Object-Oriented Programming
50:198:113 (Fall 2022)

Homework:	**4**	**Professor:**	**Suneeta Ramaswami**
Due Date:	**11/11/22**	**URL:**	`https://sites.rutgers.edu/suneeta-ramaswami`
Office:	**321 BSB**	**E-mail:**	`suneeta.ramaswami@rutgers.edu`
		Phone:	**(856)-225-6439**

Homework Assignment 4

The assignment is due by 11:59PM of the due date. The point value is indicated in square braces next to each problem. Each solution must be the student's own work. Assistance should be sought or accepted only from the course instructor. Any violation of this rule will be dealt with harshly.

The first problem of this assignment requires you to add overloaded operators to the `Time` class from Homework #3. In addition, you are asked to implement some functions (outside the class) that manipulate `Time` objects. The second problem requires the implementation of several functions for *matrices* (these are defined in Problem 2), and the third problem requires you to create a `Matrix` class. As in past (and future) assignments, you are graded not only on the correctness of the code, but also on clarity and readability. Hence, I will deduct points for poor indentation, poor choice of object names, and lack of documentation. For documentation, all functions, classes, and methods should have appropriate docstring documentation. For the rest of your code, use a common sense approach. While I do not expect every line of code to be explained, all code blocks that carry out a significant task should be documented *briefly* in clear English.

Download from Canvas the homework document `hw4.pdf`, the module `time.py` for Problem 1, the test file `test_time.py` (for Problem 1), and the file `matrix.py` (for Problem 2).

Important note: When writing each of the following programs, it is important that you name the functions exactly as described because I will assume you are doing so when testing your programs. If your program produces errors because the functions do not satisfy the stated prototype, points will be deducted. You may not use the `datetime` module anywhere in your solution for this problem.

Problem 1 [17 points] Time class. In Problem 2 of the previous homework assignment, you implemented some methods of a `Time` class. In this problem, you are asked to build on this class further by adding overloaded operators. These are described below. (*Please refer to the problem 2 description in Homework 2 if necessary.*)

Note: Download the module `time.py` from Canvas. It contains my implementation of the `Time` class methods from Homework #3. Insert your code into this module. *You may not use the* `datetime` *module anywhere in your solution for this problem.*

1. `__add__`: This method overloads the `+` operator. It returns the "sum" of a `Time` instance and a non-negative integer `mins`. The sum of a `Time` instance `T` and the non-negative integer `mins` is the `Time` instance that occurs `mins` minutes **after** `T`. For example, if `T` is the time instance 9:15AM, then `T + 653` should return the `Time` instance `8:08PM`. *Hint:* The method `total_minutes()` might be useful here. Alternatively, you may find `minute_ahead()` useful.

2. `__sub__`: This method overloads the `-` operator and subtracts a non-negative integer from a `Time` instance. The difference of a `Time` instance `T` and the non-negative integer `mins` is the `Time` instance that occurs `mins` minutes **before** `T`. For example, if `T` is the time instance 9:15AM, then `T - 653` should return the `Time` instance `10:22PM`. *Hint:* The methods `total_minutes()` or `minute_back()` may come in useful here.
3. `__lt__`: This method overloads the `<` (less than) operator and compares two `Time` objects. It returns `True` if the `Time` instance on the left hand side of the `<` operator occurs strictly before the `Time` instance on the right hand side of the `<` operator. Note that the comparison is made for a 24-hour time period beginning at midnight. (Hence, AM times always occur before PM times.)
4. `__gt__`: This method overloads the `>` operator and returns a `True` or `False` value. The meaning of this operator should be obvious from the explanation for the `<` operator above.
5. `__le__`: This method overloads the `<=` operator and returns a `True` or `False` value.
6. `__ge__`: This method overloads the `>=` operator and returns a `True` or `False` value.
7. `__eq__`: This method overloads the `==` operator and returns a `True` or `False` value.
8. `__ne__`: This method overloads the `!=` operator and returns a `True` or `False` value.

In addition, implement the following functions. **Note:** These are regular functions, **not** `Time` methods. **Do not manipulate `Time` instance attributes directly to implement these functions.** You are required to use `Time` methods to implement this function.

