# 1 Seminario 15 - Python II

En python no existe el tipo predefinido array multidimensional.

Definamos pues la clase Matrix para representar matrices de enteros.

## 1.1 Clases

En comparación con otros lenguajes de programación, el mecanismo de python para añadir clases requiere el mínimo de nueva sintaxis y semántica. Sus clases proveen las características principales de todo lenguaje de programación orientado a objetos: el mecanismo de herencia permite múltiples clases base, las clases derivadas pueden sobreescribir cualquier método de la(s) base(s) y un método puede llamar al método de la clase base con el mismo nombre. Las clases comparten la naturaleza dinámica de python: estas son creadas en tiempo de ejecución y pueden ser modificadas luego de su creación.

Como en C++, los miembros de una clase son públicos. No hay forma rápida de acceder a los miembros del objeto desde sus métodos: los métodos o funciones se declaran con un primer argumento explícito que representa al objeto en cuestión, el cual se provee implícitamente cuando se llama. Los **bult-int types** se pueden utilizar como clases bases para que el usuario extienda. También como en C++, la mayoría de los operadores con sintaxis especial( como las expresiones aritméticas, indización y otros) pueden redefinirse para instancias de clases.

La forma más simple de definir una clase sería:

```
class ClassName:
    <statement-1>
    .
    .
    .
    <statement-n>
```

En la práctica los **statements** dentro de la definición de una clase son usualmente definiciones de funciones, pero otro tipos de stamentes son posibles.

Para instanciar una clase podemos simplemente usar notación de función. En el caso anterior podemos asumir que `ClassName()` es una función que no recibe parámetros y devuelve una instancia de la clase CLassName, o sea, al hacer:

```
x = ClassName()
```

se crea una nueva instancia de la clase y se le asigna a la variable local x.

La operación de instanciación crea un objeto vacío, pero normalmente uno desea atribuir ciertas propiedades a las clases como estado inicial, con su creación. Para ello una clase puede definir un método especial llamado `__init__()` . Cuando este se ha definido, la inicialización de una clase lo invoca automáticamente para la clase recién creada. Si se proveen argumentos para la instanciación de la clase estos se pasan a `__init__()`.

Basado en lo explicado anteriormente podemos comenzar a definir nuestra clase Matriz:

```
class Matrix:
    def __init__(self, rows, columns):
        self.rows = rows
        self.columns = columns
        self._matrix = [[0 for j in range(columns)] for i in range(rows)]
```

Al hacer:

```
a = Matrix(2,3)
a_rows = a.rows
a_colums = a.colums
```

obtendremos una matriz con 2 filas y 3 columnas como dichas propiedades bien lo indican. Note que la propiedad *_matrix* lleva un underscore delante, esto es para representar que es de uso privado. Como en python no existen las variables privadas esta es la convención que se sigue para un mejor entendimiento del código.

Ahora, quisiéramos añadir ciertas funcionalidades a la clase Matriz, como la posibilidad de obtener la suma, el producto entre matrices y el producto escalar entre vectores. ¿Cómo logralo?

## 1.2 Sobrecarga de operadores

Sobrecargar de operadores u **Operator Overloading** implica extender el significado de un operador. Operadores tan simples como + tienen varios significados en dependencia de los objetos entre los que se aplique; por ejemplo, entre enteros se usa para añadirlos, entre strings para concatenarlos y entre listas para mezclarlas. Esto es posible porque el operador + está sobrecargado en las clases int y string. Precisamente a esta idea de que los mismos built-in operators muestren significados diferentes para objetos de distintas clases, es a lo que se le llama Operator Overloading.

Si intentamos utilizar el operador + entre dos objetos de nuestra clase Matriz obtendremos un error, puesto que el compilador no sabe como añadir estos dos objetos. ¿Cómo sobrecargar entonces operadores en python?

Al definir un método para un operador estamos haciendo sobrecarga de operadores. Podemos sobrecargar todos los operadores existentes, pero no es posible crear nuevos. Para efectuar estas sobrecargas Python provee algunes funciones **mágicas o especiales** que se invocan automáticamente cuando se ve involucrado el operador al que están asociadas. Por ejemplo, al usar el operador + , el método mágico `__add__()` es invocado; este define el fucionamiento de dicho operador.

