

# O arcabouço de testes - pytest

Jailma Januário da Silva

Leonardo Massayuki Takuno

## **Resumo**

*Os objetivos desta aula são: (I) Apresentar maneiras de realizar testes de unidade com o arcabouço de testes pytest. (II) Construir casos de testes que recebem os valores obtidos da função e confrontar com os valores esperados no teste.*

## **Introdução**

Atualmente, realizar testes de unidade tornou-se uma habilidade necessária para qualquer desenvolvedor de software. Este texto apresenta o Pytest (PYTEST, 2015), que é uma das ferramentas mais populares para realização de testes de unidades na linguagem Python. Você aprenderá como escrever testes de unidade para códigos Python com Pytest e como utilizá-lo para atender uma variedade de cenários de teste.

## **Pré-requisitos**

Para utilizar o pytest, execute a seguinte instrução:

**\$ pip install pytest**

A partir de agora, o pytest está disponível em seu ambiente de instalação.

O arcabouço Pytest tem a vantagem de ser livre e de código aberto (licença MIT), e tornou-se uma alternativa ao arcabouço unittest. Outra vantagem é que a sintaxe do Pytest é simples, facilitando a criação dos testes de unidade.

# Escrevendo um teste de unidade em Pytest

Para o primeiro exemplo utilizando Pytest, suponha que você tenha que construir um programa em Python para calcular as raízes de uma equação do segundo grau utilizando a fórmula de Bhaskara (KON, 2017). Uma equação do segundo grau é uma expressão matemática da forma:

$$ax^2 + bx + c$$

Os coeficientes dessa equação, representados pelas letras a, b e c, são números reais, com o valor de a diferente de zero.

Por exemplo:

i)  $x^2 + x - 9$  tem coeficientes  $a=1$ ,  $b=1$  e  $c=-9$

ii)  $x^2 - 5x + 6$  tem coeficientes  $a=1$ ,  $b=-5$  e  $c=6$

iii)  $7x^2 + 3x$  tem coeficientes  $a=7$ ,  $b=3$ ,  $c=0$

iv)  $2x^2$  tem coeficientes  $a=2$ ,  $b=0$ ,  $c=0$

v)  $3x + 5$  não é a equação do segundo grau.

A solução para a equação do segundo grau consiste em determinar os valores das suas raízes (valores de x). Para isso, utiliza-se a fórmula de Bhaskara:

$$x = \frac{-b \pm \sqrt{\Delta}}{2.a}$$

$$\Delta = b^2 - 4.a.c$$

O cálculo é dividido em duas partes:

- calcular o discriminante da fórmula de Bhaskara, e seu símbolo é a letra grega delta.

$$\Delta = b^2 - 4.a.c$$

- Se o valor de delta for maior que zero, a equação terá dois valores reais e distintos.
- Se o valor de delta for igual a zero, a equação terá somente um valor real ou dois valores iguais.
- Se o valor de delta for menor que zero, a equação não possui raízes reais.

Crie um arquivo de nome bhaskara.py de acordo com a Codificação 3.1.

#### Codificação 3.1. bhaskara.py

```
import math
```

```
def calcular_delta(a, b, c):
```

```
    return b * b - 4 * a * c
```

```
def calcular_raizes(a, b, c):
```

```
    delta = calcular_delta(a, b, c)
```

```
    if delta == 0:
```

```
        raiz = -b / (2 * a)
```

```

    return 1, raiz
else:
    if delta < 0:
        return 0
    else:
        raiz1 = (-b + math.sqrt(delta)) / (2 * a)
        raiz2 = (-b - math.sqrt(delta)) / (2 * a)

    return 2, raiz1, raiz2

```

Fonte: adaptado de KON, 2017.

Conforme exposto anteriormente, para calcular as raízes de uma equação do segundo grau deve-se levar em consideração as situações que dependem dos valores dos coeficientes da equação.

É importante, neste caso, criar um plano de testes para verificar se cada situação está contemplada pelo cálculo do programa. Observe como isto pode ser feito conforme ilustra a Tabela 3.1.

**Tabela 3.1. Planejando os cenários de testes**

calcular_raizes(a, b, c)		
#caso de teste	Parâmetros de entrada	Resultado esperado
CT0001	$x^2 = 0$ a=1, b=0, c=0	1 raiz real x = 0
CT0002	$x^2 - 5x + 6 = 0$ a=1, b=-5, c=6	2 raízes reais x1 = 3 x2 = 2
CT0003	$10x^2 + 10x + 10 = 0$ a=10, b=10, c=10	0 raízes reais
CT0004	$10x^2 + 20x + 10 = 0$ a=10, b=20, c=10	1 raiz real negativa x = -1

**Fonte: do autor, 2022.**

Para cada caso de teste é necessário criar uma função para ser executada pelo Pytest, conforme apresenta o arquivo test\_bhaskara.py como ilustra a Codificação 3.2.

