



POO II

Revisão da Linguagem Python

Prof. Romuere Silva

Modificador de Acesso

- Em Python utilizamos “__” para modificar o acesso do atributo. Em Java, por exemplo, utilizamos **private**.

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

```
>>> pessoa = Pessoa(20)  
>>> pessoa.idade  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
AttributeError: 'Pessoa' object has no attribute 'idade'
```

- Em Python nenhum atributo é verdadeiramente privado... Podemos acessar o atributo idade da seguinte forma:

```
>>> p._Pessoa__idade
```

- Entretanto, isso é considerado uma má prática!

Modificador de Acesso

- ▶ Entretanto existe uma convenção em Python que todo atributo com '_' é um atributo privado e não deve ser acessado fora da classe.

```
def __init__(self, idade):  
    self._idade = idade
```

Get e Set

- Visto que nossos atributos são privados e só devemos acessar os mesmos dentro da classe, é necessário criar modos de acessar esses atributos;
- A maioria das linguagens de programação utilizam *getters* e *setters*, para acessar e modificar atributos, respectivamente.

```
class Conta:

    def __init__(self, titular, saldo):
        self._titular = titular
        self._saldo = saldo

    def get_saldo(self):
        return self._saldo

    def set_saldo(self, saldo):
        self._saldo = saldo

    def get_titular(self):
        return self._titular

    def set_titular(self, titular):
        self._titular = titular
```

... Outra solução em Python

- Um método que é usado para obter um valor (o *getter*) é decorado com `@property`;
 - colocamos essa linha diretamente acima da declaração do método que recebe o nome do próprio atributo.
- O método que tem que funcionar como *setter* é decorado com `@saldo.setter`.
- Podemos chamar esses métodos sem os parênteses, como se fossem atributos públicos;
- É uma forma mais elegante de encapsular nossos atributos.

```
class Conta:
```

```
    def __init__(self, saldo=0.0):  
        self._saldo = saldo
```

```
    @property  
    def saldo(self):  
        return self._saldo
```

```
    @saldo.setter  
    def saldo(self, saldo):  
        if(self._saldo < 0):  
            print("saldo não pode ser negativo")  
        else:  
            self._saldo = saldo
```

```
>>> conta = Conta(1000.0)  
>>> conta.saldo = -300.0  
"saldo não pode ser negativo"
```



Atividade

- Crie o controle de acesso para os atributos da conta.
- **Obs: o atributo saldo não deve incluído, pois o mesmo deve ser atualizado através dos métodos de sacar e depositar!**

Atributos da Classe

- Para criar um atributo que controle o total de contas criadas é necessário que o “total de contas” seja um atributo da classe Conta, e não de um objeto em particular.

```
class Conta:

    total_contas = 0

    def __init__(self, saldo):
        self._saldo = saldo
        Conta.total_contas += 1

>>> c1 = Conta(100.0)
>>> c1.total_contas
1
>>> c2 = Conta(200.0)
>>> c2.total_contas
2
>>> Conta.total_contas
2
```

Atributos da Classe

- Para controlar o acesso ao atributo, vamos adicionar o “_”

```
class Conta:  
    _total_contas = 0
```

- Entretanto....

```
>>> Conta.total_contas  
Traceback (most recent call last):  
  File <stdin>, line 23, in <module>  
    Conta.total_contas  
AttributeError: 'Conta' object has no attribute 'total_contas'
```


Criando um get...

- Funciona quando chamamos este método por um instância, mas quando fazemos `Conta.get_total_contas()` o interpretador reclama pois não passamos a instância:

```
>>> c1 = Conta(100.0)
>>> c1.get_total_contas()
1
>>> c2 = Conta(200.0)
>>> c2.get_total_contas()
2
>>> Conta.get_total_contas()
Traceback (most recent call last):
  File <stdin>, line 17, in <module>
    Conta.get_total_contas()
```

```
TypeError: get_total_contas() missing 1 required positional argument: 'self'
```

```
class Conta:
```

```
    _total_contas = 0
```

```
    # __init__ e outros métodos
```

```
    def get_total_contas(self):
        return Conta._total_contas
```

Continuando...

```
>>> c1 = Conta(100.0)
>>> c2 = Conta(200.0)
>>> Conta.get_total_contas(c1)
2
```

- O código acima funciona, mas não é a maneira correta de se fazer.
- E se tirar o self do método?

```
def get_total_contas():
    return Conta._total_contas
```

```
>>> c1 = Conta(100.0)
>>> c1.get_total_contas()
Traceback (most recent call last):
  File <stdin> in <module>
    c1.get_total_contas()
TypeError: get_total_contas() takes 0 positional arguments but 1 was given
```

Solução: @staticmethod

```
@staticmethod  
def get_total_contas():  
    return Conta._total_contas
```

```
>>> c1 = Conta(100.0)
```

```
>>> c1.get_total_contas()
```

```
1
```

```
>>> c2 = Conta(200.0)
```

```
>>> c2.get_total_contas()
```

```
2
```

```
>>> Conta.get_total_contas()
```

```
2
```



Atividade

- Crie o contador de contas na sua classe Conta.
- 

Slots

- Mas como Python é uma linguagem dinâmica, nada impede que usuários de nossa classe Conta criem atributos em tempo de execução, fazendo, por exemplo:

```
>>> conta.nome = "minha conta"
```

- Esse código não acusa erro e nossa conta fica aberta a modificações ferindo a segurança da classe;
- Para evitar isso podemos utilizar uma variável embutida no Python chamada `__slots__` que pode guardar uma lista de atributos da classe definidos por nos:

```
class Conta:
```

```
    __slots__ = ['_numero', '_titular', '_saldo', '_limite']
```

```
    def __init__(self, numero, titular, saldo, limite=1000.0):  
        # inicialização dos atributos
```



Atividade

- Crie Slots em sua classe Conta;
- 

Atividade

- Similar à classe Conta criada em sala de aula, crie uma classe Fotografia.
- A classe Fotografia terá os seguintes atributos:
 - Foto -> str com endereço da imagem (o atributo deverá armazenar a imagem e não o endereço dela, dica: **from skimage.io import imread**)
 - Fotógrafo -> Pessoa com Nome, CPF, Endereço e telefone
 - Data -> data que a fotografia foi obtida
 - Proprietário -> Pessoa
 - Quantidade de Fotos -> contador para quantidade de objetos Fotografia criados
- A classe Fotografia terá os seguintes métodos:
 - Mostrar Fotografia (dica: **from matplotlib.pyplot import imshow**)
 - Propriedades da Fotografia: tamanho da Fotografia em pixels (dica: **fotografia.shape**), fotógrafo, data
 - Métodos para alterar e acessar atributos.
- Crie Slots