Desenvolvimento de sistema

Professora Letícia

O que é programação Orientado objeto?

O que é programação Orientado objeto?

é um paradigma de programação que organiza o código em torno de objetos, que são unidades que combinam dados (atributos) e comportamentos (métodos) para representar entidades do mundo real ou conceitos abstratos

O que são objetos?

Na programação, **objetos** são unidades que combinam **dados** e **ações**. Pense neles como representações digitais de coisas do mundo real.

Imagine que você comprou um carro novo. Para modelá-lo usando a programação orientada a objetos, você pode pensar nele como um objeto que possui:

- **Atributos:** As características do carro, como motor 2.0 híbrido, cor azul escura, quatro portas e câmbio automático. Esses são os dados associados ao objeto.
- **Métodos:** Os comportamentos que o carro pode executar, como acelerar, desacelerar, acender os faróis, buzinar e tocar música. Essas são as ações do objeto.

Essencialmente, o carro novo é um **objeto**, onde suas características são seus **atributos** e seus comportamentos são seus **métodos**. Essa combinação de dados e ações em uma única unidade é a base da programação orientada a objetos.

Atributos ou propriedades?

Atributos ou propriedades?

Os atributos ou propriedades se referem às informações do objeto.

Então, por exemplo, se considerar um objeto "celular", suas propriedades serão: cor, marca, modelo, ano de fabricação e daí por diante.

Métodos?

Métodos

Os métodos, por outro lado, são as operações que podem ser feitas no objeto.

Se seguir no exemplo de celular, os métodos podem ser ligar, enviar mensagem, tirar uma foto.

O que são classes?

A classe é um conjunto de características e comportamentos que definem o conjunto de objetos pertencentes à essa classe.

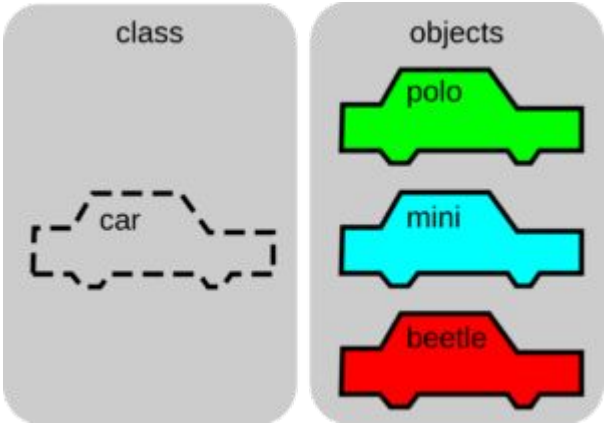
Seu carro é um objeto seu mas na loja onde você o comprou existiam vários outros, muito similares, com quatro rodas, volante, câmbio, retrovisores, faróis, dentre outras partes.

Observe que, apesar do seu carro ser único, podem existir outros com exatamente os mesmos atributos, mas que ainda são considerados *carros*.

Podemos dizer então que seu objeto pode ser classificado (isto é, seu *objeto pertence à uma classe*) como um carro.

Nesse caso, o carro que você comprou nada mais é que uma *instância* dessa *classe* chamada "carro"

O que são classes?



Class em Python

```
class Carro:
    def __init__(self, modelo):
        self.modelo = modelo;
        self.velocidade = 0
    def acelerar(self):
        # Codigo para acelerar o carro
    def frear(self):
        # Codigo para frear o carro
    def acenderFarol(self):
        # Codigo para acender o farol do carro
```

Criando as instâncias (os objetos)

```
# 1. Criando a primeira instância (objeto)
carro_esportivo = Carro(modelo="Ferrari 488")

# 2. Criando a segunda instância (outro objeto)
carro_popular = Carro(modelo="Ford Ka")

# Acessando e usando os objetos
print(f"Instância 1: {carro_esportivo.modelo}")
print(f"Instância 2: {carro_popular.modelo}\n")

# Chamando os métodos para cada objeto
carro_esportivo.acelerar()
carro_esportivo.acelerar()
carro_esportivo.acenderFarol()
print("-" * 20)
carro_popular.acelerar()
carro_popular.frear()
```

Exercício

Criar uma Classe Caneta?

e criar duas instância de objeto uma caneta BIC e outra caneta Faber Castell?

Criar uma Classe Bicicleta?

