Librería en Python para Matemáticas II (Álgebra Lineal)

https://github.com/mbujosab/LibreriaDePythonParaMates2

Marcos Bujosa

August 19, 2019

El objetivo es mostrar que escribir matemáticas y usar un lenguaje de programación son prácticamente la misma cosa. Este modo de proceder debería ser un ejercicio muy didáctico ya que:

> *Un PC es muy torpe y se limita a ejecutar literalmente lo que se le indica (un PC no interpreta interpolando para intentar dar sentido a lo que se le dice... eso lo hacemos las personas, pero no los ordenadores).*
>
> *Por tanto, este ejercicio nos impone una disciplina a la que en general no estamos acostumbrados:* el ordenador hará lo que queremos solo si las expresiones tienen sentido e indican correctamente lo que queremos. *Si el ordenador no hace lo que queremos, será porque que hemos escrito las ordenes de manera incorrecta (lo que supone que también hemos escrito incorrectamente las expresiones matemáticas).*

Con esta idea en mente:

1. La notación de las notas de clase pretende ser operativa, en el sentido de que su uso se pueda traducir en operaciones que debe realizar el ordenador.
2. Muchas demostraciones son algorítmicas (al menos las que tienen que ver con el método de Gauss), de manera que dichas demostraciones describen literalmente la programación en Python de los correspondientes algoritmos.

Aunque Python tiene librerías que permiten operar con vectores y matrices, aquí crearemos nuestra propia librería. Así lograremos que la notación empleada en las notas de clase y las expresiones en Python se parezcan lo más posible. Además, para hacer más evidente el paralelismo entre las definiciones de las notas de clase y el código de nuestra librería, las partes del código menos importantes se relegan al final[1] (véase la nota sobre *literate programming* más abajo). Destacar lo fundamental del código facilita apreciar que las definiciones de las notas de clase son implementadas de manera literal en nuestra librería de Python. Recuerde que **¡hacer matemáticas y programar son prácticamente la misma cosa!**).

Advertencia

El conjunto de símbolos disponibles para definir operadores en Python es muy limitado. Esto nos obliga a usar algunos símbolos que difieren de los usados en las notas de clase (por ejemplo, Python no dispone del símbolo "$^{\intercal}$", por ello, para denotar la transposición nos veremos forzados a usar el operador `__invert__`, es decir, la tilde "`~`" de Python, que además deberemos colocar a la izquierda de la matriz que transponemos).

Mates II	Python
$\mathbf{A}^{\intercal}$	`~A`

Afortunadamente otros símbolos si coincidirán con los usados en las notas. Por ejemplo, en Python disponemos de la barra "|" (operador `__or__`), que usaremos para la selección de componentes tanto por la derecha como por la izquierda.

Mates II	Python
$\boldsymbol{v}_{\vert i}$	`v|i`
${}_{i\vert}\boldsymbol{v}$	`i|v`
$\mathbf{A}_{\vert j}$	`A|j`
${}_{i\vert}\mathbf{A}$	`i|A`

[1] aquellas que tienen que ver con textos de ayuda, con la comprobación de que los inputs de las funciones son adecuados, con otras formas alternativas de instanciar clases, con la representación de objetos en Jupyter usándo código LaTeX, etc.

Por otra parte, algunos símbolos son necesarios en Python, aunque no se escriban explícitamente en las notas de clase. Por ejemplo, en las notas expresamos la aplicación de una operación elemental sobre las filas (columnas) de una matriz con subíndices a izquierda (derecha). Python no sabe interpretar este modo de escribir, necesita un símbolo (`&`) para indicar que la transformación elemental actua sobre la matriz.

Mates II	Python	Mates II	Python
$\mathbf{A}\,{}_{\tau\ [i \rightleftharpoons j]}$	`A & T({i,j})`	${}_{\tau\ [i \rightleftharpoons j]}\,\mathbf{A}$	`T({i,j}) & A`
$\mathbf{A}\,{}_{\tau\ [a\cdot i]}$	`A & T((i,a))`	${}_{\tau\ [a\cdot i]}\,\mathbf{A}$	`T((i,a)) & A`
$\mathbf{A}\,{}_{\tau\ [i+a\cdot j]}$	`A & T((i,j,a))`	${}_{\tau\ [i+a\cdot j]}\,\mathbf{A}$	`T((i,j,a)) & A`

Tutorial previo en un notebook

Antes de seguir, mírese el Notebook referente a **"Listas y tuplas" en Python**
O acceda a la url: https://mybinder.org/v2/gh/mbujosab/LibreriaDePythonParaMates2/master

Literate progamming con noweb

Este documento está escrito usando noweb. Es un entorno que permite escribir a la vez tanto código como la documentación del mismo.

El código se escribe a trozos o "chunks" como por ejemplo este:

2a ⟨*Chunk de ejemplo que define la lista* `a` 2a⟩≡
```
a = ["Matemáticas II es mi asignatura preferida", "Cómo mola el Python", 1492, "Noweb"]
```
This code is used in chunk 2c.

y este otro chunk:

2b ⟨*Segundo chunk de ejemplo que cambia el último elemento de la lista* `a` 2b⟩≡
```
a[-1] = 10
```
This code is used in chunk 2c.

Cada chunk tiene un nombre (que yo uso para describir lo que hace el código dentro del chunk). Lo maravilloso es que dentro de un chunk se pueden insertar otros chunks. Así, podemos programar el siguiente código que enumera los elementos de una tupla y después hace unas sumas:

2c ⟨*EjemploLiterateProgramming.py* 2c⟩≡
```
⟨Chunk de ejemplo que define la lista a 2a⟩
⟨Segundo chunk de ejemplo que cambia el último elemento de la lista a 2b⟩

for indice, item in enumerate(a, 1):
    print (indice, item)

⟨Chunk final que indica qué tipo de objeto es a y hace unas sumas 36e⟩
```
Root chunk (not used in this document).

Esto permite destacar unas partes del código y pasar por alto otras. Por ejemplo, *del chunk del recuadro de arriba me interesa que se vea el código del bucle que permite enumerar los elementos de una lista*. Lo demás es accesorio y se puede consultar en los correspondientes chunks. Fíjese que el número al final del nombre de cada chunk corresponde a la página donde se puede consultar su codigo. Por ejemplo, el último chunk de este ejemplo se encuentra en la Página 36 de este documento (y justo detrás podrá ver cómo queda el código completo).

Con ánimo de que esta documentación sea más didáctica, relegaré trozos de código (*chunks*) para poder destacar cómo esta librería de Python es una implementación literal de las definiciones dadas en mis notas de clase de Mates II.

Part I
Código principal

1 Vectores y matrices

1.1 La clase `Vector`

En las notas de clase se dice que

Un *vector* de $\mathbb{R}^n$ es un "sistema" de n números reales.

Y los representaremos con paréntesis, bien en forma de fila

$$\boldsymbol{v} = (v_1, \ldots, v_n)$$

o bien en forma de columna

$$\boldsymbol{v} = \begin{pmatrix} v_1 \\ \vdots \\ v_n \end{pmatrix}$$

Python no posee objetos que sean "vectores". Necesitamos crear dichos objetos definiendo una nueva *clase* en Python.

Tutorial previo en un notebook

Antes de seguir, mírese el Notebook referente a **"Clases" en Python**
O acceda a la url: `https://mybinder.org/v2/gh/mbujosab/LibreriaDePythonParaMates2/master`

Usando lo mostrado en el Notebook anterior, definiremos una *clase* en Python para los vectores, otra para las matrices, otra para las matrices por bloques (o matrices particionadas) y otra para las operaciones elementales.

En Python, tanto las listas como las tuplas son "sistemas" (listas ordenadas de objetos). Así pues, usaremos listas (o tuplas) de números para instanciar la clase `Vector`. El sistema de números contenido en la lista (o tupla) será guardado como atributo de `Vector`. Veamos cómo:

Descripción del código Comenzamos la clase con el método de inicio: `def __init__(self, ... )`.

- La clase `Vector` usará dos argumentos. Al primero lo llamaremos `sis` y podrá ser una lista o tupla [2]. El segundo argumento (`rpr`) nos permitirá indicar si queremos que el entorno Jupyter Notebook represente el vector en forma horizontal o en vertical. Si no decimos nada, por defecto asumirá que debe representar el vector de manera vertical (`rpr='columna'`).

- Luego aparece un texto de ayuda que Python nos mostrará si escribimos: `help Vector`.

- Por último se definen tres atributos para la clase `Vector`: los atributos `lista`, `rpr` y `n`.
  (El modo de generar el atributo `lista` *depende del tipo de objeto que es* `sis`*).*

Cuando el argumento `sis` es una lista o una tupla, en el atributo "`lista`" se guarda el correspondiente sistema en forma de una `list` de Python: `self.lista = list(sis)`

De esta manera el atributo `self.lista` contendrá la lista ordenada de números que constituye el vector. Por tanto *¡ya hemos traducido al lenguaje Python la definición de vector!*

Por conveniencia definimos un par de atributos más. El atributo `self.n` guarda el número de elementos de la lista. El atributo `self.rpr` indica si el vector ha de ser representado como fila o como columna (representación en columna por defecto).

[2] también se podrá instanciar con un `Vector`

```
4  ⟨Definición de la clase Vector 4⟩≡
     class Vector:
         def __init__(self, sis, rpr='columna'):
             ⟨Texto de ayuda de la clase Vector 30b⟩

             if isinstance(sis, (list,tuple)):
                 self.lista  =  list(sis)
             ⟨Creación del atributo lista cuando sis es un Vector 31a⟩

             self.rpr  =  rpr
             self.n    =  len (self.lista)
```

This definition is continued in chunks 6a and 32a.
This code is used in chunk 28.
Defines:
`Vector`, used in chunks 5, 7, 9–11, 14a, 24b, 25b, and 30–36.

1.2 La clase `Matrix`

En las notas usamos la siguiente definición

Llamamos *matriz* de $\mathbb{R}^{m\times n}$ a un sistema de n vectores de $\mathbb{R}^m$.

Cuando representamos las matrices, las encerramos entre corchetes

$$\mathbf{A} = [\boldsymbol{v}_1, \ldots, \boldsymbol{v}_m]$$

Por tanto, al definir la clase `Matrix`, el modo principal de instanciación será una lista de vectores (todos con el mismo número de componentes). La lista de `Vector`es será la lista de "columnas" de la matriz.
(posteriormente añadimos otros dos modos adicionales de instanciar que nos vendrán bien más adelante: con una lista (o tupla) de listas o tuplas (en este caso las listas de coeficientes serán las distintas filas de la matriz), con una `BlockMatrix`*, o con otra* `Matrix`.

Descripción del código Comenzamos la clase con el método de inicio: `def __init__(self, sis)`.

- La clase `Matrix` se instancia con el argumento `sis`.
- Luego aparece el texto de ayuda.
- Por último se definen tres atributos para la clase `Matrix`. Los atributos: `lista`, `m` y `n`.
  (El modo de generar el atributo `lista` *depende del tipo de objeto que es* `sis`*. En el siguiente recuadro se destaca el caso en que* `sis` *es una lista de* `Vector`*es).*

El atributo `self.lista` guarda una lista de `Vector`es, (una lista de columnas).

