

ASIMOV

Programação orientada a objetos

A Programação Orientada a Objetos (POO) tende a ser um dos principais obstáculos para iniciantes quando começam a aprender Python.

Há muitos, muitos tutoriais e lições que cobrem o POO, então sinta-se livre e vá ao Google pesquisar outras lições. Também coloquei alguns links para outros tutoriais úteis on-line na parte inferior deste Notebook.

Para esta lição vamos construir nosso conhecimento de POO em Python, abordando os seguintes tópicos:

- Objetos
- Usando a palavra-chave `class`
- Criando atributos de classe
- Criando métodos em uma classe
- Herança
- Métodos especiais para classes

Vamos começar a lição lembrando sobre os objetos básicos do Python. Por exemplo:

```
In [1]: l = [1,2,3]
```

Lembre-se de como podemos chamar métodos em uma lista?

```
In [3]: l.count(2)
```

```
Out[3]: 1
```

O que vamos fazer basicamente nesta palestra é explorar como podemos criar um tipo de objeto como uma lista. Já aprendemos sobre como criar funções. Então, vamos explorar objetos em geral:

Objetos

Em Python, *tudo é um objeto*. Lembre-se de palestras anteriores que podemos usar o `type()` para verificar o tipo de objeto de algo:

```
In [1]: print(type(1))
        print(type([]))
        print(type(()))
        print(type({}))
```

```
<class 'int'>
<class 'list'>
<class 'tuple'>
<class 'dict'>
```

Então sabemos que tudo isso são objetos, então, como podemos criar nossos próprios tipos de Objetos? É aí que entra a palavra-chave *class*.

class

Os objetos definidos pelo usuário são criados usando a palavra-chave da class. A classe é um modelo que define a natureza de um objeto futuro. Das classes podemos construir instâncias. Uma instância é um objeto específico criado a partir de uma determinada classe. Por exemplo, acima, criamos o objeto 'l' que era uma instância de um objeto da classe lista.

Vamos ver como podemos usar **classe**:

```
In [2]: # Crie um novo tipo de objeto chamado Sample
class Sample(object):
    pass

# Instanciando Sample
x = Sample()

print(type(x))
```

```
<class '__main__.Sample'>
```

Por convenção damos às classes um nome que começa com uma letra maiúscula. Observe como x é agora a referência para nossa nova instância da classe Sample. Em outras palavras, nós **instanciamos** a classe Sample.

Dentro da classe temos apenas um *pass*. Mas podemos definir atributos e métodos de classe.

Um **atributo** é uma característica de um objeto. Um **método** é uma operação que podemos realizar com o objeto.

Por exemplo, podemos criar uma classe chamada Dog. Um atributo de Dog pode ser sua raça ou seu nome, enquanto um método de um cão pode ser definido por um método .latir() que retorna um som.

Vamos ter uma melhor compreensão dos atributos através de um exemplo.

Atributos

A sintaxe para criar um atributo é: `self.attribute = something` Existe um método especial chamado:

init()

Esse método é usado para inicializar os atributos de um objeto. Por exemplo:

```
In [3]: class Dog(object):
        def __init__(self, raça):
```

```
        self.raça = raça
        self.idade = 10

    def envelhecer(self):
        self.idade += 3

sam = Dog(raça='Lab')
frank = Dog(raça='Huskie')
```

Vamos descrever o que temos acima. O método especial

init() é chamado automaticamente logo após o objeto ter sido criado:

def **init** (self, raça): Cada atributo em uma definição de classe começa com uma referência ao objeto de instância. É convencionalmente chamado de self. A raça é o argumento. O valor é passado durante a instanciação da classe.

```
self.breed = breed
```

Agora criamos duas instâncias da classe Dog. Com dois tipos de raças, podemos acessar estes atributos assim:

```
In [6]: sam.raça
```

```
Out[6]: 'Lab'
```

```
In [8]: frank.raça
```

```
Out[8]: 'Huskie'
```

Observe como não temos parênteses após a raça, isto é porque é um atributo e não leva nenhum argumento.

Em Python também existem *atributos de objeto de classe*. Esses atributos de objeto de classe são os mesmos para qualquer instância da classe. Por exemplo, podemos criar o atributo *especie* para a classe Dog. Os cães (independentemente da sua raça, nome ou outros atributos serão sempre mamíferos. Aplicamos essa lógica da seguinte maneira:

```
In [12]: class Dog(object):

    # Atributos de objetos de classe
    species = 'mammal'

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

```
In [13]: sam = Dog('Lab', 'Sam')
```

```
In [14]: sam.nome
```

```
Out[14]: 'Sam'
```

Note that the Class Object Attribute is defined outside of any methods in the class. Also by

convention, we place them first before the init.