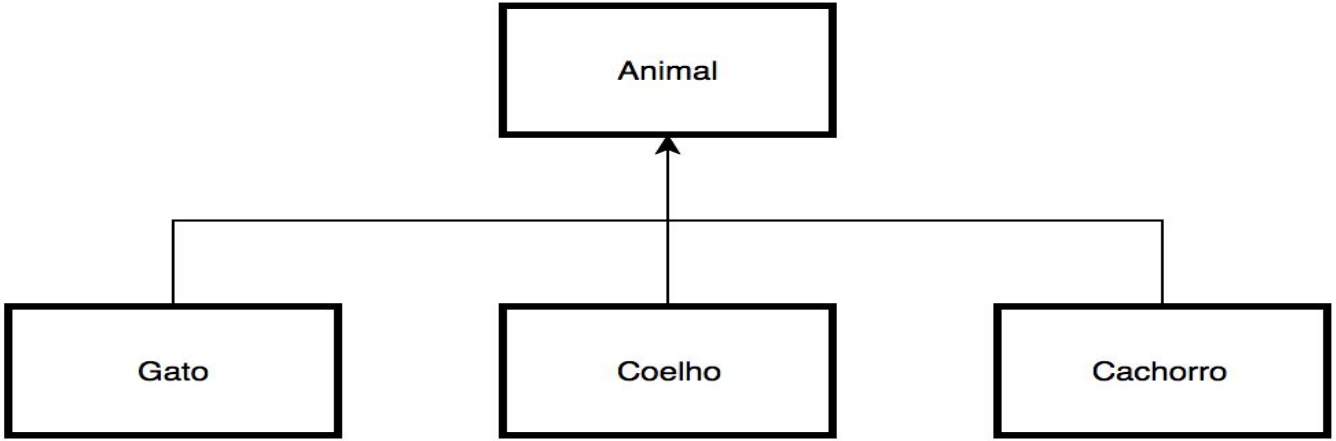
Criar 3 instância de objeto?

4 Pilares da POO

Herança

Herança

é o mecanismo que permite que uma classe (classe filha) herda atributos e métodos de outra classe (classe pai ou superclasse), promovendo a reutilização de código e a criação de hierarquias de classes.


Animal
Gato
Coelho
Cachorro

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

    def comer(self):
        print(f"O {self.__nome} está comendo")
```

```
class Gato(Animal):
    def __init__(self, nome, cor):
        super().__init__(nome, cor)
```

```
class Cachorro(Animal):
    def __init__(self, nome, cor):
        super().__init__(nome, cor)
```

```
class Coelho(Animal):
    def __init__(self, nome, cor):
        super().__init__(nome, cor)
```

super()

Note que as classes filhas só estão repassando seus dados de nome e cor para a classe Pai através do `super()` e que nenhum método foi implementado dentro dessas classes.

Agora, por herdar da classe Animal, as classes Gato, Cachorro e Coelho podem, sem nenhuma alteração, utilizar o método comer(), definido na classe Animal pois elas herdam dessa classe, logo elas possuem a permissão de invocar este método:

```
gato = Gato("Bichano", "Branco")

cachorro = Cachorro("Totó", "Preto")

coelho = Coelho("Pernalonga", "Cinza")

gato.comer()

cachorro.comer()

