Desenvolver código orientado a objetos

Professor Gabriel Pires de Barros

Senac Novo Hamburgo

2024

Introdução à Programação em Python

1. Visão Geral da Linguagem Python

1.1 História e Evolução do Python

Python é uma linguagem de programação de alto nível, criada por Guido van Rossum e lançada pela primeira vez em 1991. Seu design enfatiza a legibilidade do código, permitindo que programadores expressem conceitos de forma concisa e clara. Python é amplamente utilizado em diversas áreas como desenvolvimento web, ciência de dados, inteligência artificial, automação de scripts, entre outras.

A evolução do Python ao longo dos anos tem mantido seu foco na simplicidade e facilidade de uso. Atualmente, Python está em sua versão 3.x, que trouxe várias melhorias e mudanças em relação à versão 2.x. É importante destacar que Python 2 foi descontinuado em 2020, e todo o desenvolvimento e suporte estão concentrados na versão 3.

1.2 Instalação e Configuração do Ambiente de Desenvolvimento

Para começar a programar em Python, é necessário instalar a linguagem e configurar o ambiente de desenvolvimento. Siga os passos abaixo:

1. **Instalação do Python:**
    - Acesse o site oficial do Python e baixe a versão mais recente (Python 3.x).
    - Durante a instalação, marque a opção "Add Python to PATH" para garantir que o Python seja reconhecido pelo terminal.
2. **Escolha de um Editor de Texto ou IDE:**
    - **Visual Studio Code (VSCode):** Um editor de texto leve e poderoso. Recomendado pela sua flexibilidade e suporte a extensões.
    - **PyCharm:** Uma IDE específica para Python, com muitas funcionalidades integradas.
3. **Verificação da Instalação:**
    - Abra o terminal (Prompt de Comando no Windows, Terminal no macOS/Linux).
    - Digite `python --version` para verificar se o Python foi instalado corretamente.
    - Para testar o ambiente, digite `python` para entrar no interpretador interativo, onde você pode começar a digitar comandos em Python.

1.3 Primeiro Contato com a Sintaxe do Python

Python é conhecido por sua sintaxe simples e legível. Vamos explorar alguns conceitos básicos:

1. **Exemplo de "Hello, World!":**

```
print("Hello, World!")
```

    a. Este é o programa mais simples em Python, que imprime a mensagem "Hello, World!" na tela.
    b. Observe que o Python não requer ponto e vírgula no final das linhas e que a indentação é fundamental para definir blocos de código.
2. **Indentação:**
    a. Em Python, a indentação não é apenas uma questão de estilo; ela define a estrutura do código. Todos os blocos de código, como loops e condicionais, devem ser indentados de maneira consistente.
3. **Comentários:**
    a. Comentários de linha única são feitos com o símbolo `#`.
    b. Comentários de múltiplas linhas podem ser feitos com três aspas `'''`.

2. Elementos Básicos da Linguagem

2.1 Variáveis, Tipos de Dados e Operadores

As variáveis em Python são dinâmicas, ou seja, você não precisa declarar o tipo da variável explicitamente. O Python detecta o tipo de dados automaticamente.

```
x = 10           # Inteiro
y = 3.14         # Float
nome = "Alice"  # String
ativo = True     # Booleano
```

- **Operadores:** Python suporta os operadores aritméticos básicos (+, -, *, /, //, %, **), operadores de comparação (==, !=, >, <, >=, <=), e operadores lógicos (and, or, not).

2.2 Estruturas Condicionais (if, elif, else)

Estruturas condicionais em Python permitem que você execute diferentes blocos de código com base em certas condições.

```
idade = 20
if idade >= 18:
      print("Você é maior de idade.")
elif idade < 18 and idade > 12:
      print("Você é adolescente.")
else:
      print("Você é criança.")
```

- **if:** Verifica uma condição.
- **elif:** Verifica uma condição alternativa.
- **else:** Executa o código se nenhuma das condições anteriores for verdadeira.

2.3 Estruturas de Repetição (for, while)

As estruturas de repetição permitem executar um bloco de código várias vezes.

1. **For:**
    - Itera sobre uma sequência (como uma lista, tupla ou string).

```
for i in range(5):
      print(i)
```

While:

- Repete um bloco de código enquanto uma condição for verdadeira.

```
contador = 0
while contador < 5:
      print(contador)
      contador += 1
```

2.4 Introdução a Funções e Sub-rotinas

Funções são blocos de código que realizam uma tarefa específica e podem ser reutilizados.

```
def saudacao(nome):
      print(f"Olá, {nome}!")
```

```
saudacao("João")
```

- **Definição:** Use a palavra-chave `def` para definir uma função.
- **Argumentos:** Funções podem receber argumentos.
- **Retorno:** Funções podem retornar valores usando `return`.

3. Boas Práticas de Programação

3.1 PEP 8 – Estilo de Código Python

PEP 8 é o guia de estilo para escrever código Python. Ele fornece convenções sobre como estruturar o código para que ele seja legível e consistente.

Algumas das principais recomendações incluem:

- **Indentação:** Use 4 espaços por nível de indentação.
- **Comprimento de Linhas:** Limite as linhas a 79 caracteres.
- **Nomes de Variáveis e Funções:** Utilize nomes em minúsculas, separados por underscores (_).
- **Espaçamento:** Evite espaços em branco desnecessários.

3.2 Comentários e Documentação Básica de Código

Comentários são essenciais para explicar o que o código faz, especialmente em projetos colaborativos. Eles devem ser claros e diretos, evitando a redundância.

- **Comentários de Linha Única:**

```
# Esta função calcula a soma de dois números
def soma(a, b):
      return a + b
```

Docstrings:

- Utilizados para documentar funções, classes e módulos.

```
def soma(a, b):
      """
      Retorna a soma de dois números.

      Argumentos:
      a -- o primeiro número
      b -- o segundo número
```

```
    """
    return a + b
```

4. Atividade Prática

Reescrevendo um Programa em Python

Objetivo: Você já tem experiência com C++, e agora é hora de aplicar seus conhecimentos em Python. A atividade consiste em pegar um programa que você já desenvolveu em C++ e convertê-lo para Python, aplicando os conceitos aprendidos até agora.

Passos para a Atividade:

1. **Escolha do Programa:**
    - Selecione um programa simples que você escreveu em C++, como um sistema básico de cadastro, uma calculadora ou um jogo simples.
2. **Reescrita do Código:**
    - Converta o código para Python, adaptando a sintaxe e as estruturas de dados.
    - Use funções e siga as boas práticas discutidas (PEP 8, comentários, docstrings).
3. **Análise e Otimização:**
    - Compare o código em Python com o original em C++.
    - Reflita sobre as diferenças em termos de sintaxe, estrutura e legibilidade.
4. **Entrega:**
    - Faça a entrega via Google Class Room, na atividade com nome `"Conversão cpp to python"`

Fundamentos de Orientação a Objetos

1. Conceitos Básicos de Orientação a Objetos

1.1 O que é Orientação a Objetos?

A Orientação a Objetos (OO) é um paradigma de programação que utiliza "objetos" como unidades fundamentais para a construção de software. Esses objetos são instâncias de "classes", que podem conter atributos (dados) e métodos (funções) que definem seu comportamento. Esse modelo é inspirado na maneira como percebemos e interagimos com o mundo real, onde os objetos possuem características e comportamentos próprios.

Comparação com a Vida Real

Imagine um carro. Um carro é um objeto que tem atributos como cor, modelo, marca, e ano de fabricação. Além disso, ele possui comportamentos, como acelerar, frear, e virar. Na programação orientada a objetos, podemos representar o carro como uma classe, e cada carro específico (como o carro de João ou o carro de Maria) seria um objeto, uma instância dessa classe.

1.2 Definição de Classes e Objetos

Uma **classe** é um molde, ou um projeto, que define as características e comportamentos dos objetos que serão criados a partir dela. Um **objeto** é uma instância de uma classe, contendo dados reais e podendo executar funções específicas.

