Programação Orientada a Objetos com

Python

Prof. Saulo Santos

Sumário

21.1. Introdução

A Programação Orientada a Objetos (POO) é uma maneira de organizar e estruturar o código de um programa, pensando nele em termos de "objetos". Imagine objetos como peças de um quebra-cabeça, onde cada peça tem suas próprias características e funções. Esses objetos são criados a partir de "modelos" chamados "classes".

Cada objeto pode ser visto como uma representação de algo do mundo real, como um carro ou uma pessoa. Cada um desses objetos possui características próprias (chamadas de atributos) e ações que pode realizar (chamadas de métodos). Por exemplo, um objeto "Carro" pode ter características como "cor" e "modelo", e ações como "ligar" e "desligar".

O objetivo da POO é tornar o código mais organizado e fácil de manter. Ao usar esse modelo, você pode criar pedaços de código que são reutilizáveis e que podem ser facilmente modificados sem afetar o restante do programa. Isso ajuda a evitar problemas e a tornar o desenvolvimento mais ágil.

22.2. Classes e Objetos

Pense nas classes como formas de bolo. Ao tirarmos um bolo de uma forma, este terá o formato desta forma e esta forma sempre produzirá bolos com o mesmo formato. Assim são as classes, funcionam como moldes para a produção algum tipo de objeto. No entanto, apesar de criar bolos no mesmo formato, a classe permite especificar características que podem tornar os bolos independentes entre si. Por exemplo, ao retirar um bolo do forno, podemos adicionar recheios e coberturas que os tornem distintos. A esse conceito, damos o nome de atributos de uma classe. Um outro exemplo que posso sugerir é o da confecção de robôs em uma linha de produção. Todos os robôs saem da linha de produção idênticos entre si, pois possuem apenas uma estrutura metálica. Numa segunda etapa, eles recebem olhos, cabelo e pele sintéticos que podem diferenciá-los.

Vamos praticar com um primeiro exemplo por meio da criação de uma classe que representará uma máquina que constrói automóveis, ou seja, vamos criar um programa para gerenciar uma montadora de automóveis. A classe será chamada de **Montadora** e as características básicas do automóvel que poderão ser diferentes entre si serão: **cor**, **tipo_combustivel** e **velocidade .**

Vamos praticar! Crie um arquivo novo denominando-o 01_poo.py e digite o seguinte código:

```
class Montadora:
    cor=''
    velocidade=''
    tipo_combustivel=''
```

O código acima cria uma classe denominada Montadora com três atributos. Agora, vamos testar essa classe criando um carro ou melhor, instanciado um objeto. É importante que você mantenha a indentação do código: A instanciação do objeto deve ser feita fora da classe.

Veja como fica:

```
class Montadora:
    cor=''
    velocidade=''
    tipo_combustivel=''


objeto_1 = Montadora()
objeto_2 = Montadora()
```

Neste caso, temos 2 automóveis idênticos (objeto_1 e objeto_2) os quais serão estilizados da seguinte forma:

```
objeto_1.cor = 'preta'
objeto_1.tipo_combustivel='flex'
objeto_1.velocidade=0.0


objeto_2.cor = 'branca'
objeto_2.tipo_combustivel='diesel'
objeto_2.velocidade=0.0
```

Podemos imprimir os valores dos atributos:

```
print('Atributos de objeto_1:',objeto_1.cor, objeto_1.tipo_combustivel, objeto_1.velocidade)
print('Atributos de objeto_2:',objeto_2.cor, objeto_2.tipo_combustivel, objeto_2.velocidade)
```

Método Construtor

O que temos até o momento? Temos uma classe e dois objetos instanciados com seus atributos definidos e impressos. O que foi feito funciona, entretanto, essa não é a melhor abordagem. A elaboração de uma classe não é constituída apenas de atributos; ela possui também o que se denominada de métodos. Métodos são muito semelhantes a funções da programação procedural (imperativa) e são uma forma de manipular os atributos de uma classe, ou seja, permitem que ações sejam implementadas para os atributos. Antes de criarmos os métodos para manipular estas ações, vamos tratar de um método especial denominado método construtor. Um método construtor é automaticamente chamado (carregado) no momento em que um objeto é instanciado e tem como função transferir dados para os atributos do objeto que está sendo instanciado. Veja o exemplo que reescreve o exemplo anterior:

```
class Montadora:
    def __init__(self,p1,p2,p3):
        self.cor= p1
        self.velocidade= p2
        self.tipo_combustivel= p3

objeto_1 = Montadora('branca',0.0,'flex')
objeto_2 = Montadora('preta',0.0,'diesel')
```

```
print('Atributos de objeto_1:',objeto_1.cor, objeto_1.tipo_combustivel, objeto_1.velocidade)
print('Atributos de objeto_2:',objeto_2.cor, objeto_2.tipo_combustivel, objeto_2.velocidade)
```

No momento em que **objeto_1** é instanciado, os argumentos `('branca',0.0,'flex')` são enviados para o método `__init__(self,p1,p2,p3)` onde os parâmetros `p1,p2,p3` recebem respectivamente `'branca',0.0,'flex'` O parâmetro **self** será associado a **objeto_1**. Assim sendo, teríamos virtualmente a seguinte situação quando da instância de **objeto_1**:

```
class Montadora:
    def __init__(self,p1,p2,p3):
        self.cor= p1
        self.velocidade= p2
        self.tipo_combustivel= p3
```

```
class Montadora:
    def __init__(objeto_1,'branca',0.0,'flex'):
        objeto_1.cor= 'branca'
        objeto_1.velocidade=  0.0
        objeto_1.tipo_combustivel= 'flex'
```

Ao instanciar o **objeto_2** teríamos o equivalente:

```
class Montadora:
    def __init__(objeto_2,'preta',0.0,'diesel'):
        objeto_2.cor= 'branca'
        objeto_2.velocidade=  0.0
        objeto_2.tipo_combustivel= 'flex'
```

Métodos de Instância

Todos os métodos que operam em uma instância específica (**objeto**) da classe e podem acessar ou modificar o estado da instância (**objeto**) são considerados métodos de instância. Você pode identificar os métodos de instância pela palavra **self** presente como primeiro argumento do método. Assim, sendo, podemos afirmar que o método **construtor** é também um método de instância. Vejamos um exemplo que cria três métodos de instância para a classe **Montadora**:

```
class Montadora:
    def __init__(self, cor, tipo_de_combustivel, velocidade=0):
        self.cor = cor
        self.tipo_de_combustivel = tipo_de_combustivel
        self.velocidade = velocidade

    def acelerar(self, aumento):
        self.velocidade += aumento
        print(f'O automóvel acelerou para {self.velocidade} km/h.')

    def frear(self,reducao):
        self.velocidade -= reducao
        print(f'O automóvel reduziu para {self.velocidade} km/h.')

    def parar(self):
        self.velocidade = 0
```

Observe que eu troquei os nomes dos parâmetros **p1**, **p2** e **p3** usados anteriormente. Note também que atribui o valor **0** à **velocidade**. Caso não definamos um valor específico para a velocidade no momento da instância do objeto, o valor zero será atribuído.

```
        print('O automóvel parou.')

objeto = Montadora("Branca","flex")
objeto.acelerar(10)
objeto.acelerar(20)
objeto.frear(5)
objeto.frear(15)
objeto.parar()
```

Os métodos **acelerar**(), **frear**() e **parar**() são métodos de instância e têm o objetivo de aplicar ações aos atributos da classe **Montadora**. Observe que os nomes dos métodos são representados por verbos. Isso é comum em POO, pois os métodos representam ações.

Variável Estática ou de Classe

Uma **variável estática** (ou **variável de classe**) é uma variável que pertence à classe em si, e não a instâncias individuais da classe. Ou seja, enquanto uma variável de instância é específica para cada objeto criado a partir da classe, uma variável estática é compartilhada entre todas as instâncias da classe. Estas variáveis compartilham o mesmo valor com todas as instâncias, ou seja, se o valor de uma variável estática for alterado em uma instância, essa mudança é refletida em todas as outras instâncias da classe. Como exemplo, vamos adicionar uma variável estática à classe **Montadora**. Suponha que queremos manter um registro de quantos carros foram criados pela montadora. Poderíamos usar uma variável estática para isso.

```
class Montadora:

    total_carros_criados = 0

    def __init__(self, cor, tipo_de_combustivel, velocidade=0):
        self.cor = cor
        self.tipo_de_combustivel = tipo_de_combustivel
        self.velocidade = velocidade

        Montadora.total_carros_criados += 1

    def acelerar(self, aumento):
        self.velocidade += aumento
        print(f'O automóvel acelerou para {self.velocidade} km/h.')

    def frear(self,reducao):
        self.velocidade -= reducao
        print(f'O automóvel reduziu para {self.velocidade} km/h.')

    def parar(self):
        self.velocidade = 0
```

Adicione apenas estas linhas ao código anterior!

Este código incrementa a variável estática cada vez que um novo carro é criado. Observe que a variável é precedida pelo nome da classe.

```
        print('O automóvel parou.')

carro1 = Montadora("vermelha", "flex")
print(f"Total: {Montadora.total_carros_criados} Total:{carro1.total_carros_criados} ")
carro2 = Montadora("branca", "diesel")
print(f"Total: {Montadora.total_carros_criados} Total:{carro2.total_carros_criados} ")
carro3 = Montadora("preta", "eletricidade")
print(f"Total: {Montadora.total_carros_criados} Total:{carro3.total_carros_criados} ")
```

Neste exemplo eu mostro que a variável estática pode ser acessada fora da classe por meio de um objeto ou precedida pelo nome da classe. Mostro também que o seu valor é compartilhado com todas as instâncias.

Método Estático

Um **método estático** é um método que pertence à classe em si, e não às instâncias da classe. Diferente de métodos de instância, que têm acesso ao estado da instância (ou seja, podem acessar e modificar atributos de instância), métodos estáticos não podem acessar ou modificar o estado da instância. Eles são usados quando uma função lógica está relacionada à classe, mas não depende de qualquer instância específica da classe. Para definir um método estático em Python, usamos o decorador **@staticmethod**.

Vamos adicionar um método estático à classe **Montadora** que exiba uma mensagem genérica de boas-vindas da montadora.

```
class Montadora:

    total_carros_criados = 0

    def __init__(self, cor, tipo_de_combustivel, velocidade=0):
        self.cor = cor
        self.tipo_de_combustivel = tipo_de_combustivel
        self.velocidade = velocidade

        Montadora.total_carros_criados += 1

    @staticmethod
    def boas_vindas():
        print("Bem-vindo à Montadora!")

carro1 = Montadora("vermelha", "flex")

Montadora.boas_vindas()
# Ou também podemos chamar o método estático por uma instância, embora isso não seja comum.
carro1.boas_vindas()
```

Chamando o método estático.

Métodos estáticos são úteis para funções que estão logicamente associadas à classe, mas não dependem de informações sobre instâncias específicas da classe. Neste exemplo, **boas_vindas** é uma boa candidata a método estático porque apenas exibe uma mensagem genérica sem acessar nenhum dado específico de instância ou de classe. Repare que o método não possui **self**.

Métodos de Classe

Um **método de classe** é um método que recebe a classe como seu primeiro argumento, em vez de uma instância da classe. Esse argumento é comumente chamado de **cls**. Métodos de classe podem acessar ou modificar o estado da classe que é compartilhado entre todas as instâncias, como variáveis de classe. Para definir um método de classe, usamos o decorador **@classmethod**. Em resumo, um método de classe pode acessar e modificar variáveis de classe, que são compartilhadas entre todas as instâncias da classe.

```
class Montadora:

    total_carros_criados = 0

    def __init__(self, cor, tipo_de_combustivel, velocidade=0):
        self.cor = cor
        self.tipo_de_combustivel = tipo_de_combustivel
        self.velocidade = velocidade

        Montadora.total_carros_criados += 1

    @classmethod
    def total_carros(cls):
        print(f"Total de carros criados pela montadora: {cls.total_carros_criados}"
)

carro1 = Montadora("vermelha", "flex")

Montadora.total_carros()
# Ou também podemos chamar o método estático por uma instância, embora isso não seja comum:
carro1.total_carros()
```

Este código incrementa a variável estática cada vez que um novo carro é criado. Observe que a variável é precedida pelo nome da classe.

Chamando o método de classe.

Métodos de classe são úteis quando precisamos de funcionalidade que diz respeito à classe em geral, em vez de apenas a instâncias individuais. Neste exemplo, **total_carros** é um método de classe porque precisa acessar a variável de classe **total_carros_criados**, que armazena o número total de carros criados pela montadora.

Encapsulamento: Métodos Getters e Setters

Encapsulamento é um dos pilares fundamentais da Programação Orientada a Objetos (POO). Ele se refere à prática de restringir o acesso direto aos atributos de um objeto e, portanto, fornecer métodos controlados para leitura e modificação desses atributos. O encapsulamento é importante porque protege o estado interno de um objeto, garantindo que ele só seja alterado de maneiras previstas pelo programador, o que ajuda a evitar erros e a manter a integridade dos dados.

Em muitas linguagens de programação, como **Java**, o encapsulamento é realizado utilizando modificadores de acesso como **private**, **protected** e **public**. Em Java, por exemplo, um atributo marcado como **private** só pode ser acessado diretamente dentro da própria classe; um atributo **protected** pode ser acessado dentro da própria classe e por subclasses; e um atributo **public** pode ser acessado por qualquer outra classe. Esses modificadores de acesso permitem um controle rigoroso sobre quais partes do código têm permissão para acessar ou modificar determinados atributos, promovendo assim uma maior segurança e organização.

Em Python, o encapsulamento é tratado de maneira um pouco diferente, pois a linguagem não possui modificadores de acesso como **private** e **protected** de forma explícita. Em vez disso, Python utiliza uma convenção de nomenclatura para indicar o nível de acesso de um atributo. Para sugerir que um atributo é "privado" e não deve ser acessado diretamente fora da classe, o nome do atributo é prefixado com um underscore (_). No entanto, isso é apenas uma convenção e não impede o acesso direto ao atributo.

Para realmente proteger um atributo em Python, usamos dois underscores (__). Quando um atributo é definido com dois underscores, o Python aplica um mecanismo chamado **name mangling**, que altera internamente o nome do atributo para incluir o nome da classe como prefixo. Por exemplo, se temos um atributo **__velocidade** na classe **Automovel**, ele será internamente renomeado para **_Automovel__velocidade**. Essa renomeação torna mais difícil o acesso acidental ao atributo fora da classe, promovendo um encapsulamento mais efetivo.

Aqui está um exemplo prático de como isso funciona:

```
class Montadora:
    def __init__(self, cor, tipo_de_combustivel, velocidade=0):
        self.cor = cor
        self.tipo_de_combustivel = tipo_de_combustivel
        self.__velocidade = velocidade

    @property
    def velocidade(self):
        return self.__velocidade

    @velocidade.setter
    def velocidade(self, nova_velocidade):
        print("Setter sendo executado!")
        if nova_velocidade < 0:
            print("A velocidade não pode ser negativa.")
```

```
        else:
            self.__velocidade = nova_velocidade

    def acelerar(self, aumento):
        self.velocidade += aumento
        print(f'O automóvel acelerou para {self.velocidade} km/h.')

    def frear(self,reducao):
        self.velocidade -= reducao
        print(f'O automóvel reduziu para {self.velocidade} km/h.')

    def parar(self):
        self.velocidade = 0
        print('O automóvel parou.')

print("Início...")
carro =Montadora(cor="vermelho", tipo_de_combustivel="gasolina", velocidade=0)
print("objeto foi instanciado..")

# Acesso indireto ao atributo velocidade via método
print("Vou acelerar...")
carro.acelerar(20) # Acelera, o setter é chamado e valida a nova velocidade
# Saída esperada: O automóvel acelerou para 20 km/h.

# Acesso direto ao atributo velocidade
print("Vou acelerar novamente...")
carro.velocidade = 50 # Acessa diretamente, o setter é chamado para validar e definir a nova
velocidade
print(f'Velocidade atual: {carro.velocidade} km/h')
# Saída esperada: Velocidade atual: 50 km/h

# Tentativa de definir uma velocidade negativa diretamente
carro.velocidade = -10 # Acessa diretamente, o setter é chamado e deve bloquear a alteração
# Saída esperada: A velocidade não pode ser negativa.

# Acesso indireto ao atributo velocidade via método
carro.frear(5) # Para o automóvel, o setter é chamado e define a velocidade para 0
# Saída esperada: O automóvel reduziu para 45 km/h.
```

Exemplos

Exemplo 1

Este exemplo modela uma classe denominada **ContaBancaria**. Esta classe terá um método construtor, métodos de instância e um atributo privado para armazenar o saldo da conta. O código incluirá métodos para depositar, sacar e consultar o saldo, com a proteção adequada para garantir que o saldo não seja modificado diretamente de fora da classe.

