ... de entrada **quando** for realizar isso ... **então** é igual a ...

O que nos estamos olhando nos pontos marcados são os valores correspondente a esses três pontos, na frase a cima

Quando na pratica seria:

```
def squart(x):
    """
    função que recebe um valor numerico e
    eleva ele ao quadrado
    """
    return x**2

# minha condição de teste
# Given: dado 2 como entrada
# When: quando executar a minha função de
#   elevar um numero ao quadrado
# then: então o resultado deve ser 4

# teste
def test_squart_number():
    numero = 2 # given
    esperado = 4 # given

    result = squart(2) # when

    assert result == esperado # then
```

```
...
_
```

## Terminal do Pytest

Para rodar os nossos testes utlizaremos o terminal de testes com o comando `pytttest` mais onde esta os meus testes por exemplo:

```
pytest .\test
```

Aqui estou dizendo que tem para entrar na pasta de teste e testar o arquivos que tiverem nela, mas podemos coloca muitos comando que vão nos varia:

Flag	Description
-x	Esse comando permite ativar o que é conhecido com Fail Fast ou falha rapida, que segnifica que assim que houve um erro ele para os test, isso pode s demorar meia hora, fail fast nos permite agir mais proativamente em tempo menor
--exitfirst	mesma coisa que o -x
-v	Esse ele da mais detalhes do teste
--pdb	Abre o terminal interativo para debug no codigo de teste
-s	Se houver sai do console como um print o pytest não vai mostrar a menos que ele tenha essa saída
--junitxml onde/nome_arquivo.xml	Nos gera ao final do teste uma relatorio sobre os teste em XML
-rs	Ver a razão por que um teste foi pulado, caso haja
-k "parte do nome"	Cria um filtro para testar somente os teste que tenham parte do nome igual ao especificado
--fixtures	Permite ver as fixtures criadas

então poderíamos muito bem usar as juntas e de forma separada.

```
pytest .\test -rs -v -s --pdb
```

Algumas informações interessantes sobre as saídas, o pytest tem um padrão de resposta, de uma olhada na tabela abaixo:

Símbolo	Descrição
.	Passou
F	Falhou
x	Falha Esperada
X	Falha Esperada, que não ocorreu
s	Pulou - Skip

Agora olhando para uma saída de pytest:

```
(python-env) C:\Users\python-env>pytest .\tests --pdb -s
===== test session starts =====
platform win32 -- Python 3.10.2, pytest-7.2.0, pluggy-1.0.0
rootdir: C:\Users\thiag\python-env
plugins: anyio-3.6.2, Faker-15.3.3
collected 7 items

tests\test_calc.py ..s
tests\test_rename.py ....

===== 6 passed, 1 skipped in 0.30s =====
```

veja que antes do final da saída temos "tests\test\_calc.py" e na frente dele que no final da linha tem um "s" esse s é a mesma referência da tabela acima, na linha de baixo ele passou nos quatro teste, por isso quatro pontos

## conceito do One Step Test

Indo ainda no assunto de teste, a um conceito que a que queria empregar, no Livro TDD do Kent Beck, que é o conceito do **\*One Step Test\***, é o objetivo realizar apenas um assert por teste

Um exemplo é se tivéssemos uma função que aceita um valor entre 10 e 20, para teste seria recomendado testar um valor dentro do limite, ou seja então segundo esse conceito do **One Step Test**, então cada valor, teria o seu teste específico, por mais que fosse a mesma função sendo testada.

## mark - marcado, metadados e

Os **mark** é uma funcionalidade de marcação, e nos permite simplificar camadas ou rodar casos específicos para testes específicos, usando do estruturas de teste

No código a seguir vamos realizar algumas marcações, onde vamos marcar os testes que são "críticos" e que devem rodar durante a noite, para verificar nossas tags serão Critical e regressão, e a marcação com o mark acontece por decorator da seguinte forma:

```
@mark.nome_da_tag
```

Agora vamos ao código de exemplo:

```
from pytest import mark

@mark.critical
def test_function_return_two():
    """
    função explicativa que compara 2 igual a 2
    """
    assert 2 == 2

@mark.critical
@mark.regressao
def test_function_square_two():
    """
    função explicativa que eleva 2 ao quadrado e
    compara se é igual a 4
    """
    assert 2**2 == 4

@mark.regressao
def test_function_cube_two():
    """
    função explicativa que eleva 2 ao cubo e
    compara se é igual a 8
    """
```

```
"""
assert 2**3 == 8
```

