Librería en Python para Matemáticas II (Álgebra Lineal)

https://github.com/mbujosab/LibreriaDePythonParaMates2

Marcos Bujosa

October 4, 2019

Contents

Declaración de intenciones

Uno de los objetivos que me he propuesto para el curso Matemáticas II (Álgebra Lineal) 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.

Una librería de Python específica para la asignatura

Aunque Python tiene librerías que permiten operar con vectores y matrices, *aquí escribimos nuestra propia librería.* Con ello lograremos que la notación empleada en las notas de clase y las expresiones que usemos en Python se parezcan lo más posible.

Este documento describe tanto el uso de la librería como el modo en el que está programada; pero no es un curso de Python.

No obstante, y pese a la nota de anterior, he escrito unos notebooks de Jupyter que ofrecen unas breves nociones de programación en Python (muy incompletas). Tenga en cuenta que hay muchos cursos y material disponible en la web para aprender Python y que mi labor es enseñar Álgebra Lineal (no Python).

Para hacer más evidente el paralelismo entre las definiciones de las notas de la asignatura y el código de nuestra librería, las partes del código menos didácticas se relegan al final[1] (véase la sección *Literate programming* en la Página 48). Destacar algunas partes del código permitirá apreciar que las definiciones de las notas de la asignatura son implementadas de manera literal en nuestra librería de Python.

Tutorial previo en un Jupyter notebook

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

Y recuerde que **¡hacer matemáticas y programar son prácticamente la misma cosa!**).

[1] aquellas que tienen que ver 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 LaTeX, etc.

Part I
Código principal

Tutorial previo en un Jupyter 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, hemos definiremos una *clase* en Python para los *vectores*, otra para las *matrices*, otra para las *transformaciones elementales* y otra para las *matrices por bloques* (o matrices particionadas).

1 La clase `Vector`

En las notas de la asignatura se dice que

Un *vector* de $\mathbb{R}^n$ es un “sistema” de n números reales;

y dicho sistema se muestra entre 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 crearlos con nueva *clase* en Python. El texto de ayuda de nuestra clase `Vector` es auto-explicativo y será los que Python muestre cuando se teclee `help(Vector)`:

4 ⟨*Texto de ayuda de la clase* `Vector` 4⟩≡

```
"""Clase Vector

Un Vector es una secuencia finita (sistema) de números. Los Vectores se
pueden construir con una lista o tupla de números. Si el argumento es un
Vector, el valor devuelto es el mismo Vector. El atributo 'rpr' indica
al entorno Jupyter el vector debe ser escrito como fila o columna.

Parámetros:
    sis (list, tuple, Vector) : Sistema de números. Debe ser una lista o
        tupla de números, o bien otro Vector
    rpr (str) : Representación en Jupyter ('columna' por defecto).
        Indica la forma de representar el Vector en Jupyter. Si
        rpr='fila' se representa en forma de fila. En caso contrario se
        representa en forma de columna.

Atributos:
    lista (list): sistema de números almacenado
    n     (int) : número de elementos de la lista
    rpr   (str) : modo de representación en Jupyter

Ejemplos:
>>> # Crea un Vector a partir de una lista (o tupla) de números
>>> Vector( [1,2,3] )   # con lista
```

```
>>> Vector( (1,2,3) )   # con tupla

Vector([1,2,3])

>>> # Crea un Vector a partir de otro Vector
>>> Vector( Vector([1,2,3]) )

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

This code is used in chunk 6b.
Uses `Vector` 6b.

1.1 Implementación de los vectores en la clase `Vector`

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

Método de inicialización Comenzamos la clase con el método de inicio: `def __init__(self, ... )`.

- La clase `Vector` usará dos argumentos (o parámetros). Al primero lo llamaremos `sis` y podrá ser una lista o tupla de números, u otro `Vector`. 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 breve texto de ayuda sobre el método `__init__`, que Python nos mostrará si escribimos: `help Vector.__init__`.

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

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

  - Cuando `sis` es un `Vector`, sencillamente se copia su atributo `lista`: `self.lista = sis.lista`

  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!***

  - Cuando `sis` no es ni lista, ni tupla, ni `Vector`, se muestra un mensaje de error.
  - 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 en el entorno Jupyter como fila o como columna (por defecto la representación es en forma de columna).

6a
```
⟨Inicialización de la clase Vector 6a⟩≡
  def __init__(self, sis, rpr='columna'):
      """Inicializa un Vector con una lista, tupla, u otro Vector"""

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

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

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

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

This code is used in chunk 6b.
Uses `Vector` 6b.

La clase `Vector` junto con el listado de sus métodos aparece en el siguiente recuadro:

6b
```
⟨Definición de la clase Vector 6b⟩≡
  class Vector:
      ⟨Texto de ayuda de la clase Vector 4⟩
      ⟨Inicialización de la clase Vector 6a⟩
      ⟨Operador selector por la derecha para la clase Vector 11⟩
      ⟨Operador selector por la izquierda para la clase Vector 12b⟩
      ⟨Suma de Vectores 17b⟩
      ⟨Producto de un Vector por un escalar a su izquierda, o por otro Vector a su izquierda 19a⟩
      ⟨Producto de un Vector por un escalar a su derecha, o por una Matrix a su derecha 20a⟩
      ⟨Definición de la igualdad entre Vectores 20b⟩
      ⟨Representación de la clase Vector 34b⟩
```

This code is used in chunk 31.
Defines:
`Vector`, used in chunks 4, 6–8, 10–12, 14–21, 23, 24a, 35a, 36a, and 39a.

En esta sección hemos visto el texto de ayuda y el método de inicialización. El resto de métodos se describen en secciones posteriores.

Resumen

Los **vectores** son sistemas de números. La clase `Vector` almacena una `list` de números en su atributo `Vector.lista`:

1. Cuando se instancia un `Vector` con una lista, dicha lista se almacena en el atributo `lista`.
2. Cuando se instancia un `Vector` con otro `Vector`, se copia el atributo `lista` de dicho `Vector`.
3. Asociados a los `Vector`es hay una serie de métodos que veremos más adelante.

2 La clase Matrix

En las notas de la asignatura 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

$$\mathbf{A} = [\boldsymbol{v}_1, \dots, \boldsymbol{v}_n], \quad \text{donde } \boldsymbol{v}_i \text{ son vectores de } \mathbb{R}^m.$$

Consecuentemente, en nuestra implementación crearemos un objeto, `Matrix`, que almacene en uno de sus atributos una lista de `Vectores` (todos con el mismo número de componentes). Dicha lista será la lista de "columnas" de la matriz. El texto de ayuda de nuestra clase `Matrix` es auto-explicativo y Python lo mostrará si se teclea `help(Matrix)`.

7 ⟨*Texto de ayuda de la clase* Matrix 7⟩≡

```
"""Clase Matrix

Una Matrix es una secuencia finita (sistema) de Vectores con el mismo
número de componentes. Una Matrix se puede construir con una lista o
tupla de Vectores con el mismo número de componentes (serán las columnas
de la matriz); una lista (o tupla) de listas o tuplas con el mismo
número de componentes (serán las filas de la matriz); una Matrix (el
valor devuelto será la misma Matrix); una BlockMatrix (el valor devuelto
es la Matrix correspondiente a la matriz obtenida al unir todos los
bloques)

Parámetros:
    sis (list, tuple, Matrix, BlockMatrix): Lista (o tupla) de Vectores
        con el mismo núm. de componentes (columnas de la matriz); o
        lista (o tupla) de listas o tuplas con el mismo núm. de
        componentes (filas de la matriz); u otra Matrix; o una
        BlockMatrix (matriz particionada por bloques).

Atributos:
    lista (list): sistema de Vectores almacenado
    m     (int) : número de filas de la matriz
    n     (int) : número de columnas de la matriz

Ejemplos:
>>> # Crea una Matrix a partir de una lista de Vectores
>>> a = Vector( [1,2] )
>>> b = Vector( [1,0] )
>>> c = Vector( [9,2] )
>>> Matrix( [a,b,c] )

Matrix([Vector([1, 2]), Vector([1, 0]), Vector([9, 2])])

>>> # Crea una Matrix a partir de una lista de listas de números
>>> A = Matrix( [ [1,1,9], [2,0,2] ] )
>>> A

Matrix([Vector([1, 2]), Vector([1, 0]), Vector([9, 2])])

>>> # Crea una Matrix a partir de otra Matrix
>>> Matrix( A )
```

```
Matrix([Vector([1, 2]), Vector([1, 0]), Vector([9, 2])])

>>> # Crea una Matrix a partir de una BlockMatrix
>>> Matrix( {1}|A|{2} )

Matrix([Vector([1, 2]), Vector([1, 0]), Vector([9, 2])])
"""
```

This code is used in chunk 9.
Uses `BlockMatrix` 41, `Matrix` 9, and `Vector` 6b.