Luego para modificar el comportamiento del + en nuestra clase Matriz sólo debemos redefinir el código de `__add__()`:

```
def Matrix:
    def __add__(self, other):
        assert isinstance(
            other, Matrix
        ), "Object to add can only be a number or a Matrix"
        assert (
            self.rows == other.rows and self.columns == other.columns
        ), "Matrix most have same number of rows and columns"

        sum_matrix = Matrix(self.rows, self.columns)
        for i in range(self.rows):
            for j in range(self.columns):
                sum_matrix[i, j] = self[i, j] + other[i, j]

        return sum_matrix
```

Para definir el producto entre nuestras matrices haríamos lo mismo con el método correspondiente a la multiplicación. En general para cualquiera de los operadores clásicos podemos efectuar el mismo proceso. A continuación se muetran los **métodos mágicos** de python para operator overloading:

#### 1.2.0.1 Operadores Binarios

| Operador | Método Mágico |
|---|---|
| + | `__add__(self,other)` |
| - | `__sub__(self,other)` |
| * | `__mul__(self,other)` |
| / | `__truediv__(self,other)` |
| // | `__floordiv__(self,other)` |
| % | `__mod__(self,other)` |
| ** | `__pow__(self,other)` |

#### 1.2.0.2 Operadores de Comparación

| Operador | Método Mágico |
|---|---|
| `<` | `__it__(self,other)` |
| `>` | `__gt__(self,other)` |
| `<=` | `__le__(self,other)` |
| `>=` | `__ge__(self,other)` |
| `==` | `__eq__(self,other)` |
| `!=` | `__ne__(self,other)` |

#### 1.2.0.3 Operadores de asignación

| Operador | Método Mágico |
|---|---|
| `+=` | `__iadd__(self,other)` |
| `-=` | `__isub__(self,other)` |
| `*=` | `__imul__(self,other)` |
| `/=` | `__idiv__(self,other)` |
| `//=` | `__ifloordiv__(self,other)` |
| `%=` | `__imod__(self,other)` |
| `**=` | `__ipow__(self,other)` |

#### 1.2.0.4 Operadores unarios

| Operador | Método Mágico |
|---|---|
| `-` | `__neg__(self,other)` |
| `+` | `__pos__(self,other)` |
| `~` | `__invert__(self,other)` |

Hasta aquí hemos obtenido el conocimiento necesario para implementar algunas de las funcionalidades básicas que deberían tener nuestras matrices, pero hay algo que se nos escapa. No tenemos ahora mismo la posibilidad de indexar en una matriz dada para conocer que contiene, ni somos capaces de modificar ningún valor en esta. Vamos a cambiar esto.

## 1.3 Indización

Resulta que para permitir construcciones como a = Matriz[0,6] o Matriz[1, 2] = 9 no necesitamos hacer nada muy diferente de lo visto hasta ahora, y es que la indización o subscripting se logra mediante sobrecargas de operadores como mismo la suma y la mutiplicación.

Para obtener el valor de Matriz en [0,6] hemos de definir el método `__getitem__()`. Cuando la función apropiada no está definida al intentar efectuar a = M[0,6] se obtiene una excepción del tipo AtrributeError o TypeError. Note que al ser M una instancia de la clase Matriz, hacer M[0,6] es equivalente a hacer type(M).___getitem___(M,(0,6)).

Para el caso de Matriz[1, 2] = 9, para poder modificar un valor de una matriz dada se debe de implementar el método `__setitem__()`.

El código propuesto sería el siguiente:

```
def __getitem__(self, coordinates):
        x, y = coordinates
        return self._matrix[x][y]

    def __setitem__(self, coordinates, key):
        x, y = coordinates
        self._matrix[x][y] = key
```

Resuelto entonces el tema de la indización. ¿Qué otras funcionalidades pudieran necesitar nuestras matices?

Los objetos matrices deberían ser iterables. El iterador de una matriz con `n` filas y `m` columnas debe devolverlos elementos en el siguiente orden:

$m_{1,1}, m_{1,2}, ..., m_{1,m}, m_{2,1}, ..., m_{n,m}$

¿Cómo logramos esto?