Okay agora como vamos utilizar o mark para realizar test só na tag especifica. No terminal digite:

```
pytest .\ -m "critical"
```

Esse comando que digitamos, só deve testar, os testes com a tag: critical

Outra coisa que podemos realizar, e não realizar os testes com uma determinada tag. Vamos ver como seria se que quisesse não executar os testes com .

```
pytest .\ -m "not critical"
```

## mark do proprio pytest

mark	description
@mark.skip	Pula esse teste
@mark.skipif	Pula o teste em determinado contexto
@mark.xfail	Esperada falha no teste em certo contexto
@mark.parametrize	permite criar uma lista de parametros que devem ser passados a função permitindo realizar um teste vari

O primeiro que vamos ver e o skip é dentro do skip temos um parametro que é o reason, que identifica o motivo por que estamos pulando o teste

```
@mark.skip(reason= "não implementado ainda")
def test_create_factory():
    ...
```

E para enchargar o motivo por que esse teste vou executado temos que rodar no terminal o pytest com a flag rs de reason

```
pytest .\ -rs
```

## parametrize

O **parametrize**, nos permite passar valores como parametros para as funções de teste e executar ela com diferentes parametros.O parametrize recebe de lista com os valores que vão ser passados para esses parametros

Primeiro vamos implementar a função que vamos testar

```
def soma_2(number:int):
    """
    função de somar numero mais dois
    """
    return number+2

def squart(number:int):
    """
    Função que eleva o numero ao quadrado
    """
    return n**2
```

Agora os nossos testes:

```
@mark.parametrize(
    'entrada', [12, 45, 78, 23]
)
def test_soma_mais_dois(entrada):
    """
    Testando função que soma mais dois
    """

    result = soma_2(entrada)
    assert result == entrada + 2

@mark.parametrize(
    'entrada,esperado',
    [(11,121), (4,16), (22, 484)]
)
def test_soma_mais_dois(entrada, esperado):
    """
    Testando função que eleva a entrada ao quadrado
    """

    result = squart(entrada)
    assert result == esperado
```

Vaje que no segundo teste, para passar o nome dos dois parametros de função de teste, tivemos que passar eles separado por virgula dentro da mesma tuplas onde o primeiro numero de cada tupla e o parametro com o nome de entrada e o sugundo numero o parametro com o nome de esperado.

## xfail

O **xfail** é um tipo de parametro que nos ajuda a prepara par certas condições, ele diz que é esperado que o teste falhe, e essa tag não odeve se usada par para se preparar para certos contexto como um codigo que não roda na plataforma linux ou mac, mas roda no windows, então e esperado que se eu tiver **xfail**

```
import sys
from pytest import mark
from calc import squart

# não roda no windows
@mark.xfail(sys.plataform == 'win32')
def test_squart_two():
    assert squart(2) == 4
```

## skipif

O **skipif** serve para a mesma coisa que o xfail, se eu tiver um determinado contexto eu pulo aquele teste, se eu sei que uma coisa não roda no windows e

```
import sys
from pytest import mark
from calc import squart

# pulo se for no windows
# porque não roda
@mark.skipif(sys.plataform == 'win32')
def test_squart_two():
    assert squart(2) == 4
```

## fixtures

Antes de entrarmos no conceito de fixtures, primeiro vamos falar de outro conceito proposto no **livro XUnit Patterns de Gerard Mezarus**, onde ocorre tinhamos **3 etapas (Given, When , then)**, *é é adicionado mais um. Se formos pensar a testes que são mais complexos, por exempolo um API que escreve de banco de dados, é em alguns testes criar um banco de dados e só ir ingetando informações pode inviessar os teste e aqui, então necesse ponto nos temos o r dados, você vai criar um banco de dados, e na proximas vez que você for realizar o teste você vai criar, testar e deletar o banco de dados.*

Alguns nomes devido a esse conceito tambem mudam veja a seguir:

- Setup - Given | Dado
- Exercise - When | Quando
- Verify - Then | Então
- TearDown - Desmonta

## voltandos as fixtures

AS fixtures são forma de nos introduzimos coisas a esse contexto de teses, no exemplo da API é o banco de dados. E por padrão o pytest ja tem algumas

