01

Atributos privados

Transcrição

Nós criamos atributos e métodos na classe `Conta`, por isso, ela já funciona. Mas ainda está incompleta.

Falamos anteriormente sobre encapsulamento, no console do PyCharm, importaremos da classe `Conta`.

```
>>> from conta import Conta

>>> conta = Conta(123, "Nico", 55.5, 1000.0)
Construindo objeto ... <conta.Conta object at 0x102b89128>
```

Chamamos uma `Conta`, chamando a função construtora `__init__` por baixo dos panos. Com isso, conseguimos acessar os métodos (como `sacar()` ou `deposita()`) para acessar o atributo do objeto. Se quisermos imprimir o saldo, utilizaremos o método `extrato()` ou `saldo()`. No caso, invocaremos o último:

```
>>> conta.saldo
55.5
```

Acessaremos o objeto usando a referência. Com ela, conseguiremos também alterar o saldo da conta:

```
>>> conta.saldo = 60.0
```

Agora, se pedirmos para imprimir o extrato, o valor será atualizado:

```
>>> conta.saldo = 60.0

>>> conta.extrato()
Saldo de 60.0 do titular Nico
```

Porém, isto não deveria acontecer. O valor do saldo da conta deveria ser alterado a partir do método `deposita()`, localizalizado em `conta.py`:

```
def deposita(self, valor):
    self.saldo += valor

def saca(self, valor):
    self.saldo -= valor
```

Se quiséssemos saber o nome de alguém, seria uma falta de educação pegar diretamente o documento de identificação da pessoa, sem pedir autorização. Da mesma forma, seria mais apropriado usarmos um método para identificar o saldo, em vez de acessá-lo diretamente.

Não podemos acessar o atributo `saldo` do objeto diretamente. Teremos que usar os métodos responsáveis por encapsular o acesso ao objeto.

Então, para melhorarmos a classe `Conta`, devemos restringir o acesso a `saldo`, tornando-o **privado**, adicionando dois caracteres *underscore* (`__`).

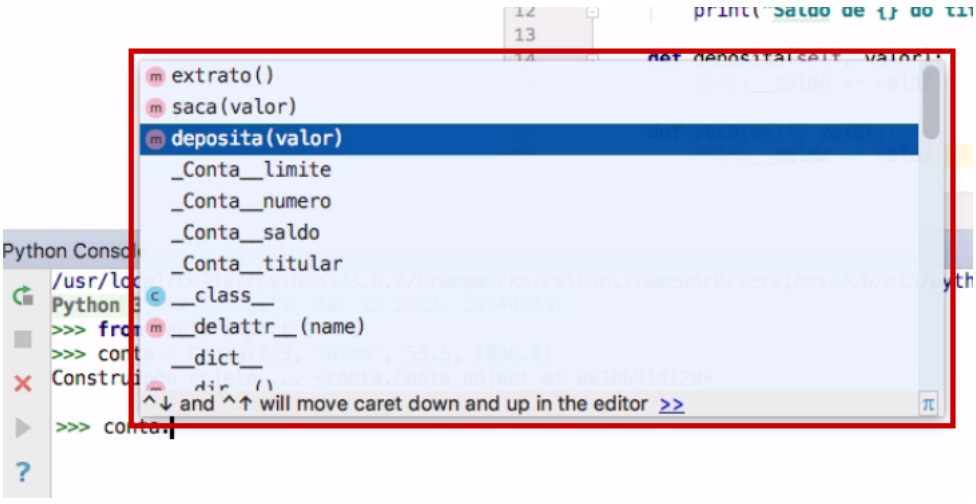
```
class Conta:

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

Em algumas linguagens como Java, a palavra ***private*** define o atributo como privado e é chamado como **modificador de visibilidade**. Porém, em Python, foi convencionado o uso `__`. Com isso, nós renomeamos os atributos seguindo uma nomenclatura especial, por exemplo, `numero` passou a se chamar `__numero`.

Para testar as alterações, reiniciaremos o console e criaremos um novo objeto.

```
>>> from conta import Conta
>>> conta = Conta(123, "Nico", 55.5, 1000.0)
Construindo objeto ... <conta.Conta object at 0x10f6f5630>
```

Tudo continua funcionando corretamente. Em seguida, tentaremos acessar o atributo referente ao saldo. Se você observar, quando digitarmos a referência `conta` no console do Pycharm, o autocomplete já nos oferecerá opções diferentes.



Primeiramente, serão listados três métodos (`extrato` , `saca` e `deposita`), depois, vemos `_Conta_limite` , `_Conta_numero` , `_Conta_saldo` , `_Conta_titular` . Vamos tentar acessar os atributos `__limite` e `__saldo` .

```
>>> conta._Conta__limite
1000.0
>>> conta._Conta__saldo
55.5
```

Nós continuamos a ter acesso aos atributos, ainda que eles tenham mudado de nome — o Python adicionou a classe antecedido por `_`. Ao escrevermos `conta._Conta__limite`, o Python informará ao desenvolvedor que o atributo `__saldo` não deve ser acessado.

O Python **avisa** que o atributo foi criado para ser usado dentro da classe, por meio dos métodos. Porém, continuaremos a ter acesso aos valores. Mas se o desenvolvedor decidir acessar o atributo igualmente, ele será alertado de que está fazendo algo inapropriado, ou seja, está "brincando com fogo".

A ação de tornar privado o acesso aos atributos, no mundo Orientado a Objetos, chamamos de **encapsulamento**. Com isso, definimos que o acesso deve ocorrer apenas por meio dos métodos.

A seguir, falaremos mais sobre encapsulamento, mas antes faça os exercícios.