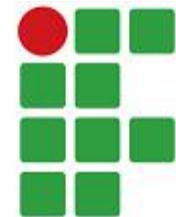


# PROGRAMAÇÃO WEB II

Curso Técnico Integrado em Informática

Lucas Sampaio Leite

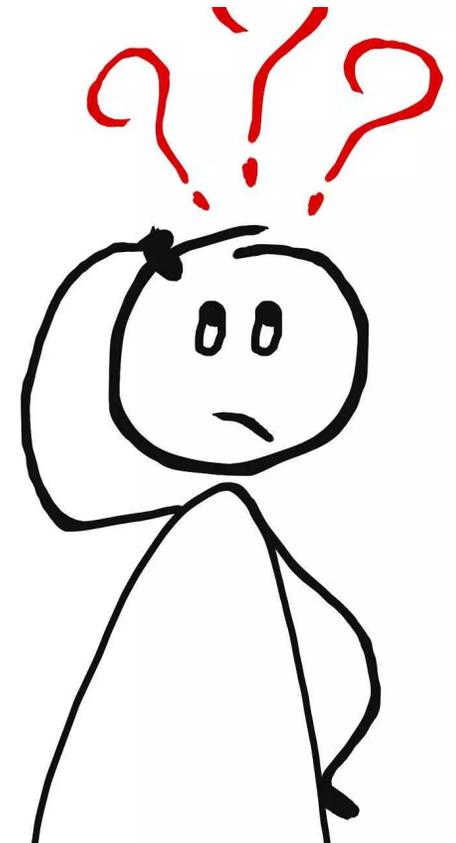


**INSTITUTO  
FEDERAL**

Baiano

# Revisão de Programação Orientada a Objetos com Python

- O que é um paradigma de programação?



# Revisão de Programação Orientada a Objetos com Python

- O que é um paradigma de programação?
  - Um paradigma de programação é um estilo ou abordagem para resolver problemas e estruturar programas em uma linguagem de programação.
  - Ele define um conjunto de conceitos, padrões e práticas que influenciam como os programas são escritos e organizados.

# Revisão de Programação Orientada a Objetos com Python

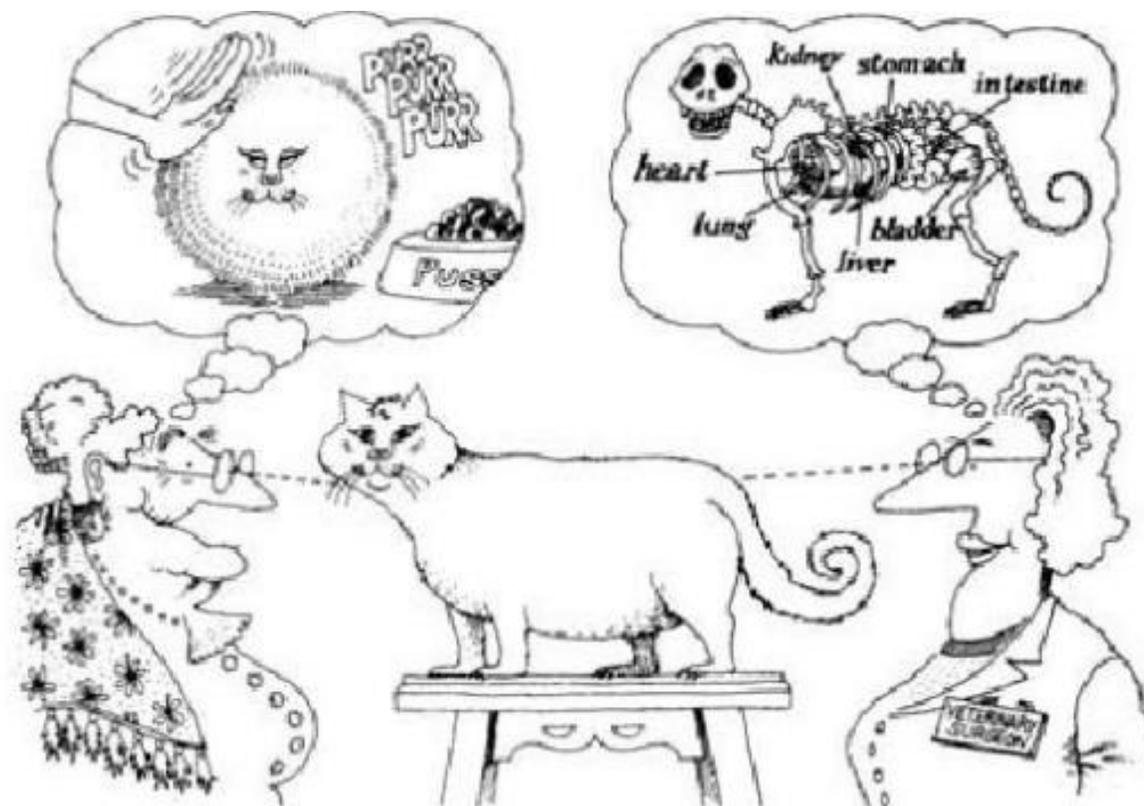
- Alguns paradigmas de programação:
  - Imperativo: Baseado em instruções sequenciais que modificam o estado do programa.
  - Orientado a Objetos (OO): Modelo baseado em objetos, que encapsulam dados e comportamentos.
  - Paradigma Funcional: Baseado em funções puras, imutabilidade e ausência de efeitos colaterais.
  - Baseado em Eventos: A execução do código ocorre em resposta a eventos.
  - Outros paradigmas: paralelo, orientado a aspectos, etc...

