Documentation pytest

O **pyteste** é uma lib super flexivel, escalavel, compativel com varios plugins, estavel com suporte ao PyPy

Executando o Pytest

O minimo para para executar o pyteste 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 varios tipos de assert, como assetIsEqualm AssertNotEqual, isso e subi

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

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

Ainda no Assunto de teste é importante entender que os testes é divido em três partes, e que provem ja de tecnicas de BDD, e são:

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

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

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

Ouando 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 utlzaremos o terminal de testes com o comando pyttest 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 varias

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 saida
--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 poderiamos muito bem usar as juntas e de forma separada.

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

Algumas infromações interransate sobre as saidas, o pytest tem um padrão de resposta, de uma olhada na tabela abaixo:

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

Agora olhando para uma saida 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 saida temos "tests\test_calc.py" e na frente dele que no final da linha tem um "s" esse s é a mesma referencia 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 tivessemos uma função que aceita um valores entre 10 e 20, para teste seria recomendado testar um valor dentro do limite, ou seja ent então segundo esse conceito do **One Step Test**, então cada valor, teria o seu teste especifico, por mais que fosse a mesma função sendo testada.

mark - marcado, medatados e

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

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

`@mark.nome_da_tag`
Agora vamos ao codigo de exemplo:

```
from pytest import mark


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

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


    @mark.regressao
    def test_funtion_cube_two():
        """
            funcão 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 permitidindo 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 dc lista com os valores que vã oser passados para esses parametros

Primeiro vamos implementar a funçã 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.