Python

# Programacão Orientada a Objetos

Classes, objetos, estado e comportamento em Python

Prof. Guilherme Soares

# Vamos trocar variáveis soltas por modelos de objetos

Na aula de hoje vamos entender quando usar classes e boas práticas.

1. **Classe vs objeto**
   modelo e instância
2. **Atributos**
   estado do objeto
3. **Metodos**
   comportamento com **self**
4. **Construtor**
   inicializacao com **__init__**
5. **Boas praticas**
   encapsulamento, heranca e dataclasses

# Objeto guarda dados e operações

Em Python, uma classe cria um novo tipo de objeto; instancias guardam estado e usam metodos.

## Antes

```
nome = "Ana"
saldo = 120.0

saldo = saldo - 30
```

## Depois

```
conta = Conta("Ana", 120.0)

conta.sacar(30)
print(conta.saldo)
```

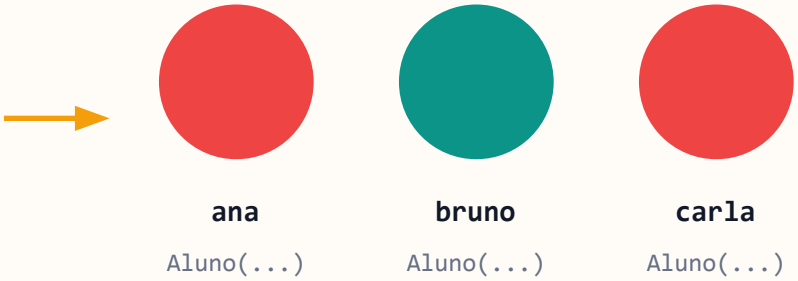
# Classes vs Instâncias

A classe define a forma e o que pode ser feito. Cada objeto tem seus proprios valores.

**class Aluno**

nome
matricula
notas

calcular_media()



# Criando uma classe

A criação de uma classe envolve: definir estado inicial e metodos que operam sobre esse estado.

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

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

conta = Conta("Ana", 100)
conta.depositar(50)
```

### class

Cria um novo tipo definido pelo programador.

### __init__

Método chamado quando o objeto é criado.

### self

Referencia ao proprio objeto que esta executando o metodo.

# Referenciando a própria instância

**SELF** permite que o metodo acesse e altere os atributos da instancia.

# Atributos de instância e de classe não são a mesma coisa

Atributos da instância pertencem ao objeto; atributos da classe são compartilhados.

```
class Aluno:
    escola = "CEFET" # classe

    def __init__(self, nome):
        self.nome = nome # instância
```

### instância

ana.nome e bruno.nome podem ter valores diferentes.

### classe

Aluno.escola é compartilhado como informação do tipo.

**Cuidado: listas/dicionários definidos na classe podem virar estado compartilhado sem querer.**

# Existem várias formas de alterar um atributo

### direto

simples, mas espalha regra pelo código

```
dog.idade = 12
```

### método

centraliza validação e mensagem

```
dog.muda_idade(12)
```

### incremento

expressa a intenção da operação

```
dog.incrementa_idade()
```

```
def muda_idade(self, idade):
  if idade >= 0:
    self.idade = idade

def incrementa_idade(self):
  self.idade += 1
```

# Formas Pythônicas para classes

Conectam a classe a operações da linguagem, como **print**, **len**, **comparação e operadores**.

### __init__
criar objeto

### __str__
texto amigável

### __repr__
representação técnica

### __len__
len(objeto)

### __eq__
objeto == outro

### __lt__
ordenação

```
def __str__(self):
  return f"{self.titular}: R$ {self.saldo:.2f}"

print(conta)
```

ENCAPSULAMENTO

# Python prefere convenção clara a bloqueio rígido

Não existe privado absoluto como em algumas linguagens, mas existe acordo de uso.

### público

parte normal da interface

```
saldo
```

### interno

convenção: uso interno

```
_saldo
```

### name mangling

evita colisão em subclasses

```
__saldo
```

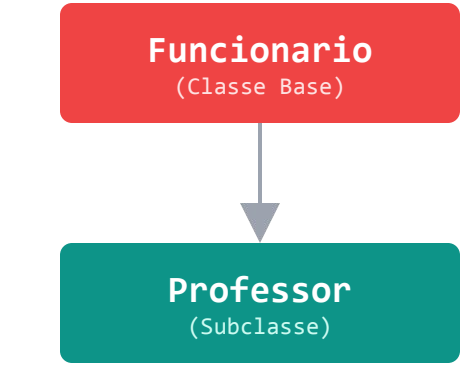
**@property** permite oferecer acesso controlado sem trocar a forma de uso: `objeto.saldo` continua parecendo atributo.

# Herança

Quando queremos especializar uma classe:

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

class Professor(Funcionario):
    def __init__(self, nome, disciplina):
        super().__init__(nome)
        self.disciplina = disciplina
```




Relacionamento: Professor **é um** Funcionário

💡 **Regra de bolso:** se a frase natural for *'tem um'*, considere composição antes de herança.

# Objetos diferentes podem responder a mesma mensagem

A classe filha sobrescreve o método; o código cliente chama todos do mesmo jeito.

```python
class Funcionario:
    def calcular_salario(self):
        return self.salario_base

class Gerente(Funcionario):
    def calcular_salario(self):
        return self.salario_base * 1.20

class Engenheiro(Funcionario):
    def calcular_salario(self):
        return self.salario_base * 1.10
```


💡 **Polimorfismo:** Múltiplas formas de realizar a mesma ação através de uma interface comum.