# Revisão de Programação Orientada a Objetos com Python

- O paradigma Orientado a Objetos (OO) organiza o código modelando problemas como objetos que representam entidades do mundo real.
- Essa abordagem melhora a modularidade, reutilização e manutenção do código.
- Dois conceitos fundamentais na Programação Orientada a Objetos (POO) são classes, que definem a estrutura e comportamento dos objetos, e objetos, que são instâncias dessas classes.

# Revisão de Programação Orientada a Objetos com Python

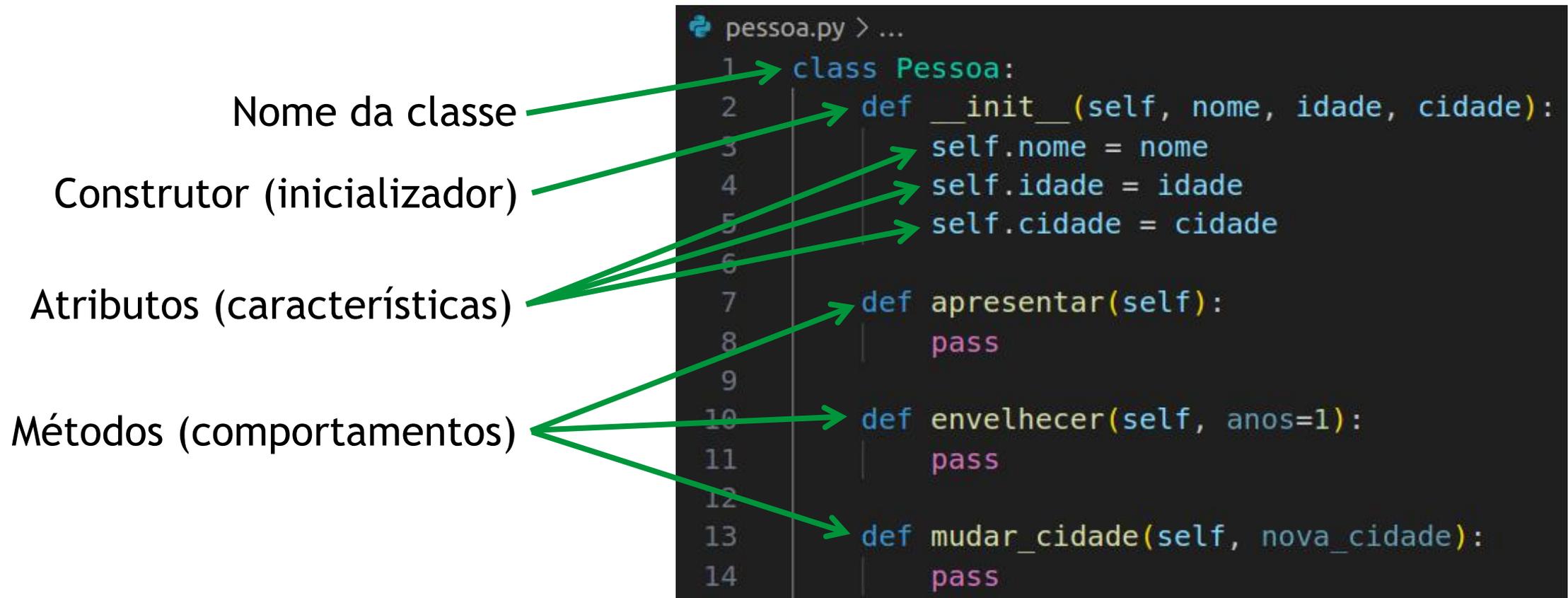
- Abstração: Consiste em identificar e extrair, a partir do domínio do problema, os elementos mais relevantes, representando-os de forma adequada na linguagem da solução



# Revisão de Programação Orientada a Objetos com Python

- Uma classe define as características (atributos) e comportamentos (métodos) que um objeto pode ter.
- Já os objetos são instâncias de uma classe, possuindo as características e comportamentos especificados por ela.

