Creando una sistema de Algebra Lineal

En esta tarea seran guiados paso a paso en como realizar un sistema de arrays en Python para realizar operaciones de algebra lineal.

Pero antes... (FAQ)

Como se hace en la realidad? En la practica, se usan *paqueterias* funcionales ya probadas, en particular `numpy`, que contiene todas las herramientas necesarias para hacer computo numerico en `Python`.

Por que hacer esta tarea entonces? `Python` es un lenguage disenado para la programacion orientada a objetos. Al hacer la tarea desarrollaran experiencia en este tipo de programacion que les permitira crear objetos en el futuro cuando lo necesiten, y entender mejor como funciona `numpy` y en general, todas las herramientas de `Python`. Ademas, en esta tarea tambien aprenderan la forma de usar `numpy` simultaneamente.

Como comenzar con numpy? En la tarea necesitaremos importar la libreria `numpy`, que contiene funciones y clases que no son parte de `Python` basico. Recuerden que Python no es un lenguage de computo cientifico, sino de programacion de proposito general. No esta disenado para hacer algebra lineal, sin embargo, tiene librerias extensas y bien probadas que permiten lograrlo. `Anaconda` es una distribucion de `Python` que ademas de instalarlo incluye varias librerias de computo cientifico como `numpy`. Si instalaron `Python` por separado deberan tambien instalar `numpy` manualmente.

Antes de comenzar la tarea deberan poder correr:

In [1]:
```
import numpy as np
```

Lo que el codigo anterior hace es asociar al nombre `np` todas las herramientas de la libreria numpy. Ahora podremos llamar funciones de numpy como `np.<numpy_fun>`. El nombre `np` es opcional, pueden cambiarlo pero necesitaran ese nombre para acceder a las funciones de `numpy` como `<new_name>.<numpy_fun>`. Otra opcion es solo inlcuir `import numpy`, en cuya caso las funciones se llaman como `numpy.<numpy_fun>`. Para saber mas del sistema de modulos pueden revisar la liga https://docs.python.org/2/tutorial/modules.html ()

I. Creando una clase Array

`Python` incluye nativo el uso de listas (e.g. x = [1,2,3]). El problema es que las listas no son herramientas de computo numerico, `Python` ni siquiera entiende una suma de ellas. De hecho, la suma la entiende como concatenacion:

In [2]:
```
x = [1,2,3]
y = [4,5,6]
x + y
```

Out[2]:
```
[1, 2, 3, 4, 5, 6]
```

Vamos a construir una clase Array que incluye a las matrices y a los vectores. Desde el punto de vista computacional, un vector es una matriz de una columna. En clase vimos que conviene pensar a las matrices como transformacion de vectores, sin embargo, desde el punto de vista computacional, como la regla de suma

y multiplicacion es similar, conviene pensarlos ambos como *arrays*, que es el nombre tradicional en programacion

Computacionalmente, que es un array? Tecnicamente, es una lista de listas, todas del mismo tamano, cada uno representando una **fila** (fila o columna es optativo, haremos filas porque asi lo hace `numpy`, pero yo previero columnas). Por ejemplo, la lista de listas

```
[[1,2,3],[4,5,6]]
```

Corresponde a la matriz

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

The numpy way

In [341]:
```
B = np.array([[1,2,3], [4,5,6]]) # habiendo corrido import numpy as np
```

Es posible sumar matrices y multiplicarlas por escalares

In [342]:
```
B + 2*B # Python sabe sumar y multiplicar arrays como algebra lineal
```

Out[342]:
```
array([[ 3,  6,  9],
       [12, 15, 18]])
```

Las matrices de numpy se pueden multiplicar con la funcion `matmul` dentro de `numpy`

In [388]:
```
np.matmul(B.transpose(), B) # B^t*B
```

Out[388]:
```
array([[17, 22, 27],
       [22, 29, 36],
       [27, 36, 45]])
```

Los arrays the numpy pueden accesarse con indices y slices

Una entrada especifica:

In [169]:
```
B[1,1]
```

Out[169]:
```
5
```

Una fila entera:

In [170]:
```
B[1,:]
```

Out[170]:
```
array([3, 6])
```

Una columna entera:

In [174]:
```
B[:,2]
```

Out[174]:
```
array([3, 6])
```

Un subbloque (notar que un slice `n:m` es `n,n+1,...,m-1`

In [175]:
```
B[0:2,0:2]
```

Out[175]:
```
array([[1, 2],
       [4, 5]])
```

En numpy podemos saber la dimension de un array con el campo `shape` de numpy

In [183]:
```
B.shape
```

Out[183]:
```
(2, 3)
```

Numpy es listo manejando listas simples como vectores

In [270]:
```
vec = np.array([1,2,3])
print(vec)
```

```
[1 2 3]
```

Comenzando desde cero...