**Codificação 3.2. test\_bhaskara.py**

```
from bhaskara import calcular_raizes

def test_bhaskara_uma_raiz():
    assert calcular_raizes(1, 0, 0) == (1, 0)

def test_bhaskara_duas_raizes():
    assert calcular_raizes(1, -5, 6) == (2, 3, 2)

def test_bhaskara_zero_raizes():
    assert calcular_raizes(10, 10, 10) == 0

def test_bhaskara_uma_raiz_negativa():

    assert calcular_raizes(10, 20, 10) == (1, -1)
```

**Fonte: adaptado de KON, 2017.**

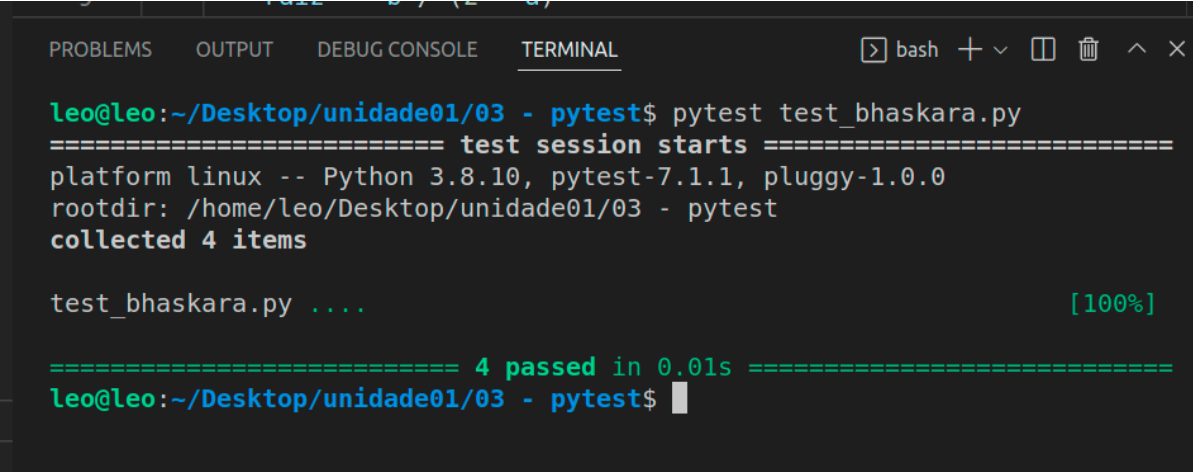
O Pytest utiliza utiliza uma convenção para nomes de arquivos e funções, como segue:

- Nomes de arquivos devem necessariamente conter a palavra test.
  - test\_\*.py
  - \*\_test.py
  - Exemplos: test\_bhaskara.py, test\_programa.py
- Nomes de funções de testes devem iniciar com a palavra test\_
  - Exemplos: test\_funcao\_soma(), test\_bhaskara\_uma\_raiz()
- Nomes de classes devem necessariamente começar por Test.

Estas convenções ajudam o Pytest a identificar as funções que tratam dos testes de unidades.

Para executar os testes de unidade, execute a instrução `pytest`, ou `pytest` seguido do nome do arquivo de teste, conforme apresenta a Figura 3.1.

