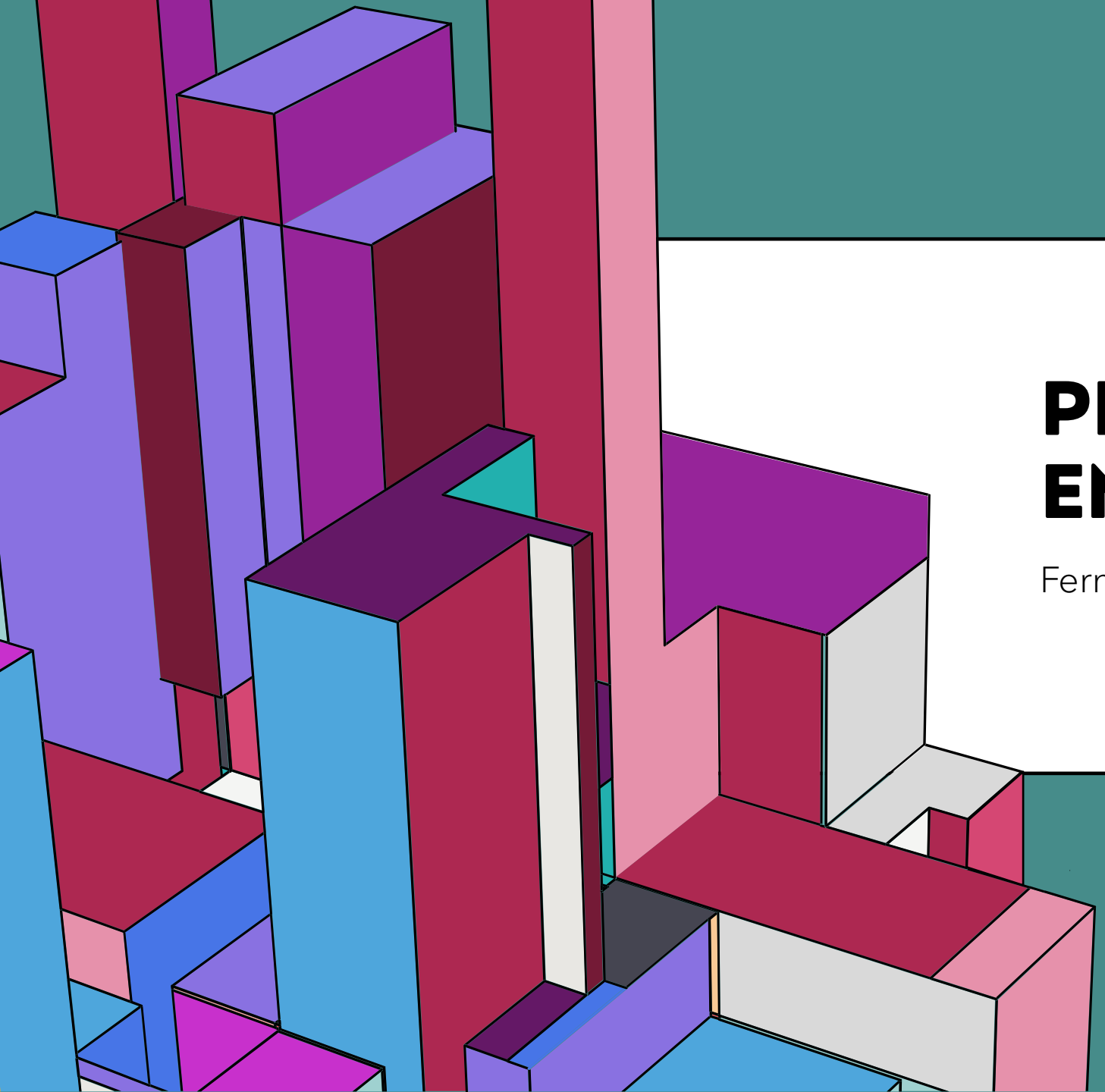


PROGRAMAÇÃO EM PYTHON

Fernando F. Santos

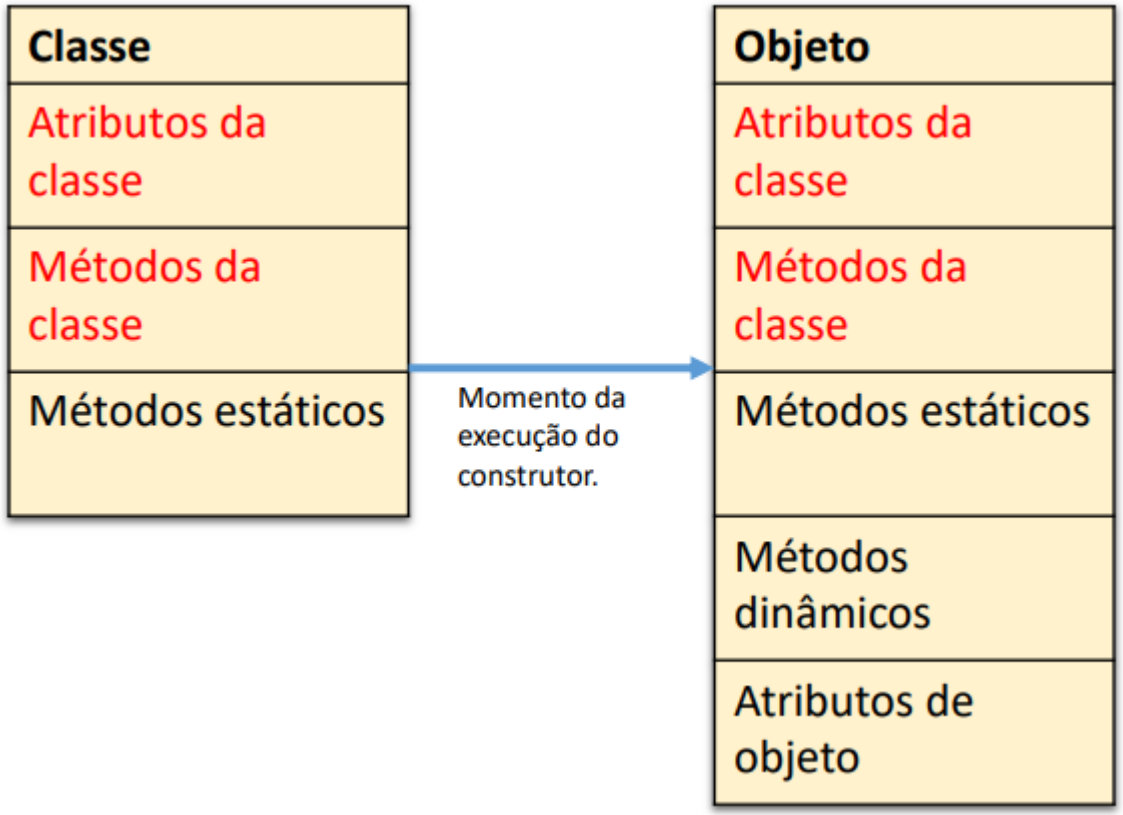
python

SENAI

AULA 13

17/03/2023

OBJETO = CLASSE()



No momento da criação do objeto, é executado o que chamamos de construtor da classe. O construtor é um método especial, chamado __new__(). Após a chamada ao construtor, o método __init__() é chamado para inicializar a nova instância. O método __init__() pode ser usado para passar argumentos , assim podemos passar valores para os atributos da classe.
Os métodos especiais em Python são identificados por nomes no padrão __nome__(). São utilizados dois underscores no início e no fim do nome

INSERINDO O MÉTODO __INIT__

```
# Classe de Herói
class Heroi:
    #Definição do método __init__ da classe:
    def __init__(self, voa, possui_arma,lanca_teia, frase_comum):
        print("Executando init...")
        self.voa = voa
        self.possui_arma = possui_arma
        self.lanca_teia = lanca_teia
        self.frase_comum = frase_comum
    #Definição dos metodos/;
    def falar(self):
        print(self.frase_comum)
    def detalhar(self):
        if self.voa:
            print("O herói voa.")
        if self.possui_arma:
            print("O herói possui arma.")
        if self.lanca_teia:
            print("O herói lança teia.")
```

A única alteração na classe heroi.py.