In [1]:
```
class Array:
    "Una clase minima para algebra lineal"
    def __init__(self, list_of_rows):
        "Constructor"
        self.data = list_of_rows
        self.shape = (len(list_of_rows), len(list_of_rows[0]))
```

In [2]:
```
A = Array([[1,2,3], [4,5,6]])
A.__dict__ # el campo escondido __dict__ permite acceder a las propiedades de clase de un o
```

Out[2]:
```
{'data': [[1, 2, 3], [4, 5, 6]], 'shape': (2, 3)}
```

In [3]:
```
A.data
```

Out[3]:
```
[[1, 2, 3], [4, 5, 6]]
```

In [4]:
```
A.shape
```

Out[4]:
```
(2, 3)
```

El campo `data` de un Array almacena la lista de listas del array. Necesitamos implementar algunos metodos

para que sea funcional como una clase de algebra lineal.

1. **Un metodo para imprimir una matriz de forma mas agradable**
2. **Validador**. Un metodo para validar que la lista de listas sea valida (columnas del mismo tamano y que las listas interiores sean numericas
3. **Indexing** Hacer sentido a expresiones A[i,j]
4. **Iniciar matriz vacia de ceros** Este metodos es muy util para preacolar espacio para guardar nuevas matrices
5. **Transposicion** B.transpose()
6. **Suma** A + B
7. **Multiplicacion escalar y matricial** 2 *A y A*B
8. **Vectores** (Opcional)

Con esto seria posible hacer algebra lineal

Metodos especiales de clase...

Para hacer esto es posible usar metodos especiales de clase `__getitem`, `__setitem__`, `__add__`, `__mul__`, `__str__`. Teoricamente es posible hacer todo sin estos metodos especiales, pero, por ejemplo, es mucho mas agradable escribir `A[i,j]` que `A.get(i,j)` o `A.setitem(i,j,newval)` que `A[i,j] = newval`.

1. Un metodo para imprimir mejor...

Necesitamos agregar un metodo de impresion. Noten que un array de numpy se imprime bonito comparado con el nuestro

In [210]:
```
Array([[1,2,3], [4,5,6]])
```

Out[210]:
```
<__main__.Array at 0x1cc60226080>
```

In [211]:
```
print(Array([[1,2,3], [4,5,6]]))
```

```
<__main__.Array object at 0x000001CC602450F0>
```

In [212]:
```
np.array([[1,2,3], [4,5,6]])
```

Out[212]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

In [209]:
```
print(np.array([[1,2,3], [4,5,6]]))
```

```
[[1 2 3]
 [4 5 6]]
```

Por que estas diferencias? Python secretamente busca un metodo llamado `__repr__` cuando un objeto es llamado sin imprimir explicitamente, y `__str__` cuando se imprime con `print` explicitamente. Por ejemplo:

In [227]:
```
class TestClass:
    def __init__(self):
        pass # this means do nothing in Python
    def say_hi(self):
        print("Hey, I am just a normal method saying hi!")
    def __repr__(self):
        return "I am the special class method REPRESENTING a TestClass without printing"
    def __str__(self):
        return "I am the special class method for explicitly PRINTING a TestClass object"
```

In [228]:
```
x = TestClass()
```

In [229]:
```
x.say_hi()
```

```
Hey, I am just a normal method saying hi!
```

In [230]:
```
x
```

Out[230]:
```
I am the special class method REPRESENTING a TestClass without printing
```

In [231]:
```
print(x)
```

```
I am the special class method for explicitly PRINTING a TestClass object
```

EJERCICIO 1

- Escribe los metodos `__repr__` y `__str__` para la clase Array de forma que se imprima legiblemente como en numpy arrays.

2. Un validador

Ok esta es la parte aburrida... Los voy a ayudar un poco. Pero no deberia ser cualquier cosa un array, pues todas las filas deberian ser del mismo tamano no? Ademas, las filas deben tener entradas solo numericas... Por ultimo, cuando en vez de listas de listas solo hay una lista, conviene pensarlo como un vector (una columna... por eso yo prefiero llenar por columnas, pero esa es la forma python...).

Todo esto se puede lograr mejorando un poco el validador.