Solução

```
class ContaBancaria:
    def __init__(self, titular, saldo_inicial=0):
        self.titular = titular
        self.__saldo = saldo_inicial

    @property
    def saldo(self):
        """Retorna o saldo da conta."""
        return self.__saldo

    def depositar(self, valor):
        """Deposita um valor na conta, se for positivo."""
        if valor > 0:
            self.__saldo += valor
            print(f'Depósito de R${valor:.2f} realizado com sucesso.')
        else:
            print('O valor do depósito deve ser positivo.')

    def sacar(self, valor):
        """Saca um valor da conta, se houver saldo suficiente e o valor for positivo."""
        if valor > 0:
            if valor <= self.__saldo:
                self.__saldo -= valor
                print(f'Saque de R${valor:.2f} realizado com sucesso.')
            else:
                print('Saldo insuficiente para realizar o saque.')
        else:
            print('O valor do saque deve ser positivo.')

    def consultar_saldo(self):
        print(f'Saldo atual: R${self.__saldo:.2f}')
```

```
# Exemplo de uso da classe ContaBancaria
conta = ContaBancaria('João Silva', 1000)
conta.consultar_saldo()  # Exibe o saldo inicial
conta.depositar(500)     # Realiza um depósito
conta.sacar(200)         # Realiza um saque
conta.consultar_saldo()  # Exibe o saldo após as transações
conta.sacar(1500)        # Tenta sacar mais do que o saldo disponível
```

Neste exemplo, o atributo **__saldo** é privado e só pode ser acessado e modificado através dos métodos da classe, promovendo o encapsulamento e a integridade dos dados.

Os métodos têm as seguintes funções:

Método Construtor **(__init__):** Inicializa uma nova instância de **ContaBancaria** com um titular e um saldo inicial (opcional). O saldo é armazenado em um atributo privado **__saldo**.

Método Getter **(saldo):** Permite acessar o valor do saldo, mas não permite modificá-lo diretamente.

Método de Instância **depositar:** Adiciona um valor ao saldo, verificando se o valor é positivo.

Método de Instância **sacar:** Subtrai um valor do saldo, garantindo que o valor seja positivo e que haja saldo suficiente.

Método de Instância **consultar_saldo:** Exibe o saldo atual da conta.

Exemplo 2

Neste exemplo, vamos criar uma classe **Livro** para gerenciar informações sobre livros, como título, autor e número de páginas. A classe incluirá métodos para atualizar o número de páginas e exibir as informações do livro.

Solução

```
class Livro:
    def __init__(self, titulo, autor, num_paginas):
        self.titulo = titulo
        self.autor = autor
        self.__num_paginas = num_paginas

    @property
    def num_paginas(self):
        """Retorna o número de páginas do livro."""
        return self.__num_paginas
```

```
    @num_paginas.setter
    def num_paginas(self, novo_num_paginas):
        """Atualiza o número de páginas se o novo valor for positivo."""
        if novo_num_paginas > 0:
            self.__num_paginas = novo_num_paginas
            print(f'Número de páginas atualizado para {novo_num_paginas}.')
        else:
            print('O número de páginas deve ser um valor positivo.')

    def exibir_informacoes(self):
        """Exibe as informações do livro."""
        print(f'Título: {self.titulo}')
        print(f'Autor: {self.autor}')
        print(f'Número de Páginas: {self.__num_paginas}')

# Exemplo de uso da classe Livro
livro = Livro('1984', 'George Orwell', 328)
livro.exibir_informacoes()
livro.num_paginas = 350 # Atualiza o número de páginas
livro.exibir_informacoes()
```

Exercícios

Exercício 1: Sistema de Gerenciamento de Funcionários

Neste exercício você deve criar uma classe **Funcionario** para gerenciar informações de funcionários, incluindo nome, cargo e salário. A classe deve incluir métodos para atualizar o salário e exibir as informações do funcionário.