POO

KARINA CASOLA

SENAC - FUTURO PROGRAMADOR

Paradigmas de Programação

Paradigmas de Programação: Os paradigmas de programação são estilos ou abordagens de resolução de problemas na computação. Cada paradigma oferece ferramentas e técnicas específicas para desenvolver software.

Principais Paradigmas:

- **Imperativo:** Instruções sequenciais que modificam o estado do programa (ex.: C, Python).
- **Funcional:** Baseado em funções matemáticas e imutabilidade (ex.: Haskell, Lisp).
- **Orientado a Objetos (OO):** Estruturado em objetos que encapsulam dados e comportamentos (ex.: Python, Java).
- **Declarativo:** Define o que deve ser feito, mas não como (ex.: SQL, Prolog).

Conceituação e Origens da Orientação a Objetos

Conceito: A Orientação a Objetos (OO) é um paradigma de programação baseado na criação de objetos que representam entidades do mundo real. Esses objetos são definidos por atributos (dados) e métodos (comportamentos). OO busca maior organização, reutilização e abstração no desenvolvimento de software.

Origem: - Nasceu na década de 1960 com a linguagem Simula, que introduziu conceitos de classes e objetos. - Ganhou popularidade com Smalltalk nos anos 1970 e 1980, influenciando linguagens modernas como Python, Java e C++.

Implementação em Python

Em Python, OO é suportada de forma natural e poderosa. Os principais conceitos incluem: - Classes e Objetos - Encapsulamento - Herança - Polimorfismo

Conceito de Herança

Definição: A herança é um mecanismo da programação orientada a objetos que permite que uma classe (chamada de classe derivada ou filha) reutilize os atributos e métodos de outra classe (chamada de classe base ou pai).

Vantagens da Herança:

- Promove a reutilização de código.
- Facilita a manutenção e extensão do software.
- Possibilita a criação de hierarquias de classes.

Exemplo de Hierarquia de Classes:

- **Classe Pai:** Animal
- **Classe Filha:** Cachorro, Gato

Conceito de Polimorfismo

Definição: Polimorfismo significa "muitas formas"e permite que objetos de diferentes classes sejam tratados de forma uniforme se compartilharem métodos ou interfaces comuns.

Tipos de Polimorfismo:

- **Polimorfismo de Sobrecarga:** Um método pode ter diferentes implementações com base no número ou tipo de argumentos.
- **Polimorfismo de Sobrescrita:** Classes derivadas podem redefinir métodos da classe base.

Vantagens:

- Reduz a complexidade do código.
- Permite maior flexibilidade e extensibilidade no design do software.

Exemplo de Herança

```python
class Animal:
    def __init__(self, nome):
        self.nome = nome

    def emitir_som(self):
        pass

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

class Gato(Animal):
    def emitir_som(self):
        return "Miau!"

# Exemplo de Uso
cachorro = Cachorro("Bobby")
gato = Gato("Mimi")
print(cachorro.emitir_som())  # Saída: Au au!
print(gato.emitir_som())      # Saída: Miau!
```

Exemplo de Polimorfismo

```python
class Forma:
    def area(self):
        pass

class Retangulo(Forma):
    def __init__(self, largura, altura):
        self.largura = largura
        self.altura = altura

    def area(self):
        return self.largura * self.altura

class Circulo(Forma):
    def __init__(self, raio):
        self.raio = raio

    def area(self):
        return 3.14 * (self.raio ** 2)

# Exemplo de Uso
formas = [Retangulo(5, 10), Circulo(7)]
for forma in formas:
    print(f"Área: {forma.area()}")
```

Exemplo: Classes e Objetos

```
class Animal:
    def __init__(self, nome, especie):
        self.nome = nome
        self.especie = especie

    def apresentar(self):
        return f"Eu sou {self.nome}, um(a) {self.especie}."

cachorro = Animal("Bobby", "Cachorro")
print(cachorro.apresentar())
```

Exemplo: Encapsulamento

```
class ContaBancaria:
    def __init__(self, titular, saldo):
        self.__titular = titular
        self.__saldo = saldo  # Atributo privado

    def depositar(self, valor):
        self.__saldo += valor

    def exibir_saldo(self):
        return f"Saldo: R$ {self.__saldo:.2f}"

conta = ContaBancaria("Karina", 1000)
conta.depositar(500)
print(conta.exibir_saldo())
```