## 1.4 Iteradores y Generadores

### 1.4.1 Iteradores

La mayoría de los objetos que actúan como **contenedores** pueden recorrerse a partir de un simple for:

```
for element in [1,2,3]:
    print(element)
```

```
for element in (1,2,3):
    print(element)

for key in {'one':1,'two':2}:
    print(key)

for char in "123":
    print(char)

for line in open("file.txt"):
    print(line)
```

Este estilo de acceso es claro, conciso y conveniente. Pudiera decirse que el uso de iteradores unifica a python. Tras la escena, la sentencia for llama a *iter()* en el objeto contenedor. Dicha función retorna un objeto iterador, que define un método `__next__()` que retorna cada elemento del contenedor de uno en uno. Cuando no quedan más elementos `__next__()` lanza una excepción llamada *StopIteration()*, la cual le indica al ciclo for que debe terminar. Para llamar al método `__next__()` se puede usar función predefinida `next()` de la siguiente forma:

```
>>>name = "Gaby"
>>>iterator = iter(name)
>>>iterator
<iterator objext at 0x00A1DB50>
>>>next(iterator)
'G'
>>>next(iterator)
'a'
>>>next(iterator)
'b'
>>>next(iterator)
'y'
>>>next(iterator)
Traceback( most recent call last):
   File "<stdin>", line 1, in ?
   next(iterator)
   StopIteration
```

Visto este mecanismo tras el protocolo de iterador es fácil añadir este comportamiento a nuestras clases. Se debe definir un método `__iter__()` que retorne un objeto con un método `__next__()`. Si la clase define `__next__()` entonces `__iter__()` puede retornar

self:

```
class Matrix_Iterator:
    def __init__(self, matrix):
        self.matrix = matrix
        self.iter_index = -1

    def __iter__(self):
        return self

    def __next__(self):
        self.iter_index+=1
        if(self.iter_index >= self.matrix.columns* self.matrix.rows):
            self.iter_index = 0
            raise StopIteration

        return self._matrix[self.iter_index//self.matrix.columns]
                           [self.iter_index%self.matrix.columns]

class Matrix:
    <staments>
    def __iter__(self):
        return Matrix_Iterator(self)
```

### 1.4.2 Generadores

Los generadores son una simple y poderosa herramienta para crear iteradores. Se escriben como funciones regulares pero utilizan la sentencia `yield` cuando necesitan retornar datos. Cada vez que se llame `next()`, el generador resume desde donde se quedó, osea, recuerda todos las valores de la última ejecución.

Todo lo que se puede hacer con generadores se puede hacer con clases basadas en iteradores como las descritas anteriormente. Lo que sucede realmente es que con los generadores los métodos `__iter()` y `__next__()` son creados automáticamente.

Resulta importante como las varibles locales y el estado de ejecución se salvan automáticamente entre llamados, por lo cual no es necesario hacer uso de variables de instancia como `iter_index`.

Cuando el generador termina se lanza automáticamente la excepción *StopIteration*.

Usando generadores nuestro código para iterar por la clase Matriz quedaría más limpio y compacto:

```
def Matrix:
    <statements>
    def __iter__(self):
        for i in range(len(self._matrix)):
            for j in range(len(self._matrix[i])):
                yield self._matrix[i][j]
```

### 1.4.3 Expresiones generadoras

Algunos generadores simples pueden codificarse de forma más fácil con una sintaxis parecida a la de list comprehension pero usando paréntesis en lugar de corchetes. Estas expresiones se designan para situaciones donde el generadore se usa justo en el momento por una función que lo encierra. Las expresiones generadoras son más compactas pero menos versátiles que los otros generadores y tienden a ser más amigables con la memoria que sus equivalentes list comprehensions.

A continuación podemos ver un ejemplo de expresiones generadoras al definir el producto punto o producto escalar entre vectores:

```
def dot(self, other):
        return sum(x[0]*y[0] for x, y in zip(self._matrix,other._matrix))
```

Con estas implementaciones la clase Matriz está más que lista para su uso. En la carpeta *Code* que se adjunta se provee el código completo con la implementación y prueba de las funcionalidades mencionadas y otras que se consideraron necesarias, como la comparación de matrices y la impresión de estas en consola.

Cabe destacar que pueden existir algunos detalles que hubieran hecho el trabajo más fácil y no hayan sido implementados, por ejemplo inicializar una matriz con un array multidimensional con valores ya predefinidos. En este caso nos hubiera gustado hacer algo al estilo C#, donde es posible definir varios constuctores(funciones en general) con un mismo nombre y según los parámetros con que se llame al método se efectúa uno en específico, pero resulta que en python esto no es posible. Pueden definirse múltiples funciones con un mismo nombre, sí, pero el programa se quedaría con la última definida, pues cada una lo que está haciendo realmente es sobreescribir la anterior. Por supuesto que esto tiene varias alternativas fáciles, como puede ser simplemente comprobar al inicio de la función que se quería sobrecargar qué pedazo de código se debe ejecutar en función de los parámetros de entrada, y dividir dichas secciones en otras pequeñas funciones, pero bueno ya sin sobrecarga de funciones no suena interesante.