2.1 Implementación de las matrices en la clase `Matrix`

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

- Una `Matrix` se instancia con el argumento `sis`, que podrá ser *una lista de* `Vector`*es con el mismo número de componentes (serán sus columnas)*; pero que también se podrá ser una lista (o tupla) de listas o tuplas con el mismo número de componentes (serán sus filas), una `BlockMatrix` u otra `Matrix`.
- Luego aparece un breve texto de ayuda del método `__init__`.
- Por último se definen tres atributos para la clase `Matrix`. Los atributos: `lista`, `m` y `n`.
  (El modo de generar el atributo `lista` *depende de qué tipo de objeto es* `sis`*. En el siguiente recuadro se muestra el caso en que* `sis` *es una lista de* `Vector`*es).*
  - El atributo `self.lista` guarda una lista de `Vector`es (que corresponden a la lista de columnas de la matriz). El modo de elaborar dicha lista difiere en función de qué tipo de objeto es el argumento `sis`. Si es
    - `list` (o `tuple`) de `Vector`es: entonces la `self.lista` es `list(sis)` (la lista de `Vector`es introducidos).
    - `list` (o `tuple`) de `list`s o `tuple`s: entonces se interpreta que `sis` es la "lista de filas" de una matriz y se construye la correspondiente lista de columnas para dicha matriz.
    - `Matrix`: entonces `self.lista` es la `lista` de dicha matriz (`self.lista = sis.lista`).
    - `BlockMatrix`: se guarda la `lista` de la `Matrix` resultante de unificar los bloques en una única matriz.

    De esta manera el atributo `self.lista` contendrá la lista ordenada de `Vector`es columna que constituye la matriz... por tanto ***¡ya hemos traducido al lenguaje Python la definición de matriz!***
  - 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.

8 ⟨*Inicialización de la clase* `Matrix` 8⟩≡

```
def __init__(self, sis):
    """Inicializa una Matrix"""

    ⟨Creación del atributo lista cuando sis no es una lista (o tupla) de Vectores 35a⟩

    elif isinstance(sis[0], Vector):
        ⟨Verificación de que todas las columnas de la matriz tendrán la misma longitud 36a⟩
        self.lista  =  list(sis)

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

This code is used in chunk 9.
Uses `Matrix` 9 and `Vector` 6b.

La clase `Matrix` junto con el listado de sus métodos aparece en el siguiente recuadro:

9 ⟨*Definición de la clase* `Matrix` 9⟩≡
```
class Matrix:
    ⟨Texto de ayuda de la clase Matrix 7⟩
    ⟨Inicialización de la clase Matrix 8⟩
    ⟨Operador selector por la derecha para la clase Matrix 13⟩
    ⟨Operador transposición para la clase Matrix 14b⟩
    ⟨Operador selector por la izquierda para la clase Matrix 16⟩
    ⟨Suma de Matrix 21b⟩
    ⟨Producto de una Matrix por un escalar a su izquierda 22b⟩
    ⟨Producto de una Matrix por un escalar, un vector o una matriz a su derecha 24a⟩
    ⟨Definición de la igualdad entre dos Matrix 24b⟩
    ⟨Transformaciones elementales de las columnas de una Matrix 29b⟩
    ⟨Transformaciones elementales de las filas de una Matrix 30b⟩
    ⟨Representación de la clase Matrix 36b⟩
```

This code is used in chunk 31.
Defines:
`Matrix`, used in chunks 7, 8, 12–16, 19–24, 26–30, 32, 35, 39b, 40, and 44.

En esta sección hemos visto el texto de ayuda y el método de inicialización. El resto de métodos se describen en secciones posteriores.

Resumen

Las **matrices** son sistemas de vectores (dichos vectores son sus columnas). La clase `Matrix` almacena una `list` 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 con una lista de `Vector`es, dicha lista se almacena en el atributo `lista`. *Esta es la forma de crear una matriz a partir de sus columnas.*
2. Por comodidad, cuando se instancia una `Matrix` con una lista (o tupla) de listas o tuplas, se interpreta que dicha lista (o tupla) son las filas de la matriz. Consecuentemente, se dan los pasos para describir dicha matriz como una lista de columnas, que se almacena en el atributo `lista`. (Esta forma de instanciar una `Matrix` se usará para programar la **transposición**).
3. Cuando se instancia con otra `Matrix`, se copia el atributo `lista` de dicha `Matrix`.
4. Cuando se instancia con una `BlockMatrix`, se unifican los bloques en una sola matriz, cuya lista de columnas es copiada en el atributo `lista`.
5. Asociados a las `Matrix` hay una serie de métodos que veremos más adelante.

Así pues,

- `Vector` guarda un sistema de números en su atributo `lista`
- `Matrix` guarda una sistema de `Vector`es en su atributo `lista`; por tanto:

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

. . . vamos con el operador selector que nos permitirá definir las operaciones de suma, producto, etc.. . .

3 Operadores selectores

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 $\begin{cases} \mathbf{A}_{|j} = \begin{pmatrix} a_{1j} \\ \vdots \\ a_{nj} \end{pmatrix} \text{ para todo } j \in \{1, \ldots, m\} \\ {}_{i|}\mathbf{A} = (a_{i1}, \ldots, a_{im}) \text{ para todo } i \in \{1, \ldots, n\} \end{cases}$.

Pero puestos a seleccionar, aprovechemos la notación para seleccionar más de un elemento:

Notación en Mates 2

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

Queremos manejar una notación similar en Python, así que tenemos que definir el operador. Y queremos hacerlo con un método de Python que tenga asociado un símbolo con el que se pueda invocar el método de selección

Tutorial previo en un Jupyter notebook

Si no recuerda a qué me estoy refiriendo con los símbolos asociados a métodos, repase de nuevo la sección "Métodos especiales con símbolos asociados" del Notebook referente a **"Clases"** en la carpeta "TutorialPython" en
`https://mybinder.org/v2/gh/mbujosab/LibreriaDePythonParaMates2/master`

Como los métodos `__or__` y `__ror__` tienen asociados la barra vertical, usaremos el siguiente convenio:

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`

3.1 Operador selector por la derecha para la clase `Vector`.

10 ⟨*Texto de ayuda para el operador selector por la derecha para la clase* `Vector` 10⟩≡

```
"""Selector por la derecha

Extrae la i-ésima componente del Vector, o genera un nuevo vector con
las componentes indicadas (los índices comienzan por la posición 1)
```

```
    Parámetros:
        i (int, list, tuple): Índice (lista de índices) de los elementos a
           seleccionar

    Resultado:
        número: Cuando i es int, devuelve el componente i-ésimo del Vector.
        Vector: Cuando i es list o tuple, devuelve el Vector formado por los
            componentes indicados en la lista de índices.

    Ejemplos:
    >>> # Selección de una componente
    >>> Vector([10,20,30]) | 2

    20

    >>> # Creación de un sub-vector a partir de una lista o tupla de índices
    >>> Vector([10,20,30]) | [2,1,2]
    >>> Vector([10,20,30]) | (2,1,2)

    Vector([20, 10, 20])
    """
```

This code is used in chunk 11.
Uses `Vector` 6b.

3.1.1 Implementación del operador 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 objetos iterables comienzan en cero, por lo que para seleccionar el elemento *i*-ésimo de `lista`, escribimos `lista[i-1]`; así `a|1` debe seleccionar el primer elemento del atributo `lista`, es decir `a.lista[0]`).

Una vez hemos definido el operador "`|`" cuando el argumento `i` es un entero (`int`), podemos usar el método (`self|a`) para definir el operador cuando el argumento `i` es una lista o tupla `(list,tuple)` de índices (entonces generamos un `Vector` con las componentes indicadas).

11 ⟨*Operador selector por la derecha para la clase* `Vector` 11⟩≡

```
  def __or__(self,i):
      ⟨Texto de ayuda para el operador selector por la derecha para la clase Vector 10⟩

      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 6b.
Uses `Vector` 6b.

3.2 Operador selector por la izquierda para la clase `Vector`.

12a ⟨*Texto de ayuda para el operador selector por la izquierda para la clase* `Vector` 12a⟩≡
```
"""Selector por la izquierda

Hace lo mismo que el método __or__ solo que operando por la izquierda
"""
```
This code is used in chunk 12b.

