TIC em trilhas

Programação Orientada a Objetos

Módulo 1

Sumário

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

O que é Programação Orientada a Objetos?

A Programação Orientada a Objetos (POO) é um paradigma de programação para podermos representar elementos do mundo real em uma linguagem de programação, esses elementos são vistos como objetos que possuem atributos (dados ou características) e métodos (ações ou comportamentos).

A POO permite a organização e reutilização do código, e isso em programação é bastante relevante.

Os programas escritos com esse paradigma são mais fáceis de ler e entender, mais fáceis de modificar e manter, e podem ser mais rápidos de programar.

Uma coisa que sempre me perguntam é sobre o tamanho do código, pois em um primeiro momento o código pode parecer maior e a necessidade de aplicar os conceitos em pequenos códigos também pode parece desnecessária, mas acredite, conforme os códigos vão ficando mais robustos, isso é extremamente útil.

Além disso, a programação orientada a objetos ajuda a representar melhor o mundo real em nossos programas.

Comparação com programação procedural.

Como mencionei, POO é um paradigma de programação, outro paradigma de programação é a programação estruturada. Enquanto a programação estruturada(procedural) se concentra em dividir o código em procedimentos autônomos, a POO adota uma abordagem mais orientada a objetos ou funcional.

Existem vários paradigmas de programação, só para exemplificar: Programação funcional, Programação orientada a eventos, Programação concorrentes, etc.

Todos possuem vantagens e desvantagens. Nosso objetivo é tratar do paradigma de orientação a objetos que é um dos mais utilizados.

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

O que são Classes?

Uma classe é uma estrutura que define um conjunto de atributos e métodos que podem ser utilizados para criar objetos. Os objetos são instâncias das classes e possuem características e comportamentos definidos pela classe.

Barbadinha. Pense assim. Um objeto no nosso código é uma variável que não é padrão da linguagem (int, float, bolear, etc). Então, precisamos dizer a linguagem como será a nossa variável. Isso é feito através de um molde que chamamos de classe.

Declarando uma classe.

Então! Precisa de uma variável para representar algo no código? Cria um molde.

Olha esse exemplo: Quero uma variável para representar meu cachorro.

```
#
# Classe Cachorro.
#
class Cachorro:
    def __init__(self, nome, raca, tamanho):
        self.nome = nome
        self.raca = raca
        self.tamanho = tamanho

    def emitir_som(self):
        print(«Latindo!»)

    def alimentar(self):
        print(“Comendo...”)

    def dormir(self):
        print(“Dormindo...”)
```

Viu só. Que barbada. Agora posso dar vários comportamentos pra minha variável.

Criando objetos a partir de uma classe.

Agora que já tenho o modelo de como será minha variável, é só criar a variável (objeto).

```
cachorro1 = Cachorro(«Banze», «Poodle», «Pequeno»)
cachorro2 = Cachorro(«Bolinha», «Labrador», “Grande»)
```

Atributos e Métodos

De forma direta os atributos são as características de um objeto, enquanto os métodos são as ações que um objeto pode realizar.

Atributos: nome, raça, tamanho. São as coisas importantes que queremos que nosso objeto represente. Esses atributos variam conforme o que queremos representar. Por exemplo, se fosse importante representar a idade, esta seria um colocada como atributo.

A mesma coisa com os métodos, se fosse importante representar o cachorro bebendo água, teríamos um método chamado beber_agua(self). Viu só é tranquilo.

Quero destacar dois tipos de atributos.

Atributos de classe e atributos de instância.

O primeiro tipo, são os atributos de classe, estes são compartilhados por todos os objetos da classe, enquanto os atributos de instância são específicos de cada objeto.

Definindo atributos em uma classe.

Os atributos são definidos dentro da classe e podem ser utilizados para qualquer objeto gerado da classe.

Olha esse exemplo:

```
class Cachorro:
    especie = “Canino”

    def __init__(self, nome, raca, tamanho):
        self.nome = nome
        self.raca = raca
        self.tamanho = tamanho
```

O atributo especie é dito de classe. E todos atributos de classes estarão definidos dentro da classe e fora de qualquer método.

Definindo atributos de instância.

Os atributos são definidos dentro do método especial __init()__, e são específicos para o objeto gerado. Claro que todos os objetos gerados terão esses mesmos atributos, mas o conteúdo é somente do objeto específico.

Olha esse exemplo:

```
class Cachorro:
    especie = “Canino”

    def __init__(self, nome, raca, tamanho):
        self.nome = nome
        self.raca = raca
        self.tamanho = tamanho
```

Os atributos nome, raca e tamanho são ditos de instância.

Acessando atributos e chamando métodos.

Os atributos e métodos são acessados usando a notação de ponto.

Exemplo: cachorro1.nome, ou então cachorro1.emitir_som().

Lembrando! Para a nossa classe cachorro, podemos instanciar os objetos da seguinte forma:

```
cachorro1 = Cachorro(«Banze», «Poodle», «Pequeno»)
cachorro2 = Cachorro(«Bolinha», «Labrador», “Grande»)
```

Como acesso o nome do meu cachorro?

Simplesmente digitando:

```
cachorro1.nome
```

Desta forma estou acessando o atributo diretamente e posso fazer o que precisar com isso.

Olha só!

```
nome_cachorro = cachorro1.nome
```

Criei uma variável chamada nome_cachorro cujo conteúdo é “Banze”.

Também posso simplesmente imprimir o nome.

```
print(f”O nome do meu cachorro é: {cachorro1.nome}»)
```

Quando for necessário acessar um método também agimos de forma semelhante:

```
print(f»Meu cachorro: {cachorro1.nome} está
{cachorro1.emitir_som()}”)
```

Chamando o método emitir_som(), estou dizendo que o meu cachorro que chama Banze está Latindo.

Construtor

O construtor é um método especial que é executado quando um objeto é criado a partir de uma classe. O construtor é usado para inicializar os atributos do objeto.

Quando instanciamos um objeto de uma classe, o primeiro código que é executado é o construtor, pois ele vai criar a estrutura do objeto e inicializar os atributos.

No exemplo do cachorro, o __init__() é que faz esse papel.

```
def __init__(self, nome, raca, tamanho):
    self.nome = nome
    self.raca = raca
    self.tamanho = tamanho
```

Lembrando: self.nome é o atributo, nome é uma variável definida no método.

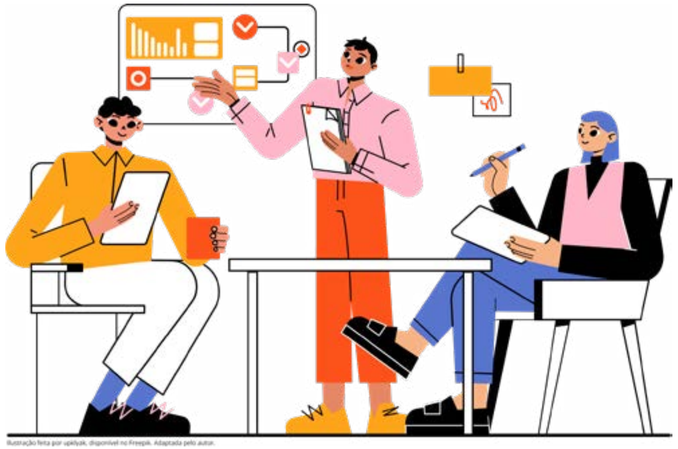

Ilustração feita por upklyak, disponível no Freepik. Adaptada pelo autor.

Encapsulamento

O encapsulamento entra a partir do momento que precisamos “dizer» para a classe o que deve ser utilizando somente internamente pela classe e o que pode ser utilizado de fora da classe.

O encapsulamento é um conceito que permite ocultar a complexidade interna de um objeto e expor apenas o que é necessário para o uso externo.

Quando declaramos os atributos e métodos em uma classe, sem que estejam encapsulados, significa que esses atributos e métodos estão públicos e podem ser acessados e utilizados em qualquer momento e de qualquer parte do código. No mundo real existem situações que nem sempre isso é interessante ou viável.

Com a intenção de ‘proteger› algum dado ou método utilizamos a ideia de encapsular. Neste sentido, para complementar esta ideia existe o conceito de visualização, onde dizemos quais partes do código de uma classe podem ser utilizadas por códigos fora do escopo da classe e quais devem ser utilizadas somente internamente na classe.

Beleza! Vamos melhorar o entendimento disto.

Atributos públicos vs. atributos privados.

Qual a diferença entre eles? Os atributos de uma classe chamados públicos são acessíveis de fora da classe, enquanto os atributos privados só podem ser acessados de dentro da classe. Todos os exemplos anteriores são de atributos e métodos públicos.

Percebendo a diferença na escrita:

Atributos Públicos:

```
def __init__(self, nome, raca, tamanho):
    self.nome = nome
    self.raca = raca
    self.tamanho = tamanho
```

Atributos Privados:

```
def __init__(self, nome, raca, tamanho):
    self.__nome = nome
    self.__raca = raca
    self.__tamanho = tamanho
```

No Caso do Python, self.nome é um atributo público, e self.__nome é um atributo privado. Portanto, quando escrevemos “__” dois underlines na frente de um atributo, significa que este passa a ser privado.

O mesmo se aplica aos métodos. Sempre que iniciar com “__”, significa que é privado.

A grande diferença é a seguinte:

Lembra deste exemplo?

```
print(f»Meu cachorro: {cachorro1.nome} está
{cachorro1.emitir_som()}”)
```

Caso o nome seja privado, essa linha já não funcionaria mais, gerando um erro de execução, pois o nome já não pode mais ser acessado pelo fato de estar declarado assim: self.__nome = nome. Isso significa que estou querendo acessar um atributo que está declarado como privado.

O mesmo se aplicaria ao método, caso tivesse definido como __emitir_som() .

Barbadinha. Viu como fica fácil quando se começa a entender melhor as coisas.

Portanto, assim conseguimos encapsular os dados. Colocando uma barreira(um invólucro, um escudo) em um dado, atributo ou método.

Mas e se eu precisar das duas coisas, protejer um atributo, mas também acessa-lo. Como fazer isso?

Barbada! Colocamos os “__” e criamos um método de acesso onde podemos controlar melhor o que será feito no atributo, fazendo um filtro e deixando passar só que for adequado.

Para isso, normalmente se acrescenta um prefixo ao nome do atributo(set) mais o nome do atributo. Exemplo:

```
def set_nome(self, nome):
    self.__nome = nome
```

Sempre que precisarmos alterar o nome, isto será feito por esse método e, é claro, dentro do método podemos ter mais código, direcionando como for necessário.

O set_nome(nome): é aplicado quando precisamos alterar o conteúdo de um atributo, mas o mesmo se aplica caso eu precise apenas “visualizar» o atributo. Neste caso usando o prefixo get, e o método para isso poderia ficar assim:

```
def get_nome(self):
    return self.__nome
```

Outra forma de encapsular dados é utilizando o property.

Vamos ver como funciona!

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

Usando “property” para encapsular.

```
class Cachorro:

    def __init__(self, nome, raca, tamanho):
        self.__nome = nome
        self.__raca = raca
        self.__tamanho = tamanho

    @property
    def nome(self):
        return self.__nome

    @nome.setter
    def nome(self, novo_nome):
        self.__nome = novo_nome

cachorro = Cachorro("Banze“, "poodle“, “Pequeno”)
cachorro.nome = “Banditti”
print(f”Nome: {cachorro.nome}”)
```

Olha que bacana! Desta forma precisamos definir uma função(normalmente o nome do atributo) colocando acima o decorador @property. Este decorador funciona como o método get_nome(), e normalmente é utilizado juntamente com o decorador @nome.setter mais o nome do método definido no property, funcionando como um set_nome(). Esta é uma forma bastante usual de encapsular os atributos.

Resumindo

Encapsulamento é um dos quatro pilares da programação orientada a objetos. Ele se refere ao agrupamento de dados e métodos em um único objeto. O objetivo do encapsulamento é controlar o acesso a esses dados e métodos, protegendo-os de alterações indesejadas.

Em Python, os atributos de uma classe podem ser públicos ou privados. Atributos públicos são acessíveis a qualquer objeto que esteja usando a classe. Atributos privados são acessíveis apenas a métodos da própria classe.

Para declarar um atributo como privado você pode usar o prefixo ”_”(1 underline) ou “__”(2 underlines).

O prefixo “_“ indica que o atributo é protegido (é permitido alterar seu conteúdo de fora da classe, mas com cuido), enquanto o prefixo “__” indica que o atributo é privado (não é permitido alterar seu conteúdo de fora da classe).

Também falamos sobre os métodos especiais, todos métodos que começam e terminam com dois underlines “__”. Esses métodos são usados pelo Python para realizar tarefas internas, como inicializar um objeto ou retornar uma representação do objeto como uma string.

Alguns exemplos de métodos especiais incluem:

- init(self): Este método é chamado quando um objeto é criado.
- str(self): Este método retorna uma representação do objeto como uma string.
- del(self): Este método é chamado quando um objeto é destruído.

E finalmente, falamos sobre as propriedades que permitem controlar o acesso a atributos de uma classe. Para uma propriedade vimos que é possível utilizar basicamente de dois elementos:

- getter: O getter é um método que é chamado para obter o valor do atributo.
- setter: O setter é um método que é chamado para definir o valor do atributo.

Mostrei também que os getters e setters podem ser utilizados através do decorador @property.

Material de Apoio

Documentação oficial sobre Python pode ser encontrada em:

https://docs.python.org/pt-br/3/reference/index.html

Classe:

https://docs.python.org/pt-br/3/tutorial/classes.html

https://www.w3schools.com/python/python_classes.asp

Herança:

https://docs.python.org/pt-br/3/tutorial/classes.html#inheritance

Métodos:

https://docs.python.org/pt-br/3/tutorial/classes.html#method-objects

Polimorfismo:

https://www.w3schools.com/python/python_polymorphism.asp

Em pdf:

https://wiki.python.org.br/DocumentacaoPython?action=AttachFile&do=view&target=modulo_b.pdf