Una nota al margen Para hacer el validador voy a usar una de las herramientas mas poderosas de Python, listas de comprehension. Son formas faciles y rapidas de iterar en un solo comando. Un ejemplo:

In [238]:
```
some_list = [1,2,3, 4, 5, 6]
[i**2 for i in some_list] # Elevar al cuadrado con listas de comprehension
```

Out[238]:
```
[1, 4, 9, 16, 25, 36]
```

En el validador uso la lista de comprehension para verificar que todas las filas tengo el mismo tamano. Tambien creo un error si no se cumple con el comando `raise`. Este es un uso avanzado, entonces si no tienen

experiencia en Python no se preocupen mucho por los detalles.

Checar que todas las entradas son numericas lo dejamos como un ejericio optativo... no vale la pena deternos mucho en eso...

In [266]:
```
class Array:
    "Una clase minima para algebra lineal"
    def __init__(self, list_of_rows):
        "Constructor y validador"
        # obtener dimensiones
        self.data = list_of_rows
        nrow = len(list_of_rows)
        #  ___caso vector: redimensionar correctamente
        if not isinstance(list_of_rows[0], list):
            nrow = 1
            self.data = [[x] for x in list_of_rows]
        # ahora las columnas deben estar bien aunque sea un vector
        ncol = len(self.data[0])
        self.shape = (nrow, ncol)
        # validar tamano correcto de filas
        if any([len(r) != ncol for r in self.data]):
            raise Exception("Las filas deben ser del mismo tamano")

    def __repr__(self):
        "Ejercicio"
        pass

    def __str__(self):
        "Ejercicio"
        pass
```

In [267]:
```
Array([[1,2,3], [4,5]])
```

```
---------------------------------------------------------------------------
Exception                                 Traceback (most recent call last)
<ipython-input-267-73c5131e7636> in <module>()
----> 1 Array([[1,2,3], [4,5]])

<ipython-input-266-ede091f3d719> in __init__(self, list_of_rows)
     15         # validar tamano correcto de filas
     16         if any([len(r) != ncol for r in self.data]):
---> 17             raise Exception("Las filas deben ser del mismo tamano")
     18
     19     def __repr__(self):

Exception: Las filas deben ser del mismo tamano
```

In [269]:
```
vec = Array([1,2,3])
vec.data
```

Out[269]:
```
[[1], [2], [3]]
```

3. Indexing and Item assignment

Tomaria varias lineas de codigo hacer un indexing/slicing tan complejo como el de numpy, pero podemos dar una version sencilla....

Queremos que las siguientes expresiones tengan sentido

```
A = Array([[1,2],[3,4]])
A[0,0]
A[0,0] = 8
`
```

Por el momento obtenemos errores

In [253]:
```
A = Array([[1,2], [3,4]])
A[0,0]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-253-59057898ec49> in <module>()
      1 A = Array([[1,2], [3,4]])
----> 2 A[0,0]

TypeError: 'Array' object is not subscriptable
```

In [254]:
```
A[0,0] = 8
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-254-9b1e9078e887> in <module>()
----> 1 A[0,0] = 8

TypeError: 'Array' object does not support item assignment
```

Para poder acceder a un index un metodo __getitem__.

In [391]:
```
class Array:
    "Una clase minima para algebra lineal"
    def __init__(self, list_of_rows):
        "Constructor y validador"
        # obtener dimensiones
        self.data = list_of_rows
        nrow = len(list_of_rows)
        #  ___caso vector: redimensionar correctamente
        if not isinstance(list_of_rows[0], list):
            nrow = 1
            self.data = [[x] for x in list_of_rows]
        # ahora las columnas deben estar bien aunque sea un vector
        ncol = len(self.data[0])
        self.shape = (nrow, ncol)
        # validar tamano correcto de filas
        if any([len(r) != ncol for r in self.data]):
            raise Exception("Las filas deben ser del mismo tamano")

    def __getitem__(self, idx):
        return self.data[idx[0]][idx[1]]
```

In [392]:
```
A = Array([[1,2],[3,4]])
A[0,1]
```

Out[392]: 2

EJERCICIO 2

- Escribe el metodo `__setitem__` para que el codigo `A[i,j] = new_value` cambie el valor de la entrada (i,j) del array. El esqueleto de la funcion es

```
class Array:
  #
  #
  def __setitem__(self, idx, new_value):
      "Ejercicio"
  #
```

4. Iniciar una matriz en ceros

Este no es un metodo de la clase, mas bien conviene que sea una funcion que crea un objeto de la clase en ceros. Vamos! Es el ejercicio mas facil, se los dejo a ustedes. Queremos que el resultado sea similar a la siguiente la funcion `zeros` de `numpy`

In [295]:
```
np.zeros((3,6))
```

Out[295]:
```
array([[ 0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.]])
```

EJERCICIO 3

1. Implementa una funcion de zeros para crear arrays "vacios"

    ```
    def zeros(shape):
     "Implementame por favor"
    ```

    Hint, encontraras utiles las listas de comprehension, por ejemplo el codigo `[0. for x in range(5)]` crea una lista de 5 ceros.
2. Implementa una funcion `eye(n)` que crea la matriz identidad de $n \times n$ (es decir, la matriz que tiene puros ceros y unos en la diagonal). El nombre `eye` es tradicional en software de algebra lineal, aunque no es muy intuitivo.

5. Transposicion

Otro ejercicio muy facil! Debe funcionar igual que numpy!

In [299]:
```
np.array([[1,2], [3,4]]).transpose()
```

Out[299]:
```
array([[1, 3],
       [2, 4]])
```

EJERCICIO 4

- Implementa la funcion de transposicion

    ```
    class Array:
      ###
      ###
      ###
      def transpose(self):
          "Implementame :)"
          ###
    ```

6. Suma

Ok aqui llegamos al primer punto dificil! Pongan mucha atencion porque ustedes van a hacer la multiplicacion!!!!

Muchas clases tienen una nocion de suma, como vimos la suma en listas y en strings de Python es concatenacion, pero la suma de Arrays debe ser suma entrada por entrada, como en los arrays de numpy. **Como logramos eso? Definiendo metodos para `__add__` que reciben como argumentos `self` y `other`**

In [300]:
```
"hola " + "tu"
```

Out[300]:
```
'hola tu'
```

In [301]:
```
[1,2,3] + [2,3,4]
```

Out[301]:
```
[1, 2, 3, 2, 3, 4]
```

In [302]:
```
np.array([1,2,3]) + np.array([2,3,4])
```

Out[302]:
```
array([3, 5, 7])
```

In [327]:
```
np.array([1,2,3]) + 10 # Broadcasted sum, es muy util
```

Out[327]:
```
array([11, 12, 13])
```

In [394]:
```
class Array:
    "Una clase minima para algebra lineal"
    def __init__(self, list_of_rows):
        "Constructor y validador"
        # obtener dimensiones
        self.data = list_of_rows
        nrow = len(list_of_rows)
        #  ___caso vector: redimensionar correctamente
        if not isinstance(list_of_rows[0], list):
            nrow = 1
            self.data = [[x] for x in list_of_rows]
        # ahora las columnas deben estar bien aunque sea un vector
        ncol = len(self.data[0])
        self.shape = (nrow, ncol)
        # validar tamano correcto de filas
        if any([len(r) != ncol for r in self.data]):
            raise Exception("Las filas deben ser del mismo tamano")

    def __add__(self, other):
        "Hora de sumar"
        if isinstance(other, Array):
            if self.shape != other.shape:
                raise Exception("Las dimensiones son distintas!")
            rows, cols = self.shape
            newArray = Array([[0. for c in range(cols)] for r in range(rows)])
            for r in range(rows):
                for c in range(cols):
                    newArray.data[r][c] = self.data[r][c] + other.data[r][c]
            return newArray
        elif isinstance(2, (int, float, complex)): # en caso de que el lado derecho sea so
            rows, cols = self.shape
            newArray = Array([[0. for c in range(cols)] for r in range(rows)])
            for r in range(rows):
                for c in range(cols):
                    newArray.data[r][c] = self.data[r][c] + other
            return newArray
        else:
            return NotImplemented # es un tipo de error particular usado en estos metodos
```

In [395]:
```
A = Array([[1,2], [3,4]])
B = Array([[5,6], [7,8]])
C = A + B
C.data
```

Out[395]:
```
[[6, 8], [10, 12]]
```

In [396]:
```
D = A + 10
```

In [397]:
```
D.data
```

Out[397]:
```
[[11, 12], [13, 14]]
```

HONESTAMENTE NO PODRIA SER MAS COOL!

Ahora veamos el error si las dimensiones no cuadran.

In [398]:
```
Array([[1,2], [3,4]]) + Array([[5,6, 5], [7,8,3]])
```

```
---------------------------------------------------------------------------
Exception                                 Traceback (most recent call last)
<ipython-input-398-0d3ffb398135> in <module>()
----> 1 Array([[1,2], [3,4]]) + Array([[5,6, 5], [7,8,3]])

<ipython-input-394-293731a066fb> in __add__(self, other)
     21         if isinstance(other, Array):
     22             if self.shape != other.shape:
---> 23                 raise Exception("Las dimensiones son distintas!")
     24             rows, cols = self.shape
     25             newArray = Array([[0. for c in range(cols)] for r in range(rows)])

Exception: Las dimensiones son distintas!
```

EJERCICIO 5

1. (dificil) En nuestra clase Array la expresion `A + 1` tiene sentido para un Array `A`, sin embargo la expresion inversa falla, por ejemplo

   ```
   1 + Array([[1,2], [3,4]])
   ```

   entrega un error. Investiga como implementar el metodo de clase `__radd__` para resolver este problema.
2. Nuestro metodo de suma no sabe restar, implementa el metodo de clase `__sub__` similar a la suma para poder calcular expresiones como A - B para A y B arrays o numeros.

