

# Fixtures e parametrização

Jailma Januário da Silva

Leonardo Massayuki Takuno

## **Resumo**

*Os objetivos desta parte são: (I) Entender o que são fixtures. (II) Como parametrizar testes no pytest. (III) Realizar exemplos de testes usando fixtures. (IV) Organizar pastas em projetos com testes. (V) Refatorar códigos de testes. (VI) Entender o papel do arquivo conftest.py.*

## **Introdução**

O pytest é um arcabouço de testes automatizado que possui várias características - dentre as quais pode-se citar a característica conhecida como *fixture*. As *fixtures* são funções que são executadas antes de uma função de teste ser executada. Com isso, é possível criar pequenos pedaços de software que podem ser utilizados em vários testes.

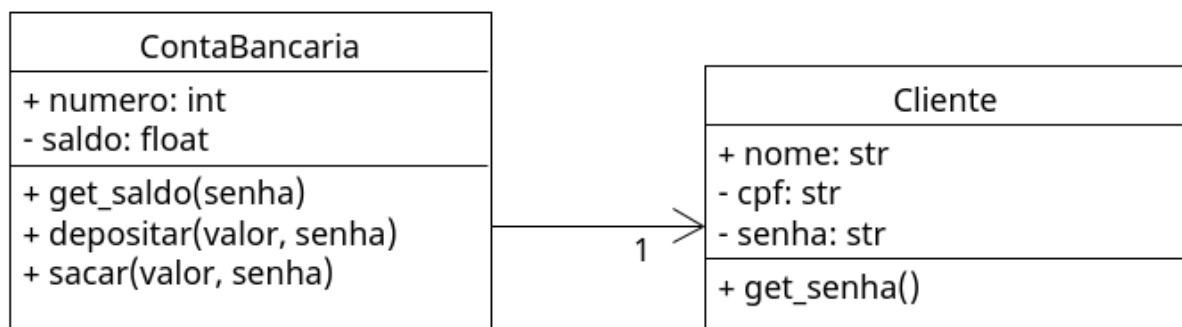
Em projetos, naturalmente é importante criar um cenário inicial para execução dos testes, ou seja, deseja-se preparar as pré-condições para a execução dos testes. Um exemplo de uso de *fixture* consiste em instanciar determinados objetos que serão utilizados no sistema sob teste (do inglês, *system under test* - SUT). Este texto apresenta exemplos de aplicação utilizando *fixture*.

## **Testes utilizando fixture**

Suponha, para este exemplo, um teste sobre as classes `ContaBancaria` e `Clientes`, conforme apresenta a Figura 5.1. A classe `Cliente` possui atributos como `nome` (público), `cpf` (privado), `senha` (privado) e, também, possui método público `get_senha()`. A classe `ContaBancaria` possui atributos como

numero (público), cliente (público), que é uma referência para a Classe Cliente, e o saldo (público), e, ainda, tem métodos como get\_saldo(), que recebe como parâmetro uma senha. Retorna o saldo da conta apenas se a senha for igual a senha do cliente. Outro método é o depositar(), que recebe como parâmetros de entrada um valor e uma senha e acrescenta esse valor ao saldo da conta apenas se a senha for igual à do cliente. E também, o método sacar(), que recebe como parâmetro de entrada um valor e uma senha e subtrai esse valor do saldo, apenas se a senha for igual à senha do cliente.

**Figura 5.1. Classes ContaBancaria e Cliente**



Fonte: do autor, 2022.

## A classe Cliente

Para o código da classe Cliente, observe o arquivo cliente.py de acordo com a Codificação 5.1. Neste código, acrescentou-se o método get\_cpf() para permitir o acesso do cpf.

**Codificação 5.1. cliente.py**

**class Cliente:**

```
def __init__(self, nome, cpf, senha):
```

```
    self.nome = nome
```

```
    self.__cpf = cpf
```

```
    self.__senha = senha
```

```
def get_cpf(self):
```

```
return self.__cpf
```

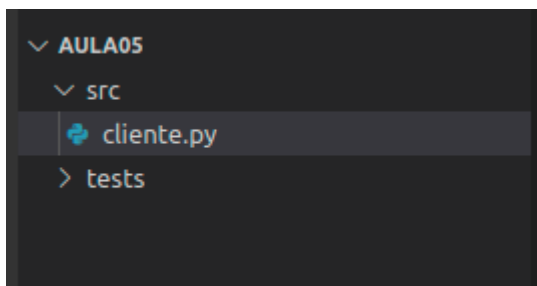
```
def get_senha(self):
```

```
return self.__senha
```

**Fonte: do autor, 2022.**

Para melhor organizar os arquivos, crie uma pasta chamada src e grave o arquivo cliente.py. Os arquivos para testes serão incluídos em uma pasta denominada tests como apresenta a Figura 5.2.

**Figura 5.2. Organização de pastas para projeto com testes**



**Fonte: do autor, 2022.**

Crie o arquivo test\_cliente.py de acordo com a Codificação 5.2 e grave o arquivo na pasta tests.