**Figura 3.1. Executando as funções de teste**



```
leo@leo:~/Desktop/unidade01/03 - pytest$ pytest test_bhaskara.py
===== test session starts =====
platform linux -- Python 3.8.10, pytest-7.1.1, pluggy-1.0.0
rootdir: /home/leo/Desktop/unidade01/03 - pytest
collected 4 items

test_bhaskara.py .... [100%]

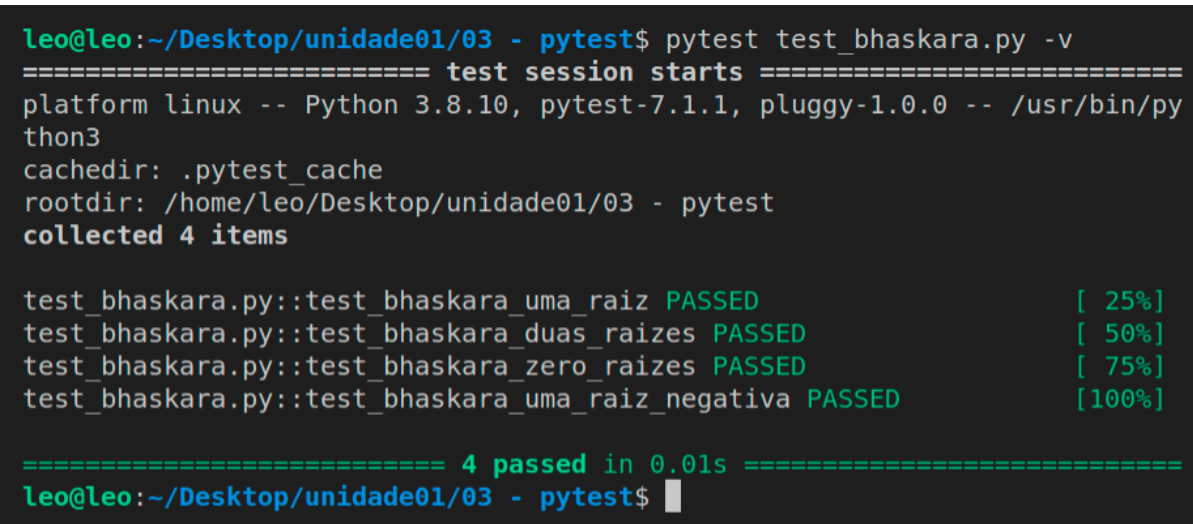
===== 4 passed in 0.01s =====
leo@leo:~/Desktop/unidade01/03 - pytest$
```

Fonte: do autor, 2022.

Perceba, que o Pytest executou 4 funções de teste que passaram com sucesso. Para cada função que executou com sucesso, o relatório do Pytest apresenta um ponto (.) na cor verde.

Para verificar os detalhes dos resultados para cada caso de testes, utilize a opção `-v` (*verbose*), conforme apresenta a Figura 3.2.

**Figura 3.2. Testes de unidade em detalhes**



```
leo@leo:~/Desktop/unidade01/03 - pytest$ pytest test_bhaskara.py -v
===== test session starts =====
platform linux -- Python 3.8.10, pytest-7.1.1, pluggy-1.0.0 -- /usr/bin/python3
cachedir: .pytest_cache
rootdir: /home/leo/Desktop/unidade01/03 - pytest
collected 4 items

test_bhaskara.py::test_bhaskara_uma_raiz PASSED [ 25%]
test_bhaskara.py::test_bhaskara_duas_raizes PASSED [ 50%]
test_bhaskara.py::test_bhaskara_zero_raizes PASSED [ 75%]
test_bhaskara.py::test_bhaskara_uma_raiz_negativa PASSED [100%]

===== 4 passed in 0.01s =====
leo@leo:~/Desktop/unidade01/03 - pytest$
```

Fonte: do autor, 2022.

## Tratando casos de exceção

Suponha, agora, que deseja-se tratar casos de exceção. Para isso, observe a Tabela 3.2 para novos cenários de testes.

**Tabela 3.2. Planejando os cenários de testes para tratar exceções**

#caso de teste	Parâmetros de entrada	Resultado esperado
CT0005	$a=0, b=0, c=0$	Não é equação do segundo grau
CT0006	$a='0', b=0, c=0$	O tipo de dados do valor a deve ser um número real
CT0007	$a=0, b='0', c=0$	O tipo de dados do valor b deve ser um número real
CT0008	$a=0, b=0, c='0'$	O tipo de dados do valor c deve ser um número real

Fonte: do autor, 2022.

Para tratar o caso de teste CT0005 da Tabela 3.2, inclua o seguinte teste de acordo com a Codificação 3.3.

