PROGRAMAÇÃO ORIENTADA A OBJETOS EM PYTHON

material produzido pelo prof. Guilherme Derenievicz

adaptado pelo prof. Jônata Tyska Carvalho

Departamento de Informática e Estatística - UFSC

---

2 - Encapsulamento

Atributos privados

Considere o **Exemplo dos carros** do **Notebook 1-Objetos e Classes**:

Classe `Carro`

> **Atributos:**
>
> - modelo
> - cor
> - placa
> - velocidade
>
> **Métodos:**
>
> - buzinar()
> - acelerar()
> - frear()

In [2]:
```
class Carro:
    def __init__(self, modelo: str, cor: str, placa: str):
        self.modelo = modelo
        self.cor = cor
        self.placa = placa
        self.velocidade = 0

    def buzinar(self):
        print('carro', self.modelo, 'buzinou')
```

```
    def acelerar(self, valor: int):
        self.velocidade += valor
        if self.velocidade > 100:
            return 'cuidado!'
        else:
            return 'velocidade aceitável.'

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


carro1 = Carro('Gol', 'Vermelho', 'ABC-1234')
carro2 = Carro('Fox', 'Prata', 'XYZ-1234')

carro1.buzinar()
print(carro1.modelo)
print(carro2.acelerar(80))
print(carro2.velocidade, 'km/h')
```

```
carro Gol buzinou
Gol
velocidade aceitável.
80 km/h
```

No exemplo acima estamos acessando os atributos `modelo` e `velocidade` do objeto `carro1` e os métodos `buzinar()` e `acelerar()`. Acessar diretamente os atributos de um objeto **pode ser perigoso**. É preferível que se acesse apenas os **métodos**, como `acelerar()` e `buzinar()`. Imagine que a seguinte atribuição ocorra no seu programa principal:

```
carro1.velocidade = -10
```

Atribuir um valor negativo à velocidade certamente não deveria ser uma operação válida. Note que o método `frear()` da classe `Carro` faz a verificação adequada de velocidade não negativa. Na verdade, pode-se imaginar que o atributo `velocidade` não deve ser alterado por qualquer forma que não seja através dos métodos `acelerar()` e `frear()`. Alterar atributos através de métodos transfere a responsabilidade de alteração ao próprio objeto, o que é uma boa prática de programação no paradigma OO. Assim, o programa principal não altera diretamente os atributos de um objeto, mas "pede" para que o objeto altere aquele atributo. Com isso, o objeto é capaz de verificar em si mesmo todas as consequências e condições desta alteração.

Com isso, entramos em um outro aspecto do paradigma OO: **atributos privados**. Um atributo que é privado de uma classe só pode ser alterado através dos métodos daquela classe. No exemplo anterior, todos os atributos são públicos, possibilitando que atribuições como `carro1.velocidade = -10` ocorram. Em Python, para tornar um atributo privado basta nomeá-lo com prefixo `__` (dois *underlines*). Assim, nossa classe ficaria com os atributos:

> - __modelo: **str**
> - __cor: **str**

> - __placa: **str**
> - __velocidade: **int**

O código completo da classe e programa principal fica:

In [4]:
```
class Carro:
    def __init__(self, modelo: str, cor: str, placa: str):
        self.__modelo = modelo
        self.__cor = cor
        self.__placa = placa
        self.__velocidade = 0

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

    def acelerar(self, valor: int):
        self.__velocidade += valor
        if self.__velocidade > 100:
            return 'cuidado!'
        else:
            return 'velocidade aceitável.'

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


carro1 = Carro('Gol', 'Vermelho', 'ABC-1234')
carro2 = Carro('Fox', 'Prata', 'XYZ-1234')

carro1.buzinar()
print(carro2.acelerar(80))
print(carro1.__modelo)
print(carro2.__velocidade, 'km/h')
```

```
carro Gol buzinou
velocidade aceitável.
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-4-fb7ec1ad7f3f> in <module>
     27 carro1.buzinar()
     28 print(carro2.acelerar(80))
---> 29 print(carro1.__modelo)
     30 print(carro2.__velocidade, 'km/h')

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

Perceba como a execução do código acima gera um erro ao tentar acessar atributos privados dos objetos. Lembrando que o Python simula a existência de atributos privados através do que chamamos de *Name Mangling*, conforme descrito na seção 9.6 private variables da documentação da linguagem https://docs.python.org/3/tutorial/classes.html.

Getters e Setters

Como você pode perceber, executar o programa acima gerou um erro de atributo (*__AttributeError__*)! *Isso ocorre porque o programa principal está tentando acessar os atributos privados** `__modelo` e `__velocidade` da classe `Carro`. Assim, para acessar valores de atributos privados **métodos *getters*** deveriam ser definidos na classe `Carro`. Um método *getter* é responsável por retornar o valor de um atributo privado da classe. Em Python, é possível definir tais métodos manualmente ou usando *decorators* específicos (se quiser saber mais sobre *decorators* em Python veja https://python101.pythonlibrary.org/chapter25_decorators.html). O exemplo exemplo abaixo mostra a definição manual de um método get para a propriedade modelo.