**Codificação 5.2. test\_cliente.py**

```
from src.cliente import Cliente
```

```
def test_criar_cliente_deve_devolver_nome_de_cliente_valido():
```

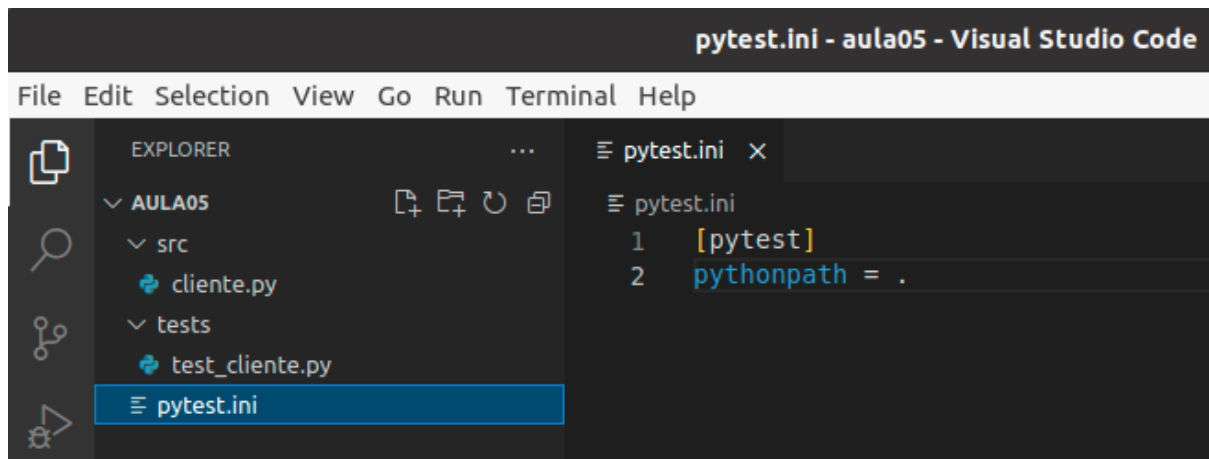
```
    cliente = Cliente('Jose da Silva', '123456789-00', '1234')
```

```
    assert cliente.nome == 'Jose da Silva'
```

**Fonte: do autor, 2022.**

Para o python trabalhar com pastas e conseguir localizar adequadamente os arquivos no projeto, será necessário criar um arquivo de configuração denominado pytest.ini, conforme a Figura 5.3.

**Figura 5.3. arquivo pytest.ini**



Fonte: do autor, 2022.

Para melhor visualização do caminho observe Figura 5.4.

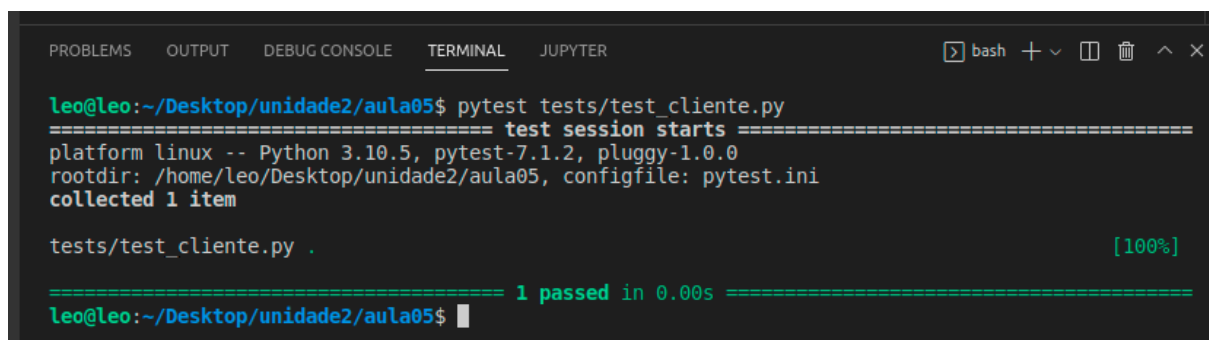
**Figura 5.4. Caminho dos arquivos do projeto**



Fonte: do autor, 2022.

O arquivo pytest.ini define uma variável chamada pythonpath que indica qual é o ponto inicial do pytest. Para executar o teste da Codificação 5.2 execute `pytest tests/test_cliente.py`, como apresenta a Figura 5.5.

**Figura 5.5. Executando o pytest**



Fonte: do autor, 2022.

Para evoluir com os testes, crie funções para testar o cpf e a senha, conforme a Codificação 5.3.

**Codificação 5.3. test\_cliente.py outros testes**

`from src.cliente import Cliente`

```
def test_criar_cliente_deve_devolver_nome_de_cliente_valido():
    cliente = Cliente('Jose da Silva', '123456789-00', '1234')
    assert cliente.nome == 'Jose da Silva'
```

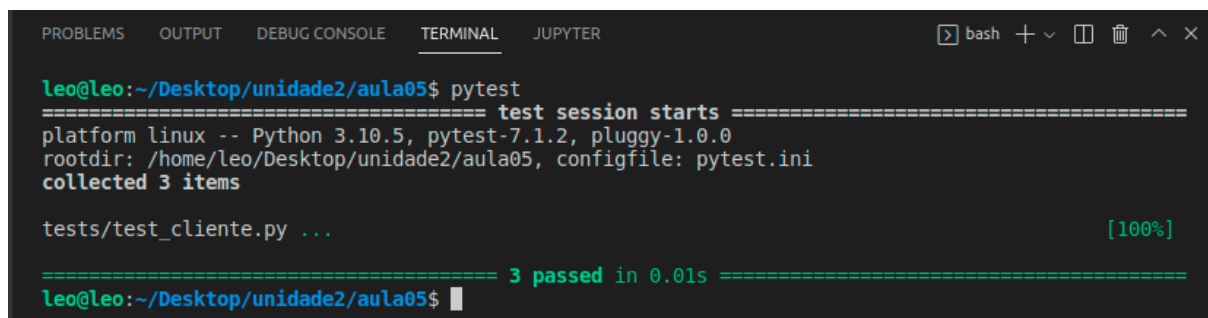
```
def test_criar_cliente_deve_devolver_cpf_do_cliente_valido():
    cliente = Cliente('Jose da Silva', '123456789-00', '1234')
    assert cliente.get_cpf() == '123456789-00'
```

```
def test_criar_cliente_deve_devolver_senha_do_cliente_valida():
    cliente = Cliente('Jose da Silva', '123456789-00', '1234')
    assert cliente.get_senha() == '1234'
```