**Codificação 3.3. test\_bhaskara.py com casos de exceção**

```
import pytest

from bhaskara import calcular_raizes

# ... continuação dos testes

def test_bhaskara_nao_eh_equacao_segundo_grau():

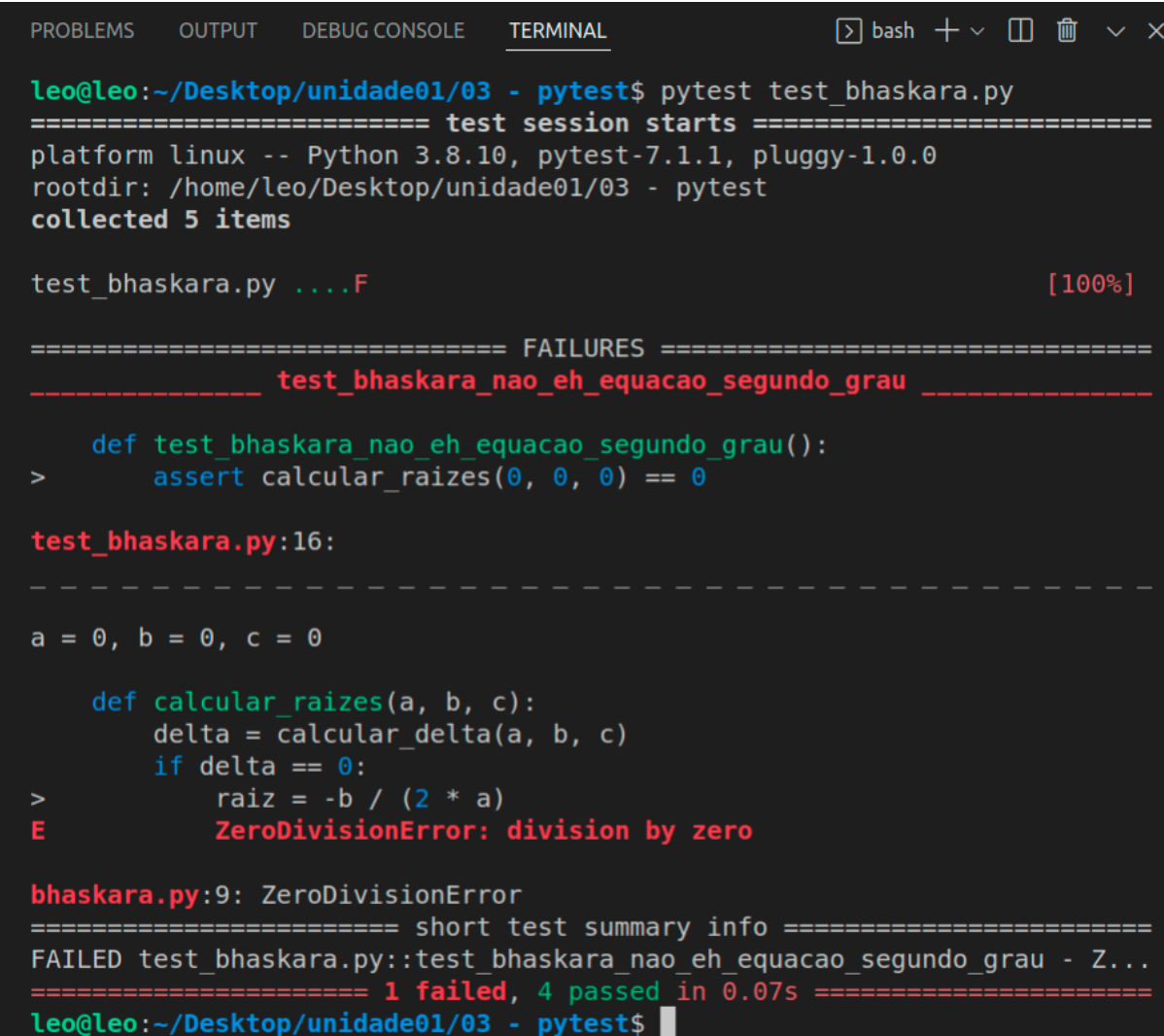
    assert calcular_raizes(0, 0, 0) == 0
```

Fonte: do autor, 2022.

Ao realizar a execução da função `test_bhaskara_nao_eh_equacao_segundo_grau()`, percebe-se que, pela Figura 3.3, houve uma falha, representada pela letra F (Failures), e também pela cor vermelha, além disso, o relatório do Pytest indica qual foi o arquivo em que

houve um teste com falhas, e ainda, o nome da função de teste que ocorreu a falha. Verifique a Figura 3.3.