coelho.comer()
```

Saída

O Bichano está comendo

O Totó está comendo

O Pernalonga está comendo

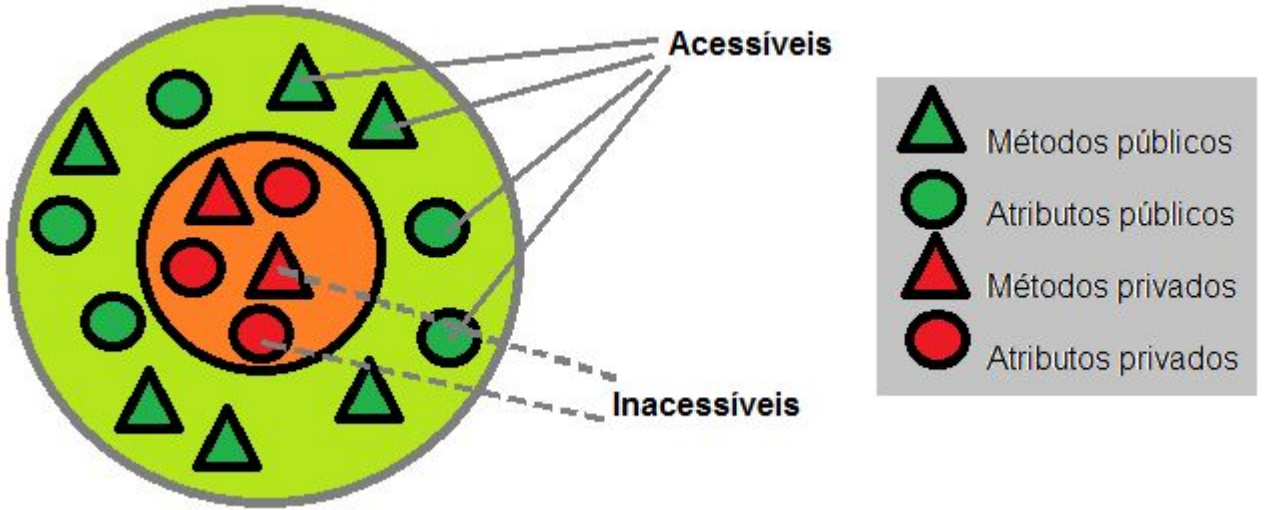
Principais benefícios da herança em Python

- **Reutilização de código:** Ao herdar de uma classe existente, podemos aproveitar os atributos e métodos já implementados, evitando a necessidade de reescrever o mesmo código várias vezes. Isso reduz a duplicação de código e facilita a manutenção.
- **Extensibilidade:** A herança permite que criemos classes especializadas a partir de classes base, adicionando funcionalidades específicas. Isso nos permite estender o comportamento das classes existentes sem modificar seu código original.
- **Polimorfismo:** O polimorfismo é um conceito importante em POO, e a herança em Python possibilita a sua utilização. Com o polimorfismo, podemos tratar objetos de diferentes classes de forma uniforme, desde que essas classes herdem de uma mesma classe base.
- **Organização e estruturação do código:** Ao utilizar a herança, podemos criar hierarquias de classes que refletem a estrutura do problema que estamos resolvendo. Isso torna o código mais organizado, modular e de fácil compreensão.

Encapsulamento

Encapsulamento

O encapsulamento de atributos e métodos impede o *vazamento de escopo*, em que um atributo ou método é visível por alguém que não deveria vê-lo, como outro objeto ou classe.



```python
# Exemplo da classe Carro em Python
class Carro:
    def __init__(self, modelo, mecanismoAceleracao):
        self.__modelo = modelo;
        self.__velocidade = 0
        self.__mecanismoAceleracao = mecanismoAceleracao
    def acelerar(self):
        mecanismoAceleracao.acelerar()
    def frear(self):
        # Codigo para frear o carro
    def acenderFarol(self):
        # Codigo para acender o farol do carro
    def getVelocidade(self):
        return self.velocidade

    def __setVelocidade(self):
        # Codigo para alterar a velocidade por dentro do objeto
    def getModelo(self):
        return self.modelo
    def getCor(self):
        return self.cor
    def setCor(self, cor):
        self.cor = cor
```

Para acessar o Atributo (Getters e Setters)

Só precisamos saber que ele vai nos dar a velocidade certa. Ler ou alterar um atributo encapsulado pode ser feito a partir de *getters e setters* (colocar referência).

Exemplo

```
class ContaBancaria:
    """
    Exemplo de encapsulamento com métodos get/set tradicionais.
    """
    def __init__(self, titular, saldo_inicial):
        self.titular = titular
        # O underscore indica que este atributo não deve ser acessado diretamente.
        self._saldo = saldo_inicial

    def get_saldo(self):
        """Método 'getter' para retornar o saldo."""
        print("Acessando o saldo através do get_saldo()...")
        return self._saldo

    def set_saldo(self, novo_valor):
        """
        Método 'setter' para alterar o saldo.
        Ele contém uma regra de negócio para não permitir valores negativos.
        """
        print(f"Tentando alterar o saldo para R$ {novo_valor:.2f}...")
        if novo_valor >= 0:
            self._saldo = novo_valor
            print("Saldo alterado com sucesso!")
        else:
            print("Erro: O saldo não pode ser negativo.")
```

Public

```
class Pessoa:
    def __init__(self, nome):
        self.nome = nome # Atributo público

p = Pessoa("Ana")
print(p.nome) # Acessando de fora
p.nome = "Ana Silva" # Modificando de fora
print(p.nome)
```

protected

