Neste exercício, você criará um sistema simples de gerenciamento de contas bancárias usando classes e escreverá testes para garantir a funcionalidade das classes.

Sistema de Gerenciamento de Contas Bancárias:

1. Crie uma classe `ContaBancaria` com os seguintes atributos e métodos:
   - Atributos: `numero_conta`, `saldo`
   - Métodos:
     - `__init__(self, numero_conta, saldo_inicial)`: Inicializa a conta com o número de conta e saldo inicial fornecidos.
     - `depositar(self, valor)`: Deposita o valor especificado na conta.
     - `sacar(self, valor)`: Saca o valor especificado da conta.
     - `obter_saldo(self)`: Retorna o saldo atual da conta.
2. Crie uma classe `ContaPoupanca` que herda de `ContaBancaria`. Adicione os seguintes métodos:

- `__init__(self, numero_conta, saldo_inicial, taxa_juros)`: Inicializa a conta poupança com o número de conta, saldo inicial e taxa de juros fornecidos.
- `calcular_juros(self)`: Calcula e adiciona juros à conta com base na taxa de juros.

Testes com pytest:

Escreva testes pytest para garantir o comportamento correto das classes `ContaBancaria` e `ContaPoupanca`. Crie um arquivo de teste separado, por exemplo, `test_contas.py`, e escreva casos de teste para os seguintes cenários:

1. Teste a classe `ContaBancaria`:
   - Teste a inicialização da conta com um número de conta e saldo inicial.
   - Teste o depósito e saque de valores na conta.
   - Teste a obtenção do saldo atual.
2. Teste a classe `ContaPoupanca`:
   - Teste a inicialização da conta poupança com um número de conta, saldo inicial e taxa de juros.
   - Teste o cálculo e adição de juros à conta poupança.

Lembre-se de estruturar seus testes usando o padrão Arrange-Act-Assert. Para cada caso de teste, crie instâncias das classes, execute ações e, em seguida, verifique os resultados esperados.

Exemplo de caso de teste pytest:

```
from contas_bancarias import ContaBancaria,
ContaPoupanca

def test_conta_bancaria():
    conta = ContaBancaria("123456", 1000)

    conta.depositar(500)
    assert conta.obter_saldo() == 1500

    conta.sacar(200)
    assert conta.obter_saldo() == 1300

def test_conta_poupanca():
    conta_poupanca = ContaPoupanca("789012",
2000, 0.05)

    conta_poupanca.calcular_juros()
    assert conta_poupanca.obter_saldo() ==
2100.0

if __name__ == "__main__":
    pytest.main()
```

Neste exercício, você praticará o design de classes, a implementação de herança e a escrita de testes pytest para garantir a correção do seu código. Certifique-se de criar as definições de classe necessárias (`ContaBancaria` e `ContaPoupanca`) em módulos separados antes de executar os testes.

conta_bancaria.py

```
class ContaBancaria:
    def __init__(self, numero_conta:str,
saldo:float):
        self._numero_conta = numero_conta

        if saldo < 0:
            raise ValueError("Não é possível
iniciar uma conta com saldo negativo!")
        self._saldo = saldo

    def obter_saldo(self):
        return self._saldo

    def depositar(self, valor:float):
        if valor <= 0:
            raise ValueError("Não é possível
depositar um valor menor ou igual a zero!")
        self._saldo += valor

    def sacar(self, valor:float):
        if valor <= 0:
            raise ValueError("Não é possível
sacar um valor menor ou igual a zero!")
        self._saldo -= valor

class ContaPoupanca(ContaBancaria):
```

```
    def __init__(self, numero_conta: str,
saldo: float, taxa_juros:float = 0.0):
        super().__init__(numero_conta, saldo)
        if taxa_juros < 0:
            raise ValueError("Não é possível
ter uma taxa de juros negativa!")
        elif taxa_juros > 1:
            raise ValueError("Não é possível
ter uma taxa de juros maior que 100%!")
        self._taxa_juros = taxa_juros

    def calcular_juros(self):
        self._saldo +=
self._saldo*self._taxa_juros
```

