



Seminario 15 de LP *"Python II"*

Yenli Gil Machado

Grupo C411

Y.GIL@ESTUDIANTES.MATCOM.UH.CU

Yunior Alexander Tejeda Illana

Grupo C412

Y.TEJEDA@ESTUDIANTES.MATCOM.UH.CU

Carlos Aryam Martínez Molina

Grupo C411

C.MOLINA@ESTUDIANTES.MATCOM.UH.CU

Juan José López Martínez

Grupo C311

J.LOPEZ2@ESTUDIANTES.MATCOM.UH.CU

Ariel Plasencia Díaz

Grupo C311

A.PLAENCIA@ESTUDIANTES.MATCOM.UH.CU

Índice

| | | |
|----------|--|----------|
| 1 | Introducción | 3 |
| 2 | Preliminares | 3 |
| 2.1 | Métodos mágicos | 3 |
| 2.2 | Iterador como patrón de diseño | 4 |
| 3 | Ejercicio | 6 |
| 3.1 | Problema | 6 |
| 3.2 | Solución | 6 |
| 3.2.1 | Inciso A | 6 |
| 3.2.2 | Inciso B | 7 |
| 3.2.3 | Inciso C | 8 |
| 3.3 | Código | 10 |

1

Introducción

El objetivo de este seminario es abordar el tema de la redefinición de operadores en el lenguaje de programación **python**, así como hacer objetos iterables. Para ello, nos basaremos en implementar la clase *Matrix*, en la cual le daremos respuesta a todos los ejercicios mandados y trataremos de explicar detalladamente nuestra solución.

2

Preliminares

2.1 Métodos mágicos

¿Qué son los **métodos mágicos**?

1. Son métodos especiales que tienen doble guión bajo al principio y al final de sus nombres.
2. Son también conocidos en inglés como *dunders* (de doble underscores).
3. Son utilizados para crear funcionalidades que no pueden ser representadas en un método regular.
4. Un uso común de estos es la sobrecarga de operadores. Esto significa definir operadores para clases personalizadas que permiten que operadores tales como $+$ o $*$ sean utilizados en ellas.
5. Sin duda corresponden a una forma de llevar a cabo el polimorfismo.

A continuación mostraremos algunos **métodos mágicos**, así como su operador y cómo usarlos:

| No. | Método | Operador | Expresión |
|-----|--|------------------------|-------------------|
| 1 | <code>__add__(self, other)</code> | $+$ (adición) | $x + y$ |
| 2 | <code>__sub__(self, other)</code> | $-$ (sustracción) | $x - y$ |
| 3 | <code>__mul__(self, other)</code> | $*$ (multiplicación) | $x * y$ |
| 4 | <code>__div__(self, other)</code> | $/$ (división) | x / y (python2) |
| 5 | <code>__truediv__(self, other)</code> | $/$ (división) | x / y (python3) |
| 6 | <code>__floordiv__(self, other)</code> | $//$ (división entera) | $x // y$ |
| 7 | <code>__mod__(self, other)</code> | $\%$ (módulo) | $x \% y$ |
| 8 | <code>__pow__(self, other)</code> | $**$ (potencia) | $x ** y$ |
| 9 | <code>__lshift__(self, other)</code> | \ll (left shift) | $x \ll y$ |
| 10 | <code>__rshift__(self, other)</code> | \gg (right shift) | $x \gg y$ |
| 11 | <code>__and__(self, other)</code> | $\&$ (and) | $x \& y$ |
| 12 | <code>__or__(self, other)</code> | $ $ (or) | $x y$ |
| 13 | <code>__xor__(self, other)</code> | \wedge (xor) | $x \wedge y$ |