Exemplo de Classe e Objeto

```
class Carro:
      def __init__(self, marca, modelo, ano):
      self.marca = marca
      self.modelo = modelo
      self.ano = ano

      def acelerar(self):
      print(f"O {self.modelo} está acelerando.")

      def frear(self):
      print(f"O {self.modelo} está freando.")

# Criando um objeto da classe Carro
meu_carro = Carro("Toyota", "Corolla", 2020)
meu_carro.acelerar()  # Saída: O Corolla está acelerando.
```

Nesse exemplo, a classe `Carro` define os atributos `marca`, `modelo`, e `ano`, além dos

métodos `acelerar` e `frear`. O objeto `meu_carro` é uma instância da classe `Carro` com valores específicos para os atributos.

1.3 Instâncias, Atributos e Métodos

- **Instância:** Quando criamos um objeto a partir de uma classe, estamos criando uma instância dessa classe. Por exemplo, `meu_carro` é uma instância da classe `Carro`.
- **Atributos:** São variáveis que pertencem a uma classe e descrevem as propriedades do objeto. No exemplo acima, `marca`, `modelo`, e `ano` são atributos da classe `Carro`.
- **Métodos:** São funções que pertencem a uma classe e definem os comportamentos dos objetos. No exemplo, `acelerar` e `frear` são métodos da classe `Carro`.

2. Encapsulamento

2.1 Visibilidade de Atributos e Métodos (Público, Privado, Protegido)

Encapsulamento é o princípio que visa proteger os dados dentro de uma classe, controlando como os atributos e métodos podem ser acessados e modificados. Em Python, a visibilidade de atributos e métodos pode ser controlada usando convenções de nomeação:

- **Público (Public):** Atributos e métodos que podem ser acessados de qualquer lugar. Em Python, eles são definidos sem nenhum caractere especial no início do nome.

```
class Carro:
      def __init__(self, marca):
      self.marca = marca  # Atributo público
```

- **Protegido (Protected):** Atributos e métodos que devem ser acessados apenas dentro da classe e por subclasses. Em Python, usa-se um underscore (_) antes do nome.

```
class Carro:
      def __init__(self, marca):
      self._marca = marca  # Atributo protegido
```

- **Privado (Private):** Atributos e métodos que só podem ser acessados dentro da própria classe. Em Python, usa-se dois underscores (__) antes do nome.

```
class Carro:
      def __init__(self, marca):
      self.__marca = marca  # Atributo privado
```

2.2 Métodos Getter e Setter

Os métodos **getter** e **setter** são usados para acessar e modificar atributos privados de uma classe, respeitando o princípio do encapsulamento.

- **Getter:** Método que retorna o valor de um atributo privado.

```
class Carro:
      def __init__(self, marca):
      self.__marca = marca

      def get_marca(self):
      return self.__marca
```

- **Setter:** Método que altera o valor de um atributo privado.

```
class Carro:
      def __init__(self, marca):
      self.__marca = marca

      def set_marca(self, marca):
      self.__marca = marca
```

3. Aplicações Práticas

3.1 Criação de uma Classe Simples e Manipulação de Objetos

Vamos criar uma calculadora simples usando o paradigma da orientação a objetos.

Exemplo: Classe Calculadora

```
class Calculadora:
      def __init__(self):
      self._resultado = 0

      def somar(self, valor):
      self._resultado += valor

      def subtrair(self, valor):
      self._resultado -= valor

      def multiplicar(self, valor):
      self._resultado *= valor
```

```
    def dividir(self, valor):
    if valor != 0:
        self._resultado /= valor
    else:
        print("Erro: Divisão por zero não é permitida.")

    def get_resultado(self):
    return self._resultado

    def reset(self):
    self._resultado = 0
```

Uso da Calculadora:

```
calc = Calculadora()
calc.somar(10)
calc.subtrair(2)
calc.multiplicar(3)
calc.dividir(4)
print(calc.get_resultado())  # Saída: 6.0
calc.reset()
```

3.2 Comparação entre Programação Procedural e Orientada a Objetos

Na programação procedural, o código é estruturado em funções e procedimentos que operam em dados. Já na orientação a objetos, os dados e comportamentos são agrupados dentro de objetos, o que facilita a organização, manutenção e reutilização do código.

Exemplo Comparativo:

Procedural:

```
def somar(a, b):
    return a + b

resultado = somar(5, 3)
```

Orientada a Objetos:

```
class Calculadora:
    def somar(self, a, b):
    return a + b
```

```
calc = Calculadora()
resultado = calc.somar(5, 3)
```

4. Exemplo Avançado: Operações Matemáticas com Polimorfismo

4.1 Classe Abstrata `Operacao`

Vamos criar uma classe abstrata `Operacao` com métodos que serão implementados por classes concretas.

```
from abc import ABC, abstractmethod

class Operacao(ABC):
      @abstractmethod
      def calcular(self, a, b):
      pass
```

4.2 Classes Concretas `Soma`, `Subtracao`, `Multiplicacao`, `Divisao`

```
class Soma(Operacao):
      def calcular(self, a, b):
      return a + b

class Subtracao(Operacao):
      def calcular(self, a, b):
      return a - b

class Multiplicacao(Operacao):
      def calcular(self, a, b):
      return a * b

class Divisao(Operacao):
      def calcular(self, a, b):
      if b != 0:
            return a / b
      else:
            return "Erro: Divisão por zero."
```

4.3 Aplicação com Polimorfismo

```
def executar_operacao(operacao, a, b):
      return operacao.calcular(a, b)

operacao = Soma()
resultado = executar_operacao(operacao, 10, 5)
print(f"Soma: {resultado}")  # Saída: Soma: 15

operacao = Subtracao()
resultado = executar_operacao(operacao, 10, 5)
print(f"Subtração: {resultado}")  # Saída: Subtração: 5

operacao = Multiplicacao()
resultado = executar_operacao(operacao, 10, 5)
print(f"Multiplicação: {resultado}")  # Saída: Multiplicação: 50

operacao = Divisao()
resultado = executar_operacao(operacao, 10, 5)
print(f"Divisão: {resultado}")  # Saída: Divisão: 2.0
```

Código completo da calculadora

```
from abc import ABC, abstractmethod

# Classe abstrata Operacao
class Operacao(ABC):
      @abstractmethod
      def calcular(self, a, b):
      pass

# Classes concretas que herdam de Operacao
class Soma(Operacao):
      def calcular(self, a, b):
      return a + b

class Subtracao(Operacao):
      def calcular(self, a, b):
      return a - b

class Multiplicacao(Operacao):
      def calcular(self, a, b):
      return a * b
```

```
class Divisao(Operacao):
      def calcular(self, a, b):
      if b != 0:
            return a / b
      else:
            return "Erro: Divisão por zero não é permitida."

# Classe Calculadora com métodos públicos e protegidos
class Calculadora:
      def __init__(self):
      self._resultado = 0
      self._operacao = None

      def definir_operacao(self, operacao):
      self._operacao = operacao

      def calcular(self, a, b):
      if self._operacao:
            self._resultado = self._operacao.calcular(a, b)
            return self._resultado
      else:
            return "Nenhuma operação definida."

      def get_resultado(self):
      return self._resultado

      def reset(self):
      self._resultado = 0

# Função principal para rodar o programa
def main():
      calc = Calculadora()

      while True:
      print("\nMenu de Operações:")
      print("1. Soma")
      print("2. Subtração")
      print("3. Multiplicação")
      print("4. Divisão")
      print("5. Sair")

      escolha = input("Escolha a operação (1/2/3/4) ou 5 para sair: ")

      if escolha == '5':
```

```
            print("Encerrando a calculadora.")
            break

      num1 = float(input("Digite o primeiro número: "))
      num2 = float(input("Digite o segundo número: "))

      if escolha == '1':
            calc.definir_operacao(Soma())
      elif escolha == '2':
            calc.definir_operacao(Subtracao())
      elif escolha == '3':
            calc.definir_operacao(Multiplicacao())
      elif escolha == '4':
            calc.definir_operacao(Divisao())
      else:
            print("Escolha inválida.")
            continue

      resultado = calc.calcular(num1, num2)
      print(f"Resultado: {resultado}")

if __name__ == "__main__":
      main()
```

5. Exercício Prático: Programa para uma Biblioteca

Objetivo: Aplicar os conceitos de orientação a objetos para criar um sistema básico de gerenciamento de uma biblioteca.