INSERINDO O MÉTODO __INIT__

```
#Programa Principal
from heroi import Heroi
# Heroi(voa, possui_arma, lanca_teia, frase_comum)
homem_aranha = Heroi(False, False, True, "")
print(homem_aranha.voa)
print(homem_aranha.lanca_teia)
he_man = Heroi(False, True, False, "Eu tenho a força!")
he_man.frase_comum = "Eu tenho a força"
he_man.falar()
homem_aranha.detalhar()
he_man.detalhar()
```

Aqui passamos os valores diretamente pro método __init__

UM POUCO MAIS SOBRE CLASSES E OBJETOS

```
# Quando vazios, as chaves são opcionais, class Classe_simples
class Classe_simples():
    pass
# Qual o tipo do objeto? => <class 'type'>
print(type(Classe_simples))
# obj é uma instância da classe simples
obj = Classe_simples()
# <class '__main__.Classe_simples'>
print(type(obj))
# True, obj é do tipo Classe_simples
print(type(obj) == Classe_simples)
```

CONTINUANDO

```
class Pessoa():
    especie = 'Humano'
print(Pessoa.especie) # Humano
Pessoa.vivo = True # Adicionado dinamicamente
print(Pessoa.vivo) # True
homem = Pessoa()
print(homem.especie) # Humano (herdado)
print(homem.vivo) # True (herdado)
Pessoa.vivo = False
print(homem.vivo) # False (herdado)
homem.nome = 'Fernando'
homem.sobrenome = 'Santos'
print(homem.nome, homem.sobrenome) #Fernando Santos
```

ESCOPO E ESPAÇO DE NOME (NAMESPACE)

Namespace é um espaço ou região dentro do programa, onde um nome (seja uma variável, uma função, etc.) é considerado válido.

Depois que o objeto de classe é criado, ele basicamente representa um namespace. Podemos chamar essa classe para criar suas instâncias. Cada instância herda os atributos e métodos da classe e recebe seu próprio namespace

ESCOPO E ESPAÇO DE NOME (NAMESPACE)

No Python temos basicamente 3 escopos:

1. Escopo local: que contém nomes locais (função atual).
2. Escopo global: escopo do módulo que contém nomes globais, acessado por todas funções do módulo.
3. Built-in names (nomes embutidos): que é o namespace que contém as funções built-in do Python (funções padrões como abs(), cmp(), etc.) e built-in exception names (usados para tratar erros específicos nas cláusulas except de blocos try).

EXEMPLO

```
def funcao_externa():
    b = 20
    a = 80
    print(f"Imprimindo 'b' em funcao_externa: {b}")
    print(f"Imprimindo 'a' em funcao_externa: {a}")
    def funcao_interna():
        c = 30
        b = 25
        a = 70
        print(f"Imprimindo 'c' em funcao_interna: {c}")
        print(f"Imprimindo 'b' em funcao_interna: {b}")
        print(f"Imprimindo 'a' em funcao_interna: {a}")

    funcao_interna()
    print(f"Imprimindo 'b' de novo em funcao_externa: {b}")

a = 10

print(f"Imprimindo 'a' no escopo global: {a}")
funcao_externa()
print(f"Imprimindo 'a' de novo no escopo global: {a}")
```

A variável “a” está no namespace global.

A variável “b” está no namespace local da função “funcao_externa”.

A variável “c” está no namespace local da função “funcao_interna”.