```
def get_modelo(self):
    return self.__modelo
```

Assim, para acessar o valor do atributo modelo bastaria utilizar o método `get_modelo()`.

Como dito anteriormente, é possível utilizar *decorators* para a definição dos métodos *getters*. Desta forma os métodos são definidos com o mesmo nome do atributo que este deve retornar, com a diretiva `@property` sobre o cabeçalho do método, como mostra o exemplo abaixo.

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

Atenção! Em Python, para chamar um método *getter* escrito desta forma não devem ser utilizados os parênteses! Parece uma pegadinha, mas escrita do código torna-se mais flutente. Na prática, basta imaginar que aquele atributo é público e acessá-lo sem o uso de `__`. Ao final deste aula veremos uma explicação do porquê desta convenção de *getters* e *setters* utilizada em Python.

```
print(carro1.modelo)
```

Já sabemos como acessar o valor de atributos privados e que podemos alterar esses valores através de métodos específicos da classe, como `acelerar()` e `frear()`. Repare que, por enquanto, não temos métodos específicos para alterar os atributos `cor` e `placa` (logicamente, o modelo de um carro não pode ser alterado). Poderíamos criar métodos específicos para isso, os quais avaliariam as condições e consequências necessárias para tais alterações. Imagine, no entanto, que possamos alterar a cor e a placa de um carro incondicionalmente e sem outras consequências para o objeto. Se os atributos fossem públicos, poderíamos fazer:

```
carro1.placa = 'ABC-4321'
carro1.cor = 'Preto'
```

Mas como os atributos são privados, é necessário definir métodos *setters* para ambos os atributos. De forma similar à definição dos *getters* usando *decorators*, um método *setter* deve ser nomeado com o nome do atributo a ser modificado, juntamente com a diretiva `@<atributo>.setter` sobre o cabeçalho do método, conforme exemplo abaixo:

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

A chamada de um método *setter* da forma `carro1.placa = 'ABC-4321'` pode ser interpretada (apenas para fins de entendimento) como `carro1.placa('ABC-4321')`, mas a notação de atribuição torna a codificação mais fluente.

Exercício: crie os métodos *getters* para todos os atributos da classe `Carro` e os métodos *setters* para os atributos `placa` e `cor`, e teste a alteração destes atributos no programa principal.

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

A classe `Carro` com os atributos privados e com os métodos adequados, incluindo *getters* e *setters* está perfeitamente **encapsulada**. Isto quer dizer que a classe tem controle sobre seus atributos, ao mesmo tempo que fornece um conjunto de métodos adequados para utilizá-la. Além da segurança, a prática do encapsulamento possibilita a abstração de objetos na qual apenas os métodos são enxergados pelo programador que fará uso de seus objetos. Isso favorece a reusabilidade e a manutenibilidade do código.

Imagine o caso de um carro real: o motorista tem acesso ao acelerador e ao freio e pode acelar ou frear o carro sempre que quiser. O motorista, porém, não precisa saber os detalhes (abstração) do que acontece no sistema mecânico do carro quando pressiona o acelerador, basta saber o que o método de acelerar faz e como deve ser utilizado para obter o resultado esperado. Nessa comparação, o sistema mecânico de aceleramento do carro está perfeito encapsulado: possui atributos privados que o motorista não conhece e não tem acesso, mas que são alterados conforme o uso dos métodos disponíveis ao usuário, como `frear()` e `acelerar()`.

Um Exemplo Mais Completo

Neste versão, os atributos `dono` e `sujo` são adicionado à classe `Carro`, além do método `lavar()`. O atributo `dono` deve ser uma instância da classe `Pessoa`, também definida no código. O atributo `sujo` é do tipo `bool` e representa se o carro está sujo (`True`) ou limpo (`False`). Este atributo não possui um método *setter*, pois só pode ser alterado pelo método `lavar()`. O que aconteceria se o atributo `sujo` fosse público e pudesse ser setado para `False` por atribuição direta?

In [14]:
```
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 Carro:
    def __init__(self, modelo: str, cor: str, placa: str, dono: Pessoa):
        self.__modelo = modelo
        self.__cor = cor
        self.__placa = placa
        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 placa(self):
        return self.__placa

