Orientação a Objetos na Linguagem Python

1. Herança

Considere a seguinte classe:

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

      def getIdade(self):
           return self.idade
```

Agora, desejamos criar a classe `Cachorro`, que vai herdar os atributos e métodos da classe `Animal`. Para isso devemos passar o nome da classe (`Animal`) na definição da classe Cachorro:

```
class Cachorro(Animal):
      pass
```

Assim, tudo que tiver na classe `Animal` também terá na classe `Cachorro`

```
dog = Cachorro("Rex", "bulldog", 3)
print(dog.getIdade())
```

Dizemos que "Animal" é a **superclasse** (ou **classe pai**) e Cachorro é a **subclasse** ou **classe derivada** ou **classe filho**

Também podemos criar sub-subclasses de subclasses. Veja o exemplo

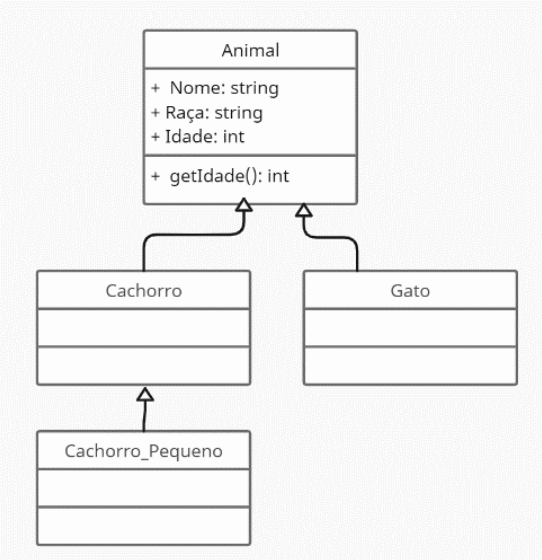
```
class Cachorro_Pequeno(Cachorro):
      pass
```

Agora tudo que tiver na classe Animal e Cachorro também terá na classe Cachorro_Pequeno

```
dog = Cachorro_Pequeno("Rex", "bulldog", 3)
print(dog.getIdade())
```

Agora dizemos que "Cachorro" é a **superclasse** e Cachorro_Pequeno é a **subclasse** ou **classe derivada**.

2. Representação UML da herança



3. Herança Múltipla

Considere o seguinte exemplo de uma classe `Bilingue` que herda da classe `Ingles` e da classe `Portugues`

```
class Ingles():
  def cumprimentar(self):
    print("Hi!")

class Portugues():
  def cumprimentar(self):
    print("Oi!")

class Bilingue(Ingles, Portugues):
  pass

pessoa = Bilingue()
pessoa.cumprimentar()
```

No Python, em casos de herança múltipla, a prioridade é definida da esquerda para a direita. Portanto, no exemplo acima será impresso "Hi!"

4. Polimorfismo

Classes derivadas (classes filhas) são capazes de invocar métodos com mesmo nome, mas com comportamento diferente da classe Pai. Se uma classe filho definir um novo método com o mesmo nome da classe pai, então será executado o método da classe filho.

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

  def falar(self):
    print("au au")

class Gato(Animal):
  def falar(self):
    print("miau miau")

a1 = Animal("Rex", "bulldog", 3)
a1.falar()
a2 = Gato("Bart", "siames", 2)
a2.falar()
```

Outra coisa sobre Polimorfismo: se a classe filho tiver mais atributos que a classe pai, então você deverá obrigatoriamente criar um novo construtor para a classe filho e chamar o construtor da classe pai.

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

  def falar(self):
    print("au au")

class Gato(Animal):
  def __init__(self, nome, raca, idade, qtdeVidas):
    super().__init__(nome, raca, idade)
    self.qtdeVidas = qtdeVidas

  def falar(self):
    print("miau miau")

a2 = Gato("Bart", "siames", 2, 7)
print(a2.qtdeVidas)
```

O comando `super()` permite chamar os métodos da classe pai

5. Atributos Estáticos

São atributos que pertencem à classe como um todo e são compartilhados por todas as instâncias dessa classe. Isso significa que, quando você altera um atributo estático, a mudança é refletida em todas as instâncias da classe. Exemplo:

```
class Carro:
    cor = "preto"  # Atributo de classe

carro1 = Carro()
carro2 = Carro()

print(carro1.cor)  # Saída: preto
print(carro2.cor)  # Saída: preto

Carro.cor = "azul"

print(carro1.cor)  # Saída: azul
print(carro2.cor)  # Saída: azul
```

No exemplo acima, o atributo `cor` é um atributo estático, pois todas as instâncias da classe `Carro` terão o mesmo valor para o atributo `cor`. Mas ele não é estático ao pé da letra, pois o valor dele poderá ser alterado. Portanto, muitas vezes esse atributo também recebe o nome de atributo de classe. Em Python, os termos "**atributo estático**" e "**atributo de classe**" são muitas vezes usados de forma intercambiável. Ambos atributos são compartilhados por todas as instâncias de uma classe, mas a diferença é que um atributo estático pode ser modificado tanto através da classe quanto das instâncias.