Exemplo: Herança

```python
class Veiculo:
    def __init__(self, marca):
        self.marca = marca

class Carro(Veiculo):
    def __init__(self, marca, modelo):
        super().__init__(marca)
        self.modelo = modelo

meu_carro = Carro("Toyota", "Corolla")
print(f"Marca: {meu_carro.marca}, Modelo: {meu_carro.modelo}")
```

Exemplo: Polimorfismo

```python
class Forma:
    def area(self):
        pass

class Retangulo(Forma):
    def __init__(self, largura, altura):
        self.largura = largura
        self.altura = altura

    def area(self):
        return self.largura * self.altura

retangulo = Retangulo(5, 10)
print(f"Área do Retângulo: {retangulo.area()}")
```

Interligação com Funções e Modularização

A Orientação a Objetos está interligada com funções e modularização porque: - Métodos dentro de classes são funções. - Classes e objetos são organizados em módulos, promovendo reutilização.

Exemplo de Modularização:

Exemplo: Modularização

```python
# modulo.py
class Calculadora:
    def somar(self, a, b):
        return a + b

# main.py
from modulo import Calculadora

calc = Calculadora()
print(calc.somar(5, 3))
```

Exercício 1: Classe com Encapsulamento

Enunciado: Crie uma classe `Produto` com atributos privados `nome` e `preco`. Implemente métodos para acessar esses atributos e calcular o preço com desconto.

Exercício 2: Herança e Polimorfismo

Enunciado: Crie uma hierarquia de classes que representem veículos. A classe `Veiculo` deve ser a base, enquanto `Carro` e `Moto` devem ser derivadas com métodos específicos.

Exercíco 1: Classe Simples

Enunciado: Crie uma classe chamada `Pessoa`, com os atributos `nome` e `idade`, e um método `apresentar()` que exiba essas informações.

Exemplo 2: Encapsulamento

Enunciado: Crie uma classe `ContaBancaria` com atributos privados `titular` e `saldo`, e métodos para depositar e verificar saldo.

Resolução 2: Encapsulamento

```python
class ContaBancaria:
    def __init__(self, titular, saldo):
        self.__titular = titular
        self.__saldo = saldo

    def depositar(self, valor):
        self.__saldo += valor

    def exibir_saldo(self):
        return f"Saldo: R$ {self.__saldo:.2f}"

conta = ContaBancaria("Karina", 1000)
conta.depositar(500)
print(conta.exibir_saldo())
```

Exemplo 3: Herança e Polimorfismo

Enunciado: Crie uma classe base `Animal` com um método genérico `emitir_som`, e classes derivadas `Cachorro` e `Gato` que sobrescrevem esse método.

Resolução 3: Herança e Polimorfismo

```python
class Animal:
    def emitir_som(self):
        pass

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

class Gato(Animal):
    def emitir_som(self):
        return "Miau!"

cachorro = Cachorro()
gato = Gato()
print(cachorro.emitir_som())  # Saída: Au au!
print(gato.emitir_som())      # Saída: Miau!
```

Exercício 4: Sistema de Biblioteca

Enunciado: Crie uma classe `Biblioteca` com métodos para adicionar e listar livros. Os livros devem ser representados por uma outra classe `Livro`, com atributos `título` e `autor`.

Exercício 5: Sistema de Pedidos

Enunciado CRUD: Crie uma classe `Pedido` com atributos `cliente` e `itens`. Adicione métodos para adicionar itens ao pedido e calcular o valor total. Cada item deve ser representado por uma classe separada `Item`.

Enunciado

Crie um sistema de gerenciamento de usuários utilizando os conceitos de Orientação a Objetos e as operações CRUD.

1. Classe `Usuario`: - Contém os atributos: `id`, `nome`, e `email`. - Inclua um método para exibir as informações do usuário.

2. Classe `GerenciadorUsuarios`: - Gerencia uma lista de objetos `Usuario`. - Deve conter os seguintes métodos: - `adicionar_usuario`. - `listar_usuarios`. - `atualizar_usuario`. - `remover_usuario`.

3. Crie um menu simples para testar as funcionalidades.