Cuadro 1: Métodos mágicos para operaciones aritméticas

| No. | Método | Operador | Expresión |
|-----|----------------------------------|------------------------|------------|
| 1 | <code>__lt__(self, other)</code> | $<$ (menor estricto) | $x < y$ |
| 2 | <code>__le__(self, other)</code> | \leq (menor o igual) | $x \leq y$ |
| 3 | <code>__eq__(self, other)</code> | $==$ (igual) | $x == y$ |
| 4 | <code>__ne__(self, other)</code> | $!=$ (distinto) | $x != y$ |
| 5 | <code>__gt__(self, other)</code> | $>$ (mayor estricto) | $x > y$ |
| 6 | <code>__ge__(self, other)</code> | \geq (mayor o igual) | $x \geq y$ |

Cuadro 2: Métodos mágicos para comparaciones

| No. | Método | Operador | Expresión |
|-----|---------------------------------------|---|---------------------------------|
| 1 | <code>__len__(self)</code> | para <code>len()</code> | <code>len(obj)</code> |
| 2 | <code>__contains__(self, item)</code> | para verificar si se encuentra | <code>x in y</code> |
| 3 | <code>__str__(self)</code> | es una representación informal del objeto | <code>print(obj)</code> |
| 4 | <code>__iter__(self)</code> | para iteración sobre objetos | <code>it = iter(obj)</code> |
| 5 | <code>__reversed__(self)</code> | iterador reverso | <code>it = reversed(obj)</code> |
| 6 | <code>__next__(self)</code> | para pedir el siguiente en una iteración | <code>x = next(it)</code> |
| 7 | <code>__init__(self)</code> | se llama justo después de crear el objeto (constructor) | <code>x = A()</code> |
| 8 | <code>__new__(self)</code> | se llama antes de crear un objeto de la clase | <code>x = A()</code> |
| 9 | <code>__del__(self)</code> | se llama justo antes de destruir el objeto (destructor) | <code>del(obj)</code> |

Cuadro 3: Métodos mágicos para hacer que las clases actúen como contenedoras

| No. | Método | Operador | Expresión |
|-----|--|--|------------------------------------|
| 1 | <code>__getitem__(self, item)</code> | para indexar | <code>obj[item]</code> |
| 2 | <code>__setitem__(self, item, value)</code> | para asignar valores indexados | <code>obj[item] = value</code> |
| 3 | <code>__delitem__(self, item)</code> | para eliminar un atributo | <code>del(obj.item)</code> |
| 4 | <code>__getattr__(self, item)</code> | se ejecuta al acceder a un atributo inexistente | <code>obj.undefined</code> |
| 5 | <code>__setattr__(self, item, value)</code> | se ejecuta al modificar un atributo inexistente | <code>obj.undefined = value</code> |
| 6 | <code>__getattribute__(self, item)</code> | se ejecuta antes de acceder a cualquier atributo | <code>obj.attr</code> |
| 7 | <code>__setattribute__(self, item, value)</code> | se ejecuta antes de modificar cualquier atributo | <code>obj.item = value</code> |

Cuadro 4: Métodos mágicos relacionados con los atributos

A lo largo de este seminario estaremos tratando algunos de los **métodos mágicos** mostrados con anterioridad y dejaremos algunos ejemplos en la parte del código para un mejor y mayor entendimiento.

2.2 Iterador como patrón de diseño

En diseño de software, el patrón de diseño Iterador, define una interfaz que declara los métodos necesarios para acceder secuencialmente a un grupo de objetos de una colección. Es un mecanismo de acceso a los elementos que constituyen una estructura de datos para la utilización de estos sin exponer su estructura interna. Además, diferentes iteradores pueden presentar diferentes tipos de recorrido sobre la estructura (recorrido de principio a fin, recorrido con saltos, entre otros). Por otro lado los iteradores no tienen por qué limitarse a recorrer la estructura, sino que podrían incorporar otro tipo de lógica (por ejemplo, filtrado de elementos). Es más, dados diferentes tipos de estructuras, el patrón iterador permite recorrerlas todas utilizando una interfaz común uniforme.

Los iteradores en **python** son objetos de **python** que devuelven un elemento a la vez. Cada vez que solicita un iterador para el elemento siguiente llama a su método `__next__`. Si hay otro valor disponible, el iterador lo devuelve. De lo contrario, se genera una excepción *StopIteration*.

