Comentários - Calculadora Matricial

1. Hierarquia de Classes

1.1. Classe Base: `Matrix`

- **Propósito**: Representar uma matriz genérica `m x n` de números reais.
- **Atributos**:
  - `rows`: Número de linhas (int)
  - `cols`: Número de colunas (int)
  - `data`: Estrutura de dados interna para armazenar os elementos. Para `Matrix` e `SquareMatrix`, será um 'array de arrays' (lista de listas em Python) para representar a matriz densa.
- **Métodos (a serem sobrecarregados ou implementados nas subclasses)**:
  - `__add__(self, other)`: Sobrecarga do operador `+` para soma matricial.
  - `__sub__(self, other)`: Sobrecarga do operador `-` para subtração matricial.
  - `__mul__(self, other)`: Sobrecarga do operador `*` para multiplicação por escalar (se `other` for um número) ou multiplicação matricial (se `other` for uma `Matrix`).
  - `transpose()`: Retorna a matriz transposta.
  - `get_element(row, col)`: Retorna o elemento na posição `(row, col)`.
  - `set_element(row, col, value)`: Define o elemento na posição `(row, col)`.
  - `is_square()`: Verifica se a matriz é quadrada.
  - `is_lower_triangular()`: Verifica se a matriz é triangular inferior.
  - `is_upper_triangular()`: Verifica se a matriz é triangular superior.
  - `is_diagonal()`: Verifica se a matriz é diagonal.
  - `to_string()`: Retorna uma representação em string da matriz para impressão.

1.2. Classe Derivada: `SquareMatrix`

- **Propósito**: Representar uma matriz quadrada `n x n`.
- **Herda de**: `Matrix`
- **Atributos**:
  - `data`: Utiliza a mesma estrutura 'array de arrays' da classe base.

- **Métodos Adicionais**:
  - `trace()`: Calcula o traço da matriz (soma dos elementos da diagonal principal).

1.3. Classe Derivada: `LowerTriangularMatrix`

- **Propósito**: Representar uma matriz triangular inferior.
- **Herda de**: `SquareMatrix` (pois triangular é sempre quadrada)
- **Atributos**:
  - `data`: 'array de listas' (lista de listas em Python), onde cada lista interna armazena apenas os elementos não nulos da linha (até a diagonal principal).
- **Métodos Adicionais**:
  - `determinant()`: Calcula o determinante da matriz (produto dos elementos da diagonal principal).
  - **Otimização**: Métodos especializados para soma e multiplicação por escalar que operam apenas nos elementos armazenados.

1.4. Classe Derivada: `UpperTriangularMatrix`

- **Propósito**: Representar uma matriz triangular superior.
- **Herda de**: `SquareMatrix`
- **Atributos**:
  - `data`: 'array de listas' (lista de listas em Python), onde cada lista interna armazena apenas os elementos não nulos da linha (da diagonal principal em diante).
- **Métodos Adicionais**:
  - `determinant()`: Calcula o determinante da matriz (produto dos elementos da diagonal principal).
  - **Otimização**: Métodos especializados para soma e multiplicação por escalar que operam apenas nos elementos armazenados.

1.5. Classe Derivada: `DiagonalMatrix`

- **Propósito**: Representar uma matriz diagonal.
- **Herda de**: `LowerTriangularMatrix` e `UpperTriangularMatrix` (ou diretamente de `SquareMatrix` e ter métodos próprios para otimização, a ser decidido na implementação).
- **Atributos**:
  - `data`: 'array simples' (lista em Python) para armazenar apenas os elementos da diagonal principal.

- **Métodos Adicionais**:
  - `determinant()`: Calcula o determinante da matriz (produto dos elementos da diagonal principal).
  - `trace()`: Calcula o traço da matriz (soma dos elementos da diagonal principal).
  - **Otimização**: Métodos especializados para soma e multiplicação por escalar que operam apenas nos elementos da diagonal.

2. Reconhecimento da Classe da Matriz e Otimização de Memória

- Ao inserir uma nova matriz, o programa deverá verificar suas propriedades (`is_square`, `is_lower_triangular`, `is_upper_triangular`, `is_diagonal`) e instanciar a classe mais específica que se encaixe, utilizando a estrutura de dados otimizada para aquela classe.
- A ordem de verificação será: `DiagonalMatrix` -> `LowerTriangularMatrix` / `UpperTriangularMatrix` -> `SquareMatrix` -> `Matrix`.

3. Operações Matriciais com Sobrecarga de Operadores e Métodos Especializados

- As operações `+`, `-`, `*` serão sobrecarregadas na classe base `Matrix`.
- Dentro desses métodos sobrecarregados, haverá lógica para verificar os tipos das matrizes operadas.
- Se ambas as matrizes forem do mesmo tipo especializado (ex: `DiagonalMatrix` + `DiagonalMatrix`), um método especializado será chamado para realizar a operação de forma otimizada, retornando uma matriz do mesmo tipo especializado.
- Se os tipos forem diferentes, ou se a otimização não for aplicável, a operação genérica será realizada, e o resultado será do tipo mais genérico que se aplique (ex: `DiagonalMatrix` + `Matrix` = `Matrix`).
- **Tratamento de Exceções**: As operações deverão incluir `try-except` para lidar com incompatibilidades de dimensões.

4. Menu de Gerenciamento de Matrizes

- Será implementado um loop de menu interativo que permitirá ao usuário:
  1. Imprimir uma, ou mais, matrizes da lista.

2. Inserir uma nova matriz lida do teclado ou de um arquivo.
3. Inserir uma matriz identidade `n x n`.
4. Alterar ou remover uma, ou mais matrizes da lista.
5. Apresentar a lista de matrizes com identificação por TIPO, dimensões ou nome.
6. Gravar a lista em arquivo (backup).
7. Ler outra lista de matrizes de arquivo (acrescentar ou substituir).
8. Zerar a lista de matrizes.

5. Próximos Passos

- Implementar a classe base `Matrix` e seus métodos genéricos.
- Implementar as estruturas de dados internas para cada tipo de matriz.
- Implementar os métodos de verificação de tipo (`is_square`, etc.).