PROGRAMAÇÃO ORIENTADA A OBJETOS EM PYTHON

material produzido pelo prof. Guilherme Derenievicz

adaptado pelo prof. Jônata Tyska Carvalho

Departamento de Informática e Estatística - UFSC

---

3 - Herança e Classes Abstratas

Considere a classe `Carro` do **Notebook 2-Encapsulamento** e a classe `Bicicleta` abaixo.

classe `Carro` :

> **Atributos:**
>
> - modelo: **str**
> - cor: **str**
> - placa: **str**
> - dono: **Pessoa**
> - velocidade: **int**
> - sujo: **bool**
>
> **Métodos:**
>
> - *getters* de todos os atributos
> - *setters* dos atributos cor, placa e dono
> - lavar()
> - buzinar()
> - acelerar()
> - frear()

classe `Bicicleta` :

> **Atributos:**
>
> - modelo: **str**

> - cor: **str**
> - dono: **Pessoa**
> - velocidade: **int**
> - sujo: **bool**
> - marchas: **int**
> - amortecedor: **bool**
>
> **Métodos:**
>
> - *getters* de todos os atributos
> - *setters* dos atributos cor, dono, marchas e amortecedor
> - lavar()
> - pedalar()
> - frear()

Pergunta: Por que as classes `Carro` e `Bicicleta` possuem atributos e métodos em comum?

Resposta: Porque ambas pertencem a uma mesma **superclasse** (classe mais geral). Nesse caso, `Veiculo`.

classe `Veiculo`:

> **Atributos:**
>
> - modelo: **str**
> - cor: **str**
> - dono: **Pessoa**
> - velocidade: **int**
> - esta_sujo: **bool**
>
> **Métodos:**
>
> - lavar()
> - frear()

`Carro` e `Bicicleta` são **tipos de** `Veiculo`, isto é, são classes específicas ou **subclasses** de `Veiculo`. Uma consequência direta desta característica é que `Carro` e `Bicicleta` **herdam** todos os atributos e métodos da **superclasse** `Veiculo`. Naturalmente, as subclasses podem incluir atributos e métodos adicionais, que são específicos daquilo que estão representando. Assim, também dizemos que `Carro` e `Bicicleta` **estendem** a classe `Veiculo`.

Em Python, uma subclasse herda **automaticamente** os métodos da superclasse, mas para herdar os atributos é necessário chamar explicitamente o construtor da superclasse através do método especial `super()`. Também é possível chamar explicitamente os métodos da superclasse utilizando o `super()`.

In [15]:
```
class Pessoa:
    def __init__(self, nome: str, cpf: str, dinheiro: float):
```

```
        self.__nome = nome
        self.__cpf = cpf
        self.__dinheiro = dinheiro

    @property
    def nome(self):
        return self.__nome

    @property
    def cpf(self):
        return self.__cpf

    @property
    def dinheiro(self):
        return self.__dinheiro

    @dinheiro.setter
    def dinheiro(self, valor: float):
        self.__dinheiro = valor


class Veiculo:
    staticVar = "abc"

    def __init__(self, modelo: str, cor: str, dono: Pessoa):
        self.__modelo = modelo
        self.__cor = cor
        self.__dono = dono
        self.__velocidade = 0
        self.__sujo = True

    @property
    def modelo(self):
        return self.__modelo

    @property
    def cor(self):
        return self.__cor

    @cor.setter
    def cor(self, nova_cor: str):
        self.__cor = nova_cor

    @property
    def dono(self):
        return self.__dono

    @dono.setter
    def dono(self, novo_dono: Pessoa):
        self.__dono = novo_dono

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

    @property
    def sujo(self):
        return self.__sujo

    def lavar(self):
```

```
        total = self.dono.dinheiro
        if total >= 30:
            self.dono.dinheiro = total - 30
            self.__sujo = False

    def frear(self):
        self.__velocidade -= 10
        if self.__velocidade < 0:
            self.__velocidade = 0

class Carro(Veiculo):
    def __init__(self, modelo: str, cor: str, placa: str, dono: Pessoa):
        super().__init__(modelo, cor, dono)
        self.__placa = placa

    @property
    def placa(self):
        return self.__placa

    @placa.setter
    def placa(self, nova_placa: str):
        self.__placa = nova_placa

    def buzinar(self):
        print(self.__modelo, 'buzinou!')

    def acelerar(self, valor: int):
        self.__velocidade += valor

class Bicicleta(Veiculo):
    def __init__(self, modelo: str, cor: str, marchas: int, amortecedor: bool, d
        super().__init__(modelo, cor, dono)
        self.__marchas = marchas
        self.__amortecedor = amortecedor

    @property
    def marchas(self):
        return self.__marchas

    @marchas.setter
    def marchas(self, novas_marchas: int):
        self.__marchas = novas_marchas

    @property
    def amortecedor(self):
        return self.__amortecedor

    @amortecedor.setter
    def amortecedor(self, novo_amortecedor: bool):
        self.__amortecedor = novo_amortecedor

    def pedalar(self, valor: int):
        self.__velocidade += valor

joao = Pessoa('Joao Silva', '123.456.789-0', 50)
carro1 = Carro('Gol', 'Vermelho', 'ABC-1234', joao)
bicicleta1 = Bicicleta('Caloi 10', 'Vermelha', 12, False, joao)

print(joao.nome, ', Dinheiro = R$', joao.dinheiro)
print(carro1.modelo, carro1.placa, ', Velocidade =', carro1.velocidade, 'km/h')
```

```
print(bicicleta1.modelo, ',', bicicleta1.marchas, 'marchas, Velocidade =', bicic

carro1.acelerar(80)
bicicleta1.pedalar(5)
carro1.lavar()
bicicleta1.lavar()

print(joao.nome, ', Dinheiro = R$', joao.dinheiro)
print(carro1.modelo, carro1.placa, ', Velocidade =', carro1.velocidade, 'km/h')
print(bicicleta1.modelo, ',', bicicleta1.marchas, 'marchas, Velocidade =', bicic
```

```
Joao Silva , Dinheiro = R$ 50
Gol ABC-1234 , Velocidade = 0 km/h
Caloi 10 , 12 marchas, Velocidade = 0 km/h
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-15-5c17c6f1d47b> in <module>
    123 print(bicicleta1.modelo, ',', bicicleta1.marchas, 'marchas, Velocidade
=', bicicleta1.velocidade, 'km/h')
    124
--> 125 carro1.acelerar(80)
    126 bicicleta1.pedalar(5)
    127 carro1.lavar()

<ipython-input-15-5c17c6f1d47b> in acelerar(self, valor)
     88
     89     def acelerar(self, valor: int):
---> 90         self.__velocidade += valor
     91
     92 class Bicicleta(Veiculo):

AttributeError: 'Carro' object has no attribute '_Carro__velocidade'
```

Por que esse código não funciona? `__velocidade` é um atributo privado da classe `Veiculo` e não possui método *setter*, portanto, não pode ser alterado por nenhuma outra classe (mesmo que seja sua subclasse!). Um alternativa é definir um método *setter* para esse atributo na classe `Veiculo`. Assim, as subclasses `Carro` e `Bicicleta` também devem ser alteradas para que façam acesso a este atributo através do *setter*, conforme abaixo (note a ausência dos *underlines* na frente do atributo `velocidade`).

```
class Veiculo:
    # ...

    @velocidade.setter
    def velocidade(self, valor: int):
        self.__velocidade = valor

    # ...


class Carro(Veiculo):
    # ...

    def acelerar(self, valor: int):
        self.velocidade += valor

    # ...
```

```
class Bicicleta(Veiculo):
    # ...

    def pedalar(self, valor: int):
        self.velocidade += valor

    # ...
```

Sobreposição (*override*)

Note que os métodos `frear()` e `lavar()` são os mesmos para bicicleta e carro, pois ambas as classes herdam esses métodos da superclasse `Veiculo`. Isso pode não ser uma representação adequada para os casos em que tais métodos devam apresentar comportamentos específicos dependendo do tipo de veículo. Nesse caso, os métodos podem ser **sobrepostos** por novas implementações nas respectivas subclasses. Assim, há duas alternativas:

- A classe `Veiculo` contém implementações *default* dos métodos `frear()` e `lavar()`. Se uma subclasse não implementa algum desses métodos, herdará a implementação *default*; mas se ela implementa, então o método será sobreposto com a nova implementação.
- A classe `Veiculo` contém apenas o protótipo dos métodos `frear()` e `lavar()`. Assim, as subclasses devem implementar ambos os métodos.

Exemplo: Considere as seguintes modificações no código dos veículos. Aqui consideramos que a superclasse `Veiculo` contém uma implementação *default* do método `lavar()`, mas apresenta apenas o cabeçalho do método `frear()`, deixando a sua implementação para as subclasses. A subclasse `Carro` implementa apenas o método `frear()`, herdando a implementação *default* do método `lavar` da superclasse `Veiculo`. Por outro lado, a subclasse `Bicicleta` contém implementação de ambos os métodos, sobrepondo a implementação *default* do método `lavar()` para todas as suas instâncias.

```
class Veiculo:
    # ...

    def lavar(self):
        total = self.dono.dinheiro
        if total >= 30:
            self.dono.dinheiro = total - 30
            self.__sujo = False

    def frear(self):
        pass

    # ...
```

```
class Carro(Veiculo):
    # ...

    def frear(self):
        self.velocidade -= 10
        if self.velocidade < 0:
            self.velocidade = 0


class Bicicleta(Veiculo):
    # ...

    def lavar(self):
        total = self.dono.dinheiro
        if total >= 10:
            self.dono.dinheiro = total - 10
            self.sujo = False

    def frear(self):
        self.velocidade -= 1
        if self.velocidade < 0:
            self.velocidade = 0
```

Obs.: Aqui há um detalhe importante (quase uma pegadinha!): note que no método `lavar()` da classe `Veiculo` o atributo **privado** `__sujo` é alterado, enquanto no método `lavar()` da classe `Bicicleta` o atributo acessado é `sujo`, sem *underlines*. Isso acontece porque a classe `Bicicleta` não consegue acessar diretamente o atributo privado da classe `Veiculo` e deve fazer a alteração do atributo através de um método *setter*. Portanto, para que esse código funcione, é necessário incluir na classe `Veiculo` também um método *setter* para o atributo `sujo`.

