😎 Matplotlib

Es una biblioteca para la generación de gráficos.

```
pip install matplotlib
```

Interfaces (Pyplot vs Interfaz Orientada a Objetos)

Una característica potencialmente confusa de Matplotlib son sus interfaces duales: una conveniente interfaz basada en estado al estilo de MATLAB y una interfaz orientada a objetos más poderosa.

Interfaz pyplot

MATLAB Matplotlib se escribió originalmente como una alternativa de Python para los usuarios de MATLAB, y gran parte de su sintaxis refleja ese hecho. Las herramientas de estilo MATLAB están contenidas en la interfaz pyplot (plt).

Es importante tener en cuenta que esta interfaz tiene estado: realiza un seguimiento de la figura y los ejes "actuales", que es donde se aplican todos los comandos plt. Se puede obtener una referencia a estos usando las rutinas plt.gcf() -get current figure- y plt.gca() -get current axes-

Interfaz orientada a objetos

La interfaz orientada a objetos está disponible para situaciones más complicadas y para cuando se quiera tener más control sobre la figura. En lugar de depender de alguna noción de una figura o ejes "activos", en la interfaz orientada a objetos **las funciones de trazado son métodos de objetos explícitos de Figura y Ejes**.

Terminología

Primero es importante entender los componentes de una figura en Matplotlib. Los componentes son jerárquicos:

- **Objeto Figure: toda la figura**
- **Objeto Axes: pertenece a un objeto Figure y es el espacio donde agregamos los datos a visualizar**
- **Objeto Axis: pertenece a un objeto Axes. El objeto Axis se puede categorizar como XAxiso YAxis**

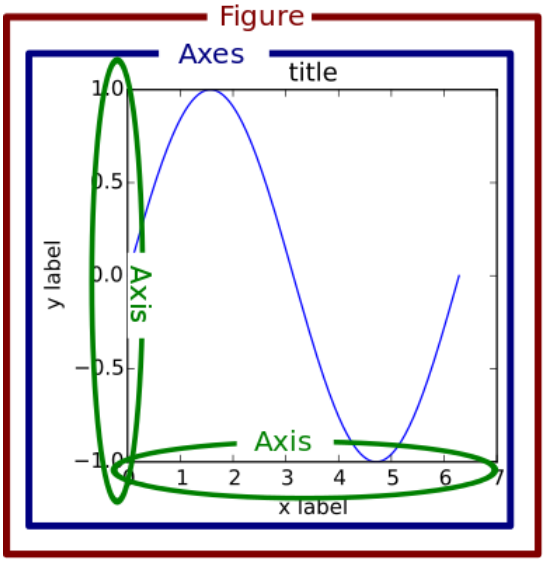
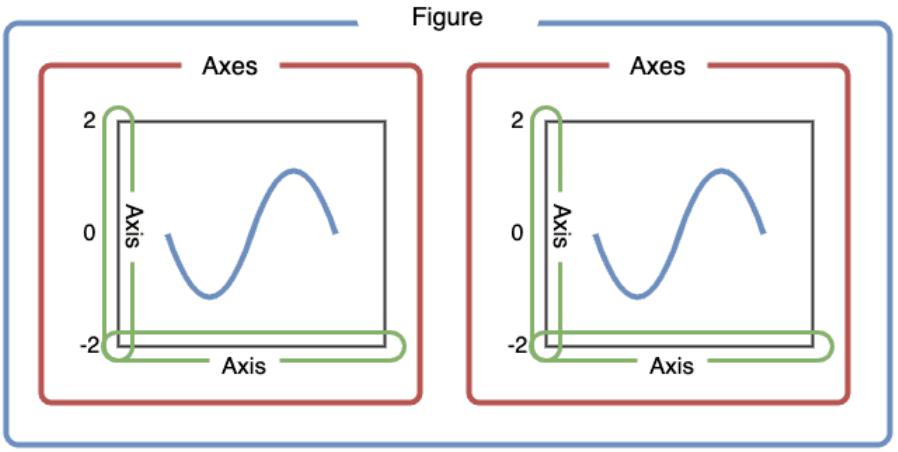
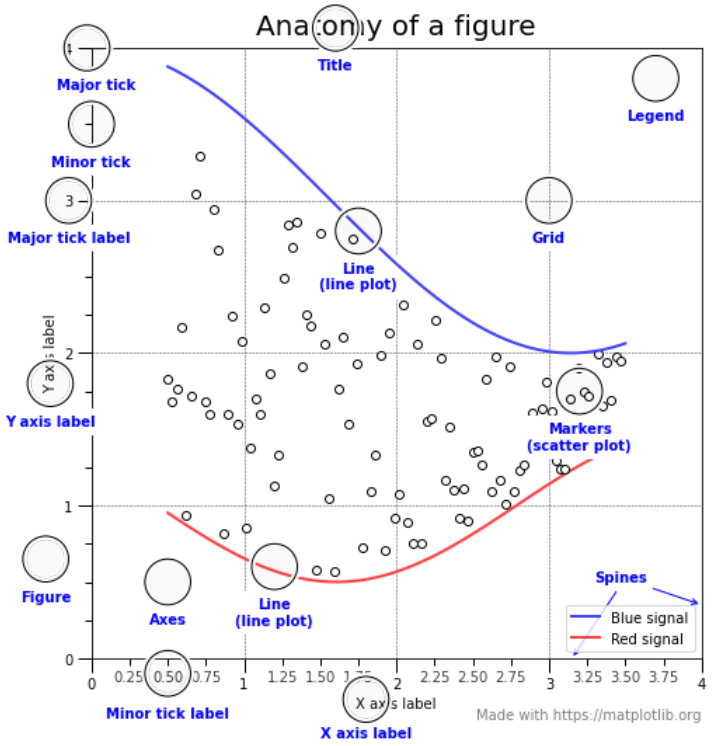