Fonte: do autor, 2022.

Execute o pytest e verifique pela Figura 5.6 a execução dos testes sobre a instanciação de um objeto da classe Cliente.

Figura 5.6. Executando testes sobre a classe cliente



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL JUPYTER
Leo@Leo:~/Desktop/unidade2/aula05$ pytest
===== test session starts =====
platform linux -- Python 3.10.5, pytest-7.1.2, pluggy-1.0.0
rootdir: /home/leo/Desktop/unidade2/aula05, configfile: pytest.ini
collected 3 items

tests/test_cliente.py ... [100%]

===== 3 passed in 0.01s =====
Leo@Leo:~/Desktop/unidade2/aula05$
```

Fonte: do autor, 2022.

Os testes estão passando, agora pode-se fazer uma refatoração. Observe pela Codificação 5.3 que o código de instanciar o cliente ocorre nos três testes. Pois as validações estão sendo feitas pelas propriedades do objeto. A repetição de código indica um mau cheiro de código (do inglês, *bad smell*), e o remédio para isso, chama-se refatoração (do inglês, *refactoring*), que significa, organizar melhor o código sem modificar a funcionalidade. Neste ponto, criar uma *fixture* será útil para realizar a refatoração, conforme a Codificação 5.4.

#### Codificação 5.4. Refatorando o test\_cliente.py

```
import pytest
```

```
from src.cliente import Cliente
```

```
@pytest.fixture
```

```
def cliente():
```

```
    return Cliente('Jose da Silva', '123456789-00', '1234')
```

```
def test_criar_cliente_deve_devolver_nome_de_cliente_valido(cliente):
```

```
    assert cliente.nome == 'Jose da Silva'
```

```
def test_criar_cliente_deve_devolver_cpf_do_cliente_valido(cliente):
```

```
    assert cliente.get_cpf() == '123456789-00'
```

```
def test_criar_cliente_deve_devolver_senha_do_cliente_valida(cliente):
```

```
    assert cliente.get_senha() == '1234'
```

Fonte: do autor, 2022.

Para criar uma *fixture*, utiliza-se o decorator `@pytest.fixture`, que modifica a função para ser utilizada nos testes. Após isso, cada função de teste deve receber a *fixture* como parâmetro na função. Neste exemplo, pela Codificação 5.4 cada função de teste deve receber a *fixture* cliente.

## A classe ContaBancaria

Para o código da classe ContaBancaria, observe o arquivo conta\_bancaria.py de acordo com a Codificação 5.5 e grave o arquivo na pasta src do projeto.

Codificação 5.5. conta\_bancaria.py

```
class ExcecaoSenhaInvalida(Exception):
```

```
    pass
```

```
class ContaBancaria:
```

```
    def __init__(self, numero, cliente):
```

```
        self.numero = numero
```

```
        self.__saldo = 0
```

```

self.cliente = cliente

def get_saldo(self, senha):
    if self.cliente.get_senha() == senha:
        return self.__saldo
    else:
        raise ExcecaoSenhaInvalida

def depositar(self, valor, senha):
    if self.cliente.get_senha() == senha:
        self.__saldo += valor
    else:
        raise ExcecaoSenhaInvalida

def sacar(self, valor, senha):
    if self.cliente.get_senha() == senha:
        self.__saldo -= valor
    else:
        raise ExcecaoSenhaInvalida

```

**Fonte: do autor, 2022.**

De acordo com a Codificação 5.5, pode-se observar a classe ContaBancaria que utiliza-se de uma classe ExcecaoSenhaInvalida, a qual é lançada caso a senha que for passado como parâmetro para o método seja diferente da senha do cliente.

Agora, crie o arquivo test\_conta\_bancaria.py e grave na pasta tests. O arquivo test\_conta\_bancaria.py terá em seu conteúdo funções de testes para a classe ContaBancaria.

Escreva um teste denominado test\_criar\_conta\_bancaria\_devolve\_número\_correto, que instancia um objeto da classe ContaBancaria, e verifique se a propriedade número, que indica o

número da conta, está adequada de acordo com a criação. Observe a Codificação 5.6, que contém o código do arquivo `test_conta_bancaria.py`.

#### Codificação 5.6. `test_conta_bancaria.py`

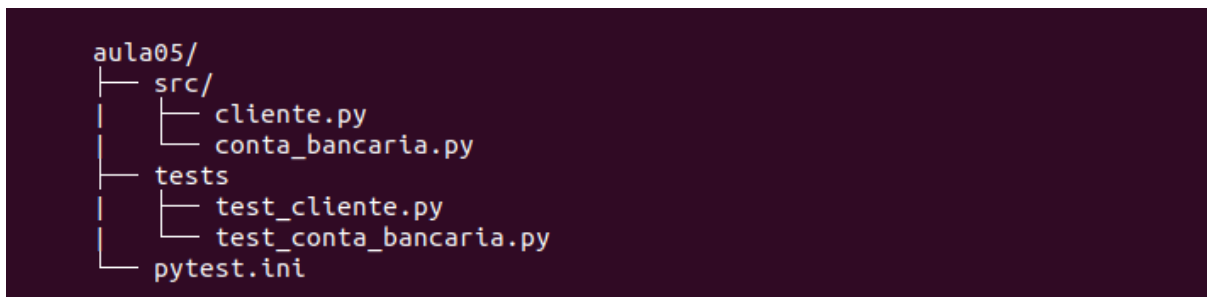
```
from src.conta_bancaria import ContaBancaria
from src.cliente import Cliente

