Decisões de projeto da linguagem de programação Python 3.7.4 (OOP)

Pedro Probst Minini | 201710013

ELC1088 - Implementação de Linguagens de Programação

2019-08-22

---

1. Atributos e métodos

Em Python, há diferenças entre *atributos de classe* e *atributos de instância*.

Atributos de instância são os atributos que pertencem a apenas um objeto; essas variáveis são acessíveis no escopo desse objeto e são definidas em seu método construtor.

Atributos de classe são os atributos que pertencem a uma determinada classe e não a apenas um objeto específico. São compartilhados entre todos os objetos da classe e são definidos fora do construtor.

```
class Planeta:
    # Atributo de classe
    galaxia = "Via Láctea"

    # Método construtor
    def __init__(self, nome, raio, gravidade, sistema):
        # Atributos de instância
        self.nome = nome
        self.raio = raio # km
        self.gravidade = gravidade # m/s^2
        self.sistema = sistema

    # Método destrutor
    def __del__(self):
        print("Planeta destruído.")

# 'marte' é uma instância da classe Planeta
marte = Planeta("Marte", 3389.5, 3.711, "Sistema Solar")
```

Python possui três tipos de métodos: *métodos de instância* (padrão), *métodos de classe* e *métodos estáticos*.

Métodos de instância recebem o parâmetro `self`, que aponta para a instância da classe quando o método é chamado. Pelo parâmetro `self`, métodos de instância podem acessar livremente atributos e outros métodos do mesmo objeto. Métodos de instância também podem acessar a classe fazendo uso do atributo `self.__class__`.

Métodos de classe recebem o parâmetro `cls`, que apontam para a classe em questão (e não para a instância) quando o método é chamado. Métodos de classe podem modificar o estado da classe que se aplica a todas as instâncias da mesma, mas não de um objeto em específico.

Por fim, métodos estáticos não recebem parâmetros `self` ou `cls`, ou seja, não podem modificar tanto o estado do objeto quanto o estado da classe. Métodos estáticos só tem acesso aos parâmetros recebidos.

Do exemplo acima:

```
class Planeta:
    galaxia = "Via Láctea"

    def __init__(self, nome, raio, gravidade, sistema):
        # corpo
```

```
    # Método de instância
    def explodir(self):
        return f"{self.nome} explodiu no {self.sistema}, localizado na
        {self.__class__.galaxia}."

    # Método de classe
    @classmethod
    def localizacao(cls):
        return f"{cls.galaxia}"

    # Método estático
    @staticmethod
    def calculo(a = 2500, b = 6):
        return a * b

marte = Planeta("Marte", 3389.5, 3.711, "Sistema Solar")
print(marte.explodir())
    # Marte explodiu no Sistema Solar, localizado na Via Láctea.
print(Planeta.localizacao())
    # Via Láctea
print(Planeta.calculo())
    # 15000
```

2. Herança

Tendo suporte a orientação a objetos, Python naturalmente possui suporte a *herança*. A sintaxe é a seguinte:

```
class NomeClasseDerivada(NomeClasseBase):
    # corpo
```

Herança múltipla também é suportada em Python:

```
class NomeClasseDerivada(Base1, Base2, Base3):
    # corpo
```

Exemplo clássico:

```
class Pessoa:
    def __init__(self, nome, sobrenome):
        self.nome = nome
        self.sobrenome = sobrenome

    def print_nome(self):
        print(self.nome, self.sobrenome)

class Estudante(Pessoa):
    def __init__(self, nome, sobrenome, matricula):
        # super() para herdar os métodos e propiedades da classe base Pessoa
        super().__init__(nome, sobrenome)
        self.matricula = matricula

pedro = Estudante("Pedro", "Probst", "201710013")
```

3. Polimorfismo

Diferentemente de Java, Python utiliza *duck typing*. Isso quer dizer, por exemplo, que ao invés de nos perguntarmos se um objeto é um pato, a pergunta correta seria se o objeto faz "quack" como um pato e anda como um pato. O polimorfismo em Python, portanto, se baseia em semelhanças.

Por exemplo, duas classes distintas precisam de uma interface comum, então criaremos métodos que são distintos mas que possuem o mesmo nome.

```
class Tubarao():
    def nadar(self):
        print("O tubarão está nadando.")

    def nadar_para_tras(self):
        print("O tubarão não pode nadar para trás, mas pode afundar para trás")

    def esqueleto(self):
        print("O esqueleto do tubarão é feito de cartilagem.")

class PeixePalhaco():
    def nadar(self):
        print("O peixe-palhaço está nadando.")

    def nadar_para_tras(self):
        print("O peixe-palhaço pode nadar para trás.")

    def esqueleto(self):
        print("O esqueleto do peixe-palhaço é feito de osso.")
```

Agora, vamos instanciar essas classes em dois objetos:

```
...
alamo = Tubarao()
alamo.esqueleto()

guri = PeixePalhaco()
guri.esqueleto()
```

Quando rodarmos o programa, o output será o seguinte:

```
O esqueleto do tubarão é feito de cartilagem.
O esqueleto do peixe-palhaço é feito de osso.
```

Como agora temos dois objetos que fazem uso de uma interface comum, nós podemos usar os dois objetos do mesmo modo, indepentendemento dos seus tipos individuais.

