Matt Bonnell

*Épreuve Synthèse*

Computer Programming

Professor Robert Vincent

19 May 2017

**Design of a matrix class for simple matrix operations**

Matt Bonnell, Marianopolis College, Montreal, Quebec, Canada.

In mathematics, a matrix is a two-dimensional array of values or expressions treated as a single entity. The matrix is a particularly useful construct for representing and solving systems of linear equations. The purpose of this project is to define a matrix class in the Python programming language. The class will enable the user to create matrix objects of their desired dimensions, add rows and columns to existing matrices, test their properties and perform basic operations on them such as addition, subtraction, multiplication and transposition, as well as computing their determinant. This paper serves two purposes: firstly, to describe the design criteria and programming methodology of the matrix class and secondly, to serve as documentation of its structure and functionality.

Key words: matrix, mathematics, linear algebra, Python, class, object-oriented programming

Table of Contents

[Design and Programming Techniques 3](#_Toc482980079)

[1. Design 3](#_Toc482980080)

[2. Programming Techniques 3](#_Toc482980081)

[Documentation 5](#_Toc482980082)

[1. Instantiation and manipulation 5](#_Toc482980083)

[2. Property methods 6](#_Toc482980084)

[4. Implementation of special methods 8](#_Toc482980085)

# Design and Programming Techniques

## 1. Design

The Matrix class is a subclass of Python’s list class. As stated in the abstract, a matrix is by definition a two-dimensional array. The list class is Python’s built-in class for handling arrays of values, making it the obvious choice of base class on which to implement a class for representing matrices. From a design perspective, the matrix objects are simply two-dimensional lists.

## 2. Programming Techniques

Subclassing the list class allows the Matrix class to make use of list methods in its internal workings.The list.append(obj) method is used in the initialization of new Matrix objects to add the row arguments to the underlying list, thus building the two-dimensional array which stores the matrix’s entries. The list.append(obj) method is also at the core of the matrix.add\_rows(\*rows) and matrix.add\_columns(\*columns) methods. From the user’s perspective, the add\_rows method may seem superfluous, seeing as the append method ostensibly serves the same purpose. Behind the scenes, however, the add\_rows method addresses the key issues which make the append method unsuitable for adding rows to an existing matrix object. Firstly, it validates that the incoming rows have the right number of entries for the matrix, and secondly, it updates the matrix’s \_\_rows property, which is used throughout the Matrix class. The add\_columns method addresses these same issues while also greatly simplifying the process of adding columns to an existing matrix.

The len(list) method is another method inherited from list which is used extensively in the Matrix class. The len(list) method is at the core of the matrix.get\_rows() and matrix.get\_columns() methods and is used throughout the Matrix class’s methods to get the lengths of individual rows and columns.

The bulk of the Matrix class’s internal computations are performed using nested for loops and if statements to iterate over rows and columns and access their entries. Recursion is used in the matrix.determinant() method in conjuction with for loops and if statements to automate the process of breaking the given matrix down into sub-matrices and then breaking these down into sub-matrices.

# Documentation

## 1. Instantiation and manipulation

**Creating a new matrix**

**def** \_\_init\_\_(self, \*rows):

The user creates a new matrix object by passing the rows as arguments to the initializer.

Ex: A = Matrix([1,2,3],[4,5,6],[7,8,9])

The matrix can have any number of rows and columns so long as each row has the same number of entries. If each row doesn’t have the same number of entries, Python will return an error.

**Adding rows to an existing matrix**

**def** add\_rows(self, \*rows):

The add\_rows method allows the user to add rows to an existing matrix object. It takes a variable number of rows as its arguments. The new rows must have the same number of entries as the existing rows or Python will return an error.

**Adding columns to an existing matrix**

**def** add\_columns(self, \*columns):

The add\_columns method allows the user to add columns to an existing matrix object. It takes a variable number of columns as its arguments. The new columns must have the same number of entries as the existing columns or Python will return an error.

**Accessing the entries of a matrix**

The user can access a matrix’s entries using standard index notation. The syntax is matrix[i][j], where i is the row index and j is the column index. Note that the indexing in Python, as with most other programming languages, is zero-based, meaning it starts at 0 instead of 1. This means that for a matrix A, the first entry of the first row is located at A[0][0].

## 2. Property methods

**def** get\_rows(self):

The get\_rows method returns the number of rows the matrix object currently has.

**def** get\_columns(self):

The get\_columns method returns the number of columns the matrix object currently has.

**def** is\_square(self):

The is\_square method returns True if the matrix is square. A matrix is called square if it has the same number of rows and columns.

**def** is\_invertible(self):

The is\_invertible method returns True if the matrix is invertible. An n-by-n square matrix A is invertible if there exists an n-by-n square matrix B such that AB = BA = In , where In denotes the n-by-n identity matrix.

**def** is\_symmetric(self):

The is\_invertible method returns True if the matrix is symmetric. A square matrix is called symmetric if it is equal to its transpose.

**def** is\_skew\_symmetric(self):

The is\_skew\_symmetric method returns True if the matrix is skew-symmetric. A square matrix is called skew-symmetric if it is equal to the negative of its transpose.

**def** is\_upper\_triangular(self):

The is\_upper\_triangular method returns True if the matrix is upper triangular. A square matrix is called upper triangular if all the entries below the main diagonal are zero.

**def** is\_lower\_triangular(self):

The is\_lower\_triangular method returns True if the matrix is lower triangular. A square matrix is called lower triangular if all the entries above the main diagonal are zero.

**def** is\_triangular(self):

The is\_triangular method returns True if the matrix is upper or lower triangular.

**def** determinant(self):

The determinant method returns the determinant of the matrix.

**def** transpose(self):

The transpose method returns the transpose of the matrix.

**def** is\_the\_same\_size\_as(self, other):

The is\_the\_same\_size\_as method takes another matrix as an argument and returns True if the matrices have the same number of rows and columns.

**def** can\_be\_multiplied\_by(self, other):

The can\_be\_multiplied\_by takes another matrix as an argument and returns True if the the matrix calling the method can be multiplied by the other matrix. A matrix A can be multiplied by a matrix B (A\*B) if B has the same number of rows as A has columns. Note that matrix multiplication is not commutative, ie A\*B != B\*A.

## 4. Implementation of special methods

**def** \_\_repr\_\_(self):

The \_\_repr\_\_ special method specifies how objects of a class should be represented. The \_\_repr\_\_ implementation for the matrix class dictates that matrix objects should be represented as rows of floating point values.

**def** \_\_add\_\_(self):

The \_\_add\_\_ special method specifies how the addition operation should be performed on objects of a class. The \_\_add\_\_ implementation for the matrix class dictates that matrix objects can only be added to other matrix objects, and that the matrices need to be the same size for addition to be valid.

**def** \_\_sub\_\_(self):

The \_\_sub\_\_ special method specifies how the subtraction operation should be performed on objects of a class. The \_\_sub\_\_ implementation for the matrix class dictates that matrix objects can only be subtracted from other matrix objects, and that the matrices need to be the same size for subtraction to be valid.

**def** \_\_mul\_\_(self):

The \_\_mul\_\_ special method specifies how the multiplication operation should be performed on objects of a class. The \_\_mul\_\_ implementation for the matrix class dictates that for matrix multiplication, the first matrix must have the same number of columns as the second matrix has rows, and that for scalar multiplication, each entry in the matrix should be multiplied by the scalar.