Segundo Proyecto de LP – Python Mágico

Equipo 2:

Alejandro Escobar Giraudy C312

Airelys Collazo Pérez C312

Henri Daniel Peña Dequero C311

Orientación:

En **Python** no existe el tipo predefinido array multidimensional.

1 - Implemente la clase **Matriz**, para representar matrices con las operaciones de suma y producto. Implemente además otras funcionalidades que crea necesarias.

2 - Implemente la indización para la clase **Matriz** de forma tal que se puedan hacer construcciones como las siguientes: **a = m[0, 6]** o **m[1, 2] = 9**.

3 - Implemente la indización para la clase **Matriz** por medio de acceso a campos de la forma: **a =m.\_0\_6** o **m.\_1\_2 = 9**.

4 - Los objetos matrices deberán ser iterables. El iterador de una matriz con **n** filas y **m** columnas debe devolver los elementos en el siguiente orden : **$m\_{1,1}**, **m\_{1,2}**, ..., **m\_{1,m}**,**m\_{2,1}**, ...,**m\_{n,m}$**

5 - Al tipo **Matriz** se podrá aplicar siempre el método **as\_type()** que devuelve una nueva **Matriz** con todos los tipos convertidos al tipo **type**. Suponga que existe un constructor en **type** que convierte de cualquier tipo a **type**.

Por ejemplo:

**m=Matriz(2,3)** # crea una matriz de **int** con valor 0s.

**mf = m.as\_float()** # **mf** es una matriz de 0s pero de tipo **float**.

Analizar: Resolución de miembros y métodos en **Python**. **Métodos mágicos**. **Iteradores**. El **builtineval**.

-------------------------------------------------------------------------------------------------------------------------------

Introducción

El objetivo de este proyecto en efecto es la realización de la clase **Matrix**, como variante a tener un **Array Bidimensional** en **Python**, e implementarle los métodos adecuados para trabajar como tal.

1-)

La redefinición de operaciones en **Python**, como es el caso de las operaciones de **suma** y **resta**, ocurre a partir de la utilización de unos métodos con características diferentes a los normales. Estos son los llamados **Métodos Mágicos**.

Los **Métodos Mágicos** constituyen el conjunto de Métodos que representan las acciones que se pueden realizar con una clase (dígase **suma**, **resta**, **multiplicación**, **indexación**, **iteración**, etc), son una lista finita de métodos cuya principal característica es que su nombre comienza y termina con guión bajo doble (**\_\_**), ejemplo de estos son **\_\_add\_\_** (para redefinir el operador suma), **\_\_mul\_\_** (para redefinir el operador multiplicación), **\_\_len\_\_** (para redefinir el resultado de aplicarle la función **len** a la clase), entre otros.

Estos reciben cantidad de argumentos variables dependiendo específicamente de la función, pero siempre reciben **self** como primer argumento, que representa la instancia del objeto al que le estamos redefiniendo la operación

En el caso de operadores binarios por ejemplo, reciben como argumento también la instancia que está del otro lado del operador, puedes recibir incluso más argumentos en el caso de por ejemplo, **\_\_setitem\_\_(self, pos, value)**, el cual permite asignar valores por indexación a la clase, **pos** representa la posición a indexar y **value** el valor que se va a agregar

Para redefinir los operadores de **suma** y **resta**, por ejemplo, podemos hacerlo de la siguiente forma:

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

**if self.rows != another.rows or self.columns != another.columns:**

**raise("Solo se pueden sumar matrices con iguales dimensiones")**

**sum = Matrix(self.rows, self.columns)**

**for i in range (sum.rows):**

**for i in range (sum.columns):**

**sum.values[i][j] = self.values[i][j] + another.values[i][j]**

**return sum**

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

**if self.columns != another.rows:**

**raise Exception("Deben coincidir la cantidad de columnas de la 1ra matriz con la cantidad de filas de la 2da")**

**mul = Matrix(self.rows, another.columns)**

**for i in range (mul.rows):**

**for j in range (mul.columns):**

**actual\_sum = 0**

**for k in range (self.columns):**

**actual\_sum = actual\_sum + (self.values[i][k] \* another.values[k][j])**

**return mul**

En el caso de nuestro proyecto no fueron redefinidos así exactamente, sino modificando líneas como **self.values[i][j]**, y en vez de eso tenemos **self[i,j]**, el cual será explicado posteriormente. Esto lo hicimos para respetar la privacidad de los atributos de la clase, y a su vez poder usar la indexación pedida en este proyecto.