- Implement a function called `time_interval` with two parameters, `T1` and `T2`, both of which are `Time` objects. This function returns the number of hours and minutes in the time interval between `T1` and `T2`. The function should return *a pair* `(hrs, mins)`, where `hrs` is the (maximal) number of hours and `mins` is the number of minutes between `T1` and `T2`. For example, if `T1` is the time 10:45AM and `T2` is the time 4:20PM, then `time_interval(T1, T2)` should return (5, 35) because there are 5 hours and 35 minutes between `T1` and `T2`.
- Implement a function called `time_schedule` with three parameters: `start_time` (a `Time` object), `duration` (an integer), and `N` (an integer). The function should return a list of length `N`. The list contains the `N` `Time`s that occur every `duration` minutes starting at `start_time`. For example, if `T1` is the time 10:45AM, then `L = time_schedule(T1, 12, 10)` creates a list `L` containing the 10 times that occur every 12 minutes starting at 10:45AM. Therefore, `print(L)` should print `[10:45AM, 10:57AM, 11:09AM, 11:21AM, 11:33AM, 11:45AM, 11:57AM, 12:09PM, 12:21PM, 12:33PM]`. **Note:** The function returns a list of `Time` objects, **not** a list of strings.

Complete the implementation of the `Time` class, `time_interval` function, and `time_schedule` function in the module `time.py` (*you must download this module from Canvas*). I have also provided a test file called `test_time.py` that you can use to test your implementation. That module will simply import the `time.py` module. Edit `time.py` to implement the overloaded operators for `Time` as described above. When you are ready to test your implementation, type `python3 test_time.py` to test your implementation.

Problem 2 [28 points] Matrix Operations. This problem requires you to work with a list of lists. *You are expected to use list comprehension whenever suitable* in this module. Download module `matrix.py` from Canvas. Insert your implementation at the beginning of the module. This module contains some test code for you to test the implementation of your functions.

In this problem, you are asked to write several functions to manipulate *matrices*. An $m \times n$ *matrix* is a rectangular array of numbers consisting of m rows and n columns. If $m = n$, the matrix is said to be a *square* matrix. An example of a 2×3 matrix A and a 3×3 (square) matrix B is shown below.

$$A = \begin{bmatrix} 5 & 3 & -1 \\ 9 & 4 & 12 \end{bmatrix}$$

$$B = \begin{bmatrix} 6 & 9 & 12 \\ -8 & 6 & -4 \\ 7 & 11 & 13 \end{bmatrix}$$

For an $m \times n$ matrix A, the element of the matrix in the i-th row and j-th column is denoted by a_{ij}, where $1 \leq i \leq m$ and $1 \leq j \leq n$. Using this notation, we can now define a number of operations on matrices.

- The **sum** of two $m \times n$ matrices A and B is an $m \times n$ matrix C such that $c_{ij} = a_{ij} + b_{ij}$. We write this as $C = A + B$.
- The **difference** of two $m \times n$ matrices A and B is an $m \times n$ matrix C such that $c_{ij} = a_{ij} - b_{ij}$. We write this as $C = A - B$.
- The **product** of two matrices A and B is defined when A is an $m \times k$ matrix and B is a $k \times n$ matrix (that is, the number of columns in A is the same as the number of rows in B). In this case, the product C of A and B, written as $C = AB$, is an $m \times n$ matrix, where c_{ij} is obtained by multiplying the i-th row of A with the j-th column of B as follows:

  $$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + a_{i3}b_{3j} + \ldots + a_{ik}b_{kj}$$

  Note that the number of elements in the i-th row of A is equal to the number of elements in the j-th column of B.
- The **determinant** of a *square* matrix A, denoted $\det(A)$, is a function that calculates a real value from a matrix. It is defined as follows:

  $$\text{If } A = [a], \det(A) = a$$

  $$\text{If } A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}, \det(A) = ad - bc$$

  For all larger $n \times n$ matrices, let A_{1j} denote the $(n-1) \times (n-1)$ matrix obtained by deleted row 1 and column j from A. Then $\det(A)$ is defined recursively as follows:

  $$\det(A) = a_{11}\det(A_{11}) - a_{12}\det(A_{12}) + a_{13}\det(A_{13}) - a_{14}\det(A_{14}) + \ldots + (-1)^{n-1}a_{1n}\det(A_{1n})$$