def test_criar_conta_bancaria_devolve_numero_correto():
    cliente = Cliente('Jose da Silva', '123456789-00', '1234')
    conta = ContaBancaria(1, cliente)
    assert conta.numero == 1
```

Fonte: do autor, 2022.

Observe a Figura 5.7, a organização das pastas e arquivos após a criação do arquivo `conta_bancaria.py` e o arquivo `test_conta_bancaria.py`.

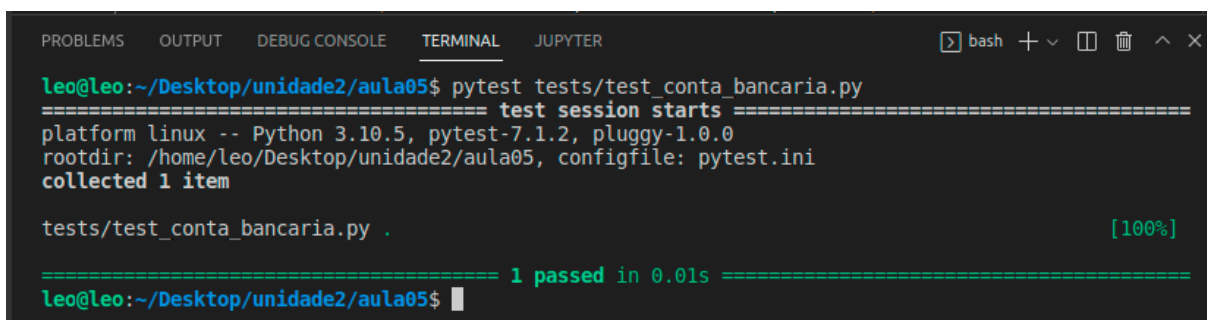
Figura 5.7. Organização dos arquivos



Fonte: do autor, 2022.

Execute o pytest sobre o arquivo `test_conta_bancaria.py`, verifique que o teste passou conforme apresenta a Figura 5.8.

Figura 5.8. Executando o teste sobre a classe `ContaBancaria`



Fonte: do autor, 2022.



Em seguida, crie alguns testes que verificam o saldo, como por exemplo, ao criar um objeto da conta bancária, o saldo deve ser igual a zero. Ao depositar um certo valor, o saldo deve ser alterado, ou seja, deve-se observar o saldo anterior e o saldo após o depósito. Ao sacar um certo valor, o saldo deve ser alterado, ou seja, deve-se observar o saldo anterior e o saldo após o saque. Para isso, observe a Codificação 5.7, e observe os testes criados.

#### Codificação 5.7. Testes sobre saldo, depósito e saque

```
from src.conta_bancaria import ContaBancaria
from src.cliente import Cliente

def test_criar_conta_bancaria_devolve_numero_correto():
    cliente = Cliente('Jose da Silva', '123456789-00', '1234')
    conta = ContaBancaria(1, cliente)
    assert conta.numero == 1

def test_criar_conta_bancaria_devolve_saldo_zerado():
    cliente = Cliente('Jose da Silva', '123456789-00', '1234')
    conta = ContaBancaria(1, cliente)
    assert conta.get_saldo(senha='1234') == 0

def test_depositar_valor_em_conta_devolve_saldo_aumentado():
    cliente = Cliente('Jose da Silva', '123456789-00', '1234')
    conta = ContaBancaria(1, cliente)
    assert conta.get_saldo(senha='1234') == 0
    conta.depositar(100, '1234')
    assert conta.get_saldo(senha='1234') == 100

def test_sacar_valor_em_conta_bancaria_devolve_saldo_menor():
    cliente = Cliente('Jose da Silva', '123456789-00', '1234')
    conta = ContaBancaria(1, cliente)
    conta.depositar(100, '1234')
    assert conta.get_saldo(senha='1234') == 100
    conta.sacar(20, '1234')
```

