

Librería en Python para Matemáticas II (Álgebra Lineal)

<https://github.com/mbujosab/LibreriaDePythonParaMates2>

Marcos Bujosa

August 20, 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 final¹ (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 “ \top ”, por ello, para denotar la transposición nos veremos forzados a usar el operador `__invert__`, es decir, la tilde “ \sim ” de Python, que además deberemos colocar a la izquierda de la matriz que transponemos).

Mates II	Python
\mathbf{A}^\top	$\sim \mathbf{A}$

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
$\mathbf{v}_{ i}$	$\mathbf{v} i$
${}_i \mathbf{v}$	$i \mathbf{v}$
$\mathbf{A}_{ j}$	$\mathbf{A} j$
${}_i \mathbf{A}$	$i \mathbf{A}$

¹aquellas 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 usando código L^AT_EX, 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 actúa sobre la matriz.

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

Tutorial previo en un notebook

Antes de seguir, repase el Notebook “**Listas y tuplas**” en la carpeta “TutorialPython” en <https://mybinder.org/v2/gh/mbujosab/LibreriaDePythonParaMates2/master>

Literate programming 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 \langle Chunk de ejemplo que define la lista a 2a $\rangle \equiv$

```
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 \langle Segundo chunk de ejemplo que cambia el último elemento de la lista a 2b $\rangle \equiv$

```
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 \langle EjemploLiterateProgramming.py 2c $\rangle \equiv$

\langle Chunk de ejemplo que define la lista a 2a \rangle

\langle Segundo chunk de ejemplo que cambia el último elemento de la lista a 2b \rangle

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

\langle Chunk final que indica qué tipo de objeto es a y hace unas sumas 37 \rangle

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 código. Por ejemplo, el último chunk de este ejemplo se encuentra en la Página 37 de este documento (y justo detrás podrá ver cómo queda el código completo).

Con ánimo de que la 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

$$\mathbf{v} = (v_1, \dots, v_n)$$

o bien en forma de columna

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

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 la carpeta “TutorialPython” en <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).

²también 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 26b>

            if isinstance(sis, (list,tuple)):
                self.lista = list(sis)
            <Creación del atributo lista cuando sis es un Vector 27a>

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

```

This definition is continued in chunk 6a.

This code is used in chunk 23b.

Defines:

Vector, used in chunks 5, 7, 9–11, 14a, and 26–34.

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

$$\mathbf{A} = [v_1, \dots, v_m]$$

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 **Vectores** 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 **Vectores***).

El atributo `self.lista` guarda una lista de **Vectores**, (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** es una “una lista de filas”), o *si el primer elemento de **sis** es un **Vector** (entonces la lista es `list(sis)`, la lista de **Vectores** 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 28b>

            <Creación del atributo lista cuando sis no es una lista o tupla de Vectores 29a>

            elif isinstance(sis[0], Vector):
                <Verificación de que todas las columnas de la matriz tendrán la misma longitud 29d>
                self.lista = list(sis)

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

This definition is continued in chunk 6b.
This code is used in chunk 23b.
Defines:
    Matrix, used in chunks 8, 9, 11–14, 16, 17, 22–24, 28–30, and 32–34.
Uses Vector 4.

```

Resumen

Los **vectores** son listas ordenadas de números. La clase **Vector** almacena una lista ordenada 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 ordenadas de vectores (dichos vectores son sus columnas). La clase **Matrix** almacena una lista ordenada de **Vectores** 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 **Vectores**, 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 **Vectores** 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 **Vectores** 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 **Vectores** (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 \langle Definición de la clase **Vector** 4 $\rangle + \equiv$
 \langle Operador selector por la derecha para la clase **Vector** 7 \rangle
 \langle Operador selector por la izquierda para la clase **Vector** 8a \rangle
 \langle Suma de **Vectores** 10 \rangle
 \langle Producto de un **Vector** por un escalar a su izquierda, o por otro **Vector** a su izquierda 11a \rangle
 \langle Producto de un **Vector** por un escalar a su derecha, o por una **Matrix** a su derecha 11b \rangle
 \langle Definición de la igualdad entre **Vectores** 12a \rangle
 \langle Métodos de representación de la clase **Vector** 28a \rangle

This code is used in chunk 23b.

6b \langle Definición de la clase **Matrix** 5 $\rangle + \equiv$
 \langle Operador selector por la derecha para la clase **Matrix** 8b \rangle
 \langle Operador transposición para la clase **Matrix** 9a \rangle
 \langle Operador selector por la izquierda para la clase **Matrix** 9b \rangle
 \langle Suma de **Matrix** 12b \rangle
 \langle Producto de una **Matrix** por un escalar a su izquierda 13 \rangle
 \langle Producto de una **Matrix** por un escalar, un vector o una matriz a su derecha 14a \rangle
 \langle Definición de la igualdad entre dos **Matrix** 14b \rangle
 \langle Transformaciones elementales de las columnas de la clase **Matrix** 17 \rangle
 \langle Transformaciones elementales de las filas de la clase **Matrix** 18 \rangle
 \langle Métodos de representación de la clase **Matrix** 31a \rangle

This code is used in chunk 23b.

2 Operador selector

Notación en Mates 2

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

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

Mates II	Python
$\mathbf{v} _i$	<code>v i</code>
${}_i \mathbf{v}$	<code>i v</code>
$\mathbf{A} _j$	<code>A j</code>
${}_i \mathbf{A}$	<code>i A</code>

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

- $(i_1, \dots, i_r)|\mathbf{v} = (\mathbf{v}_{i_1}, \dots, \mathbf{v}_{i_r}) = \mathbf{v}|_{(i_1, \dots, i_r)}$ (es un vector formado por elementos de \mathbf{v})
- $(i_1, \dots, i_r)|\mathbf{A} = \begin{bmatrix} {}_{i_1}|\mathbf{A}, \dots, {}_{i_r}|\mathbf{A} \end{bmatrix}^\top$ (es una matriz cuyas filas son filas de \mathbf{A})
- $\mathbf{A}|_{(j_1, \dots, j_r)} = \begin{bmatrix} \mathbf{A}|_{j_1}, \dots, \mathbf{A}|_{j_r} \end{bmatrix}$ (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 seleccionadas).

```

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 27b>
        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 27c>
        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 `Vectores`, 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 30a>
        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 21a>
```

This code is used in chunk 6b.

Uses `Matrix` 5.

Operador transposición. Como paso intermedio vamos a definir el operador transposición, 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 transposición. Basta instanciar `Matrix` con la lista de los n atributos “`lista`” correspondientes a los consecutivos n `Vectores` columna. (Recuerde también que `range(1,self.m+1)` recorre los números: $1, 2, \dots, m$).

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

Mates II	Python
\mathbf{A}^\top	<code>~A</code>

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 30b>
        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!.

$(\sim self)|j$

(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 30a>
        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 20b>
```

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 Python (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

$$(a + b)_{|i} = (a)_{|i} + (b)_{|i} \quad \text{para } i = 1, \dots, n.$$

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 Vectores 10>≡
    def __add__(self, other):
        <Texto de ayuda para el operador suma en la clase Vector 31b>
        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 a por un escalar x a su izquierda como

$$(xa)_{|i} = x(a_{|i}) \quad \text{para } i = 1, \dots, n.$$

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 x y a en \mathbb{R}^n es

$$x \cdot a = \sum_{i=1}^n x_i a_i \quad \text{para } i = 1, \dots, n.$$

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 32a>
          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,

$$bx = xb$$

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 **a** de \mathbb{R}^n por una matriz **B** con *n* filas es

$$aB = B^T a$$

cuya transcripción será

$$(\sim x) * self$$

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 31c>
          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):
    """a==b es True si a.lista es igual que b.lista. False en caso contrario"""
    return self.lista == other.lista
This code is used in chunk 6a.
```

Matrices

Suma de matrices Hemos definido la suma de matrices como

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

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 32b>
    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 **A** por un escalar x a su izquierda como

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

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 32c>
        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,

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

cuya transcripción será

```
x * self
```

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

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

$$\mathbf{Ax} = \sum_{j=1}^n x_j \mathbf{A}_{|j} \quad \text{para } j = 1, \dots, n.$$

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 $\mathbf{A}_{m \times k}$ por otra matrix $\mathbf{X}_{k \times n}$ a su derecha se define como

$$(\mathbf{AX})_{|j} = \mathbf{A}(\mathbf{X}_{|j}) \quad \text{para } j = 1, \dots, n.$$

cuya transcripción será

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

donde `self` es la matrix y `x` es otra matrix (`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 32d>
          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):
          """A==B es True si A.lista es igual que B.lista. False en caso contrario"""
          return self.lista == other.lista

```

This code is used in chunk 6b.

4 Transformaciones elementales

Notación en Mates 2

Si \mathbf{A} es una matriz, consideramos las siguientes transformaciones:

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

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

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

Como una transformación 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 completamente³ una matriz elemental basta indicar su orden n y el coeficiente que no coincide con los de \mathbf{I}_n .
2. Las transformaciones elementales, indicando dicho coeficiente, omiten el orden n .

Escogemos la siguiente traducción de esta notación:

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

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 en la última posición y multiplica a la columna (fila) con el índice que le precede.

Además vamos a extender esta notación para expresar las secuencias de transformaciones elementales $\mathbf{A} \xrightarrow{\tau_k \cdots \tau_1} \mathbf{A}$ y $\mathbf{A}_{\tau_1 \cdots \tau_k}$ con una lista de transformaciones elementales, de manera que logremos las siguientes equivalencias entre expresiones:

$$\begin{aligned}
 t_k \& \cdots \& t_2 \& t_1 \& \mathbf{A} &= [t_1, t_2, \dots, t_k] \& \mathbf{A} \\
 \mathbf{A} \& t_1 \& t_2 \& \cdots \& t_k &= \mathbf{A} \& [t_1, t_2, \dots, t_k]
 \end{aligned}$$

Para ello, vamos a definir un nuevo tipo de objeto: \mathbf{T} (transformación elemental) que nos permitirá encadenar transformaciones elementales.

³Fíjese que la notación usada en las notas de clase para las matrices elementales \mathbf{E} , no las describe completamente, pues se deja al lector la deducción de cuál es su orden (el adecuado para poder realizar el producto $\mathbf{A}\mathbf{E}$ o $\mathbf{E}\mathbf{A}$)

```

16  <Definición de la clase T (Transformación Elemental) 16>≡
    class T:
        def __init__(self, t):
            """ Inicializa una transformación elemental """
            self.t = t

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

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

            0 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 23b.
 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):


```

17  <Transformaciones elementales de las columnas de la clase Matrix 17>≡
    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 6b.
Uses Matrix 5.

Observación 1. Las transformaciones elementales modifican la matriz.

Y para transformaciones por filas usamos el truco de aplicar las operaciones sobre la filas de la transpuesta y de nuevo transponer 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 (con la función `reversed`):

```

18  <Transformaciones elementales de las filas de la clase Matrix 18>≡
    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 6b.

Observación 2. Las trasformaciones elementales modifican la matriz.

5 Matrices particionadas (o matrices por bloques)

Esta sección no es muy importante para seguir el curso, aunque si es importante para el funcionamiento de la librería. Piense que cuando invierte una matriz o resuelve un sistema de ecuaciones, usa una matriz particionada (con dos bloques: una matriz arriba, y la matriz identidad con idéntico número de columnas debajo). Como esta librería replica lo que se ve en clase, es necesario definir las matrices particionadas.

Si quiere, puede saltarse inicialmente esta sección: el modo de particionar una matriz es sencillo y se puede aprender rápidamente con el siguiente Notebook

Tutorial previo en un notebook

Este Notebook es un vistazo sobre el uso de nuestra librería para Mates 2 en la carpeta “TutorialPython” en

<https://mybinder.org/v2/gh/mbujosab/LibreriaDePythonParaMates2/master>

Las matrices por bloques o cajas \boxed{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 `sis` es una lista (o tupla) de listas de matrices, cada una de las listas de matrices es una fila de bloques (o submatrices con el mismo número de filas).

19 \langle Definición de la clase `BlockMatrix` 19 $\rangle \equiv$

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

\langle Repartición de las columnas de una `BlockMatrix` 21b \rangle
 \langle Repartición de las filas de una `BlockMatrix` 22a \rangle
 \langle Métodos de representación de la clase `BlockMatrix` 35 \rangle

This code is used in chunk 23b.

Defines:

`BlockMatrix`, used in chunks 20–22, 24b, and 28–30.

El atributo `self.m` contiene el número de filas (de bloques o submatrices) y `self.n` contiene el número de columnas (de bloques o submatrices). Añadimos el atributo `self.ln`, que es una lista con el número de filas que tienen las submatrices de cada fila, y `self.lm` con el número de columnas de las submatrices de cada columna.

5.1 Particionado de matrices

Vamos a completar las capacidades de los operadores “`i|`” y “`|j`” sobre matrices. Hasta ahora, si los argumentos `i` o `j` eran *enteros* (`int`), se seleccionaba una fila o una columna respectivamente; y si los argumentos `i` o `j` eran *listas o tuplas* de índices, se generaba una submatriz con las filas o las columnas indicadas.

Aquí, si los argumentos `i` o `j` son conjuntos de enteros, asumimos que dicho números enteros indican las filas o columnas por las que se debe particionar una `Matrix` según el siguiente cuadro explicativo:

Notación en Mates 2

- Si $n \leq m \in \mathbb{N}$ denotaremos con $(n : m)$ a la secuencia $n, n + 1, \dots, m$, (es decir, a la lista ordenada de los números de $\{k \in \mathbb{N} | n \leq k \leq m\}$).
- Si $j_1, \dots, j_s \in \mathbb{N}$ con $j_1 < \dots < j_s \leq m$ donde m es el número de columnas de \mathbf{A} , entonces $\mathbf{A}_{\{j_1, \dots, j_s\}}$ es la matriz de bloques^a

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

- Si $i_1, \dots, i_r \in \mathbb{N}$ con $i_1 < \dots < i_r \leq n$ donde n es el número de filas de \mathbf{A} , entonces $\mathbf{A}_{\{i_1, \dots, i_r\}}$ es la matriz de bloques

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

^aFalta incluir esta notación en las notas de clase

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

```
20a  <Definición del método particion 20a>≡
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 23b.

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

```
20b  <Partición de una matriz por filas de bloques 20b>≡
elif isinstance(i,set):
    return BlockMatrix ([ [a|self] for a in particion(i,self.m) ])
```

This code is used in chunk 9b.

Uses BlockMatrix 19.

21a \langle Partición de una matriz por columnas de bloques 21a $\rangle \equiv$

```

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

```

This code is used in chunk 8b.
Uses BlockMatrix 19.

Pero aún nos falta algo:

Notación en Mates 2

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

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

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

$$\{i_1, \dots, i_r\} | \left(\mathbf{A} | \{j_1, \dots, j_s\} \right) \quad \text{y} \quad \left(\{i_1, \dots, i_r\} | \mathbf{A} \right) | \{j_1, \dots, j_s\}$$

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.

21b \langle Repartición de las columnas de una BlockMatrix 21b $\rangle \equiv$

```

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) ])

     $\langle$ Caso general de reparticion por columnas 22c $\rangle$ 

```

This code is used in chunk 19.
Uses BlockMatrix 19.

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

```

22a  <Repartición de las filas de una BlockMatrix 22a>≡
      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 23a>

```

This code is used in chunk 19.
Uses BlockMatrix 19.

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. Así:

$$\begin{aligned}
 \langle i'_1, \dots, i'_r | \left(\langle i_1, \dots, i_k | \mathbf{A} | \langle j_1, \dots, j_s \rangle \right) &= \langle i'_1, \dots, i'_r | \mathbf{A} | \langle j_1, \dots, j_s \rangle \\
 \left(\langle i_1, \dots, i_k | \mathbf{A} | \langle j_1, \dots, j_s \rangle \right) | \langle j'_1, \dots, j'_r \rangle &= \langle i_1, \dots, i_k | \mathbf{A} | \langle j'_1, \dots, j'_r \rangle
 \end{aligned}$$

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

```

22b  <Definición del procedimiento de generación del conjunto clave para particionar 22b>≡
      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 23b.

Así, los casos generales consisten en reparticionar de nuevo:

```

22c  <Caso general de reparticion por columnas 22c>≡
      elif self.n > 1:
          return (key(self.lm) | Matrix(self)) | j

```

This code is used in chunk 21b.
Uses Matrix 5.

```

23a  <Caso general de reparticion por filas 23a>≡
      elif self.m > 1:
          return i | (Matrix(self) | key(self.ln))

```

This code is used in chunk 22a.
Uses **Matrix** 5.

Observación 3. 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 $\{1,2,3\} | A$.

6 Librería completa

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

```

23b  <notacion.py 23b>≡
      # coding=utf8

      <Copyright y licencia GPL 36>

      from fractions import Fraction

      <Métodos html y latex generales 25a>
      <Métodos html y latex para fracciones 25b>

      <Definición de inverso 26a>

      <Definición de la clase Vector 4>
      <Definición de la clase Matrix 5>
      <Definición de la clase T (Transformación Elemental) 16>
      <Definición de la clase BlockMatrix 19>

      <Definición del método particion 20a>
      <Definición del procedimiento de generación del conjunto clave para particionar 22b>

      <Definición de vector nulo: VO 33a>
      <Definición de matriz nula: MO 33b>
      <Definición de la matriz identidad: I 34b>
      <Definición de vector fila o columna de la matriz identidad e 34a>

      <normal 24a>
      <sistema 24b>

```

Root chunk (not used in this document).

7 Ejemplo de uso

Con este código ya podemos hacer muchísimas cosas. Por ejemplo, eliminación gaussiana para encontrar el espacio nulo de una matriz!

```
24a <normal 24a>≡
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 23b.
Uses Matrix 5.

```
24b <sistema 24b>≡
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 23b.
Uses BlockMatrix 19 and Matrix 5.

Tutorial previo en un notebook

Este Notebook es un vistazo sobre el **uso de nuestra librería para Mates 2** en la carpeta
"TutorialPython" en

<https://mybinder.org/v2/gh/mbujosab/LibreriaDePythonParaMates2/master>

Part II

Trozos de código secundarios

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

El método `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 \LaTeX de 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 método `latex`, convertirá en cadena de caracteres el input si éste es un número, y en caso contrario llamará al método `latex` de la clase desde la que se invocó a este método (es un truco recursivo para que trate de manera parecida las expresiones en \LaTeX y 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).

```
25a <Métodos html y latex generales 25a>≡
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 23b.

```
25b <Métodos html y latex para fracciones 25b>≡
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)
```

This code is used in chunk 23b.

```
26a  <Definición de inverso 26a>≡
      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

      This code is used in chunk 23b.
```

A Completando la clase Vector

```
26b  <Texto de ayuda de la clase Vector 26b>≡
      """ 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 `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.

```
27a <Creación del atributo lista cuando sis es un Vector 27a>≡
    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

A.2.1 Textos de ayuda

```
27b <Texto de ayuda para el operador selector por la derecha para la clase Vector 27b>≡
    """ 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.

```
27c <Texto de ayuda para el operador selector por la izquierda para la clase Vector 27c>≡
    """ 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

$$\mathbf{v} = (v_1, \dots, v_n)$$

o bien en forma de columna

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

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 línea 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.

```
28a <Métodos de representación de la clase Vector 28a>≡
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 6a.

B Completando la clase Matrix

B.1 Textos de ayuda

```
28b <Texto de ayuda de la clase Matrix 28b>≡
""" 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` 19, `Matrix` 5, and `Vector` 4.

B.2 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.

29a *⟨Creación del atributo lista cuando sis no es una lista o tupla de Vectores 29a⟩≡*

```
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 29b⟩

```
elif isinstance(sis[0], (list, tuple)):
    ⟨Verificación de que todas las filas de la matriz tendrán la misma longitud 29c⟩
    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` 19, `Matrix` 5, and `Vector` 4.

B.3 Códigos que verifican que los argumentos son correctos

29b *⟨Verificación de que al instanciar Matrix el argumento sis es indexable 29b⟩≡*

```
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 29a.

Uses `BlockMatrix` 19 and `Matrix` 5.

29c *⟨Verificación de que todas las filas de la matriz tendrán la misma longitud 29c⟩≡*

```
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 29a.

29d *⟨Verificación de que todas las columnas de la matriz tendrán la misma longitud 29d⟩≡*

```
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.4 Operador selector y transposición para la clase Matrix

```

30a <Texto de ayuda para el operador selector por la derecha para la clase Matrix 30a>≡
    """ 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 30c.
This code is used in chunks 8b and 9b.
Uses BlockMatrix 19, Matrix 5, and Vector 4.

30b <Texto de ayuda para el operador transposición de la clase Matrix 30b>≡
    """ Devuelve la matriz traspuesta
    >>> ~Matrix([[1,2,3]])

    Matrix([[1], [2], [3]])
    """
This code is used in chunk 9a.
Uses Matrix 5.

30c <Texto de ayuda para el operador selector por la derecha para la clase Matrix 30a>+≡
    """ 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 19, Matrix 5, and Vector 4.

```

B.5 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 $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ que representa las matrices entre corchetes como en las notas de clase)

```
31a  <Métodos de representación de la clase Matrix 31a>≡
      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 6b.
```

B.6 Operaciones con vectores y matrices

B.6.1 Textos de ayuda

```
31b  <Texto de ayuda para el operador suma en la clase Vector 31b>≡
      """ 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`.

```
31c  <Texto de ayuda para el operador producto por la derecha en la clase Vector 31c>≡
      """ 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`.

32a \langle *Texto de ayuda para el operador producto por la izquierda en la clase Vector 32a* $\rangle \equiv$

```

""" 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.

32b \langle *Texto de ayuda para el operador suma en la clase Matrix 32b* $\rangle \equiv$

```

""" 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.

32c \langle *Texto de ayuda para el operador producto por la derecha en la clase Matrix 32c* $\rangle \equiv$

```

""" 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.

32d \langle *Texto de ayuda para el operador producto por la izquierda en la clase Matrix 32d* $\rangle \equiv$

```

""" 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.

C Vectores y Matrices especiales

Notación en Mates 2

Los vectores cero $\mathbf{0}$ y las matrices cero $\mathbf{0}$ 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):

```
33a  <Definición de vector nulo: V0 33a>≡
      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 23b.
Uses `Vector` 4.

Y lo mismo hacemos para matrices

```
33b  <Definición de matriz nula: M0 33b>≡
      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 23b.
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 $i|_j = \begin{cases} 1 & \text{si } j = i \\ 0 & \text{si } j \neq i \end{cases}$.

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

```

34a  <Definición de vector fila o columna de la matriz identidad e 34a>≡
      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 23b.
Uses Vector 4.

```

Lo importante es la matriz identidad.

```

34b  <Definición de la matriz identidad: I 34b>≡
      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 23b.
Uses Matrix 5.

```

C.1 Representación de la clase BlockMatrix

A continuación definimos las reglas de representación para las matrices por bloques. `Matrix` y `BlockMatrix` son objetos distintos. Los bloques se separan con líneas 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.

```

35  <Métodos de representación de la clase BlockMatrix 35>≡
    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 19.

D Code chunks

- <Caso general de reparticion por columnas 22c> 21b, [22c](#)
- <Caso general de reparticion por filas 23a> 22a, [23a](#)
- <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 37> [2c](#), [37](#)
- <Copyright y licencia GPL 36> [23b](#), [36](#)
- <Creación del atributo lista cuando sis es un Vector 27a> [4](#), [27a](#)
- <Creación del atributo lista cuando sis no es una lista o tupla de Vectores 29a> [5](#), [29a](#)
- <Definición de inverso 26a> [23b](#), [26a](#)
- <Definición de la clase BlockMatrix 19> [19](#), [23b](#)
- <Definición de la clase Matrix 5> [5](#), [6b](#), [23b](#)
- <Definición de la clase T (Transformación Elemental) 16> [16](#), [23b](#)
- <Definición de la clase Vector 4> [4](#), [6a](#), [23b](#)
- <Definición de la igualdad entre Vectores 12a> [6a](#), [12a](#)

<Definición de la igualdad entre dos **Matrix** 14b> 6b, [14b](#)
 <Definición de la matriz identidad: **I** 34b> 23b, [34b](#)
 <Definición de matriz nula: **MO** 33b> 23b, [33b](#)
 <Definición de vector fila o columna de la matriz identidad **e** 34a> 23b, [34a](#)
 <Definición de vector nulo: **V0** 33a> 23b, [33a](#)
 <Definición del método **particion** 20a> [20a](#), 23b
 <Definición del procedimiento de generación del conjunto clave para particionar 22b> [22b](#), 23b
 <Ejemplo `LiterateProgramming.py` 2c> [2c](#)
 <Métodos de representación de la clase **BlockMatrix** 35> 19, [35](#)
 <Métodos de representación de la clase **Matrix** 31a> 6b, [31a](#)
 <Métodos de representación de la clase **Vector** 28a> 6a, [28a](#)
 <Métodos `html` y `latex` generales 25a> 23b, [25a](#)
 <Métodos `html` y `latex` para fracciones 25b> 23b, [25b](#)
 <`normal` 24a> 23b, [24a](#)
 <`notacion.py` 23b> [23b](#)
 <Operador selector por la derecha para la clase **Matrix** 8b> 6b, [8b](#)
 <Operador selector por la derecha para la clase **Vector** 7> 6a, [7](#)
 <Operador selector por la izquierda para la clase **Matrix** 9b> 6b, [9b](#)
 <Operador selector por la izquierda para la clase **Vector** 8a> 6a, [8a](#)
 <Operador transposición para la clase **Matrix** 9a> 6b, [9a](#)
 <Partición de una matriz por columnas de bloques 21a> 8b, [21a](#)
 <Partición de una matriz por filas de bloques 20b> 9b, [20b](#)
 <Producto de un **Vector** por un escalar a su derecha, o por una **Matrix** a su derecha 11b> 6a, [11b](#)
 <Producto de un **Vector** por un escalar a su izquierda, o por otro **Vector** a su izquierda 11a> 6a, [11a](#)
 <Producto de una **Matrix** por un escalar a su izquierda 13> 6b, [13](#)
 <Producto de una **Matrix** por un escalar, un vector o una matriz a su derecha 14a> 6b, [14a](#)
 <Repartición de las columnas de una **BlockMatrix** 21b> 19, [21b](#)
 <Repartición de las filas de una **BlockMatrix** 22a> 19, [22a](#)
 <Segundo chunk de ejemplo que cambia el último elemento de la lista **a** 2b> [2b](#), 2c
 <`sistema` 24b> 23b, [24b](#)
 <Suma de **Matrix** 12b> 6b, [12b](#)
 <Suma de **Vectores** 10> 6a, [10](#)
 <Texto de ayuda de la clase **Matrix** 28b> 5, [28b](#)
 <Texto de ayuda de la clase **Vector** 26b> 4, [26b](#)
 <Texto de ayuda para el operador producto por la derecha en la clase **Matrix** 32c> 13, [32c](#)
 <Texto de ayuda para el operador producto por la derecha en la clase **Vector** 31c> 11b, [31c](#)
 <Texto de ayuda para el operador producto por la izquierda en la clase **Matrix** 32d> 14a, [32d](#)
 <Texto de ayuda para el operador producto por la izquierda en la clase **Vector** 32a> 11a, [32a](#)
 <Texto de ayuda para el operador selector por la derecha para la clase **Matrix** 30a> 8b, 9b, [30a](#), [30c](#)
 <Texto de ayuda para el operador selector por la derecha para la clase **Vector** 27b> 7, [27b](#)
 <Texto de ayuda para el operador selector por la izquierda para la clase **Vector** 27c> 8a, [27c](#)
 <Texto de ayuda para el operador suma en la clase **Matrix** 32b> 12b, [32b](#)
 <Texto de ayuda para el operador suma en la clase **Vector** 31b> 10, [31b](#)
 <Texto de ayuda para el operador transposición de la clase **Matrix** 30b> 9a, [30b](#)
 <Transformaciones elementales de las columnas de la clase **Matrix** 17> 6b, [17](#)
 <Transformaciones elementales de las filas de la clase **Matrix** 18> 6b, [18](#)
 <Verificación de que al instanciar **Matrix** el argumento **sis** es indexable 29b> 29a, [29b](#)
 <Verificación de que todas las columnas de la matriz tendrán la misma longitud 29d> 5, [29d](#)
 <Verificación de que todas las filas de la matriz tendrán la misma longitud 29c> 29a, [29c](#)

Chunk de Licencia

36 <Copyright y licencia `GPL` 36>≡
 # Copyright (C) 2019 Marc Cor Bujosa

```

# This program is free software: you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, either version 3 of the License, or

```

```
# (at your option) any later version.

# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
# GNU General Public License for more details.

# You should have received a copy of the GNU General Public License
# along with this program. If not, see <https://www.gnu.org/licenses/
```

This code is used in chunk 23b.

Último chunk del ejemplo de literate programming de la introducción

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

37 *<Chunk final que indica qué tipo de objeto es a y hace unas sumas 37>*≡

```
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
```