Enunciado do Exercício

Você deve criar um programa que simule o funcionamento de uma biblioteca. O programa deve incluir as seguintes funcionalidades:

1. **Classes a serem Criadas:**
   - `Livro`: Representa um livro, com atributos como `título`, `autor`, `ISBN`, e `disponibilidade`.
   - `Autor`: Representa um autor, com atributos como `nome` e `nacionalidade`.
   - `Usuario`: Representa um usuário da biblioteca, com atributos como `nome`, `id` e `livros_emprestados`.
2. **Métodos a serem Implementados:**
   - `Livro.adicionar()`: Adiciona um novo livro ao sistema.
   - `Livro.buscar()`: Permite buscar livros pelo título ou autor.

- `Livro.emprestar()`: Empresta um livro para um usuário.
- `Livro.devolver()`: Devolve um livro para a biblioteca.

3. **Outras Funcionalidades:**
   - Use coleções (como listas) para armazenar os livros e usuários.
   - Utilize encapsulamento para proteger os atributos das classes.
   - Implemente métodos getter e setter conforme necessário.
   - Utilize polimorfismo se achar necessário.

Entrega: O programa deve ser entregue via Classroom, na atividade com o nome "Biblioteca OO".

Os alunos devem testar suas soluções e garantir que todas as funcionalidades estejam implementadas corretamente.

Tópico 3: Estruturas de Dados em Python

Neste tópico, vamos explorar as diferentes estruturas de dados que o Python oferece, entender como manipulá-las e como utilizá-las eficientemente em nossos programas. Além disso, vamos abordar o manejo de memória, coleta de lixo, e como esses conceitos afetam o desempenho e a confiabilidade dos nossos códigos.

Estruturas de Dados Comuns

Listas

As listas são uma das estruturas de dados mais versáteis em Python. Elas são ordenadas, mutáveis (podem ser alteradas após a criação), e podem conter elementos de diferentes tipos.

```
# Criando uma lista
frutas = ["maçã", "banana", "laranja"]

# Acessando elementos
print(frutas[0])  # Saída: maçã

# Adicionando elementos
frutas.append("uva")  # Adiciona no final
frutas.insert(1, "abacaxi")  # Adiciona na posição 1

# Removendo elementos
frutas.remove("banana")  # Remove o primeiro elemento igual a "banana"
fruta = frutas.pop(0)  # Remove e retorna o primeiro elemento
print(frutas)
```

Tuplas

As tuplas são similares às listas, mas são imutáveis (não podem ser alteradas após a criação). São úteis para armazenar dados que não devem ser modificados.

```
# Criando uma tupla
coordenadas = (10, 20)

# Acessando elementos
print(coordenadas[0])  # Saída: 10
```

```
# Não é possível modificar uma tupla
# coordenadas[0] = 15  # Isso causaria um erro
```

Dicionários

Os dicionários são coleções de pares chave-valor. Eles são úteis para armazenar dados que precisam ser associados de forma única.

```
# Criando um dicionário
aluno = {"nome": "João", "idade": 20, "curso": "Engenharia"}

# Acessando valores
print(aluno["nome"])  # Saída: João

# Adicionando ou modificando elementos
aluno["idade"] = 21  # Modifica o valor da chave "idade"
aluno["cidade"] = "São Paulo"  # Adiciona uma nova chave-valor

# Removendo elementos
del aluno["curso"]  # Remove a chave "curso"
```

Conjuntos

Os conjuntos são coleções não ordenadas de elementos únicos. Eles são úteis para operações matemáticas como união e interseção.

```
# Criando um conjunto
numeros = {1, 2, 3, 4, 4}  # O valor 4 duplicado será removido
automaticamente

# Adicionando e removendo elementos
numeros.add(5)  # Adiciona o valor 5
numeros.discard(2)  # Remove o valor 2, se existir

# Operações de conjuntos
outros_numeros = {3, 4, 5, 6}
uniao = numeros | outros_numeros  # União: {1, 3, 4, 5, 6}
interseccao = numeros & outros_numeros  # Interseção: {3, 4, 5}
```

Manipulação e Iteração sobre Estruturas de Dados

Manipular e iterar sobre estruturas de dados é uma habilidade essencial. Em Python, isso pode ser feito de várias formas.

```
# Iterando sobre uma lista
for fruta in frutas:
      print(fruta)

# Adicionando valores em uma lista
frutas.append("melancia")

# Removendo valores de um dicionário
if "cidade" in aluno:
      del aluno["cidade"]
```

Pilha, Fila, Deque e Árvores

Pilha

Uma pilha segue o princípio LIFO (Last In, First Out). O último elemento a ser inserido é o primeiro a ser removido.

```
# Implementação básica de uma pilha
class Pilha:
      def __init__(self):
      self.items = []

      def push(self, item):
      self.items.append(item)

      def pop(self):
      return self.items.pop()

      def peek(self):
      return self.items[-1] if self.items else None

      def is_empty(self):
      return len(self.items) == 0

# Uso da Pilha
pilha = Pilha()
pilha.push(10)
```

```
pilha.push(20)
print(pilha.pop())  # Saída: 20
```

Fila

Uma fila segue o princípio FIFO (First In, First Out). O primeiro elemento a ser inserido é o primeiro a ser removido.

```
# Implementação básica de uma fila
class Fila:
      def __init__(self):
      self.items = []

      def enqueue(self, item):
      self.items.append(item)

      def dequeue(self):
      return self.items.pop(0)

      def is_empty(self):
      return len(self.items) == 0


# Uso da Fila
fila = Fila()
fila.enqueue(10)
fila.enqueue(20)
print(fila.dequeue())  # Saída: 10
```

Deque

Deque (Double-Ended Queue) permite inserções e remoções em ambas as extremidades.

```
from collections import deque

# Implementação de Deque
deq = deque([1, 2, 3])
deq.append(4)  # Adiciona no final
deq.appendleft(0)  # Adiciona no início
print(deq)  # Saída: deque([0, 1, 2, 3, 4])

deq.pop()  # Remove do final
deq.popleft()  # Remove do início
```

```
print(deq)  # Saída: deque([1, 2, 3])
```

Árvores

Árvores são estruturas de dados hierárquicas, onde cada elemento é um nó, e os nós podem ter "filhos".

```
# Implementação básica de uma árvore binária
class No:
      def __init__(self, valor):
      self.valor = valor
      self.esquerda = None
      self.direita = None

class ArvoreBinaria:
      def __init__(self):
      self.raiz = None

      def adicionar(self, valor):
      if self.raiz is None:
            self.raiz = No(valor)
      else:
            self._adicionar(valor, self.raiz)

      def _adicionar(self, valor, no_atual):
      if valor < no_atual.valor:
            if no_atual.esquerda is None:
                  no_atual.esquerda = No(valor)
            else:
                  self._adicionar(valor, no_atual.esquerda)
      else:
            if no_atual.direita is None:
                  no_atual.direita = No(valor)
            else:
                  self._adicionar(valor, no_atual.direita)

      def busca(self, valor):
      return self._busca(valor, self.raiz)

      def _busca(self, valor, no_atual):
      if no_atual is None:
            return False
      elif valor == no_atual.valor:
            return True
```

```
        elif valor < no_atual.valor:
                return self._busca(valor, no_atual.esquerda)
        else:
                return self._busca(valor, no_atual.direita)

# Uso da árvore
arvore = ArvoreBinaria()
arvore.adicionar(10)
arvore.adicionar(5)
arvore.adicionar(15)
print(arvore.busca(7))  # Saída: False
print(arvore.busca(10))  # Saída: True
```

Estruturas Mais Complexas

Estruturas de dados mais complexas como árvores AVL, árvores B e grafos, são utilizadas para resolver problemas específicos que exigem um desempenho eficiente em grandes volumes de dados.