**Figura 3.3. Testes com falhas**



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
leo@leo:~/Desktop/unidade01/03 - pytest$ pytest test_bhaskara.py
===== test session starts =====
platform linux -- Python 3.8.10, pytest-7.1.1, pluggy-1.0.0
rootdir: /home/leo/Desktop/unidade01/03 - pytest
collected 5 items

test_bhaskara.py ....F [100%]

===== FAILURES =====
_____ test_bhaskara_nao_eh_equacao_segundo_grau _____

    def test_bhaskara_nao_eh_equacao_segundo_grau():
>     assert calcular_raizes(0, 0, 0) == 0

test_bhaskara.py:16:
-----
a = 0, b = 0, c = 0

    def calcular_raizes(a, b, c):
        delta = calcular_delta(a, b, c)
        if delta == 0:
>         raiz = -b / (2 * a)
E         ZeroDivisionError: division by zero

bhaskara.py:9: ZeroDivisionError
===== short test summary info =====
FAILED test_bhaskara.py::test_bhaskara_nao_eh_equacao_segundo_grau - Z...
===== 1 failed, 4 passed in 0.07s =====
leo@leo:~/Desktop/unidade01/03 - pytest$
```

**Fonte: do autor, 2022.**

O relatório do Pytest indicou uma exceção de Divisão por zero, que no Python chama-se **ZeroDivisionError**, e a função que causou este problema é exatamente a função para calcular o discriminador, ou seja, a função **calcular\_delta()**.

É possível tratar a exceção, e aguardar que quando o valor do coeficiente  $a$  da equação seja 0, a função para calcular as raízes da equação sempre devolve a exceção **ZeroDivisionError**, a função de teste deve aguardar esta exceção



ser gerada conforme apresenta a Codificação 3.4 que utiliza a instrução `pytest.raises()` para aguardar que uma exceção seja lançada.

Codificação 3.4. `test_bhaskara.py` com tratamento de exceção

```
from bhaskara import calcular_raizes
```

```
# ... continuação dos testes
```

```
def test_bhaskara_nao_eh_equacao_segundo_grau():
```

```
    with pytest.raises(ZeroDivisionError):
```

```
        calcular_raizes(0, 0, 0)
```

Fonte: do autor, 2022.

Outra maneira, que também é adequada, seria criar uma exceção para indicar que os coeficientes fornecidos para a função `calcular_raizes()` não são coeficientes de uma equação de segundo grau. A Codificação 3.5 indica o lançamento de exceção para o caso de o coeficiente `a` seja um valor igual a zero.

Codificação 3.5. `bhaskara.py` com lançamento de exceção

```
import math
```

```
class ExcecaoNaoEhEquacaoSegundoGrau(Exception):
```

```
    pass
```

```
def calcular_delta(a, b, c):
```

```
    return b * b - 4 * a * c
```

```
def calcular_raizes(a, b, c):
```

```
    if a == 0:
```

```
        raise ExcecaoNaoEhEquacaoSegundoGrau()
```

```
    delta = calcular_delta(a, b, c)
```

```
    if delta == 0:
```

```
        raiz = -b / (2 * a)
```

```
        return 1, raiz
```

```

else:
    if delta < 0:
        return 0
    else:
        raiz1 = (-b + math.sqrt(delta)) / (2 * a)
        raiz2 = (-b - math.sqrt(delta)) / (2 * a)

    return 2, raiz1, raiz2

```

Fonte: adaptado de KON, 2017.

Nesta situação, a função **calcular\_raizes()** deixa de gerar a exceção de divisão por zero e passa a gerar a exceção chamada **ExcecaoNaoEhEquacaoSegundoGrau()**. Agora, a função para realizar o teste deve aguardar que essa exceção seja gerada, como se pode observar na Codificação 3.6.

Codificação 3.6. test\_bhaskara.py com tratamento de exceção proprietária

```

import pytest

from bhaskara import calcular_raizes

```

# ... continuação dos testes

```

def test_bhaskara_nao_eh_equacao_segundo_grau():
    with pytest.raises(ExcecaoNaoEhEquacaoSegundoGrau):
        calcular_raizes(0, 0, 0)

```

Fonte: do autor, 2022.

Para os casos de teste CT0006, CT0007, CT0008, da Tabela 3.2, basta verificar o tipo de dados e gerar a exceção de TypeError caso o tipo de dados dos coeficientes sejam diferentes do tipo real e diferente do tipo inteiro. Observe a Codificação 3.7 para tratar esses casos de testes.

Codificação 3.7. bhaskara.py com lançamento de exceção

```

import math

class ExcecaoNaoEhEquacaoSegundoGrau(Exception):
    pass

```

```

def calcular_delta(a, b, c):
    return b * b - 4 * a * c

def calcular_raizes(a, b, c):
    if type(a) != float and type(a) != int:
        raise TypeError()

    if type(b) != float and type(b) != int:
        raise TypeError()

    if type(c) != float and type(c) != int:
        raise TypeError()

    if a == 0:
        raise ExcecaoNaoEhEquacaoSegundoGrau()

    delta = calcular_delta(a, b, c)
    if delta == 0:
        raiz = -b / (2 * a)
        return 1, raiz
    else:
        if delta < 0:
            return 0
        else:
            raiz1 = (-b + math.sqrt(delta)) / (2 * a)
            raiz2 = (-b - math.sqrt(delta)) / (2 * a)

            return 2, raiz1, raiz2

```

Fonte: adaptado de KON, 2017.

Para realizar os testes para os casos de teste CT0006, CT0007, CT0008, da Tabela 3.2, inclua funções que aguardam uma exceção do tipo `TypeError` ser lançada, caso o tipo de dados seja diferente de inteiro ou float, como apresenta a Codificação 3.8.

Codificação 3.8. test\_bhaskara.py com tratamento de exceção

```
import pytest

from bhaskara import calcular_raizes

# ... continuação dos testes

def test_bhaskara_nao_eh_equacao_segundo_grau():
    with pytest.raises(ExcecaoNaoEhEquacaoSegundoGrau):
        calcular_raizes(0, 0, 0)

def test_bhaskara_tipo_coeficiente_a_invalido():
    with pytest.raises(TypeError):
        calcular_raizes('0', 0, 0)

def test_bhaskara_tipo_coeficiente_b_invalido():
    with pytest.raises(TypeError):
        calcular_raizes(0, '0', 0)

def test_bhaskara_tipo_coeficiente_c_invalido():
    with pytest.raises(TypeError):
        calcular_raizes(0, 0, '0')
```

Fonte: do autor, 2022.

Observe, pela Figura 3.4, a execução dos casos de testes para o tratamento da equação do segundo grau.

**Figura 3.4. Casos de testes**

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
leo@leo:~/Desktop/unidade01/03 - pytest$ pytest test_bhaskara.py -v
===== test session starts =====
platform linux -- Python 3.8.10, pytest-7.1.1, pluggy-1.0.0 -- /usr/bin/python3
cachedir: .pytest_cache
rootdir: /home/leo/Desktop/unidade01/03 - pytest
collected 8 items

test_bhaskara.py::test_bhaskara_uma_raiz PASSED [ 12%]
test_bhaskara.py::test_bhaskara_duas_raizes PASSED [ 25%]
test_bhaskara.py::test_bhaskara_zero_raizes PASSED [ 37%]
test_bhaskara.py::test_bhaskara_uma_raiz_negativa PASSED [ 50%]
test_bhaskara.py::test_bhaskara_nao_eh_equacao_segundo_grau PASSED [ 62%]
test_bhaskara.py::test_bhaskara_tipo_coeficiente_a_invalido PASSED [ 75%]
test_bhaskara.py::test_bhaskara_tipo_coeficiente_b_invalido PASSED [ 87%]
test_bhaskara.py::test_bhaskara_tipo_coeficiente_c_invalido PASSED [100%]

===== 8 passed in 0.01s =====
leo@leo:~/Desktop/unidade01/03 - pytest$
```

Fonte: do autor, 2022.

## O padrão Arrange-Act-Assert

Após estudar algumas maneiras de se realizar testes de unidades em Python, é possível perceber um padrão para se construir esses testes, que é conhecido como Arrange-Act-Assert. Esse padrão já havia sido mencionado na aula 01 desta unidade com os nomes configuração, chamada e afirmação, respectivamente. O padrão Arrange-Act-Assert consiste em três partes:

- **Arrange:** Nesta etapa, deve-se organizar a entrada de dados. Algumas vezes, para realizar tais testes, é necessário definir um cenário inicial, ou criar alguns objetos que serão utilizados nos testes, ou mesmo inserir alguns dados em um banco de dados. Isto causa uma dependência de elementos externos, que podem ser resolvidos por elementos que simulam o uso desses elementos externos, e são chamados de dublês (veremos profundamente estes termos em aulas posteriores).

- **Act:** Esta etapa refere-se à execução da função ou método no cenário do teste de unidade.
- **Assert:** Após a etapa do Act, a função sob teste devolve um valor que deve ser confrontado com um certo valor esperado. Caso estes dois valores sejam iguais, a função comporta-se adequadamente dado os valores de entrada. Caso contrário, a função não se comporta como esperado, e deve ser corrigida. Ao encontrar uma falha na função diz-se que o teste executou com sucesso, pois conseguiu identificar um problema na implementação.

A Codificação 3.9 apresenta explicitamente o código seguindo as etapas Arrange-Act-Assert.