```
In [7]: sam.species
```

```
Out[7]: 'mammal'
```

Métodos

Métodos são funções definidos dentro do corpo de uma classe. Eles são usados para executar operações com os atributos de nossos objetos. Os métodos são essenciais no conceito de encapsulamento em OOP. Isso é essencial para dividir as responsabilidades na programação, especialmente em grandes aplicações.

Você pode basicamente pensar em métodos como funções que atuam em um Objeto que levam o próprio Objeto através de seu argumento *self*.

Vamos passar por um exemplo de criação de uma classe Circle:

```
In [1]: class Circle(object):
        pi = 3.14

        # O círculo é instanciado com um raio (o padrão é 1)
        def __init__(self, radius=1):
            self.radius = radius

        # Método de cálculo da área. Observe o uso de si mesmo.
        def area(self):
            return self.radius * self.radius * Circle.pi

        # Método que redefine a área
        def setRadius(self, radius):
            self.radius = radius

        # Método para obter raio (Mesmo que apenas chamar .radius)
        def getRadius(self):
            return self.radius

c = Circle()

c.setRadius(2)
print('O raio é :', c.getRadius())
print('A área é', c.area())
```

```
O raio é : 2
A área é 12.56
```

Ótimo! Observe como nós usamos a notação *self*. para atributos de referência da classe dentro das chamadas do método. Revise como o código acima funciona e tente criar seu próprio método

Herança

A herança é uma forma de formar novas classes usando classes que já foram definidas. As classes recém formadas são chamadas classes derivadas, as classes de que derivamos são chamadas de classes base. Os benefícios importantes da herança são a reutilização de

códigos e a redução da complexidade de um programa. As classes derivadas (descendentes) substituem ou estendem a funcionalidade das classes base (ancestrais).

Vejamos um exemplo incorporando nosso trabalho anterior na classe Dog:

```
In [18]: class Animal(object):
        def __init__(self):
            print("Animal created")

        def whoAmI(self):
            print("Animal")

        def eat(self):
            print("Eating")

        class Dog(Animal):
            def __init__(self):
                Animal.__init__(self)
                print("Dog created")

            def whoAmI(self):
                print("Dog")

            def bark(self):
                print("Woof!")
```

```
In [19]: d = Dog()
```

```
Animal created
Dog created
```

```
In [20]: d.whoAmI()
```

```
Dog
```

```
In [21]: d.eat()
```

```
Eating
```

```
In [22]: d.bark()
```

```
Woof!
```

Neste exemplo, temos duas classes: Animal e Dog. O animal é a classe base, o cão é a classe derivada.

A classe derivada herda a funcionalidade da classe base.

- É mostrado pelo método eat().

A classe derivada modifica o comportamento existente da classe base.

- mostrado pelo método whoAmI().

Finalmente, a classe derivada estende a funcionalidade da classe base, definindo um novo método de bark().

Métodos especiais

Finalmente, vamos dar uma olhada em métodos especiais. Classes em Python podem implementar determinadas operações com nomes de métodos especiais. Esses métodos não são realmente chamados diretamente, mas pela sintaxe de linguagem específica de Python. Por exemplo, crie uma classe de livro:

```
In [23]: class Book(object):
        def __init__(self, title, author, pages):
            print("A book is created")
            self.title = title
            self.author = author
            self.pages = pages

        def __str__(self):
            return "Title:%s , author:%s, pages:%s " %(self.title, self.author, self.pages)

        def __len__(self):
            return self.pages

        def __del__(self):
            print("A book is destroyed")
```

```
In [24]: book = Book("Python Rocks!", "Rodrigo Tadewald", 159)

        # Métodos especiais
        print(book)
        print(len(book))
        del book
```

```
A book is created
Title:Python Rocks! , author:Rodrigo Tadewald, pages:159
159
A book is destroyed
```

Os métodos `init ()`, `str ()`, `len ()` e `del ()`. Esses métodos especiais são definidos pelo uso de sublinhados. Eles nos permitem usar funções específicas do Python em objetos criados através da nossa classe.

Ótimo! Após esta palestra, você deve ter uma compreensão básica de como criar seus próprios objetos com classe em Python. Você estará utilizando isso fortemente em seu próximo projeto!

Para maiores recursos neste tópico, confira:

[Post de Jeff Knupp](#)

[Correio da Mozilla](#)

[Tutorial's Point](#)

[Documentação oficial](#)