```
assert conta.get_saldo(senha='1234') == 80
```

Fonte: do autor, 2022.

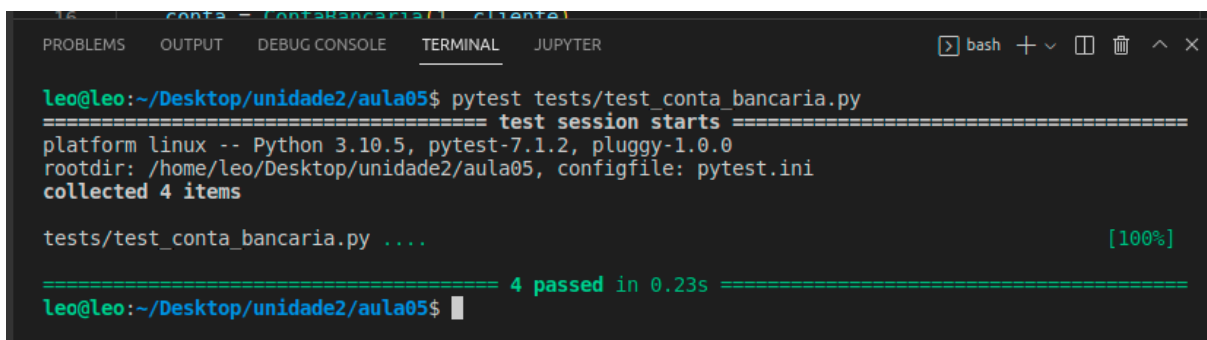
A função `test_criar_conta_bancaria_devolve_saldo_zerado()`, aloca um objeto do tipo `Cliente`, e após isso cria um objeto da `ContaBancaria` e verifica que o saldo é igual a zero.

A função `test_depositar_valor_em_conta_devolve_saldo_aumentado()` aloca novamente os objetos `Cliente` e `ContaBancaria`, após isso, o teste verifica que o saldo é igual a zero e, em seguida, invoca o método `depositar` passando os valores do valor do depósito e da senha do cliente. Por fim, o teste verifica novamente se o saldo foi atualizado com o valor do depósito.

A função `test_sacar_valor_em_conta_bancaria_devolve_saldo_menor()` aloca, outra vez, os objetos `Cliente` e `ContaBancaria`, e invoca o método `depositar` para o valor 100 e verifica se o saldo foi atualizado corretamente, em seguida, invoca o método `sacara` com o valor 20, e novamente, verifica que o saldo foi atualizado corretamente.

Execute o `pytest` sobre o arquivo `test_conta_bancaria.py` que contém os testes sobre saldo, depósito e saque e verifique pela Figura 5.9.

**Figura 5.9. Executando os testes sobre saldo, depósito e saque**



```
leo@leo:~/Desktop/unidade2/aula05$ pytest tests/test_conta_bancaria.py
===== test session starts =====
platform linux -- Python 3.10.5, pytest-7.1.2, pluggy-1.0.0
rootdir: /home/leo/Desktop/unidade2/aula05, configfile: pytest.ini
collected 4 items

tests/test_conta_bancaria.py .... [100%]

===== 4 passed in 0.23s =====
leo@leo:~/Desktop/unidade2/aula05$
```

Fonte: do autor, 2022.

Observe que pela Codificação 5.7 que há muito código repetido. Seguindo o princípio de arquitetura conhecido por DRY - *Don't Repeat Yourself*, que diz

para evitar redundância de código, então, neste caso, é possível incluir uma *fixture*, conforme a Codificação 5.8.

Codificação 5.8. Testes sobre saldo, depósito e saque com *fixture*

```
from pytest import fixture
```

```
from src.conta_bancaria import ContaBancaria
from src.cliente import Cliente
```

**@fixture**

```
def conta():
```

```
    cliente = Cliente('Jose da Silva', '123456789-00', '1234')
```

```
    conta = ContaBancaria(1, cliente)
```

```
    return conta
```

```
def test_criar_conta_bancaria_devolve_numero_correto(conta):
```

```
    assert conta.numero == 1
```

```
def test_criar_conta_bancaria_devolve_saldo_zerado(conta):
```

```
    assert conta.get_saldo(senha='1234') == 0
```

```
def test_depositar_valor_em_conta_devolve_saldo_aumentado(conta):
```

```
    assert conta.get_saldo(senha='1234') == 0
```

```
    conta.depositar(100, '1234')
```

```
    assert conta.get_saldo(senha='1234') == 100
```

```
def test_sacar_valor_em_conta_bancaria_devolve_saldo_menor(conta):
```

```
    conta.depositar(100, '1234')
```

```
    assert conta.get_saldo(senha='1234') == 100
```

```
    conta.sacar(20, '1234')
```

```
    assert conta.get_saldo(senha='1234') == 80
```

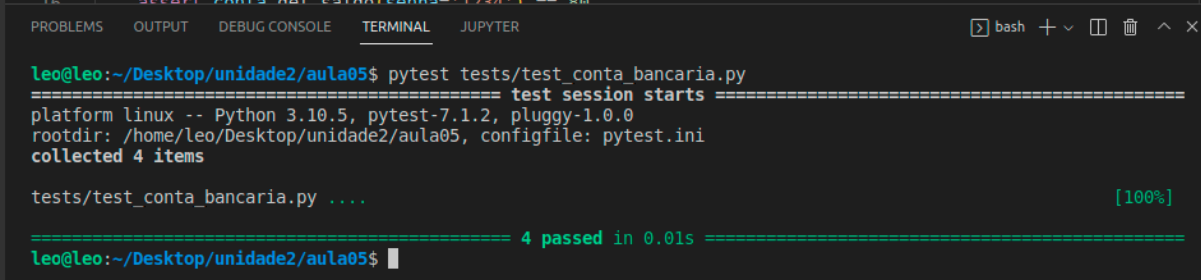
Fonte: do autor, 2022.

Nesta Codificação 5.8, criou-se uma *fixture* com o decorator **@fixture** sobre a função **conta()**, que cria um objeto da classe *Cliente* e associa a um objeto da

classe `ContaBancaria`, e devolve essa instância. Todas as funções de testes recebem como parâmetro a *fixture* `conta` e utiliza-o para realizar os testes sobre saldo, depósitos e saques.

Execute o `pytest` sobre o arquivo `test_conta_bancaria.py`, como apresenta a Figura 5.10, e verifique que os testes ainda continuam passando.

**Figura 5.10. Executando os testes após a refatoração**



