Tema 1 - Material de apoyo

Curso de Física Computacional

M. en C. Gustavo Contreras Mayén

5 de septiembre de 2012

Contenido

Contenido

Contenido

Contenido

Contenido

¿Para qué Numpy?

El objeto principal de Numpy es el arreglo multidimensional homogéneo. Es una tabla de elementos (normalmente números) donde todos ellos son del mismo tipo, indizados por una tupla de enteros positivos. En Numpy las *dimensiones* son llamadas ejes. Al número de ejes se llama rango.

Por ejemplo, las coordenadas de un punto en el espacio 3D $[1, 2, 1]$ es un arreglo de rango 1, ya que tiene sólo un eje. Ese eje tiene una longitud de 3.

$$[[1., 0., 0.],$$
$$[0., 1., 2.]]$$

El arreglo es de rango 2 (por que es de dimensión 2), la primera dimensión (o eje) tiene una longitud de 2, la segunda dimensión tiene longitud 3.

La clase en Numpy para arreglos es `ndarray`, que también es conocido con el alias de `array`. Nótese que `numpy.array` no es el mismo elemento de la librería estándar de Python `array.array`, ya que ésta sólo maneja arreglos de 1D y ofrece menos funcionalidad.

Atributos de ndarray

Los atributos más importantes de un objeto ndarray son:

- **ndarray.dim**: es el número de ejes en el arreglo. En el Universo Python, el número de dimensiones se conoce como *rango*.
- **ndarray.shape**: son las dimensiones del arreglo. Es una tupla de enteros que indican el tamaño del arreglo en cada dimensión. Para un arreglo (matriz) con *n* renglones y *m* columnas, shape será (n,m). La longitud de la tupla shape, es por tanto el rango, o número de dimensiones, ndim.

Atributos de ndarray

- **ndarray.size**: es el total de número de elementos en el arreglo. Es igual al producto de los elementos de `shape`.
- **ndarray.dtype**: es un objeto que describe el tipo de elementos en el arreglo. Podemos crear un d-tipeado específico, usando los tipos estándar de Python; Numpy proporciona tipos propios: `numpy.int32`, `numpy.int16` y `numpy.float64` como ejemplos.

Ejemplos 1

```
>>> from numpy import *
```

Ejemplos 1

```
>>> from numpy import *
>>> a = arange(15).reshape(3,5)
```

Ejemplos 1

```
>>> from numpy import *
>>> a = arange(15).reshape(3,5)
>>> a
```

Ejemplos 1

```
>>> from numpy import *
>>> a = arange(15).reshape(3,5)
>>> a
array ([[0,1,2,3,4],
       [5, 6, 7, 8, 9],
       [10, 11, 12, 13, 14]])
```

Ejemplos 1

```
>>> from numpy import *
>>> a = arange(15).reshape(3,5)
>>> a
array ([[0,1,2,3,4],
       [5, 6, 7, 8, 9],
       [10, 11, 12, 13, 14]])
>>> a.shape
```

Ejemplos 1

```
>>> from numpy import *
>>> a = arange(15).reshape(3,5)
>>> a
array ([[0,1,2,3,4],
       [5, 6, 7, 8, 9],
       [10, 11, 12, 13, 14]])
>>> a.shape
(3,5)
```

Ejemplos 1

```
>>> from numpy import *
>>> a = arange(15).reshape(3,5)
>>> a
array ([[0,1,2,3,4],
       [5, 6, 7, 8, 9],
       [10, 11, 12, 13, 14]])
>>> a.shape
(3,5)
>>> a.ndim
```

Ejemplos 1

```
>>> from numpy import *
>>> a = arange(15).reshape(3,5)
>>> a
array ([[0,1,2,3,4],
       [5, 6, 7, 8, 9],
       [10, 11, 12, 13, 14]])
>>> a.shape
(3,5)
>>> a.ndim
2
```

Ejemplos 1

```
>>> from numpy import *
>>> a = arange(15).reshape(3,5)
>>> a
array ([[0,1,2,3,4],
       [5, 6, 7, 8, 9],
       [10, 11, 12, 13, 14]])
>>> a.shape
(3,5)
>>> a.ndim
2
>>> a.dtype.name
```

Ejemplos 1

```
>>> from numpy import *
>>> a = arange(15).reshape(3,5)
>>> a
array ([[0,1,2,3,4],
       [5, 6, 7, 8, 9],
       [10, 11, 12, 13, 14]])
>>> a.shape
(3,5)
>>> a.ndim
2
>>> a.dtype.name
'int32'
```

Ejemplos 1

```
>>> from numpy import *
>>> a = arange(15).reshape(3,5)
>>> a
array ([[0,1,2,3,4],
       [5, 6, 7, 8, 9],
       [10, 11, 12, 13, 14]])
>>> a.shape
(3,5)
>>> a.ndim
2
>>> a.dtype.name
'int32'
>>> a.itemsize
```

Ejemplos 1

```
>>> from numpy import *
>>> a = arange(15).reshape(3,5)
>>> a
array ([[0,1,2,3,4],
       [5, 6, 7, 8, 9],
       [10, 11, 12, 13, 14]])
>>> a.shape
(3,5)
>>> a.ndim
2
>>> a.dtype.name
'int32'
>>> a.itemsize
4
```

Ejemplos 2

```
>>> a.size
```

Ejemplos 2

```
>>> a.size
15
```

Ejemplos 2

```
>>> a.size
15
>>> type(a)
```

Ejemplos 2

```
>>> a.size
15
>>> type(a)
numpy.ndarray
```

Ejemplos 2

```
>>> a.size
15
>>> type(a)
numpy.ndarray
>>> b = array([6,7,8])
```

Ejemplos 2

```
>>> a.size
15
>>> type(a)
numpy.ndarray
>>> b = array([6,7,8])
>>> b
```

Ejemplos 2

```
>>> a.size
15
>>> type(a)
numpy.ndarray
>>> b = array([6,7,8])
>>> b
array ([6,7,8])
```

Ejemplos 2

```
>>> a.size
15
>>> type(a)
numpy.ndarray
>>> b = array([6,7,8])
>>> b
array ([6,7,8])
>>> type(b)
```

Ejemplos 2

```
>>> a.size
15
>>> type(a)
numpy.ndarray
>>> b = array([6,7,8])
>>> b
array ([6,7,8])
>>> type(b)
numpy.ndarray
```

Creando un arreglo

Existen diversas maneras para crear los arreglos.

Por ejemplo, podemos crear un arreglo de una lista o tupla regular de Python, usando la función `array`. El tipo de de arreglo resultante se deduce del tipo de elementos en la secuencia.

```
>>> from numpy import *
```

Creando un arreglo

Existen diversas maneras para crear los arreglos.

Por ejemplo, podemos crear un arreglo de una lista o tupla regular de Python, usando la función `array`. El tipo de de arreglo resultante se deduce del tipo de elementos en la secuencia.

```
>>> from numpy import *
>>> a = array([2,3,4])
```

Creando un arreglo

Existen diversas maneras para crear los arreglos.

Por ejemplo, podemos crear un arreglo de una lista o tupla regular de Python, usando la función `array`. El tipo de de arreglo resultante se deduce del tipo de elementos en la secuencia.

```
>>> from numpy import *
>>> a = array([2,3,4])
>>> a
```

Creando un arreglo

Existen diversas maneras para crear los arreglos.

Por ejemplo, podemos crear un arreglo de una lista o tupla regular de Python, usando la función `array`. El tipo de de arreglo resultante se deduce del tipo de elementos en la secuencia.

```
>>> from numpy import *
>>> a = array([2,3,4])
>>> a
array([2,3,4])
```

Creando un arreglo

Existen diversas maneras para crear los arreglos.

Por ejemplo, podemos crear un arreglo de una lista o tupla regular de Python, usando la función `array`. El tipo de de arreglo resultante se deduce del tipo de elementos en la secuencia.

```
>>> from numpy import *
>>> a = array([2,3,4])
>>> a
array([2,3,4])
>>> a.dtype
```

Creando un arreglo

Existen diversas maneras para crear los arreglos.

Por ejemplo, podemos crear un arreglo de una lista o tupla regular de Python, usando la función `array`. El tipo de de arreglo resultante se deduce del tipo de elementos en la secuencia.

```
>>> from numpy import *
>>> a = array([2,3,4])
>>> a
array([2,3,4])
>>> a.dtype
dtype('int32')
```

Creando un arreglo

Existen diversas maneras para crear los arreglos.

Por ejemplo, podemos crear un arreglo de una lista o tupla regular de Python, usando la función `array`. El tipo de de arreglo resultante se deduce del tipo de elementos en la secuencia.

```
>>> from numpy import *
>>> a = array([2,3,4])
>>> a
array([2,3,4])
>>> a.dtype
dtype('int32')
>>> b = array ([1.2, 3.5, 5.1])
```

Creando un arreglo

Existen diversas maneras para crear los arreglos.

Por ejemplo, podemos crear un arreglo de una lista o tupla regular de Python, usando la función `array`. El tipo de de arreglo resultante se deduce del tipo de elementos en la secuencia.

```
>>> from numpy import *
>>> a = array([2,3,4])
>>> a
array([2,3,4])
>>> a.dtype
dtype('int32')
>>> b = array ([1.2, 3.5, 5.1])
>>> b.dtype
```

Creando un arreglo

Existen diversas maneras para crear los arreglos.

Por ejemplo, podemos crear un arreglo de una lista o tupla regular de Python, usando la función `array`. El tipo de de arreglo resultante se deduce del tipo de elementos en la secuencia.

```
>>> from numpy import *
>>> a = array([2,3,4])
>>> a
array([2,3,4])
>>> a.dtype
dtype('int32')
>>> b = array ([1.2, 3.5, 5.1])
>>> b.dtype
```

Precaución

Un error que se presenta frecuentemente es llamar a la función `array` con múltiples argumentos, en lugar de proporcionar una única lista de números.

Precaución

Un error que se presenta frecuentemente es llamar a la función `array` con múltiples argumentos, en lugar de proporcionar una única lista de números.

```
>>> a = array (1,2,3,4)
```

Mal hecho

Precaución

Un error que se presenta frecuentemente es llamar a la función `array` con múltiples argumentos, en lugar de proporcionar una única lista de números.

```
>>> a = array (1,2,3,4)
>>> a = array ([1,2,3,4])
```

Mal hecho

Bien hecho

A menudo, los elementos de un arreglo no se conocen desde el inicio del problema, pero su tamaño si; Numpy proporciona algunas funciones que generan arreglos con un contenido inicial, con la finalidad de minimizar el crecimiento de arreglos, que es a su vez, una tarea que gasta recursos.

1. `zeros`: genera un arreglo donde todos los elementos son ceros.
2. `ones`: genera un arreglo donde todos los elementos del mismo son unos.
3. `empty`: genera un arreglo con valores iniciales aleatorios, por defecto, el tipo de dato es `float64`.

A menudo, los elementos de un arreglo no se conocen desde el inicio del problema, pero su tamaño si; Numpy proporciona algunas funciones que generan arreglos con un contenido inicial, con la finalidad de minimizar el crecimiento de arreglos, que es a su vez, una tarea que gasta recursos.

1. `zeros`: genera un arreglo donde todos los elementos son ceros.
2. `ones`: genera un arreglo donde todos los elementos del mismo son unos.
3. `empty`: genera un arreglo con valores iniciales aleatorios, por defecto, el tipo de dato es `float64`.

A menudo, los elementos de un arreglo no se conocen desde el inicio del problema, pero su tamaño si; Numpy proporciona algunas funciones que generan arreglos con un contenido inicial, con la finalidad de minimizar el crecimiento de arreglos, que es a su vez, una tarea que gasta recursos.

1. `zeros`: genera un arreglo donde todos los elementos son ceros.
2. `ones`: genera un arreglo donde todos los elementos del mismo son unos.
3. `empty`: genera un arreglo con valores iniciales aleatorios, por defecto, el tipo de dato es `float64`.

```
>>> zeros((3,4))
```

```
>>> zeros((3,4))
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
```

```
>>> zeros((3,4))
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
>>> ones((2,3,4), dtype=int16)
```

```
>>> zeros((3,4))
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
>>> ones((2,3,4), dtype=int16)
array([[[1, 1, 1, 1],
        [1, 1, 1, 1],
        [1, 1, 1, 1]],
       [[1, 1, 1, 1],
        [1, 1, 1, 1],
        [1, 1, 1, 1]]], dtype=int16)
```

```
>>> zeros((3,4))
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
>>> ones((2,3,4), dtype=int16)
array([[[1, 1, 1, 1],
        [1, 1, 1, 1]],
       [[1, 1, 1, 1],
        [1, 1, 1, 1]]], dtype=int16)
>>> empty((2,3))
```

La función arange

Para crear una secuencia de números, Numpy cuenta con una función análoga a `range`, que devuelve arreglos en lugar de listas.

```
>>> arange(10, 30, 5)
```

La función arange

Para crear una secuencia de números, Numpy cuenta con una función análoga a `range`, que devuelve arreglos en lugar de listas.

```
>>> arange(10, 30, 5)
([10, 15, 20, 25])
```

La función arange

Para crear una secuencia de números, Numpy cuenta con una función análoga a `range`, que devuelve arreglos en lugar de listas.

```
>>> arange(10, 30, 5)
([10, 15, 20, 25])
>>> arange(0, 2, 0.3)
```

La función arange

Para crear una secuencia de números, Numpy cuenta con una función análoga a `range`, que devuelve arreglos en lugar de listas.

```
>>> arange(10, 30, 5)
([10, 15, 20, 25])
>>> arange(0, 2, 0.3)
([0., 0.3, 0.6, 0.9, 1.2, 1.5, 1.8])
```

Función linspace

Cuando arange se usa con argumentos de punto flotante, en general no es posible predecir el número de elementos obtenidos, debido a la precisión de punto flotante. Por tanto, es mejor usar la función linspace, en el argumento de la función, se indica el número de elementos que queremos, en vez de un paso.

```
>>> linspace(0,2,9)
```

Función linspace

Cuando arange se usa con argumentos de punto flotante, en general no es posible predecir el número de elementos obtenidos, debido a la precisión de punto flotante. Por tanto, es mejor usar la función linspace, en el argumento de la función, se indica el número de elementos que queremos, en vez de un paso.

```
>>> linspace(0,2,9)
array([0., 0.25, 0.5, 0.75, 1., 1.25,
1.5, 1.75, 2.])
```

Función linspace

Cuando arange se usa con argumentos de punto flotante, en general no es posible predecir el número de elementos obtenidos, debido a la precisión de punto flotante. Por tanto, es mejor usar la función linspace, en el argumento de la función, se indica el número de elementos que queremos, en vez de un paso.

```
>>> linspace(0,2,9)
array([0., 0.25, 0.5, 0.75, 1., 1.25,
1.5, 1.75, 2.])
>>> x = linspace(0,2*pi, 100)
```

Función linspace

Cuando arange se usa con argumentos de punto flotante, en general no es posible predecir el número de elementos obtenidos, debido a la precisión de punto flotante. Por tanto, es mejor usar la función linspace, en el argumento de la función, se indica el número de elementos que queremos, en vez de un paso.

```
>>> linspace(0,2,9)
array([0., 0.25, 0.5, 0.75, 1., 1.25,
1.5, 1.75, 2.])
>>> x = linspace(0,2*pi, 100)
>>> f = sin(x)
```

Graficando datos

En el curso tendremos la necesidad de graficar datos obtenidos a partir de una aproximación numérica, para ello, usaremos `matplotlib` que es un módulo de Python con el que tendremos la oportunidad de elaborar gráficas desde básicas hasta unas más elaboradas.

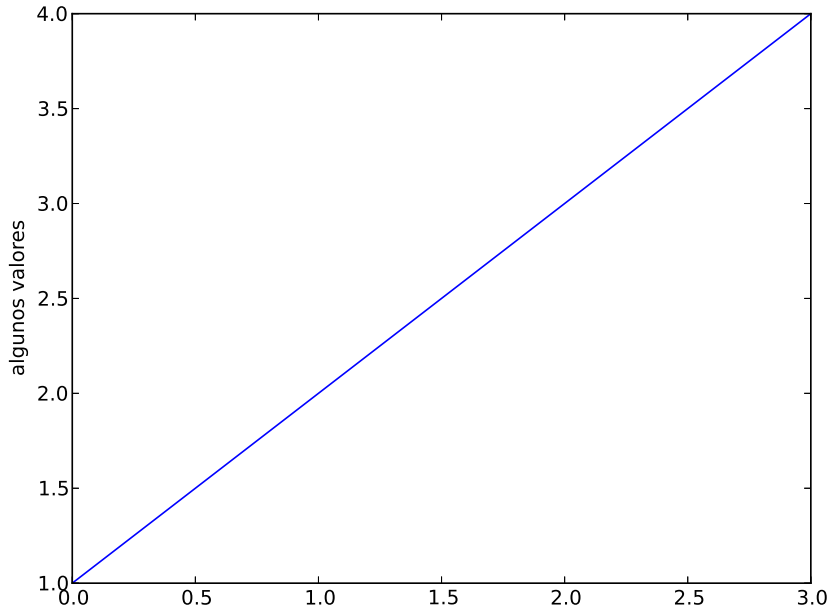
Como hemos mencionado, ya un abordaje más profundo en este módulo requiere de tiempo que tendrán que dedicarle por su cuenta.