Uma das primieras que vamos ver é o **capsys** é um fixture responsavel por servir como espião, geralmente utilizamos, quando queremos ver o que ocorre olhar em uma função imprime no console de forma esperada, e para isso precisaremos moniorar no console a saida.

```
def numero_par(numero):
    """
    imprimir no console se o
    numero é impar o par
    """
    if par % 2 == 0:
        print('par')
    else:
        print('impar')
```

agora ao teste:

```
def test_numero_par_to_par(capsys):
    """
    dado o numero par a função testada
    deve retonar no terminal par, verifica
    se esse evento ocorre
    """

    numero_par(2)
    resultado = capsys.readout()
    assert resultado.out == "par\n"
```

veja que a fixture ela entra como uma parametro e é chamada como uma função dentro do testes, e a função do capsys.readout(), para ler o terminal e n python no print, então ele acaba sendo sempre adicionado no final)

## Criando suas fixtures

para criarmos uma fixtures vamos criar uma função com o decorator fixture como o seguinte

```
from pytest import fixture
```

```
@fixture
def return_3():
    return 3
```

```
def test_return_3(return_3):
    result = return_3
    assert result == 3
```

Poder criar as nossas próprias fixtures nos permite reutilizar códigos, nos nossos testes, evitando assim repetição. Entretanto vale dizer que é comum passar a dica e se usar um fixture quando o código começar a repetir

Outra coisa importante de dizer, referente ao código é a limpeza dele é que geralmente criamos um arquivo com o nome **conftest.py** e nele colocamos as fixtures do **conftest** como parte do escopo global e vamos poder utilizar elas livremente nos testes

## escopo da fixture

No exemplo o Fixture é responsável por tanto criar o dado como desmontá-lo depois da execução do teste.

Dentro do fixture podemos definir um escopo, como assim um escopo, por padrão o fixture tem o escopo de function, ou seja a cada função onde for função. E se por exemplo eu criar uma classe de teste e cada método de teste, eu crio um objeto com fixture e eu preciso que em todos os métodos seja o mesmo escopo para as fixtures e são eles:

- function - default | padrão
- class
- module
- package
- session - que atua em cima da suite de teste

E onde isso fica visível no meu código:

```
from pytest import fixture

# passo o escopo como parametro no meu fixture
@fixture(scope='function')
def return_3():
    return 3

def test_return_3(return_3):
    result = return_3
    assert result == 3
```

Eu acabo passando a partir do parametro scope o nome do escopo que ele vai agir nessa função

Mas uma coisa, para uma fixture chamar outra fixture

Sim, não é um problema uma fixture chamar outra e isso pode ser algo recorrente, que nos ajudará a não duplicar linhas de código então é totalmente possível

Classes a testar:

```
class Casa:

    def __init__(self, paredes = None):
        self.paredes = paredes or []

    def adiciona_parede(self, parede):
        self.paredes.append(parede)
        return True

    def get_paredes(self):
        return self.paredes

class Parede:
    def __init__(self, localizacao):
        self.localizacao = localizacao

    def get_localizacao(self):
        return self.localizacao
```

no **conftest.py**:

```
from pytest import fixture
from src.casa import Casa, Parede

@fixture
def criar_casa():
    return Casa()
```

```

@fixture
def adicionar_parede(criar_casa):
    casa = criar_casa
    casa.adicionar_parede(Parede('sudoeste'))
    return casa

```

No arquivo de test:

```

def test_criar_parede(adicionar_parede):
    casa = adicionar_parede

    total_de_paredes = len(casa.get_paredes())

    assert total_de_paredes == 1

```

...

### aproveitando o exemplo anterior e se eu quiser passar paramentros para um fixture

E totalmente possivel se adicionarmos um mark paratrise que vai passar os parametros de forma indireta para o fixture

```

class Casa():

    def __init__(self, paredes = None):
        self.paredes = paredes or []

    def adiciona_parede(self, parede):
        self.paredes.append(parede)
        return True

    def get_paredes(self):
        return self.paredes

class Parede():
    def __init__(self, localizacao):
        self.localizacao = localizacao

    def get_localizacao(self):
        return self.localizacao

```

Veja que adicionamos mais uma parametro a fixture "adicionar\_parede", que é request que vai receber o valor

```

from src.arquivs import Casa, Parede
from pytest import fixture, mark

@fixture
def criar_casa():
    return Casa()

@fixture
def adicionar_parede(criar_casa, request):
    casa = criar_casa
    local = request.param
    casa.adiciona_parede(Parede(local))
    return casa

```

E aqui utlizamos o parametrize para realizar a operação

```

@mark.parametrize(
    'adicionar_parede',
    ['sudoeste', 'noroeste', 'norte', 'oeste'],
    indirect=True
)
def test_criar_parede(adicionar_parede):
    casa = adicionar_parede
    total_de_paredes = len(casa.get_paredes())

    assert total_de_paredes == 1

```