Aula 6 – Introdução à Programação Orientada a Objetos
Parte 1 - "Você é um objeto"

EMC410235 - Programação Científica para Engenharia e Ciência Térmicas

Prof. Rafael F. L. de Cerqueira

2025.2

Motivação

- Em Python, podemos organizar programas como uma sequência de comandos e funções (modo procedural).
- Mas e quando o programa começa a crescer? Quando temos muitos dados inter-relacionados?
- A Programação Orientada a Objetos (POO) permite organizar melhor o código:
  - Reaproveitamento de lógica (herança)
  - Organização dos dados e comportamentos (métodos e atributos)
  - Código mais modular, legível e reutilizável
- Vamos começar modelando algo que vocês conhecem bem: **vocês mesmos!**

Conceitos-chave da Programação Orientada a Objetos

- **Classe**: modelo ou "molde" para criar objetos (ex: Aluno, Professor).
- **Objeto**: instância concreta de uma classe (ex: Vinicius, matriculada em 20231234).
- **Atributos**: características que um objeto possui (ex: nome, matrícula, nota).
- **Métodos**: comportamentos ou ações de um objeto (ex: apresentar(), calcular_media()).
- **Herança**: uma classe pode herdar atributos e métodos de outra (ex: AlunoMSC herda de Aluno).
- **Composição**: objetos podem conter outros objetos (ex: Turma contém Alunos e um Professor).

Exemplo: Classe `Pessoa`

Definindo uma pessoa genérica

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

 def apresentar(self):
  print(f"Olá, eu sou {self.nome}.")
```

- A função `__init__` é o "construtor".
- `self` representa o próprio objeto.
- Atributos: `nome`, `cpf`.
- Método: `apresentar()`.

Classe Aluno: um tipo de Pessoa

Herdando atributos e métodos de Pessoa

```
class Aluno(Pessoa):
 def __init__(self, nome, cpf, matricula):
  super().__init__(nome, cpf)
  self.matricula = matricula
  self.nota = None

 def set_nota(self, nota):
  self.nota = nota

 def resumo(self):
  return f"Aluno {self.nome} (matrícula {self.matricula})"
```

- Usa `super()` para reaproveitar o construtor da `Pessoa`.
- Adiciona novos atributos: `matrícula`, `nota`.
- Define novos comportamentos: `set_nota()`, `resumo()`.

E se não usarmos `super()`?

Repetição desnecessária (evitável)

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

class Aluno(Pessoa):
 def __init__(self, nome, cpf, matricula):
  self.nome = nome    # duplicado
  self.cpf = cpf      # duplicado
  self.matricula = matricula
```

- O código acima funciona, mas repete atributos que já são definidos em `Pessoa`.
- Se mudarmos `Pessoa`, teremos que mudar `Aluno` também!

Tipos de Aluno: Graduação, Mestrado e Doutorado

Herança e sobrescrita de métodos

```
class AlunoGrad(Aluno):
 def resumo(self):
  return f"[Graduação] {super().resumo()}"

class AlunoMSC(Aluno):
 def resumo(self):
  return f"[Mestrado] {super().resumo()}"

class AlunoDSC(Aluno):
 def resumo(self):
  return f"[Doutorado] {super().resumo()}"
```

- Cada classe herda de `Aluno` e especializa o método `resumo()`.
- Ilustra o conceito de **polimorfismo**: mesmo método, comportamentos diferentes.
- Reutilização via `super().resumo()`.

Criando e usando objetos de diferentes tipos

Instanciando alunos

```
a1 = AlunoGrad("Ana", "111.111.111-11", 20231234)
a2 = AlunoMSC("Bruno", "222.222.222-22", 20241235)
a3 = AlunoDSC("Carlos", "333.333.333-33", 20251236)

alunos = [a1, a2, a3]

for aluno in alunos:
 print(aluno.resumo())
```

- Todos os objetos estão na mesma lista: isso é possível porque todos são do tipo `Aluno`.
- Cada objeto executa seu próprio `resumo()`, mesmo com a mesma chamada.
- Isso demonstra **polimorfismo em ação**.

Classe `Professor`: outro tipo de Pessoa

Definindo o professor da turma

```
class Professor(Pessoa):
 def __init__(self, nome, cpf, siape):
  super().__init__(nome, cpf)
  self.siape = siape

 def apresentar(self):
  print(f"Olá, sou o professor {self.nome} (SIAPE {self.siape}).")
```

- Herda atributos de `Pessoa`.
- Adiciona o atributo específico `siape`.
- Sobrescreve o método `apresentar()`.