```
leo@leo:~/Desktop/unidade2/aula05$ pytest tests/test_conta_bancaria.py
===== test session starts =====
platform linux -- Python 3.10.5, pytest-7.1.2, pluggy-1.0.0
rootdir: /home/leo/Desktop/unidade2/aula05, configfile: pytest.ini
collected 4 items

tests/test_conta_bancaria.py .... [100%]

===== 4 passed in 0.01s =====
leo@leo:~/Desktop/unidade2/aula05$
```

Fonte: do autor, 2022.

## O arquivo `conftest.py`

Como se pode perceber, até o momento, que para cada classe em um projeto, é possível ter um arquivo para testes, e que alguns objetos podem ser reutilizados em outros arquivos de testes. Neste exemplo, o objeto da classe `Cliente` foi reutilizado diversas vezes o arquivo `test_cliente.py`, e também diversas vezes no arquivo `test_conta_bancaria.py`. Uma maneira melhor de reorganizar as *fixtures* de maneira a aproveitar o reuso é através de um arquivo denominado `conftest.py`, que como o próprio nome sugere, é um arquivo de configuração dos testes.

Na pasta `tests`, crie o arquivo `conftest.py` de acordo com a Codificação 5.9, observe que agora, a *fixture* `cliente` é reutilizada na *fixture* `conta`.

**Codificação 5.9. `conftest.py`**

```
from pytest import fixture
```

```
from src.conta_bancaria import ContaBancaria
```

```
from src.cliente import Cliente
```

**@fixture**

**def cliente():**

**return** Cliente('Jose da Silva', '123456789-00', '1234')

**@fixture**

**def conta(cliente):**

**conta** = ContaBancaria(1, cliente)

**return** conta

**Fonte: do autor, 2022.**

Agora, o arquivo test\_cliente.py deve ser modificado conforme a Codificação 5.10, o parâmetro cliente será criado por causa da *fixture* cliente que foi definida no arquivo conftest.py.

**Codificação 5.10. Refatorando o test\_cliente.py**

**def test\_criar\_cliente\_deve\_devolver\_nome\_de\_cliente\_valido(cliente):**

**assert** cliente.nome == 'Jose da Silva'

**def test\_criar\_cliente\_deve\_devolver\_cpf\_do\_cliente\_valido(cliente):**

**assert** cliente.get\_cpf() == '123456789-00'

**def test\_criar\_cliente\_deve\_devolver\_senha\_do\_cliente\_valida(cliente):**

**assert** cliente.get\_senha() == '1234'

**Fonte: do autor, 2022.**

Em seguida, altere o arquivo test\_conta\_bancaria.py conforme a Codificação 5.11, o parâmetro conta será criado por causa da *fixture* conta que foi definida no arquivo conftest.py.

**Codificação 5.11. Refatorando o test\_conta\_bancaria.py**

**def test\_criar\_conta\_bancaria\_devolve\_numero\_correto(conta):**

**assert** conta.numero == 1

**def test\_criar\_conta\_bancaria\_devolve\_saldo\_zerado(conta):**

**assert** conta.get\_saldo(senha='1234') == 0

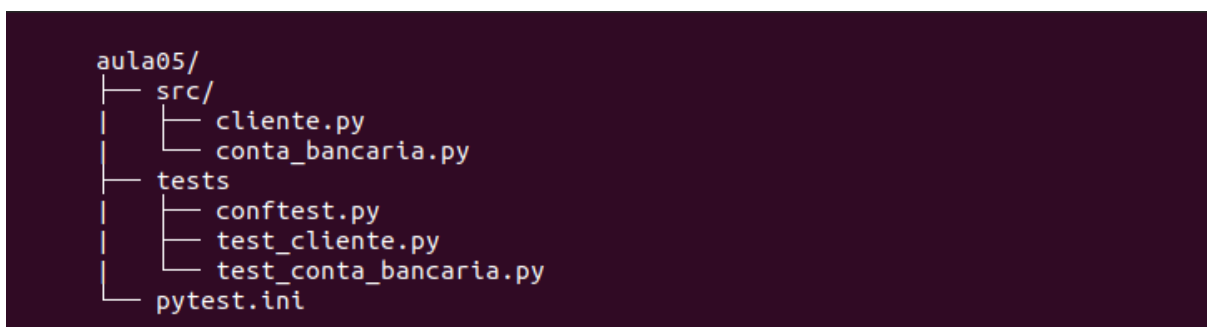
```
def test_depositar_valor_em_conta_devolve_saldo_aumentado(conta):
    assert conta.get_saldo(senha='1234') == 0
    conta.depositar(100, '1234')
    assert conta.get_saldo(senha='1234') == 100
```