# Revisão de Programação Orientada a Objetos com Python



```
1 class Pessoa:
2     def __init__(self, nome, idade, cidade):
3         self.nome = nome
4         self.idade = idade
5         self.cidade = cidade
6
7     def apresentar(self):
8         pass
9
10    def envelhecer(self, anos=1):
11        pass
12
13    def mudar_cidade(self, nova_cidade):
14        pass
```

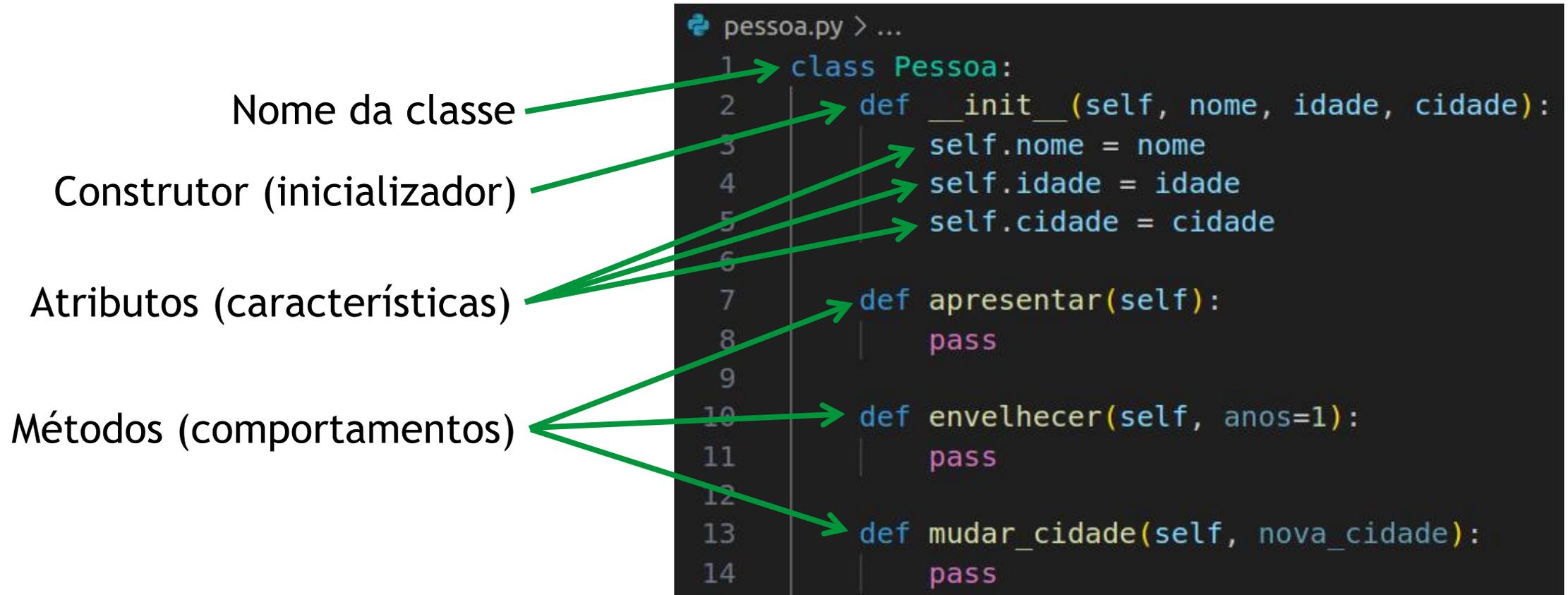
Nome da classe

Construtor (inicializador)

Atributos (características)

Métodos (comportamentos)

# Revisão de Programação Orientada a Objetos com Python



```
1 class Pessoa:
2     def __init__(self, nome, idade, cidade):
3         self.nome = nome
4         self.idade = idade
5         self.cidade = cidade
6
7     def apresentar(self):
8         pass
9
10    def envelhecer(self, anos=1):
11        pass
12
13    def mudar_cidade(self, nova_cidade):
14        pass
```

Nome da classe

Construtor (inicializador)

Atributos (características)

Métodos (comportamentos)

Em Python, o self é uma referência à instância atual da classe. Ele é necessário para que os métodos possam acessar e modificar os atributos da instância.