Quando estamos na funcao_interna, “c” é local, “a” é global e “b” é nonlocal. Se tentarmos atribuir um valor a “b”, será criada uma variável local para funcao_interna diferente da b nonlocal.

A mesma coisa acontece ao atribuirmos um valor a variável global “a”.

EXEMPLO 2

```
def funcao_externa():
    b = 20
    global a
    a = 80
    print(f"Imprimindo 'b' em funcao_externa: {b}")
    print(f"Imprimindo 'a' em funcao_externa: {a}")
    def funcao_interna():
        c = 30
        b = 25
        global a
        a = 70
        print(f"Imprimindo 'c' em funcao_interna: {c}")
        print(f"Imprimindo 'b' em funcao_interna: {b}")
        print(f"Imprimindo 'a' em funcao_interna: {a}")

    funcao_interna()
    print(f"Imprimindo 'b' de novo em funcao_externa: {b}")

a = 10

print(f"Imprimindo 'a' no escopo global: {a}")
funcao_externa()
print(f"Imprimindo 'a' de novo no escopo global: {a}")
```

Ao definir a variável "a" dentro das funções especificando "global" será referenciada a variável "a" do escopo global, ao alterar seu valor, será alterado o valor da variável "a" global.

SOMBREAMENTO DE ATRIBUTOS

Quando um atributo em um objeto não é encontrado, o Python continua buscando na classe que foi usada para criar esse objeto (e continua pesquisando até que seja encontrado ou o fim da cadeia de herança seja alcançado). Isso leva a um comportamento de sombreamento.

SOMBREAMENTO DE ATRIBUTOS:

```
class Ponto():
    x = 10
    y = 7
p = Ponto()
print(p.x) # 10 (do atributo da classe)
print(p.y) # 7 (do atributo da classe)
p.x = 12 # p obtém seu próprio atributo "x"
print(p.x) # 12 (encontrado na instância)
print(Ponto.x) # 10 (O atributo da classe ainda é o mesmo)
del p.x # Apagando o atributo da instância
print(p.x) # 10 (Agoa que não existe "x" na instância, será retornado da classe)
p.z = 3
print(p.z) # 3
print(Ponto.z) # O objeto Ponto não tem o atributo "z"
# AttributeError: type object 'Ponto' has no attribute 'z
```

O QUE É O SELF?

Dentro de um método de classe, podemos nos referir a uma instância por meio de um argumento especial, chamado self por convenção. Self é sempre o primeiro atributo de um método de instância.

```
class Quadrado():
    lados = 8
    def area(self): # self é uma referencia a uma instância
        return self.lados ** 2
quadrado = Quadrado()
print(quadrado.area()) # 64 ('lados' foi encontrado na classe)
print(Quadrado.area(quadrado)) # 64 (equivalente a quadrado.area())
quadrado.lados = 10
print(quadrado.area()) # 100 ('lados' foi encontrado na instância)
```

SELF - EXEMPLO 2

```
class Calculo():
    def calcular_total(self, quantidade, desconto):
        return (self.preco * quantidade - desconto)
calc = Calculo()
calc.preco = 15
print(calc.calcular_total(15, 10))
print(Calculo.calcular_total(calc, 15, 10))
```

HERANÇA

Herança nos permite criar uma versão modificada de uma classe existente, adicionando novos atributos e métodos.

