Revisão da Aula Anterior — Fundamentos de POO com Python

Conceitos básicos revisados:

- **Classe** → é um molde, um modelo.
- **Objeto (Instância)** → é algo real criado a partir da classe.
- **Atributos** → características do objeto.
- **Métodos** → comportamentos ou ações que o objeto pode executar.

Exemplo revisado (Sistema de Biblioteca):

```
# Classe Livro
class Livro:
    def __init__(self, titulo, autor, ano_publicacao):
        self.titulo = titulo
        self.autor = autor
        self.ano_publicacao = ano_publicacao
        self.status = 'disponivel'

    def emprestar(self):
        if self.status == 'disponivel':
            self.status = 'emprestado'
        else:
            print('Livro indisponível')

    def devolver(self):
        self.status = 'disponivel'

    def detalhes(self):
        return f'Título: {self.titulo}, autor: {self.autor}, ano:
{self.ano_publicacao}, status: {self.status}'
```

Revisão dos conceitos práticos:

Criar objeto:

```
livro1 = Livro('Dom Quixote', 'Cervantes', 1605)
```

Usar métodos:

```
livro1.emprestar()
print(livro1.detalhes())
```

Alterar atributo diretamente:

```
livro1.titulo = 'Dom Quixote - Volume 2'
```

Discussão:
Perceba que qualquer pessoa pode alterar os atributos diretamente, até de forma incorreta, como:

```
livro1.status = 'voando'  # Isso não faz sentido!
```

Isso já abre caminho para falarmos de **Encapsulamento**.

Novo Conteúdo: Pilares da Programação Orientada a Objetos

1. Encapsulamento — Proteção dos Dados

Motivação real:

Imagine que você tem um sistema bancário. Você cria a seguinte classe:

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

    def sacar(self, valor):
        if valor <= self.saldo:
            self.saldo -= valor
        else:
            print("Saldo insuficiente")
```

Agora, qualquer um pode fazer isso:

```
conta = Conta('Ana', 1000)
conta.saldo = -999999  # Isso quebrou completamente o sistema
bancário!
```

- Isso é um **problema gravíssimo!**
- Não podemos permitir acesso direto a dados sensíveis.

➡️ **Solução:** aplicar **Encapsulamento**, que é ocultar os dados e controlar o acesso por meio de métodos.

Encapsulamento aplicado:

```python
class Conta:
    def __init__(self, titular, saldo):
        self.titular = titular
        self.__saldo = saldo  # __saldo torna-se privado

    def sacar(self, valor):
        if valor <= self.__saldo:
            self.__saldo -= valor
        else:
            print("Saldo insuficiente")

    def depositar(self, valor):
        if valor > 0:
            self.__saldo += valor

    def ver_saldo(self):
        return self.__saldo
```

🔍 **O atributo __saldo não pode ser acessado diretamente.**
➡️ Só pode ser alterado de forma segura através dos métodos.

🧬 **2. Herança — Reaproveitamento de Código**

📖 **Conceito:**

Permite que uma classe herde atributos e métodos de outra.

✅ **Exemplo — Petshop:**

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

    def emitir_som(self):
        print("Som genérico")

    def detalhes(self):
        return f'{self.nome}, {self.idade} anos'
```

Criando classes que herdam de Animal:

```
class Cachorro(Animal):
    def emitir_som(self):
        print("Au au")

class Gato(Animal):
    def emitir_som(self):
        print("Miau")
```

➡️ A classe Cachorro não precisou reescrever detalhes(), herdou da classe Animal.

🦸 **3. Função super() — Herdar e complementar**

✅ **Por que usar?**

Às vezes a subclasse precisa aproveitar o que tem na superclasse, mas adicionando coisas.

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

class Cachorro(Animal):
    def __init__(self, nome, idade, raca):
        super().__init__(nome, idade)  # Chama o construtor da classe
Animal
        self.raca = raca

    def detalhes(self):
        return f'{self.nome}, {self.idade} anos, raça {self.raca}'
```

Testando:

➡️ super() permite reaproveitar e estender os atributos e métodos da classe mãe.

🎭 **4. Polimorfismo — Muitos Formatos, Mesmo Nome**

📖 **Conceito:**

Permite que diferentes classes implementem métodos com o **mesmo nome**, mas com **comportamentos diferentes**.

Exemplo no Petshop:

```
animais = [
    Cachorro('Rex', 5, 'Labrador'),
    Gato('Mimi', 2)
]

for animal in animais:
    animal.emitir_som()  # Cada um executa seu próprio emitir_som()
```

🔸 **Mesma chamada de método (emitir_som()), mas comportamento diferente dependendo do objeto.**

🧠 **5. Abstração — Esconder a Complexidade (Conceitual)**

📖 **Conceito:**

Mostra só o que é necessário, escondendo detalhes internos.

➡️ **Na prática, usamos a abstração quando criamos classes e métodos que representam conceitos do mundo real, sem precisar que quem use saiba como funciona por dentro.**

🔥 **Exemplo simples:**

O cliente de um petshop não precisa saber como o sistema calcula, processa ou armazena dados, ele só vê:

```
petshop.cadastrar_animal()
petshop.listar_animais()
petshop.realizar_atendimento()
```

Ele **não sabe nem precisa saber** como esses métodos funcionam por dentro.
Isso é **abstração**.

🏗️ **Aplicação Final — Sistema do PetShop**

🔥 **Classe Animal e Subclasses**

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

    def emitir_som(self):
        print('Som genérico')
```

```
    def detalhes(self):
        return f'{self.nome}, {self.idade} anos'

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

    def emitir_som(self):
        print('Au au')

    def detalhes(self):
        return f'{self.nome}, {self.idade} anos, raça {self.raca}'

class Gato(Animal):
    def emitir_som(self):
        print('Miau')
```

🔥 **Classe PetShop com Encapsulamento**

```
class PetShop:
    def __init__(self, nome):
        self.nome = nome
        self.__animais = []  # Encapsulado

    def cadastrar_animal(self, animal):
        self.__animais.append(animal)

    def listar_animais(self):
        print(f'Animais no {self.nome}:')
        for animal in self.__animais:
            print(animal.detalhes())

    def emitir_sons(self):
        for animal in self.__animais:
            animal.emitir_som()
```

✅ **Testando:**

```
petshop = PetShop('PetAmigo')

dog = Cachorro('Rex', 5, 'Labrador')
cat = Gato('Mimi', 3)

petshop.cadastrar_animal(dog)
petshop.cadastrar_animal(cat)

petshop.listar_animais()

petshop.emitir_sons()
```

Tratamento de Erros com Python + POO + Banco de Dados

1. Motivação – Por que tratar erros?

- **Erros acontecem:** entrada inválida, arquivo ausente, falha de conexão, divisão por zero, etc.
- Sem tratamento: o programa **quebra e fecha**.
- Com tratamento: o programa **continua**, mostra mensagens claras e dá opções ao usuário.

2. Sintaxe básica do try...except

```
try:
    # código que pode gerar erro
except TipoDeErro:
    # tratamento do erro
```

3. Exemplo básico: divisão com tratamento

```
try:
    numero = int(input("Digite um número: "))
    print(10 / numero)
except ZeroDivisionError:
    print("Erro: divisão por zero.")
except ValueError:
    print("Erro: valor inválido.")
```

4. else e finally

- else: executado se **não ocorrer erro**.
- finally: executado **sempre**, com ou sem erro.

```
try:
    n = int(input("Número: "))
    r = 10 / n
except Exception:
    print("Erro ocorreu.")
else:
    print("Resultado:", r)
finally:
    print("Operação encerrada.")
```

5. Tratando erros com arquivos

```
try:
    with open("arquivo.txt", "r") as f:
        print(f.read())
except FileNotFoundError:
    print("Arquivo não encontrado.")
```

6. Aplicando com Orientação a Objetos (POO)

Vamos agora usar **POO + tratamento de erro** criando uma **classe de conexão com banco de dados MySQL**.

7. Exemplo POO + Banco + Try/Except

```
pip install mysql-connector-python
```

Classe de conexão:

```
import mysql.connector
from mysql.connector import Error

class ConexaoBanco:
    def __init__(self, host, usuario, senha, banco):
        self.host = host
        self.usuario = usuario
        self.senha = senha
```

```
        self.banco = banco
        self.conexao = None

    def conectar(self):
        try:
            self.conexao = mysql.connector.connect(
                host=self.host,
                user=self.usuario,
                password=self.senha,
                database=self.banco
            )
            if self.conexao.is_connected():
                print("Conexão bem-sucedida!")
        except Error as erro:
            print(f"Erro ao conectar: {erro}")
        finally:
            print("Tentativa de conexão encerrada.")

    def desconectar(self):
        if self.conexao and self.conexao.is_connected():
            self.conexao.close()
            print("Conexão encerrada.")
```

Uso prático da classe:

```
db = ConexaoBanco("localhost", "root", "senha", "escola")
db.conectar()
db.desconectar()
```

8. Incluindo tratamento em consultas

```
def buscar_dados(self, query):
    try:
        cursor = self.conexao.cursor()
        cursor.execute(query)
        resultado = cursor.fetchall()
        return resultado
    except Error as erro:
        print(f"Erro ao executar consulta: {erro}")
        return []
```

Exemplo de uso completo:

```
db = ConexaoBanco("localhost", "root", "1234", "escola")
db.conectar()

dados = db.buscar_dados("SELECT * FROM alunos")

for aluno in dados:
    print(aluno)

db.desconectar()
```

9. Benefício do Encapsulamento + Tratamento de Erros

- **Toda lógica de conexão/erro fica escondida na classe.**
- **O usuário da classe não precisa saber como lidar com exceções específicas.**
- **O código principal fica limpo e seguro.**

10. Exercício sugerido para os alunos:

Crie uma aplicação orientada a objetos que:

- Possua uma classe Aluno com nome, idade e curso.
- Possua uma classe BancoAluno que:
    - Conecta ao banco
    - Insere alunos
    - Lista alunos
- Trate todos os erros possíveis com try/except.