# Revisão de Programação Orientada a Objetos com Python

- Implementando os comportamentos:

```
peessoa.py > ...  
1 class Pessoa:  
2     def __init__(self, nome, idade, cidade):  
3         self.nome = nome  
4         self.idade = idade  
5         self.cidade = cidade  
6  
7     def apresentar(self):  
8         return f"Olá, meu nome é {self.nome}, tenho {self.idade} anos e moro em {self.cidade}."  
9  
10    def envelhecer(self, anos=1):  
11        if anos > 0:  
12            self.idade += anos  
13            return f"{self.nome} agora tem {self.idade} anos."  
14        return "A idade não pode diminuir."  
15  
16    def mudar_cidade(self, nova_cidade):  
17        self.cidade = nova_cidade  
18        return f"{self.nome} agora mora em {self.cidade}."
```

Acessando atributos

Argumento padrão

# Revisão de Programação Orientada a Objetos com Python

```
peessoa.py > ...
1 class Pessoa:
2     def __init__(self, nome, idade, cidade):
3         self.nome = nome
4         self.idade = idade
5         self.cidade = cidade
6
7     def apresentar(self):
8         return f"Olá, meu nome é {self.nome}, tenho {self.idade} anos e moro em {self.cidade}."
9
10    def envelhecer(self, anos=1):
11        if anos > 0:
12            self.idade += anos
13            return f"{self.nome} agora tem {self.idade} anos."
14        return "A idade não pode diminuir."
15
16    def mudar_cidade(self, nova_cidade):
17        self.cidade = nova_cidade
18        return f"Você mudou de cidade para {self.cidade}."
```

O self sempre deve existir como primeiro parâmetro dos métodos de instância em Python – mas não precisa necessariamente ter esse nome.

# Revisão de Programação Orientada a Objetos com Python

```
main.py > ...  
1  from pessoa import Pessoa  
2  
3  p1 = Pessoa("Lucas", 18, "Recife - PE")  
4  
5  # Testando os métodos  
6  print(p1.apresentar())  
7  print(p1.envelhecer(5))  
8  print(p1.mudar_cidade("Senhor do Bonfim - BA"))
```

Importando a Classe Pessoa

Criando (instanciando)  
um objeto do tipo Pessoa



```
Olá, meu nome é Lucas, tenho 18 anos e moro em Recife - PE.  
Lucas agora tem 23 anos.  
Lucas agora mora em Senhor do Bonfim - BA.
```

# Revisão de Programação Orientada a Objetos com Python

- O método `__str__` em Python é um método especial usado para definir a representação em forma de string de um objeto, ou seja, como o objeto será exibido quando você usar a função `print()` ou `str()` sobre ele.

```
main.py > ...  
1  from pessoa import Pessoa  
2  
3  p1 = Pessoa("Lucas", 18, "Recife - PE")  
4  
5  p1.apresentar()  
6  p1.envelhecer(5)  
7  p1.mudar_cidade("Senhor do Bonfim - BA")  
8  print(p1)
```



Endereço de memória do objeto

```
<pessoa.Pessoa object at 0x7f18d73c2750>
```

# Revisão de Programação Orientada a Objetos com Python

```
def __str__(self):  
    return f"Pessoa(nome='{self.nome}', idade={self.idade}, cidade='{self.cidade}')
```

```
main.py > ...  
1 from pessoa import Pessoa  
2  
3 p1 = Pessoa("Lucas", 18, "Recife - PE")  
4  
5 p1.apresentar()  
6 p1.envelhecer(5)  
7 p1.mudar_cidade("Senhor do Bonfim - BA")  
8 print(p1)
```



```
Pessoa(nome='Lucas', idade=23, cidade='Senhor do Bonfim - BA')  
Pessoa(nome='Lucas', idade=23, cidade='Senhor do Bonfim - BA')
```

# Revisão de Programação Orientada a Objetos com Python

- Construtor `__init__`:
  - Método especial chamado automaticamente ao criar um objeto.
  - Define e inicializa os atributos da instância.
  - Também chamado de inicializador.

```
def __init__(self, nome, idade, cidade):  
    self.nome = nome  
    self.idade = idade  
    self.cidade = cidade
```

Em Python, só existe um método `__init__` por classe – então, tecnicamente, só há um construtor

# Revisão de Programação Orientada a Objetos com Python

- Parâmetros opcionais no `__init__`:
  - É possível definir valores padrão e tratar comportamentos de forma diferentes.

```
def __init__(self, nome="Desconhecido", idade=0, cidade=""):  
    self.nome = nome  
    self.idade = idade  
    self.cidade = cidade
```

```
p1 = Pessoa()  
print(p1)  
p2 = Pessoa("Maria")  
print(p2)  
p3 = Pessoa("Carlos", 25)  
print(p3)  
p4 = Pessoa("João", 40, "Salvador")  
print(p4)
```



```
Pessoa(nome='Desconhecido', idade=0, cidade='')  
Pessoa(nome='Maria', idade=0, cidade='')  
Pessoa(nome='Carlos', idade=25, cidade='')  
Pessoa(nome='João', idade=40, cidade='Salvador')
```

# Revisão de Programação Orientada a Objetos com Python

- Destruitor `__del__`:
  - Método especial chamado quando o objeto é destruído (liberado da memória).
  - Pouco usado em Python (por conta do garbage collector).

```
def __del__(self):  
    print(f"{self.nome} foi removido da memória.")
```

# Revisão de Programação Orientada a Objetos com Python

- Destruitor `__del__`:
  - Método especial chamado quando o objeto é destruído (liberado da memória).
  - Pouco usado em Python (por conta do garbage collector).

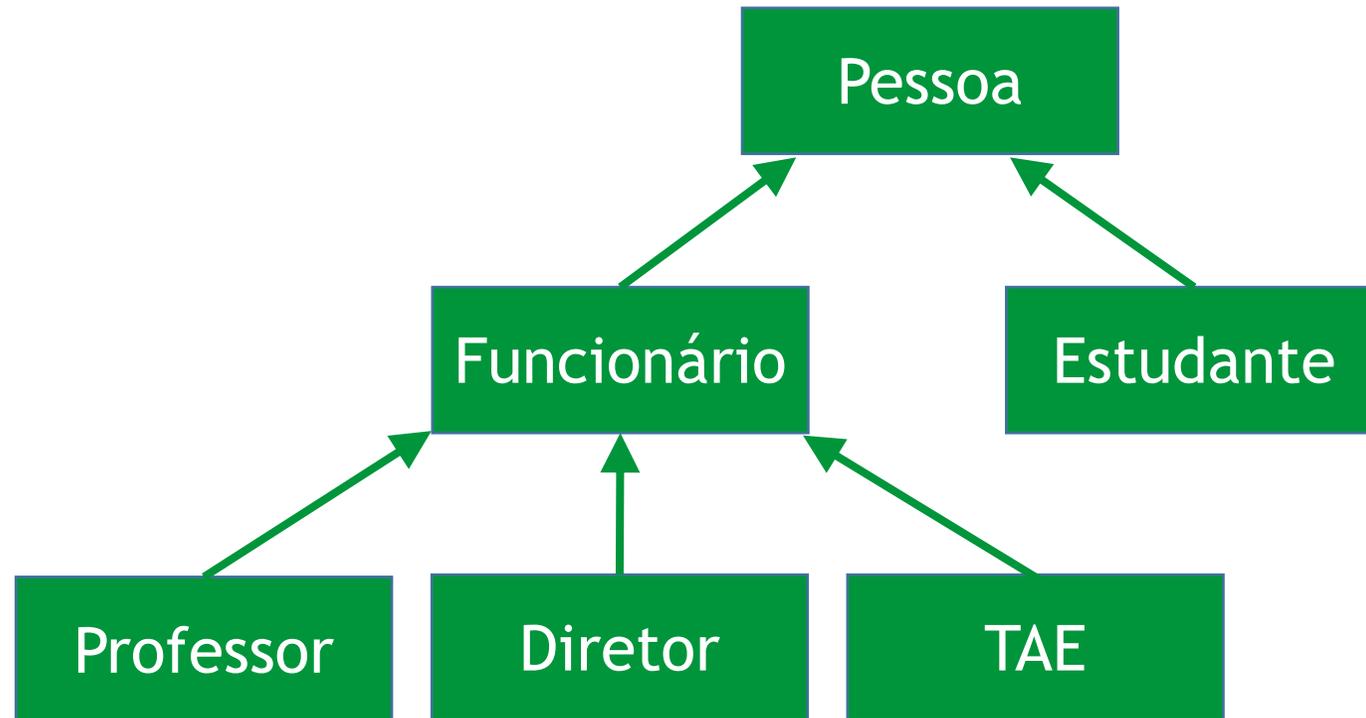
```
def __del__(self):  
    print(f"{self.nome} foi removido da memória.")
```

- Quando utilizado?
  - Encerrar conexões com banco de dados ou sockets.
  - Liberar recursos externos ou temporários (arquivos temporários, memória compartilhada, etc.).
  - Gravar logs ou mensagens de rastreamento quando o objeto for descartado

# Revisão de Programação Orientada a Objetos com Python

- Herança
  - Permite que uma classe filha (subclasse) herde atributos e métodos de uma classe pai (superclasse).
  - Promove reuso de código e facilita a organização hierárquica.
  - É de natureza transitiva, logo, se uma classe B herdar da classe A, todas as subclasses de herdarão automaticamente de A.