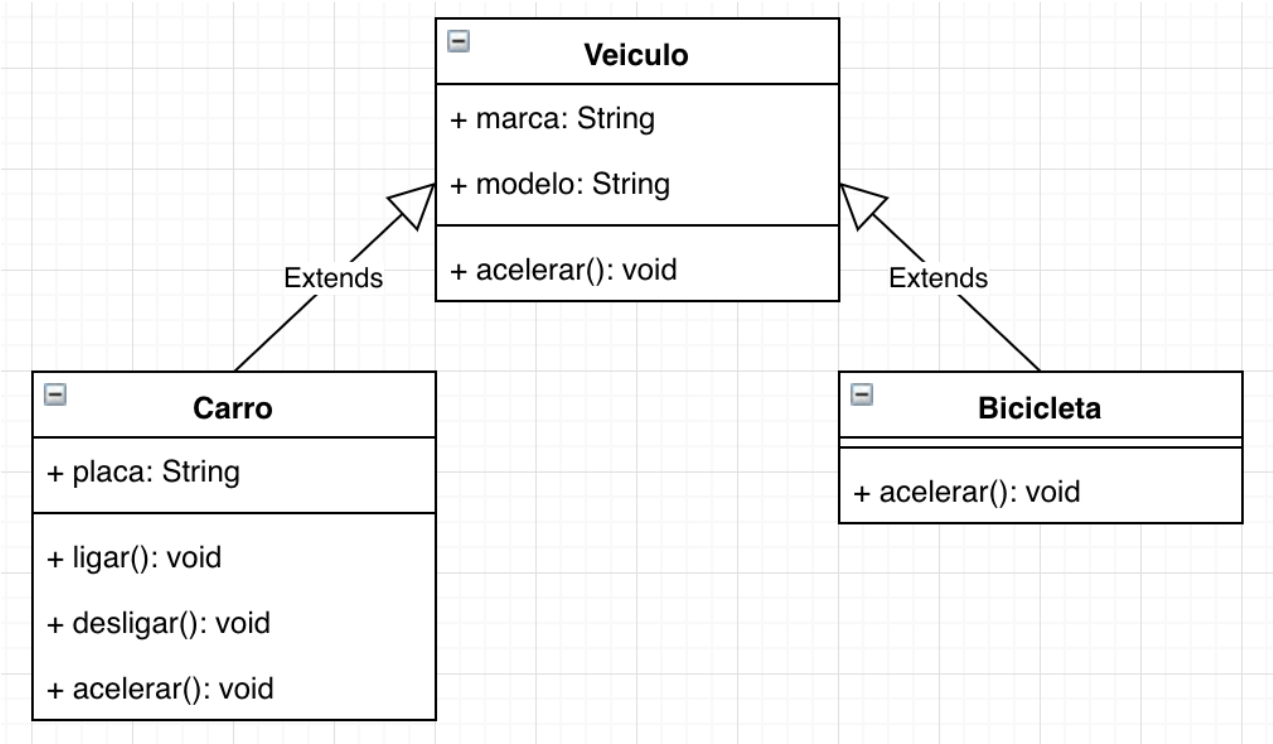
Com herança podemos adaptar o comportamento de classes existentes sem termos que modificá-las.

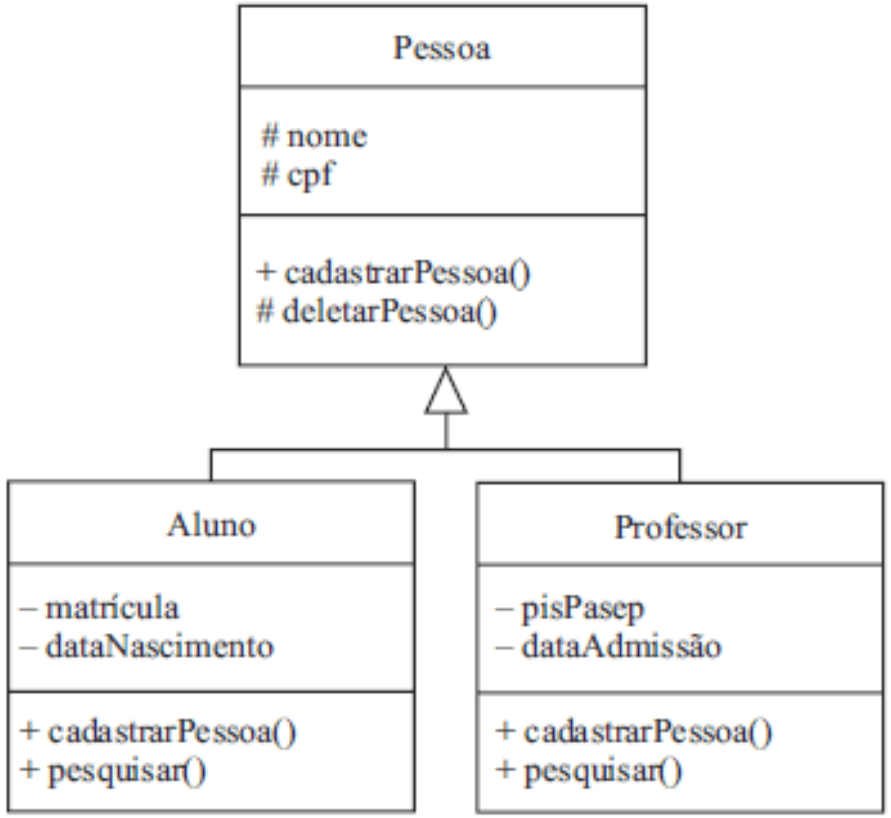
A nova classe herda todos os métodos da classe existente. A classe existente pode ser chamada de classe mãe, classe base ou superclasse e a nova classe, pode ser chamada de classe filha, classe derivada, ou subclasse.

Herança facilita o reuso de código pois podemos adaptar o comportamento de classes existentes, sem ter que modifica-las.

Para localizar os métodos e atributos, o Python procura na classe derivada, retornando pela cadeia de classes base até encontrá-los, similar ao que acontece nos **namespaces** local e global.

EXEMPLO


Veiculo
+ marca: String
+ modelo: String
+ acelerar(): void
Extends
Extends
Carro
+ placa: String
+ ligar(): void
+ desligar(): void
+ acelerar(): void
Bicicleta
+ acelerar(): void


Pessoa
nome
cpf
+ cadastrarPessoa()
# deletarPessoa()
Aluno
– matrícula
– dataNascimento
+ cadastrarPessoa()
+ pesquisar()
Professor
– pisPasep
– dataAdmissão
+ cadastrarPessoa()
+ pesquisar()

EXEMPLO

```python
class Classe_base():
    def __init__(self, valor1, valor2):
        print("Método __init__ da classe base")
        self.valor1 = valor1
        self.valor2 = valor2

    def somar(self):
        return self.valor1 + self.valor2

    def subtrair(self):
        return self.valor1 - self.valor2
```

```python
from classe_base import Classe_base

class Classe_derivada2(Classe_base):
    def multiplicar(self, num1, num2):
        return num1 * num2;
```

```python
from classe_base import Classe_base

class Classe_derivada1(Classe_base):
    def __init__(self, v1, v2):
        print("Método __init__ da classe derivada1.")
        super().__init__(v1, v2)

    def imprimir(self, texto):
        print(texto)
```

```python
from classe_derivada1 import Classe_derivada1
from classe_derivada2 import Classe_derivada2

class Classe_teste():
    calculo = Classe_derivada1(10, 25)
    resultado = calculo.somar()
    print(resultado)
    resultado = calculo.subtrair()
    print(resultado)
    calculo.imprimir("Olá, este é um texto.")

    calc = Classe_derivada2(70,85)
    resultado = calc.multiplicar(20, 10)
    print(resultado)
    resultado = calc.somar()
    print(resultado)
```

O USO DE SUPER()

```
# Criando a classe base
class Pai():
    def __init__(self):
        print('Construindo a classe Pai')
# Classe filha herda da classe pai
class Filha(Pai):
    def __init__(self):
        Pai.__init__(self) # Chamando o construtor da classe pai direto
# Criada classe mãe
class Mae():
    def __init__(self):
        print('Construindo a classe Mãe')
# Mudar a classe filha para herdar de Mae
class Filha(Mae):
    def __init__(self):
        Pai.__init__(self)  # Chamando o construtor
                            # da classe pai direto. E agora?
# Em vez de fixar a classe Pai, melhor seria usar super() para definir
# que o método __init__ chamado é o da classe base.
class Filha(Pai):
    def __init__(self):
        super().__init__() # Chamando o construtor da super classe (classe base)
```

EXEMPLO

```python
class Veiculo:
    def __init__(self, possui_motor, qtd_rodas):
        self.possui_motor = possui_motor
        self.qtd_rodas = qtd_rodas
        self.ligado = False

    def ligar(self):
        if self.possui_motor:
            self.ligado = True
            print("Ligou")
        else:
            print("Não tem motor.")

    def desligar(self):
        if self.possui_motor:
            if self.ligado:
                print("Desligou.")
            else:
                print("Não está ligado")
        else:
            print("Não tem motor.")

    def andar(self):
        print("O veículo começou a andar.")

    def parar(self):
        print("O veículo parou.")
```

```python
from veiculo import Veiculo

class Carro(Veiculo):
    def __init__(self, qtd_rodas):
        super().__init__(True, qtd_rodas)
```

```python
from veiculo import Veiculo

class Bicicleta(Veiculo):
    possui_guidao = True

    def __init__(self, qtd_rodas):
        super().__init__(False, 2)

    def empinar(self):
        print("A bicicleta empinou.")
```

```python
from carro import Carro
from bicicleta import Bicicleta

bike = Bicicleta(2)
print(bike.possui_motor)
print(bike.possui_guidao)
print(bike.qtd_rodas)
bike.ligar()
bike.andar()
bike.empinar()
bike.parar()
bike.desligar()

carro = Carro(4)
print(carro.qtd_rodas)
carro.desligar()
carro.ligar()
carro.andar()
carro.parar()
carro.desligar()
```

EXEMPLO 2

```python
class Pessoa:
    def __init__(self, nome, idade):
        self.nome = nome
        self.idade = idade
```

```python
from pessoa_fisica import Pessoa_Fisica
from pessoa_juridica import Pessoa_Juridica

pf = Pessoa_Fisica("012.345.678-90", "Evaldo", 38)
pj = Pessoa_Juridica("01.000.123/0001-00", "Loja Teste", 5)

print(pf.nome)
print(pf.cpf)
print(pf.idade)
print(pj.nome)
print(pj.cnpj)
print(pj.idade)
```

```python
from pessoa import Pessoa

class Pessoa_Juridica(Pessoa):
    def __init__(self, cnpj, nome, idade):
        super().__init__(nome, idade)
        self.cnpj = cnpj
```

```python
from pessoa import Pessoa

class Pessoa_Fisica(Pessoa):
    def __init__(self, cpf, nome, idade):
        super().__init__(nome, idade)
        self.cpf = cpf
```

EXERCÍCIOS

1. Crie uma classe chamada Ingresso, que possui um valor em reais e um método imprimeValor()
   - Crie uma classe VIP, que herda de Ingresso e possui um valor adicional. Crie um método que retorne o valor do ingresso VIP (com o adicional incluído)
2. Crie uma classe chamada Forma, que possui os atributos area e perimetro.
   - Implemente as subclasses Retangulo e Triangulo, que devem conter os métodos calculaArea e calculaPerimetro. A classe Triangulo deve ter também o atributo altura
   - No código de teste crie um objeto da classe Triangulo e outro da Classe Retangulo. Verifique se os dois são mesmo instancias de Forma (use instanceof) , e calcule a área de cada um.
3. Considere as classes ContaCorrente e Poupanca apresentadas em sala de aula. Crie uma classe ContaImposto que herda de conta e possui um atributo percentualImposto. Esta classe também possui um método calculaImposto() que subtrai do saldo, o valor do próprio saldo multiplicado pelo percentual do imposto. Crie um programa para criar objetos, testar todos os métodos e exibir atributos das 3 classes (ContaCorrente, Poupanca e ContaImposto).