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-)

Otros métodos mágicos importantes son **\_\_getattr\_\_** y **\_\_setattr\_\_** que se ejecutan al acceder a una propiedad inexistente y al modificar una propiedad inexistente, respectivamente.

Analicemos **\_\_getattr\_\_(self, item)** donde **self** representa la instancia de la clase e **item** el atributo. En nuestro caso la usaremos para poder indexar la clase **Matrix** de la forma **m.\_0\_0.** Como esta se llama cuando se accede a una propiedad que no existe lo que hacemos es matchear el atributo que sería de la forma **\_i\_j**, donde **\_i\_j** se refiere a la posición en la matriz, si este matchea retornamos el valor que se encuentra en dicha posición, si no matchea llamamos a **\_\_getattribute\_\_** que es un método que se llama cuando ya tenemos ese atributo.

**def \_\_getattr\_\_(self, item):**

**match = re.match( INDEX\_PATT, item)**

**if match:**

**i = int(match.groups()[0])**

**j = int(match.groups()[1])**

**if(i < 0 or i >= self.rows or j < 0 or j >= self.columns):**

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

**return self.values[i][j]**

**return self.\_\_getattribute\_\_(item)**

Nota: **INDEX\_PATT = r"^\_(\d+)\_(\d+)$"** que sería la expresión regular de la forma **\_i\_j**. La librería **re** es usada para matchear expresiones regulares, por eso cuando hacemos **re.match()** matcheamos la expresión mencionada anteriormente. Luego con el **.groups()** podemos acceder a los valores matcheados.

Veamos ahora **\_\_setattr\_\_(self, item, value)** donde **self** igualmente es la instancia de la clase**, item** el atributo y **value** el valor. Al establecer un valor para un atributo se llamará a esta función y cada método de establecer un valor ingresará a este. Este lo usaremos para el acceso a campos de la forma **m.\_0\_0 = 0.** Aquí usamos una idea semejante a la del método anterior si el atributo que sería de la forma **\_i\_j** matchea guardamos el valor en dicha posición, sino, con ayuda de **super()**,el cual nos permite acceder a métodos de la clase base (sería como decir object en CSharp), llamamos al **\_\_setattr\_\_** por defecto para que nos guarde el nuevo valor.

**def \_\_setattr\_\_(self, item, value):**

**match = re.match( INDEX\_PATT, item)**

**if match:**

**i = int(match.groups()[0])**

**j = int(match.groups()[1])**

**if(i < 0 or i >= self.rows or j < 0 or j >= self.columns):**

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

**self.values[i][j] = value**

**super().\_\_setattr\_\_(item,value)**

4-)

Un iterador es un objeto que permite recorrer uno a uno los elementos almacenados en una estructura de datos y opera con ellos. Cada secuencia (en este caso la clase **Matriz**) implementan el método mágico **\_\_iter\_\_** que devuelve un iterador capaz de recorrer la secuencia como corresponda.

En nuestro caso 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}.** En Python los iteradores tienen que implementar un método **next** que debe devolver los elemento de uno por vez, comenzando por el primero, y al llegar al final de la estructura debe levantar una excepción de tipo **StopIteration.**

**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**

Y otra sería:

**for i in m:**

**print(i, end=' ')**

**print()**

5- )

Ahora queremos que al aplicar una función de la forma **as\_type()**, nos devuelva una matriz que sea un clon donde cada uno de los elementos están convertido al tipo **type**.

Para resolver este problema volvemos a usar la función mágica **\_\_getattr\_\_** para capturar el método inexistente **as\_type()**.

Asimismo resolvimos el problema 3, esta vez tenemos un método de la forma **as\_type**, donde nos es de especial interés la palabra en **type**, usando expresiones regulares buscandomatchear todo método que inicie de la forma **as\_** y obteniendo el resto del nombre.

Para ello en el método **\_\_getattr\_\_** agregamos esta nueva forma de obtener métodos

**def \_\_getattr\_\_(self, item):**

**match = re.match( INDEX\_PATT, item)**

**… #Resto de la implementacion ya vista**

**match = re.match( AS\_TYPE\_PATT, item)**

**if match:**

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

**for i, item in enumerate(self):**

**ev = eval(match.groups()[0]+"("+str(item)+")")**

**temp[i // self.columns, i % self.columns] = ev**

**return lambda: temp**

**return self.\_\_getattribute\_\_(item)**

Nota: **AS\_TYPE\_PATT** **= r"^as\_(\S+)$"** que sería la expresión regular de la forma **as\_type** usando la librería re.

Nota 2: La implementación puesta en este informe está ligeramente cambiada a la entregada dado a que fue necesario usar la variable **ev** para no perder la **indentación**.

En este código usamos la función **buildin eval()** que nos permite tomar una cadena y ejecutarla como código, así ejecutamos **eval(“type(item)”)**, donde **item** es el elemento a convertir convertido a **str** primero, así convertimos cada elemento con la función **eval()**.

Acá podemos también usar una implementación con la función \_\_setattr\_\_ para convertir cada tipo de nuestra función en otro tipo en lugar de clonarla, sería implementar el mismo código en **\_\_setattr\_\_** y al finalizar entonces igualar **self** a **temp**, pero consideramos que esta fuera de los límites del trabajo aunque lo mencionamos aquí dado que nos parece interesante.

Para resolver las preguntas de este trabajo hemos utilizado los llamados métodos mágicos, iteradores y función **buildin eval()**, cada uno de ellos de gran importancia a la hora de crear nuestras clases y agregarle funcionalidades personalizadas para tu trabajo.

Gracias por leernos