04

Métodos da classe

Transcrição

Vamos conhecer mais um recurso oferecido pelas classes. Anteriormente, falamos sobre os métodos privados — com o uso do *underscore* (_), sinalizamos para o desenvolvedor quais são eles.

A seguir, imagine que estamos criando um sistema para o Banco do Brasil, no qual todas as contas baseadas nesta classe são referentes ao banco. Geralmente, cada instituição financeira tem um código associado. O código referente ao Banco do Brasil é `001` . Então, dentro do método `__init__` de `Conta` , adicionaremos mais um atributo:

```
class Conta:

    def __init__(self, numero, titular, saldo, limite):
        print("Construindo objeto ... {}".format(self))
        self.__numero = numero
        self.__titular = titular
        self.__saldo = saldo
        self.__limite = limite
        self.__codigo_banco = "001"
```

Criamos o atributo privado `__codigo__banco` , com o valor fixo. Quando instanciarmos um objeto, automaticamente, será inserido o código do Banco do Brasil. Para acessarmos um atributo, criaremos um método `codigo_banco()` , abaixo de `@limite.setter` . Acima do novo método, incluiremos `@property` , para que ele possa ser executado sem o parênteses.

```
@property
def codigo_banco(self):
    return self.__codigo_banco
```

No console, vamos digitar:

```
>>> from conta import Conta
>>> conta = Conta(123, "Nico", 55.5, 1000.0)
Construindo objeto ... <conta.Conta object at 0x10d670208>

>>>  conta.codigo_banco
'001'
```

Conseguimos executar `codigo_banco` sem utilizarmos `()` . Com isso, obtivemos o retorno `001` .

Criaremos outro objeto, agora, referente ao cliente `Marco` .

```
>>> conta = Conta(321, "Marco" 55.5, 1000.0)
Construindo objeto ... <conta.Conta object at 0x10d43fd30>
>>> conta.codigo_banco
'001'
```

Novamente, teremos o mesmo retorno. Em seguida, vamos reiniciar o console e importar a `Conta`.

```
>>> from conta import Conta
```

Neste momento, o objeto ainda não foi criado baseado na classe `Conta`. Se quisermos saber qual é o código do banco, precisamos criar o objeto primeiro. Porém, faria sentido já termos acesso ao código do banco, porque é algo comum entre as contas — uma informação que deveria estar disponível, mesmo antes da criação da conta. O código do banco não depende do objeto.

Então, nosso próximo objetivo é acessar `codigo_banco`, sem ter o objeto criado. No momento, se tentarmos fazer isso, teremos o seguinte resultado.

```
>>> conta.codigo_banco
<property object at 0x10db42e58>
```

Os métodos que estamos trabalhando fazem parte da classe e o objeto é representado pelo `self`. Nós queremos chamar o método `codigo_banco()`, sem a inclusão do objeto, por isso, já podemos remover o `self`:

```
@property
def codigo_banco():
    return self.__codigo_banco
```

Esse métodos que conseguimos chamar sem uma referência recebem o nome de **estáticos**, porque eles fazem parte da classe. Todas as linguagens orientadas a objeto trabalham com **métodos estáticos**, mas para que eles sejam utilizados, iremos configurar os métodos. Fica inapropriado usar `property`, porque ele sempre precisa do `self`. A configuração correta será `@staticmethod`.

```
@staticmethod
def codigo_banco():
    return "001"
```

Em seguida, vamos apagar o atributo `__codigo_banco` dentro do `__init__`.

```
class Conta:

    def __init__(self, numero, titular, saldo, limite):
        print("Construindo objeto ... {}".format(self))
        self.__numero = numero
        self.__titular = titular
        self.__saldo = saldo
        self.__limite = limite
```

Agora se testarmos no console, nosso código funcionará corretamente:

```
>>> from conta import Conta
>>> Conta.codigo_banco()
'001'
```

Observe que nenhum objeto foi criado, mas conseguimos chamar o método estático. Nós especificamos o nome da classe, e depois de acessá-la, chamamos o método.

O próximo passo será criar o passo que devolve todos os códigos dos bancos. Usaremos uma lista com o código de três bancos:

```
{'BB': '001', 'Caixa': '104', 'Bradesco':'237'}
```

Usaremos esse dicionário dentro do `codigos_bancos()`.

```
@staticmethod
def codigos_bancos():
    return {'BB': '001', 'Caixa': '104', 'Bradesco':'237'}
```

Esses códigos foram adicionados apenas como exemplo, mas vocês não precisam conhecê-los. No `return` do método, incluiremos chaves e valores.

```
>>> from conta import Conta
>>> codigos = Conta.codigos_bancos()
>>> codigos
{'BB': '001', 'Caixa': '104', 'Bradesco':'237'}
```

Colocamos a chamada para o método dentro de uma variável. Outra maneira de acessarmos um código específico é por meio de colchetes (`[]`):

```
>>> codigos['BB']
'001'
>>> codigos['Caixa']
'104'
```

Nosso foco está nos métodos estáticos que são da classe, e mesmo sem o objeto, conseguimos executar o método. Em algumas situações isso pode ser útil. Porém, precisamos ser cautelosos com o uso dos métodos estáticos. A ideia do mundo OO é criar objetos. Se usarmos apenas a classe `Conta`, sem ter um objeto, deixaremos de trabalhar com Orientação a Objeto.

Quando todos os objetos compartilham algo em comum, faz sentido usar esses métodos — como no exemplo em que compartilhamos todos os códigos do banco. Mas se utilizarmos apenas métodos estáticos, não utilizaremos mais objetos e nos aproximaremos do mundo procedural.

Vimos alguns conceitos que podem ser praticados com os exercícios sobre métodos estáticos. Continuamos a seguir.