Pergunta: o que acontece se, ao invés de criarmos o método *setter* como dito acima, alterarmos o código do método `lavar()` da classe `Bicicleta` para alterar diretamente o atributo privado: `self.__sujo = False`? O código apresentará um erro? Funcionará corretamente? O que acontece? **Teste isso!**

Classe Abstrata

Uma classe é dita **abstrata** se ela não é instanciada no programa. No nosso exemplo, `Veiculo` pode ser considerada uma classe abstrata, isto é, não há objetos que sejam veículos e não sejam bicicletas ou carros (nesse caso, esses objetos são instâncias das subclasses `Carro` e `Bicicleta`, e não da classe `Veiculo`). Uma classe abstrata pode conter **métodos abstratos**, que são protótipos de métodos que devem ser implemetnados pelas subclasses.

Em Python, existe uma maneira de indicar que uma classe é abstrata e que seus métodos são abstratos. Assim, a tentativa de instanciação dessa classe, bem como a não definição de seus métodos, irá gerar um erro.

> Uma classe é abstrata se ela é uma subclasse da classe `ABC`, do módulo `abc`.

> Um método é abstrato se ele possui a diretiva `@abstractmethod`.

Veja o exemplo abaixo. `FormaGeometrica` é uma classe abstrata que nunca é instanciada, mas possui subclasses que, estas sim, são instanciadas. Podemos interpretar essa representação da seguinte maneira: na geometria existem diversos "objetos" tais como *quadrados*, *retângulos* e *círculos*, e todos eles são *formas geométricas*. No entanto, o conceito de *forma geométrica* é abstrato: não basta definirmos que um objeto é uma forma geométrica, precisamos também definir **qual tipo** de forma geométrica ele é, se um quadrado, um retângulo ou um círculo.

A classe `FormaGeometrica` possui um único atributo privado `area`, pois consideramos que qualquer forma geométrica possui esse atributo. Essa classe possui também um método abstrato `calcula_area()`, que não possui uma implementação *default*, visto que a maneira de calcular a área de uma forma geométrica depende de qual tipo de forma geométrica estamos nos referindo. Enquanto a superclasse `ABC` indica que a classe `FormaGeometrica` é abstrata, a diretiva `@abstractmethod` indica que as subclasses de `FormaGeometrica` devem implementar o método `calcula_area()`.

Observe a utilidade de classes abstratas: criamos uma lista de objetos das subclasses `Quadrado`, `Retangulo` e `Circulo`, cujas áreas são calculadas e mostradas na tela. Sabemos que o método `calcula_area()` pode ser chamado para cada um desses objetos porque a superclasse abstrata `FormaGeometrica` define a existência deste método para cada subclasse. Se essa superclasse não existisse, teríamos que olhar cada uma das classes `Quadrado`, `Retangulo` e `Circulo` para nos assegurar que este método de fato existe e possuí a mesma assinatura.

In [6]:
```python
from abc import ABC, abstractmethod
import math

class FormaGeometrica(ABC):
    def __init__(self):
        self.__area = 0

    @property
    def area(self):
        return self.__area

    @area.setter
    def area(self, area: float):
        self.__area = area

    @abstractmethod
    def calcula_area(self):
        pass


class Quadrado(FormaGeometrica):
    def __init__(self, lado: float):
        super().__init__()
        self.__lado = lado

    def calcula_area(self):
```

```
        self.area = self.__lado * self.__lado


class Retangulo(FormaGeometrica):
    def __init__(self, base: float, altura: float):
        super().__init__()
        self.__base = base
        self.__altura = altura

    def calcula_area(self):
        self.area = self.__base * self.__altura


class Circulo(FormaGeometrica):
    def __init__(self, raio: float):
        super().__init__()
        self.__raio = raio

    def calcula_area(self):
        self.area = math.pi * self.__raio**2


figuras = [Quadrado(2), Retangulo(2,3), Circulo(2), Circulo(3), Quadrado(3), Qua

for fig in figuras:
    fig.calcula_area()
    print(fig.area)
```

```
4
6
12.566370614359172
28.274333882308138
9
4
3
```

Exercício

Aplique os conceitos de classe abstrata e método abstrato no exemplo dos veículos.

In [17]:
```
#escreva seu código aqui
```

Referências

Este material foi desenvolvido com base nas Notas de Aulas da disciplina *Desenvolvimento de Sistemas Orientados a Objetos I* do curso de Sistemas de Informação do INE-UFSC, de autoria do prof. Jean Carlo Rossa Hauck, e no material *Aulas de Introdução à Computação em Python*, do Detartamento de Ciência da Computação do IME-USP, disponível em https://panda.ime.usp.br/aulasPython/static/aulasPython/index.html

In [ ]: