De MATLAB a Python

Jorge De Los Santos

Versión 0.1

Contenido

Acerca de...

Estos pequeños apuntes han sido creados con la finalidad de proporcionar una herramienta didáctica que facilite la *transición* de MATLAB a Python. Ha de mencionarse que este texto no constituye una introducción formal al lenguaje Python, sino más bien algo próximo a un *How to...*

1. Los primeros pasos

¿Qué con MATLAB?

Para comenzar, este texto no tiene la finalidad de "despotricar" contra nadie, así que vamos a ser *justos* con los *pros* y *contras* de cada lenguaje. MATLAB es un lenguaje de programación de alto nivel, interpretado, multiparadigma*, multiplataforma, de tipado dinámico, entre otras características, que le han convertido con el paso de los años en un estándar para aplicaciones de ingeniería y del ámbito científico, y más aún en el aspecto académico.

Pero ha existido desde siempre cierta disyuntiva en lo que respecta a los costos de las licencias, que generalmente suelen ser altos. Guardando las distancias, MathWorks es algo así como un Microsoft en lo que respecta a la forma en que se manejan. Y claro, no sólo eso, sino también cuestiones relacionadas con la distribución de aplicaciones para usuarios finales, que a más de uno le ha representado complicaciones. Además suele ser poco recomendable para el desarrollo de grandes proyectos. En cuanto a la rapidez de ejecución, MATLAB dada su condición de lenguaje interpretado suele ser considerado como "un poco lento", pero no obstante, para la mayoría de situaciones puede arreglarse mediante la capacidad de agregar librerías externas escritas en lenguajes compilados como C y Fortran.

¿Por qué Python?

Python es un lenguaje desarrollado a finales de los ochenta y cuyas primeras versiones aparecieron al principio de los noventa. Es un lenguaje interpretado, de tipado dinámico, multiplataforma, multiparadigma, con una sintaxis sencilla y distribuido gratuitamente, siendo además de código abierto, lo cual ha permitido que detrás del proyecto exista una cantidad enorme de usuarios que contribuyen con el mismo.

La diferencia y ventaja más marcada de Python respecto a MATLAB es su condición de lenguaje de propósito general, lo que permite una flexibilidad considerable en el desarrollo de proyectos amplios.

Python se distribuye con una librería estándar, que por sí misma no proporciona las herramientas para el desarrollo de aplicaciones en el

ámbito cientifico / ingeniería. Por lo cual es necesario instalar de forma independiente dichas librerías, y de las cuales se hablará en la siguiente sección.

¿Qué necesitamos?

Desde luego tener Python instalado, para ello puede dirigirse a la página de descargas (https://www.python.org/downloads/) y seguir los procedimientos que ahí se indican de acuerdo al sistema operativo que disponga.

El tema de las versiones 2.x y 3.x de Python es un punto de "discusión" muy común, lo cierto es que sería más recomendable utilizar una versión 3.x, pero en este texto se usará la versión 2.7. Pero claro, la elección es solamente del lector. Remarcando el hecho que deberá utilizar las librerías compatibles con la versión de Python instalada.

Ahora, existen múltiples librerías para Python destinadas al computo científico, especializadas en menor o mayor grado, pero para comenzar la transición MATLAB-Python existen, a criterio del autor, tres librerías esenciales, a saber:

- NumPy
- SciPy
- Matplotlib

Enseguida una breve descripción de las librerías listadas.

NumPy

Pagina del proyecto: http://www.numpy.org

NumPy es una biblioteca extensa que provee los objetos numéricos bases para trabajar con vectores y matrices de forma eficiente.

SciPy

Página del proyecto: http://www.scipy.org

SciPy es una biblioteca que contiene múltiples algoritmos para aplicaciones en ingeniería, matemáticas y todo el ámbito científico. Utiliza NumPy como base.

Matplotlib

Página del proyecto: http://matplotlib.org

Matplotlib es una librería que proporciona herramientas de visualización gráfica en 2D de gran calidad, y una capacidad aceptable para 3D. Es muy notorio también que en cierta medida está inspirada en MATLAB, teniendo una sintaxis muy similar, lo cual ayudará mucho en esta *transición*.

Para instalar las librerías puede acceder a las páginas de cada proyecto y proceder conforme se especifica en la sección de descarga correspondiente. O bien utilizar herramientas de instalación para módulos de Python como `pip` o `easy_install`.

2. Los tipos de datos

Tipos numéricos

Enteros

En MATLAB existen enteros de 8, 16, 32 y 64 bits, que para declararlos como tal hace falta una conversión explícita utilizando las funciones `int8`, `int16`, `int32`, e `int64`.

Python proporciona dos tipos de enteros: los tipos `int` y `long`. Generalmente int es un entero de 32 bits y long un entero de 64 bits (sujeto al equipo en que se esté ejecutando Python).

Puede consultar el máximo entero soportado ejecutando las siguientes instrucciones en el intérprete de Python:

```
>>> import sys
>>> max_int = sys.maxint
>>> max_long = sys.maxsize
```

De coma flotante

MATLAB dispone de dos tipos de datos de coma flotante: `double` y `single`, de doble y simple precisión respectivamente. El tipo double no hace falta declararlo explícitamente, dado que todo tipo numérico en MATLAB se considera de tipo double, a menos que se especifique lo contrario.

En Python existe únicamente el tipo `float`. Para que un número sea considerado de coma flotante debe utilizarse la función `float`, o bien, colocar la parte decimal del número, aún cuando esta sea nula, por ejemplo:

```
>>> a=3.0
>>> b=float(3)
>>> c=7.5
>>> type(a)
<type 'float'>
>>> type(b)
<type 'float'>
>>> type(c)
<type 'float'>
```

Números complejos

En MATLAB los números complejos no son estrictamente un tipo de dato, si no que son considerados como tipos `double`, diferenciándolos mediante la característica *attributes*, por ejemplo:

```
>> whos(complex(2,3))
```

En Python, de igual manera se usa la función `complex` para definir un número complejo, pero aquí si que retorna un objeto de tipo `complex`, y desde luego es considerado un *built-in type*:

```
>>> type(complex(2,3))
<type 'complex'>
```

Tipo booleano

La siguiente tabla resume las diferencias en ambos lenguajes:

MATLAB	Python
Constantes	
`true`	`True`
`false`	`False`
Funciones	
logical(n)	bool(n)

Cadenas de caracteres

En Python las cadenas de caracteres pueden crearse utilizando las comillas dobles o simples, por ejemplo:

```
>>> cad1='Esto es una cadena de caracteres'
>>> cad2="Y esto también"
>>> type(cad1)
<type 'str'>
>>> type(cad2)
<type 'str'>
```

En MATLAB, como sabemos, la única forma es utilizando las comillas simples.

Estructuras de datos

3. Estructuras de control

Sentencia if

En MATLAB la sentencia if múltiple tiene la siguiente estructura:

```
if cond1
    % ...
elseif cond2
    % ...
    % ...
    % ...
elseif condN
    % ...
else
    % ...
end
```

Algo similar ocurre con Python, con la diferencia que se deben colocar dos puntos después de cada instrucción, que la identación obviamente es obligatoria, el `elseif` se "acorta" a `elif` además que en Python no se coloca la instrucción `end` para indicar el final del bloque:

```
if cond1:
    # ...
elif cond2:
    # ...
    # ...
    # ...
elif condN:
    # ...
else:
    # ...
```

Veamos un ejemplo concreto reproducido en ambos lenguajes:

```
% Ejemplo MATLAB
n = input('Inserte un número: ');
if n > 0
    disp('Positivo');
elseif n < 0
    disp('Negativo');
else
```

```
    disp('Cero');
end
```

```
# Ejemplo python
n = input('Inserte un número: ')
if n > 0:
    print "Positivo"
elif n < 0:
    print "Negativo"
else:
    print "Cero"
```

Sentencia switch

La sentencia switch de MATLAB, característica de lenguajes inspirados por la sintaxis de C, no tiene un equivalente *directo* en Python, pero puede emularse para algunos casos utilizando los diccionarios.

Por ejemplo, suponga que quiere desarrollar un programa en el cual ingrese dos números y enseguida seleccione la operación arimética que realizará con ellos, en MATLAB y utilizando switch sería algo como:

```
a = input('a = ');
b = input('b = ');
oper = input('Operación a realizar \n\n1. Suma \n2.Resta \n3.Multiplicación \n4.División\n\n');
switch oper
    case 1
        r = a+b;
    case 2
        r = a-b;
    case 3
        r = a*b;
    case 4
        r = a/b;
end
fprintf('Resultado = %g', r);
```

En Python un código "equivalente" utilizando diccionarios sería:

```
a = input('a = ')
b = input('b = ')
oper = raw_input('Operación a realizar \n\n1. Suma \n2.Resta \n3.Multiplicación \n4.División\n\n')
opciones = {'1':a+b,'2':a-b,'3':a*b,'4':a/b}
resultado = opciones.get(oper)
if resultado is not None:
    print "Resultado = %g"%(resultado)
else:
    print "Opción incorrecta"
```

Bucle for

En este bucle MATLAB y Python guardan cierta similitud: ambos recorren un arreglo u objeto iterable, que puede ser una lista, tupla (Python) o un array (MATLAB).

En MATLAB es común hacer lo siguiente al utilizar un ciclo for:

```
for i = 1:10
    % ... código útil
    % ... algo más de código
end
```

O bien:

```
for k = 1:0.2:100
    % ...
    % ...
end
```

Pero también es posible hacerlo de cualquiera de las siguientes maneras:

```
str = 'hola';
celda = {1,2,3,4,5,10,2,-2};
mat = rand(4);

for ii = str
    disp(ii); % Imprime cada caracter de str
end

for jj = celda
    disp(jj); % Imprime cada componente de celda
end

for kk = mat
    disp(kk); % Imprime cada elemento de la matriz aleatoria mat
end
```

En Python funciona de manera similar, con una clara variación de sintaxis. Un ejemplo muy básico es:

```
for x in range(10):
    print x
```

Que imprime lo siguiente:

```
0
1
2
3
4
5
6
7
8
9
```

Note que en lugar del signo igual (=) de MATLAB, Python utiliza el nexo **in**, y claro, los dos puntos que se muestran al finalizar la expresión inicial son obligatorios y necesarios como en el caso de la sentencia `if`, además de que la instrucción `end` de MATLAB no se presenta en Python. Es importante mencionar también que la función `range` devuelve una lista de enteros en el intervalo semi-abierto [0,N), es decir, desde cero hasta N, excluyendo a este último.

Pero, como se ha mencionado, con el bucle `for` se puede recorrer cualquier objeto iterable de Python. Por ejemplo, para recorrer una cadena de caracteres:

```
for letra in "hola mundo":
    print letra
```

```
h
o
l
a

m
u
n
d
o
```

O incluso imprimir un clásico triángulo de caracteres en pantalla:

```
car = "*"
for n in xrange(1,11):
    print n*car
```

```
*
**
***
****
*****
******
*******
********
*********
**********
```

De lo anterior, mención *especial* merece la función `xrange`, que es muy similar a `range` (de hecho la sintaxis es la misma), pero que en lugar de devolver una lista de enteros, devuelve un objeto iterable, lo cual le hace más eficiente en cuestión de tiempo y memoria. Algo que quizá por ahora no resulte tan crítico, pero que muy posiblemente le sea útil posteriormente.

Bucle while

El bucle `while` sigue una *filosofía* similar en ambos lenguajes, difiriendo simplemente en los dos puntos e indentación de Python y la terminación `end` de MATLAB, véase el ejemplo siguiente:

En MATLAB:

```
k = 0;
while k < 10
    disp(k);
    k = k + 1;
end
```

En Python:

```
k = 0
while k < 10:
    print k
    k += 1
```

4. Funciones

Definiendo funciones

Como sabrá, las funciones en MATLAB deben definirse una a la vez por cada fichero y tener el nombre de este. Aunque pueden incluirse otras funciones dentro del mismo fichero, estas serán *subfunciones* o funciones auxiliares de la principal, implicando que sólo resulten "visibles" y utilizables en el ámbito de la función principal. La sintaxis más general de una función MATLAB sería:

```
function [out1, out2,...] = nfun(arg1, arg2, ...)
% ...
% ... cuerpo de la función
% ...
end
```

Donde `nfun` es el nombre que se la ha dado a la función; out1 y out2 son valores de retorno de la función; y arg1 y arg2 son argumentos de entrada formales de la función `nfun`. Evidentemente los puntos suspensivos indican que los parámetros de salida y/o entrada podrían ser más o menos que los mostrados como ejemplo.

En Python se pueden definir múltiples funciones en un mismo fichero, lo cual se traduce en una enorme ventaja, y no sólo puede definir funciones varias, sino también definiciones de clases, ese es el *plus* de Python: la gran capacidad de modularización y organización del código sin tantas restricciones.

Para definir una función en Python se utiliza la palabra reservada `def` seguida del nombre de la función, tal como se indica a continuación:

```
def nfun(arg1,arg2,...):
    # ...
    # Cuerpo de la función
    # ...
    return out1,out2,...
```

Note que en Python se utiliza la instrucción `return` para indicar los valores de retorno de la función.

Utilizando funciones

Vamos a poner en práctica la definición de funciones, para ello se creará una función llamada `es_par` que identificará si un número es par, se devolverá un valor lógico verdadero en caso de ser así o falso en caso contrario.

En MATLAB:

```
function r =  es_par(n)
if ~rem(n,2)
        r = true;
else
        r = false;
end
end
```

En Python:

```
def es_par(n):
    if not(n%2):
        r = True
    else:
        r = False
    return r
```

La forma de llamar la función es exactamente la misma:

```
val = es_par(7)
```

Donde val se espera sea un valor lógico falso para este caso.

No obstante si los valores de retorno son múltiples, entonces existe una variación en la forma de llamar a la función, a saber:

```
>> [a,b,c] = fun(x,y,z) % Para MATLAB

>>> a,b,c = fun(x,y,z) # Para Python
```

Funciones con argumentos variables

Tanto MATLAB como Python tienen la capacidad de soportar funciones con argumentos de entrada variables. En MATLAB se utiliza `varargin` para indicar que una función puede recibir un número indefinido de argumentos, por ejemplo:

```
function [a,b,...] = nfun(varargin)
% ...
% Cuerpo de la función
% ...
end
```

En Python la notación utilizada para ello es anteponer un asterisco (*) al nombre del parámetro formal:

```
def nfun(*args):
    # ...
    # Cuerpo de la función
    # ...
    return a,b,...
```

El concepto manejado es muy similar: todos los argumentos de entrada se guardan en un arreglo, un cell array en el caso de MATLAB y una tupla en Python, de modo que el programador definirá en el cuerpo de la función los criterios que habrán de seguirse para utilizar cada uno de los valores de entrada. Para ejemplificar mejor este proceso vamos a crear una función cuyo nombre será `maximo`, y que pueda recibir como argumento un arreglo de valores o bien dos números reales cualesquiera.

En MATLAB:

```
function m = maximo(varargin)
if length(varargin)==1
    v = varargin{1};
    m = v(1);
    for i = 2:length(v)
        if v(i)>m
            m = v(i);
        end
    end
elseif length(varargin)==2
    m = varargin{1};
```

```
    a = varargin{2};
    if a > m
        m = a;
    end
else
    error('Número de argumentos de entrada inválidos');
end
end
```

En Python:

```
def maximo(*args):
    if len(args)==1:
        v = args[0]
        m = v[0]
        for i in v:
            if i > m:
                m = i
    elif len(args)==2:
        m = args[0]
        a = args[1]
        if a > m:
            m = a
    else:
        raise SyntaxError
    return m
```

Una diferencia notoria en los ejemplos anteriores es la manera de acceder a los elementos de un arreglo, mientras en MATLAB se utilizan paréntesis y llaves, en Python se utiliza el corchete. Además, en MATLAB los índices de un arreglo comienzan en uno, no así en Python que sigue la convención de la mayoría de los lenguajes y comienza en cero.

Un plus en Python: keyword arguments

Las funciones en Python permiten el uso de *keyword arguments*. Para mostrar esta característica vamos a definir una función `info_contacto` que mostrará en pantalla información de un contacto (nombre, dirección, ...):

```
def info_contacto(**kwargs):
    nombre = kwargs.get('nombre')
    direccion = kwargs.get('direccion')
```

```
    telefono = kwargs.get('telefono')
    email = kwargs.get('email')
    info = """
    Nombre : {}
    Dirección : {}
    Teléfono : {}
    E-mail : {}
    """.format(nombre,direccion,telefono,email)
    print info
```

Revisemos un poco: para indicar que se utilizarán *keyword arguments* es necesario anteceder el nombre del parámetro con dos asteriscos (**nombre_parametro), desde luego el nombre del parámetro o argumento puede ser cualquiera (evitando, claro, las palabras reservadas del lenguaje). Los parámetros de la función son pasados como se indica a continuación:

```
funcion(arg1=valor1, arg2=valor2, arg3=valor3, ...)
```

Es decir, el argumento real asignado al formal. Todos los argumentos que se pasan a la función son "almacenados" en un diccionario en la forma:

```
kwargs = {'arg1':valor1, 'arg2':valor2, 'arg3':valor3, ...}
```

Por lo cual para acceder al valor de un determinado argumento puede utilizarse el método `get` del diccionario, por ejemplo:

```
valor1 = kwargs.get('arg1')
valor2 = kwargs.get('arg2')
valor3 = kwargs.get('arg3')
...
```

La función `info_contacto` definida anteriormente podríamos ejecutarla como sigue:

```
info_contacto(nombre="Ana", direccion="Av. Siempreviva 742", telefono="1892312333", email="ana@gmail.com")
```

Y obtener en pantalla:

```
Nombre : Ana
Dirección : Av. Siempreviva 742
```

```
Teléfono : 1892312333
E-mail : ana@gmail.com
```

Es importante mencionar que los argumentos pasados por clave-valor pueden ir en cualquier orden, de modo que en la función anterior pudo haberse colocado primero la dirección o cualquier otro argumento, sin afectar el resultado.

Valores por defecto de un argumento

Otra posibilidad que brinda Python es colocar un valor por defecto al argumento de una función, esto mediante la notación *keyword arguments*. Véase el siguiente ejemplo:

```
def tri(n, car="*"):
    for x in range(1,n+1):
        print x*car
```

La función `tri` imprime en pantalla un triángulo de caracteres de base **n**, utilizando asteriscos como caracteres por defecto. Por ejemplo:

```
>>> tri(5)
*
**
***
****
*****
```

Es decir, se puede omitir el segundo argumento y entonces Python tomará el que se le ha indicado en la forma clave-valor. Pero si insertamos el segundo argumento se "desechará" el valor por defecto y se tomará el valor pasado por el usuario:

```
>>> tri(7, "o")
o
oo
ooo
oooo
ooooo
oooooo
ooooooo
```

5. Los módulos en Python

¿Qué es un módulo?

En términos informales un módulo es un fichero con extensión .py o bien un conjunto de ficheros, que contienen definiciones de clases, funciones y constantes.

Los módulos constituyen la base de la programación en Python (...)

Importar y utilizar módulos

Normalmente cuando se inicia el entorno de MATLAB, se tienen disponibles todas las funciones incluidas en los Toolboxs que se tengan instalados, sin necesidad de importar librerías o paquetes, esto puede considerarse una ventaja, pero también una enorme desventaja, algo paradójico. Vamos, que si necesita utilizar algunas funciones matemáticas elementales, no hace falta más que escribirlas, por ejemplo:

```
% Algunas funciones matemáticas elementales en MATLAB
>> exp(1)
>> cos(0)
>> theta = pi/6
>> cos(theta)^2+sin(theta)^2
```

No obstante, en Python, cuando se inicia el interpréte no se tienen disponibles todas estas funciones, sino sólo algunas que son consideradas como *built-in functions*. Para utilizar cualquier otro tipo de funciones hace falta importar el módulo en el cual se encuentren estas. Por ejemplo para ejecutar el ejemplo anterior en Python habría que importar el módulo **math** tal cómo se indica a continuación:

```
>>> from math import *
>>> exp(1)
2.718281828459045
>>> cos(0)
1.0
>>> theta=pi/6
>>> cos(theta)**2+sin(theta)**2
1.0
```

Si no se hubiese importado el módulo **math**, Python lanza un error de tipo *NameError*, indicando que una función o variable no está definida:

```
>>> exp(1)

Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    exp(1)
NameError: name 'exp' is not defined
```

Existen, generalmente, tres formas de importar un módulo, a saber:

```
1) import modulo
2) import modulo as md
3) from modulo import algo
```

Primer forma

La primer forma es la más común y, sobre todo, recomendable, de importar un módulo. Se coloca la palabra clave `import` seguida del nombre del módulo que habrá de importarse. Vamos a repetir el ejemplo anterior de las funciones matemáticas:

```
>>> import math
>>> math.exp(1)
2.718281828459045
>>> math.cos(0)
1.0
>>> theta=math.pi/6
>>> math.cos(theta)**2+math.sin(theta)**2
1.0
```

Cómo habrá notado, es necesario anteponer a cada función utilizada el nombre del módulo al cual pertenecen, en este caso **math**. La ventaja de este tipo de notación es que evita cometer errores cuando se tienen dos funciones o clases con el mismo nombre, provenientes de módulos diferentes.

Segunda forma

6. Vectores y matrices

En este capítulo abordaremos el uso del módulo NumPy para el manejo de arreglos numéricos (matrices y vectores), de modo que aún cuando no se especifique de forma explícita, se asumirá que se ha importado la librería NumPy utilizando el alias (pseudónimo) `np`, tal como se indica enseguida:

```
>>> import numpy as np
```

Definiendo un vector

Tomaremos el vector **v=<3,-1,8>** como ejemplo, así en MATLAB para crear dicho vector sería:

```
>> v = [3,-1,8];
```

Para Python utilizaremos la función `array` del módulo NumPy, pasando como argumentos una lista con los elementos que conforman el vector:

```
>>> v = np.array([3,-1,8])
>>> type(v)
<type 'numpy.ndarray'>
```

Podría utilizarse también la función `range`, por ejemplo:

```
>>> v = np.array(range(100))
>>> len(v)
100
```

O una lista por comprensión:

```
>>> v=[x*0.1 for x in range(11)]
>>> for elemento in v:
            print elemento


0.0
0.1
0.2
0.3
```

```
0.4
0.5
0.6
0.7
0.8
0.9
1.0
```

Algunos vectores predefinidos

Al igual que MATLAB, en Python/NumPy se tienen funciones que generan vectores numéricos predefinidos y que resultan muy útiles. Ejemplo de ello es `linspace`, que genera un arreglo de puntos linealmente equiespaciados en un intervalo, la cantidad de puntos es modificable, siendo 50 por defecto, por ejemplo, para generar un arreglo numérico de 0 a 10 con 50 puntos sería:

```
>>> v = np.linspace(0,10)
>>> print v
[  0.          0.20408163   0.40816327   0.6122449    0.81632653
   1.02040816   1.2244898    1.42857143   1.63265306   1.83673469
   2.04081633   2.24489796   2.44897959   2.65306122   2.85714286
   3.06122449   3.26530612   3.46938776   3.67346939   3.87755102
   4.08163265   4.28571429   4.48979592   4.69387755   4.89795918
   5.10204082   5.30612245   5.51020408   5.71428571   5.91836735
   6.12244898   6.32653061   6.53061224   6.73469388   6.93877551
   7.14285714   7.34693878   7.55102041   7.75510204   7.95918367
   8.16326531   8.36734694   8.57142857   8.7755102    8.97959184
   9.18367347   9.3877551    9.59183673   9.79591837  10.        ]
```

O si requiere una cantidad de puntos menores:

```
>>> v = np.linspace(1,5,5)
>>> print v
[ 1.  2.  3.  4.  5.]
```

Incluso puede crear un vector en orden decreciente:

```
>>> v = np.linspace(10,0,11)
>>> print v
[ 10.   9.   8.   7.   6.   5.   4.   3.   2.   1.   0.]
```

Otra de esas funciones es `logspace`, la cual crea un vector logarítmicamente espaciado, siendo la sintaxis:

```
>>> np.logspace(A,B,N)
```

Lo anterior define un vector de N puntos en el intervalo `10eA` a `10eB`.

Por ejemplo:

```
>>> v = np.logspace(1,10,10)
>>> print v
[  1.00000000e+01   1.00000000e+02   1.00000000e+03   1.00000000e+04
   1.00000000e+05   1.00000000e+06   1.00000000e+07   1.00000000e+08
   1.00000000e+09   1.00000000e+10]
```

Además de las anteriores, también existe la función `arange` que trabaja de forma similar a `linspace`, excepto que en vez de especificarse el número de puntos, se especifica el "paso" entre cada elemento, siendo la sintaxis:

```
>>> np.arange(a,b,paso)
```

Por ejemplo:

```
>>> v = np.arange(1,2,0.25)
>>> print v
[ 1.    1.25  1.5   1.75]
```

Definiendo una matriz

Suponga que requiere crear la matriz siguiente:

Utilizando MATLAB:

```
>> A = [1,5,2;-3,1,7;0,4,8];
```

Para Python usaremos la función `matrix`, que deberá recibir como argumento una lista de listas, donde cada sub-lista es una fila de la matriz.

```
>>> A=np.matrix([[1,5,2],[-3,1,7],[0,4,8]])
>>> print A
[[ 1  5  2]
 [-3  1  7]
 [ 0  4  8]]
```

Además de la forma anterior, es posible definir matrices "especiales" utilizando funciones predefinidas del módulo NumPy, por ejemplo una matriz aleatoria:

```
>>> np.random.random((3,3))
array([[ 0.02477974,  0.21974255,  0.25890696],
       [ 0.13803954,  0.6985599 ,  0.55097814],
       [ 0.0294198 ,  0.80517378,  0.688599  ]])
```

Una matriz conformada por *unos*:

```
>>> np.ones((4,3))
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
```

O una conformada por ceros:

```
>>> np.zeros((5,5))
array([[ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.]])
```

En todos los casos anteriores habrá notado que el argumento de entrada es una tupla de dos elementos que indica el número de filas y columnas que tendrá la matriz. Adicionalmente podría especificar el tipo de dato mediante el *keyword argument* dtype.

Operaciones básicas con matrices

Suma y resta

Definimos dos matrices A y B de 3x3:

```
>>> A=np.matrix([[1,-1,2],[8,3,0],[-5,7,4]])
>>> B=np.matrix([[3,2,0],[11,-4,1],[6,1,2]])
>>> print A
[[ 1 -1  2]
 [ 8  3  0]
 [-5  7  4]]
```

```
>>> print B
[[ 3  2  0]
 [11 -4  1]
 [ 6  1  2]]
```

Por definición es necesario que las matrices a sumar o restar tengan las mismas dimensiones, puesto que son operaciones realizadas elemento a elemento. Para llevar a cabo las operaciones simplemente debe hacerse como lo haría con valores escalares (al igual que en MATLAB):

```
>>> A+B
matrix([[ 4,  1,  2],
        [19, -1,  1],
        [ 1,  8,  6]])
>>> A-B
matrix([[ -2,  -3,   2],
        [ -3,   7,  -1],
        [-11,   6,   2]])
>>> B-A
matrix([[ 2,  3, -2],
        [ 3, -7,  1],
        [11, -6, -2]])
```

7. Gráficas

Las gráficas en MATLAB son parte de su enorme popularidad, puesto que dispone de un amplio catálogo de posibilidades, y una flexibilidad importante.

La librería Matplotlib de Python no es menos interesante, puede que sea menos *madura*, pero mantiene una coherencia y simplicidad extraordinaria. Además que permite la manipulación de los entes gráficos como objetos de una determinada clase, lo cual le convierte en la mayoría de los casos en una poderosa herramienta.

Matplotlib dispone también de un módulo llamado `pylab`, el cual es prácticamente un mini-MATLAB, en cuestiones de gráficas, claro. Pero para nuestros propósitos omitiremos este módulo, y en su lugar utilizaremos el módulo `pyplot` que tiene una estructura más *pythonica*.

Gráficas en dos dimensiones

En MATLAB haríamos algo como lo siguiente:

```
x = linspace(0,10);
y = cos(x);
plot(x,y);
```

Con Matplotlib:

```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0,10)
y = np.cos(x)

plt.plot(x,y)
plt.show()
```

Ahora lo importante: con las primeras dos líneas importamos los módulos que utilizaremos, `matplotlib.pyplot` y `numpy` en este caso, además de usar un pseudónimo para cada uno. El resto de lineas es prácticamente similar, con la necesidad de anteponer el pseudónimo del módulo correspondiente para cada una de las funciones. La instrucción `plt.show()` sirve para mostrar la gráfica que hemos

creado, si no colocamos esto el programa no mostraría ningún resultado. Esa es una de las principales diferencias respecto a MATLAB.

Modificando etiquetas y título

Vamos a hacer unas pequeñas modificaciones para colocar un título y etiquetas a la gráfica anterior.

Para MATLAB:

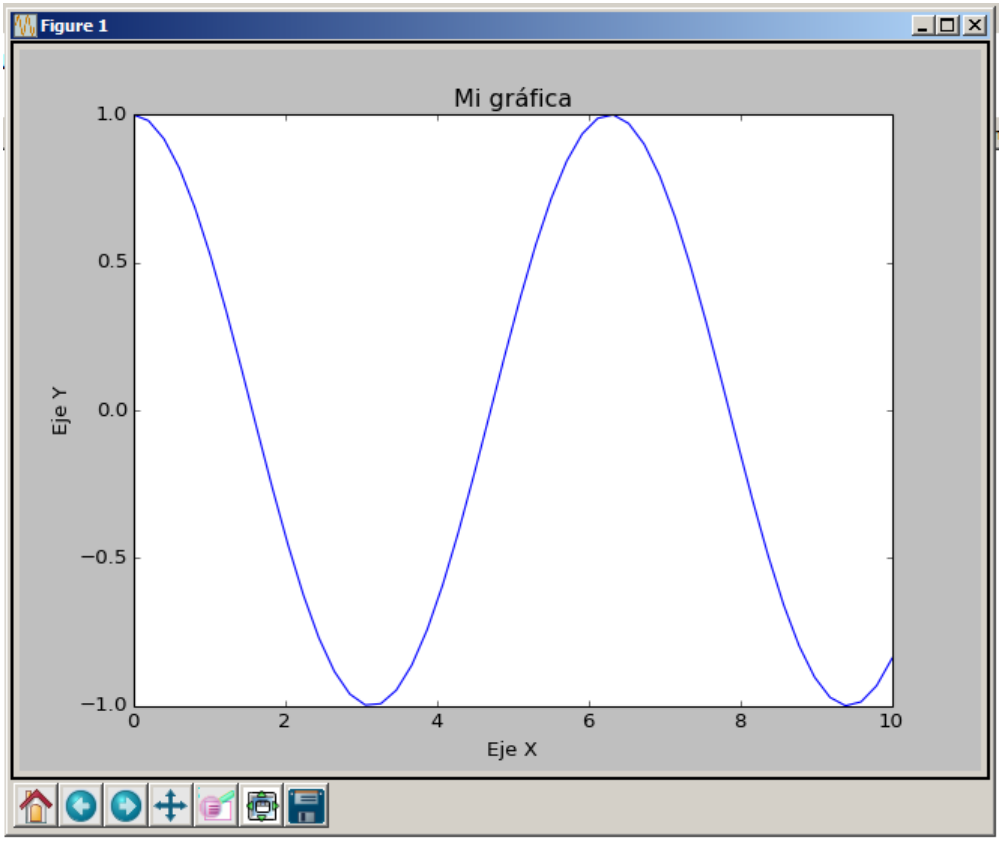
```
x = linspace(0,10);
y = cos(x);
plot(x,y);
xlabel('Eje X');
ylabel('Eje Y');
title('Mi gráfica');
```

En Python / Matplotlib:

```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0,10)
y = np.cos(x)

plt.plot(x,y)
plt.xlabel('Eje X')
plt.ylabel('Eje Y')
plt.title(u'Mi gráfica')
plt.show()
```