```
class Carro:
    def __init__(self, cor):
        self.cor = cor  # Atributo de
instância

carro1 = Carro("preto")
carro2 = Carro("vermelho")

print(carro1.cor)  # Saída: preto
print(carro2.cor)  # Saída: vermelho
```

No exemplo acima, o atributo `cor` é um atributo de classe, e cada instância da classe Carro tem seu próprio valor para esse atributo.

6. Classe Abstrata

Uma classe abstrata é uma classe que não pode ser instanciada diretamente, mas serve como uma classe base para outras classes. Ela pode conter **métodos abstratos, que são métodos sem implementação**, e métodos concretos, que têm implementação. As subclasses devem implementar todos os métodos abstratos da classe abstrata para serem instanciáveis.

A classe abstrata é útil quando você deseja definir uma estrutura comum para um conjunto de classes relacionadas, mas não deseja que a classe base seja instanciada diretamente.

Exemplo de uma classe abstrata em Python usando o módulo abc (Abstract Base Classes):

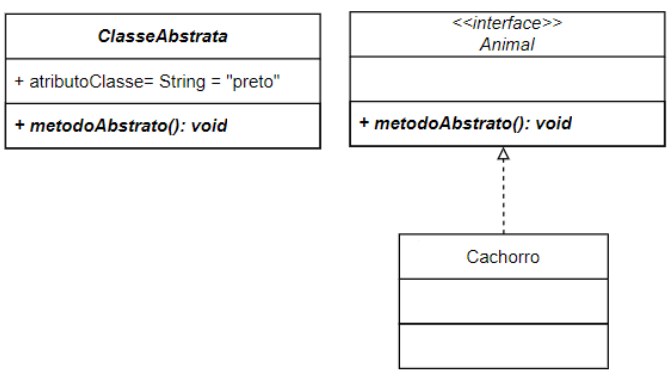
```
from abc import ABC, abstractmethod

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

    @abstractmethod
    def fazer_som(self):
        pass

class Cachorro(Animal):
    def fazer_som(self):
        return "Au au!"

class Gato(Animal):
    def fazer_som(self):
        return "Miau!"

# Tentar instanciar a classe abstrata
diretamente resultará em um erro
# animal = Animal("Fido")

cachorro = Cachorro("Fido")
print(cachorro.fazer_som())  # Saída: Au au!

gato = Gato("Whiskers")
print(gato.fazer_som())  # Saída: Miau!
```

No exemplo acima, Animal é uma classe abstrata que contém um **método abstrato** fazer_som(). As classes Cachorro e Gato são subclasses de Animal e implementam o método fazer_som(). O código tenta instanciar a classe Animal diretamente, o que resultaria em um erro, pois é uma classe abstrata. Em vez disso, as subclasses Cachorro e Gato são instanciadas e usadas normalmente. Se apagar o método fazer_som() de alguma das subclasses (Cachorro ou Gato), você terá erro.

7. Interface

Uma interface é um conjunto de métodos que uma classe deve implementar. É uma forma de definir um **contrato** ou um conjunto de regras que outras classes devem seguir se desejarem ser consideradas do mesmo tipo. Em Python, é possível criar interfaces através de classes abstratas. Porém, na interface **todos** os métodos são abstratos.

8. Representação UML



Exercício resolvido

Vamos criar um pequeno sistema bancário em Python, usando herança.

Primeiro, vamos criar a classe Conta que vai ser a superclasse, o escopo, o esqueleto de toda conta desse banco. Ela tem um atributo, o *saldo*, que representa o valor na conta e os métodos *getSaldo()* que retorna esse valor e o método *depositar()* que aumenta o saldo. Ela inicia o saldo com valor zero.

```
class Conta:
 def __init__(self):
    self.saldo = 0

 def depositar(self, valor):
    self.saldo += valor

 def getSaldo(self):
    return self.saldo
```

Depois, criamos a classe PF para representar as contas de pessoas físicas. Ela é subclasse da Conta e tem um método diferente, o *sacar()*, que faz a movimentação bancária e cobra cinco reais de cada operação.

```
class PF (Conta):
  def sacar(self, valor):
    self.saldo -= (valor + 5)
```

Depois, criamos a classe PJ, para pessoas jurídicas, ela é igual a PF, porém seu método *sacar()* cobra dez reais de cada operação.

```
class PJ (Conta):
  def sacar(self, valor):
    self.saldo -= (valor + 10)
```

Então, fazemos nosso script rodar. Inicialmente ele vai te perguntar se quer abrir uma conta PF ou PJ, dependendo do que o usuário escolher, instanciamos um objeto da classe PF ou PJ. Depois, um looping infinito pergunta se você deseja ver o saldo, depositar ou sacar (até que o usuário digite o número 4).

```
print("1.Criar conta Pessoa Física")
print("2.Criar conta Pessoa Jurídica")
op = int(input("Opção:"))

if op==1:
   conta = PF()
elif op==2:
   conta = PJ()

while True:
   print("1. Ver saldo")
   print("2. Depositar")
   print("3. Sacar")
   print("4. Sair")

   op = int(input("Opção:"))

 if(op == 4):
    break

 if op == 1:
    print("Você tem R$ ", conta.getSaldo())
 elif op==2:
    val = float(input("Valor p/ deposito:"))
    conta.depositar(val)
 elif op==3:
    val = float(input("Valor p/ sacar:"))
    conta.sacar(val)
```

Melhorias no código desenvolvido

1) A classe Conta é uma classe "geral". Ao ir em um banco, nós não criamos uma nova Conta, mas sim uma Conta Corrente ou uma Conta Poupança. Sendo assim, não faz sentido que a classe Conta possa ser instanciada, já que é um erro na regra de negócio caso isso ocorra. É aí que entra o termo "abstrato" desse tipo de classe, por não haver a necessidade de criar objetos com base em uma classe "pai", não há por que ela permitir a instanciação de novos objetos. Ao invés de criarmos um objeto do tipo Conta, só será permitido a criação de objetos do tipo Conta PF ou Conta PJ, o que faz mais sentido. **Como fazer isso?**

2) No cadastro de contas não estamos pedindo o número que será cadastrado na nova conta. No banco, duas contas não podem ter o mesmo número. Para garantir que o número das contas será único, devemos utilizar um atributo universal que gere um número único para cada conta criada. **Como fazer isso?**