Manejo de Memória e Coleta de Lixo

Como o Python Gerencia Memória

Python utiliza um sistema de gerenciamento de memória automática, onde a memória é alocada e liberada conforme necessário. Python usa um mecanismo de contagem de referências para rastrear quando um objeto não é mais utilizado e pode ser liberado.

Coleta de Lixo

O Python possui um coletor de lixo que remove objetos que não são mais acessíveis, liberando memória para novos objetos.

```
# Exemplo básico de contagem de referências
a = [1, 2, 3]
b = a  # 'b' e 'a' referenciam o mesmo objeto
del a  # 'b' ainda referencia o objeto, então ele não é coletado

# Gerenciamento incorreto de memória
import sys

def criar_lista_grande():
        return [i for i in range(1000000)]
```

```
lista1 = criar_lista_grande()
print(sys.getrefcount(lista1))  # Exibe a contagem de referências

# A função a seguir pode causar um vazamento de memória se não
liberarmos a referência manualmente
def vazamento_memoria():
      lista = criar_lista_grande()
      return lista

# Liberando memória
del lista1  # Agora, o objeto pode ser coletado

# Coleta de lixo manual (não é necessário na maioria dos casos)
import gc
gc.collect()
```

Exercício Prático: Simulador de Pilha

Enunciado:

Crie um simulador de pilha utilizando apenas listas em Python. A pilha deve permitir as operações básicas:

- **push**: Adiciona um elemento ao topo da pilha.
- **pop**: Remove e retorna o elemento do topo da pilha.
- **peek**: Retorna o elemento do topo da pilha sem removê-lo.
- **is_empty**: Retorna True se a pilha estiver vazia, False caso contrário.

Adicione uma funcionalidade:

- **max_size**: Define o tamanho máximo da pilha. Se o usuário tentar adicionar um elemento quando a pilha estiver cheia, o programa deve imprimir uma mensagem de erro e não adicionar o elemento.

Crie um programa principal que:

- Crie uma pilha com um tamanho máximo definido pelo usuário.
- Permita que o usuário realize operações na pilha (push, pop, peek).
- Imprima o estado da pilha após cada operação.

Desafio:

- **Gerenciamento de memória**: Certifique-se de que a pilha não ultrapasse o tamanho máximo definido.
- **Retorno correto**: O programa deve sempre retornar o estado correto da pilha, mesmo após operações inválidas.

Como o Erro de Memória Afeta o Output

- **Overflow**: Se o aluno não implementar a verificação de tamanho máximo, ao tentar adicionar um elemento em uma pilha cheia, o programa pode gerar um IndexError ou adicionar o elemento de forma incorreta, alterando o estado da pilha.
- **Underflow**: Se o aluno não verificar se a pilha está vazia antes de realizar uma operação de pop ou peek, o programa pode gerar um IndexError.
- **Vazamento de memória**: Embora Python tenha um gerenciamento de memória automático, o aluno pode criar estruturas de dados complexas que, se não forem gerenciadas corretamente, podem consumir muita memória.

Entrega: O programa deve ser entregue via Classroom, na atividade com o nome "Estrutura de dados e memória OO".

Tópico 4: Programação Orientada a Objetos em Python

Definição de Classes e Objetos em Python

A **Programação Orientada a Objetos (POO)** é um dos pilares da programação moderna e Python oferece suporte robusto para este paradigma. Em POO, modelamos o mundo real utilizando **classes** e **objetos**. Cada classe é como um molde, e os objetos são instâncias criadas a partir desse molde.

Sintaxe para Criação de Classes

Para criar uma classe em Python, usamos a palavra-chave `class`:

```
class NomeDaClasse:
      pass
```

Exemplo Básico:

```
class Pessoa:
      def __init__(self, nome, idade):
      self.nome = nome  # Atributo de instância
      self.idade = idade  # Atributo de instância

      def cumprimentar(self):  # Método de instância
      return f"Olá, meu nome é {self.nome}."

# Criando um objeto da classe Pessoa
pessoa1 = Pessoa("Carlos", 30)
pessoa2 = Pessoa("João", 18)
print(pessoa1.cumprimentar())
```

Atributos de Classe e de Instância

- **Atributos de instância** são únicos para cada objeto criado a partir de uma classe. Eles são definidos dentro do método `__init__`, como visto acima.
- **Atributos de classe** são compartilhados por todos os objetos da classe.

Exemplo de Atributos de Classe:

```
class Carro:
      rodas = 4  # Atributo de classe
```

```
    def __init__(self, marca, modelo):
    self.marca = marca  # Atributo de instância
    self.modelo = modelo  # Atributo de instância
```

Aqui, `rodas` é um atributo de classe, acessível por todos os objetos da classe `Carro`.

Métodos de Instância, Classe e Estáticos

- **Métodos de instância**: Acessam e manipulam os atributos dos objetos. Usam o `self` como primeiro parâmetro.
- **Métodos de classe**: Usam o `cls` como primeiro parâmetro e podem acessar atributos de classe.
- **Métodos estáticos**: Não dependem de nenhum parâmetro especial, usados para utilidades ou funcionalidades que não alteram o estado da classe ou dos objetos.

Exemplo de Métodos:

```
class Exemplo:
    contador = 0  # Atributo de classe

    def __init__(self, valor):
    self.valor = valor  # Atributo de instância
    Exemplo.contador += 1  # Modifica atributo de classe

    @classmethod
    def total_instancias(cls):
    return f"Total de instâncias criadas: {cls.contador}"

    @staticmethod
    def mensagem():
    return "Este é um método estático."

# Exemplo de uso:
obj1 = Exemplo(5)
obj2 = Exemplo(10)

print(Exemplo.total_instancias())
print(Exemplo.mensagem())
```

Herança

Conceito de Herança e Reutilização de Código

A **herança** permite que uma classe (subclasse) herde atributos e métodos de outra classe (superclasse). Isso promove a reutilização de código.

Exemplo de Herança:

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

      def fazer_barulho(self):
      return "Algum som de animal."

class Cao(Animal):
      def fazer_barulho(self):  # Sobrescrevendo o método
      return "Latido!"

class Gato(Animal):
      def fazer_barulho(self):
      return "Miau!"

# Testando
cao = Cao("Rex")
gato = Gato("Mingau")
print(cao.fazer_barulho())  # Latido!
print(gato.fazer_barulho())  # Miau!
```

Implementação de Herança Simples e Múltipla

A **herança simples** ocorre quando uma classe herda diretamente de uma única classe base. A **herança múltipla** permite que uma classe herde de várias classes.

Exemplo de Herança Múltipla:

```
class Voar:
      def voar(self):
      return "Voando alto!"

class Passaro(Animal, Voar):
      pass

passaro = Passaro("Papagaio")
print(passaro.voar())  # Voando alto!
```

Polimorfismo

O **polimorfismo** permite que diferentes classes usem o mesmo nome de método, mas com implementações diferentes.

Métodos Sobrescritos

Sobrescrever métodos ocorre quando uma subclasse implementa um método com o mesmo nome da classe base, alterando seu comportamento.

Exemplo:

```
class Inimigo:
      def atacar(self):
      return "Inimigo atacando!"

class Zumbi(Inimigo):
      def atacar(self):  # Sobrescrevendo o método
      return "Zumbi mordendo!"

inimigo = Inimigo()
zumbi = Zumbi()
print(inimigo.atacar())  # Inimigo atacando!
print(zumbi.atacar())  # Zumbi mordendo!
```

Métodos Sobrecarrregados

Python não suporta diretamente **sobrecarga** de métodos como algumas linguagens, mas podemos simular esse comportamento utilizando argumentos padrões ou variáveis.

Exemplo:

```
class Calculadora:
      def somar(self, a, b, c=0):
      return a + b + c

calc = Calculadora()
print(calc.somar(2, 3))  # 5
print(calc.somar(2, 3, 4))  # 9
```

Programa Completo Aplicando Polimorfismo

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

```
        self.idade = idade

    def fazer_barulho(self):
        pass

class Cao(Animal):
    def fazer_barulho(self):
        return "Latido"

class Gato(Animal):
    def fazer_barulho(self):
        return "Miau"

class Passaro(Animal):
    def fazer_barulho(self):
        return "Piu Piu"

# Lista de animais com polimorfismo
animais = [Cao("Rex", 5), Gato("Mingau", 3), Passaro("Tweety", 1)]
for animal in animais:
    print(f"O {animal.nome} faz: {animal.fazer_barulho()}")
```

Enunciado Completo da Atividade: Programa Zoo OO

Objetivo:

Criar um programa em Python que simule um zoológico, utilizando os conceitos de Programação Orientada a Objetos (POO). O programa deve permitir o cadastro e gerenciamento de diferentes tipos de animais, demonstrando a aplicação de herança e polimorfismo.

Requisitos:

1. **Criação de Classes:**
    - **Animal:** Classe base com os seguintes atributos:
        - `nome`: Nome do animal
        - `idade`: Idade do animal
        - `barulho`: Som característico do animal
        - `movimento`: Forma de locomoção do animal
        - `alimentacao`: Dieta do animal
        - `habitat`: Habitat natural do animal
        - `vizinhos`: Lista com os nomes dos animais vizinhos (máximo 2)
        - `horas_alimentacao`: Horário de alimentação
    - **Subclasses:** Crie subclasses para pelo menos três categorias de animais (mamíferos, aves, répteis), com atributos e métodos específicos. Exemplos de subclasses:
        - **Mamifero:** Leão, Tigre, Elefante
        - **Ave:** Águia, Coruja, Pinguim
        - **Reptil:** Cobra, Crocodilo, Tartaruga
2. **Cadastro de Animais:**
    - O programa deve permitir o cadastro de novos animais, solicitando as informações necessárias para cada atributo.
    - Os animais cadastrados devem ser armazenados em uma lista.
3. **Menu de Opções:**
    - O programa deve apresentar um menu com as seguintes opções:
        - **Listar todos os animais:** Exibir uma lista com os nomes de todos os animais cadastrados.
        - **Buscar animal:** Permitir a busca por um animal específico pelo nome, exibindo suas informações detalhadas.
        - **Listar animais por categoria:** Permitir listar todos os animais de uma determinada categoria (mamífero, ave, réptil).

        - **Listar vizinhos de um animal:** Exibir os nomes dos animais vizinhos de um animal específico.
        - **Simular alimentação:** Simular a alimentação dos animais, exibindo uma mensagem informando que o animal foi alimentado.
    - **Sair do programa:** Encerrar a execução do programa.
4. **Entrega:**

    - O programa deve ser desenvolvido em Python.
    - O código fonte deve ser organizado em um repositório no GitHub.
    - O link do repositório deve ser enviado para a atividade "Programa Zoo OO" no Google Classroom.
    - A estrutura do repositório deve ser clara e organizada, com um arquivo README explicando o funcionamento do programa.

Lista de Animais para Cadastro Inicial:

1. Leão (mamífero)
2. Águia (ave)
3. Cobra (réptil)
4. Elefante (mamífero)
5. Coruja (ave)
6. Tartaruga (réptil)

Dicas:

- Utilize comentários para explicar o código e facilitar a compreensão.
- Explore os conceitos de herança e polimorfismo para criar um código mais eficiente e reutilizável.
- Utilize listas e dicionários para armazenar as informações dos animais.
- Implemente a validação de dados para garantir que as informações inseridas pelo usuário sejam consistentes.

Tópico 5: UML e Análise de Sistemas Orientada a Objetos

Introdução à UML

UML (Unified Modeling Language) é uma linguagem de modelagem visual usada para representar, especificar, construir e documentar artefatos de sistemas orientados a objetos. Ela é amplamente utilizada no desenvolvimento de software para modelar os aspectos estruturais e comportamentais de um sistema.