Modificando el color, ancho y estilo de linea

En MATLAB existe la posibilidad de pasar un tercer argumento a la función `plot` en forma de string, el cual indica un color y estilo de linea, por ejemplo:

```
>> plot(x,y,'r--');
```

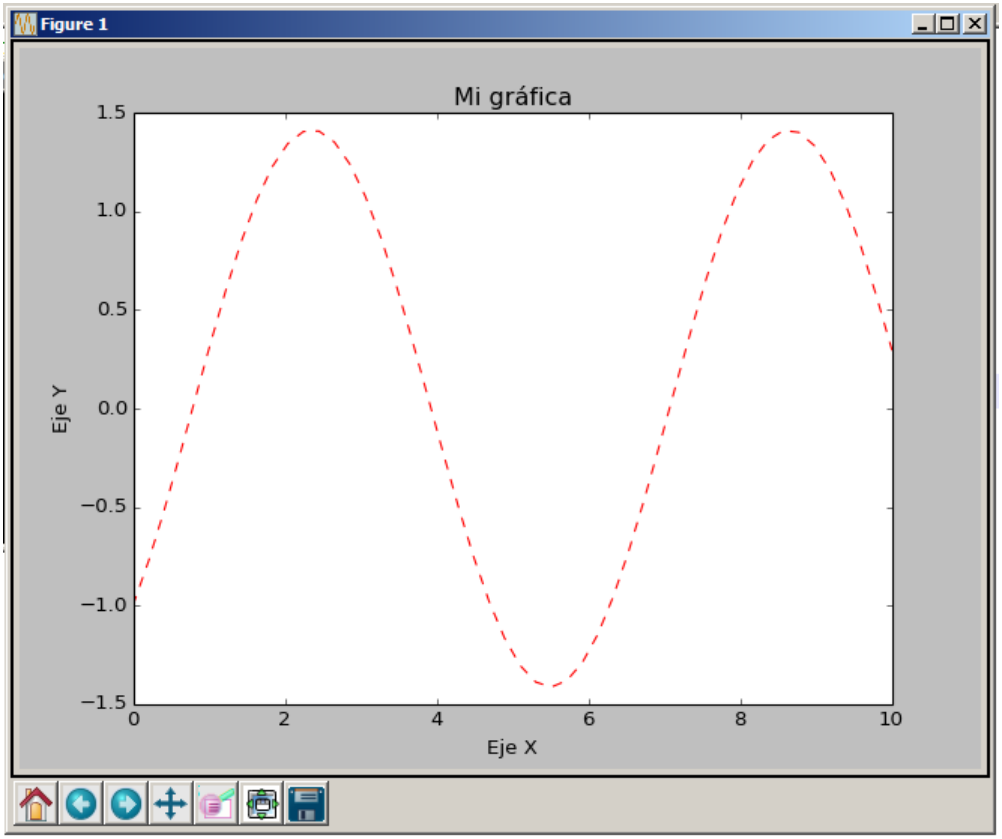
La linea anterior traza una gráfica de color rojo y "punteada" o discontinua.

Matplotlib funciona exactamente igual, puede especificar de esa forma un color y estilo de linea, por ejemplo:

```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0,10)
y = np.sin(x)-np.cos(x)
```

```
plt.plot(x,y,'r--')
plt.xlabel('Eje X')
plt.ylabel('Eje Y')
plt.title(u'Mi gráfica')
plt.show()
```



En MATLAB se utilizan los argumentos *pareados* para modificar las características de un objeto gráfico, por ejemplo para modificar el ancho de línea de una gráfica:

```
plot(x,y,'r','linewidth',3);
```

En Python existe el concepto de *keyword argument* (vea la sección correspondiente a funciones) que permite pasar argumentos como se muestra enseguida:

```
plt.plot(x,y,'r',linewidth=3)
```

Con lo anterior se estaría modificando el ancho de línea.

Para modificar el color y estilo de línea también es posible hacerlo mediante *keyword arguments*, vea el ejemplo mostrado a continuación:

```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0,10)
y = np.sin(x)-np.cos(x)

plt.plot(x,y,color=(0,1,0),linewidth=3,linestyle='--')
plt.xlabel('Eje X')
plt.ylabel('Eje Y')
plt.title(u'Mi gráfica')
plt.show()
```

Puede notar que el color puede especificarse mediante una tupla de tres elementos, cuyos valores corresponden al modelo de color RGB, en un intervalo de 0 a 1. Pero Matplotlib presenta además una flexibilidad extraordinaria, permitiendo especificar el color mediante valores hexadecimales (muy común en la programación web), es decir, para indicar que se requiere un color rojo puede hacerlo de las siguientes formas equivalentes entre sí:

```
plt.plot(x,y,color="r")
plt.plot(x,y,color=(1,0,0))
plt.plot(x,y,color="#ff0000")
```