Here are some examples to illustrate the above matrix operations. Let A and B be the example matrices shown on the previous page. In addition, let

$$C = \begin{bmatrix} 0 & -21 & -1 \\ 11 & 13 & 17 \end{bmatrix}$$

Then, we have

$$A + C = \begin{bmatrix} 5 & -18 & -2 \\ 20 & 17 & 29 \end{bmatrix}$$

$$A - C = \begin{bmatrix} 5 & 24 & 0 \\ -2 & -9 & -5 \end{bmatrix}$$

$$AB = \begin{bmatrix} 30 - 24 - 7 & 45 + 18 - 11 & 60 - 12 - 13 \\ 54 - 32 + 84 & 81 + 24 + 132 & 108 - 16 + 156 \end{bmatrix} = \begin{bmatrix} -1 & 52 & 35 \\ 106 & 237 & 248 \end{bmatrix}$$

$$\begin{aligned} \det(B) &= 6\det(B_{11}) - 9\det(B_{12}) + 12\det(B_{13}) \\ &= 6\det(\begin{bmatrix} 6 & -4 \\ 11 & 13 \end{bmatrix}) - 9\det(\begin{bmatrix} -8 & -4 \\ 7 & 13 \end{bmatrix}) + 12\det(\begin{bmatrix} -8 & 6 \\ 7 & 11 \end{bmatrix}) \\ &= 6(78 + 44) - 9(-104 + 28) + 12(-88 - 42) \\ &= 732 + 684 - 1560 \\ &= -144 \end{aligned}$$

In our Python program, we will represent an $m \times n$ matrix as a list of m elements (one for each row), where each element is itself a list of n numbers. In other words, a matrix will be stored as a *list of lists*. For example, the above matrix A will be represented as `A = [[5, 3, -2], [9, 4, 12]]`. Note that in the mathematical matrix notation, the row numbers go from 1 to m and the column numbers go from 1 to n. Hence, row i of the matrix is the $(i-1)$-th element of `A`, and column j of the matrix is the $(j-1)$-th element of each list in `A`. This means that the element a_{ij} is stored in `A[i-1][j-1]`.

In order to implement the above matrix operations in Python, you are asked to write the following functions. **Important:** Do not import any modules when implementing or testing this program.

1. *(1 point)* A function called `dimension` with a single parameter, a matrix `M`. The function returns the number of rows and the number of columns in `M`. The returned object is thus a 2-tuple.
2. *(1 point)* A function called `row` with two parameters, a matrix `M` and a positive integer `i`. The function returns the `i`-th row of `M`. Recall that row numbers in a matrix start at 1. Raise an exception if `i` is an invalid number. Observe that the returned object is a list.
3. *(3 points)* A function called `column` with two parameters, a matrix `M` and a positive integer `j`. The function returns the `j`-th column of `M`. Recall that row numbers in a matrix start at 1. Raise an exception if `j` is an invalid number. Observe that the returned object is a list.

4. *(3 points)* A function called `matrix_sum` with two parameters, a matrix `A` and a matrix `B`. If `A` and `B` have the same dimensions, the function should return the matrix sum of `A` and `B`. If `A` and `B` do not have the same dimensions, the function should raise an exception.
5. *(2 points)* A function called `matrix_difference` with two parameters, a matrix `A` and a matrix `B`. If `A` and `B` have the same dimensions, the function should return the matrix difference of `A` and `B`. If `A` and `B` do not have the same dimensions, the function should raise an exception.
6. *(6 points)* A function called `matrix_product` with two parameters, a matrix `A` and a matrix `B`. If `A` and `B` are product compatible (that is, if the number of columns in `A` is equal to the number of rows in `B`), then the function should return the matrix product of `A` and `B`. If they are not not product compatible, the function should raise an exception. Use functions `row` and `column` to implement this function.
7. *(4 points)* A function called `reduce_matrix` with three parameters: a matrix `M`, a positive integer `i`, and a positive integer `j`. The function returns the matrix obtained from `M` by removing the `i`-th row and `j`-th column of `M`. Raise an exception if `i` and `j` are invalid numbers for `M`. Note that the row and column dimensions of the returned matrix are one less than the row and column dimensions of `M`. *Important:* Do not modify `M` itself when computing the reduced matrix. Create a *new* matrix with the reduced dimensions and return that.
8. *(6 points)* A function called `determinant` with a single parameter, a matrix `M`. If the matrix is *not* a square matrix, the function should raise an exception. If it is a square matrix, the function returns the determinant of `M`. Implement the function recursively; the function `reduce_matrix` will come in handy here.
9. *(2 points)* A function called `pretty_print` with a single parameter, a matrix `M`. The function should print the matrix in a neatly formatted way (use the usual row-wise order); use string formatting with field widths. We use this function to print the matrix in a readable format, since printing it out as a list of lists is not easy to read, particularly for large matrices.