# Composição monta objetos a partir de outros objetos

Muitas vezes é mais simples, flexível e fácil de testar do que herdar.

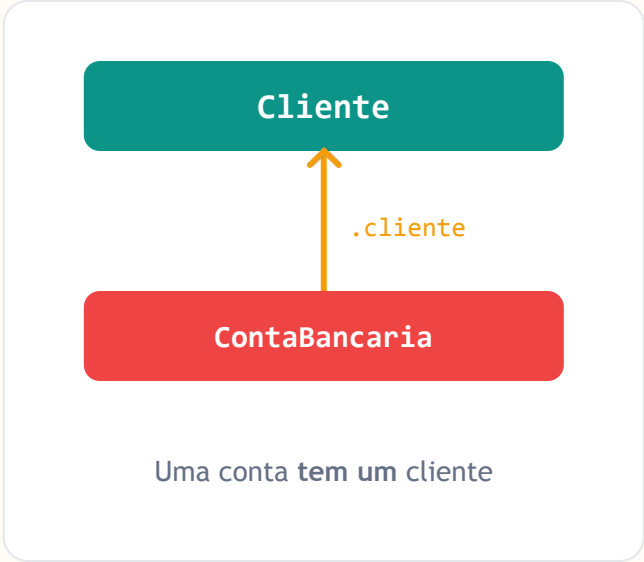

```python
class Curso:
    def __init__(self, professor, disciplina):
        self.professor = professor
        self.disciplina = disciplina
        self.alunos = []
```

# Um objeto pode guardar outro objeto

O exemplo Cliente + ContaBancaria mostra associação: a conta tem um cliente.

```python
class Cliente:
    def __init__(self, nome, cpf):
        self.nome = nome
        self.cpf = cpf

class ContaBancaria:
    def __init__(self, numero, cliente):
        self.numero = numero
        self.cliente = cliente
        self.saldo = 0.0
```




Esse desenho prepara a turma para sistemas com varios objetos colaborando, sem heranca desnecessaria.

# @dataclass reduz código repetitivo

Python gera automaticamente __init__, __repr__ e métodos de comparação para classes de dados.

```python
from dataclasses import dataclass

@dataclass
 class Produto:
    nome: str
    preco: float
    estoque: int = 0

def valor_total(self):
    return self.preco * self.estoque
```

### bom uso

Entidades simples, registros de dados, configurações e modelos de entrada/saída (DTOs).

### cuidado

Não substitui a modelagem rica: invariantes complexas e comportamentos pesados ainda exigem atenção.

# Modularização: Organizando Classes em Arquivos

Para manter o projeto organizado, separe a definição da classe da sua utilização principal.

restaurante.py

```python
# Definição da Classe
class Restaurante:
def __init__(self, nome, cozinha):
self.nome = nome
self.cozinha = cozinha

def descrever(self):
print(self.nome, self.cozinha)
```

main.py

```python
from restaurante import Restaurante

r = Restaurante("Sabor", "mineira")
r.descrever()
```

**Dica de Ouro:** Use módulos quando a classe for reutilizável ou se o arquivo `main.py` estiver misturando muita lógica de modelo, entrada e saída.

MODELAGEM

# Uma boa classe nasce de perguntas simples

Antes de digitar `class`, vale decidir responsabilidade, estado e operações.

### Responsabilidade

Que ideia do problema essa classe representa?

### Estado

Quais informações cada objeto precisa lembrar?

### Comportamento

Que operações pertencem ao objeto?

### Invariantes

O que nunca deveria ficar inválido?

### Colaboradores

Com quais outros objetos ela conversa?

**Atividade rápida:** Pegue um objeto do cotidiano e responda essas cinco perguntas antes de escrever código.

ERROS COMUNS

# OOP em Python costuma quebrar por quatro motivos

A maioria dos bugs aparece quando misturamos estado, responsabilidade e convenções.

### Classe grande demais

Uma classe tentando resolver o sistema inteiro.

### Herança cedo demais

Subclasses criadas antes de existir necessidade real.

### Estado compartilhado

Listas/dicts como atributos de classe por acidente.

### Get/set automático

Criar métodos sem comportamento; só troca a sintaxe.

**Boa pergunta para revisar código:** este método está no objeto que realmente sabe fazer isso?

EXERCÍCIO EM SALA

# Modele um pequeno sistema de biblioteca

Inspirado no material de referência: comece simples e evolua para classes colaborando.

## Requisitos mínimos

1. **Classe Livro:** titulo, autor, disponivel
2. **Classe Usuario:** nome, matricula
3. **Classe Biblioteca:** lista de livros e empréstimos
4. **Metodos:** emprestar(), devolver(), listar_disponiveis()
5. Evitar emprestar livro indisponível

```
# desafio extra

# Teste sua lógica:
biblioteca.listar_disponiveis()
biblioteca.emprestar(u1, l1)
biblioteca.devolver(u1, l1)
```

**entrega**
Arquivo .py rodando no terminal + 3 objetos.

**nível 2**
Modularização: Separar classes em biblioteca.py.

# Referências

Links para consulta posterior e aprofundamento.

**Python Tutorial: Classes**
https://docs.python.org/3/tutorial/classes.html

**Python Data Model: special method names**
https://docs.python.org/3/reference/datamodel.html#special-method-names

**Python Built-in Functions: property() e super()**
https://docs.python.org/3/library/functions.html

**Python dataclasses**
https://docs.python.org/3/library/dataclasses.html

**PEP 8: convenções de estilo e nomes**
https://peps.python.org/pep-0008/

**Real Python: Object-Oriented Programming in Python**
https://realpython.com/python3-object-oriented-programming/