04

Propriedades

Transcrição

Falamos sobre o uso dos *getters* e *setters*, no mundo de Orientação a Objetos. Com eles podemos obter ou alterar um valor específico do nosso objeto.

Usamos como exemplo os métodos `get_saldo()`, `get_titular()`, `get_limite()`, `set_limite()`, mas o **único** que utilizaremos no nosso código é `set_limite`.

Atualmente, o arquivo `conta.py` está assim:

```
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

    def extrato(self):
        print("Saldo de {} de titular{}".format.saldo, self._titular))

    def deposita(self, valor):
        self.__saldo += valor

    def saca(self, valor):
        self.__saldo -= valor

    def transfere(self, valor, destino):
        self.saca(valor)
        destino.deposita(valor)

    def get_saldo(self):
        return self._saldo

    def get_titular(self):
        return self._titular

    def get_limite(self):
        return self._limite

    def set_limite(self, limite):
        self._limite = limite
```

Nós evitamos a criação de métodos como `set_numero`, porque uma conta não deve mudar de número. Observe que não criamos o método `set_saldo`, considerando que o total do saldo também muda. Evitamos fazer isso, porque temos métodos de mais alto nível e mais expressivos, como `transfere()`, para realizar esse tipo de alteração:

```
def transfere(self, valor, destino):
    self.saca(valor)
    destino.deposita(valor)
```

A seguir, mostraremos uma sintaxe alternativa para sintaxe dos *getters*, para isto, criaremos a classe `Cliente`, no arquivo `cliente.py`. Lembrando que não vamos inventar funcionalidades desnecessárias ou que terão utilidade apenas no futuro.

> Existe uma expressão em inglês conhecida na Engenharia de software que é: "***You Ain't Gonna Need It***" (YAGNI, abreviada). Trata-se de uma orientação para que programadores evitem criar funcionalidades para o código fonte de um programa até que estas sejam necessárias.

```
class Cliente:

    def __init__(self, nome):
        self.nome = nome
```

Incluímos no início da classe `__init__()`. Por enquanto, sabemos que os parâmetros necessários serão `self` e `nome`. Para o atributo `nome`, atribuímos o parâmetro `nome`. No console, criaremos um novo `cliente`

```
>>> from cliente import Cliente
>>> cliente = Cliente("Marco")
>>> cliente
<cliente.Cliente object at 0x10b114f28>
```

O construtor da classe `Cliente` recebe o nome do cliente `Marco`.

Quando criamos o método `__init__()` não usamos a sintaxe `__`, adotada pelo Python. Desta forma, o desenvolvedor consegue facilmente acessar o atributo apenas usando a referência. O atributo `nome`, conseguimos alterar.

```
>>> cliente.nome = "Nico"
>>> cliente.nome
'Nico'
```

Agora já podemos pensar em criar uma classe que será aproveitada por outras relacionadas a `Cliente`. Provavelmente, precisaremos do método `get` para validar o dado do atributo `nome`, talvez, para garantir que o nome do titular comece com a letra maiúscula. Por exemplo, no caso de atribuirmos o nome `nico`, com a primeira letra minúscula.

Quando acessarmos o atributo `nome` de `cliente`, queremos que seja executado o método `title()`. Desta forma, o resultado continuará sendo `Nico`, porque o atributo recebeu o tratamento do `get_nome()`.

Vamos alterar o método que passará a se chamar `nome()`, no entanto, isso ainda não será o suficiente. No console, precisaremos dos parênteses para que o método seja executado. Mas nosso objetivo é que a execução ocorra, mesmo sem os parênteses.

Na linguagem Python, os métodos que dão acesso são nomeados como ***properties***. Desta forma, indicaremos para o Python nossa intenção de ter acesso ao objeto.

A declaração de uma *property* é feita com o uso do caractere `@` .

```
@property
```

Com isto, indicamos que este método representa uma propriedade — um termo já recorrente em outras linguagens, como Delphi e C#. Com `@property` , indicamos que estamos trabalhando com uma propriedade. Faremos isso com o método `nome()` .

```
class Cliente:

    def __init__(self, nome):
        self.nome = nome

    @property
    def nome(self):
        return self.nome.title()
```

Agora, quando digitarmos nos console `cliente.nome` , sem a adição dos parênteses, e conseguiremos que o método seja executado como antes.

Para explicitarmos que `nome()` está sendo executado por baixo dos panos, imprimiremos a mensagem `chamando @property nome()` , adicionando um `print()` ao método. Também tornaremos privado o atributo `nome` que será antecedido por `__` .

```
class Cliente:

    def __init__(self, nome):
        self.__nome = nome

    @property
    def nome(self):
        print("chamando @property nome()")
        return self .__nome.title()
```

Se esquecermos de adicionar `__` ao atributo `nome` e torná-lo privado, receberemos uma mensagem de erro quando tentarmos acessá-lo no console. Após as alterações no código, vamos fazer testes no console.

Começaremos criando a referência `cliente` para a conta do `nico` (com a letra **minúscula**).

```
>>> cliente = Cliente("nico")

>>> cliente.nome
chamando @property nome()
'Nico'
```

A maneira como escrevemos no console, parece que estamos acessando diretamente o atributo, porém o método `nome()` foi chamado. Começamos a classe de forma bastante simples, apenas com a função inicializadora, depois,

sentimos a necessidade de criar o getter. No caso, optamos em incluir `@property` para continuarmos com a mesma sintaxe do atributo, mas com o método sendo executado internamente.

Da mesma forma como fazemos isso para um getter, faremos para um setter. Novamente, criaremos um método de `nome()`, logo abaixo da propriedade do getter:

```
def nome(self, nome):
    print("chamando setter nome()")
    self.__nome = nome
```

Criamos um setter sem a adicionação do `set` antes do nome do método. Mas para que ele funcione, teremos que adicionar também uma configuração: `@nome.setter`.

```
@nome.setter
def nome(self, nome):
    print("chamando setter nome()")
    self.__nome = nome
```

Especificamos qual atributo receberá o setter.

Testaremos no console para garantirmos que a nossa sintaxe simplificada está funcionando.

```
>>> from cliente import Cliente
>>> cliente = Cliente("nico")
>>> cliente.nome
```

Se tentarmos mudar o atributo `nome` para `marco`, o método será executado mesmo sem o uso dos parênteses.

```
>>> from cliente import Cliente
>>> cliente = Cliente("nico")
>>> cliente.nome = "marco"
chamando setter nome()
>>> cliente.nome
chamando @property nome()
'Marco'
```

A seguir, acessaremos o arquivo `conta.py` e vamos trabalhar com o método `get_limite()`, adicionando `@property`. Agora não precisaremos mais da palavra `get`.

```
def get_titular(self):
    return self._titular

@property
def limite(self)
    return self.__limite

@limite.setter
def limite(self, limite):
    self.__limite = limite
```

Observem que retiramos o `get` do `get__limite` e `set` do `set__limite`. Temos a opção de fazer a mesma alteração com outros métodos, mas faremos isso mais adiante nos exercícios. A seguir, tentaremos acessar os dados de uma conta.

```
>>> from conta import Conta
>>> conta = Conta(123, "Nico", 55.5, 1000.0)
Construindo objeto ... <conta.Conta   o bject at 0x1019df3c8>
>>> conta.limite
1000.0
```

A execução de `conta.limite` é semelhante a de getter. Podemos utilizar o setter também:

```
>>> conta.limite = 2000.0
>>> conta.limite
2000.0
```

Na linha em que escrevemos `conta.limite = 2000.0` pareceu que estávamos atribuindo, mas estávamos na realidade executando setter.

Temos ferramentas suficientes para trabalhar nos exercícios, falamos sobre as *properties*, que colaboram para manter a nossa sintaxe amigável e chamam por baixo dos panos os métodos `get` e `set`.