Podemos mostrar como Python pode usar cada uma dessas classes distintas do mesmo modo com o código abaixo.

```
...
alamo = Tubarao()

guri = PeixePalhaco()

for peixe in (alamo, guri):
    peixe.nadar();
    peixe.nadar_para_tras();
    peixe.esqueleto();
```

Quando rodamos o programa, o output é o seguinte:

```
O tubarão está nadando.
O tubarão não pode nadar para trás, mas pode afundar para trás.
O esqueleto do tubarão é feito de cartilagem.
O peixe-palhaço está nadando.
O peixe-palhaço pode nadar para trás.
O esqueleto do peixe-palhaço é feito de osso.
```

Isso mostra que Python usa esses métodos sem se importar sobre qual tipo de classe cada um desses objetos é. Ou seja, usando esses métodos de um modo polimórfico.

Em Python também podemos fazer polimorfismo com funções. No caso, criaremos uma função que pode receber qualquer objeto.

```
...
def no_pacifico(peixe):
    peixe.nadar()

alamo = Tubarao()

guri = PeixePalhaco()

no_pacifico(alamo)
no_pacifico(guri)
```

Output:

```
O tubarão está nadando.
O peixe-palhaço está nadando.
```

4. Encapsulamento

Variáveis "privadas" que não podem ser acessadas exceto de dentro de um objeto não existem em Python. Entretanto, há a seguinte convenção: um nome prefixado de `_` deve ser tratado como uma parte não-pública da API (pense "protected"). Nomes prefixados com `__` são "private".

Em suma, o encapsulamento em Python não passa de uma convenção; formalmente, ele é inexistente na linguagem.

```
class Copo:
    def __init__(self):
        self.cor = None
        self.__conteudo = None

    def encher(self, bebida):
        self.__content = bebida

    def esvaziar(self):
        self.__content = None

redCup = Copo("vermelho")
# Com certo esforço sintático, o 'conteudo' pode ser acessado:
redCup._Copo__conteudo = "chá" # não é uma boa prática
```

5. Instanciação

É comum instanciar (criar um objeto de uma classe) usando métodos construtores `__init__()`, que são chamados no momento de instanciação. O método `__del__()` é o método destrutor; ele é chamado quando todas as referências ao objeto foram deletadas (por exemplo, quando um objeto é *garbage collected*; assim sendo, o destrutor manual não é necessário a menos em casos muito específicos).

Ver o primeiro exemplo de código na seção 1 (*Atributos e métodos*), que exemplifica de bom modo os métodos construtor e destrutor.

6. Interfaces

Como já foi mencionado, Python utiliza *duck typing* (ou interfaces automáticas) e uma boa implementação de herança múltipla, o que dispensa o uso de interfaces.

Ver exemplos das seções 2 (*Herança*) e 3 (*Polimorfismo*).

7. Classes aninhadas

Em Python, uma classe pode ter múltiplas classes aninhadas no seu corpo.

```
class Carro():
    def __init__(self, cor):
        self.cor = cor

    class Motor():
        def __init__self(self, modelo):
            self.modelo = modelo

        def corre(self):
            print("VROOOOOOOOOOOOOM")

camaro = Carro("Azul").Motor("V8")
camaro.corre()
    # VROOOOOOOOOOOOM
```

8. Classes anônimas

Python não fornece classes anônimas oficialmente, mas é possível fazê-las usando a função *built-in* `type` .

```
class Exemplo:
    pass

Exemplo= type('', (Exemplo,), {'sair': lambda self: print('AQUI!')})()
Exemplo.sair()
    # AQUI!
```

9. Genéricos

Novamente, Python usa *duck typing*, o que dispensa o uso de uma sintaxe especial para lidar com múltiplos tipos.

10. *Closure*

A ideia geral no uso de *closures* é acessar variáveis que são definidas fora do escopo corrente. Considere:

```
b = 6

def f1(a):
    print(a)
    print(b)

    def f2():
        c = a + b
        return c * 3

    return f2 # retorna uma versão não executada de f2

f2 = f1(10)
    # 10
    # 6
c = f2()
print(c)
    # 48
```

No exemplo acima, `f2()` foi executada fora do escopo de `f1()` , mas ainda assim teve acesso à variável `a` definida em `f1()` ; `a` não foi *garbage collected* pois o escopo interno de `f1()` ainda estava em uso por `f2()` . A referência que `f2()` tem sobre aquele escopo é chamada de *closure*.

11. *Mixins*

Mixins são facilmente criados a partir de herança múltipla, sem exigir nenhum esforço sintático extra.

```
class Exemplo(Mixin1, Mixin2, ClasseBase):
    # corpo
```

Referências

- *"Python 3.7 Documentation"*
  - https://docs.python.org/3.7/index.html
- *"Python Class Attributes: An Overly Thorough Guide"*
  - https://www.toptal.com/python/python-class-attributes-an-overly-thorough-guide
- *"Python's Instance, Class, and Static Methods Demystified"*
  - https://realpython.com/instance-class-and-static-methods-demystified/
- *"How To Apply Polymorphism to Classes in Python 3"*
  - https://www.digitalocean.com/community/tutorials/how-to-apply-polymorphism-to-classes-in-python-3
- *"Private, protected and public in Python"*
  - https://radek.io/2011/07/21/private-protected-and-public-in-python/
- "Programação Orientada a Objeto - Python"
  - https://wiki.python.org.br/ProgramacaoOrientadaObjetoPython#A6_Classes_Aninhadas
- *"Does Python have something like anonymous inner classes of Java*?"
  - https://stackoverflow.com/questions/357997/does-python-have-something-like-anonymous-inner-classes-of-java