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

    @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
        else:
```

```
            print(f"Desculpe, não será possível lavar o {self.modelo} de placas

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

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

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

joao = Pessoa('Joao Silva', '123.456.789-0', 50)
carro1 = Carro('Gol', 'Vermelho', 'ABC-1234', joao)
carro2 = Carro('Fox', 'Preto', 'XYZ-9876', joao)

print(carro1.modelo, ', Sujo =', carro1.sujo)
print(carro2.modelo, ', Sujo =', carro2.sujo)
print(joao.nome, ', Dinheiro = R$', joao.dinheiro)

carro1.lavar()
carro2.lavar()

print(carro1.modelo, ', Sujo =', carro1.sujo)
print(carro2.modelo, ', Sujo =', carro2.sujo)
print(joao.nome, ', Dinheiro = R$', joao.dinheiro)
```

```
Gol , Sujo = True
Fox , Sujo = True
Joao Silva , Dinheiro = R$ 50
Desculpe, não será possível lavar o Fox de placas XYZ-9876. A lavagem custa 30
reais e o dono do carro possui apenas 20
Gol , Sujo = False
Fox , Sujo = True
Joao Silva , Dinheiro = R$ 20
```

Note o comando `total = self.dono.dinheiro` : o *getter* `dono` é chamado e retorna uma instância da classe `Pessoa` , da qual é chamado o *getter* `dinheiro` . Uma forma alternativa seria fazer o comando em duas linhas:

```
class Carro:
    #...
    def lavar(self):
        dono = self.dono
        total = dono.dinheiro
        #...
```

Troca de Mensagens

Um outro aspecto de OO que fica claro no exemplo anterior é que neste paradigma o problema é resolvido através da **troca de mensagens** entre objetos. O objeto `carro1` passa mensagens ao objeto `joao` "pedindo" para acessar seu atributo `dinheiro` através do *getter* e do *setter*.

Sobre variáveis privadas

Na verdade em Python a nomenclatura de atributos inicada com `__` apenas dificulta seu acesso, mas não o torna impossível. Na prática o Python troca o nome do atributo `__atributo` para `_Classe__atributo` (*name mangling*). Por exemplo, você pode testar que é possível acessar diretamente o atributo `__modelo` do `carro1` no código acima fazendo `carro1._Carro__modelo`, mesmo que o método *getter* deste atributo não esteja definido.

Exercício

Considere a seguinte solução do exercício das formas geométricas proposto na **Notebook 1-Objetos e Classes**. Note que se fosse necessário obter a área de uma figura várias vezes no decorrer do programa, não seria necessário recalcular a área toda vez, bastaria criar um atributo `__area` que seria inicializado com o valor correto no momento da construção do objeto. Implemente esta modificação e os conceitos de encapsulamento no seu programa, respondendo às seguintes questões:

1. Por quê o atributo `__area` deve ser privado?
2. Quais outros atributos devem ser privados?
3. O método `area()` será usado apenas para iniciar o atributo `__area` e, portanto, pode ser um **método privado**. Você sabe como fazer isso?
4. Acrescente ao seu programa as funcionalidades de **perímetro**, **diâmetro** e **diagonal**.

In [ ]:
```python
import math