El modo de elaborar dicha lista difiere en función de si el argumento `sis` con el que se instancia `Matrix` es otra `Matrix` (entonces `self.lista` es la `lista` de dicha matriz), si es una `BlockMatrix` (entonces se construye una matriz a partir de la matriz por bloques), de si el primer elemento de `sis` es una lista o tupla (entonces se interpreta que `sis` una "una lista de filas"), o si el primer elemento de `sis` es un `Vector` (entonces la `lista` es `list(sis)`, la lista de `Vector`es introducidos). Por tanto *¡ya hemos traducido al lenguaje Python la definición de vector!*

Por conveniencia definimos un par de atributos más. El atributo `self.m` guarda el número de filas de la matriz, y `self.n` guarda el número de columnas.

```
5  ⟨Definición de la clase Matrix 5⟩≡
     class Matrix:
         def __init__(self, sis):
             ⟨Texto de ayuda de la clase Matrix 32b⟩

             ⟨Creación del atributo lista cuando sis no es una lista o tupla de Vectores 33a⟩

             elif isinstance(sis[0], Vector):
                 ⟨Verificación de que todas las columnas de la matriz tendrán la misma longitud 33d⟩
                 self.lista  =  list(sis)

             self.m  =  self.lista[0].n
             self.n  =  len(self.lista)
```

This definition is continued in chunks 6b, 23, 24a, and 35a.
This code is used in chunk 28.
Defines:
  Matrix, used in chunks 8, 9, 11–14, 20, 22, 23, 25–27, 32–34, and 36.
Uses Vector 4.

Resumen

Los **vectores** son listas ordendas de números. La clase `Vector` almacena una lista ordenanda en el atributo `lista` de dos modos distintos:

1. Cuando se instancia la clase con una lista o tupla, dicha lista o tupla se almacena en el atributo `lista`.
2. Cuando se instancia la clase con otro `Vector`, se copia el atributo `lista` de dicho `Vector`.

Las **matrices** son listas ordendas de vectores (dichos vectores son sus columnas). La clase `Matrix` almacena una lista ordenanda de `Vector`es en el atributo `lista` de cuatro modos distintos (el código de los tres últimos se puede consultar en la Parte II de este documento):

1. Cuando se instancia la clase con una lista o tupla de `Vector`es, dicha lista o tupla se almacena en el atributo `lista`. *Esta es la forma de crear una matriz a partir de sus columnas.*
2. Cuando se instancia la clase con una lista (o tupla) de listas o tuplas, se interpreta que dicha lista (o tupla) describe la filas de una matriz. Consecuentemente, se dan los pasos para describir esa misma matriz como una lista de `Vector`es que se almacena en el atributo `lista`. (Esta forma de instanciar una `Matrix` se usará para programar la transposición).
3. Cuando se instancia la clase con otra `Matrix`, se copia el atributo `lista` de dicha `Matrix`.
4. Cuando se instancia la clase con una `BlockMatrix`, se unifican los bloques en una sola matriz, cuya lista de `Vector`es es copiada en el atributo `lista`.

Así pues, `Vector` es una lista ordenada (almacenada en su atributo `lista`) y `Matrix` es una lista ordenada de `Vector`es (almacenada en su atributo `lista`). Y por tanto:

¡Ya hemos implementado en Python los vectores y matrices tal y como se definen en las notas de clase!

Incorporamos los procedimientos que se describen en el resto de secciones a sus respectivas clases...

6a ⟨*Definición de la clase* `Vector` 4⟩+≡
⟨*Operador selector por la derecha para la clase* `Vector` 7⟩
⟨*Operador selector por la izquierda para la clase* `Vector` 8a⟩
⟨*Suma de* `Vector`*es* 10⟩
⟨*Producto de un* `Vector` *por un escalar a su izquierda, o por otro* `Vector` *a su izquierda* 11a⟩
⟨*Producto de un* `Vector` *por un escalar a su derecha, o por una* `Matrix` *a su derecha* 11b⟩
⟨*Definición de la igualdad entre* `Vector`*es* 12a⟩
This code is used in chunk 28.

6b ⟨*Definición de la clase* `Matrix` 5⟩+≡
⟨*Operador selector por la derecha para la clase* `Matrix` 8b⟩
⟨*Operador transposición para la clase* `Matrix` 9a⟩
⟨*Operador selector por la izquierda para la clase* `Matrix` 9b⟩
⟨*Suma de* `Matrix` 12b⟩
⟨*Producto de una* `Matrix` *por un escalar a su izquierda* 13⟩
⟨*Producto de una* `Matrix` *por un escalar, un vector o una matriz a su derecha* 14a⟩
⟨*Definición de la igualdad entre dos* `Matrix` 14b⟩
This code is used in chunk 28.

2 Operador selector

Notación en Mates 2

- Si $\boldsymbol{v} = (v_1, \ldots, v_n)$ entonces ${}_{i|}\boldsymbol{v} = \boldsymbol{v}_{|i} = v_i$ para todo $i \in \{1, \ldots, n\}$.
- Si $\mathbf{A} = \begin{bmatrix} a_{11} & \cdots & a_{1m} \\ \vdots & & \vdots \\ a_{n1} & \cdots & a_{nm} \end{bmatrix}$ entonces $\begin{cases} \mathbf{A}_{|j} = \begin{pmatrix} a_{1j} \\ \vdots \\ a_{nj} \end{pmatrix} \text{ para todo } j \in \{1, \ldots, m\} \\ {}_{i|}\mathbf{A} = (a_{i1}, \ldots, a_{im}) \text{ para todo } i \in \{1, \ldots, n\} \end{cases}$.

Queremos manejar la anterior notación, así que tenemos que definir el operador selector en Python. Usaremos el siguiente convenio:

Mates II	Python
$\boldsymbol{v}_{\|i}$	`v|i`
${}_{i\|}\boldsymbol{v}$	`i|v`
$\mathbf{A}_{\|j}$	`A|j`
${}_{i\|}\mathbf{A}$	`i|A`

Emplearemos los métodos especiales `__or__` y `__ror__`, que son las barras verticales a derecha e izquierda respectivamente. Pero puestos a seleccionar, aprovechemos la notación para seleccionar más de un elemento:

Notación en Mates 2

- ${}_{(i_1,\ldots,i_r)|}\boldsymbol{v} = \left(\boldsymbol{v}_{i_1}, \ldots, \boldsymbol{v}_{i_r}\right) = \boldsymbol{v}_{|(i_1,\ldots,i_r)}$ (es un vector formado por elementos de $\boldsymbol{v}$)
- ${}_{(i_1,\ldots,i_r)|}\mathbf{A} = \left[{}_{i_1|}\mathbf{A}, \ldots, {}_{i_r|}\mathbf{A}\right]^{\mathsf{T}}$ (es una matriz cuyas filas son filas de $\mathbf{A}$)
- $\mathbf{A}_{|(j_1,\ldots,j_r)} = \left[\mathbf{A}_{|j_1}, \ldots, \mathbf{A}_{|j_r}\right]$ (es una matriz formada por columnas de $\mathbf{A}$)

Selector por la derecha para la clase `Vector`. Cuando el argumento `i` es un número entero (`int`), seleccionamos el correspondiente elemento del atributo `lista` del `Vector` (recuerde que en Python los índices de listas y tuplas comienzan en cero, por lo que para seleccionar el elemento i-ésimo de `lista`, escribimos `lista[i-1]`).

Una vez hemos definido el operador "`|`" cuando el argumento `i` es un entero (`int`), podemos usarlo (`self|a`) para definir el operador cuando el argumento `i` es una lista o tupla `(list,tuple)` de índices (que genera un `Vector` de componentes selecionadas).

7 ⟨*Operador selector por la derecha para la clase* `Vector` 7⟩≡

```
def __or__(self,i):
    ⟨Texto de ayuda para el operador selector por la derecha para la clase Vector 31b⟩
    if isinstance(i,int):
        return self.lista[i-1]
    elif isinstance(i, (list,tuple) ):
        return Vector ([ (self|a) for a in i ])
```

This code is used in chunk 6a.
Uses `Vector` 4.

El selector por la izquierda hace lo mismo, así que basta con llamar al selector por la derecha: `self|i`

```
8a  ⟨Operador selector por la izquierda para la clase Vector 8a⟩≡
      def __ror__(self,i):
          ⟨Texto de ayuda para el operador selector por la izquierda para la clase Vector 31c⟩
          return self | i

    This code is used in chunk 6a.
```

Operador selector para la clase `Matrix`. Para las matrices es algo más complicado: recuerde que no es lo mismo operar por la derecha que por la izquierda. Además, un poco más adelante también extenderemos el uso de estos operadores para particionar matrices en bloques (las matrices por bloques están definidas en la clase `BlockMatrix`).

Selector por la derecha para la clase `Matrix`. Como el objeto `Matrix` es una lista de `Vector`es, el código para el selector por la derecha es casi idéntico al de la clase `Vector`. Como antes, una vez definido el operador "|" por la derecha para seleccionar una única columna (es decir, cuando el argumento `j` es un entero), podremos usar repetidamente el procedimiento `(self|j)` para crear una submatriz formada por una selección de columnas (es decir, cuando el parámetro `j` es una lista, o tupla, de índices).

(*la explicación de cómo se particiona una matriz en bloques de columnas de matrices se verá más adelante*).

```
8b  ⟨Operador selector por la derecha para la clase Matrix 8b⟩≡
      def __or__(self,j):
          ⟨Texto de ayuda para el operador selector por la derecha para la clase Matrix 34a⟩
          if isinstance(j,int):
              return self.lista[j-1]

          elif isinstance(j, (list,tuple)):
              return Matrix ([ self|a for a in j ])

          ⟨Partición de una matriz por columnas de bloques 17c⟩

    This code is used in chunk 6b.
    Uses Matrix 5.
```

Operador transposición. Como paso intermedio vamos a definir el operador transposisción, que usaremos después para definir el operador selector por la izquierda (selección de filas).

Recuerde que con la segunda forma de instanciar el objeto `Matrix` (véase el resumen de la página 5), creamos una matriz a partir de la lista de sus filas. Así podemos construir fácilmente el operador trasposición. Basta instanciar `Matrix` con la lista de los n atributos "`lista`" correspondientes a los consecutivos n `Vector`es columna.
(Recuerde también que `range(1,self.m+1)` recorre los números: $1, 2, \ldots, m$).

Como se indicó en la introducción de este documento, en Python no disponemos del símbolo habitual para transponer "$^\intercal$", así que nos vemos obligados a usar el símbolo `__invert__` por la izquierda "~":

Mates II	Python
$\mathbf{A}^\intercal$	`~A`

Para transponer, sencillamente creamos una `Matrix` con la lista de atributos `lista` de los vectores columna:

9a ⟨*Operador transposición para la clase* `Matrix` 9a⟩≡

```
def __invert__(self):
    ⟨Texto de ayuda para el operador transposición de la clase Matrix 34b⟩
    return Matrix ([ (self|j).lista for j in range(1,self.n+1) ])
```

This code is used in chunk 6b.
Uses `Matrix` 5.

Selector por la izquierda para la clase `Matrix`. Con el operador selector por la derecha y la transposición, podemos definir el operador selector por la izquierda que selecciona *filas*... ¡que son las columnas de la matriz transpuesta!.

```
(~self)|j
```

(Para recordar que el vector ha sido obtenido de la fila de una matriz, lo representaremos en horizontal)

De nuevo, una vez definido el operador por la izquierda, podemos usar dicho procedimiento repetidas veces para formar una submatriz con varias filas (cuyos índices estén recogidos en una lista o tupla).

9b ⟨*Operador selector por la izquierda para la clase* `Matrix` 9b⟩≡

```
def __ror__(self,i):
    ⟨Texto de ayuda para el operador selector por la derecha para la clase Matrix 34a⟩
    if isinstance(i,int):
        return Vector ( (~self)|i , rpr='fila')

    elif isinstance(i, (list,tuple)):
        return Matrix ([ (a|self).lista  for a in i ])

    ⟨Partición de una matriz por filas de bloques 17b⟩
```

This code is used in chunk 6b.
Uses `Matrix` 5 and `Vector` 4.

Resumen

¡Ahora también hemos implementado en Python el operador "|" tanto por la derecha como por la izquierda tal y como se define en las notas de clase!

Ya estamos listos para definir el resto de operaciones con vectores y matrices...

3 Operaciones con vectores y matrices

Una vez definidas las clases `Vector` y `Matrix` junto con el operador selector "`|`", ya podemos definir las operaciones de suma y producto. Fíjese que las definiciones de las operaciones en Pyhton (usando el operador "`|`") son idénticas a las empleadas en las notas de clase:

Suma de vectores En las notas de clase hemos definido la suma de vectores como

$$\boxed{(\boldsymbol{a}+\boldsymbol{b})_{|i} = (\boldsymbol{a})_{|i} + (\boldsymbol{b})_{|i}} \qquad \text{para } i = 1, \ldots, n.$$

ahora usando el operador selector, podemos literalmente transcribir esta definición

```
Vector ([ (self|i) + (other|i) for i in range(1,self.n+1) ])
```

donde `self` es el vector y `other` es otro vector.

10 ⟨*Suma de* `Vector`*es* 10⟩≡

```
def __add__(self, other):
    ⟨Texto de ayuda para el operador suma en la clase Vector 35b⟩
    if isinstance(other, Vector):
        if self.n == other.n:
            return Vector ([ (self|i) + (other|i) for i in range(1,self.n+1) ])
        else:
            print("error en la suma: vectores con distinto número de componentes")
```

This code is used in chunk 6a.
Uses `Vector` 4.

Producto de un vector por un escalar, por otro vector o por una matriz En las notas hemos definido

- El producto de $\boldsymbol{a}$ por un escalar x a su izquierda como

  $$\boxed{(x\boldsymbol{a})_{|i} = x(\boldsymbol{a}_{|i})} \qquad \text{para } i = 1, \ldots, n.$$

  cuya transcripción será

  ```
  Vector ([ x*(self|i) for i in range(1,self.n+1) ])
  ```

  donde `self` es el vector y `x` es un número entero, de coma flotante o fracción (`int, float, Fraction`).

- El *producto punto* (o producto escalar usual en $\mathbb{R}^n$) de dos vectores $\boldsymbol{x}$ y $\boldsymbol{a}$ en $\mathbb{R}^n$ es

  $$\boxed{\boldsymbol{x}\cdot\boldsymbol{a} \;=\; \sum_{i=1}^{n} x_i a_i} \qquad \text{para } i = 1, \ldots, n.$$

  cuya transcripción será

  ```
  sum([ (x|i)*(self|i) for i in range(1,self.n+1) ])
  ```

  donde `self` es el vector y `x` es otro vector (`Vector`).

11a ⟨*Producto de un* `Vector` *por un escalar a su izquierda, o por otro* `Vector` *a su izquierda* 11a⟩≡

```
def __rmul__(self, x):
    ⟨Texto de ayuda para el operador producto por la izquierda en la clase Vector 36a⟩
    if isinstance(x, (int, float, Fraction)):
        return Vector ([ x*(self|i) for i in range(1,self.n+1) ])

    elif isinstance(x, Vector):
        if self.n == x.n:
            return sum([ (x|i)*(self|i) for i in range(1,self.n+1) ])
        else:
            print("error en producto: vectores con distinto número de componentes")
```

This code is used in chunk 6a.
Uses `Vector` 4.

- Por otra parte, en las notas se acepta que el producto de un vector por un escalar es conmutativo. Por tanto,

$$\boxed{\boldsymbol{b}x = x\boldsymbol{b}}$$

cuya transcripción será

```
x * self
```

donde `self` es el vector y `x` es un número entero, o de coma flotante o una fracción (`int, float, Fraction`).

- El producto de un vector $\boldsymbol{a}$ de $\mathbb{R}^n$ por una matriz $\mathbf{B}$ con n filas es

$$\boxed{\boldsymbol{a}\mathbf{B} = \mathbf{B}^\intercal\boldsymbol{a}}$$

cuya transcripción será

```
(~x) * self
```

donde `self` es el vector y `x` es una matriz (`Matrix`). Para recordar que es una combinación lineal de las filas, su representación es en forma de fila.

11b ⟨*Producto de un* `Vector` *por un escalar a su derecha, o por una* `Matrix` *a su derecha* 11b⟩≡

```
def __mul__(self, x):
    ⟨Texto de ayuda para el operador producto por la derecha en la clase Vector 35c⟩
    if isinstance(x, (int, float, Fraction)):
        return x*self

    elif isinstance(x, Matrix):
        if self.n == x.m:
            return Vector( (~x)*self, rpr='fila')
        else:
            print("error en producto: Vector y Matrix incompatibles")
```

This code is used in chunk 6a.
Uses `Matrix` 5 and `Vector` 4.

Igualdad entre vectores En las notas de clase se dice que *dos vectores son iguales solo cuando lo son las listas correspondientes a ambos vectores.*

Dos vectores serán iguales si y solo si son idénticos los correspondientes sistemas:

12a ⟨*Definición de la igualdad entre* `Vectores` 12a⟩≡

```
def __eq__(self, other):
    return self.lista == other.lista
```

This code is used in chunk 6a.

Matrices

Suma de matrices Hemos definido la suma de matrices como

$$\boxed{(\mathbf{A}+\mathbf{B})_{|j} = (\mathbf{A})_{|j} + (\mathbf{B})_{|j}} \qquad \text{para } i = 1, \ldots, n.$$

de nuevo, usando el operador selector podemos transcribir literalmente esta definición

```
Matrix ([ (self|i) + (other|i) for i in range(1,self.n+1) ])
```

donde `self` es la matriz y `other` es otra matriz.

12b ⟨*Suma de* `Matrix` 12b⟩≡

```
def __add__(self, other):
    ⟨Texto de ayuda para el operador suma en la clase Matrix 36b⟩
    if isinstance(other,Matrix) and self.m == other.m and self.n == other.n:
        return Matrix ([ (self|i) + (other|i) for i in range(1,self.n+1) ])
    else:
        print("error en la suma: matrices con distinto orden")
```

This code is used in chunk 6b.
Uses `Matrix` 5.

Producto de una matriz por un escalar, por un vector o por otra matriz En las notas hemos definido

- El producto de $\mathbf{A}$ por un escalar x a su izquierda como

$$\boxed{(x\mathbf{A})_{|j} = x(\mathbf{A}_{|j})} \qquad \text{para } i = 1, \ldots, n.$$

cuya transcripción será

```
Matrix ([ x*(self|i) for i in range(1,self.n+1) ])
```

donde `self` es la matriz y `x` es un número entero, de coma flotante o fracción (`int, float, Fraction`).

13 ⟨*Producto de una* `Matrix` *por un escalar a su izquierda* 13⟩≡
```
def __rmul__(self,x):
    ⟨Texto de ayuda para el operador producto por la derecha en la clase Matrix 36c⟩
    if isinstance(x, (int, float, Fraction)):
        return Matrix ([ x*(self|i) for i in range(1,self.n+1) ])
```
This code is used in chunk 6b.
Uses `Matrix` 5.

- En las notas se acepta que el producto de una matrix por un escalar es conmutativo. Por tanto,

$$\boxed{\mathbf{A}x = x\mathbf{A}}$$

cuya transcripción será

```
x * self
```

donde `self` es la matriz y `x` es un número entero, o de coma flotante o una fracción (`int, float, Fraction`).

- El producto de $\underset{m\times n}{\mathbf{A}}$ por un vector $\boldsymbol{x}$ de $\mathbb{R}^n$ a su derecha se define como

$$\boxed{\mathbf{A}\boldsymbol{x} \;=\; \sum_{j=1}^{n} x_j \mathbf{A}_{|j}} \qquad \text{para } j = 1, \ldots, n.$$

cuya transcripción será

```
sum([ (x|j)*(self|j) for j in range(1,self.n+1) ])
```

donde `self` es el vector y `x` es otro vector (`Vector`).

- El producto de $\underset{m\times k}{\mathbf{A}}$ por otra matriz $\underset{k\times n}{\mathbf{X}}$ de $\mathbb{R}^n$ a su derecha se define como

$$\boxed{(\mathbf{A}\mathbf{X})_{|j} = \mathbf{A}(\mathbf{X}_{|j})} \qquad \text{para } j = 1, \ldots, n.$$

cuya transcripción será

```
Matrix( [ self*(x|j) for j in range(1,x.n+1)] )
```

donde `self` es la matriz y `x` es otra matriz (`Matrix`).

14a ⟨*Producto de una* `Matrix` *por un escalar, un vector o una matriz a su derecha* 14a⟩≡

```
def __mul__(self,x):
    ⟨Texto de ayuda para el operador producto por la izquierda en la clase Matrix 36d⟩
    if isinstance(x, (int, float, Fraction)):
        return x*self

    elif isinstance(x, Vector):
        if self.n == x.n:
            return sum( [(x|j)*(self|j) for j in range(1,self.n+1)], V0(self.m) )
        else:
            print("error en producto: vector y matriz incompatibles")

    elif isinstance(x, Matrix):
        if self.n == x.m:
            return Matrix( [ self*(x|j) for j in range(1,x.n+1)])
        else:
            print("error en producto: matrices incompatibles")
```

This code is used in chunk 6b.
Uses `Matrix` 5 and `Vector` 4.

Igualdad entre matrices *Dos matrices son iguales solo cuando lo son las listas correspondientes a ambas.*

Dos matrices serán iguales si y solo si son idénticos los correspondientes sistemas:

14b ⟨*Definición de la igualdad entre dos* `Matrix` 14b⟩≡

```
def __eq__(self, other):
    return self.lista == other.lista
```

This code is used in chunk 6b.

4 Matrices particionadas (o matrices por bloques)

Las matrices por bloques o cajas $\boxed{\mathbf{A}}$ son tablas de matrices de modo que todas las matrices de una misma fila comparten el mismo número de filas, y todas las matrices de una misma columna comparten el mismo número de columnas. Por ello al "pegar" todas ellas obtenemos una gran matriz.

El argumento de inicialización `lista` es una lista (o tupla) de listas de matrices, cada una de las listas de matrices es una fila de bloques.

15 ⟨*Definición de la clase* `BlockMatrix` 15⟩≡

```
class BlockMatrix:
    def __init__(self, lista):
        """ Inicializa una matriz por bloques usando una lista de listas de matrices.
        """
        self.lista = lista
        self.m     = len(lista)
        self.n     = len(lista[0])
        self.lm    = [fila[0].m for fila in lista]
        self.ln    = [c.n for c in lista[0]]
```

This definition is continued in chunks 16, 18, and 19a.
This code is used in chunk 28.
Defines:
`BlockMatrix`, used in chunks 17–19, 27b, and 32–34.

El atributo `self.m` contiene el número de filas (de matrices) y `self.n` contiene el número de columnas (de matrices). Vemos que básicamente es una matriz a la que añadimos los atributos `self.ln`, que es una lista por el número de filas que tienen las matrices de cada fila, y `self.lm` con el número de columnas de la matrices de cada columna.

A continuación definimos las reglas de representación para las matrices por bloques. `Matrix` y `BlockMatrix` son objetos distintos, y los bloques se separan con lineas verticales y horizontales. Pero si hay un único bloque, no habrá ninguna línea vertical u horizontal por medio de la representación de la `BlockMatrix`. Así, si una matriz por bloques tienen un único bloque, pintaremos una caja alrededor para distinguirla de una matriz ordinaria.

16 ⟨*Definición de la clase* BlockMatrix 15⟩+≡

```
        def __repr__(self):
            """ Muestra una matriz en su representación python """
            return 'BlockMatrix(' + repr(self.lista) + ')'

        def _repr_html_(self):
            """ Construye la representación para el  entorno jupyter notebook """
            return html(self.latex())

        def latex(self):
            """ Escribe el código de LaTeX """
            if self.m == self.n == 1:
                return \
                  '\\begin{array}{|c|}' + \
                  '\\hline ' + \
                  '\\\\ \\hline '.join( \
                        ['\\\\'.join( \
                        ['&'.join( \
                        [latex(self.lista[0][0]) ]) ]) ]) + \
                  '\\\\ \\hline ' + \
                  '\\end{array}'
            else:
                return \
                  '\\left[' + \
                  '\\begin{array}{' + '|'.join([n*'c' for n in self.ln]) + '}' + \
                  '\\\\ \\hline '.join( \
                        ['\\\\'.join( \
                        ['&'.join( \
                        [latex(self.lista[i][j]|k|s) \
                        for j in range(self.n) for k in range(1,self.ln[j]+1) ]) \
                        for s in range(1,self.lm[i]+1) ]) for i in range(self.m) ]) + \
                  '\\\\' + \
                  '\\end{array}' + \
                  '\\right]'
```

This code is used in chunk 28.

4.1 Particionado de matrices

Volvamos al particionado de matrices:

Notación en Mates 2

- Si $n \leq m \in \mathbb{N}$ denotaremos con $(n : m)$ a la secuencia $n, n+1, \ldots, m$, (es decir, a la lista ordenada de los números de $\{k \in \mathbb{N} | n \leq k \leq m\}$).
- Si $i_1, \ldots, i_r \in \mathbb{N}$ con $i_1 < \ldots < i_r \leq n$ donde n es el número de filas de $\mathbf{A}$, entonces ${}_{\{i_1,\ldots,i_r\}|}\mathbf{A}$ es la matriz de bloques

$$\left[\begin{array}{c} {}_{(1:i_1)|}\mathbf{A} \\ \hline {}_{(i_1+1:i_2)|}\mathbf{A} \\ \hline \vdots \\ \hline {}_{(i_r+1:n)|}\mathbf{A} \end{array}\right]$$

- Si $j_1, \ldots, j_s \in \mathbb{N}$ con $j_1 < \ldots < j_s \leq m$ donde m es el número de columnas de $\mathbf{A}$, entonces $\mathbf{A}_{|\{j_1,\ldots,j_s\}}$ es la matriz de bloques

$$\left[\begin{array}{c|c|c|c} \mathbf{A}_{|(1:j_1)} & \mathbf{A}_{|(j_1+1:j_2)} & \cdots & \mathbf{A}_{|(j_s+1:m)} \end{array}\right]$$

Comencemos construyendo la partición a partir del conjunto y un número (que indique el número total de filas o columnas);

17a ⟨*Definición del método* `particion` 17a⟩≡

```
def particion(s,n):
    """ genera la lista de particionamiento a partir de un conjunto y un número
    >>> particion({1,3,5},7)

    [[1], [2, 3], [4, 5], [6, 7]]
    """
    p = list(s | set([0,n]))
    return [ list(range(p[k]+1,p[k+1]+1)) for k in range(len(p)-1) ]
```

This code is used in chunk 28.

y ahora el método de partición por filas o columnas resulta inmediato:

17b ⟨*Partición de una matriz por filas de bloques* 17b⟩≡

```
elif isinstance(i,set):
    return BlockMatrix ([ [a|self] for a in particion(i,self.m) ])
```

This code is used in chunk 9b.
Uses `BlockMatrix` 15.

17c ⟨*Partición de una matriz por columnas de bloques* 17c⟩≡

```
elif isinstance(j,set):
    return BlockMatrix ([ [self|a for a in particion(j,self.n)] ])
```

This code is used in chunk 8b.
Uses `BlockMatrix` 15.

Pero aún nos falta algo:

Notación en Mates 2

- Si $i_1, \ldots, i_r \in \mathbb{N}$ con $i_1 < \ldots < i_r \leq n$ donde n es el número de filas de $\mathbf{A}$ y $j_1, \ldots, j_s \in \mathbb{N}$ con $j_1 < \ldots < j_s \leq m$ donde m es el número de columnas de $\mathbf{A}$ entonces

$$ {}_{\{i_1,\ldots,i_y\}|}\mathbf{A}_{|\{j_1,\ldots,j_s\}} = \left[\begin{array}{c|c|c|c} {}_{(1:i_1)|}\mathbf{A}_{|(1:j_1)} & {}_{(1:i_1)|}\mathbf{A}_{|(j_1+1:j_2)} & \cdots & {}_{(1:i_1)|}\mathbf{A}_{|(j_s+1:m)} \\ \hline {}_{(i_1+1:i_2)|}\mathbf{A}_{|(1:j_1)} & {}_{(i_1+1:i_2)|}\mathbf{A}_{|(j_1+1:j_2)} & \cdots & {}_{(i_1+1:i_2)|}\mathbf{A}_{|(j_s+1:m)} \\ \hline \vdots & \vdots & \cdots & \vdots \\ \hline {}_{(i_k+1:n)|}\mathbf{A}_{|(1:j_1)} & {}_{(i_k+1:n)|}\mathbf{A}_{|(j_1+1:j_2)} & \cdots & {}_{(i_k+1:n)|}\mathbf{A}_{|(j_s+1:m)} \end{array}\right] $$

es decir, queremos poder particionar una matriz de bloques. Los casos que nos interesan son hacerlo por el lado contrario por el que se particionó la matriz de partida , es decir,

$$ {}_{\{i_1,\ldots,i_r\}|}\left(\mathbf{A}_{|\{j_1,\ldots,j_s\}}\right) \qquad \text{y} \qquad \left({}_{\{i_1,\ldots,i_r\}|}\mathbf{A}\right)_{|\{j_1,\ldots,j_s\}} $$

que, por supuesto, da el mismo resultado.

Para dichos casos, programamos el siguiente código que particiona una `BlockMatrix`. Primero el procedimiento para particionar por columnas (inicialmente si sólo hay una fila de matrices, ya que el caso general se vera un poco más abajo):

Es sencillo, si `self.n == 1` la matriz por bloques tiene una única columna.

18 ⟨*Definición de la clase* `BlockMatrix` 15⟩+≡

```
def __or__(self,j):
    """ Reparticiona por columna una matriz por cajas """
    if isinstance(j,set):
        if self.n == 1:
            return BlockMatrix([ [ self.lista[i][0]|a  \
                                   for a in particion(j,self.lista[0][0].n)] \
                                   for i in range(self.m) ])

        ⟨Caso general de reparticion por columnas 20a⟩
```

This code is used in chunk 28.
Uses `BlockMatrix` 15.

y hacemos lo mismo para particionar por filas cuando `self.m == 1` (la matriz por bloques tiene una única fila):

19a ⟨*Definición de la clase* `BlockMatrix` 15⟩+≡

```
def __ror__(self,i):
    """ Reparticiona por filas una matriz por cajas """
    if isinstance(i,set):
        if self.m == 1:
            return BlockMatrix([[ a|self.lista[0][j]  \
                                  for j in range(self.n) ] \
                                  for a in particion(i,self.lista[0][0].m)])

        ⟨Caso general de reparticion por filas 20b⟩
```

This code is used in chunk 28.
Uses `BlockMatrix` 15.

Pero aún nos falta el código del caso general. Debemos decidir el significado de reparticionar una matriz por el mismo lado por el que ya ha sido particionada. Seguiremos un criterio práctico... eliminar el anterior particionado y aplicar el nuevo.

$$\langle i'_1,\ldots,i'_r\rangle| \left(\langle i_1,\ldots,i_k\rangle|\mathbf{A}|\langle j_1,\ldots,j_s\rangle \right) = \langle i'_1,\ldots,i'_r\rangle|\mathbf{A}|\langle j_1,\ldots,j_s\rangle$$

$$\left(\langle i_1,\ldots,i_k\rangle|\mathbf{A}|\langle j_1,\ldots,j_s\rangle \right)|\langle j'_1,\ldots,j'_r\rangle = \langle i_1,\ldots,i_k\rangle|\mathbf{A}|\langle j'_1,\ldots,j'_r\rangle$$

Para ello nos viene bien extraer el conjunto selector a partir del resultado:

19b ⟨*Definición del procedimiento de generación del conjunto clave para particionar* 19b⟩≡

```
def key(L):
    """ genera el conjunto clave a partir de una secuencia de tamaños
    número
    >>> key([1,2,1])

    {1, 3, 4}
    """
    return set([ sum(L[0:i]) for i in range(1,len(L)+1) ])
```

This code is used in chunk 28.

Así, los casos generales consisten en reparticionar de nuevo:

20a ⟨*Caso general de reparticion por columnas* 20a⟩≡

```
elif self.n > 1:
    return (key(self.lm) | Matrix(self)) | j
```

This code is used in chunk 18.
Uses `Matrix` 5.

20b ⟨*Caso general de reparticion por filas* 20b⟩≡

```
elif self.m > 1:
    return i | (Matrix(self) | key(self.ln))
```

This code is used in chunk 19a.
Uses `Matrix` 5.

Observación 1. El método `__or__` está definido para conjuntos . . . realiza la unión. Por tanto si `A` es una matriz, la orden `{1,2}|({3}|A)` no da igual que `({1,2}|{3})|A`. La primera es igual da `{1,2}|A`, mientras que la segunda da `{1,2,3}|A`.

¿Deben extenderse el resto de funcionalidades a las matrices por cajas? . . . depende de las necesidades (por ahora, así dejamos las `BlockMatrix`).

5 Transformaciones elementales

Notación en Mates 2

Si $\mathbf{A}$ es una matriz

Tipo I: $\underset{[i+\lambda\cdot j]}{\tau}\mathbf{A}$ suma a la fila i la fila j multiplicada por λ; $\mathbf{A}\underset{[i+\lambda\cdot j]}{\tau}$ lo mismo con las columnas.

Tipo II: $\underset{[\lambda\cdot i]}{\tau}\mathbf{A}$ multiplica la fila i por λ; y $\mathbf{A}\underset{[\lambda\cdot i]}{\tau}$ multiplica la columna i por λ.

Intercambio: $\underset{[i\rightleftharpoons j]}{\tau}\mathbf{A}$ intercambia las filas i y j; y $\mathbf{A}\underset{[i\rightleftharpoons j]}{\tau}$ intercambia las columnas.

Como una trasformación elemental es el resultado del producto con una matriz elemental, esta notación busca el parecido con la notación del producto matricial. Con ello se gana, entre otras cosas, que la notación sea asociativa. Pero entonces se plantea ¿qué ventaja tiene introducir en el discurso las transformaciones elementales en lugar de utilizar simplemente matrices elementales? En principio hay dos:

1. Una matriz cuadrada es un objeto muy pesado... n^2 coeficientes para una matriz de orden n. Afortunadamente una matriz elemental es casi una matriz identidad salvo por uno de sus elementos; por tanto, para describir una matriz elemental basta indicar su orden n y el coeficiente que no coincide con los de $\mathbf{I}_n$.
2. Las trasformaciones elementales, indicando dicho coeficiente, omiten el orden n.

Escogemos la siguiente traducción de esta notación:

Mates II	Python	Mates II	Python
$\mathbf{A}\underset{[i\rightleftharpoons j]}{\tau}$	`A & T({i,j})`	$\underset{[i\rightleftharpoons j]}{\tau}\mathbf{A}$	`T({i,j}) & A`
$\mathbf{A}\underset{[a\cdot i]}{\tau}$	`A & T((i,a))`	$\underset{[a\cdot i]}{\tau}\mathbf{A}$	`T((i,a)) & A`
$\mathbf{A}\underset{[i+a\cdot j]}{\tau}$	`A & T((i,j,a))`	$\underset{[i+a\cdot j]}{\tau}\mathbf{A}$	`T((i,j,a)) & A`

Vemos que:

1. Representar el intercambio con un conjunto, permite admitir la repetición del índice $\{i,i\}=\{i\}$ como un caso especial en el que la matriz no cambia. Esto simplificará el método de Gauss.
2. Tanto en los pares (i,a) como en las ternas (i,j,a)
   (a) La columna (fila) que cambia es la del índice que aparece en primera posición.
   (b) El escalar aparece el la ultima posición y multiplica a la columna (fila) con el índice que le precede.

Además vamos a extender esta notación para secuencias de trasformaciones elementales ($\underset{\tau_k\cdots\tau_1}{\tau}\mathbf{A}$ y $\mathbf{A}\underset{\tau_1\cdots\tau_k}{\tau}$) ...lo que nos vendrá de perlas para programar a lo Python:

$$t_k\ \&\cdots\&\ t_2\ \&\ t_1\ \&\ \mathbf{A} \;=\; [t_1,t_2,\ldots,t_k]\ \&\ \mathbf{A}$$
$$\mathbf{A}\ \&\ t_1\ \&\ t_2\ \&\cdots\&\ t_k \;=\; \mathbf{A}\ \&\ [t_1,t_2,\ldots,t_k]$$

Para ello, vamos a definir un nuevo tipo de dato: `T` (transformación elemental) que nos permitirá encadenar transformaciones elementales.

22 ⟨*Definición de la clase de transformaciones elementales:* T 22⟩≡

```
class T:
    def __init__(self, t):
        """ Inicializa una transformación elemental """
        self.t = t

    def __and__(self,t):
        """ Crea una trasformación composición de dos
        >>> T((1,2)) & T({2,4})

        T([(1,2), {2,4}])

        O aplica la transformación sobre una matriz A
        >>> A & T({1,2})    (intercambia las dos primeras columnas de A)
        """
        def CreaLista(a):
            """Transforma una una tupla en una lista que contiene la tupla"""
            return (a if isinstance(a,list) else [a])

        if isinstance(t,T):
            return T(CreaLista(self.t) + CreaLista(t.t))

        if isinstance(t,Matrix):
            return t.__rand__(self)
```

This code is used in chunk 28.
Uses Matrix 5.

Y ahora definimos las tres operaciones elementales (incluimos el intercambio, aunque usted sabe que realmente es una composición de los otros dos tipos de transformaciones):

23 ⟨*Definición de la clase* `Matrix` 5⟩+≡

```
def __and__(self,t):
    """ Aplica una o una secuencia de transformaciones elementales por columnas:
    >>>  A & T({1,3})                # intercambia las columnas 1 y 3
    >>>  A & T((1,5))                # multiplica la columna 1 por 5
    >>>  A & T((1,2,5))              # suma a la columna 1 la 2 por 5
    >>>  A & T([{1,3},(1,5),(1,2,5)]) # aplica la secuencia de transformaciones
    """
    if isinstance(t.t,set) and len(t.t) == 2:
        self.lista = Matrix( [(self|max(t.t)) if k==min(t.t) else \
                              (self|min(t.t)) if k==max(t.t) else \
                              (self|k) for k in range(1,self.n+1)]).lista

    elif isinstance(t.t,tuple) and len(t.t) == 2:
         self.lista = Matrix([ t.t[1]*(self|k) if k==t.t[0] else (self|k) \
                               for k in range(1,self.n+1)] ).lista

    elif isinstance(t.t,tuple) and len(t.t) == 3:
         self.lista = Matrix([ (self|k) + t.t[2]*(self|t.t[1]) if k==t.t[0] else \
                               (self|k) for k in range(1,self.n+1)] ).lista
    elif isinstance(t.t,list):
         for k in t.t:
             self & T(k)
    return self
```

This code is used in chunk 28.
Uses `Matrix` 5.

Observación 2. Las trasformaciones elementales modifican la matriz.

Y para transformaciones por filas filas usamos el truco de aplicar las operaciones sobre la filas de la transpuesta y de nuevo transponemos el resultado. Para una sucesión de transformaciones por la izquierda, tenemos en cuenta que se aplican en el orden inverso a como aparecen en la lista de transformaciones:

24a ⟨*Definición de la clase* `Matrix` 5⟩+≡

```
        def __rand__(self,t):
            """ Aplica una o una secuencia de transformaciones elementales por filas:
            >>>    {1,3} & A              # intercambia las filas 1 y 3
            >>>    (1,5) & A              # multiplica la fila 1 por 5
            >>>  (1,2,5) & A              # suma a la fila 1 la 2 por 5

            >>>  [(1,2,5),(1,5),{1,3}] & A # aplica la secuencia de transformaciones
            """
            if isinstance(t.t,set) | isinstance(t.t,tuple):
                self.lista = (~(~self & t)).lista

            elif isinstance(t.t,list):
                for k in reversed(t.t):
                    T(k) & self

            return self
```

This code is used in chunk 28.

Observación 3. Las trasformaciones elementales modifican la matriz.

6 Vectores y Matrices especiales

Notación en Mates 2

Los vectores cero **0** y las matrices cero **0** se pueden implementar como subclases de la clase `Vector` y `Matrix` (pero tenga en cuenta que Python necesita conocer el número de componentes del vector y el orden de la matriz):

24b ⟨*Definición de vector nulo:* `V0` 24b⟩≡

```
  class V0(Vector):
        def __init__(self, n ,rpr = 'columna'):
            """ Inicializa el vector nulo de n componentes"""

            super(self.__class__ ,self).__init__([0 for i in range(n)],rpr)
```

This code is used in chunk 28.
Uses `Vector` 4.

Y lo mismo hacemos para matrices

25a ⟨*Definición de matriz nula:* `M0` 25a⟩≡

```
class M0(Matrix):

    def __init__(self, m, n=None):
        """ Inicializa una matriz nula de orden n """
        if n is None:
            n = m

        super(self.__class__ ,self).__init__( \
                      [[0 for i in range(n)] for j in range(m)])
```

This code is used in chunk 28.
Uses `Matrix` 5.

También debemos definir la matriz identidad de orden n (y sus filas y columnas). En los apuntes de clase no solemos indicar expresamente el orden de la matriz identidad (pues normalmente se sobrentiende por el contexto). Pero esta habitual imprecisión no nos la podemos permitir con el ordenador.

Notación en Mates 2

- $\mathbf{I}$ (de orden n) es la matriz tal que ${}_{i|}\mathbf{I}_{|j} = \begin{cases} 1 & \text{si } j = i \\ 0 & \text{si } j \neq i \end{cases}$.

La definición de la fila o columna i-ésima de la identidad ($\mathbf{I}_{|i} = {}_{i|}\mathbf{I}$) creo que me la puedo ahorrar.

25b ⟨*Definición de vector fila o columna de la matriz identidad* `e` 25b⟩≡

```
class e(Vector):

    def __init__(self, i,n ,rpr = 'columna'):
        """ Inicializa el vector e_i  de tamaño n """

        super(self.__class__ ,self).__init__([((i-1)==k)*1 for k in range(n)],rpr)
```

This code is used in chunk 28.
Uses `Vector` 4.

Lo importante es la matriz identidad.

26 ⟨*Definición de la matriz identidad:* `I` 26⟩≡

```
class I(Matrix):

    def __init__(self, n):
        """ Inicializa la matriz identidad de tamaño n """

        super(self.__class__ ,self).__init__(\
                    [[(i==j)*1 for i in range(n)] for j in range(n)])
```

This code is used in chunk 28.
Uses `Matrix` 5.

7 Ejemplo de uso

Aunque todavía falta definir las operaciones de suma y producto entre matrices y producto de un vector con una matriz, con esto ya podemos hacer muchísimas cosas. Por ejemplo, ¡eliminación gaussiana para encontrar el espacio nulo de una matriz!

27a ⟨*normal* 27a⟩≡

```
class Normal(Matrix):
    def __init__(self, data):
        """ Escalona por Gauss obteniendo una matriz cuyos pivotes son unos """
        def pivote(v,k):
            """ Devuelve el primer índice mayor que k de de un
            un coeficiente no nulo del vector v. En caso de no existir
            devuelve 0
            """
            return ([x[0] for x in enumerate(v.lista, 1) \
                                if (x[1] !=0 and x[0] > k)]+[0])[0]

        A = Matrix(data)
        r = 0
        self.rank = []
        for i in range(1,A.n+1):
           p = pivote((i|A),r)
           if p > 0:
              r += 1
              A & T({p,r})
              A & T((r,inverso(i|A|r)))
              A & T([(k, r, -(i|A|k)) for k in range(r+1,A.n+1)])

           self.rank+=[r]

        super(self.__class__ ,self).__init__(A.lista)
```

This code is used in chunk 28.
Uses Matrix 5.

27b ⟨*sistema* 27b⟩≡

```
def homogenea(A):
     """ Devuelve una BlockMatriz con la solución del problema homogéneo """
     stack=Matrix(BlockMatrix([[A],[I(A.n)]]))
     soluc=Normal(stack)
     col=soluc.rank[A.m-1]
     return {A.m} | soluc | {col}
```

This code is used in chunk 28.
Uses BlockMatrix 15 and Matrix 5.

. . . creamos la librería `notacion.py`. . .

28 ⟨*notacion.py* 28⟩≡

```
# coding=utf8

from fractions import Fraction

⟨Métodos html y latex generales 30a⟩

def _repr_html_(self):
    return html(self.latex())

def latex_fraction(self):
    if self.denominator == 1:
         return repr(self.numerator)
    else:
         return "\\frac{"+repr(self.numerator)+"}{"+repr(self.denominator)+"}"

setattr(Fraction, '_repr_html_', _repr_html_)

setattr(Fraction, 'latex', latex_fraction)

def inverso(x):
    if x==1 or x == -1:
        return x
    else:
        y = 1/Fraction(x)
        if y.denominator == 1:
             return y.numerator
        else:
             return y

⟨Definición de la clase Vector 4⟩
⟨Definición de la clase Matrix 5⟩
⟨Definición de la clase BlockMatrix 15⟩
⟨Definición del método particion 17a⟩
⟨Definición del procedimiento de generación del conjunto clave para particionar 19b⟩
⟨Definición de vector nulo: VO 24b⟩
⟨Definición de matriz nula: MO 25a⟩
⟨Definición de vector fila o columna de la matriz identidad e 25b⟩
⟨Definición de la matriz identidad: I 26⟩
⟨Definición de la clase de transformaciones elementales: T 22⟩
⟨normal 27a⟩
⟨sistema 27b⟩
```

Root chunk (not used in this document).

8 Code chunks

⟨*Caso general de reparticion por columnas* 20a⟩ 18, 20a
⟨*Caso general de reparticion por filas* 20b⟩ 19a, 20b
⟨*Chunk de ejemplo que define la lista* a 2a⟩ 2a, 2c
⟨*Chunk final que indica qué tipo de objeto es* a *y hace unas sumas* 36e⟩ 2c, 36e
⟨*Creación del atributo* lista *cuando* sis *es un* Vector 31a⟩ 4, 31a

Fuera de las clase `Vector`, `Matrix`, etc. se definen dos métodos para la representación en Jupyter.

El primero, que llamaremos `html`, escribe el inicio y el final de un párrafo en html y en medio del párrafo escribirá la cadena TeX (que contendrá el código LaTeXde las expresiones matemáticas que queremos que se muestren en pantalla cuando usamos Jupyter Notebook que a su vez usa la librería de Java MathJax que interpreta código LaTeX).

El segundo, el método `latex`, convertirá en cadena de caracteres el input si éste es un número, y en caso contrario llamara al método latex de la clase desde la que se invocó a este método (es un truqui recursivo para que trate de manera parecida la expresiones en LaTeXy los tipos de datos que corresponden a números cuando se trata de escribir algo en LaTeX, así, si el componente de un vector es una fracción, el método `latex` general llamará el método `latex` de la clase fracción para representar la fracción —ello nos permitirá más adelante representar vectores o matrices con, por ejemplo, polinomios u otros objetos).

30a ⟨*Métodos html y latex generales* 30a⟩≡

```
def html(TeX):
    """ Plantilla HTML para insertar comandos LaTeX """
    return "<p style=\"text-align:center;\">$" + TeX + "$</p>"

def latex(a):
     if isinstance(a,float) | isinstance(a,int):
         return str(a)
     else:
         return a.latex()
```

This code is used in chunk 28.

Part II
Trozos de código secundarios

A Completando la clase `Vector`

30b ⟨*Texto de ayuda de la clase* `Vector` 30b⟩≡

```
""" Inicializa un vector a partir de distintos tipos de datos:

1) De una lista o tupla
>>> Vector([1,2,3])

Vector([1,2,3])

2) De otro vector (realiza una copia)
"""
```

This code is used in chunk 4.
Uses `Vector` 4.

A.1 Otras formas de instanciar un `Vector`

Si se ha instanciado el `Vector` con otro `Vector`, es decir, si `sis` es un `Vector` entonces se copia en `self.lista` la `lista` de dicho `Vector` (es decir, `sis.lista`). Dicho de otra forma, creamos una copia del vector.

Si el argumento no es correcto se informa con un error.

31a ⟨*Creación del atributo* `lista` *cuando* `sis` *es un* `Vector` 31a⟩≡

```
elif isinstance(sis, Vector):
    self.lista = sis.lista

else:
    raise ValueError('¡el argumento: debe ser una lista, tupla o Vector')
```

This code is used in chunk 4.
Uses `Vector` 4.

A.2 Operador selector para la clase `Vector`

31b ⟨*Texto de ayuda para el operador selector por la derecha para la clase* `Vector` 31b⟩≡

```
""" Extrae la i-esima componente de un vector por la derecha
>>> Vector([10,20,30]) | 2

20

o un sub-vector a partir de una lista o tupla de índices
>>> Vector([10,20,30]) | [2,3]
>>> Vector([10,20,30]) | (2,3)

Vector([20, 30])
"""
```

This code is used in chunk 7.
Uses `Vector` 4.

31c ⟨*Texto de ayuda para el operador selector por la izquierda para la clase* `Vector` 31c⟩≡

```
""" lo mismo que __or__ solo que por la izquierda
>>> 1 | Vector([10,20,30])

10

>>> [2,3] | Vector([10,20,30])
>>> (2,3) | Vector([10,20,30])

Vector([20, 30])
"""
```

This code is used in chunk 8a.
Uses `Vector` 4.

A.3 Representación de la clase `Vector`

Ahora necesitamos indicar a Python cómo representar los objetos de tipo `Vector`.

Los vectores, son secuencias finitas de números que representaremos con paréntesis, bien en forma de fila

$$\boldsymbol{v} = (v_1, \ldots, v_n)$$

o bien en forma de columna

$$\boldsymbol{v} = \begin{pmatrix} v_1 \\ \vdots \\ v_n \end{pmatrix}$$

Definimos tres representaciones distintas. Una para la línea de comandos de Python de manera que escriba `Vector` y a continuación encierre la representación de `self.lista` (el sistema de números), entre paréntesis. Por ejemplo, si la lista es `[a,b,c]`, Python nos mostrará en la linea de comandos: `Vector([a,b,c])`.

La representación en LaTeX encierra un vector (en forma de fila o de columna) entre paréntesis; y es usada a su vez por la representación html usada por el entorno Jupyter.

32a ⟨*Definición de la clase* `Vector` 4⟩+≡

```
def __repr__(self):
    """ Muestra el vector en su representación python """
    return 'Vector(' + repr(self.lista) + ')'

def _repr_html_(self):
    """ Construye la representación para el entorno jupyter notebook """
    return html(self.latex())

def latex(self):
    """ Construye el comando LaTeX """
    if self.rpr == 'fila':
        return '\\begin{pmatrix}' + \
               ',&'.join([latex(self|i)   for i in range(1,self.n+1)]) + \
               '\\end{pmatrix}'
    else:
        return '\\begin{pmatrix}' + \
               '\\\\'.join([latex(self|i) for i in range(1,self.n+1)]) + \
               '\\end{pmatrix}'
```

This code is used in chunk 28.

B Completando la clase `Matrix`

32b ⟨*Texto de ayuda de la clase* `Matrix` 32b⟩≡

```
""" Inicializa una matriz a partir de distintos tipos de datos:

1) De una lista de vectores (columnas)
>>> Matrix([Vector([1,2,3]),Vector([4,5,6])])

Matrix([Vector([1,2,3]),Vector([4,5,6])])

2) De una lista de listas de coeficientes (filas)
>>> Matrix([[1, 4], [2, 5], [3, 6]])
```

```
  Matrix([Vector([1,2,3]),Vector([4,5,6])])

  3) De una BlockMatrix (reune todas las matrices)

  4) De otra matriz (realiza una copia)
  """
```

This code is used in chunk 5.
Uses BlockMatrix 15, Matrix 5, and Vector 4.

B.1 Otras formas de instanciar una Matrix

Si se introduce una lista (tupla) de listas o tuplas, creamos una matriz fila a fila. Si se introduce una Matrix creamos una copia de la matriz. Si se introduce una BlockMatrix se elimina el particionado y que crea una única matriz. Si el argumento no es correcto se informa con un error.

33a ⟨*Creación del atributo* lista *cuando* sis *no es una lista o tupla de* Vector*es* 33a⟩≡

```
  if isinstance(sis, Matrix):
      self.lista  =  sis.lista

  elif isinstance(sis, BlockMatrix):
      self.lista  =  [Vector([ sis.lista[i][j]|k|s  \
                             for i in range(sis.m) for s in range(1,(sis.lm[i])+1) ])  \
                             for j in range(sis.n) for k in range(1,(sis.ln[j])+1) ]

  ⟨Verificación de que al instanciar Matrix el argumento sis es indexable 33b⟩

  elif isinstance(sis[0], (list, tuple)):
      ⟨Verificación de que todas las filas de la matriz tendrán la misma longitud 33c⟩
      self.lista  =  [ Vector([ sis[i][j] for i in range(len(sis   )) ]) \
                                          for j in range(len(sis[0])) ]
```

This code is used in chunk 5.
Uses BlockMatrix 15, Matrix 5, and Vector 4.

B.2 Códigos que verifican que los argumentos son correctos

33b ⟨*Verificación de que al instanciar* Matrix *el argumento* sis *es indexable* 33b⟩≡

```
  elif not isinstance(sis, (str, list, tuple)):
      raise ValueError('¡el argumento debe ser una lista o tupla de vectores una lista (o tupla) de listas
```

This code is used in chunk 33a.
Uses BlockMatrix 15 and Matrix 5.

33c ⟨*Verificación de que todas las filas de la matriz tendrán la misma longitud* 33c⟩≡

```
  it = iter(sis)
  the_len = len(next(it))
  if not all(len(l) == the_len for l in it):
      raise ValueError('no todas las listas (filas) tienen la misma longitud!')
```

This code is used in chunk 33a.

33d ⟨*Verificación de que todas las columnas de la matriz tendrán la misma longitud* 33d⟩≡

```
  it = iter(sis)
  the_len = len(next(it).lista)
  if not all(len(l.lista) == the_len for l in it):
      raise ValueError('no todos los vectores (columnas) tienen la misma longitud!')
```

This code is used in chunk 5.

B.3 Operador selector y transposición para la clase Matrix

34a ⟨*Texto de ayuda para el operador selector por la derecha para la clase* Matrix 34a⟩≡

```
"""  Extrae el i-ésimo vector columna de  una matriz
>>> Matrix([[1,2,3],[4,5,6]]) | 2

Vector([2, 5])

y también una matriz formada por una serie de vectores columna
>>> Matrix([[1,2,3],[4,5,6]]) | [2,3]

Matrix([[2, 3], [5, 6]])

o

>>> Matrix([[1,2,3],[4,5,6]]) | (2,3)
Matrix([[2, 3], [5, 6]])

y también particiona una matriz por columnas
>>> Matrix([[1,2,3],[4,5,6],[5,6,7]]) | {2}

BlockMatrix([[Matrix([[1, 2], [4, 5], [5, 6]]), Matrix([[3], [6], [7]])]])
"""
```

This definition is continued in chunk 34c.
This code is used in chunks 8b and 9b.
Uses BlockMatrix 15, Matrix 5, and Vector 4.

34b ⟨*Texto de ayuda para el operador transposición de la clase* Matrix 34b⟩≡

```
"""  Devuelve la matriz traspuesta
>>> ~Matrix([[1,2,3]])

Matrix([[1],[2],[3]])
"""
```

This code is used in chunk 9a.
Uses Matrix 5.

34c ⟨*Texto de ayuda para el operador selector por la derecha para la clase* Matrix 34a⟩+≡

```
"""  Extrae el i-ésimo vector fila de  una matriz
>>> 2 | Matrix([[1,2,3],[4,5,6],[5,6,7]])

Vector([4, 5, 6])

y también una sub-matriz a partir de una lista o tupla de índices de filas
>>> [2,3] | Matrix([[1,2,3],[4,5,6],[5,6,7]])
>>> (2,3) | Matrix([[1,2,3],[4,5,6],[5,6,7]])

Matrix([[4, 5, 6], [5, 6, 7]])

y también particiona una matriz por filas
>>> {2} | Matrix([[1,2,3],[4,5,6],[5,6,7]])

BlockMatrix([[Matrix([[1, 2, 3], [4, 5, 6]])], [Matrix([[5, 6, 7]])]])
"""
```

This code is used in chunks 8b and 9b.
Uses BlockMatrix 15, Matrix 5, and Vector 4.

B.4 Representación de la clase Matrix

Y como en el caso de los vectores, construimos los dos métodos de presentación. Una para la consola de comandos que escribe Matrix y entre paréntesis la lista de listas (es decir la lista de filas); y otra para el entorno Jupyter (que a su vez usa la representación LaTeX que representa las matrices entre corchetes como en las notas de clase)

35a ⟨*Definición de la clase* Matrix 5⟩+≡

```
def __repr__(self):
    """ Muestra una matriz en su representación python """
    return 'Matrix(' + repr(self.lista) + ')'

def _repr_html_(self):
    """ Construye la representación para el  entorno jupyter notebook """
    return html(self.latex())

def latex(self):
    """ Construye el comando LaTeX """
    return '\\begin{bmatrix}' + \
           '\\\\'.join(['&'.join([latex(i|self|j) for j in range(1,self.n+1) ]) \
                                                  for i in range(1,self.m+1) ]) + \
           '\\end{bmatrix}'
```

This code is used in chunk 28.

B.5 Operaciones con vectores y matrices

35b ⟨*Texto de ayuda para el operador suma en la clase* Vector 35b⟩≡

```
""" Suma de vectores
>>> Vector([10,20,30]) + Vector([0,1,1])

Vector([10,21,31])
"""
```

This code is used in chunk 10.
Uses Vector 4.

35c ⟨*Texto de ayuda para el operador producto por la derecha en la clase* Vector 35c⟩≡

```
""" Multiplica un vector por un número a su derecha
>>> Vector([10,20,30]) * 3

Vector([30,60,90])

o multiplica un vector por otro (producto escalar usual o producto punto)
>>> Vector([1, -1])*Vector([1, 1])

0
"""
```

This code is used in chunk 11b.
Uses Vector 4.

36a ⟨*Texto de ayuda para el operador producto por la izquierda en la clase* `Vector` 36a⟩≡

```
""" Multiplica un vector por un número a su izquierda
>>> 3 * Vector([10,20,30])

Vector([30,60,90])
"""
```

This code is used in chunk 11a.
Uses `Vector` 4.

36b ⟨*Texto de ayuda para el operador suma en la clase* `Matrix` 36b⟩≡

```
""" Suma de matrices
>>> Matrix([[10,20], [30,40]]) + Matrix([[1,2], [-30,4]])

Matrix([[11,22], [0,44]])
"""
```

This code is used in chunk 12b.
Uses `Matrix` 5.

36c ⟨*Texto de ayuda para el operador producto por la derecha en la clase* `Matrix` 36c⟩≡

```
""" Multiplica una matriz por un número a su derecha
>>> Matrix([[1,2],[3,4]]) * 10

Matrix([[10,20], [30,40]])
"""
```

This code is used in chunk 13.
Uses `Matrix` 5.

36d ⟨*Texto de ayuda para el operador producto por la izquierda en la clase* `Matrix` 36d⟩≡

```
""" Multiplica una matriz por un número a su izquierda
>>> 10 * Matrix([[1,2],[3,4]])

Matrix([[10,20], [30,40]])
"""
```

This code is used in chunk 14a.
Uses `Matrix` 5.

Último chunk del ejemplo de literate programming de la introducción

Este es uno de los trozos de código del ejemplo de la introducción.

36e ⟨*Chunk final que indica qué tipo de objeto es* `a` *y hace unas sumas* 36e⟩≡

```
type(a)
2+2
10*3+20
```

This code is used in chunk 2c.

El código completo del ejemplo usado para explicar cómo funciona el "Literate Programming" queda así:

```
a = ["Matemáticas II es mi asignatura preferida", "Cómo mola el Python", 1492, "Noweb"]
a[-1] = 10

for indice, item in enumerate(a, 1):
    print (indice, item)

type(a)
2+2
10*3+20
```

Tutorial previo en un notebook

Este Notebook es un vistazo sobre el **uso de nuestra librería para Mates 2**
O acceda a la url: https://mybinder.org/v2/gh/mbujosab/LibreriaDePythonParaMates2/master

