Objetos em Python

Carlos Tavares

3 de Dezembro de 2024

Objetos em Python i

Python é uma linguagem de múltiplos paradigmas: imperativa, estruturada e, em certa medida, funcional

Também permite programação orientada a objetos, o que a torna também uma linguagem orientada a objetos.

Objetos em Python ii

O que é um objeto?

Para definir um objeto, é necessário definir uma **classe**: uma coleção de elementos que partilham propriedades semelhantes. Por exemplo, a classe de todas as pessoas, a classe de todos os carros, a classe de todas as casas, a classe de todos os triângulos.

Um objeto é uma instância de uma classe. Por exemplo, o Ferrari FXX é uma instância de carro, e um triângulo isósceles é uma instância de triângulo.

Objetos em Python iii

Em Python, assim como na maioria das linguagens de programação, um objeto é algo que permite agregar dados e comportamentos.

Um tipo de tuplo (dados) com funções associadas.

Existem muitos exemplos em Python: **strings** ou **ficheiros**.

Sempre que usamos **entidade.função()**, a entidade é um objeto.

Classes e objetos i

Como definir uma classe em Python:

class nome_da_classe:
(...)

Depois da classe ser definida, podemos instanciar objetos:

objeto = Classe()

Classes e objetos ii

Exemplo:

A **classe vazia**:

```
class vazio:
    pass

obj = vazio()
```

Classes e objetos iii

Outro exemplo: uma imitação de um tuplo

```
class imitacao_tuplo:
    nome = "none"
    idade = 0

obj = imitacao_tuplo()
```

Podemos ver os valores em obj:

```
» obj.nome
"none"
» obj.idade
0
```

Classes e objetos iv

Também podemos **modificar** os valores em obj:

» obj.nome = "Tarzan"
» obj.idade = "150"

Também podemos instanciar outros objetos da mesma classe:

» obj2 = imitacao_tuplo ()
» obj2.nome = "Mogli"
» obj2.idade = 13

Adicionar comportamento aos objetos: construtor i

Objetos consistem em dados e comportamento.

Podemos adicionar métodos aos objetos. Um método particularmente importante é o método construtor:

```
class classe_aleatoria:
    def __init__(self, argumentos):
        (...)
```

O método init é invocado quando o objeto é instanciado. Todos os métodos que pertencem a uma classe devem receber self como argumento.

Adicionar comportamento aos objetos: construtor ii

Exemplo: tuplo com construtor

```
class tuplo_com_construtor:
    def __init (self, nome = "nenhum", idade = 0):
        self.nome = nome
        self.idade = idade
```

Isto cria um objeto com os nomes e idades dados como argumentos ao construtor. Este construtor aceita argumentos opcionais, pois ambos têm valores padrão.

Adicionar comportamento aos objetos: outros métodos i

Ainda sobre a adição de métodos (podemos adicionar tantos métodos quanto quisermos):

```
class alguma_classe:
    def __init__(self, argumentos):
        (...)

    def outro_metodo(self, argumentos):
        (...)
```

O método init é invocado quando o objeto é instanciado. Todos os métodos que pertencem a uma classe devem receber self como argumento.

Adicionar comportamento aos objetos: outros métodos ii

Exemplo: tupla com construtor e método

```
class tuplo_que_faz_algo:
    def __init__(self, nome = , idade = 0):
        self.nome = nome
        self.idade = idade

    def definir_como_tarzan(self):
        self.nome = "Tarzan"
        self.idade = 120
```

Herança i

As funções já permitem a reutilização de código...

...então porque objetos?

Objetos permitem a reutilização de um conjunto completo de dados e comportamentos através de herança, o que oferece uma vantagem estrutural.

O principal mecanismo de reutilização é a herança: é possível derivar classes a partir de outras classes.

Herança ii

Exemplo: Vamos supor que temos uma classe Plane

```
class Plane:

    def __init__(self, model, capacity, range_km):
        self.model = model
        self.capacity = capacity
        self.range_km = range_km
        self.is_flying = False

    def take_off(self):
        if not self.is_flying:
            self.is_flying = True
            print(f"self.model has taken off.")
        else:
```

Herança iii

```
        print(f"self.model is already flying.")

    def land(self):
        if self.is_flying:
            self.is_flying = False
            print(f"self.model has landed.")
        else:
            print(f"self.model is already on the ground.")
```

Herança iv

A classe FighterPlane herda todos os métodos e propriedades da classe Plane. Também é possível substituir métodos da classe pai (veja o método viver) e invocar métodos da classe pai via super.metodo (veja o método construtor).

```
class FighterPlane(Plane):

    def __init__(self, model, capacity, range_km, weapon_type):
        super().__init__(model, capacity, range_km)
        self.weapon_type = weapon_type

    def engage(self):
        if self.is_flying:
```

Herança v

```
            print(f"self.model is engaging the enemy with
self.weapon_type.")
        else:
            print(f"self.model can’t engage while on the ground.")
```

Python admite programação imperativa, estruturada, funcional (funções de ordem superior) e orientada a objetos.

Exercicio. Faça um programa para gerir contas bancárias que tem duas entidades:

- Conta bancária:
  - Pode ter vários proprietários;
  - Permite acrescentar fundos, apenas aos proprietarios;
  - Permite levantar fundos, retornado ao utilizador o capital levantado, apenas aos proprietarios;
  - Tem uma capitalização feita periodicamente num juro de valor dado no inicio da conta.
- Proprietario:
  - Pode levantar dinheiro acrescentando ao seu proprio capital;
  - Só pode ter uma conta;
  - Dois perfis de proprietario: poupadinho e o gastador. O primeiro levanta sempre um montante constante, o segundo levanta um montante que dobra de cada vez que levanta dinheiro.

Perguntas?