Python: utilizando POO na Engenharia de Dados

Marcus Almeida
30/05/2023

O bom desenvolvimento de código é essencial na Engenharia de Dados para a construção de sistemas robustos, eficientes, e que consigam trabalhar com fluxos otimizados e grandes volumes de dados. Para facilitar o desenvolvimento do produto em equipe, é importante seguir também boas práticas de Engenharia de Software, e nesse contexto, surge o uso de paradigmas de programação diferentes, como a **Programação Orientada a Objetos** (POO).

Nesse artigo, vamos conhecer um pouco mais sobre a Programação Orientada a Objetos através de exemplos de código em Python explorando as características desse conceito.

O que é Programação Orientada a Objetos (POO)?

A Programação Orientada a Objetos (POO), também conhecida no inglês como *Object Oriented Programming (OOP)*, é um **paradigma de programação**, uma maneira de organizar código que propõe a ideia de pôr em conjunto as seções do código com dados e comportamentos relacionados, de maneira **encapsulada**. É também uma abordagem poderosa para estruturar softwares para análises e interpretação de grandes volumes de dados.

Como o nome do paradigma sugere, esse formato se baseia na criação de **objetos**, que são os representantes de um modelo (ou ideia) que chamamos de **classe**. Uma classe, por sua vez, é uma estrutura que possui atributos (características) e métodos (funções) associados a ela.

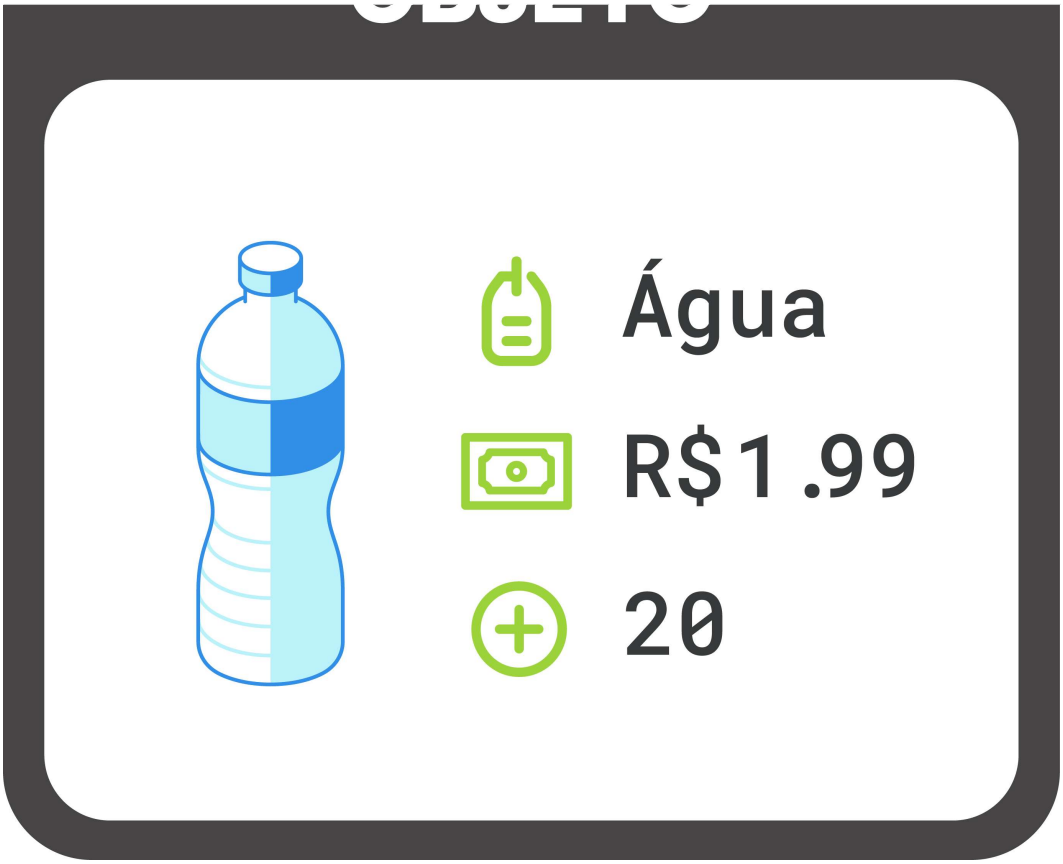
Para exemplificar, podemos pensar na definição de uma classe para representar um **Produto** em um estoque. E vamos considerar que esse produto tem atributos de *nome*, *preço* e *quantidade*. O código abaixo mostra a criação dessa classe no Python.

```
class Produto:

  def __init__(self, nome, preco, quantidade):
    self.nome = nome
    self.preco = preco
    self.quantidade = quantidade
```

A partir desse momento, já temos um modelo do que é um produto. E pode-se criar as instâncias desse modelo, que chamamos de objetos. Por exemplo, criamos um objeto nomeado `p1` que possui os atributos: nome “Água”, preço “0.99”, e quantidade “20”.

```
# Criação de uma instância da classe Produto, chamada produto1.
p1 = Produto("Água", 1.99, 20)
```



E pronto, criamos um objeto da nossa classe Produto. Para conseguirmos observar uma resposta clara sobre seus atributos e/ou comportamentos, precisamos criar um **método** para realizar ações com essa classe.