Importância da UML no desenvolvimento de software

A UML é essencial porque facilita a comunicação entre desenvolvedores, analistas e stakeholders de um projeto, ajudando a garantir que todos tenham uma compreensão clara dos requisitos e da estrutura do sistema. A UML permite:

- Visualizar a arquitetura do sistema antes de codificar.
- Aumentar a compreensão e reduzir erros de desenvolvimento.
- Promover a reutilização de componentes.
- Melhorar a documentação do sistema.

Ferramentas para criação de diagramas UML

Existem diversas ferramentas disponíveis para criar diagramas UML. Algumas das mais utilizadas incluem:

- **Lucidchart**: Uma ferramenta online que permite criar e compartilhar diagramas UML de maneira colaborativa.
- **StarUML**: Software desktop para modelagem UML com suporte para múltiplos tipos de diagramas.
- **Draw.io**: Uma ferramenta gratuita e online que permite criar diagramas UML de maneira simples.
- **Enterprise Architect**: Ferramenta robusta para modelagem UML e desenvolvimento de software em larga escala.
- **Visual Paradigm**: Ferramenta poderosa para criar diagramas UML e modelar processos de negócios.

Principais Diagramas UML

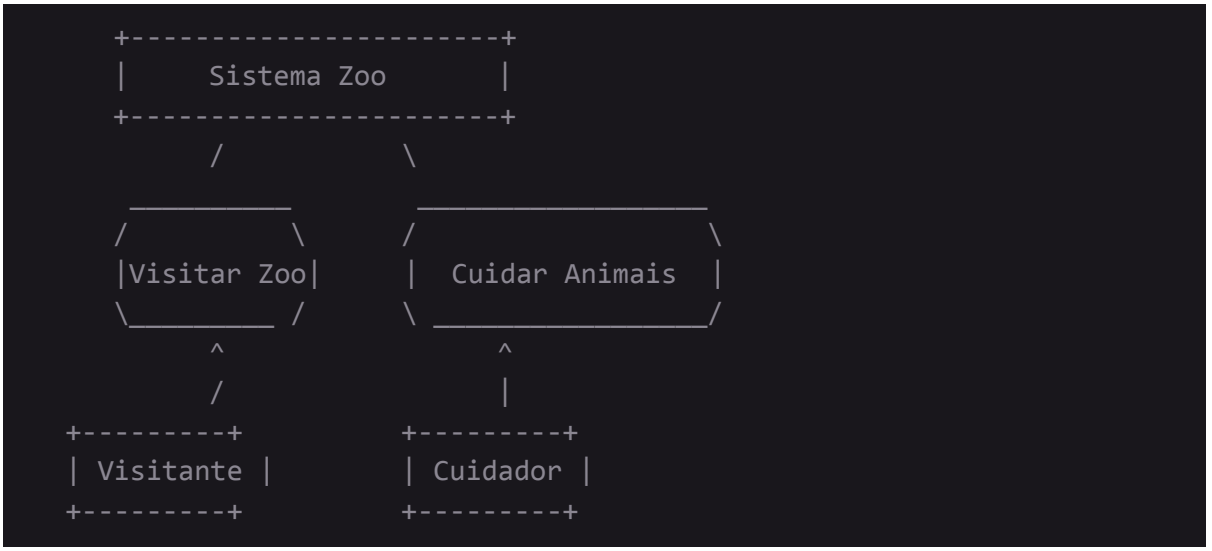
Diagrama de Caso de Uso

O diagrama de caso de uso é uma das representações mais básicas da UML. Ele descreve as interações entre os atores (usuários ou sistemas externos) e o sistema, representando as funcionalidades principais (casos de uso) que o sistema deve realizar.

Elementos principais:

- **Ator**: Representa uma entidade que interage com o sistema (pode ser uma pessoa ou outro sistema).
- **Caso de uso**: Representa uma funcionalidade ou comportamento do sistema.
- **Relacionamento**: Conecta atores aos casos de uso com os quais eles interagem.

Exemplo de Diagrama de Caso de Uso:



Neste exemplo, temos dois atores: **Visitante** e **Cuidador**, e dois casos de uso: **Visitar Zoo** e **Cuidar Animais**.

Diagrama de Classes

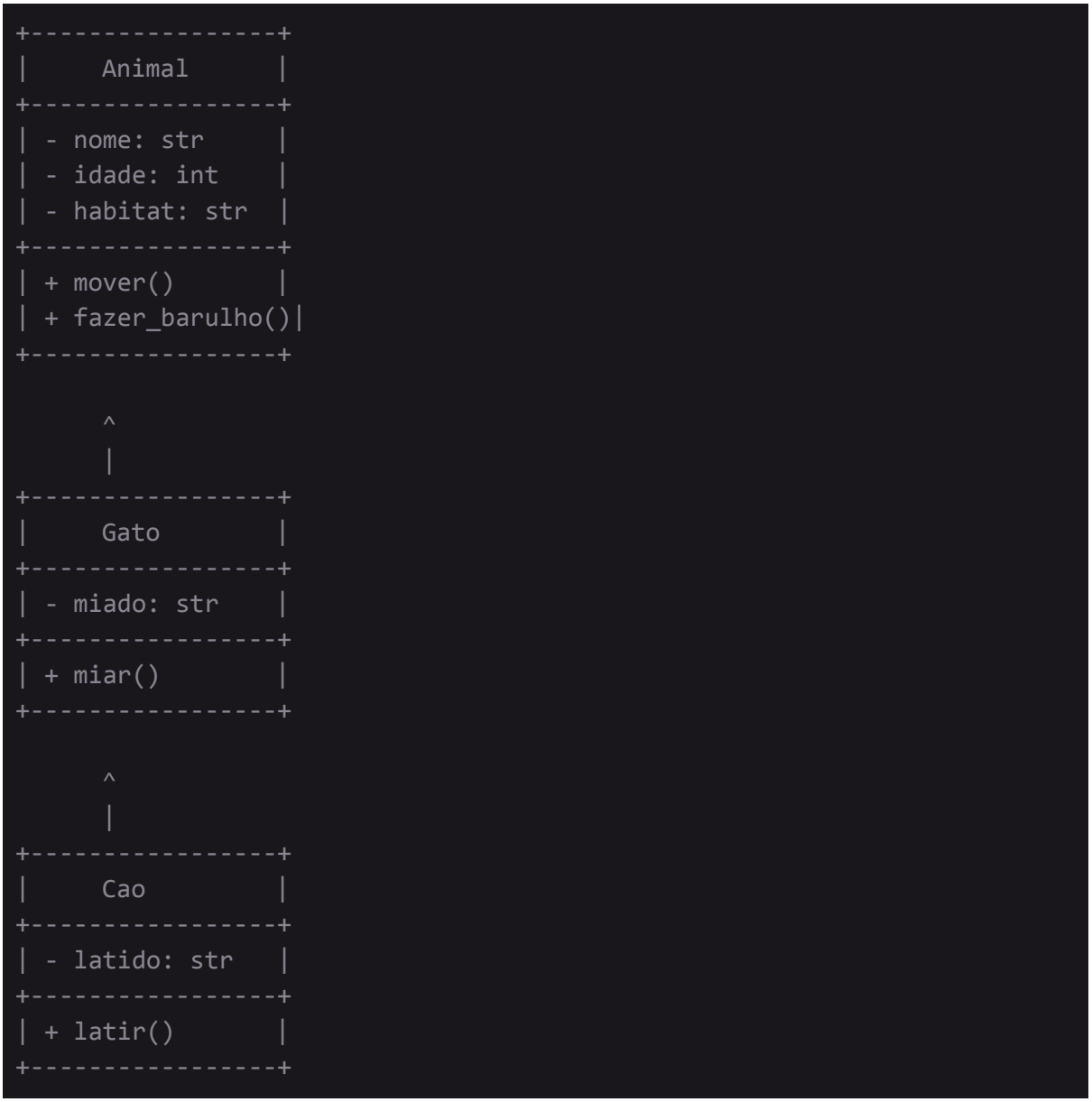
O diagrama de classes é utilizado para descrever a estrutura de um sistema, mostrando as classes, seus atributos, métodos e os relacionamentos entre elas.

Elementos principais:

- **Classe**: Representada como um retângulo com três divisões. A primeira divisão é o nome da classe, a segunda contém os atributos, e a terceira contém os métodos.

- **Relacionamentos**: Podem ser associações, generalizações (herança) e dependências.
- **Atributos e métodos**: Atributos são as propriedades de uma classe, e métodos são as operações que podem ser realizadas.

Exemplo de Diagrama de Classes:



Aqui temos a classe base **Animal** com os atributos `nome`, `idade` e `habitat`, além dos métodos `mover()` e `fazer_barulho()`. As classes **Gato** e **Cao** herdam de **Animal**, e cada uma possui seu próprio método especializado (`miar()` e `latir()`).

Análise e Modelagem de Sistemas

A UML ajuda na análise e modelagem de sistemas orientados a objetos, permitindo que os requisitos sejam identificados e organizados de maneira visual.

Análise de Requisitos usando UML

1. **Identificação de atores**: Quem ou o que irá interagir com o sistema? (ex.: usuários, dispositivos, outros sistemas).
2. **Identificação de casos de uso**: Quais são as funcionalidades principais que o sistema precisa realizar?
3. **Identificação de classes**: Quais são os principais objetos do sistema que têm atributos e comportamentos?

Modelagem de um sistema básico utilizando diagramas UML

Vamos considerar um sistema de gerenciamento de biblioteca. Começamos pela análise de requisitos e, com base nela, criamos os diagramas UML.

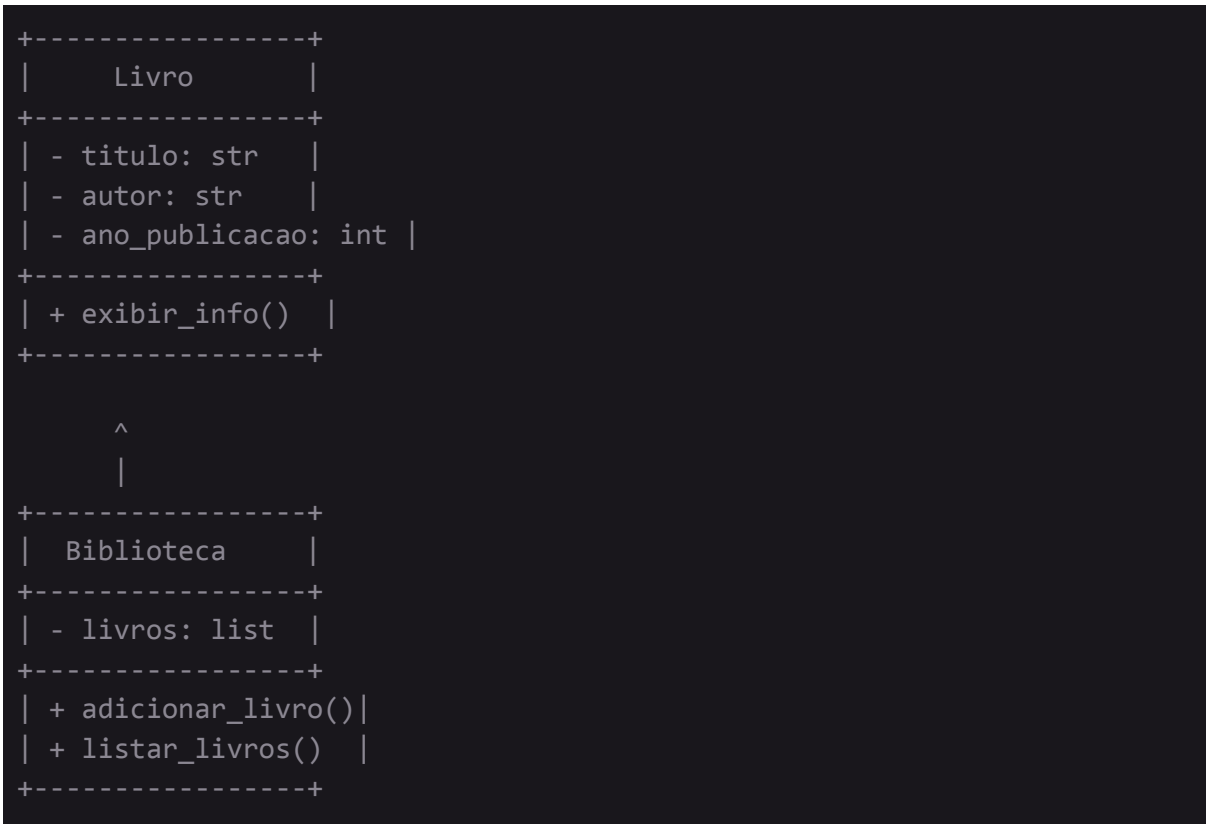
Programa Exemplo em Python

Vamos criar um sistema simples de gerenciamento de uma biblioteca. Para isso, usaremos as classes **Livro** e **Biblioteca** para ilustrar como podemos modelar o sistema e representar o diagrama UML correspondente.

```
class Livro:
      def __init__(self, titulo, autor, ano_publicacao):
      self.titulo = titulo
      self.autor = autor
      self.ano_publicacao = ano_publicacao

      def exibir_info(self):
      print(f"Título: {self.titulo}")
      print(f"Autor: {self.autor}")
      print(f"Ano de Publicação: {self.ano_publicacao}")

class Biblioteca:
      def __init__(self):
      self.livros = []

      def adicionar_livro(self, livro):
      self.livros.append(livro)
```

```
      def listar_livros(self):
      if self.livros:
            for livro in self.livros:
                  livro.exibir_info()
                  print("-" * 20)
      else:
            print("Nenhum livro cadastrado.")

# Exemplo de uso
biblioteca = Biblioteca()

livro1 = Livro("Python para Iniciantes", "Guido van Rossum", 2023)
livro2 = Livro("Orientação a Objetos com Python", "James Gosling", 2021)

biblioteca.adicionar_livro(livro1)
biblioteca.adicionar_livro(livro2)

biblioteca.listar_livros()
```

Diagrama UML para o Sistema de Biblioteca



Neste exemplo, temos as classes **Livro** e **Biblioteca**, onde **Biblioteca** contém uma lista de livros, demonstrando a relação entre as classes. A **Biblioteca** pode adicionar e listar livros.

Polimorfismo em Python

O polimorfismo ocorre quando diferentes classes compartilham um método comum, mas cada classe implementa esse método de forma específica. No contexto de Python, isso é feito utilizando métodos que são sobrescritos em subclasses.

Vamos expandir o exemplo do zoológico:

Exemplo de Polimorfismo

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

      def fazer_barulho(self):
d
d
d
d
d
d
          raise NotImplementedError("Este método deve ser sobrescrito
pela subclasse.")

class Cao(Animal):
      def fazer_barulho(self):
      return "Latido"

class Gato(Animal):
      def fazer_barulho(self):
      return "Miau"

# Lista de animais
animais = [Cao("Rex", 5), Gato("Mimi", 3)]

# Polimorfismo
for animal in animais:
      print(f"{animal.nome} faz {animal.fazer_barulho()}")
```

Neste exemplo, vemos o uso de polimorfismo com o método `fazer_barulho()`, que é implementado de maneira diferente em cada subclasse. O mesmo método é chamado, mas o comportamento varia dependendo do tipo de animal.

Atividade

Enunciado: "Sistema de Gestão de Zoológico"

Conceitos: Classes, Atributos, Métodos, Herança, Polimorfismo.

Atividade:

- Crie uma classe base **Animal** com atributos como `nome`, `idade`, `barulho_que_faz`, `como_se_move`, `o_que_come`, `habitat`, `animais_vizinhos`, e `horas_de_alimentacao`.
- Crie subclasses para animais específicos como **Leão**, **Elefante** e **Pássaro**, cada um com seu comportamento específico para o método `fazer_barulho`.
- O sistema deve permitir cadastrar os animais e listá-los, exibindo suas informações.
- Demonstre o polimorfismo na prática, criando uma lista de animais e exibindo os barulhos que eles fazem.

Programa Exemplo para Análise de UML:

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

      def mover(self):
      raise NotImplementedError("Subclasse deve implementar esse
método.")

class Leao(Animal):
      def mover(self):
      return "Corre rapidamente."

class Elefante(Animal):
      def mover(self):
      return "Anda lentamente."
```

```
class Passaro(Animal):
      def mover(self):
      return "Voa."

# Exemplo de uso
zoologico = [Leao("Simba", 5, "Savana"), Elefante("Dumbo", 8,
"Floresta"), Passaro("Zazu", 3, "Céu")]

for animal in zoologico:
      print(f"{animal.nome}, que vive no {animal.habitat},
{animal.mover()}")
```

Análise UML

- Crie um diagrama de classes para representar o sistema de gestão de zoológico.
- Identifique as classes, atributos, métodos e os relacionamentos entre elas.

Entrega:

- Utilize uma ferramenta para criação de UML.
- Compartilhe como imagem ou link.
- O link ou imagem deve ser enviado para a atividade "Criação de UML OO" no Google Classroom.

Tópico 6: Interfaces Gráficas em Python

As interfaces gráficas (GUIs) são parte essencial de muitas aplicações modernas, permitindo que os usuários interajam com o software de maneira intuitiva. Em Python, há várias bibliotecas que facilitam a criação dessas interfaces de forma eficiente e simples.

Introdução a Interfaces Gráficas

Conceito e Importância de Interfaces Gráficas (GUIs)

Uma **interface gráfica de usuário (GUI)** é a camada de interação entre o usuário e o software, composta por elementos visuais como botões, menus, formulários e outros componentes gráficos. Ela facilita o uso de um programa, tornando-o acessível a pessoas que não possuem conhecimento técnico avançado.

A importância das GUIs está na sua capacidade de melhorar a experiência do usuário (UX), tornando os sistemas mais fáceis de entender e usar. Em vez de trabalhar com comandos em um terminal de texto, os usuários podem interagir visualmente com o software.

Ferramentas e Bibliotecas para Desenvolvimento de GUIs em Python

Python oferece várias bibliotecas para o desenvolvimento de interfaces gráficas, entre elas:

- **Tkinter**: A biblioteca GUI padrão de Python, amplamente usada pela simplicidade.
- **PyQt**: Uma biblioteca poderosa para desenvolvimento de GUIs mais complexas e sofisticadas.
- **Kivy**: Ideal para a criação de interfaces para dispositivos móveis e desktops, com suporte a multitouch.
- **WxPython**: Outra opção para criação de GUIs robustas e multiplataformas.

Para usar essas bibliotecas, o **pip**, que é o gerenciador de pacotes do Python, facilita a instalação de pacotes adicionais. Para instalar uma biblioteca como o **Tkinter**, por exemplo, você pode utilizar o comando abaixo no terminal:

```
pip install tk
```

Agora, vamos focar em como construir GUIs em Python usando **Tkinter**, explorando desde a criação de interfaces simples até a navegação entre telas e a implementação de ações.

Criação de Interfaces Simples

Estrutura Básica de uma Aplicação GUI

A estrutura de uma aplicação básica usando **Tkinter** envolve a criação de uma janela principal, a inserção de elementos (chamados de widgets), e o controle de eventos, como cliques de botões ou interações com campos de texto.

Abaixo está um exemplo básico de um programa em Tkinter que cria uma janela com um título e um botão:

```
import tkinter as tk

# Criação da janela principal
janela = tk.Tk()
janela.title("Minha primeira GUI")

# Tamanho da janela
janela.geometry("400x300")

# Criação de um botão
botao = tk.Button(janela, text="Clique Aqui!")
botao.pack()

# Iniciar o loop principal
janela.mainloop()
```

Neste exemplo:

- **tk.Tk()**: Cria a janela principal da aplicação.
- **title()**: Define o título da janela.
- **geometry()**: Define o tamanho da janela.
- **Button()**: Cria um botão com o texto "Clique Aqui!".
- **pack()**: Posiciona o botão na janela.

Adicionando Widgets: Labels, Inputs, Sliders, Imagens e Mais

Vamos criar uma interface mais completa com diversos elementos, como botões, labels (rótulos), campos de texto, sliders (barras deslizantes), e uma imagem de perfil arredondada.