# Revisão de Programação Orientada a Objetos com Python

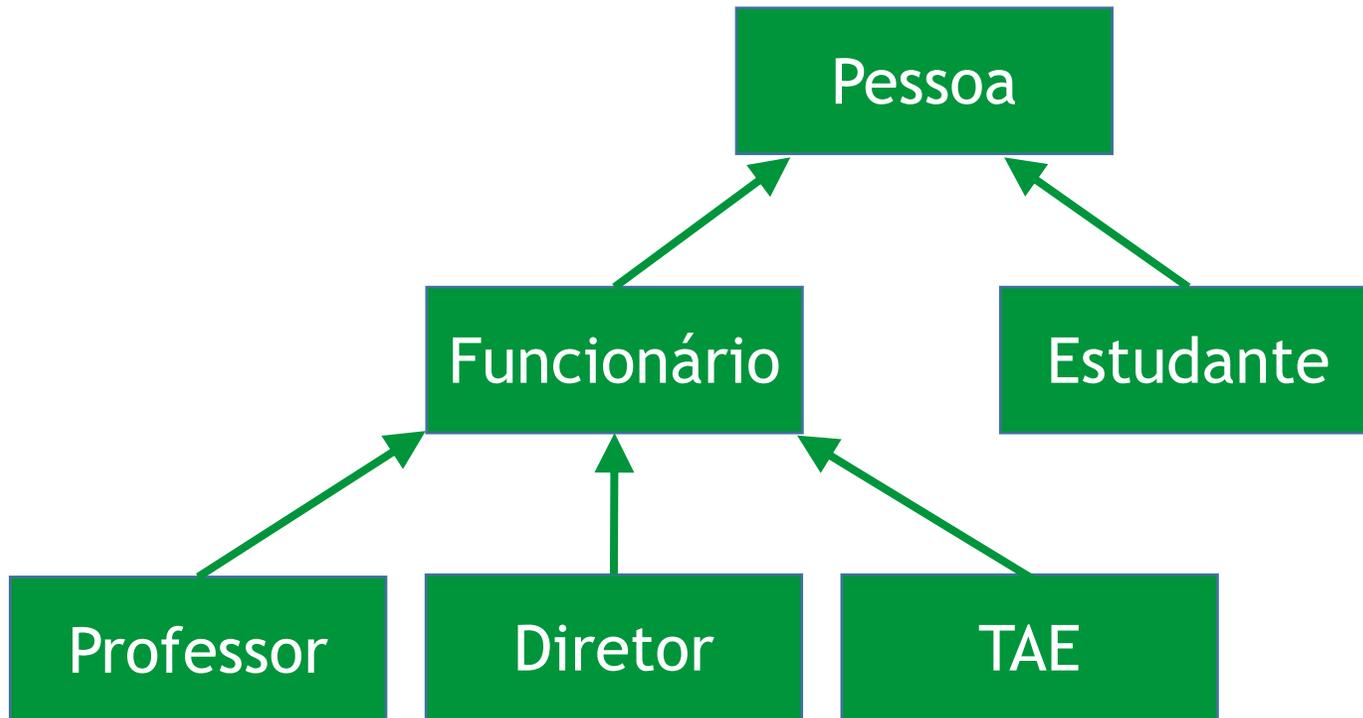


Relação é um!

# Revisão de Programação Orientada a Objetos com Python

- Herança em Python
  - Palavra-chave: `class Subclasse(Superclasse)`
  - A subclasse herda atributos e métodos da superclasse.
  - Pode adicionar novos comportamentos (especialização) ou sobrescrever métodos existentes.

# Revisão de Programação Orientada a Objetos com Python



```
class Pessoa:
|     pass

class Funcionario(Pessoa):
|     pass

class Estudante(Pessoa):
|     pass

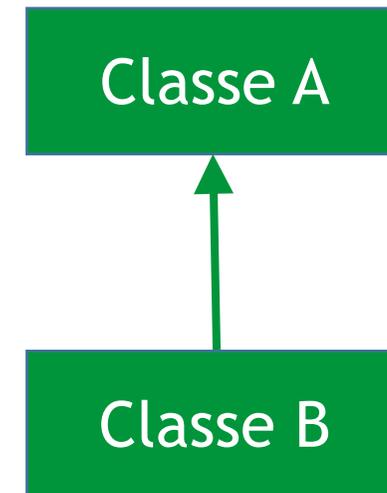
class Professor(Funcionario):
|     pass

class Diretor(Funcionario):
|     pass

class Tae(Funcionario):
|     pass
```

# Revisão de Programação Orientada a Objetos com Python

- Herança simples:
  - Uma subclasse herda de uma única superclasse.
  - Vantagens: simplicidade, legibilidade e reuso.



# Revisão de Programação Orientada a Objetos com Python

```
peessoa.py > ...
1 class Pessoa:
2     def __init__(self, nome, idade, cidade):
3         self.nome = nome
4         self.idade = idade
5         self.cidade = cidade
6
7     def apresentar(self):
8         return f"Olá, meu nome é {self.nome}, tenho {self.idade} anos e moro em {self.cidade}"
9
10    def envelhecer(self, anos=1):
11        if anos > 0:
12            self.idade += anos
13            return f"{self.nome} agora tem {self.idade} anos."
14        return "A idade não pode diminuir."
15
16    def mudar_cidade(self, nova_cidade):
17        self.cidade = nova_cidade
18        return f"{self.nome} agora mora em {self.cidade}."
19
20    def __str__(self):
21        return f"Pessoa(nome='{self.nome}', idade={self.idade}, cidade='{self.cidade}')
```