C Librería completa

```
# coding=utf8

from fractions import Fraction

def html(TeX):
    """ Plantilla HTML para insertar comandos LaTeX """
    return "<p style=\"text-align:center;\">$" + TeX + "$</p>"

def latex(a):
     if isinstance(a,float) | isinstance(a,int):
         return str(a)
     else:
         return a.latex()


def _repr_html_(self):
    return html(self.latex())

def latex_fraction(self):
    if self.denominator == 1:
         return repr(self.numerator)
    else:
         return "\\frac{"+repr(self.numerator)+"}{"+repr(self.denominator)+"}"

setattr(Fraction, '_repr_html_', _repr_html_)

setattr(Fraction, 'latex', latex_fraction)

def inverso(x):
    if x==1 or x == -1:
        return x
    else:
        y = 1/Fraction(x)
```

```
        if y.denominator == 1:
             return y.numerator
        else:
             return y

class Vector:
    def __init__(self, sis, rpr='columna'):
        """ Inicializa un vector a partir de distintos tipos de datos:

        1) De una lista o tupla
        >>> Vector([1,2,3])

        Vector([1,2,3])

        2) De otro vector (realiza una copia)
        """


        if isinstance(sis, (list,tuple)):
            self.lista  =  list(sis)

        elif isinstance(sis, Vector):
            self.lista = sis.lista

        else:
            raise ValueError('¡el argumento: debe ser una lista, tupla o Vector')


        self.rpr  =  rpr
        self.n    =  len (self.lista)

    def __or__(self,i):
        """ Extrae la i-esima componente de un vector por la derecha
        >>> Vector([10,20,30]) | 2

        20

        o un sub-vector a partir de una lista o tupla de índices
        >>> Vector([10,20,30]) | [2,3]
        >>> Vector([10,20,30]) | (2,3)

        Vector([20, 30])
        """
        if isinstance(i,int):
            return self.lista[i-1]
        elif isinstance(i, (list,tuple) ):
            return Vector ([ (self|a) for a in i ])

    def __ror__(self,i):
        """ lo mismo que __or__ solo que por la izquierda
        >>> 1 | Vector([10,20,30])

        10

        >>> [2,3] | Vector([10,20,30])
        >>> (2,3) | Vector([10,20,30])
```

```
        Vector([20, 30])
        """
        return self | i

    def __add__(self, other):
        """ Suma de vectores
        >>> Vector([10,20,30]) + Vector([0,1,1])

        Vector([10,21,31])
        """
        if isinstance(other, Vector):
            if self.n == other.n:
                return Vector ([ (self|i) + (other|i) for i in range(1,self.n+1) ])
            else:
                print("error en la suma: vectores con distinto número de componentes")

    def __rmul__(self, x):
        """ Multiplica un vector por un número a su izquierda
        >>> 3 * Vector([10,20,30])

        Vector([30,60,90])
        """
        if isinstance(x, (int, float, Fraction)):
            return Vector ([ x*(self|i) for i in range(1,self.n+1) ])

        elif isinstance(x, Vector):
            if self.n == x.n:
                return sum([ (x|i)*(self|i) for i in range(1,self.n+1) ])
            else:
                print("error en producto: vectores con distinto número de componentes")

    def __mul__(self, x):
        """ Multiplica un vector por un número a su derecha
        >>> Vector([10,20,30]) * 3

        Vector([30,60,90])

        o multiplica un vector por otro (producto escalar usual o producto punto)
        >>> Vector([1, -1])*Vector([1, 1])

        0
        """
        if isinstance(x, (int, float, Fraction)):
            return x*self

        elif isinstance(x, Matrix):
            if self.n == x.m:
                return Vector( (~x)*self, rpr='fila')
            else:
                print("error en producto: Vector y Matrix incompatibles")

    def __eq__(self, other):
        return self.lista == other.lista
    def __repr__(self):
        """ Muestra el vector en su representación python """
        return 'Vector(' + repr(self.lista) + ')'
```

```
    def _repr_html_(self):
        """ Construye la representación para el entorno jupyter notebook """
        return html(self.latex())

    def latex(self):
        """ Construye el comando LaTeX """
        if self.rpr == 'fila':
            return '\\begin{pmatrix}' + \
                   ',&'.join([latex(self|i)   for i in range(1,self.n+1)]) + \
                   '\\end{pmatrix}'
        else:
            return '\\begin{pmatrix}' + \
                   '\\\\'.join([latex(self|i) for i in range(1,self.n+1)]) + \
                   '\\end{pmatrix}'

class Matrix:
    def __init__(self, sis):
        """ Inicializa una matriz a partir de distintos tipos de datos:

        1) De una lista de vectores (columnas)
        >>> Matrix([Vector([1,2,3]),Vector([4,5,6])])

        Matrix([Vector([1,2,3]),Vector([4,5,6])])

        2) De una lista de listas de coeficientes (filas)
        >>> Matrix([[1, 4], [2, 5], [3, 6]])

        Matrix([Vector([1,2,3]),Vector([4,5,6])])

        3) De una BlockMatrix (reune todas las matrices)

        4) De otra matriz (realiza una copia)
        """



        if isinstance(sis, Matrix):
            self.lista  =  sis.lista

        elif isinstance(sis, BlockMatrix):
            self.lista  =  [Vector([ sis.lista[i][j]|k|s  \
                                   for i in range(sis.m) for s in range(1,(sis.lm[i])+1) ])  \
                                   for j in range(sis.n) for k in range(1,(sis.ln[j])+1) ]

        elif not isinstance(sis, (str, list, tuple)):
            raise ValueError('¡el argumento debe ser una lista o tupla de vectores una lista (o tupla) de

        elif isinstance(sis[0], (list, tuple)):
            it = iter(sis)
            the_len = len(next(it))
            if not all(len(l) == the_len for l in it):
                raise ValueError('no todas las listas (filas) tienen la misma longitud!')

            self.lista  =  [ Vector([ sis[i][j] for i in range(len(sis   )) ]) \
                                                for j in range(len(sis[0])) ]

        elif isinstance(sis[0], Vector):
```

```
            it = iter(sis)
            the_len = len(next(it).lista)
            if not all(len(l.lista) == the_len for l in it):
                raise ValueError('no todos los vectores (columnas) tienen la misma longitud!')

            self.lista  =  list(sis)

        self.m  =  self.lista[0].n
        self.n  =  len(self.lista)

    def __or__(self,j):
        """ Extrae el i-ésimo vector columna de  una matriz
        >>> Matrix([[1,2,3],[4,5,6]]) | 2

        Vector([2, 5])

        y también una matriz formada por una serie de vectores columna
        >>> Matrix([[1,2,3],[4,5,6]]) | [2,3]

        Matrix([[2, 3], [5, 6]])

        o

        >>> Matrix([[1,2,3],[4,5,6]]) | (2,3)
        Matrix([[2, 3], [5, 6]])

        y también particiona una matriz por columnas
        >>> Matrix([[1,2,3],[4,5,6],[5,6,7]]) | {2}

        BlockMatrix([[Matrix([[1, 2], [4, 5], [5, 6]]), Matrix([[3], [6], [7]])]])
        """
        """ Extrae el i-ésimo vector fila de  una matriz
        >>> 2 | Matrix([[1,2,3],[4,5,6],[5,6,7]])

        Vector([4, 5, 6])

        y también una sub-matriz a partir de una lista o tupla de índices de filas
        >>> [2,3] | Matrix([[1,2,3],[4,5,6],[5,6,7]])
        >>> (2,3) | Matrix([[1,2,3],[4,5,6],[5,6,7]])

        Matrix([[4, 5, 6], [5, 6, 7]])

        y también particiona una matriz por filas
        >>> {2} | Matrix([[1,2,3],[4,5,6],[5,6,7]])

        BlockMatrix([[Matrix([[1, 2, 3], [4, 5, 6]])], [Matrix([[5, 6, 7]])]])
        """
        if isinstance(j,int):
            return self.lista[j-1]

        elif isinstance(j, (list,tuple)):
            return Matrix ([ self|a for a in j ])

        elif isinstance(j,set):
            return BlockMatrix ([ [self|a for a in particion(j,self.n)] ])

    def __invert__(self):
```

```
        """ Devuelve la matriz traspuesta
        >>> ~Matrix([[1,2,3]])

        Matrix([[1],[2],[3]])
        """
        return Matrix ([ (self|j).lista for j in range(1,self.n+1) ])

    def __ror__(self,i):
        """ Extrae el i-ésimo vector columna de  una matriz
        >>> Matrix([[1,2,3],[4,5,6]]) | 2

        Vector([2, 5])

        y también una matriz formada por una serie de vectores columna
        >>> Matrix([[1,2,3],[4,5,6]]) | [2,3]

        Matrix([[2, 3], [5, 6]])

        o

        >>> Matrix([[1,2,3],[4,5,6]]) | (2,3)
        Matrix([[2, 3], [5, 6]])

        y también particiona una matriz por columnas
        >>> Matrix([[1,2,3],[4,5,6],[5,6,7]]) | {2}

        BlockMatrix([[Matrix([[1, 2], [4, 5], [5, 6]]), Matrix([[3], [6], [7]])]])
        """
        """ Extrae el i-ésimo vector fila de  una matriz
        >>> 2 | Matrix([[1,2,3],[4,5,6],[5,6,7]])

        Vector([4, 5, 6])

        y también una sub-matriz a partir de una lista o tupla de índices de filas
        >>> [2,3] | Matrix([[1,2,3],[4,5,6],[5,6,7]])
        >>> (2,3) | Matrix([[1,2,3],[4,5,6],[5,6,7]])

        Matrix([[4, 5, 6], [5, 6, 7]])

        y también particiona una matriz por filas
        >>> {2} | Matrix([[1,2,3],[4,5,6],[5,6,7]])

        BlockMatrix([[Matrix([[1, 2, 3], [4, 5, 6]])], [Matrix([[5, 6, 7]])]])
        """
        if isinstance(i,int):
            return Vector ( (~self)|i , rpr='fila')

        elif isinstance(i, (list,tuple)):
            return Matrix ([ (a|self).lista  for a in i ])

        elif isinstance(i,set):
            return BlockMatrix ([ [a|self] for a in particion(i,self.m) ])

    def __add__(self, other):
        """ Suma de matrices
        >>> Matrix([[10,20], [30,40]]) + Matrix([[1,2], [-30,4]])
```

```
        Matrix([[11,22], [0,44]])
        """
        if isinstance(other,Matrix) and self.m == other.m and self.n == other.n:
            return Matrix ([ (self|i) + (other|i) for i in range(1,self.n+1) ])
        else:
            print("error en la suma: matrices con distinto orden")
    def __rmul__(self,x):
        """ Multiplica una matriz por un número a su derecha
        >>> Matrix([[1,2],[3,4]]) * 10

        Matrix([[10,20], [30,40]])
        """
        if isinstance(x, (int, float, Fraction)):
            return Matrix ([ x*(self|i) for i in range(1,self.n+1) ])
    def __mul__(self,x):
        """ Multiplica una matriz por un número a su izquierda
        >>> 10 * Matrix([[1,2],[3,4]])

        Matrix([[10,20], [30,40]])
        """
        if isinstance(x, (int, float, Fraction)):
            return x*self

        elif isinstance(x, Vector):
            if self.n == x.n:
                return sum( [(x|j)*(self|j) for j in range(1,self.n+1)], V0(self.m) )
            else:
                print("error en producto: vector y matriz incompatibles")

        elif isinstance(x, Matrix):
            if self.n == x.m:
                return Matrix( [ self*(x|j) for j in range(1,x.n+1)])
            else:
                print("error en producto: matrices incompatibles")

    def __eq__(self, other):
        return self.lista == other.lista
    def __and__(self,t):
        """ Aplica una o una secuencia de transformaciones elementales por columnas:
        >>>  A & T({1,3})                # intercambia las columnas 1 y 3
        >>>  A & T((1,5))                # multiplica la columna 1 por 5
        >>>  A & T((1,2,5))              # suma a la columna 1 la 2 por 5
        >>>  A & T([{1,3},(1,5),(1,2,5)]) # aplica la secuencia de transformaciones
        """
        if isinstance(t.t,set) and len(t.t) == 2:
            self.lista = Matrix( [(self|max(t.t)) if k==min(t.t) else \
                                  (self|min(t.t)) if k==max(t.t) else \
                                  (self|k) for k in range(1,self.n+1)]).lista

        elif isinstance(t.t,tuple) and len(t.t) == 2:
             self.lista = Matrix([ t.t[1]*(self|k) if k==t.t[0] else (self|k) \
                                   for k in range(1,self.n+1)] ).lista

        elif isinstance(t.t,tuple) and len(t.t) == 3:
             self.lista = Matrix([ (self|k) + t.t[2]*(self|t.t[1]) if k==t.t[0] else \
                                   (self|k) for k in range(1,self.n+1)] ).lista
        elif isinstance(t.t,list):
```

```
            for k in t.t:
                self & T(k)
        return self

    def __rand__(self,t):
        """ Aplica una o una secuencia de transformaciones elementales por filas:
        >>>    {1,3} & A              # intercambia las filas 1 y 3
        >>>    (1,5) & A              # multiplica la fila 1 por 5
        >>>  (1,2,5) & A              # suma a la fila 1 la 2 por 5

        >>>  [(1,2,5),(1,5),{1,3}] & A # aplica la secuencia de transformaciones
        """
        if isinstance(t.t,set) | isinstance(t.t,tuple):
            self.lista = (~(~self & t)).lista

        elif isinstance(t.t,list):
            for k in reversed(t.t):
                T(k) & self

        return self

    def __repr__(self):
        """ Muestra una matriz en su representación python """
        return 'Matrix(' + repr(self.lista) + ')'

    def _repr_html_(self):
        """ Construye la representación para el  entorno jupyter notebook """
        return html(self.latex())

    def latex(self):
        """ Construye el comando LaTeX """
        return '\\begin{bmatrix}' + \
               '\\\\'.join(['&'.join([latex(i|self|j) for j in range(1,self.n+1) ]) \
                                                      for i in range(1,self.m+1) ]) + \
               '\\end{bmatrix}'

class BlockMatrix:
    def __init__(self, lista):
        """ Inicializa una matriz por bloques usando una lista de listas de matrices.
        """
        self.lista = lista
        self.m     = len(lista)
        self.n     = len(lista[0])
        self.lm    = [fila[0].m for fila in lista]
        self.ln    = [c.n for c in lista[0]]

    def __repr__(self):
        """ Muestra una matriz en su representación python """
        return 'BlockMatrix(' + repr(self.lista) + ')'

    def _repr_html_(self):
        """ Construye la representación para el  entorno jupyter notebook """
        return html(self.latex())

    def latex(self):
        """ Escribe el código de LaTeX """
        if self.m == self.n == 1:
```

```
            return \
              '\\begin{array}{|c|}' + \
              '\\hline ' + \
              '\\\\ \\hline '.join( \
                    ['\\\\'.join( \
                    ['&'.join( \
                    [latex(self.lista[0][0]) ]) ]) ])  + \
              '\\\\ \\hline ' + \
              '\\end{array}'
        else:
            return \
              '\\left[' + \
              '\\begin{array}{' + '|'.join([n*'c' for n in self.ln])  + '}' + \
              '\\\\ \\hline '.join( \
                    ['\\\\'.join( \
                    ['&'.join( \
                    [latex(self.lista[i][j]|k|s) \
                    for j in range(self.n) for k in range(1,self.ln[j]+1) ]) \
                    for s in range(1,self.lm[i]+1) ]) for i in range(self.m) ])  + \
              '\\\\' + \
              '\\end{array}' + \
              '\\right]'

    def __or__(self,j):
        """ Reparticiona por columna una matriz por cajas """
        if isinstance(j,set):
            if self.n == 1:
                return BlockMatrix([ [ self.lista[i][0]|a  \
                                       for a in particion(j,self.lista[0][0].n)] \
                                       for i in range(self.m) ])

            elif self.n > 1:
                 return (key(self.lm) | Matrix(self)) | j

    def __ror__(self,i):
        """ Reparticiona por filas una matriz por cajas """
        if isinstance(i,set):
            if self.m == 1:
                return BlockMatrix([[ a|self.lista[0][j]  \
                                      for j in range(self.n) ] \
                                      for a in particion(i,self.lista[0][0].m)])

            elif self.m > 1:
                return i | (Matrix(self) | key(self.ln))


def particion(s,n):
    """ genera la lista de particionamiento a partir de un conjunto y un número
    >>> particion({1,3,5},7)

    [[1], [2, 3], [4, 5], [6, 7]]
    """
    p = list(s | set([0,n]))
    return [ list(range(p[k]+1,p[k+1]+1)) for k in range(len(p)-1) ]

def key(L):
    """ genera el conjunto clave a partir de una secuencia de tamaños
```

```
    número
    >>> key([1,2,1])

    {1, 3, 4}
    """
    return set([ sum(L[0:i]) for i in range(1,len(L)+1) ])

class V0(Vector):
    def __init__(self, n ,rpr = 'columna'):
        """ Inicializa el vector nulo de n componentes"""

        super(self.__class__ ,self).__init__([0 for i in range(n)],rpr)

class M0(Matrix):

    def __init__(self, m, n=None):
        """ Inicializa una matriz nula de orden n """
        if n is None:
            n = m

        super(self.__class__ ,self).__init__( \
                    [[0 for i in range(n)] for j in range(m)])

class e(Vector):

    def __init__(self, i,n ,rpr = 'columna'):
        """ Inicializa el vector e_i  de tamaño n """

        super(self.__class__ ,self).__init__([((i-1)==k)*1 for k in range(n)],rpr)

class I(Matrix):

    def __init__(self, n):
        """ Inicializa la matriz identidad de tamaño n """

        super(self.__class__ ,self).__init__(\
                    [[(i==j)*1 for i in range(n)] for j in range(n)])

class T:
    def __init__(self, t):
        """ Inicializa una transformación elemental """
        self.t = t

    def __and__(self,t):
        """ Crea una trasformación composición de dos
        >>> T((1,2)) & T({2,4})

        T([(1,2), {2,4}])

        O aplica la transformación sobre una matriz A
        >>> A & T({1,2})    (intercambia las dos primeras columnas de A)
        """
        def CreaLista(a):
            """Transforma una una tupla en una lista que contiene la tupla"""
            return (a if isinstance(a,list) else [a])

        if isinstance(t,T):
```

```
            return T(CreaLista(self.t) + CreaLista(t.t))

        if isinstance(t,Matrix):
            return t.__rand__(self)

class Normal(Matrix):
    def __init__(self, data):
        """ Escalona por Gauss obteniendo una matriz cuyos pivotes son unos """
        def pivote(v,k):
            """ Devuelve el primer índice mayor que k de de un
            un coeficiente no nulo del vector v. En caso de no existir
            devuelve 0
            """
            return ([x[0] for x in enumerate(v.lista, 1) \
                                if (x[1] !=0 and x[0] > k)]+[0])[0]

        A = Matrix(data)
        r = 0
        self.rank = []
        for i in range(1,A.n+1):
           p = pivote((i|A),r)
           if p > 0:
              r += 1
              A & T({p,r})
              A & T((r,inverso(i|A|r)))
              A & T([(k, r, -(i|A|k)) for k in range(r+1,A.n+1)])

           self.rank+=[r]

        super(self.__class__ ,self).__init__(A.lista)

def homogenea(A):
     """ Devuelve una BlockMatriz con la solución del problema homogéneo """
     stack=Matrix(BlockMatrix([[A],[I(A.n)]]))
     soluc=Normal(stack)
     col=soluc.rank[A.m-1]
     return {A.m} | soluc | {col}
```