3.2.1 Implementación del operador selector por la derecha para la clase `Vector`.

Como hace lo mismo que el selector por la derecha, basta con llamar al selector por la derecha: `self|i`

12b ⟨*Operador selector por la izquierda para la clase* `Vector` 12b⟩≡
```
def __ror__(self,i):
    ⟨Texto de ayuda para el operador selector por la izquierda para la clase Vector 12a⟩

    return self | i
```
This code is used in chunk 6b.

3.3 Operador selector por la derecha para la clase `Matrix`.

12c ⟨*Texto de ayuda para el operador selector por la derecha para la clase* `Matrix` 12c⟩≡
```
"""
Extrae la i-ésima columna de Matrix; o crea una Matrix con las columnas
indicadas; o crea una BlockMatrix particionando una Matrix por las
columnas indicadas (los índices comienzan por la posición 1)

Parámetros:
    j (int, list, tuple): Índice (lista de índices) de las columnas a
          seleccionar
      (set): Conjunto de índices de las columnas por donde particionar

Resultado:
    Vector: Cuando j es int, devuelve la columna j-ésima de Matrix.
    Matrix: Cuando j es list o tuple, devuelve la Matrix formada por las
        columnas indicadas en la lista de índices.
    BlockMatrix: Cuando j es un set, devuelve la BlockMatrix resultante
        de particionar la matriz por las columnas indicadas

Ejemplos:
>>> # Extrae la j-ésima columna la matriz
>>> Matrix([Vector([1,0]), Vector([0,2]), Vector([3,0])]) | 2
```

```
Vector([0,2])

>>> # Matrix formada por Vectores columna indicados en la lista (tupla)
>>> Matrix([Vector([1,0]), Vector([0,2]), Vector([3,0])]) | [2,1]
>>> Matrix([Vector([1,0]), Vector([0,2]), Vector([3,0])]) | (2,1)

Matrix( [Vector([0,2]), Vector([1,0])] )

>>> # BlockMatrix de la partición de la matriz por la segunda columna
>>> Matrix([Vector([1,0]), Vector([0,2]), Vector([3,0])]) | {2}

BlockMatrix( [ [ Matrix([Vector([1, 0]), Vector([0, 2])]),
                 Matrix([Vector([3, 0])]) ] ] )
"""
```

This code is used in chunk 13.
Uses `BlockMatrix` 41, `Matrix` 9, and `Vector` 6b.

3.3.1 Implementación del operador 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 que selecciona una única columna (cuando el argumento `j` es un entero), usaremos repetidamente el procedimiento `(self|j)` para crear una `Matrix` formada por las columnas indicadas (cuando el parámetro `j` es una lista, o tupla, de índices).
(*la partición en bloques de columnas de matrices se verá más adelante, en la sección de la clase* `BlockMatrix`).

13 ⟨*Operador selector por la derecha para la clase* `Matrix` 13⟩≡

```
def __or__(self,j):
    ⟨Texto de ayuda para el operador selector por la derecha para la clase Matrix 12c⟩
    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 42c⟩
```

This code is used in chunk 9.
Uses `Matrix` 9.

3.4 Operador transposición de una `Matrix`.

Implementar el operador selector por la izquierda es aquí algo más complicado que en el caso de los vectores, pues ya no es lo mismo operar por la derecha que por la izquierda. Como paso intermedio vamos a definir el operador transposición, que usaremos después para definir el operador selector por la izquierda (selección de filas).

Notación en Mates 2

Denotamos la *transpuesta* de $\mathbf{A}$ con: $\mathbf{A}^{\intercal}$; y es la matriz tal que $\mathbf{A}^{\intercal}{}_{|j} = {}_{j|}\mathbf{A}$; $j = 1:n$.

14a ⟨*Texto de ayuda para el operador transposición de la clase* `Matrix` 14a⟩≡

```
"""
Devuelve la traspuesta de una matriz

Ejemplo:
>>> ~Matrix([Vector([1]), Vector([2]), Vector([3])])

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

This code is used in chunk 14b.
Uses `Matrix` 9 and `Vector` 6b.

3.4.1 Implementación del operador transposición.

Desgraciadamente Python no dispone del símbolo "$^{\intercal}$". Así que hemos de usar un símbolo distinto para indicar transposición. Y además no tenemos muchas opciones, pues el conjunto de símbolos asociados a métodos especiales es muy limitado.

Tutorial previo en un Jupyter notebook

Si no recuerda a qué me estoy refiriendo con los símbolos asociados a métodos, repase de nuevo la sección "Métodos especiales con símbolos asociados" del Notebook referente a **"Clases"** en la carpeta "TutorialPython" en
`https://mybinder.org/v2/gh/mbujosab/LibreriaDePythonParaMates2/master`

Para implementar la transposición haremos uso del método `__invert__`, que tiene asociado el símbolo del la tilde "~", símbolo que además deberemos colocar a la izquierda de la matriz.

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

Recuerde que con la segunda forma de instanciar una `Matrix` (véase el resumen de la página 9), 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: $1, 2, \ldots, m$).

14b ⟨*Operador transposición para la clase* `Matrix` 14b⟩≡

```
def __invert__(self):
    ⟨Texto de ayuda para el operador transposición de la clase Matrix 14a⟩

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

This code is used in chunk 9.
Uses `Matrix` 9.

3.5 Operador selector por la izquierda para la clase `Matrix`.

15 ⟨*Texto de ayuda para el operador selector por la izquierda para la clase* `Matrix` 15⟩≡

```
"""Operador selector por la izquierda

Extrae la i-ésima fila de Matrix; o crea una Matrix cuyas filas son las
indicadas; o crea una BlockMatrix particionando una Matrix por las filas
indicadas (los índices comienzan por la posición 1)

Parámetros:
    i (int, list, tuple): Índice (lista de índices) de las filas a
         seleccionar
      (set): Conjunto de índices de las filas por donde particionar

Resultado:
    Vector: Cuando i es int, devuelve la fila i-ésima de Matrix.
    Matrix: Cuando i es list o tuple, devuelve la Matrix cuyas filas son
        las indicadas en la lista de índices.
    BlockMatrix: Cuando i es un set, devuelve la BlockMatrix resultante
        de particionar la matriz por las filas indicadas

Ejemplos:
>>> # Extrae la j-ésima columna la matriz
>>> 2 | Matrix([Vector([1,0]), Vector([0,2]), Vector([3,0])])

Vector([0, 2, 0])

>>> # Matrix formada por Vectores fila indicados en la lista (tupla)
>>> [1,1] | Matrix([Vector([1,0]), Vector([0,2]), Vector([3,0])])
>>> (1,1) | Matrix([Vector([1,0]), Vector([0,2]), Vector([3,0])])

Matrix([Vector([1, 1]), Vector([0, 0]), Vector([3, 3])])

>>> # BlockMatrix de la partición de la matriz por la primera fila
>>> {1} | Matrix([Vector([1,0]), Vector([0,2])])

BlockMatrix( [ [Matrix([Vector([1]),Vector([0])])],
               [Matrix([Vector([0]),Vector([2])])] ] )
"""
```

This code is used in chunk 16.
Uses `BlockMatrix` 41, `Matrix` 9, and `Vector` 6b.

3.5.1 Implementación del operador por la izquierda para la clase `Matrix`.

Con el operador selector de columnas y la transposición, es inmediato definir un operador selector de *filas*. . . ¡pues son las columnas de la matriz transpuesta!

`(~self)|j`

(para recordar que se ha obtenido una fila de la matriz, representamos el `Vector` en horizontal: `rpr='fila'`)

Una vez definido el operador por la izquierda (`(i|self)`), podemos usarlo repetidas veces para crear una `Matrix` con las filas escogidas.
(*la partición en bloques de filas de matrices se verá más adelante, en la sección de la clase* `BlockMatrix`).

16 ⟨*Operador selector por la izquierda para la clase* `Matrix` 16⟩≡

```
  def __ror__(self,i):
      ⟨Texto de ayuda para el operador selector por la izquierda para la clase Matrix 15⟩

      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 42b⟩
```

This code is used in chunk 9.
Uses `Matrix` 9 and `Vector` 6b.

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 la asignatura!

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

4 Operaciones con vectores y matrices

Una vez definidas las clases `Vector` y `Matrix` junto con los respectivos operadores selectores "`|`", 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 la asignatura:

4.1 Suma de vectores

En las notas de la asignatura hemos definido la suma de dos vectores de $\mathbb{R}^n$ como

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

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, `other` es otro vector y `range(1,self.n+1)` es el rango de valores: $1, \dots, n$.

17a
```
⟨Texto de ayuda para el operador suma en la clase Vector 17a⟩≡
  """Devuelve el Vector resultante de sumar dos Vectores

  Parámetros:
      other (Vector): Otro vector con el mismo número de elementos

  Ejemplo
  >>> Vector([10, 20, 30]) + Vector([-1, 1, 1])

  Vector([9, 21, 31])
  """
```
This code is used in chunk 17b.
Uses `Vector` 6b.

4.1.1 Implementación

17b
```
⟨Suma de Vectores 17b⟩≡
  def __add__(self, other):
      ⟨Texto de ayuda para el operador suma en la clase Vector 17a⟩

      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 6b.
Uses `Vector` 6b.

4.2 Producto de un vector por un escalar u otro vector que estén a su izquierda

En las notas hemos definido

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

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

  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

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

  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`).

18 ⟨*Texto de ayuda para el operador producto por la izquierda en la clase* `Vector` 18⟩≡

```
"""Multiplica un Vector por un número o Vector que estén a su izquierda

Parámetros:
    x (int, float o Fraction): Número por el que se multiplica
      (Vector): Vector con el mismo número de componentes.

Resultado:
    Vector: Cuando x es int, float o Fraction, devuelve el Vector que
        resulta de multiplicar cada componente por x
    Número: Cuando x es Vector, devuelve el producto punto entre
        vectores (producto escalar usual en R^n)

Ejemplos:
>>> 3 * Vector([10, 20, 30])

Vector([30, 60, 90])

>>> Vector([1, 1, 1]) * Vector([10, 20, 30])

60
"""
```

This code is used in chunk 19a.
Uses `Vector` 6b.

4.2.1 Implementación

19a

```
⟨Producto de un Vector por un escalar a su izquierda, o por otro Vector a su izquierda 19a⟩≡
  def __rmul__(self, x):
      ⟨Texto de ayuda para el operador producto por la izquierda en la clase Vector 18⟩
      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 6b.
Uses Vector 6b.

4.3 Producto de un vector por un escalar o una Matrix que estén a su derecha

- 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.

19b

```
⟨Texto de ayuda para el operador producto por la derecha en la clase Vector 19b⟩≡
  """Multiplica un Vector por un número o Matrix a su derecha.

  Parámetros:
      x (int, float o Fraction): Número por el que se multiplica
        (Matrix): Matrix con tantas filas como componentes tiene el Vector

  Resultado:
      Vector: Cuando x es int, float o Fraction, devuelve el Vector que
         resulta de multiplicar cada componente por x
            Cuando x es Matrix, devuelve el Vector combinación lineal de
```

```
        las filas de Matrix (componentes del Vector son los coeficientes
        de la combinación lineal)

  Ejemplos:
  >>> Vector([10, 20, 30]) * 3

  Vector([30, 60, 90])

  >>> a = Vector([1, 1])
  >>> B = Matrix([Vector([1, 2]), Vector([1, 0]), Vector([9, 2])])
  >>> a * B

  Vector([3, 1, 11])
  """
```

This code is used in chunk 20a.
Uses `Matrix` 9 and `Vector` 6b.

4.3.1 Implementación

20a ⟨*Producto de un* `Vector` *por un escalar a su derecha, o por una* `Matrix` *a su derecha* 20a⟩≡

```
  def __mul__(self, x):
      ⟨Texto de ayuda para el operador producto por la derecha en la clase Vector 19b⟩

      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 6b.
Uses `Matrix` 9 and `Vector` 6b.

4.4 Igualdad entre vectores

Dos vectores son iguales cuando lo son los sistemas de números correspondientes a ambos vectores.

20b ⟨*Definición de la igualdad entre* `Vector`*es* 20b⟩≡

```
  def __eq__(self, other):
      """Indica si es cierto que dos vectores son iguales"""
      return self.lista == other.lista
```

This code is used in chunk 6b.

4.5 Suma de matrices

En las notas de la asignatura hemos definido la suma de matrices como

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

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.

21a ⟨*Texto de ayuda para el operador suma en la clase* `Matrix` 21a⟩≡

```
"""Devuelve la Matrix resultante de sumar dos Matrices

Parámetros:
    other (Matrix): Otra Matrix con el mismo número de filas y columnas

Ejemplo:
>>> A = Matrix( [Vector([1,0]), Vector([0,1])] )
>>> B = Matrix( [Vector([0,2]), Vector([2,0])] )
>>> A + B

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

This code is used in chunk 21b.
Uses `Matrix` 9 and `Vector` 6b.

4.5.1 Implementación

21b ⟨*Suma de* `Matrix` 21b⟩≡

```
def __add__(self, other):
    ⟨Texto de ayuda para el operador suma en la clase Matrix 21a⟩

    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 9.
Uses `Matrix` 9.

4.6 Producto de una matriz por un escalar a su izquierda

En las notas hemos definido

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

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

  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`).

22a ⟨*Texto de ayuda para el operador producto por la izquierda en la clase* `Matrix` 22a⟩≡

```
"""Multiplica una Matrix por un número a su izquierda.

Parámetros:
    x (int, float o Fraction): Número por el que se multiplica

Resultado:
    Matrix: Devuelve el múltiplo de la Matrix

Ejemplo:
>>> 10 * Matrix([[1,2],[3,4]])

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

This code is used in chunk 22b.
Uses `Matrix` 9.

4.7 Implementación

22b ⟨*Producto de una* `Matrix` *por un escalar a su izquierda* 22b⟩≡

```
def __rmul__(self,x):
    ⟨Texto de ayuda para el operador producto por la izquierda en la clase Matrix 22a⟩

    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 9.
Uses `Matrix` 9.

4.8 Producto de una matriz por un escalar, un vector o una matriz a su derecha

- 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

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

  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`).

23 ⟨*Texto de ayuda para el operador producto por la derecha en la clase* `Matrix` 23⟩≡

```
"""Multiplica una Matrix por un número, Vector o Matrix a su derecha

Parámetros:
    x (int, float o Fraction): Número por el que se multiplica
      (Vector): Vector con tantos componentes como columnas tiene Matrix
      (Matrix): con tantas filas como columnas tiene la Matrix

Resultado:
    Matrix: Si x es int, float o Fraction, devuelve la Matrix que
       resulta de multiplicar cada columna por x
    Vector: Si x es Vector, devuelve el Vector combinación lineal de las
       columnas de Matrix (los componentes del Vector son los
       coeficientes de la combinación)
    Matrix: Si x es Matrix, devuelve el producto entre las matrices

Ejemplos:
>>> # Producto por un número
>>> Matrix([[1,2],[3,4]]) * 10

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

>>> # Producto por un Vector
>>> Matrix([Vector([1, 3]), Vector([2, 4])]) * Vector([1, 1])

Vector([3, 7])

>>> # Producto por otra Matrix
>>> Matrix([Vector([1, 3]), Vector([2, 4])]) * Matrix([Vector([1,1])]))

Matrix([Vector([3, 7])])
"""
```

This code is used in chunk 24a.
Uses `Matrix` 9 and `Vector` 6b.

4.9 Implementación

24a ⟨*Producto de una* Matrix *por un escalar, un vector o una matriz a su derecha* 24a⟩≡
```
def __mul__(self,x):
    ⟨Texto de ayuda para el operador producto por la derecha en la clase Matrix 23⟩

    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 9.
Uses `Matrix` 9, `V0` 39a, and `Vector` 6b.

4.9.1 Igualdad entre matrices

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

24b ⟨*Definición de la igualdad entre dos* Matrix 24b⟩≡
```
def __eq__(self, other):
    """Indica si es cierto que dos matrices son iguales"""
    return self.lista == other.lista
```
This code is used in chunk 9.

5 La clase transformación elemental `T`

Notación en Mates 2

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

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

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

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

Comentario sobre la notación. Como una trasformación elemental es el resultado del producto con una matriz elemental, esta notación busca el parecido con la notación del producto matricial:

> Al poner la *abreviatura* "$\boldsymbol{\tau}$" de la transformación elemental a derecha (o izquierda), es como si multiplicáramos la matriz $\mathbf{A}$ por la derecha (o por la izquierda) por la correspondiente matriz elemental $\mathbf{I}_{\boldsymbol{\tau}} = \mathbf{E}$ (o ${}_{\boldsymbol{\tau}}\mathbf{I} = \mathbf{E}$).