# Revisão de Programação Orientada a Objetos com Python

Herança Simples

Construtor da superclasse

Sobrescrita de método

Métodos específicos

```
trabalhador.py > ...
1  from pessoa import Pessoa
2
3
4  class Trabalhador(Pessoa):
5      def __init__(self, nome, idade, cidade, cpf, salario):
6          super().__init__(nome, idade, cidade)
7          self.cpf = cpf
8          self.salario = salario
9
10     def apresentar(self):
11         return f"{super().apresentar()}, CPF: {self.cpf}, Salário: {self.salario}"
12
13     def aumentar_salario(self, percentual):
14         if percentual > 0:
15             aumento = self.salario * (percentual / 100)
16             self.salario += aumento
17             return f"Novo salário: R$ {self.salario:.2f}"
18         return "Percentual inválido."
19
20     def salario_anual(self):
21         return self.salario * 12
```

# Revisão de Programação Orientada a Objetos com Python

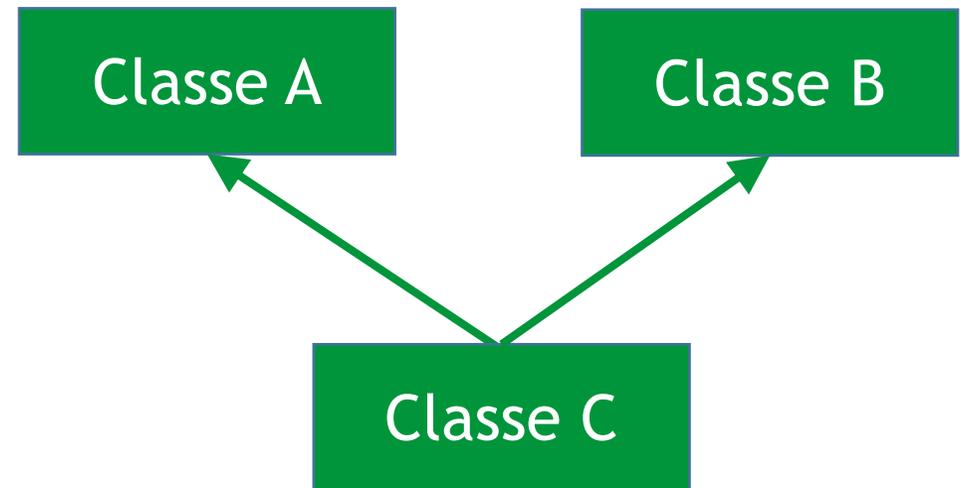
```
main.py > ...  
1  from pessoa import Pessoa  
2  from trabalhador import Trabalhador  
3  
4  p1 = Pessoa("Lucas", 18, "Recife - PE")  
5  p1.envelhecer(5)  
6  print(p1.apresentar())  
7  
8  p2 = Trabalhador("João", 20, "Senhor do Bonfim - BA", "000.000.000-00", 10000.00)  
9  p2.envelhecer(3)  
10 p2.aumentar_salario(10)  
11 print(p2.apresentar())
```



```
Olá, meu nome é Lucas, tenho 23 anos e moro em Recife - PE  
Olá, meu nome é João, tenho 23 anos e moro em Senhor do Bonfim - BA, CPF: 000.000.000-00, Salário: 11000.0
```

# Revisão de Programação Orientada a Objetos com Python

- Herança Múltipla:
  - Uma subclasse herda de duas ou mais superclasses.
  - Potencial de maior reutilização.
  - Conflito de métodos (ex: Método com mesmo nome em duas superclasses)



# Revisão de Programação Orientada a Objetos com Python

- Imagine um veículo que pode dirigir em ruas e voar como um drone – um carro voador autônomo.

```
class Terrestre:
    def mover(self):
        print("Deslocando-se pelas ruas...")

class Aereo:
    def voar(self):
        print("Voando pelo céu...")

class CarroVoadorAutonomo(Terrestre, Aereo):
    def navegar(self):
        print("Sistema autônomo ativado!")
        self.mover()
        self.voar()
```

```
veiculo = CarroVoadorAutonomo()
veiculo.navegar()
```

Qual será a saída?