#### **Codificação 3.9. Padrão Arrange-Act-Assert**

**bhaskara** **import** calcular\_raizes

**def** test\_bhaskara\_uma\_raiz():

**# arrange**

    a = 1

    b = 0

    c = 0

    valor\_esperado = (1, 0)

**# act**

    valor\_obtido = calcular\_raizes(a, b, c)

**# assert**

**assert** valor\_obtido == valor\_esperado

**def** test\_bhaskara\_duas\_raizes():

**# arrange**

    a = 1

    b = -5

```
c = 6
valor_esperado = (2, 3, 2)

# act
valor_obtido = calcular_raizes(a, b, c)

# assert
assert valor_obtido == valor_esperado
```

```
def test_bhaskara_zero_raizes():
    # arrange
    a = 10
    b = 10
    c = 10
    valor_esperado = 0

    # act
    valor_obtido = calcular_raizes(a, b, c)

    # assert
    assert valor_obtido == valor_esperado
```

Fonte: do autor, 2022.

## Considerações adicionais

Para escrever bons testes:

- Selecione casos em que o seu programa pode falhar.
- Escolha valores de entrada para atender várias situações.
- Em uma unidade mais complexa, escolha caminhos percorridos pelo seu programa.

## Vamos praticar?

1) De acordo com uma tabela médica, o peso ideal está relacionado com a altura e o sexo de uma pessoa. A tabela a seguir contém os cálculos para o peso ideal.

Sexo	Peso ideal
Masculino	$(72,7 * altura) - 58$
Feminino	$(62,1 * altura) - 44,7$

Importe o módulo `calculadora_de_peso.py` abaixo para um programa de teste e escreva testes unitários com o arcabouço Pytest para a função `obter_peso_ideal()` do módulo:

'''

**função: obter\_peso\_ideal()**

**entrada: altura, sexo**

**saída: O valor do peso ideal**

**Valores válidos:**

- altura (entre 1,0 m e 2,5 m)

- sexo (string 'M' ou 'F')

'''

**def obter\_peso\_ideal(altura, sexo):**

**if sexo == 'M':**

**return (72.7 \* altura) - 58**

**else:**

**return (62.1 \* altura) - 44.7**

Utilize os valores abaixo como parâmetros de entrada e saída, corrija a função caso seja necessária:



obter\_peso\_ideal(altura, sexo)

Entrada	Saída (valor aproximado)
<u>altura</u> = 1.5 <u>sexo</u> = 'M'	51.05
<u>altura</u> = 1.5 <u>sexo</u> = 'F'	48.45
<u>altura</u> = 1.6 <u>sexo</u> = 'M'	58.32
<u>altura</u> = 1.6 <u>sexo</u> = 'F'	54.66
<u>altura</u> = 1.7 <u>sexo</u> = 'M'	65.59
<u>altura</u> = 1.7 <u>sexo</u> = 'F'	60.86
<u>altura</u> = 1.8 <u>sexo</u> = 'M'	72.86
<u>altura</u> = 1.8 <u>sexo</u> = 'F'	67.08
<u>altura</u> = 1.9 <u>sexo</u> = 'M'	80.13
<u>altura</u> = 1.9 <u>sexo</u> = 'F'	73.28
<u>altura</u> = 2.0 <u>sexo</u> = 'M'	87.4
<u>altura</u> = 2.0 <u>sexo</u> = 'F'	79.5

Observação: Para testes de funções que devolvem número do tipo Float, deve-se levar em consideração a precisão de casas decimais do número. Para a linguagem Python um número, por exemplo, 51.05000000011 é diferente de 51.05. Para realizar este tipo de teste o pytest fornece uma função para realizar uma aproximação levando em consideração o número de casas decimais que se deseja comparar. Segue um exemplo de utilização do **pytest.approx()**, que realiza a aproximação do número para duas casas decimais (0.01).

```
import pytest
```

```
from calculadora_de_peso import obter_peso_ideal
```

```
def test_calcular_peso_homem_1_50_m():
```

```
    assert obter_peso_ideal(1.5, 'M') == pytest.approx(51.05, 0.01)
```

```
# escreva os demais testes para completar o exercício
```

2) Suponha que você foi contratado para desenvolver um programa que receba como entrada a idade de um nadador e classifica em uma das seguintes categorias.

Categoria	Idade
Infantil A	5 a 7 anos
Infantil B	8 a 10 anos
Juvenil A	11 a 13 anos
Juvenil B	14 a 17 anos
Sênior	<u>maiores de 18 anos.</u>

a) Escreva uma função chamada **obter\_categoria()** que receba como parâmetro a idade do atleta e devolve a categoria.

b) Construa o planejamento dos casos de testes determinando os valores de entrada, e valores de saída esperados.

c) Escreva os testes automatizados para cada caso de teste definido no item b.