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 completamente[2] 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
$\mathbf{A}\underset{[\boldsymbol{i}\rightleftharpoons\boldsymbol{j}]}{\boldsymbol{\tau}}$	`A & T( {i,j} )`	$\underset{[\boldsymbol{i}\rightleftharpoons\boldsymbol{j}]}{\boldsymbol{\tau}}\mathbf{A}$	`T( {i,j } ) & A`
$\mathbf{A}\underset{[a\cdot\boldsymbol{i}]}{\boldsymbol{\tau}}$	`A & T( (i,a) )`	$\underset{[a\cdot\boldsymbol{i}]}{\boldsymbol{\tau}}\mathbf{A}$	`T( (i,a) ) & A`
$\mathbf{A}\underset{[\boldsymbol{i}+a\cdot\boldsymbol{j}]}{\boldsymbol{\tau}}$	`A & T( (i,j,a) )`	$\underset{[\boldsymbol{i}+a\cdot\boldsymbol{j}]}{\boldsymbol{\tau}}\mathbf{A}$	`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 expresar las secuencias de transformaciones elementales ${}_{\boldsymbol{\tau}_k\cdots\boldsymbol{\tau}_1}\mathbf{A}$ y $\mathbf{A}_{\boldsymbol{\tau}_1\cdots\boldsymbol{\tau}_k}$ con una lista de transformaciones elementales, de manera que logremos las siguientes equivalencias entre expresiones:

$$\mathbf{A}\ \texttt{\&}\ \texttt{T(}\ t_1\texttt{)}\ \texttt{\&}\ \texttt{T(}\ t_2\ \texttt{)}\ \texttt{\&}\cdots\texttt{\&}\ \texttt{T(}\ t_k\texttt{)} \quad = \quad \mathbf{A}\ \texttt{\&}\ \texttt{T(}\ \texttt{[}t_1\texttt{,}t_2\texttt{,}\ldots\texttt{,}t_k\texttt{]}\ \texttt{)}$$

$$\texttt{T(}\ t_k\texttt{)}\ \texttt{\&}\cdots\texttt{\&}\ \texttt{T(}\ t_2\ \texttt{)}\ \texttt{\&}\ \texttt{T(}\ t_1\texttt{)}\ \texttt{\&}\ \mathbf{A} \quad = \quad \texttt{T(}\ \texttt{[}t_1\texttt{,}t_2\texttt{,}\ldots\texttt{,}t_k\texttt{]}\ \texttt{)}\ \texttt{\&}\ \mathbf{A}$$

[2] Fíjese que la notación usada en las notas de la asignatura para las matrices elementales $\mathbf{E}$, no las describe completamente, pues se deja al lector la deducción de cuál es el orden adecuado para poder realizar el producto $\mathbf{AE}$ o $\mathbf{EA}$

26 ⟨*Texto de ayuda de la clase* T *(Transformación Elemental)* 26⟩≡

```
"""Clase T

T es un objeto que denominaremos transformación elemental. Guarda en su
atributo 't' una abreviatura de una transformación elemental o una
secuencia de abreviaturas de transformaciones elementales. Con el método
__and__ actúa sobre otra T para crear una T que es composición de
transformaciones elementales (la lista de abreviaturas), o actúa sobre
una Matrix (para transformar sus filas)

Atributos:
    t (set)  : {índice, índice}. Abreviatura de un intercambio entre los
                 vectores correspondientes a dichos índices
      (tuple): (índice, número). Abreviatura de transformación Tipo II
                 que multiplica el vector correspondiente al índice por
                 el número
             : (índice1, índice2, número). Abreviatura de transformación
                 Tipo I que suma al vector correspondiente al índice1 el
                 vector correspondiente al índice2 multiplicado por el
                 número
      (list) : Lista de conjuntos y tuplas. Secuencia de abreviaturas de
                 transformaciones como las anteriores.

Ejemplos:
>>> # Intercambio entre vectores
>>> T( {1,2} )

>>> # Trasformación Tipo II (multiplica por 5 el segundo vector)
>>> T( (2,5) )

>>> # Trasformación Tipo I (resta al primer vector el tercero)
>>> T( (1,3,-1) )

>>> # Secuencia de las tres transformaciones anteriores
>>> T( [{1,2}, (2,5), (1,3,-1)] )
"""
```

This code is used in chunk 28c.
Uses `Matrix` 9 and `T` 28c.

5.1 Implementación

Por los Notebooks de Jupyter que acompañan a esta documentación ya sabemos que Python ejecuta las órdenes de izquierda a derecha. Fijándonos en la expresión

$$\mathbf{A}\ \texttt{\&}\ \texttt{T(}\ t_1\texttt{)}\ \texttt{\&}\ \texttt{T(}\ t_2\ \texttt{)}\ \texttt{\&}\cdots\texttt{\&}\ \texttt{T(}\ t_k\texttt{)}$$

podríamos pensar que podemos implementar la transformación elemental como un método de la clase `Matrix`. Así, al definir el método `__and__` por la derecha de la matriz podemos indicar que $\mathbf{A}\ \texttt{\&}\ \texttt{T(}\ t_1\texttt{)}$ es una nueva matriz con las columnas modificadas. Python no tiene problema en ejecutar $\mathbf{A}\ \texttt{\&}\ \texttt{T(}\ t_1\texttt{)}\ \texttt{\&}\ \texttt{T(}\ t_2\ \texttt{)}\ \texttt{\&}\cdots\texttt{\&}\ \texttt{T(}\ t_k\texttt{)}$ pues ejecutar de izquierda a derecha, es lo mismo que ejecutar $\Big(\big((\mathbf{A}\ \texttt{\&}\ \texttt{T(}\ t_1\texttt{)})\texttt{\&}\ \texttt{T(}\ t_2\ \texttt{)}\big)\texttt{\&}\cdots\Big)\texttt{\&}\ \texttt{T(}\ t_k\texttt{)}$ donde la expresión dentro de cada paréntesis es una `Matrix`, por lo que las operaciones están definidas. La dificultad aparece con

$$\texttt{T(}\ t_k\texttt{)}\ \texttt{\&}\cdots\texttt{\&}\ \texttt{T(}\ t_2\ \texttt{)}\ \texttt{\&}\ \texttt{T(}\ t_1\texttt{)}\ \texttt{\&}\ \mathbf{A}$$

Lo primero que Python tratara de ejecutar es `T(` t_k `) & T(` t_{k-1}`)`, pero ni `T(` t_k`)` ni `T(` t_{k-1}`)` son matrices, por lo que esto no puede ser programado como un método de la clase `Matrix`.

Así pues, necesitamos definir una nueva clase que almacene las *abreviaturas* "$\boldsymbol{\tau}$"de las operaciones elementales, de manera que podamos definir `T(` t_k `) & T(` t_{k-1}`)`, como un método que "compone" dos transformaciones elementales para formar una secuencias de abreviaturas (u operaciones a ejecutar sobre una `Matrix`).

Definimos un nuevo tipo de objeto: `T` (transformación elemental) que nos permitirá encadenar transformaciones elementales (es decir, almacenar una lista de abreviaturas). El código del siguiente recuadro inicializa la clase. Con el atributo `rpr` controlaremos el modo de representación en el entorno Jupyter.

27a ⟨*Inicialización de la clase* `T` *(Transformación Elemental)* 27a⟩≡
```
def __init__(self, t, rpr='V'):
    """Inicializa una transformación elemental"""
    self.t   = t
    self.rpr = rpr
```
This code is used in chunk 28c.

5.1.1 Composición de transformaciones elementales

27b ⟨*Texto de ayuda para la composición de Transformaciones Elementales* `T` 27b⟩≡
```
"""Composición de transformaciones elementales (o transforma filas)

Crea una T con la lista de abreviaturas de transformaciones elementales

Parámetros:
    other (T): Realiza la composición de transformaciones (lista de
                abreviaturas)
          (Matrix): Llama al método de la clase Matrix que modifica las
                filas de una Matrix
Ejemplos:
>>> # Composición de dos transformaciones elementales
>>> T( {1, 2} ) & T( (2, 4) )

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

>>> # Composición de una trasformación con una composición de varias
>>> T( {1, 2} ) & T( [(2, 4), (1, 2), {3, 1}] )

T( [{1, 2}, (2, 4), (1, 2), {3, 1}] )

>>> # Transformación de las filas de una Matrix
>>> T( [{1,2}, (2,4)] ) & A # (intercambia las dos primeras filas y
                            # luego multiplica la segunda por 4)
"""
```
Root chunk (not used in this document).
Uses `Matrix` 9 and `T` 28c.

Describimos la composición de transformaciones `T(`t_1`) & T(`t_2`)`, creando una lista de abreviaturas `[`t_1, t_2`]` (mediante la concatenación de listas)[3]. Si el atributo del método `__and__` de la clase `T` es una `Matrix`, llama al método `__rand` de la clase `Matrix`, (`T & Matrix` que transforma las filas de la matriz y que veremos un poco más abajo).

28a
```
⟨Composición de Transformaciones Elementales o aplicación sobre las filas de una Matrix 28a⟩≡
  def __and__(self, other):
      ⟨Método auxiliar CreaLista que devuelve listas de abreviaturas 28b⟩
      if isinstance(other, T):
          return T(CreaLista(self.t) + CreaLista(other.t), rpr=other.rpr)

      if isinstance(other, Matrix):
          return other.__rand__(self)
```
This code is used in chunk 28c.
Uses `CreaLista` 28b, `Matrix` 9, and `T` 28c.

La composición de transformaciones elementales usa el siguiente procedimiento auxiliar que nos permitirá concatenar listas de abreviaturas.

28b
```
⟨Método auxiliar CreaLista que devuelve listas de abreviaturas 28b⟩≡
  def CreaLista(t):
      """Si t es una lista, devuelve t; si no devuelve la lista: [t]"""
      return ( t if isinstance(t, list) else [t] )
```
This code is used in chunk 28a.
Defines:
`CreaLista`, used in chunk 28a.

La clase `T` junto con el listado de sus métodos aparece en el siguiente recuadro:

28c
```
⟨Definición de la clase T (Transformación Elemental) 28c⟩≡
  class T:
      ⟨Texto de ayuda de la clase T (Transformación Elemental) 26⟩
      ⟨Inicialización de la clase T (Transformación Elemental) 27a⟩
      ⟨Composición de Transformaciones Elementales o aplicación sobre las filas de una Matrix 28a⟩
      ⟨Representación de la clase T 38⟩
```
This code is used in chunk 31.
Defines:
`T`, used in chunks 26–30, 32a, and 38.

[3]Recuerde que la suma de listas (`list + list`) concatena las listas

6 Transformaciones elementales de una Matrix

6.1 Transformaciones elementales de las columnas de una Matrix

29a ⟨*Texto de ayuda de las transformaciones elementales de las columnas de una* Matrix 29a⟩≡

```
"""Transforma las columnas de una Matrix

Atributos:
    t (T): transformaciones a aplicar sobre las columnas de Matrix

Ejemplos:
>>>  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 anteriores
"""
```

This code is used in chunk 29b.
Uses Matrix 9 and T 28c.

Implementación de la aplicación de las transformaciones elementales sobre las columnas de una Matrix (incluimos el intercambio, aunque ya se sabe que realmente es una composición de los otros dos tipos de transformaciones).

29b ⟨*Transformaciones elementales de las columnas de una* Matrix 29b⟩≡

```
def __and__(self,t):
    ⟨Texto de ayuda de las transformaciones elementales de las columnas de una Matrix 29a⟩

    if isinstance(t.t,set):
        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.copy()

    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.copy()

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

    return self
```

This code is used in chunk 9.
Uses Matrix 9 and T 28c.

Observación 1. Al actuar sobre `self.lista`, las transformaciones elementales modifican la matriz.

6.2 Transformaciones elementales de las filas de una Matrix

30a

```
⟨Texto de ayuda de las transformaciones elementales de las filas de una Matrix 30a⟩≡
  """Transforma las filas de una Matrix

  Atributos:
      t (T): transformaciones a aplicar sobre las filas de Matrix

  Ejemplos:
  >>>    {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
  """
```

This code is used in chunk 30b.
Uses Matrix 9 and T 28c.

Para implementar las transformaciones elementales de las filas usamos el truco de aplicar las operaciones sobre las columnas de la transpuesta y de nuevo transponer el resultado: `~(~self & t)`.

Al aplicar 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`):

$$\texttt{T([}t_1\texttt{,}t_2\texttt{,...,}t_k\texttt{]) \& }\mathbf{A} \quad = \quad \texttt{T(}t_k\texttt{) \&}\cdots\texttt{\& T(}t_2\texttt{) \& T(}t_1\texttt{) \& }\mathbf{A}$$

30b

```
⟨Transformaciones elementales de las filas de una Matrix 30b⟩≡
  def __rand__(self,t):
      ⟨Texto de ayuda de las transformaciones elementales de las filas de una Matrix 30a⟩

      if isinstance(t.t,set) | isinstance(t.t,tuple):
          self.lista = (~(~self & t)).lista.copy()

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

      return self
```

This code is used in chunk 9.
Uses T 28c.

Observación 2. Al actuar sobre `self.lista`, las transformaciones elementales modifican la matriz.

7 Librería completa

Finalmente creamos la librería `notacion.py` concatenando los trozos de código que se describen en este fichero de documentación (recuerde que el número que aparece detrás de nombre de cada trozo de código indica la página de este documento donde encontrar el código).

Pero antes de todo, y para que los vectores funcionen como un espacio vectorial, es esencial importar la clase `Fraction` de la librería `fractions` [4]. Así pues, antes de nada, importamos la clase Fraction de números fraccionarios con el código:

```
from fractions import Fraction
```

31 ⟨*notacion.py* 31⟩≡
```
# coding=utf8

from fractions import Fraction

⟨Método html general 33a⟩
⟨Método latex general 33b⟩
⟨Métodos html y latex para fracciones 34a⟩

⟨Definición de la clase Vector 6b⟩
⟨Definición de la clase Matrix 9⟩
⟨Definición de la clase T (Transformación Elemental) 28c⟩
⟨Definición de la clase BlockMatrix 41⟩

⟨Definición del método partición 42a⟩
⟨Definición del procedimiento de generación del conjunto clave para particionar 44a⟩

⟨Definición de vector nulo: V0 39a⟩
⟨Definición de matriz nula: M0 39b⟩
⟨Definición de la matriz identidad: I 40⟩

⟨normal 32a⟩
⟨sistema 32b⟩
```
Root chunk (not used in this document).

Tutorial previo en un Jupyter notebook

Este Notebook es un ejemplo sobre el **uso de nuestra librería para Mates 2** en la carpeta "TutorialPython" en
`https://mybinder.org/v2/gh/mbujosab/LibreriaDePythonParaMates2/master`

Para empaquetar esta librería en el futuro: `https://packaging.python.org/tutorials/packaging-projects/`

[4]el tipo de datos `float` no tiene estructura de cuerpo. Por ello vamos a emplear números fraccionarios del tipo $\frac{a}{b}$, donde a y b son enteros. Así pues, en el fondo estaremos trabajando con vectores de $\mathbb{Q}^n$ (en lugar de vectores de $\mathbb{R}^n$). Más adelante intentaré emplear un cuerpo de números más grande, que incluya números reales tales como $\sqrt{2}$... y también el cuerpo de fracciones de polinomios (para obtener el polinomio característico vía eliminación gaussiana, en lugar de vía determinantes).

8 Ejemplo de uso

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

32a ⟨*normal* 32a⟩≡

```
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, 1/Fraction(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 31.
Uses `Matrix` 9 and `T` 28c.

32b ⟨*sistema* 32b⟩≡

```
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 31.
Uses `BlockMatrix` 41 and `Matrix` 9.

Part II

Otros trozos de código

A Métodos de representación para el entorno 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. En el navegador, la librería MathJax de Javascript se encargará de convertir la expresión LaTeX en la grafía correspondiente.

33a ⟨*Método html general* 33a⟩≡

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

This code is used in chunk 31.

El método `latex`, convertirá en cadena de caracteres los inputs de tipo `float` o `int` [5], y en el resto de casos 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 LaTeX y los tipos de datos que corresponden a números `int` o `float`).

33b ⟨*Método latex general* 33b⟩≡

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

This code is used in chunk 31.

Si el objeto a representar no es un número de coma flotante (`float`) ni tampoco un entero (`int`), el método general `latex` llamará el método `latex` de la clase correspondiente. Por tanto, si `a` es un `Vector`, una `Matrix`, o una transformación elemental (`T`), se llama al método `a.latex` definido en la clase correspondiente a dicho objeto `a`.[6] Sin embargo, la clase `Fraction` no tiene definidos los métodos de representación `html` o `latex`. Así pues, para representar fracciones necesitamos incorporar estos métodos en la preexistente clase `Fraction` que hemos importado desde la librería `fractions`. Primero definimos el método `_repr_html_fraction` (que sencillamente llamara al método `latex`) y luego definimos el método `latex_fraction` (en el nombre de ambos métodos he incluido la coletilla `fraction` para recordar que son los métodos que usaremos para la clase `fraction`). Si en LaTeX queremos representar la fración $\frac{a}{b}$ escribimos el código: `\frac{a}{b}`. Pero cuando el denominador es $b = 1$, no nos gusta escribir $\frac{a}{1}$, preferimos mostrar solamente el numerador a. Esto es precisamente lo que hace el método `latex_fraction` de más abajo.

Finalmente, con la función `setattr`, añadimos a la clase `Fraction` un método que se llamará '`_repr_html_`' (y que hace lo que hemos indicado al definir `_repr_html_fraction`), y un método que se llamará '`latex`' (y que hace los que hemos indicado al definir `latex_fraction`).

[5]resulta que para los tipos de datos `int` y `float` no es posible definir un nuevo método de representación. Afortunadamente la cadena de caracteres que representa el número nos vale perfectamente en ambos casos (tanto para los números enteros, `int`, como los números con decimales, `float`).

[6]más adelante tendré que incluir métodos de representación para otros tipos de datos, por ejemplo para poder representar vectores o matrices con polinomios.

34a

```
⟨Métodos html y latex para fracciones 34a⟩≡
  def _repr_html_fraction(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_fraction)
  setattr(Fraction, 'latex', latex_fraction)
This code is used in chunk 31.
```

B Completando la clase Vector

B.1 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

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

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.

34b

```
⟨Representación de la clase Vector 34b⟩≡
  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 6b.

C Completando la clase Matrix

C.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.

35a ⟨*Creación del atributo* lista *cuando* sis *no es una lista (o tupla) de* Vector*es* 35a⟩≡

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

  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 35b⟩

  elif isinstance(sis[0], (list, tuple)):
      ⟨Verificación de que todas las filas de la matriz tendrán la misma longitud 35c⟩
      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 8.
Uses BlockMatrix 41, Matrix 9, and Vector 6b.

C.2 Códigos que verifican que los argumentos son correctos

35b ⟨*Verificación de que al instanciar* Matrix *el argumento* sis *es indexable* 35b⟩≡

```
  elif not isinstance(sis, (str, list, tuple)):
      raise ValueError(\
  '¡argumento: list (tuple) de Vectores (lists o tuples);  BlockMatrix; o Matrix!')
```
This code is used in chunk 35a.
Uses BlockMatrix 41 and Matrix 9.

35c ⟨*Verificación de que todas las filas de la matriz tendrán la misma longitud* 35c⟩≡

```
  if not all ( (type(sis[0])==type(v)) and (len(sis[0])==len(v)) for v in iter(sis) ):
      raise ValueError('no todas son listas o no tienen la misma longitud!')
```

This code is used in chunk 35a.

36a ⟨*Verificación de que todas las columnas de la matriz tendrán la misma longitud* 36a⟩≡
```
if not all ( isinstance(v, Vector) and (sis[0].n == v.n) for v in iter(sis)):
    raise ValueError('no todos son vectores, o no tienen la misma longitud!')
```

This code is used in chunk 8.
Uses `Vector` 6b.

C.3 Representación de la clase `Matrix`

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

36b ⟨*Representación de la clase* `Matrix` 36b⟩≡
```
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 9.

D Completando la clase `T`

D.1 Representación de la clase `T`

De nuevo construimos los dos métodos de presentación. Uno para la consola de comandos que escribe `T` y entre paréntesis la tupla (o conjunto) que representa la transformación elemental (o intercambio). Así,

- `T( {1, 5} )` : intercambio entre los vectores primero y quinto.
- `T( (2, 6) )` : multiplica el segundo vector por seis.
- `T( (3, 2, -1) )`: al tercer vector le suma el segundo multiplicado por -1.

La otra representación es para el entorno Jupyter y replica la notación usada en los apuntes de la asignatura:

Python	Representación en Jupyter
`T( {1, 5} )`	$\underset{[\mathbf{1}\rightleftharpoons\mathbf{5}]}{\boldsymbol{\tau}}$
`T( (2, 6) )`	$\underset{[6\cdot\mathbf{2}]}{\boldsymbol{\tau}}$
`T( (3, 2, -1) )`	$\underset{[\mathbf{3}-1\cdot\mathbf{2}]}{\boldsymbol{\tau}}$

Los apuntes de la asignatura usan una notación matricial, y por tanto es una notación que discrimina entre operaciones sobre las filas o las columnas, situando los operadores a la izquierda o a la derecha de la matriz. En este sentido, nuestra notación en Python hace lo mismo. Así, en la siguiente tabla, la columna de la izquierda corresponde a operaciones sobre las filas, y la columna de la derecha a las operaciones sobre las columnas:

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

D.1.1 Secuencias de transformaciones

Imagine que quiere

1. primero multiplicar el primer vector por 2, es decir, aplicar: `T( (1, 2) )`
2. después intercambiar el tercer vector por cuarto, es decir, aplicar: `T( {3, 4} )`.

Vamos a indicar dicha secuencia de transformaciones con una lista de tuplas y conjuntos: `[(1, 2), {3, 4}]`; donde el orden de la lista indica el orden en el que se aplican las transformaciones elementales y/o intercambios. De esta manera, cuando componemos ambas operaciones: `T( (1,2) ) & T( {3, 4} )`, nuestra librería nos devuelve la trasformación composición de las dos operaciones **en el orden en el que han sido escritas**:

`T( [ (1, 2), {3, 4} ] )` es equivalente a escribir `T( (1,2) ) & T( {3, 4} )`

Por tanto, si queremos realizar dichas operaciones sobre las columnas de la matriz $\mathbf{A}$, podemos hacerlo de dos formas: de una en una (escribiendo las operaciones en el mismo orden en que queremos que se ejecuten y a la derecha de la matriz), o aplicando la transformación composición (descrita con la lista de operaciones en el orden en que queremos que se ejecuten) a la derecha de la matriz.

- `A & T( (1,2) ) & T( {3, 4} )` (indicando las transformaciones de una en una)
- `A & T( [(1, 2), {3, 4}] )` (usando la transformación composición de todas ellas)

y si queremos operar sobre la filas hacemos exactamente igual, pero a la izquierda de la matriz

- `T( (1,2) ) & T( {3, 4} ) & A`
- `T( [(1, 2), {3, 4}] ) & A`

Representación en Jypyter de una secuencia de transformaciones. En los apuntes de la asignatura se usa una notación matricial (que discrimina entre operaciones sobre las filas o columnas situando los operadores a la izquierda o a la derecha de la matriz). Pero con dicha notación el órden en que actuan las tranformaciones depende de lo cerca que están de la matriz (y no que operación aparece más a la izquierda). Por ejemplo, la sucesión de más arriba actuando sobre las columnas de $\mathbf{A}$ (donde primero se multiplica la primera columna por dos, y luego se intercambian las columnas tres y cuatro) se escribe así

$$\Big(\mathbf{A}\underset{[2\cdot\mathbf{1}]}{\tau}\Big)\underset{[\mathbf{3}\rightleftharpoons\mathbf{4}]}{\tau} = \mathbf{A}\underset{[2\cdot\mathbf{1}]}{\tau}\underset{[\mathbf{3}\rightleftharpoons\mathbf{4}]}{\tau}.$$

Pero al operar por filas, la transformación más próxima a la matriz es la que aparece más a la derecha, es decir, si primero se multiplica la primera fila por dos, y luego se intercambian las filas tres y cuatro, la escritura queda así:

$$\underset{[\mathbf{3}\rightleftharpoons\mathbf{4}]}{\tau}\Big(\underset{[2\cdot\mathbf{1}]}{\tau}\mathbf{A}\Big) = \underset{[\mathbf{3}\rightleftharpoons\mathbf{4}]}{\tau}\underset{[2\cdot\mathbf{1}]}{\tau}\mathbf{A}.$$

¡Qué lio! la sucesión de transformaciones `T( [(1, 2), {3, 4}] )` se escribe como $\underset{[2\cdot\mathbf{1}]}{\tau}\underset{[\mathbf{3}\rightleftharpoons\mathbf{4}]}{\tau}$ si se opera sobre las columnas, y como $\underset{[\mathbf{3}\rightleftharpoons\mathbf{4}]}{\tau}\underset{[2\cdot\mathbf{1}]}{\tau}$ si se opera sobre las filas. Una sencilla solución es escribir las operaciones en vertical, de manera que actuan antes las operaciones que están más arriba. Así, la representación en vertical permite escribir los símbolos en el mismo orden tanto por el lado izquierdo de la matriz (filas) como por el derecho (columnas):

`T( [(1, 2), {3, 4}] ) & A` $= \underset{\begin{array}{c}\left[2\cdot\mathbf{1}\right]\\ \left[\mathbf{3}\rightleftharpoons\mathbf{4}\right]\end{array}}{\boldsymbol{\tau}}\mathbf{A};$ `A & T( [(1, 2), {3, 4}] )` $= \mathbf{A}\underset{\begin{array}{c}\left[2\cdot\mathbf{1}\right]\\ \left[\mathbf{3}\rightleftharpoons\mathbf{4}\right]\end{array}}{\boldsymbol{\tau}}.$

Por si fuera necesario representar las transformaciones en formato horizontal, he implementado la representación `rpr='H'` que escribe las operaciones en horizontal para el caso de actuar sobre las columnas (por la derecha), y la representación `rpr='Hfilas'` para el caso de la actuación sobre las filas (por la izquierda). Por defecto, la representación será vertical.

Python	Representación en Jupyter por defecto
`T( [(1, 2), 3, 4]`	$\underset{\begin{array}{c}\left[2\cdot\mathbf{1}\right]\\ \left[\mathbf{3}\rightleftharpoons\mathbf{4}\right]\end{array}}{\boldsymbol{\tau}}$

38 ⟨*Representación de la clase* T 38⟩≡

```
def __repr__(self):
    """ Muestra T en su representación python """
    return 'T(' + repr(self.t) + ')'

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

def latex(self):
    """ Construye el comando LaTeX """
    def signo(v):
        """Escribe '-' si el argumento es negativo y '+' en el resto de casos"""
        return '-' if v<0 else '+'

    def simbolo(t):
        """Escribe el símbolo que denota una trasformación elemental particular"""
        if isinstance(t,set):
            return '\\left[\\mathbf{' + latex(min(t)) + \
              '}\\rightleftharpoons\\mathbf{' + latex(max(t)) + '}\\right]'
        if isinstance(t,tuple) and len(t) == 2:
            return '\\left[' + \
              latex(t[1]) + '\\cdot\\mathbf{'+ latex(t[0]) + '}\\right]'
        if isinstance(t,tuple) and len(t) == 3:
            return '\\left[\\mathbf{' + latex(t[0]) + '}' + signo(t[2]) + \
              latex(t[2]) + '\\cdot\\mathbf{' + latex(t[1]) + '}\\right]'

    if isinstance(self.t, (set, tuple) ):
        return '\\underset{' + simbolo(self.t) + '}{\\mathbf{\\tau}}'
    elif isinstance(self.t, list):
        if self.rpr == 'H':
            return '\\,'.join([latex(T(i, rpr=self.rpr)) for i in self.t])
        if self.rpr == 'Hfilas':
            return '\\,'.join([latex(T(i, rpr=self.rpr)) for i in reversed(self.t)])
        else:
            return '\\underset{\\begin{array}{c}' + \
              '\\\\'.join([simbolo(i) for i in self.t]) + \
              '\\end{array}}{\\mathbf{\\tau}}'
```

This code is used in chunk 28c.
Uses T 28c.

E 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):

`V0` es una subclase de `Vector` (por tanto hereda los atributos de la clase `Vector`), pero el código inicia (y devuelve) un objeto de su superclase, es decir, inicia y devuelve un `Vector`.

39a ⟨*Definición de vector nulo:* `V0` 39a⟩≡

```
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 31.
Defines:
`V0`, used in chunks 24a and 39b.
Uses `Vector` 6b.

Y lo mismo hacemos para matrices

39b ⟨*Definición de matriz nula:* `M0` 39b⟩≡

```
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__([ V0(m) for j in range(n)])
```

This code is used in chunk 31.
Uses `Matrix` 9 and `V0` 39a.

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

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

40 ⟨*Definición de la matriz identidad:* `I` 40⟩≡

```
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 31.
Uses `Matrix` 9.

F La clase `BlockMatrix`. Matrices particionadas (o matrices por bloques)

Las matrices particionadas no son tan importantes para seguir el curso, aunque si se usan en esta 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 esta sección****: el modo de particionar una matriz es sencillo y se puede aprender rápidamente con el siguiente Notebook*

Tutorial previo en un Jupyter notebook

Este Notebook es un ejemplo 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 $\boxed{\mathbf{A}}$ 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 `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).

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.

41 ⟨*Definición de la clase* BlockMatrix 41⟩≡
```
class BlockMatrix:
    def __init__(self, sis):
        """Inicializa una BlockMatrix con 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]]

    ⟨Repartición de las columnas de una BlockMatrix 43a⟩
    ⟨Repartición de las filas de una BlockMatrix 43b⟩
    ⟨Representación de la clase BlockMatrix 45⟩
```

This code is used in chunk 31.
Defines:
BlockMatrix, used in chunks 7, 12c, 15, 32b, 35, 42, and 43.

F.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 ${}_{\{i_1,\dots,i_r\}|}\mathbf{A}$ es la matriz de bloques

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

[a] Falta 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);

42a ⟨*Definición del método* `partición` 42a⟩≡
```
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 31.

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

42b ⟨*Partición de una matriz por filas de bloques* 42b⟩≡
```
elif isinstance(i,set):
    return BlockMatrix ([ [a|self] for a in particion(i,self.m) ])
```
This code is used in chunk 16.
Uses `BlockMatrix` 41.

42c ⟨*Partición de una matriz por columnas de bloques* 42c⟩≡
```
elif isinstance(j,set):
    return BlockMatrix ([ [self|a for a in particion(j,self.n)] ])
```
This code is used in chunk 13.
Uses `BlockMatrix` 41.

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 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 A entonces

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

es decir, queremos poder particionar una BlockMatrix. Los casos interesantes son cuando particionamos por el lado contrario por el que se particionó la matriz de partida, es decir,

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

que, por supuesto, debe dar el mismo resultado. Para dichos casos, programamos el siguiente código que particiona una BlockMatrix. Primero el procedimiento para particionar por columnas cuando hay una única columna de matrices (el caso general se verá un poco más abajo):

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

43a ⟨*Repartición de las columnas de una* `BlockMatrix` 43a⟩≡

```
  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 partición(j,self.lista[0][0].n)] \
                                    for i in range(self.m) ])

          ⟨Caso general de repartición por columnas 44b⟩
```

This code is used in chunk 41.
Uses `BlockMatrix` 41.

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

43b ⟨*Repartición de las filas de una* `BlockMatrix` 43b⟩≡

```
      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 repartición por filas 44c⟩
```

This code is used in chunk 41.
Uses `BlockMatrix` 41.

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:

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

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

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

44a ⟨*Definición del procedimiento de generación del conjunto clave para particionar* 44a⟩≡

```
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 31.

Así, los casos generales consisten en reparticionar de nuevo:

44b ⟨*Caso general de repartición por columnas* 44b⟩≡

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

This code is used in chunk 43a.
Uses `Matrix` 9.

44c ⟨*Caso general de repartición por filas* 44c⟩≡

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

This code is used in chunk 43b.
Uses `Matrix` 9.

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`.

F.2 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 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.

45 ⟨*Representación de la clase* `BlockMatrix` 45⟩≡

```
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 41.

G Secciones de código

Licencia

47 ⟨*Copyright y licencia GPL* 47⟩≡

```
# Copyright (C) 2019  Marcos 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/
```

Root chunk (not used in this document).

Part III
Sobre este documento

Con ánimo de que esta documentación sea más didáctica, en la Parte I muestro las partes más didácticas del código, y relego las otras a la Parte II. Así puedo destacar cómo la librería de Python es una implementación literal de las definiciones dadas en mis notas de la asignatura de Mates II. Para lograr presentar el código en un orden distinto del que realmente tiene en la librería uso la herramienta noweb. Una breve explicación aparece en la siguiente sección...

Literate programming con noweb

Este documento está escrito usando noweb. Es una herramienta que permite escribir a la vez tanto código como su documentación. El código se escribe a trozos o "chunks" como por ejemplo este:

48a
```
⟨Chunk de ejemplo que define la lista a 48a⟩≡
  a = ["Matemáticas II es mi asignatura preferida", "Python mola", 1492, "Noweb"]
```
This code is used in chunk 48c.

y este otro chunk:

48b
```
⟨Segundo chunk de ejemplo que cambia el último elemento de la lista a 48b⟩≡
  a[-1] = 10
```
This code is used in chunk 48c.

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

48c
```
⟨EjemploLiterateProgramming.py 48c⟩≡

  ⟨Chunk de ejemplo que define la lista a 48a⟩
  ⟨Segundo chunk de ejemplo que cambia el último elemento de la lista a 48b⟩

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

  ⟨Chunk final que indica qué tipo de objeto es a y hace unas sumas 50⟩
```
Root chunk (not used in this document).

Este modo de escribir el código permite destacar unas partes 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. Como el nombre de dichos chunks es auto-explicativo, mirando el recuadro anterior es fácil hacerse una idea de que hace el programa "`EjemploLiterateProgramming.py`" en su conjunto.

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 50 de este documento.

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

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

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

type(a)

2+2

10*3+20
```

Último chunk del ejemplo de Literate Programming

Este es uno de los trozos de código del ejemplo.

50 ⟨*Chunk final que indica qué tipo de objeto es* `a` *y hace unas sumas* 50⟩≡

```
type(a)

2+2

10*3+20
```

This code is used in chunk 48c.