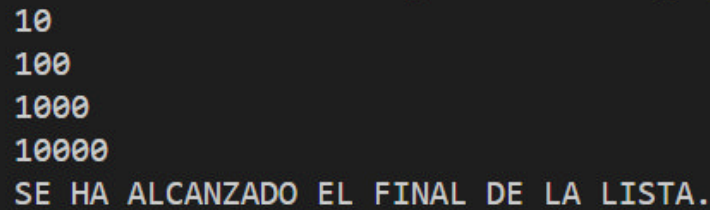
Este comportamiento, que solo devuelve el siguiente elemento cuando se le solicita, tiene dos ventajas principales:

1. Los iteradores en **python** necesitan menos espacio en la memoria. Recuerdan el último valor y una regla para llegar al siguiente valor en lugar de memorizar todos y cada uno de los elementos de una secuencia que potencialmente puede ser muy larga.
2. Los iteradores en **python** no controlan cuánto tiempo puede durar la secuencia que producen. Por ejemplo, no necesitan saber cuántas líneas tiene un archivo o cuántos archivos hay en una carpeta para iterar a través de ellas.

La función `iter()` se suele emplear para mostrar cómo funciona en realidad un bucle implementado con `for/in`. Antes del inicio del bucle la función `iter()` retorna el objeto iterable con el método subyacente `__iter__()`. Una vez iniciado el bucle, el método `__next__()` permite avanzar, en cada iteración, al siguiente elemento hasta alcanzar el último. Cuando el puntero se encuentra en el último elemento si se ejecuta

nuevamente el método `__next__()` el programa produce la excepción *StopIteration*. Veámos un ejemplo sencillo.

```
lista = [ 10, 100, 1000, 10000 ]
iterador = iter(lista)
try:
    while True:
        print(iterador.next())
except StopIteration:
    print("SE_HA_ALCANZADO_EL_FINAL_DE_LA_LISTA.")
```



```
10
100
1000
10000
SE HA ALCANZADO EL FINAL DE LA LISTA.
```

Figura 1: Salida del código anterior

En nuestra humilde opinión, no estamos de acuerdo con el mecanismo que existe en **python** para indicar el fin de iteración, que es lanzar una excepción *StopIteration*. Creemos que este mecanismo aunque sigue la filosofía del lenguaje, es decir, cumple con el *Zen de Python* escrito por su desarrollador Tim Peters no es la mejor manera de especificar el fin de una iteración. Entendemos que unas posibles soluciones al mecanismo anterior serían: tener un valor especial que indique el final de la colección, tener una función *end()* que indique si terminó o no la iteración o reutilizar la excepción *IndexError*.

Zen de Python:

1. Bello es mejor que feo.
2. Explícito es mejor que implícito.
3. Simple es mejor que complejo.
4. Complejo es mejor que complicado.
5. Plano es mejor que anidado.
6. Disperso es mejor que denso.
7. La legibilidad cuenta.
8. Los casos especiales no son tan especiales como para quebrantar las reglas.
9. Lo práctico gana a lo puro.
10. Los errores nunca deberían dejarse pasar silenciosamente.
11. A menos que hayan sido silenciados explícitamente.
12. Frente a la ambigüedad, rechaza la tentación de adivinar.
13. Debería haber una, y preferiblemente sólo una, manera obvia de hacerlo.
14. Aunque esa manera puede no ser obvia al principio a menos que usted sea holandés.
15. Ahora es mejor que nunca.

16. Aunque nunca es a menudo mejor que ya mismo.
17. Si la implementación es difícil de explicar, es una mala idea.
18. Si la implementación es fácil de explicar, puede que sea una buena idea.
19. Los espacios de nombres (namespaces) son una gran idea ¡Hagamos más de esas cosas!

