

Programação Estruturada e Orientada a Objetos com Python

Módulo 4

Preâmbulo: Neste módulo você dominará os pilares da Programação Orientada a Objetos em Python.

Versão: 1.1

Sumário

1	informações Especificas para este Modulo	2
II	Exercício 1: Introdução a Classes	4
ш	Exercício 2: Utils	6
IV	Exercício 3: Classe	7
\mathbf{V}	Exercício 4: Encapsulamento e métodos	9
VI	Exercício 5: Herança	11
VII	Exercício 6: Mais Classes	12
VIII	Entrega e Avaliação entre Pares	14

Capítulo I

Informações Específicas para este Módulo

Escrevendo testes eficazes

Testes automatizados são ferramentas essenciais no desenvolvimento de software. Eles ajudam a garantir que seu código funcione como esperado, facilitam a manutenção e tornam mudanças futuras mais seguras. Quando bem escritos, também servem como documentação útil para você e para outras pessoas que lerem seu código.

Incorpore testes como parte natural do desenvolvimento, com foco em clareza, confiabilidade e cobertura dos principais comportamentos.

O que testar

Nem tudo precisa ser testado. Escreva testes para os comportamentos que são importantes e que envolvem alguma lógica, como:

- Regras de negócio e decisões importantes do código;
- Situações com diferentes tipos de entrada (válidas, inválidas, vazias, etc.);
- Interações entre objetos, quando um método de uma classe depende do funcionamento de outra.

O que pode ser deixado de fora

Evite gastar tempo testando comportamentos triviais ou que não agregam valor real. Por exemplo:

- Métodos que apenas retornam um valor fixo ou armazenam uma variável (como getters e setters simples);
- Chamadas a funções de bibliotecas externas (exceto se forem parte do seu fluxo);
- Impressões no console ou mensagens de log (a menos que a lógica dependa delas).

Como testar?

Você pode escrever os testes **antes** ou **durante** a implementação. Cada abordagem tem seus benefícios. O importante é garantir que os testes existam, que cubram os comportamentos relevantes e que ofereçam confiança sobre o funcionamento do seu código.

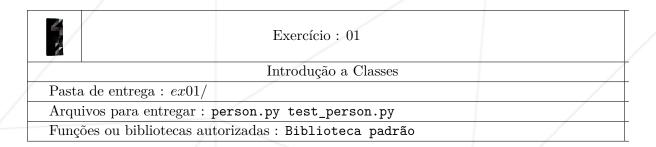
Uma boa prática é validar:

- O caso mais simples de uso.
- Um ou mais casos-limite.
- Comportamentos incorretos ou entradas inválidas.



Neste módulo, todos os exercícios devem ser acompanhados de testes automatizados. Eles farão parte da entrega e da avaliação.

Capítulo II



Em Python, uma classe é uma forma de organizar dados (chamados de atributos) e comportamentos (chamados de métodos) relacionados. Classes nos permitem modelar conceitos do programa, como usuários, produtos, ou, neste caso, uma pessoa.

Neste exercício, você criará a classe Person para representar uma pessoa com nome, idade e um comportamento que simula um aniversário, aumentando a idade em um ano.

- 1. Crie um arquivo chamado person.py e implemente a classe Person com os seguintes atributos:
 - name: string
 - age: int
- 2. O construtor da classe deve receber esses dois valores nesta ordem, como no protótipo def __init__(self, name: str, age: int):
- 3. Crie um método def birthday(self) -> None que aumenta a idade da pessoa em 1.
- 4. A classe deve se comportar conforme o exemplo abaixo:

```
>>> p = Person("Alice", 30)
>>> p.name
'Alice'
>>> p.age
30
>>> p.birthday()
>>> p.age
```

1. Abaixo, um teste simples do comportamento de sua classe:

```
from person import Person

def test_person_initialization():
    p = Person("Alice", 30)
    assert p.name == "Alice"
    assert p.age == 30
```



class Person

Capítulo III



Exercício: 02

Utils

Pasta de entrega : ex02/

Arquivos para entregar: utils.py, test_utils.py

Funções ou bibliotecas autorizadas : Biblioteca padrão

- 1. Crie uma função format_cents no arquivo utils.py que permita fazer a conversão de int para string com o formato adequado. Você poderá utilizá-la nas classes que criar adiante.
 - Valores positivos: [+] R\$ X.XXX,XX
 - Valores negativos: [-] R\$ X.XXX,XX
- 2. A função deverá se comportar da seguinte maneira:

```
>>> import utils
>>> print(utils.format_cents(11_222_00))
[+] R$ 11.222,00
```



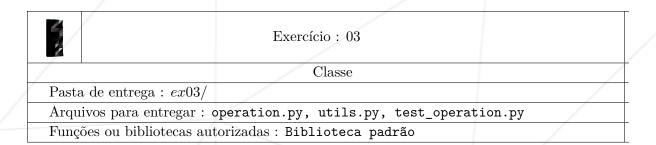
No Python, o caractere de sublinhado (_) é frequentemente usado em números inteiros para melhorar a legibilidade, sem afetar o valor do número.



Utilize os métodos implementados no tipo 'str' para ajudar a fazer a formatação.

f-strings também podem ser úteis.

Capítulo IV



- 1. Implemente uma classe simples chamada Operation para representar uma transação financeira.
- 2. A classe deve possuir três atributos públicos:
 - - cents: um int representando um valor em centavos (1234 significa R\$ 12,34);
 - - operation_type: uma string com a natureza da operação;
 - - description: uma string com a descrição da operação;
- 3. O construtor da classe deve aceitar apenas os argumentos cents e description, nesta ordem.
- 4. O atributo operation_type, deverá ser automaticamente definido com base no sinal do valor:
 - - Se cents > 0, então operation_type = 'credit'
 - - Se cents < 0, então operation_type = 'debit'
- 5. O valor de cents deve ser diferente de zero, ou uma exceção do tipo ValueError deverá ser levantada. Como estamos criando uma classe para ser consumida externamente (por um programa, no caso), a exceção não deverá ser tratada

1. Para facilitar a depuração, a representação de uma instância de Operation deve aparecer assim no console do Python:

```
>>> t = Operation(11_222_00, 'ATM deposit')
>>> t
Operation(cents=1122200, operation_type='credit', description='ATM deposit')
```

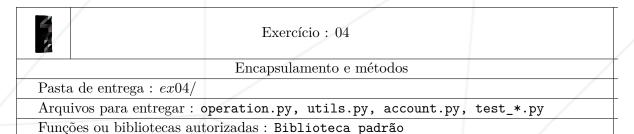
2. Para facilitar a exibição para usuários finais, quando exibida via print a instância de Operation deve aparecer assim:

```
>>> print(t)
[+] R$ 11.222,00 (ATM deposit)
```



'Classe simples', neste exercício, significa uma classe que herda diretamente - e implicitamente - de 'object', sem utilizar '@dataclass'. A classe não deve usar @dataclass. Implemente os métodos especiais __init__, __repr__ e __str__ manualmente.

Capítulo V



Crie uma classe simples Account.

- 1. O construtor deve aceitar um identificador de conta como int e um CPF como string.
- 2. Os atributos deverão ser:
 - account_id: int
 - cpf: string
 - __balance: atributo privado inicializado com 0
 - operations: atributo privado que conterá a lista de operações realizadas
- 3. Os métodos públicos da classe deverão ser:
 - deposit(self, amount: int, description: string): realiza um depósito na conta, aumentando o saldo. O valor é dado em centavos.
 - withdraw(self, amount: int, description: string): realiza um saque da conta, diminuindo o saldo.
 - statement(self): exibe as operações e balanço da conta.
- 4. Caso o saldo seja insuficiente ou o valor seja inválido, deve levantar uma exceção.
- 5. Você pode criar métodos adicionais que julgar necessário.

- 6. Você também deve implementar os dunder methods __str__ e __repr__
- 7. Veja abaixo o comportamento esperado no console.

```
>>> ac = Account(123, '123.456.789-01')
>>> ac
Account(123, '123.456.789-01')
>>> print(ac)
Account: 123
Balance: [+] R$ 0,00
>>> ac.deposit(1122200, 'ATM deposit')
>>> ac.statement()
[+] R$ 11.222,00 (ATM deposit)
Balance: [+] R$ 11.222,00
```

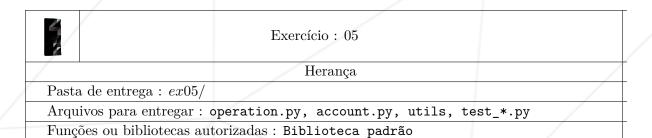


Os métodos 'deposit' e 'withdraw' só devem aceitar operações com valor > 0, caso contrário deve levantar a exceção 'ValueError' com a mensagem 'valor deve ser > 0'.



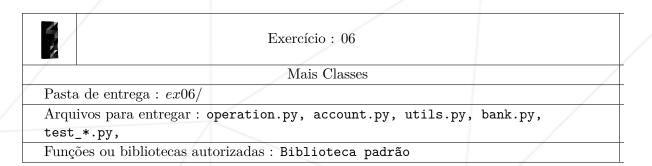
Não se preocupe em validar os CPFs.

Capítulo VI



- 1. Crie uma nova exceção derivada de Exception com o nome InsufficientBalance. Ajuste sua classe Account para utilizá-la.
- 2. Crie um novo tipo derivado de Enum, chamado OperationType. Ajuste sua classe Operation para utilizá-lo em operation_type, ao invés de um tipo 'str'.

Capítulo VII



- 1. Crie uma classe simples Bank para gerenciar contas bancárias. Esta classe deverá armazenar contas e permitir operações entre elas.
- 2. A classe deve conter um atributo privado __accounts para armazenar Accounts.
- 3. Os seguintes métodos públicos devem ser implementados:
 - add_account(account: Account)
 - get_account_by_cpf(cpf: str)
 - get account by id(account id: int)
 - transfer(source_account: int, destination_account: int, value: int, description: string)
- 4. O método transfer deve realizar um saque na conta de origem e um depósito na conta de destino, ambos com o mesmo valor e descrição.

Sua classe deverá se comportar da seguinte forma:

```
>>> bank = Bank()
>>> ac1 = Account(123, '123.456.789-01')
>>> ac2 = Account(456, '234.567.890-12')
>>> bank.add_account(ac1)
>>> bank.add_account(ac2)
>>> len(bank)
```

```
>>> 123 in bank
True
>>> bank[123]
Account(123, '123.456.789-01')
>>> bank[123].deposit(10000, "Initial deposit")
>>> bank.transfer(123, 456, 5000, "Payment")
>>> bank[456].statement()
[+] R$ 50,00 (Payment)
Balance: [+] R$ 50,00
```



Adicione exceções personalizadas e implemente o tratamento de erros conforme julgar necessário.



Os getters devem retornar apenas a primeira instância encontrada.



Dunder methods: __contains__, __len__, __getitem__

Capítulo VIII

Entrega e Avaliação entre Pares

- Entregue seu projeto em seu repositório *Git* disponível na página do projeto na intranet.
- Apenas o trabalho dentro do seu repositório será avaliado durante a defesa. Não hesite em verificar os nomes de seus arquivos e pastas para garantir que estejam corretos.
- No horário da avaliação, o avaliado se dirigirá à estação de trabalho do aluno avaliador para realizar os testes. Um clone do repositório deverá ser realizado em uma nova pasta, e estes são os arquivos que serão avaliados.