Overview: In this lab, you will review previous Python experience, and work on some new concepts, by writing a class that represents a mathematical matrix object that consists of integer values.

Assignment: For this lab, enter your code in VS Code (or your editor/IDE of choice), and then use the terminal to execute your program. All of your code should go in the same source file, which you should name `Matrix.py`.

- The name of your class must be `Matrix`.
- Use the `__slots__` capability to define your instance variables.
- You are **not allowed to import <u>any</u> libraries** for your solution. For example, use of the `numpy` library, etc., is prohibited. The goal is for you to think about algorithmic implementations here, rather than relying on external libraries to quickly solve your task.
- You **<u>must</u>** use a Python `list` type for your instance variable that stores the matrix data. (Whether you internally store as a 1D list, and then handle all the 2D translation inside particular methods, or store directly as a 2D list (list of lists) is up to you. However, I recommend the latter.)
- There will be two different ways to initialize a `Matrix` object:
  - either by allowing the user to specify (in order) the number of rows, the number of columns, and a single 1D list containing all the necessary (integer) data given in row-major order; or
  - allowing the user to specify a filename, from which you will read the matrix information.

  For example:

  ```
  m = Matrix(3, 2, [1, 2, 3, 4, 5, 6])
  ```

  should define a 3×2 matrix whose first row contains 1 and 2, whose second row contains 3 and 4, and whose third row contains 5 and 6. Raise a `ValueError` with a reasonable message if the dimensions do not match the amount of data given.

  Alternatively, if the construction happens as:

  ```
  m = Matrix(filename = "matrix_input.txt")
  ```

  then you must read the corresponding file for the matrix input, assuming that the file can consist of any number of lines, but each line of the file will have exactly the same number of integers. An example for the previous 3×2 matrix is shown below. Raise a `ValueError` with a reasonable message if the provided file does not have the appropriate format (e.g., inconsistent number of integers per line).

  ```
  1 2
  3 4
  5 6
  ```

  Hint: Since a Python class can have only a single `__init__` method, your method needs to:

  - include parameters for the number of rows, number of columns, and the data — each having default values;
  - include a special `*` parameter, indicating that what follows must be a keyword argument (i.e., you must specify the name of the parameter when you provide the argument, as in the second construction example above);
  - include a keyword-based parameter for the filename.

  We will discuss this in class. You can also visit this Real Python link for more help.

  My advice would be to have two private (i.e., begin with an underscore) helper methods: one to handle when the data is provided directly, and the other to handle when a filename is provided. Call these methods appropriately from `__init__`.

- Include two getters: `getNumRows(self)` and `getNumCols(self)`.

- Include a `__str__(self)` method to allow direct printing of a `Matrix` object. I want you to follow a strict format for this. Specifically:
  - For each row in your resulting string, begin with a pipe (|) and two spaces, and end with a pipe.
  - All values in that row (between the items above) should be right-justified, using the width of the longest value (i.e., the value having the longest length when represented as a string, which may be a negative value) for the justification width.
  - Include two spaces after each value in a row.
  - Include a newline after the pipe that ends that row, except for the last row.

  For example, given the following `Matrix` construction:

  ```
  m_2x3 = Matrix(2, 3, [12,234,3456,12345,23,3])
  ```

  the resulting string built and returned should be the equivalent of:

  ```
  "|     12    234   3456  |\n|  12345     23      3  |"
  ```

  (noting the width of 12345 dictates the justification width of all other entries) so that, when printed via `print(m_2x3)`, the output will look like:

  ```
  |     12    234   3456  |
  |  12345     23      3  |
  ```

Then, implement each of the following, raising an exception whenever appropriate:

- `__getitem__(self, row_col: tuple[int]) -> int:`
  - Returns a value from the matrix at the specified row,col tuple (assume both start at 0).
  - For example, for the `Matrix` object immediately above, using `m_2x3[1,1]` will return 23, the entry at the (1,1) position (second row, second column).
  - Raise an `IndexError` exception, with meaningful message, if the specified tuple has invalid indices.
- `__eq__(self, other: 'Matrix') -> bool:`
  - Allows two objects to be compared using ==.
  - Two matrices are equal if they have the same number of rows, the same number of columns, and the data values are identical at each location.
  - Note that for list(s) of integers (even a 2D list of integers), using == on those lists will do the right thing here. (You are using list(s) as an instance variable, correct?)
  - Raise a `TypeError` exception, with meaningful message, if the `other` is not an instance of the `Matrix` class.
- `__add__(self, other: 'Matrix') -> 'Matrix':`
  - Allows two objects to be added using +.
  - You must create and build a new `Matrix` object that results from the addition of the two given objects, and return that new object.
  - This must not modify either the `self` object nor the `other` object – i.e., those are immutable.
  - To add two matrices, they must have the same number of rows and the same number of columns. The result will be of the same dimensions as the two operands. See `https://en.wikipedia.org/wiki/Matrix_addition`.
  - Raise a `ValueError` exception, with a reasonable message, if the dimensions are not appropriate for adding.
- `transpose(self) -> 'Matrix':`
  - Returns a new matrix that is the transpose of the given matrix.
  - You must create and build a new `Matrix` object.
  - This must not modify the `self` object i.e., the original object is immutable.
  - The transpose of an $r \times c$ matrix will be a $c \times r$ matrix, where each column in the original becomes a row in the transpose. See `https://en.wikipedia.org/wiki/Transpose`.

- `__mul__(self, other: 'Matrix') -> 'Matrix'`:
  - Returns a new matrix that is the product of the two matrices (allows multiplying two `Matrix` objects using `*`).
  - You must create and build a new `Matrix` object that results from the multiplication of the two given objects, and return that new object.
  - This must not modify either the `self` object nor the `other` object – i.e., those are immutable.
  - To multiply two matrices, the number of columns in the first matrix must equal the number of rows in the second matrix.
  - Raise a `ValueError` exception, with a reasonable message, if the dimensions are not suitable for multiplying.
  - The product of an $r_1 \times c_1$ matrix and an $r_2 \times c_2$ matrix requires that c_1 and r_2 be the same, and will result in a new $r_1 \times c_2$ matrix. The details for computing the so-called "dot products" that become the values in the new matrix can be found here: https://www.mathsisfun.com/algebra/matrix-multiplying.html.

    Note: I strongly encourage you to try the matrix multiplication in a separate tester program until you get the logic correct. Then you can incorporate that logic into your `__mul__` method here.

Again, ensure that **objects passed as arguments are immutable** (i.e., that your methods do not alter the objects).

Reflection: In the online Lyceum text, provide answers to each of the following:

1. What grade (numeric and letter) would you objectively give to your work on this assignment? Justify your choice.
2. What was challenging on this assignment, if anything?
3. What do you still need work on, in terms of concepts covered in this assignment?
4. List all online-help resources you used in completing this assignment (which you should have also included as comments in your program's top block comment). You do not have to cite Python's documentation.

Style: Use good style, including good naming conventions, type hints, docstrings, plenty of comments when appropriate, and use of exceptions when appropriate.

Testing: In your `main` function inside `Matrix.py`, make sure to include at least three different executing tests for each of your methods above. For the methods that potentially raise exceptions, make sure to include additional tests (each wrapped in its own try-except block) inside `main`.

Ed Autograder: When you are ready to test your solution against the autograder, visit the appropriate Ed link provided on Lyceum, and follow the same process that we used for the previous testing.