1 October 2019

Orientação a Objetos em Python

Encapsulamento, Atributos e Métodos de classe
SIDIA - Setembro/2019

Jailson P. Januário
jpj.ads@uea.edu.br
Github:@jailsonpj

Apresentação

- **Jailson Pereira Januário**

- Graduando em Sistemas de Informação (EST/UEA)
- Pesquisador no Laboratório de Sistemas Inteligentes
- Buritech

- *Machine Learning*
- *Deep Learning*
- Entusiasta Python
- Coordenador do PyData Manaus

O que veremos?

- Controle de acesso a atributos
- Encapsulamento pytônico
- Entendendo a @property
- Construtores com __new__ e __init__
- Atributos de classe e de instância
- @staticmethod e @classmethod

Revisão

- Declaração de classes
- Declaração de atributos e métodos
- Criação de objetos
- Manipulação de dados de objetos
- Variáveis de referência
- Associação entre classes
- Tudo é objeto

O que temos por aqui?

- Como restringir o acesso a atributos de instância?
- Como trabalhar com construtores?
- Qual a diferença entre variáveis de classe e instância?
- O que são e como usar métodos estáticos?

Encapsulamento

- Encapsulamento é a proteção dos atributos ou métodos de uma classe
- Em Python existem somente o *public* e o *private*, e são definidos no próprio nome do atributo ou método.
- O Python não utiliza o termo **private**, que é um *modificador de acesso* e também chamado de *modificador de visibilidade*
- No Python inserimos dois *underscores*('__') ao atributo para adicionar esta característica

Encapsulamento - Exemplo

```
1  class Pessoa(object):
2      def __init__(self, idade):
3          self.__idade = idade
4
5  pessoa = Pessoa(20)
6  pessoa.idade
```

Traceback (most recent call last):
File "stdin", line 1, in module
AttributeError: 'Pessoa' object has no attribute 'idade'

Dessa maneira não conseguimos acessar o atributo **idade** de um objeto do tipo **Pessoa** fora da classe

Encapsulamento

- O *underscore* alerta que ninguém deve modificar, nem mesmo ler, o atributo em questão.
- Como fazer para mostrar ou modificar um atributo, já que não devemos acessá-lo para leitura diretamente?
- Utilização de métodos
- *getters* e *setters*

Encapsulamento - Decorators

- Como utilizar os métodos privados, para que possam alterar a implementação sem precisar alterar a interface?
- Decorando os métodos com um *decorator* chamado **properties**

Decorator - Exemplo

```
1  class Pessoa(object):
2      def __init__(self):
3          self.__nome = 'sem nome'
4
5      @property
6      def nome(self):
7          return self.__nome
8
9      @nome.setter
10     def nome(self, nome):
11         self.__nome = nome
12
13 pessoa = Pessoa()
14 pessoa.nome = 'Maria'
15 print(pessoa.nome)
```

Decorator - Exemplo

```
1  @property
2  def nome(self):
3      return self.__nome
```

Método que devolve o valor de __nome

```
1  @nome.setter
2  def nome(self, nome):
3      self.__nome = nome
```

Método que altera o valor de __nome

Exercício

- Volte ao arquivo model.py
- Altere a classe **ContaBancaria**
- Altere a variável saldo para __saldo
- Crie a **property** saldo, para oferecer acesso de leitura ao atributo __saldo
- Altere os métodos **sacar** e **depositar**, indicando a variável correta

Convenções

- Todas as classes devem herdar de object
- Nomenclatura de classes (maiúsculas), atributos e métodos (minúsculas):
  - **Classes:** ContaBancaria, Cliente
  - **Atributos e métodos:** saldo, ver_saldo()

Decorator - Exemplo

```
1  @property
2  def nome(self):
3      return self.__nome
```

Método que devolve o valor de __nome

```
1  @nome.setter
2  def nome(self, nome):
3      self.__nome = nome
```

Método que altera o valor de __nome

Atributos de classe e instância

- Nosso banco também quer controlar a quantidade de contas existentes no sistema. Como poderíamos fazer isso?

```
1 total_contas = 0
2 conta = ContaBancaria(200.0)
3 total_contas += 1
4 conta2 = ContaBancaria(300.0)
5 total_contas += 1
```

Como total_contas tem vínculo com a classe **Conta**, ele deve ser um atributo controlado pela *classe* que deve incrementá-lo toda vez que instanciarmos um objeto, ou seja, quando chamamos o método __init__()

Atributos de classe e instância

```
1  class ContaBancaria(object):
2      total_contas = 0
3      def __init__(self,valor):
4          self.agencia = None
5          self.numero = None
6          self.cliente = None
7          self.__saldo = valor
8          self.total_contas += 1
```

c1, c2 = ContaBancaria(200.0), ContaBancaria(300.0)
c2.total_contas
1

Atributos de classe e instância

```
1  class ContaBancaria(object):
2      total_contas = 0
3      def __init__(self,valor):
4          self.agencia = None
5          self.numero = None
6          self.cliente = None
7          self.__saldo = valor
8          ContaBancaria.total_contas += 1
```

Veja que *saldo* é um **atributo de instância** e *total_contas* um **atributo de classe.**

Atributos de classe e instância

- Em orientação a objetos, os atributos podem ser de dois tipos:
  - **Atributos de instância:** Pertencem aos objetos. Acesso via *self.nome_atributo*
  - **Atributos de classe:** Pertecencem à classe. Acesso via *Classe.nome_atributo*. Chamados de atributos estáticos.

Métodos estáticos

- Mas não queremos que ninguém venha acessar nosso atributo *total_contas* e modificá-lo
- Devemos torná-lo privado acrescentando dois '__'
- Criar um método para acessar o atributo

Métodos estáticos

```
1  class ContaBancaria(object):
2      __total_contas = 0
3      # __init__ e outros métodos
4
5      def get_total_contas(self):
6          return ContaBancaria.__total_contas
```

Se executarmos **Conta.get_total_contas()** o no interpretador reclama pois não passamos a instância, e apresentará o seguinte erro:

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

Métodos estáticos

- Queremos um método que seja chamado via classe e via instância sem a necessidade de passar a referência deste objeto, como fazer?
- O Python resolve isso usando **métodos estáticos**
- Não precisam de uma referência e não recebem um primeiro argumento especial (self)
- Para que um método seja considerado estático basta adicionarmos um decorador que se chama *@staticmethod*

Métodos estáticos

```
1  class ContaBancaria(object):
2      __total_contas = 0
3      # __init__ e outros métodos
4
5      @staticmethod
6      def get_total_contas():
7          return ContaBancaria.__total_contas
```

Exercício

- Crie a classe **Funcionario**
- A *matricula* do funcionário deve ser auto incremento e não pode ser alterada
- Na hora de criar um objeto, *nome*, *email* e *salario* são atributos opcionais
- Teste a nova classe

Funcionario
- matricula : int - nome : String - email : String - salario : double

Referências

- LIVRO: Apress - Beginning Python From Novice to Professional
- LIVRO: O'Relly - Learning Python