3) Suponha que você foi contratado para realizar testes automatizados para um sistema médico. Uma funcionalidade do programa consiste em tomar como entrada a idade e o peso do paciente e calcular a dosagem de um determinado medicamento. Considere que o medicamento em questão possui 500 mg por ml, e que cada ml corresponde a 20 gotas.

- Adultos e adolescentes desde 12 anos, inclusive, se tiverem peso igual ou acima de 60 kg devem tomar 1000 mg; com peso abaixo de 60 kg devem tomar 875 mg.
- Para crianças e adolescentes abaixo de 12 anos a dosagem é calculada pelo peso corpóreo conforme a tabela a seguir:

<u>peso</u>	<u>dosagem</u>
5 kg a 9 kg	125 <u>mg</u>
9.1 kg a 16 kg	250 <u>mg</u>
16.1 kg a 24 kg	375 <u>mg</u>
24.1 kg a 30 kg	500 <u>mg</u>
<u>acima de 30 kg</u>	750 <u>mg</u>

Importe o módulo `escolar.py` abaixo para um programa de teste e escreva testes unitários com o arcabouço Pytest para a função **`calcular_dosagem()`** do módulo:

'''

**função: `calcular_dosagem`**

**entrada: idade e peso**

**saída: dosagem de um determinado medicamento**

**Valores válidos:**

- idade (entre 1 ano a 130 anos)

- peso (entre 5 kg a 200 kg)

'''

**`def calcular_dosagem(idade, peso):`**

**`if idade >= 12:`**

**`if peso >= 60:`**

**`return 1000`**

**`else:`**

**`return 875`**

**`else:`**

**`if peso >= 5 and peso <= 9:`**

```
    return 125
elif peso >= 9.1 and peso <= 16:
    return 250
elif peso >= 16.1 and peso <= 24:
    return 375
elif peso >= 24.1 and peso <= 30:
    return 500
elif peso >= 30:
    return 750
```

Utilize os valores abaixo como parâmetros de entrada e saída, corrija a função caso seja necessária:

<code>calcular_dosagem(idade, peso)</code>	
Entrada	Saída
<code>idade = -1</code> <code>peso = 5</code>	exceção: <code>ValueError</code>
<code>idade = 250</code> <code>peso = 5</code>	exceção: <code>ValueError</code>
<code>idade = 1</code> <code>peso = -1</code>	exceção: <code>ValueError</code>
<code>idade = 1</code> <code>peso = 250</code>	exceção: <code>ValueError</code>
<code>idade = 20</code> <code>peso = 60</code>	1000
<code>idade = 12</code> <code>peso = 60</code>	1000
<code>idade = 20</code> <code>peso = 59</code>	875
<code>idade = 12</code> <code>peso = 59</code>	875
<code>idade = 1</code> <code>peso = 5</code>	125
<code>idade = 1</code> <code>peso = 9</code>	125
<code>idade = 2</code> <code>peso = 9.1</code>	250
<code>idade = 2</code> <code>peso = 16</code>	250
<code>idade = 3</code> <code>peso = 16.1</code>	375
<code>idade = 6</code> <code>peso = 16.1</code>	675

# Referências

KON, F. **CCSL - IME/USP - Introdução à ciência da computação com python - vídeo 35.** YouTube, 2017. Disponível em: <[https://www.youtube.com/watch?v=RYpV10216pw&list=PLcoJJSvnDgcKpOi\\_UeneTNTIVOigRQwcn&index=42&ab\\_channel=CCSLdoIME%2FUSP](https://www.youtube.com/watch?v=RYpV10216pw&list=PLcoJJSvnDgcKpOi_UeneTNTIVOigRQwcn&index=42&ab_channel=CCSLdoIME%2FUSP)>. Acesso em: 17 jun. 2022.

PYTHON SOFTWARE FOUNDATION (org.). **Documentação python 3.10.4.** 2001. Disponível em: <<https://docs.python.org/pt-br/3/library/unittest.html>>. Acesso em: 01 jun. 2022.

PYTEST. **Documentação versão de python 3.7+.** 2015. Disponível em: <<https://docs.pytest.org/en/7.1.x/index.html>>. Acesso em: 11 jun. 2022.

SALE, D. **Testing python:** applying unit testing, TDD, BDD, and accepting testing. Wiley, 2014.