Criação de métodos

Uma classe é dotada de **métodos**, que são funções para **manipular** atributos da classe e representar todos os seus comportamentos. Por exemplo, poderíamos fazer um método chamado `mostrar_info` para mostrar os valores de nome, preço e quantidade.

```
class Produto:

  def __init__(self, nome, preco, quantidade):
    self.nome = nome
    self.preco = preco
```

```
    print(f"Nome: {self.nome}")
    print(f"Preço: R${self.preco}")
    print(f"Quantidade: {self.quantidade}")
```

E agora, podemos instanciar novamente um objeto novo, e chamar esse método da classe.

```
p1 = Produto("Água", 1.99, 20)
p1.mostrar_info()
```

Feito isso, teremos a saída do código com as informações do nosso produto:

```
Nome: Água
Preço: R$1.99
Quantidade: 20
```

Para um segundo produto, também teremos o mesmo funcionamento:

```
p2 = Produto("Refrigerante", 4.99, 25)
p2.mostrar_info()
```

```
Nome: Refrigerante
Preço: R$4.99
Quantidade: 25
```

O trabalho com métodos também permite criar comportamentos e regras utilizando os atributos da classe. Por exemplo, podemos criar um método `mostrar_valor_total_estoque()` , que calcula a multiplicação do valor da quantidade e o valor do preço unitário dos produtos disponíveis em estoque.

```
class Produto:

  def __init__(self, nome, preco, quantidade):
    self.nome = nome
    self.preco = preco
    self.quantidade = quantidade

  def mostrar_info(self):
    print(f"Nome: {self.nome}")
    print(f"Preço: R${self.preco}")
    print(f"Quantidade: {self.quantidade}")

  def mostrar_valor_total_estoque(self):
    valor_total = self.preco * self.quantidade
    print(f"O valor total de estoque deste produto é R${round(valo
```

```
p2.mostrar_valor_total_estoque()
```

```
Nome: Refrigerante
Preço: R$4.99
Quantidade: 25
O valor total de estoque deste produto é R$124.75
```

E pronto, temos dois métodos para mostrar informações gerais sobre um produto, e também mostrar o valor total que será obtido com a venda desses itens.

Pilares da Programação Orientada a Objetos

São quatro os pilares da programação orientada a objetos:

- Abstração
- Encapsulamento
- Herança
- Polimorfismo

Vamos conhecer cada um desses princípios?

Abstração

Essa característica nos permite representar modelos e estruturas complexas de forma simplificada e fácil de entender, por meio do uso de classes e métodos, que vão representar comportamentos e estados de um objeto. Por exemplo, uma classe chamada *Animal* pode ser construída como um modelo que tem informações como "nome" e "tamanho", e comportamentos como "comer" e "dormir". Essa classe pode representar vários animais diferentes, de forma **abstraída**, sem especificar diretamente o animal, mas conseguindo representá-los.

Outro exemplo abstraído seria um modelo de Machine Learning para regressão, como uma Regressão Linear ou Logística. Essas duas técnicas possuem métodos em comum, um de treino (`fit`) e outro para predição (`predict`), que podemos generalizar e construir uma classe `Regressor` . Essa classe, por sua vez, pode depois servir de base para construção de outras classes derivadas.

organizar funções e métodos semelhantes em um mesmo objeto, ocultando informações internas que não deveriam ser acessadas por outros métodos e/ou objetos. O Python providencia mecanismos para isolar e restringir acesso de atributos em métodos fora de uma classe. A maneira mais comum é por meio de uma técnica chamada name mangling.

Herança

Representa a capacidade de criar novas classes a partir de outras classes existentes, com a função de estender funcionalidades, e possibilita reutilizar código. A herança permite que classes **compartilhem comportamentos** e atributos comuns, ao mesmo tempo que permitem que as classes filhas derivadas possuam novas funcionalidades específicas. Por meio da herança, também é possível definir uma ordem de hierarquia entre os objetos.

Retomando ao exemplo da classe `Produto`, nós poderíamos criar uma classe derivada chamada `ProdutoPerecivel`. Essa classe derivada tem todas as informações da classe pai e possui um atributo adicional chamado `data_validade` para representar o vencimento do produto.