Ejemplo 1

```
1 import matplotlib.pyplot as plt
2 plt.plot([1,2,3,4])
3 plt.ylabel('algunos valores')
4 plt.show()
```

Ejemplo 1

```
1 import matplotlib.pyplot as plt
2 plt.plot([1,2,3,4])
3 plt.ylabel('algunos valores')
4 plt.show()
```

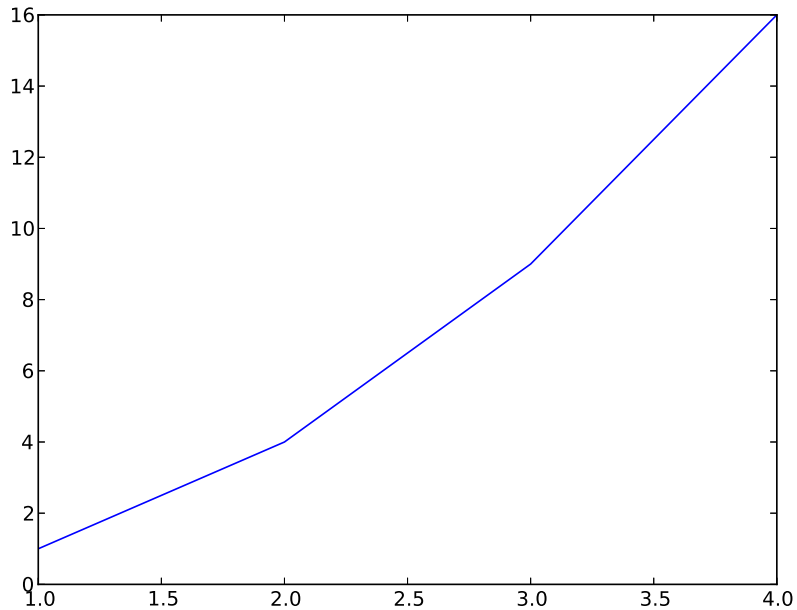


Ejemplo 2

```
1 import matplotlib.pyplot as plt
2 plt.plot([1,2,3,4],[1,4,9,16])
3 plt.show()
```

Ejemplo 2

```
1 import matplotlib.pyplot as plt
2 plt.plot([1,2,3,4],[1,4,9,16])
3 plt.show()
```

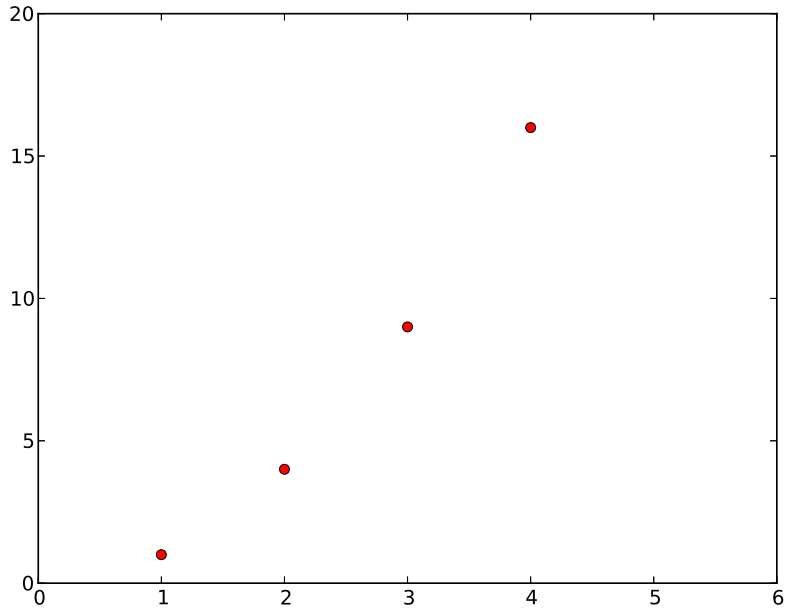


Ejemplo 3

```
1 import matplotlib.pyplot as plt
2 plt.plot([1,2,3,4], [1,4,9,16], 'ro')
3 plt.axis([0, 6, 0, 20])
4 plt.show()
```

Ejemplo 3

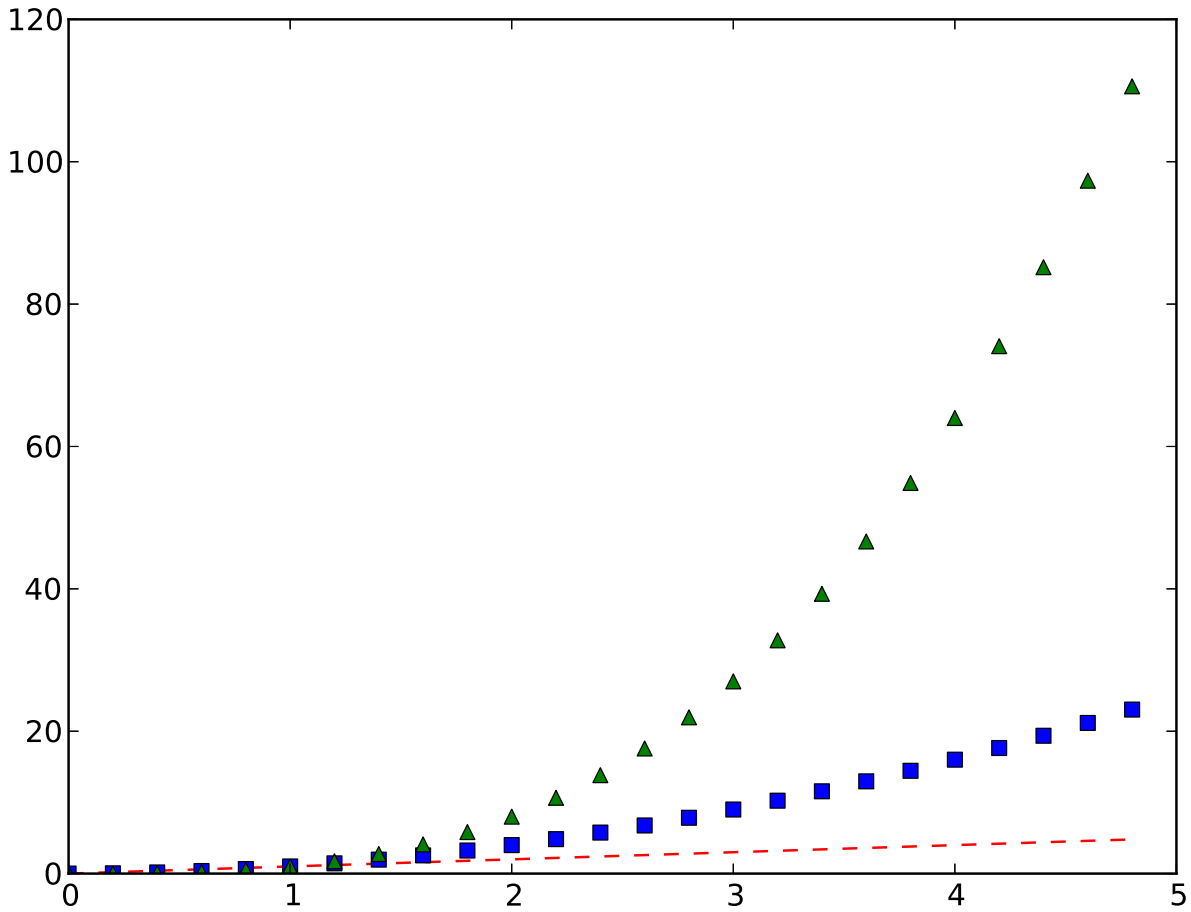
```
1 import matplotlib.pyplot as plt
2 plt.plot([1,2,3,4], [1,4,9,16], 'ro')
3 plt.axis([0, 6, 0, 20])
4 plt.show()
```



Ejemplo 4

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 t = np.arange(0., 5., 0.2)
5
6 # lineas rojas, cuadros azules y triangulos verdes
7 plt.plot(t, t, 'r--', t, t**2, 'bs', t, t**3, 'g^')
8 plt.show()
```

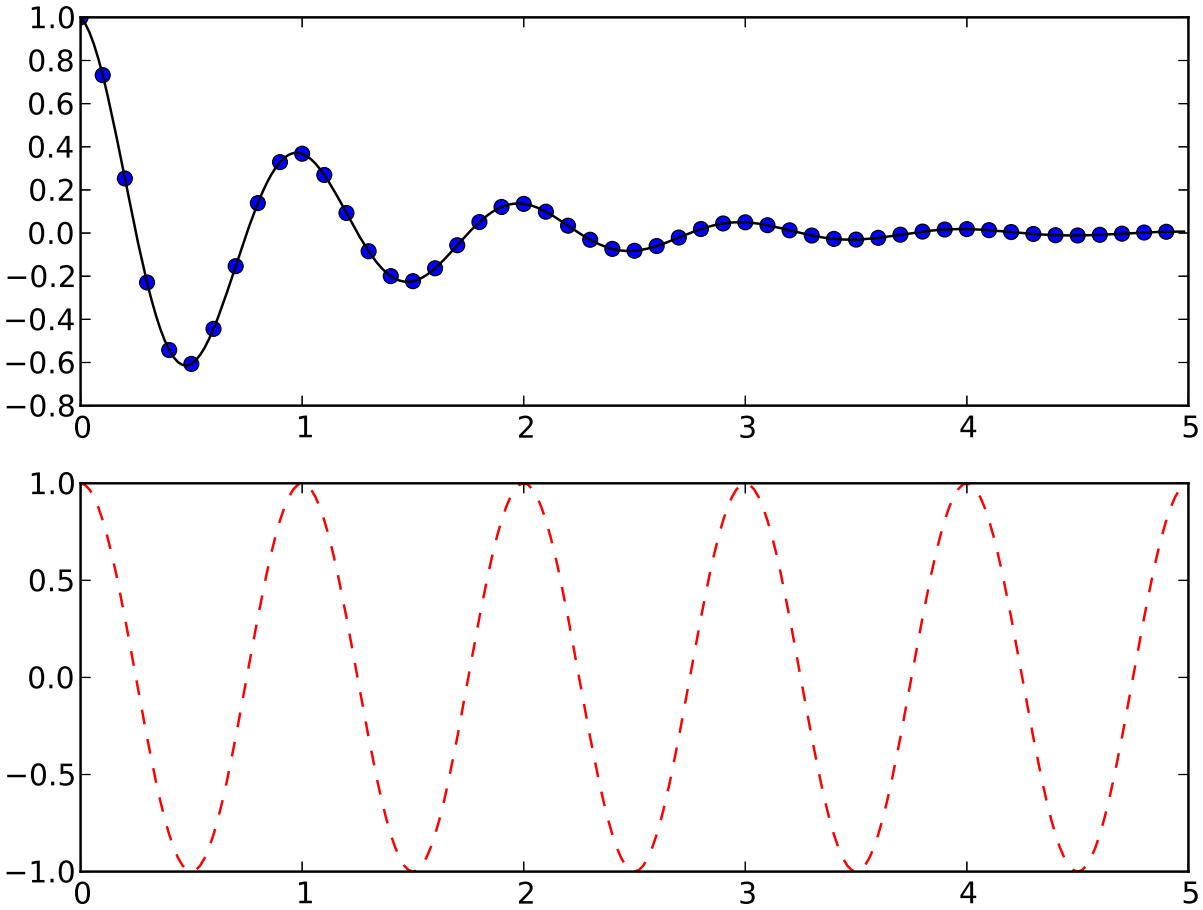
Gráfica del Ejemplo 4



Ejemplo 5 - Múltiples gráficas

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  def f(t):
5      return np.exp(-t) * np.cos(2*np.pi*t)
6
7  t1 = np.arange(0.0, 5.0, 0.1)
8  t2 = np.arange(0.0, 5.0, 0.02)
9
10 plt.figure(1)
11 plt.subplot(211)
12 plt.plot(t1, f(t1), 'bo', t2, f(t2), 'k')
13
14 plt.subplot(212)
15 plt.plot(t2, np.cos(2*np.pi*t2), 'r--')
16 plt.show()
```

Gráfica del Ejemplo 5

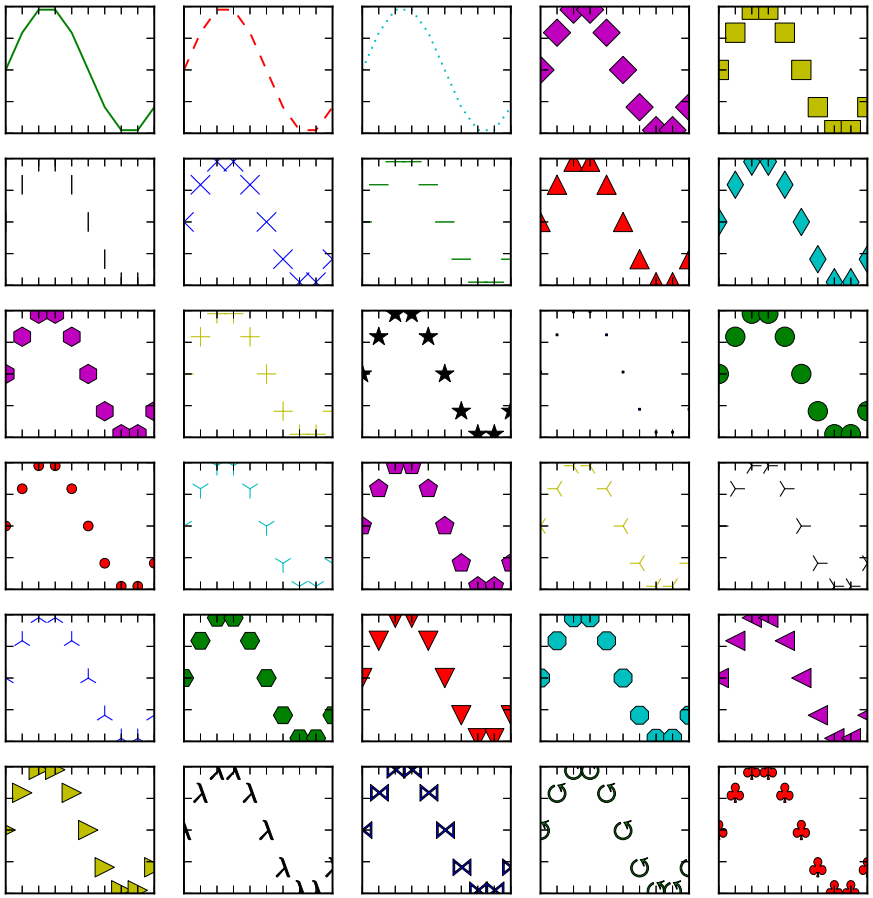


Aclaración sobre múltiples gráficas.

El comando `figure` es opcional, ya que `figure(1)` se genera por defecto, su equivalente es `subfigure(111)` que se crea también por defecto.

El comando `subplot()` define `numrows, numcols, fignum`, donde el rango de `fignum` varía entre 1 y `numrows*numcols`. Las comas en el comando `subplot` son opcionales si `numrows*numcols`<10, por lo que es lo mismo `subplot(211)` y `subplot(2,1,1)`.

Gráficas Múltiples



Ejemplo 6

```
1  #importar Numpy y matplotlib
2  mu, sigma = 100, 15
3  x = mu + sigma * np.random.randn(10000)
4
5  #el histograma de los datos
6  n, bins, patches = plt.hist(x, 50, normed=1,
       facecolor='g', alpha=0.75)
7
8  plt.xlabel('Genios')
9  plt.ylabel('Probabilidad')
10 plt.title('Histograma de IQ')
11 plt.text(60, .025, r'$\mu=100,\ \sigma=15$')
12 plt.axis([40, 160, 0, 0.03])
13 plt.grid(True)
14 plt.show()
```

Gráfica del Ejemplo 6