test_conta.py

```
import pytest
from banco import ContaBancaria, ContaPoupanca

def test_conta_bancaria_existe():
    conta_bancaria = ContaBancaria("0", 0)
    assert conta_bancaria

def test_conta_poupanca_existe():
    conta_poupanca = ContaPoupanca("0", 0, 0.0)
    assert conta_poupanca

def
test_conta_bancaria_nao_pode_comecar_com_saldo_
```

```
negativo():
    with pytest.raises(ValueError, match="Não é
possível iniciar uma conta com saldo
negativo!"):
        conta_bancaria = ContaBancaria("0",
-20)

def
test_conta_poupanca_nao_pode_comecar_com_saldo_
negativo():
    with pytest.raises(ValueError, match="Não é
possível iniciar uma conta com saldo
negativo!"):
        conta_poupanca = ContaPoupanca("0",
-20)

def
test_conta_bancaria_nao_pode_sacar_saldo_negati
vo():
    conta_bancaria = ContaBancaria("0", 0)
    with pytest.raises(ValueError, match="Não é
possível sacar um valor menor ou igual a
zero!"):
        conta_bancaria.sacar(-20)

def
test_conta_poupanca_nao_pode_sacar_saldo_negati
vo():
    conta_poupanca = ContaPoupanca("0", 0)
    with pytest.raises(ValueError, match="Não é
```

```
possível sacar um valor menor ou igual a
zero!"):
        conta_poupanca.sacar(-20)

def
test_conta_bancaria_nao_pode_sacar_saldo_negati
vo():
    conta_bancaria = ContaBancaria("0", 0)
    with pytest.raises(ValueError, match="Não é
possível depositar um valor menor ou igual a
zero!"):
        conta_bancaria.depositar(-0.001)

def
test_conta_poupanca_nao_pode_sacar_saldo_negati
vo():
    conta_poupanca = ContaPoupanca("0", 0)
    with pytest.raises(ValueError, match="Não é
possível depositar um valor menor ou igual a
zero!"):
        conta_poupanca.depositar(-0.001)

def
test_conta_bancaria_sacar_todo_dinheiro_aos_pou
cos():
    conta_bancaria = ContaBancaria("1029",
1000)
    for _ in range(1000):
        conta_bancaria.sacar(1)
    assert conta_bancaria.obter_saldo()==0
```

```
def
test_conta_poupanca_sacar_todo_dinheiro_aos_pou
cos():
    conta_poupanca = ContaPoupanca("1029",
1000)
    for _ in range(1000):
        conta_poupanca.sacar(1)
    assert conta_poupanca.obter_saldo()==0

def test_conta_poupanca_juros_menor_que_zero():
    with pytest.raises(ValueError, match="Não é
possível ter uma taxa de juros negativa!"):
        conta_poupanca = ContaPoupanca("1029",
1000, -0.2)

def test_conta_poupanca_juros_maior_que_um():
    with pytest.raises(ValueError, match="Não é
possível ter uma taxa de juros maior que
100%!"):
        conta_poupanca = ContaPoupanca("1029",
1000, 1.2)

def test_conta_poupanca_juros_um_porcento():
    conta_poupanca = ContaPoupanca("1029", 100,
0.01)
    conta_poupanca.calcular_juros()
    assert conta_poupanca.obter_saldo()==101

def
```

```
test_conta_poupanca_juros_cinco_porcento_tres_v
ezes():
    conta_poupanca = ContaPoupanca("1029", 100,
0.05)
    conta_poupanca.calcular_juros()
    conta_poupanca.calcular_juros()
    conta_poupanca.calcular_juros()
    assert conta_poupanca.obter_saldo()==
pytest.approx(100 * (1.05)**3)
```

- Importante: toda propriedade da classe (getter) deve ter o seu setter! Eu quebrei a cabeça um pouco para descobrir como fazer a herança da maneira correta, com encapsulação