Programação Orientada a Objetos (POO) em Python — Conceitos Fundamentais

A Programação Orientada a Objetos (POO) é um dos paradigmas mais utilizados no desenvolvimento de software moderno. Ela fornece uma forma poderosa e expressiva de estruturar programas, permitindo que sistemas complexos sejam modelados de maneira mais intuitiva, baseada em entidades do mundo real. O coração da POO está nos conceitos de **classe**, **objeto**, **encapsulamento**, **herança**, **polimorfismo**.

Classe e Objeto

Em POO, uma **classe** é uma estrutura que define ações (métodos) e os dados (atributos) de um conjunto de objetos. Pense nela como um projeto ou um molde. Já um **objeto** é uma instância concreta dessa classe — com atributos específicos e capaz de realizar ações.

```
class Pessoa:
        def __init__(self, nome: str, idade: int):
        self.nome = nome
        self.idade = idade


        def apresentar(self) -> None:
        print(f"Olá, meu nome é {self.nome} e tenho {self.idade} anos.")


p1 = Pessoa("Pedro", 20)
p1.apresentar()
```

Por que usar classes e objetos?

- Para modelar entidades reais do sistema.

- Para reaproveitar código em diferentes partes do programa.
- Para organizar e modularizar funcionalidades.

Encapsulamento

O **encapsulamento** refere-se ao princípio de ocultar os detalhes internos de uma classe, expondo apenas o necessário para o uso externo. Em Python, embora não existam modificadores de acesso rígidos como `public`, `private` e `protected` de outras linguagens, há convenções:

- `atributo`: público — pode ser acessado diretamente.
- `_atributo`: protegido — sinaliza que o uso deve ser restrito à própria classe ou subclasses.
- `__atributo`: privado — impede acesso direto externo.

Getters e Setters com `@property`

Para proteger os dados e ainda permitir acesso controlado, utilizamos os métodos **getter** e **setter**, especialmente através da funcionalidade `@property` do Python.

```
class Pessoa:
    def __init__(self, nome:str):
        self._nome = nome

    @property
    def nome(self) ->str:
        return self._nome
```

```
@nome.setter
    def nome(self, novo_nome) ->None:
        if isinstance(novo_nome, str) and novo_nome.strip():
            self._nome = novo_nome
        else:
            raise ValueError("Nome inválido")
```

Por que usar encapsulamento?

- **Proteção de dados sensíveis**.
- **Validação de entrada antes da modificação**.
- **Estabilidade para código cliente**: o uso de `@property` permite transformar métodos em atributos sem afetar quem já utiliza a classe.

Herança

A **herança** permite criar uma nova classe a partir de uma classe existente. Isso é útil quando há um relacionamento de especialização entre entidades.

```
class Pessoa:
    def __init__(self, nome:str, idade:int):
        self.nome = nome
        self.idade = idade

class Aluno(Pessoa):
    def __init__(self, nome:str, idade:int, matricula:str):
        super().__init__(nome, idade)
        self.matricula = matricula
```

A classe `Aluno` herda os atributos e métodos da classe `Pessoa`, podendo adicionar novos comportamentos ou sobrescrever os existentes.

Por que usar herança?

- **Reutilização de código comum** entre classes relacionadas.
- **Organização hierárquica** do sistema (por exemplo: Pessoa > Aluno > Bolsista).
- **Facilidade de manutenção**: alterações na superclasse propagam-se.

Porém, herança excessiva pode levar a hierarquias confusas. Use com moderação.

Polimorfismo

O **polimorfismo** permite que classes diferentes implementem métodos com o mesmo nome, mas comportamentos distintos. Isso torna o código mais genérico e adaptável.

```
class Pessoa:
    def falar(self) -> None:
        print("Pessoa falando...")

class Aluno(Pessoa):
    def falar(self) -> None:
        print("Aluno apresentando trabalho.")

def apresentar(pessoa) -> None:
    pessoa.falar()

apresentar(Pessoa())
apresentar(Aluno())
```

Neste exemplo, o método `falar()` se comporta de forma distinta dependendo da instância. Isso é **polimorfismo de sobrescrita (override)**.

Por que usar polimorfismo?

- Para **abstrair diferenças entre objetos relacionados**.
- Para permitir que **métodos sejam chamados sem conhecer a classe exata**.
- Para tornar o sistema **mais flexível e extensível**.

Métodos de Classe e Fábricas

Métodos de classe, definidos com `@classmethod`, recebem a referência à própria classe (`cls`) como primeiro argumento. São ideais para **métodos fábrica**, que criam instâncias com parâmetros pré-definidos.

```
class Pessoa:
    def __init__(self, nome, idade):
        self.nome = nome
        self.idade = idade

    @classmethod
    def criar_pessoa_20(cls, nome):
        return cls(nome, 20)

p = Pessoa.criar_pessoa_20("Maria")
```

Classes Abstratas (e Abstração)

A **abstração** permite representar conceitos genéricos em estruturas reutilizáveis. Em Python, usamos `abc.ABC` e `@abstractmethod` para criar classes abstratas que **não podem ser instanciadas diretamente**, e que **exigem que subclasses implementem determinados métodos**.

```
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def emitir_som(self):
        pass

class Cachorro(Animal):
    def emitir_som(self):
        print("Au au!")
```

Relações entre classes

Esses conceitos representam diferentes tipos de relacionamento entre objetos:

- **Associação**: relação genérica onde uma classe usa outra, sem dependência forte. Exemplo: um Professor pode dar aula para vários Alunos.

- **Agregação**: relação "todo-parte" onde os objetos podem existir separadamente. Exemplo: uma Sala contém Cadeiras, mas se a Sala for destruída, as Cadeiras ainda existem.

- **Composição**: relação mais forte de "todo-parte" onde as partes não existem sem o todo. Exemplo: um Corpo humano é composto por órgãos; se o Corpo for destruído, os órgãos também deixam de existir.

Boas Práticas na POO

- **Favor composição sobre herança**: quando possível, prefira compor objetos a criar hierarquias profundas. Composição torna o código mais flexível e fácil de modificar, já que evita o acoplamento rígido entre classes.

- **Mantenha o princípio da responsabilidade única**: cada classe deve ter uma única responsabilidade ou motivo para mudar. Isso facilita a manutenção, testes e entendimento do código.

- **Use encapsulamento para proteger o estado interno do objet:** proteja o estado interno dos objetos, fornecendo apenas os métodos necessários para sua manipulação. Isso reduz o risco de efeitos colaterais inesperados e melhora a segurança e a integridade dos dados.

- **Documente classes e métodos**: facilite a compreensão e reutilização.

- **Evite acoplamento excessivo entre classes**: classes muito dependentes umas das outras dificultam testes, manutenção e reaproveitamento. Prefira interfaces bem definidas e mantenha as dependências sob controle.