```
def test_sacar_valor_em_conta_bancaria_devolve_saldo_menor(conta):
    conta.depositar(100, '1234')
    assert conta.get_saldo(senha='1234') == 100
    conta.sacar(20, '1234')
    assert conta.get_saldo(senha='1234') == 80
```

Fonte: do autor, 2022.

Observe a Figura 5.11, que apresenta a árvore de arquivos do projeto com o arquivo conftest.py.

Figura 5.11. Arquivos do projeto



Fonte: do autor, 2022.

Execute o pytest, e verifique que os testes continuam passando, conforme apresenta a Figura 5.12. Note que ao executar apenas a instrução pytest, sem especificar o nome do arquivo, o pytest realizará todos os testes.

Figura 5.12. Executando o pytest para todos os arquivos de testes

```

leo@leo:~/Desktop/unidade2/aula05$ pytest
===== test session starts =====
platform linux -- Python 3.10.5, pytest-7.1.2, pluggy-1.0.0
rootdir: /home/leo/Desktop/unidade2/aula05, configfile: pytest.ini
collected 7 items

tests/test_cliente.py ... [ 42%]
tests/test_conta_bancaria.py .... [100%]

===== 7 passed in 0.02s =====
leo@leo:~/Desktop/unidade2/aula05$
  
```

Fonte: do autor, 2022.

# Combinando *fixtures* com testes parametrizados

O Pytest fornece uma forma de execução de um conjunto de entradas para uma função de teste denominado teste parametrizado. Para utilizar o teste parametrizado, define-se quais são os parâmetros de entrada de uma função de teste, em seguida, define-se um conjunto de valores que serão fornecidos para a função. Este recurso é bastante útil para tornar o código de testes menos repetitivo. Além disso, é possível combinar *fixtures* com testes parametrizados, isso pode ser observado na Codificação 5.12.

## Codificação 5.12. *Fixtures* e código parametrizado

```
import pytest
```

```
def test_criar_conta_bancaria_devolve_numero_correto(conta):  
    assert conta.numero == 1
```

```
def test_criar_conta_bancaria_devolve_saldo_zerado(conta):  
    assert conta.get_saldo(senha='1234') == 0
```

```
def test_depositar_valor_em_conta_devolve_saldo_aumentado(conta):  
    assert conta.get_saldo(senha='1234') == 0  
    conta.depositar(100, '1234')  
    assert conta.get_saldo(senha='1234') == 100
```

```
def test_sacar_valor_em_conta_devolve_saldo_menor(conta):  
    conta.depositar(100, '1234')  
    assert conta.get_saldo(senha='1234') == 100  
    conta.sacar(20, '1234')  
    assert conta.get_saldo(senha='1234') == 80
```

```

@pytest.mark.parametrize("valor_deposito,          valor_saque,
valor_esperado",[
    (30, 10, 20),
    (20, 2, 18),
    (50, 25, 25),
    (15, 15, 0)
])
def test_transacao(conta, valor_deposito, valor_saque, valor_esperado):
    conta.depositar(valor_deposito, '1234')
    conta.sacar(valor_saque, '1234')
    conta.get_saldo(senha='1234') == valor_esperado

```

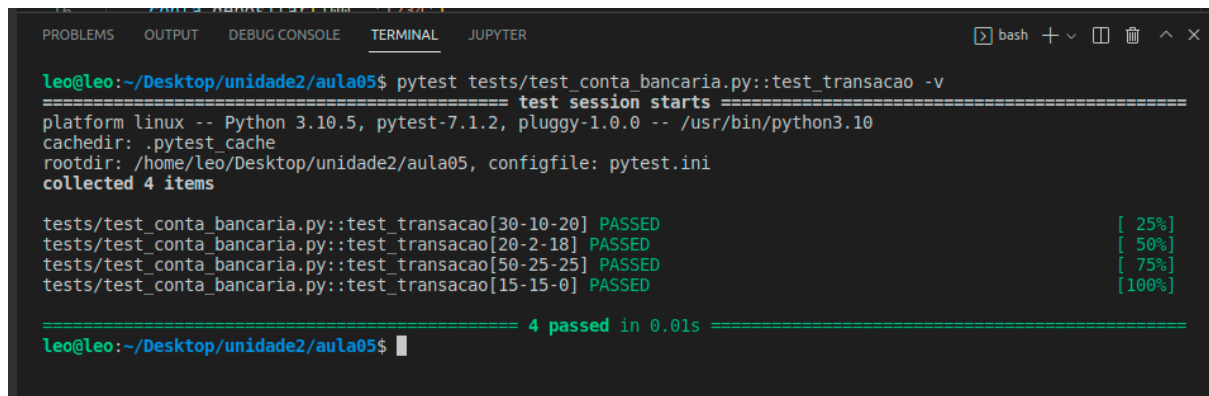
Fonte: do autor, 2022.

Pela Codificação 5.12, é possível observar o decorator **@pytest.mark.parametrize**, cujo primeiro parâmetro é uma string que indica quais serão os parâmetros da função de teste. O segundo parâmetro é uma lista de tuplas, com três valores em cada tupla, que indicam os valores que serão passados como parâmetros para a função **test\_transacao()**. Além da parametrização, o primeiro parâmetro da função **test\_transacao()** é a fixture **conta** que foi criada no arquivo **confest.py**.

Para executar apenas a função **test\_transacao**, e com detalhes, execute **pytest tests/test\_conta\_bancaria.py:: test\_transacao -v**, conforme apresenta a Figura 5.13.

**Figura 5.13. Executando apenas o test\_transacao do arquivo test\_conta\_bancaria.py**