A classe `ProdutoPerecivel` pode ser construída da seguinte forma:

```
class ProdutoPerecivel(Produto):

  # Adição de um novo atributo "data_validade"
  def __init__(self, nome, preco, quantidade, data_validade):
    super().__init__(nome, preco, quantidade)
    self.data_validade = data_validade

  # Novo comportamento de mostrar a validade do Produto.
  def mostrar_validade(self):
    print(f"O produto vence no dia {self.data_validade}")
```

Ao criar um novo objeto da classe `ProdutoPerecivel`, podemos utilizar tanto os métodos da classe pai (Produto), quanto também o método novo `mostrar_validade`.

```
p3.mostrar_validade() # Método novo
```

```
Nome: Leite
Preço: R$7.99
Quantidade: 10
O valor total de estoque deste produto é R$79.9
O produto vence no dia 10/05/2023
```

Polimorfismo

Representa a capacidade de um objeto ser utilizado com comportamentos de maneiras diferentes, a depender do contexto em que é inserido. Essa característica é trabalhada por meio da **sobrecarga de métodos**, uma técnica em que implementações diferentes são fornecidas por meio de diferentes tipos de parâmetros ou por meio de herança, onde, por exemplo, as classes filhas podem ter comportamentos diferentes das classes pai (super classes).

No caso das classes `Produto` e `ProdutoPerecivel`, o método `mostrar_info()` pode ter um comportamento diferente na classe derivada `ProdutoPerecivel`, onde podemos adicionar um aviso informando que o produto é perecível.

```
class ProdutoPerecivel(Produto):

  def __init__(self, nome, preco, quantidade, data_validade):
    super().__init__(nome, preco, quantidade)
    self.data_validade = data_validade

  def mostrar_validade(self):
    print(f"O produto vence no dia {self.data_validade}")
```

```
    print(f"Esse produto é perecível!")
    print("="*30)

p4 = ProdutoPerecivel('Maçã', 0.99, 15, '28/06/2025')
p4.mostrar_info()
p4.mostrar_valor_total_estoque()
p4.mostrar_validade() # Método sobrecarregado
```

```
Nome: Maçã
Preço: R$0.99
Quantidade: 15
==============================
Esse produto é perecível!
==============================
O valor total de estoque deste produto é R$14.85
O produto vence no dia '28/06/2025'
```

A característica de uma classe derivada compartilhar o mesmo nome de uma classe pai facilita a aplicação e uso do mesmo método em laços de repetição e outras situações quando precisa-se usar objetos distintos de várias classes derivadas. No exemplo abaixo, utilizamos um laço de repetição para aplicar o mesmo método `mostrar_info()` em uma lista com um carrinho de compras dos objetos criados.

```
carrinho_produtos = [
    Produto("Água", 1.99, 20),
    Produto("Refrigerante", 4.99, 25),
    ProdutoPerecivel('Leite', 7.99, 10, '10/05/2023'),
    ProdutoPerecivel('Maçã', 0.99, 15, '28/06/2025')
]
```

```
    print("-=" * 30)
```

```


Nome: Água
Preço: R$1.99
Quantidade: 20
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
Nome: Refrigerante
Preço: R$4.99
Quantidade: 25
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
Nome: Leite
Preço: R$7.99
Quantidade: 10
==============================
Esse produto é perecível!
==============================
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
Nome: Maçã
Preço: R$0.99
Quantidade: 15
==============================
Esse produto é perecível!
==============================
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
```

Benefícios de utilizar POO em Engenharia de Dados

Profissionais que trabalham com programação e dados, principalmente em papéis de [Engenharia de Dados](), precisam estar alinhados com práticas de

grandes equipes.

Para ajudar a sanar essas necessidades, o paradigma POO possui vários benefícios que atendem a esses requisitos. Dentre eles, podemos listar os principais:

1. **Reutilização de código**: Por meio da herança e composição, podemos criar novas classes com base em outras já existentes. Economizando tempo e esforço, sem reescrita de código do zero. Processos como limpeza e tratamento de dados, pipelines de ETL, conexões com bancos e fontes de dados podem ser feitos através do uso de classes, o que pode reduzir o tamanho do código e melhorar a legibilidade.
2. **Modularidade**: Ao utilizar o encapsulamento, podemos construir partes do código dedicadas a funções específicas, criando também métodos que ocultam detalhes de implementação e expõem apenas a interface pública de um objeto.
3. **Abstração**: É possível representar objetos complexos de forma mais simples e fácil de entender, tornando o código mais legível e compreensível para outros devs. Pode-se trabalhar desde modelos matemáticos e processos de fluxo de trabalho até a estrutura de dados específicas de cada projeto.
4. **Flexibilidade**: O código permite adaptação à mudanças e adição de novos recursos mais facilmente, de acordo com a necessidade de cada negócio. Em cenários de Big Data, a escalabilidade é um dos fatores principais para garantir que uma aplicação e projeto continue apresentando bom desempenho.
5. **Segurança**: A interação entre os objetos pode ser limitada quanto a dados internos de um objeto, promovendo um meio mais seguro de interação e prevenindo o acesso não-autorizado.
6. **Maior produtividade**: A reutilização de código e modularidade tornam o código uma prática produtiva para a equipe, pois facilita a criação, o teste e a manutenção de código, reduzindo a quantidade de trabalho manual.

Conclusão