3

Ejercicio

3.1 Problema

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

- a) Implemente la clase *Matrix*, para representar matrices de enteros, con las operaciones de suma, producto y producto por un escalar. Implemente además otras funcionalidades que crea necesarias.
- b) Implemente la indización para la clase *Matrix* de forma tal que se puedan hacer construcciones como las siguientes: $a = \text{Matrix}[0, 6]$ o $\text{Matrix}[1, 2] = 9$.
- c) 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: $m_{11}, m_{12}, \dots, m_{1m}, m_{21}, \dots, m_{nm}$.

3.2 Solución

En esta sección solo enseñaremos los métodos pedidos en el ejercicio anterior, pero cabe destacar que en la sección 3.3 mostraremos el código completo con la implementación de otras muchas funcionalidades implementadas por nuestro equipo.

3.2.1 INCISO A

A continuación solo mostraremos los métodos `__init__`, `__add__` y `__mul__`, redefiniendo así los operadores `+` y `*`, equivalentes a suma entre matrices y tanto producto entre matrices como el producto entre una matriz y un número entero respectivamente.

```
class Matrix:
    def __init__(self, rows, columns, value=0):
        if rows <= 0 or columns <= 0:
            raise Exception('LAS_DIMENSIONES_DEBEN_SER_POSITIVAS.')
        if not isinstance(value, int):
            raise Exception('LA_MATRIZ_DEBE_SER_DE_ENTEROS.')

        self.rows = rows
        self.columns = columns
        self.matrix = [[value for _ in range(self.columns)] for _ in range(self.rows)]

    def __add__(self, other):
        if not isinstance(other, Matrix):
            raise Exception('LAS_MATRICES_SE_SUMAN_ENTRE_ELLAS.')
        if self.rows != other.rows or self.columns != other.columns:
            raise Exception('LAS_MATRICES_DEBEN_TENER_DIMENSIONES_IGUALES.')

        matrix_add = Matrix(self.rows, self.columns)

        for i in range(self.rows):
            for j in range(self.columns):
                matrix_add.matrix[i][j] = self.matrix[i][j] + other.matrix[i][j]
        return matrix_add

    def __mul__(self, other):
        if not isinstance(other, Matrix) and not isinstance(other, int):
```

```

        raise Exception

    if isinstance(other, int):
        matrix_mul = Matrix(self.rows, self.columns)
        for i in range(self.rows):
            for j in range(self.columns):
                matrix_mul.matrix[i][j] = self.matrix[i][j] * other
        return matrix_mul
    else:
        if self.columns != other.rows:
            raise Exception('NO_SE_PUEDEN_MULTIPLICAR_LAS_MATRICES.')

        matrix_mul = Matrix(self.rows, other.columns)
        for i in range(self.rows):
            for j in range(other.columns):
                for k in range(self.columns):
                    matrix_mul.matrix[i][j] += self.matrix[i][k] * other.matrix[k][j]

        return matrix_mul

```

Los métodos anteriores no tienen nada de complicados. Sin más, pasaremos a probar algunos ejemplos. Cabe señalar que sobrescribimos el método `__str__` de la clase *Matrix* con el objetivo de imprimir las matrices con mayor facilidad.

```

# creamos una matriz de 2x3 con valor 1 en todas sus casillas
m1 = Matrix(2, 3, 1)
# creamos una matriz de 2x3 con valor 2 en todas sus casillas
m2 = Matrix(2, 3, 2)
# creamos una matriz de 3x4 con valor 5 en todas sus casillas
m3 = Matrix(3, 4, 5)

print(m1 + m2)
print()
print(m1 * m3)
print()
print(m3 * 2)

```

```

3 3 3
3 3 3

15 15 15 15
15 15 15 15

10 10 10 10
10 10 10 10
10 10 10 10