```
leo@leo:~/Desktop/unidade2/aula05$ pytest tests/test_conta_bancaria.py::test_transacao -v
===== test session starts =====
platform linux -- Python 3.10.5, pytest-7.1.2, pluggy-1.0.0 -- /usr/bin/python3.10
cachedir: .pytest_cache
rootdir: /home/leo/Desktop/unidade2/aula05, configfile: pytest.ini
collected 4 items

tests/test_conta_bancaria.py::test_transacao[30-10-20] PASSED [ 25%]
tests/test_conta_bancaria.py::test_transacao[20-2-18] PASSED [ 50%]
tests/test_conta_bancaria.py::test_transacao[50-25-25] PASSED [ 75%]
tests/test_conta_bancaria.py::test_transacao[15-15-0] PASSED [100%]

===== 4 passed in 0.01s =====
leo@leo:~/Desktop/unidade2/aula05$
```

Fonte: do autor, 2022.

Observe pela Figura 5.13 que houve quatro execuções da função **test\_transacao()** com parâmetros definidos como entrada para a função.

Ainda faltam testes a serem realizados, é preciso tratar as exceções. Dependendo das regras de negócio, o método sacar pode gerar uma exceção caso o valor de saque seja maior que o saldo existente. Como exercício, elabore os demais casos de testes, incluindo os casos de teste para exceção.

## Considerações finais

Sempre que for possível refatore, organize o código, deixe o mais claro possível para que outros desenvolvedores possam se utilizar do código de teste.

Pense que o código de teste serve para ajudar a entender o código criado, entender as possíveis entradas e saídas das funções e métodos.

Os benefícios de um código de teste bem escrito são:

- Legibilidade nas regras de negócio.
- Auxílio para novos desenvolvedores que entrarem no projeto.
- Auxílio nas correções de bugs do sistema.

## Vamos praticar?

1) Seja o código da classe Bhaskara abaixo para calcular as raízes da equação do segundo grau.

```
import math
```

```
class Bhaskara:
```

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

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

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

```
        delta = self.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
```

arquivo: bhaskara.py

Seja o código de testes sobre a classe Bhaskara, a seguir:

```
from bhaskara import Bhaskara
```

```
def test_bhaskara_uma_raiz():
```

```
    bhaskara = Bhaskara()
```

```
    assert bhaskara.calcular_raizes(1, 0, 0) == (1, 0)
```

```
def test_bhaskara_duas_raizes():
```

```
    bhaskara = Bhaskara()
```

```
    assert bhaskara.calcular_raizes(1, -5, 6) == (2, 3, 2)
```

```
def test_bhaskara_zero_raizes():
```

```
bhaskara = Bhaskara()
assert bhaskara.calcular_raizes(10, 10, 10) == 0
```

```
def test_bhaskara_uma_raiz_negativa():
    bhaskara = Bhaskara()
    assert bhaskara.calcular_raizes(10, 20, 10) == (1, -1)
arquivo: bhaskara.py
```

Pede-se:

- a) Refatore o código utilizando *fixture*.
- b) Modifique os testes para utilizar testes parametrizados.

2) Seja o código da função fizzbuzz a seguir:

```
def fizzbuzz(n):
    if n % 3 == 0 and n % 5 == 0:
        return "FizzBuzz"
    if n % 5 == 0:
        return "Buzz"
    if n % 3 == 0:
        return "Fizz"

    return str(n)
```

Dado o plano de testes considerando alguns casos de testes:

<b>fizzbuzz(n)</b>		
#caso de teste	Parâmetros de entrada	Resultado esperado
CT0001	n=1	1
CT0002	n=2	2
CT0003	n=3	Fizz
CT0004	n=4	4
CT0005	n=5	Buzz
CT0006	n=6	Fizz
CT0007	n=7	7
CT0008	n=8	8
CT0009	n=9	Fizz
CT0010	n=10	Buzz
CT0011	n=11	11
CT0012	n=12	Fizz
CT0013	n=13	13
CT0014	n=14	14
CT0015	n=15	FizzBuzz
CT0016	n=16	16
CT0017	n=17	17
CT0018	n=18	Fizz
CT0019	n=19	19
CT0020	n=20	Buzz

Pede-se:

- Escreva uma função de teste parametrizado para cada caso de teste.

3) Seja o código da função cubo a seguir:

**def** cubo(x):

**return** x \* x \* x

Dado o plano de teste com alguns casos de teste a seguir.

cubo(x)		
#caso de teste	Parâmetros de entrada	Resultado esperado
CT0001	x=0	0
CT0002	x=1	1
CT0003	x=2	8
CT0004	x=-2	-8
CT0005	x=10	1000

Pede-se:

- Escreva uma função de teste parametrizado para cada caso de teste.

## Referências

ABOUT FIXTURES. **Documentação pytest versão de python 3.7+** - About fixture. 2015. Disponível em: <<https://docs.pytest.org/en/7.1.x/explanation/fixtures.html#about-fixtures>>. Acesso: 27 jun. 2022.

FIXTURES REFERENCE. **Documentação pytest versão de python 3.7+** - Fixture reference. 2015. Disponível em: <<https://docs.pytest.org/en/7.1.x/reference/fixtures.html#fixture>>. Acesso: 27 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.