```
class Conta:

    def __init__(self, saldo):

        self._saldo = saldo # Atributo protegido


c = Conta(100)

print(c._saldo) # Funciona, mas é considerado má prática!
```

private

```
class Funcionario:

    def __init__(self, nome, cpf):

        self.nome = nome

        self.__cpf = cpf # Atributo "privado"


    def exibir_dados(self):

        print(f"Nome: {self.nome}, CPF: {self.__cpf}") # Acesso interno funciona
```

```
f = Funcionario("Carlos", "123.456.789-00")

# print(f.__cpf) # ISSO DARÁ UM ERRO! AttributeError


# Jeito de burlar (NÃO FAÇA ISSO, é só para entender como funciona):

print(f._Funcionario__cpf)
```

Tabela de Modificadores

Modificador	Na própria classe	Classes Filhas	Outras Classes
public	✅ Sim	✅ Sim	✅ Sim
protected	✅ Sim	✅ Sim	❌ Não
private	✅ Sim	❌ Não	❌ Não

Exercício

Os Modificadores em Python

1. **Public (`nome`)**: Acesso livre de qualquer lugar.
2. **Protected (`_nome`)**: Um sublinhado. É um aviso: "Ei, não mexa aqui a menos que você seja uma subclasse". O Python não impede o acesso, mas é uma regra de etiqueta entre programadores.
3. **Private (`__nome`)**: Dois sublinhados (**Name Mangling**). O Python dificulta o acesso externo mudando o nome do atributo internamente.

Exemplo Prático: Classe `Funcionario`

Aqui está a implementação do exercício usando o estilo Pythônico com o decorador `@property` (que é a forma elegante de fazer Getters e Setters).

```
class Funcionario:
    def __init__(self, nome, salario):
        self.nome = nome            # Public
        self.__salario = salario    # Private (com dois underlines)

    # GETTER (usando @property)
    @property
    def salario(self):
        return self.__salario

    # SETTER
    @salario.setter
    def salario(self, novo_salario):
        if novo_salario >= 1412:
            self.__salario = novo_salario
            print(f"Salário de {self.nome} atualizado para R$ {novo_salario}")
        else:
            print("Erro: O salário não pode ser menor que o mínimo (R$
1.412,00)!")

# --- Testando o Código ---

func1 = Funcionario("Carla", 2000)

# Acesso Público
print(f"Funcionário: {func1.nome}")
```

```
# Tentando acessar o privado
diretamente (Isso daria erro de
atributo)
# print(func1.__salario)

# Usando o Getter
print(f"Salário Atual: R$
{func1.salario}")

# Usando o Setter com valor inválido
func1.salario = 1000

# Usando o Setter com valor válido
func1.salario = 2500
```

`@property`: Transforma o método em um "atributo virtual". Você chama `func1.salario` em vez de `func1.get_salario()`.

Por que usar `@property` em vez de métodos comuns?

Imagine que você começou um projeto com um atributo público simples: `self.preco = 50`.

Depois de um mês, seu chefe diz: *"Não podemos ter preços negativos!"*.

- **Sem `@property`:** Você teria que mudar `self.preco` para `self.set_preco()` em **todos** os arquivos do sistema. Um pesadelo.
- **Com `@property`:** Você apenas transforma o `preco` em uma property. O resto do código continua escrevendo `objeto.preco = -10`, mas agora o seu Setter vai interceptar e barrar o erro.

- `@salario.setter`: Define a lógica que roda quando você tenta fazer uma atribuição (`func1.salario = valor`).
- **Encapsulamento Real**: Se você tentar acessar `func1.__salario` de fora da classe, o Python dirá que esse atributo não existe. Ele o renomeou para `_Funcionario__salario` por debaixo dos panos.

O "Segredo" do Private em Python (__)

Você deve ter notado que usamos `self.__saldo`. O Python faz algo chamado **Name Mangling** (Desfiguração de Nome).

Quando você coloca `__` na frente de um atributo dentro da classe `Conta`, o Python o renomeia internamente para `_Conta__saldo`.

> **A Verdade Nua e Crua:** O encapsulamento no Python é baseado no "Princípio de Adultos Consentidos". Nada impede alguém de acessar `objeto._Conta__saldo` se ele realmente quiser "quebrar" seu código. O `__` é um aviso luminoso dizendo: **"Não toque aqui, use a Property!"**