Classe Turma: composição de objetos

```
class Turma:
 def __init__(self, codigo, professor):
  self.codigo = codigo
  self.professor = professor
  self.alunos = []

 def adicionar_aluno(self, aluno):
  self.alunos.append(aluno)

 def listar(self):
  print(f"Turma {self.codigo} - Professor: {self.professor.nome}")
  for aluno in self.alunos:
   print(aluno.resumo())

 def media(self):
  notas_validas = [a.nota for a in self.alunos if a.nota is not None]
  if notas_validas:
   return sum(notas_validas) / len(notas_validas)
  return None
```

Exemplo final: Integrando tudo

Criando uma turma com alunos e professor

```
prof = Professor("Rafael", "000.000.000-00", "1234567")

a1 = AlunoGrad("Ana", "111.111.111-11", 20231234)
a2 = AlunoMSC("Bruno", "222.222.222-22", 20241235)
a3 = AlunoDSC("Carlos", "333.333.333-33", 20251236)

turma = Turma("EMC410235", prof)
turma.adicionar_aluno(a1)
turma.adicionar_aluno(a2)
turma.adicionar_aluno(a3)

turma.listar()
```

- Demonstração completa do uso de objetos interagindo.
- Ilustração de um sistema simples e bem organizado.

Estrutura de Diretórios

- Até o momento, toda a implementação foi realizada em um único script `.py`.
- Agora, precisamos organizar o programa em diferentes pastas e arquivos, de modo que possamos criar uma aplicação

- `app.py` – Arquivo principal que executa o programa.
- `src/` – Pacote com os módulos da aplicação.
  - `__init__.py` – Indica que `src` é um pacote Python
  - `aluno.py` – Classe `Aluno`
  - `aluno_tipos.py` – Enumeração de tipos de aluno – Classes `AlunoGrad`, `AlunoMSC`, `AlundoDSC`
  - `pessoa.py` – Classe `Pessoa`, base para `Aluno` e `Professor`
  - `professor.py` – Classe `Professor`
  - `turma.py` – Classe `Turma`

Para que serve o `__init__.py`?

- Indica ao Python que a pasta deve ser tratada como um **pacote**.
- Necessário para permitir **imports entre módulos** da pasta, como:

```
from src.aluno import Aluno
```

- Mesmo um arquivo `__init__.py` **vazio** já é suficiente.
- Permite uso de **imports relativos** dentro do pacote:

```
# dentro de professor.py
from .pessoa import Pessoa
```

- Sem ele, o Python pode não reconhecer a pasta como parte de um pacote, gerando erros de importação.

Exemplo de Imports Internos no Pacote

src/aluno.py

```
from .pessoa import Pessoa
from .aluno_tipos import TipoAluno
```

src/aluno_tipos.py

```
from enum import Enum
```

src/professor.py

```
from .pessoa import Pessoa
```

Exemplo de Uso em app.py

Importação das classes

```
from src.professor import Professor
from src.aluno_tipos import AlunoDSC, AlunoMSC, AlunoGrad
from src.turma import Turma
```

- Cria instâncias de `Professor` e `Aluno`
- Adiciona os alunos em uma `Turma`
- Exibe a turma com `print(turma)`

app.py

```
from src.professor import Professor
from src.aluno_tipos import AlunoDSC, AlunoMSC, AlunoGrad
from src.turma import Turma

professor = Professor("Rafael", "444.444.444-44", 2023786)
turma_prog_term = Turma('EMC410235', professor)
a1 = AlunoGrad("Ana", "111.111.111-11", 20231234)
a2 = AlunoMSC("Bruno", "222.222.222-22", 20241235)
a3 = AlunoDSC("Carlos", "333.333.333-33", 20251236)

alunos = [a1, a2, a3]
for aluno in alunos:
  turma_prog_term.adicionar_aluno(aluno)

turma_prog_term.listar()
```

Tarefa de Casa - Criando e Gerenciando a Turma

- Você receberá um arquivo chamado `turma_EMC410235.csv`
- Ele contém os dados da turma, com informações sobre os participantes.
- Sua tarefa será:
  1. Ler o arquivo linha por linha.
  2. Identificar professor e alunos (Graduação, Mestrado ou Doutorado).
  3. Criar objetos correspondentes usando as classes que implementamos.
  4. Adicionar esses objetos em uma instância da classe `Turma`.
- Use os conceitos de:
  - Herança (`Aluno`, `AlunoGrad`, `Professor`)
  - Composição (`Turma` contendo alunos e professor)
- Ao final, imprima o resumo da turma com `turma.listar()`.