Problem 3 [30 points] A Matrix Class. In Problem 2, you wrote several functions to manipulate matrices. Indeed, a matrix is a good example of a new type of object for which it would make sense to use object-oriented design. In this problem, you will *create* a class called `Matrix` in a module called `matrixclass.py`. The methods you are asked to implement are very similar to the functions you were asked to implement in Problem 2. In fact, you should adapt the code in the previous problem to implement this class. You will also need to <u>rewrite</u> the test code (the code that appears in the body of the `if __name__ == "__main__"` block) to test your `Matrix` class. In other words, you are asked to mimic the test code provided in the previous problem by creating `Matrix` instances and making appropriate calls to `Matrix` methods to test out your `Matrix` class implementation.

Implement the following methods for your `Matrix` class. Recall that all class methods must have `self` as the first parameter.

- __init__: The constructor initializes the matrix by storing its elements as a list of lists. Hence, the constructor will have a list of equal-length lists as its parameter. Raise an exception if the parameter is not a list of equal-length lists. **<u>Important note:</u>** The instance attribute must be private.

- `dimension(self)`: Returns the dimension of the matrix.
- `row(self, i)`: Returns the `i`-th row of the matrix. Raise an exception if `i` is an invalid row number.
- `column(self, j)`: Returns the `j`-th column of the matrix. Raise an exception if `j` is an invalid row number.
- `__add__`: The overloaded operator that returns the matrix sum of its two matrix parameters. Note that a `Matrix` object is returned. Raise an exception if the matrices being added do not have the same dimensions.
- `__sub__`: The overloaded operator that returns the matrix difference of its two matrix parameters. Note that a `Matrix` object is returned. Raise an exception if the matrices being added do not have the same dimensions.
- `__mul__`: The overloaded operator that returns the matrix product of its two matrix parameters. Note that a `Matrix` object is returned. Raise an exception if the matrices being multiplied are not product-compatible.
- `reduce_matrix(self, i, j)`: Returns the matrix obtained by deleting row `i` and column `j` from the matrix. Raise an exception if `i` and `j` are invalid numbers for the matrix. Note that a `Matrix` object is returned.
- `determinant`: Returns the determinant of the matrix. Raise an exception if the matrix is not square.
- `__str__`: Returns a printable string representation of the matrix. The string should be neatly formatted, in a manner similar to the `pretty_print` function of the previous problem.
- Finally, *rewrite* the test code provided to you (in `matrix.py`) to create `Matrix` objects and make appropriate calls to `Matrix` methods.

Point distribution will be similar to Problem 2, but reduced by about 1 point for some of the methods. The remaining points will be assigned to the test code portion of the problem.

Submission Guidelines

Implement the first problem in the file called `time.py` (downloaded from Canvas), the second problem in the file called `matrix.py` (downloaded from Canvas), and the third one in a file called `matrixclass.py`. **Your name and RUID should appear as a comment at the very top of each file.**

Test each of your programs thoroughly before submitting your homework. When you are ready to submit, upload your files on Canvas as follows:

1. Go to the "Assignments" tab of the Canvas site for this course.
2. Click on "Programming Assignment #4" under Homework Assignments.
3. Download the homework document (`hw4.pdf`) and the Python files `time.py`, `test_time.py`, and `matrix.py`.
4. Use this same link to upload your files `time.py`, `matrix.py`, and `matrixclass.py`.

You must submit your assignment at or before 11:59PM on November 11, 2022.