7. Multiplicacion Matricial

Ahora si su ejercicio mas dificil y mas importante. Queremos que `A * B` sea multiplicacion matricial y `a*A` ser multiplicacion escalar cuando a es un numero.

NOTA IMPORTANTE!!!! Esta es la primera cosa que no esta implementada tal cual como numpy, pero es mucho mas padre hacerlo asi para algebra lineal. Numpy, al igual que las matrices de R, asumen que la multiplicacion debe ser entrada por entrada como += la suma. Otros lenguages como Julia y Matlab disenados para computo cientifico asumen que la multiplicacion de matrices es la default.

EJERCICIO 6

Implementa las funciones `__mul__` y `__rmul__` para hacer multiplicacion matricial (y por un escalar). **Hint**. Entiende y modifica el codigo de suma del punto anterior

8. Vectores (Herencia!!!)

La clase de Arrays ya hace todo lo necesario para hacer algebra lineal, pero es un poco incomoda para trabajar con vectores. Quisieramos tener una clase Vector que heredara el comportamiento de los Arrays pero con facilidades adicionales para trabajar con vectores. Este es el concepto de "herencia" en programacion orientada a objetos. Lo poderoso de la herencia es que permite trabajar con objetos mas especializados, que pueden tener campos o metodos adicionales a los de su clase padre, pero heredan el comportamiento. Tambien puede hacer un override de metodos de los padres.

In [400]:
```
class Vector(Array): # declara que Vector es un tipo de Array
    def __init__(self, list_of_numbers):
        self.vdata = list_of_numbers
        list_of_rows = [[x] for x in list_of_numbers]
        return Array.__init__(self, list_of_rows)
    def __repr__(self):
        return "Vector(" + str(self.vdata) + ")"
    def __str__(self):
        return str(self.vdata)
    def __add__(self, other):
        new_arr = Array.__add__(self, other)
        return Vector([x[0] for x in new_arr.data])
```

In [401]:
```
Vector([1,2,3]).__dict__
```

Out[401]:
```
{'data': [[1], [2], [3]], 'shape': (3, 1), 'vdata': [1, 2, 3]}
```

In [402]:
```
Vector([1,2,3])
```

Out[402]:
```
Vector([1, 2, 3])
```

In [404]:
```
Vector([1,2,3]) + Vector([5,-2,0])
```

Out[404]:
```
Vector([6, 0, 3])
```

In [405]:
```
Vector([1,2,3]) + 10
```

Out[405]:
```
Vector([11, 12, 13])
```

Yo recomiendo no usar durante el resto de este ejercicio para evitar estar implementando metodos adicionales (por ejemplo, multiplicacion Matriz Vector debe regresar Vector). Si lo hacen y funciona en el resto del codigo consideren muchos puntos adicionales.

II. El Gran Final

La prueba de oro para ver si su sistema de Algebra Lineal sirve es resolver sistemas de ecuaciones lineales.

EJERCICIO 7-10

- Implementa una funcion `forward_subs` que resuelva sistemas de ecuaciones de la forma $Lx = y$ con L triangular inferior y y cualquier Vector o Array de una columna. Detalles en link (https://en.wikipedia.org/wiki/Triangular_matrix#Forward_and_back_substitution)
- Implementa una funcion `backward_subs` que resuelva sistemas $Ux = y$ con U triangular superior y y Vector o Array de una columna. Detalles en link (https://en.wikipedia.org/wiki/Triangular_matrix#Forward_and_back_substitution)
- Implementa una funcion `LU` que reciba un Array A y devuelva 3 arrays L,U y P tales que $PA = LU$ con L trangular inferior, U triangular superior y P matriz de permutacion. Mas information link (https://en.wikipedia.org/wiki/LU_decomposition)
- Implementa una funcion `lu_linsolve` que resuelva cualquier sistema de ecuaciones `Ax = y` con `A` un Array y y un Vector o Array de una columna. La solución es muy sencilla usando la descomposición LU de los puntos anteriores link (https://en.wikipedia.org/wiki/LU_decomposition#Solving_linear_equations)

In [ ]: