TIC em trilhas

Programação Orientada a Objetos

Módulo 2

Foto de Goran Ivos, disponível na Unsplash. Adaptada pelo autor.

Sumário

Ilustração feita por upklyak, disponível no Freepik. Adaptada pelo autor.

Herança

Beleza, agora que entendemos o que é uma classe, vamos para o próximo degrau que é aplicar herança a classes definidas.

A herança é um conceito fundamental da programação orientada a objetos (POO) que permite a criação de uma nova classe a partir de uma classe existente. A nova classe recebe o nome de classe derivada ou subclasse ou ainda classe filha, e a classe da qual ela é construída chamada de classe base ou superclasse ou classe mãe.

Aqui está um exemplo simples para entender o que é proposto em herança:

```
#
# Classe Animal.
# também referenciada como base, superclasse, ou classe mãe.
#
class Animal:
    def __init__(self, nome):
        self.nome = nome

    def emitir_som(self):
        pass

#
# Classe Cachorro.
# também referenciada como derivada, subclasse, ou classe filha.
#
class Cachorro(Animal):
    def emitir_som(self):
        return “Latindo!»

#
# Classe Gato.
# também referenciada como derivada, subclasse, ou classe filha.
#
class Gato(Animal):
    def emitir_som(self):
        return “Miando!»


banze = Cachorro("Banze“)
mika = Gato("Mika“)

print(banze.falar())
print(mika.falar())
```

Olha que bacana. Neste caso, Animal é a classe base e Cachorro e Gato são subclasses. As subclasses herdam todos os atributos e métodos da classe base, mas também podem sobrescrever os métodos da classe base, como demonstrado pelo método emitir_som().

A herança permite a reutilização de código, torna o código mais fácil de manter e representa uma relação do tipo “é um”. Por exemplo, um Cachorro é um Animal, portanto, faz sentido que Cachorro herde de Animal.

A ideia é que coloquemos na classe base tudo que for genérico e que deva valer para todas as subclasses. E as particularidades ficam definidas para cada subclasse.

Polimorfismo

O polimorfismo é outro conceito interessante da programação orientada a objetos, pois permite que um objeto seja tratado de várias formas diferentes, dependendo de seu tipo ou classe. Em outras palavras, o polimorfismo permite que uma classe única represente uma variedade de tipos ou classes.

Aqui está um exemplo simples de polimorfismo:

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

    def emitir_som(self):
        pass

class Cachorro(Animal):
    def emitir_som(self):
        return “Latindo!»

class Gato(Animal):
    def emitir_som(self):
        return “Miando!»
```

Ilustração feita por upklyak, disponível no Freepik. Adaptada pelo autor.

```
#
# Função que aplica polimorfismo
#
def gerar_som(animal):
    print(animal.falar())

banze = Cachorro("Banze“)
mika = Gato("Mika“)

print(banze.falar())
print(mika.falar())
```

Perceba o seguinte, a função gerar_som() é capaz de aceitar qualquer objeto que seja uma instância de Animal ou de uma de suas subclasses. Isso é polimorfismo sendo aplicado. A função gerar_som() pode interagir com diferentes tipos de objetos (Cachorro, Gato, etc.) desde que eles sigam a mesma implementação (ou seja, tenham um método gerar_som()).

O polimorfismo é um recurso que permite que o código seja mais flexível e extensível. Você pode adicionar novas classes ao seu programa sem alterar as funções que usam polimorfismo, desde que as novas classes sigam a mesma classe base ou interface.

Interfaces

Uma interface, no Python, é um ‘acordo’ que define quais métodos uma classe deve implementar. É outro conceito fundamental na programação orientada a objetos que promove a reutilização de código e a implementação de polimorfismo.

Em algumas linguagens de programação, como Java e TypeScript dentre outras, as interfaces são um recurso de linguagem explícito. No entanto, em outras linguagens como Python, que é a linguagem que estamos utilizando nos exemplos, as interfaces são implícitas e são conhecidas como “protocolos”.

Falo em ‘acordo’, justamente por isso, não é uma coisa explicita da linguagem como em java que possui uma palavra reservada ‘interface’ para explicitar a interface. Para o Python é apenas uma classe abstrata.

Aqui está um exemplo do que estamos falando:

```
from abc import ABC, abstractmethod

# Interface
class Animal(ABC):
    @abstractmethod
    def emitir_som(self):
        pass

# Classe que implementa a interface
class Cachorro(Animal):
    def emitir_som(self):
        return “Latindo!»

# Classe que implementa a interface
class Gato(Animal):
    def emitir_som(self):
        return “Miando!»
```

Conseguiu perceber? A classe Animal é uma classe abstrata, pois herda ABC.

A classe Animal também é a interface que define um método abstrato emitir_som().

As classes Cachorro e Gato implementam essa interface fornecendo suas próprias implementações do método emitir_som().

As interfaces são úteis porque permitem que você escreva código que não depende de implementações específicas, tornando seu código mais flexível e extensível. Neste exemplo você pode escrever uma função que aceita qualquer objeto que implemente a interface Animal, e essa função será capaz de interagir com qualquer tipo de animal que você criar no futuro, desde que esse novo tipo de animal implemente a interface Animal.

Explicando melhor uma classe abstrata.

abc: é um módulo do python que fornece tudo que é necessário para criar classes abstratas.

ABC: é uma classe definida dentro do módulo que permite criar classes abstratas.

abstractmethod: é um decorador usado para definir métodos abstratos. Esses métodos devem sempre ser implementados nas subclasses.

Beleza, então vamos detalhar melhor o que é um método abstrato.

Métodos Abstratos

Um método abstrato é um método que é declarado na superclasse (ou interface) e que deve ser implementado nas subclasses.

Em Python, métodos abstratos são criados usando o módulo abc (Abstract Base Classes) e a função decoradora @abstractmethod.

Viu só! Bem Tranquilo.

```
from abc import ABC, abstractmethod
# Interface
class Animal(ABC):
    @abstractmethod
    def emitir_som(self):
        pass

# Classe que implementa a interface
class Cachorro(Animal):
    def emitir_som(self):
        return “Latindo!»

# Classe que implementa a interface
class Gato(Animal):
    def emitir_som(self):
        return “Miando!»
```

Neste exemplo, Animal é uma classe abstrata e **emitir_som()** é um método abstrato. Qualquer classe que herde de Animal deve fornecer sua própria implementação do método **emitir_som()**, caso contrário, pode dar problema.

A classe Cachorro herda da classe Animal e fornece sua própria implementação do método **emitir_som()**. Mesma coisa para a classe Gato.

Métodos abstratos são úteis porque permitem que você defina um “contrato” que todas as subclasses devem seguir, tornando seu código mais organizado e previsível.

Ilustração feita por upklyak, disponível no Freepik. Adaptada pelo autor.

Métodos Estáticos

Vou aproveitar para falar também dos Métodos estáticos. Estes são métodos que não estão vinculados a nenhum objeto específico. Eles podem ser chamados sem criar um objeto da classe.

Eu sei, isso pode ser meio estranho agora, mas de qualquer forma é algo que pode ser implementado.

Então, quanto usar?

São uteis quando queremos organizar funções de uso geral e por um contexto representado pela classe.

Para criar um método estático em Python, você deve usar o decorador staticmethod. Aqui está um exemplo:

```
class MinhaClasse:
    @staticmethod
    def meu_metodo():
        print("Barbadinha!")
        print("Este método é estático. ")

MinhaClasse.meu_metodo()
```

Viu só. Consegue perceber a diferença? Esta é a forma de utilizar o método estático. É só chamar o método através da classe e não pelo objeto.

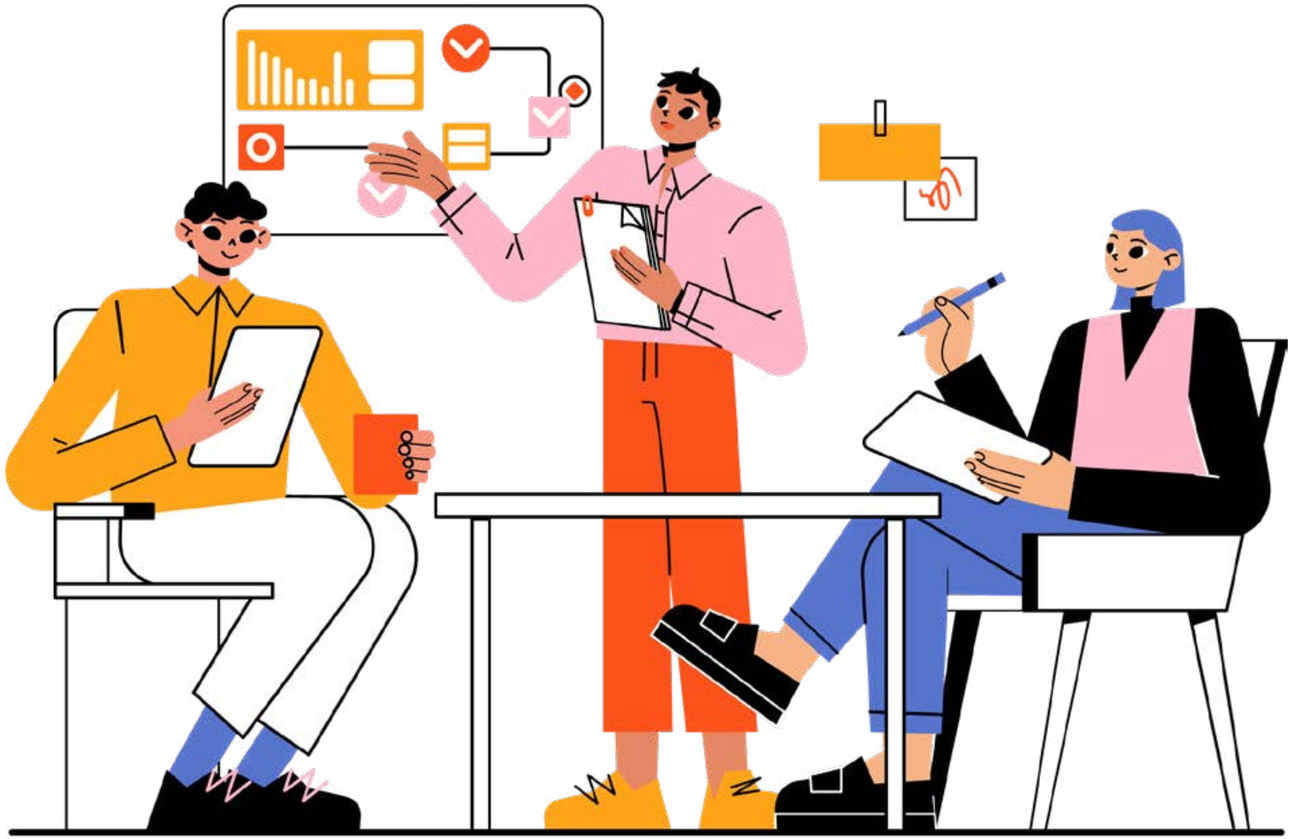

Ilustração feita por upklyak, disponível no Freepik. Adaptada pelo autor.

Conclusão

O encapsulamento, os métodos especiais, as propriedades, a herança e os métodos estáticos são conceitos importantes na programação orientada a objetos. Compreender esses conceitos ajudará a escrever códigos mais robustos e seguros.

Observe o seguinte! Apresentei conceitos importantes para a orientação a objetos, mas como iniciante na programação, não significa que você terá que utilizar tudo isso ao mesmo tempo. Esses conceitos vão se consolidando conforme você vai praticando e desenvolvendo novos projetos. O objetivo desta trilha é te mostrar que isto existe, e apresentar exemplos de aplicações para que você consiga identificar as situações e corrigi-las.

Perceba que a utilização desses conceitos são bem particulares ao projeto que será desenvolvido. Por exemplo, cada novo projeto vai precisar de uma representação de classes diferentes e que o modelo de classe Pessoa, ou Carro utilizados aqui, provavelmente não servirá em um outro projeto, pois isto são coisas bem específicas para cada projeto.

Desenvolver um sistema não é só saber programar (isto é uma etapa importante, mas uma etapa).

Material de Apoio:

Métodos Abstratos e Estáticos (traduzir para português)

https://diveintopython.org/learn/classes/methods

(Procurar item 7 - Herança e polimorfismo)

https://wiki.python.org.br/ProgramacaoOrientadaObjetoPython#A3._Nomes