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

    def area(self):
        return self.lado * self.lado


class Retangulo:
    def __init__(self, base: float, altura: float):
        self.base = base
        self.altura = altura

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


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

    def area(self):
        return math.pi * self.raio**2
```

```
def main():
    tipo = input('Digite a figura:')
    fig = None

    if tipo == 'quadrado':
        lado = float(input('Digite o lado do quadrado: '))
        fig = Quadrado(lado)
    elif tipo == 'retangulo':
        base = float(input('Digite a base do retangulo: '))
        altura = float(input('Digite a altura do retangulo: '))
        fig = Retangulo(base, altura)
    elif tipo == 'circulo':
        raio = float(input('Digite o raio do circulo: '))
        fig = Circulo(raio)

    a = fig.area()
    print('Area do', tipo, ':', a)

main()
```

Outra forma de implementar *setters* e *getters*

Se você conhece alguma outra linguagem orientada a objetos, certamente já viu os conceitos de *getters* e *setters* como métodos que permitem o acesso a atributos privados de uma classe. Em geral, este métodos pode ser escritos da forma `get_atributo()` e `set_atributo()` e são utilizados normalmente como se fossem quaisquer outros métodos, um pouco diferente das noções de *getter* e *setter* de Python apresentadas aqui.

Considere o seguinte problema:

- Na sua empresa você criou a classe `NotaFiscal` de um sistema de informação;
- Na classe `NotaFiscal` há um atributo **público** chamado `valor_de_venda`, o qual deve ser setado no momento da emissão da nota fiscal;
- Outros programadores da sua empresa escreveram outras classes e funções do sistema, utilizando-se da classe `NotaFiscal` que você criou (e, consequentemente, do atributo `valor_de_venda`), conforme o exemplo abaixo:

In [ ]:
```
class NotaFiscal:
    def __init__(self):
        self.valor_de_venda = 0

    def emite_nota_fiscal(self):
        print('Emitindo nota fiscal no valor de R$', self.valor_de_venda)


def main():

    nf = NotaFiscal()
    nf.valor_de_venda = 100
    nf.emite_nota_fiscal()

main()
```

- Após o sistema estar quase pronto, você percebe que é melhor fazer uma alteração na classe `NotaFiscal` de modo a verificar se o valor de venda atribuído a uma nota é positivo;
- Deste modo, você transforma o atributo `valor_de_venda` em privado, criando um método *setter* especial para ele:

In [ ]:
```
class NotaFiscal:
    def __init__(self):
        self.__valor_de_venda = 0

    def set_valor_de_venda(self, valor: float):
        if valor >= 0:
            self.__valor_de_venda = valor

    def emite_nota_fiscal(self):
        print('Emitindo nota fiscal no valor de R$', self.__valor_de_venda)


def main():

    nf = NotaFiscal()
    nf.valor_de_venda = 100
    nf.emite_nota_fiscal()

main()
```

- Neste exemplo, a função `main` não funciona mais como deveria, pois não consegue mais setar o valor de venda da nota fiscal. Como agora este atributo tornou-se privado, o `main` deve ser alterado para usar o método `set_valor_de_venda()`;
- Agora imagine que essa alteração terá que ser feita em **todo** o código do sistema!! Todos os programadores terão que reescrever seus respectivos códigos!

Python fornece uma solução para esse problema com formas alternativas de declarar os métodos *getters* e *setters*, de maneira que o acesso continua idêntico à forma de se acessar um atributo público. De bandeja, ainda ganhamos uma forma mais limpa de

acessar atributos privados sem a necessidade de ficar usando métodos *getters* e *setters* (embora, internamente, eles ainda existam!)

In [ ]:
```
class NotaFiscal:
    def __init__(self):
        self.__valor_de_venda = 0

    @property
    def valor_de_venda(self):
        return self.__valor_de_venda

    @valor_de_venda.setter
    def valor_de_venda(self, valor: float):
        if valor >= 0:
            self.__valor_de_venda = valor

    def emite_nota_fiscal(self):
        print('Emitindo nota fiscal no valor de R$', self.__valor_de_venda)


def main():

    nf = NotaFiscal()
    nf.valor_de_venda = 100
    nf.emite_nota_fiscal()

main()
```

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 [ ]:
```
```