```

Figura 2: Salida del código anterior

3.2.2 INCISO B

A continuación solo mostraremos los métodos `__getitem__` y `__setitem__`, redefiniendo así los operadores para indexar, equivalentes a realizar construcciones de la forma $matrix[0, 1] = 5$.

```

class Matrix:
    def __getitem__(self, indexs):
        if not isinstance(indexs, tuple) or len(indexs) != 2:
            raise Exception('EL INDICE DEBE SER DE LA FORMA (fila, columna).')
        if indexs[0] >= self.rows or indexs[1] >= self.columns:
            raise Exception('INDICES FUERA DE RANGO.')

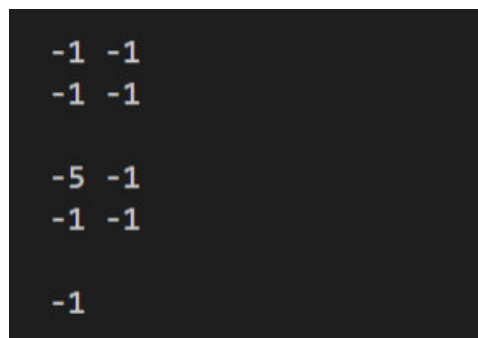
        i, j = indexs
        return self.matrix[i][j]

    def __setitem__(self, indexs, value):
        if not isinstance(indexs, tuple) or len(indexs) != 2:
            raise Exception('EL INDICE DEBE SER DE LA FORMA (fila, columna).')
        if not isinstance(value, int):
            raise Exception('LA MATRIZ DEBE SER DE ENTEROS.')
        if indexs[0] >= self.rows or indexs[1] >= self.columns:
            raise Exception('INDICES FUERA DE RANGO.')

        i, j = indexs
        self.matrix[i][j] = value

# creamos una matriz de 2x2 con valor -1 en todas sus casillas
m4 = Matrix(2, 2, -1)
# imprimimos la matriz m4 sobrescribiendo el método __str__
print(m4)
print()
# modificamos la casilla 0,0 al valor -5
m4[0,0] = -5
# imprimimos nuevamente m4 para chequear el cambio antes echo
print(m4)
print()
# imprimimos el valor de la casilla 1,1
print(m4[1,1])

```



```

-5 -1
-1 -1

-1

```

Figura 3: Salida del código anterior

3.2.3 INCISO C

A continuación mostraremos el mecanismo usado para lograr que la clase *Matrix* sea iterable. En este epígrafe solo enseñaremos la iteración a partir de la casilla 1, 1, pero en el código mostraremos también cómo iterar una matriz a partir de la última celda o casilla.

```

class Matrix:
    def __iter__(self):
        return MatrixIterator(self)

```



```

class MatrixIterator:
    def __init__(self, matrix):
        if not isinstance(matrix, Matrix):
            raise Exception('ES_UN_ITERADOR_DEL_OBJETO_Matrix.')

        self.matrix = matrix
        self.current = 0

    def move_next(self):
        return self.current < self.matrix.rows * self.matrix.columns

    def __iter__(self):
        return self

    def __next__(self):
        if self.move_next():
            new_row = self.current // self.matrix.columns
            new_column = self.current % self.matrix.columns
            value = self.matrix.matrix[new_row][new_column]
            self.current += 1
            return value
        else:
            self.current = 0
            raise StopIteration('NO_HAY_MAS_ELEMENTOS_A_ITERAR.')

```

Cabe destacar que hicimos una nueva clase para la implementación del iterador ya que constituye una buena práctica y para un mejor entendimiento y legibilidad por parte de nuestros lectores.

```

m5 = Matrix(2, 2)
# creamos la matriz [ [ 1 , 2 ] , [ 3 , 4 ] ]
m5[0,0], m5[0,1], m5[1,0], m5[1,1] = 1, 2, 3, 4
print(m5)
print()

# iteramos la matriz m5
iterator5 = iter(m5)
while True:
    try:
        current = next(iterator5)
        print(current, end=' ')
    except StopIteration:
        print()
        break

# otra forma de iterar
for i in m5:
    print(i, end=' ')

```

```
1 2
3 4

1 2 3 4
1 2 3 4
```

Figura 4: Salida del código anterior

3.3 Código

[Solución en Python](#)