Además de estos, fueron implementados en el proyecto:

**\_\_init\_\_(self)**, como constructor de la clase

**\_\_sub\_\_\_(self, another)**, para redefinir la resta de matrices

**\_\_str\_\_(self)**, para poder utilizar **print** en esta clase

2-)

Los métodos mágicos también permiten habilitar la indexación en una clase, tanto para acceder a un valor de la matriz como para asignarle, a partir de **[i,j,k…]**. Esto se logra mediante los métodos **\_\_getitem\_\_** y **\_\_setitem\_\_** respectivamente.

Una forma de implementarlo (la cual es justamente la forma utilizada en el proyecto) es:

**def \_\_getitem\_\_(self, pos):**

**if(pos[0] < 0 or pos[0] >= self.rows or pos[1] < 0 or pos[1] >= self.columns):**

**raise Exception("Indices fuera del rango de la Matriz")**

**return self.values[pos[0]][pos[1]]**

**def \_\_setitem\_\_(self, pos, new\_value):**

**if(pos[0] < 0 or pos[0] >= self.rows or pos[1] < 0 or pos[1] >= self.columns):**

**raise Exception("Indices fuera del rango de la Matriz")**

**self.values[pos[0]][pos[1]] = new\_value**

3-)

Queremos indexar la clase **Matriz** por medio de acceso a campos de la forma: **a =m.\_0\_6** o **m.\_1\_2 = 9**. Para lograr esto vamos a definirle una propiedad a cada elemento de la matriz, por ejemplo, el elemento que está en la posición **1,1** va a tener la propiedad con nombre **\_1\_1** de forma tal que podamos hacer **m.\_1\_1** y para asignar los valores a estas propiedades nos vamos a auxiliar de la función builtin **setattr (object, name, value)** que asigna valores a las propiedades.

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

**if rows < 0 or columns < 0:**

**raise Exception("Las dimensiones de una matriz deben ser positivas")**

**self.rows = rows**

**self.columns = columns**

**self.values = []**

**for i in range (rows):**

**actual\_row = []**

**for j in range (columns):**

**actual\_row.append(0)**

**setattr(type(self), f'\_{i}\_{j}', self.create\_property(i,j) )**

**self.values.append(actual\_row)**

Como se puede ver en el fragmento de código anterior, en nuestro método mágico **\_\_init\_\_** usamos la función **setattr** que esta se va a apoyar en el método **create\_property** paracrear las propiedades **\_{i}\_{j}** y asignarle sus valores.

El método **create\_property** quedaría**:**

**def create\_property( self, i, j):**

**def property\_setter( self, value):**

**self.values[i][j] = value**

**return property (lambda self: self.values[i][j], property\_setter)**

4-)

Creamos la clase **Iterator** para definir el iterador de la clase **Matriz**, de forma que esta pueda ser iterable. Esta clase será llamada desde el **\_\_iter\_\_** de la clase **Matriz.**

**def \_\_iter\_\_(self):**

**return Iterator(self)**

Primeramente, inicializamos nuestro **Iterator** con **\_\_init\_\_**, donde inicializamos el current en 0 e instanciamos la matriz.

**def \_\_init\_\_(self, matrix):**

**if not isinstance(matrix, Matrix):**

**raise Exception('Es un iterador de Matrix.')**

**self.matrix = matrix**

**self.current = 0**

Definimos el método **there\_is\_next** para saber si hay un próximo elemento a iterar.

**def there\_is\_next(self):**

**return self.current < self.matrix.rows \* self.matrix.columns**

Luego, redefinimos el **\_\_next\_\_** de forma que el iterador devuelva los elementos iterados de la forma **m\_{1,1}**, **m\_{1,2}**, ..., **m\_{1,m}**,**m\_{2,1}**, ...,**m\_{n,m}**

**def \_\_next\_\_(self):**

**if self.there\_is\_next():**

**i = self.current // self.matrix.columns**

**j = self.current % self.matrix.columns**

**value = self.matrix.values[i][j]**

**self.current += 1**

**return value**

**else:**

**self.current = 0**

**raise StopIteration('Terminamos de iterar')**

Una forma de iterar los elementos sería:

**iterator = iter(m)**

**while True:**

**try:**

**current = next(iterator)**

**print(current, end=' ')**

**except StopIteration:**

**print()**

**break**