```
import tkinter as tk
from tkinter import ttk
from PIL import Image, ImageTk

# Criação da janela principal
janela = tk.Tk()
janela.title("Interface Completa")
janela.geometry("600x400")

# Criação de um Label (rótulo)
label = tk.Label(janela, text="Bem-vindo à Interface Gráfica",
font=("Arial", 16))
label.grid(row=0, column=0, columnspan=2)

# Criação de um campo de entrada (input)
input_texto = tk.Entry(janela, width=30)
input_texto.grid(row=1, column=0, padx=10, pady=10)

# Criação de um botão
botao = tk.Button(janela, text="Enviar")
botao.grid(row=1, column=1)

# Criação de um slider (barra deslizante)
slider = tk.Scale(janela, from_=0, to=100, orient="horizontal")
slider.grid(row=2, column=0, columnspan=2, pady=10)

# Criação de uma imagem de perfil arredondada
imagem = Image.open("perfil.jpg")  # Substitua pelo caminho da sua
imagem
imagem = imagem.resize((100, 100))
imagem = ImageTk.PhotoImage(imagem)
label_imagem = tk.Label(janela, image=imagem)
label_imagem.grid(row=3, column=0, columnspan=2)

# Criação de um Frame (quadro) para agrupar widgets
frame = tk.Frame(janela, borderwidth=2, relief="solid")
frame.grid(row=4, column=0, columnspan=2, padx=10, pady=10)

# Adicionando botões dentro do frame
botao1 = tk.Button(frame, text="Botão 1")
botao1.pack(side="left", padx=5)
```

```
botao2 = tk.Button(frame, text="Botão 2")
botao2.pack(side="left", padx=5)

# Loop principal da janela
janela.mainloop()
```

Neste exemplo, usamos vários widgets e ferramentas de layout como **grid()** para organizar os elementos na janela:

- **Label**: Exibe um texto na tela.
- **Entry**: Um campo de texto para o usuário digitar algo.
- **Scale**: Um controle deslizante que permite selecionar valores entre um intervalo.
- **Image**: Para exibir uma imagem de perfil arredondada.
- **Frame**: Agrupa widgets para organização.

Navegação entre Telas

Para criar interfaces mais complexas, pode ser necessário permitir que o usuário navegue entre diferentes telas. Isso pode ser feito criando diferentes frames e alternando entre eles.

```
import tkinter as tk

def mostrar_tela1():
      frame_tela2.pack_forget()
      frame_tela1.pack()

def mostrar_tela2():
      frame_tela1.pack_forget()
      frame_tela2.pack()

def change_frame(current, next):
    current.pack_forget()
    next.pack()

# Janela principal
janela = tk.Tk()
janela.geometry("400x300")

# Frame para a primeira tela
frame_tela1 = tk.Frame(janela)
botao_tela2 = tk.Button(frame_tela1, text="Ir para Tela 2",
command=mostrar_tela2)
botao_tela2.pack(pady=20)
frame_tela1.pack()
```

```
# Frame para a segunda tela
frame_tela2 = tk.Frame(janela)
botao_tela1 = tk.Button(frame_tela2, text="Voltar para Tela 1",
command=mostrar_tela1)
botao_tela1.pack(pady=20)

janela.mainloop()
```

Nesse código, temos duas telas (frames) e botões que alternam entre elas usando o método **pack_forget()** para esconder uma tela e **pack()** para exibir a outra.

Criação de Ações para os Elementos

Os elementos de uma interface gráfica precisam ser interativos, e para isso associamos **ações** (eventos) aos widgets. Abaixo, criamos um exemplo onde, ao clicar em um botão, o conteúdo de um campo de texto é exibido em um **Label**.

```
import tkinter as tk

def mostrar_texto():
      texto = input_texto.get()
      label_texto.config(text=texto)

# Janela principal
janela = tk.Tk()
janela.geometry("300x200")

# Campo de texto
input_texto = tk.Entry(janela)
input_texto.pack(pady=10)

# Botão que chama a função mostrar_texto
botao = tk.Button(janela, text="Exibir Texto", command=mostrar_texto)
botao.pack(pady=10)

# Label que exibe o texto digitado
label_texto = tk.Label(janela, text="")
label_texto.pack(pady=10)

janela.mainloop()
```

Neste exemplo:

- O botão **botao** está ligado à função **mostrar_texto**, que é chamada quando o botão é pressionado.
- A função **mostrar_texto** captura o texto do **input_texto** e o exibe no **label_texto**.

Enunciado da Atividade: Calculadora GUI OO

Objetivo:

Desenvolver uma calculadora utilizando a linguagem Python, a biblioteca Tkinter para criar a interface gráfica e os conceitos de Programação Orientada a Objetos (POO). A calculadora deve permitir realizar as operações básicas de adição, subtração, multiplicação e divisão, além de oferecer funcionalidades como reset e uso do resultado anterior em novas operações.

Requisitos:

1. **Interface Gráfica:**
    - Utilize a biblioteca Tkinter para criar uma interface intuitiva e fácil de usar.
    - A interface deve conter botões para os números de 0 a 9, os operadores (+, -, *, /), o botão de igual (=), o botão de limpar (C) e um display para mostrar os números e resultados.
    - A disposição dos elementos na interface deve ser organizada e esteticamente agradável.
2. **Funcionalidades:**
    - **Operações básicas:** A calculadora deve realizar as quatro operações aritméticas básicas: adição, subtração, multiplicação e divisão.
    - **Histórico de operações:** A calculadora deve armazenar o resultado da última operação, permitindo que este seja utilizado como um dos operandos da próxima operação.
    - **Botão de reset:** Ao clicar no botão "C", a calculadora deve limpar o display e reiniciar as operações.
    - **Tratamento de erros:** A calculadora deve tratar erros como divisão por zero e operações inválidas, exibindo uma mensagem de erro apropriada.
3. **Programação Orientada a Objetos:**
    - Utilize classes para representar os diferentes componentes da calculadora, como botões, display e a própria calculadora.
    - Utilize métodos para implementar as funcionalidades da calculadora, como realizar cálculos, atualizar o display e tratar eventos.
    - Aplique os conceitos de herança e polimorfismo, se possível, para organizar o código de forma mais eficiente.
4. **Entrega:**
    - Crie um novo repositório no GitHub para armazenar o código fonte do seu projeto.

- Organize o código em arquivos separados para cada classe e para o arquivo principal da aplicação.
- Inclua um arquivo README.md no repositório, explicando como executar o programa e as principais decisões de design.
- Envie o link do seu repositório para a atividade "Calculadora GUI OO" no Google Classroom.

Dicas:

- **Planejamento:** Antes de começar a codificar, crie um diagrama de classes para visualizar a estrutura do seu programa.
- **Modularização:** Divida o código em funções e métodos menores para facilitar a organização e a manutenção.
- **Teste:** Teste a sua calculadora com diferentes combinações de números e operações para garantir que ela funcione corretamente.
- **Estilo de código:** Utilize um estilo de codificação consistente para melhorar a legibilidade do código.

Exemplo de estrutura básica:

import tkinter as tk

```
class Calculadora:
      def __init__(self, tk):
      # ... inicialização da interface ...

      def calcular(self, operacao):
      # ... implementação dos cálculos ...

# ... outras classes e funções ...

if __name__ == "__main__":
      calculadora = Calculadora()
      calculadora.rodar()
```