# Revisão de Programação Orientada a Objetos com Python

- Imagine um veículo que pode dirigir em ruas e voar como um drone – um carro voador autônomo.

```
class Terrestre:
    def mover(self):
        print("Deslocando-se pelas ruas...")

class Aereo:
    def voar(self):
        print("Voando pelo céu...")

class CarroVoadorAutonomo(Terrestre, Aereo):
    def navegar(self):
        print("Sistema autônomo ativado!")
        self.mover()
        self.voar()
```

```
veiculo = CarroVoadorAutonomo()
veiculo.navegar()
```



```
Sistema autônomo ativado!
Deslocando-se pelas ruas...
Voando pelo céu...
```

# Revisão de Programação Orientada a Objetos com Python

- Exemplo com conflito:

```
class Terrestre:
    def iniciar(self):
        print("Ligando motor terrestre...")

class Aereo:
    def iniciar(self):
        print("Ligando turbinas aéreas...")

class CarroVoadorAutonomo(Terrestre, Aereo):
    def iniciar_viagem(self):
        print("Iniciando procedimentos de voo e direção:")
        self.iniciar() # Qual iniciar será chamado?
```

```
veiculo = CarroVoadorAutonomo()
veiculo.iniciar_viagem()
```

Qual será a saída?

# Revisão de Programação Orientada a Objetos com Python

- Exemplo com conflito:

```
class Terrestre:
    def iniciar(self):
        print("Ligando motor terrestre...")

class Aereo:
    def iniciar(self):
        print("Ligando turbinas aéreas...")

class CarroVoadorAutonomo(Terrestre, Aereo):
    def iniciar_viagem(self):
        print("Iniciando procedimentos de voo e direção:")
        self.iniciar() # Qual iniciar será chamado?
```

```
veiculo = CarroVoadorAutonomo()
veiculo.iniciar_viagem()
```



```
Iniciando procedimentos de voo e direção:
Ligando motor terrestre...
```

# Revisão de Programação Orientada a Objetos com Python

- Exemplo com conflito:

```
class Terrestre:
    def iniciar(self):
        print("Ligando motor terrestre...")

class Aereo:
    def iniciar(self):
        print("Ligando turbinas aéreas...")

class CarroVoadorAutonomo(Terrestre, Aereo):
    def iniciar_viagem(self):
        print("Iniciando procedimentos de voo e direção:")
```

```
veiculo = CarroVoadorAutonomo()
veiculo.iniciar_viagem()
```



```
Iniciando procedimentos de voo e direção:
Ligando motor terrestre...
```

O MRO (Ordem de Resolução de Métodos) define a ordem na qual as classes são percorridas quando você chama um método ou acessa um atributo em um objeto que participa de uma hierarquia de herança.

# Revisão de Programação Orientada a Objetos com Python

- Exemplo com conflito:

```
veiculo = CarroVoadorAutonomo()  
veiculo.iniciar_viagem()  
print(CarroVoadorAutonomo.__mro__)
```



```
Iniciando procedimentos de voo e direção:  
Ligando motor terrestre...  
(<class '__main__.CarroVoadorAutonomo'>, <class '__main__.Terrestre'>, <class '__main__.Aereo'>, <class 'object'>)
```

# Revisão de Programação Orientada a Objetos com Python

- Exemplo com conflito:

```
class Terrestre:
    def iniciar(self):
        print("Ligando motor terrestre...")

class Aereo:
    def iniciar(self):
        print("Ligando turbinas aéreas...")

class CarroVoadorAutonomo(Terrestre, Aereo):
    def iniciar_viagem(self):
        print("Iniciando viagem em modo híbrido:")
        Terrestre.iniciar(self) # chamada explícita
        Aereo.iniciar(self)     # chamada explícita
```

```
veiculo = CarroVoadorAutonomo()
veiculo.iniciar_viagem()
```



```
Ligando motor terrestre...
Ligando turbinas aéreas...
```

# Revisão de Programação Orientada a Objetos com Python

- Benefícios da herança:
  - Evita repetição de código.
  - Permite especializar comportamentos.
  - Facilita manutenção e expansão do sistema.
- Cuidados com Herança em Python:
  - Conflito de nomes em herança múltipla.
  - Dificuldade de entender o MRO.
  - Código difícil de manter em hierarquias grandes
- Prefira herança simples sempre que possível

# Exercícios

- Problema - Sistema Bancário com Herança:
  - Implemente um sistema bancário com uma superclasse Conta, que possui os atributos titular e saldo, além dos métodos depositar(valor) e sacar(valor). A partir dela, crie duas subclasses: ContaCorrente e Poupanca. A ContaCorrente permite saques com uma taxa fixa de R\$ 2,00 por operação, enquanto a Poupanca permite saques apenas se houver saldo suficiente e possui um método adicional render\_juros() que aplica um rendimento de 0,5% ao saldo atual. Reescreva o método sacar em ambas as subclasses para respeitar essas regras. Demonstre o uso das classes com exemplos de depósitos, saques e rendimento.

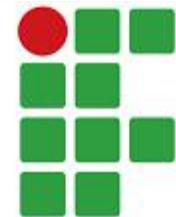
# Dúvidas



# PROGRAMAÇÃO WEB II

Curso Técnico Integrado em Informática

Lucas Sampaio Leite



**INSTITUTO  
FEDERAL**

Baiano