Figure se puede considerar como un contenedor único que contiene todos los objetos que representan ejes, gráficos, texto y etiquetas. Los Axes es el cuadro delimitador con marcas y etiquetas, que eventualmente contendrá los elementos de la trama que componen nuestra visualización.



La siguiente figura también muestra algunos de los elementos más detallados que componen una figura en Matplotlib:



Partes de una figura (https://matplotlib.org/stable/tutorials/introductory/usage.html#parts-of-a-figure)

- **Figure**: La figura es el contenedor principal. En la figura se definen todos los parámetros de como es que la o las graficas se van a acomodar o agrupar, o sea que es la ventana o página general en la que se dibuja todo, es el componente de nivel superior de todos lo que se considerará en los siguientes puntos.
- **Axes**: Es el contenedor de la grafica. Una figura puede contener varios axes, y un axes puede contener varias graficas. El parámetro de axes tiene como miembros los títulos, los limites de los ejes, los incrementos de los ejes, entre otros valores. Aquí se grafican los datos con funciones tales como plot() y scatter() y que pueden tener etiquetas asociadas.
- **Axis**: Son los números que definen los ticks que lleva la grafica. Corresponde uno para cada uno de los ejes de la grafica. Cada eje tiene un eje x y otro eje y, y cada uno de ellos contiene una numeración. También existen las etiquetas de los ejes, el título y la leyenda que se deben tener en cuenta cuando se quieren personalizar los ejes.
- **Artist**: Son los parámetros de estilo de la grafica, como la línea, texto, espesor, color, entre otros.

Enfoques para trazar

Como se mencionó, hay dos enfoques principales para trazar en Matplotlib:

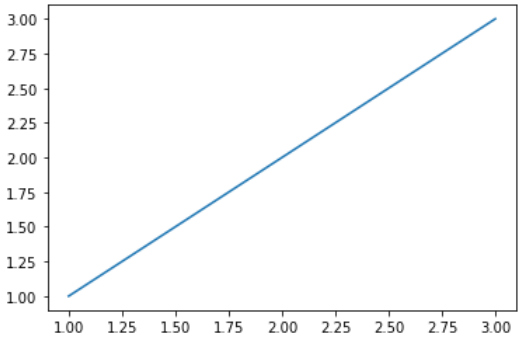
- **Interfaz orientada a objetos**
- **Interfaz Pyplot**

Interfaz orientada a objetos

En este enfoque, creamos objetos Figure que contendrán objetos Axes (es decir, plots). Hacemos esto explícitamente, y esto a su vez permite un mayor nivel de control y personalización al construir nuestras figuras. Usaremos el nombre de variable **fig** para referirnos a una instancia de figura y **ax** para referirnos a una instancia de ejes o grupo de instancias de ejes.

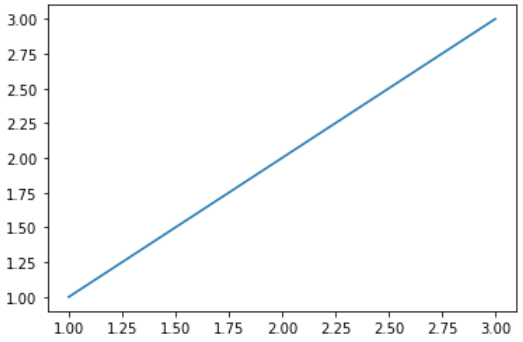
El siguiente fragmento de código demuestra cómo trazar un gráfico de líneas utilizando la interfaz orientada a objetos:

In [1]:
```
import matplotlib.pyplot as plt

# Inicializar un objeto figura
fig = plt.figure()

# Agregue un objeto Axes a la Figura usando add_subplot(1,1,1)
# (1,1,1) aquí le dice que agregue una cuadrícula de 1x1, 1ra subtrama
ax = fig.add_subplot(1,1,1)

x = [1,2,3]
y = [1,2,3]

# Trazar los datos y mostrarlos
ax.plot(x, y)
plt.show()
```



En lugar de generar el objeto Figure y Axes por separado, podemos inicializarlos juntos

In [2]:
```
fig, ax = plt.subplots()
ax.plot(x, y)
plt.show()
```

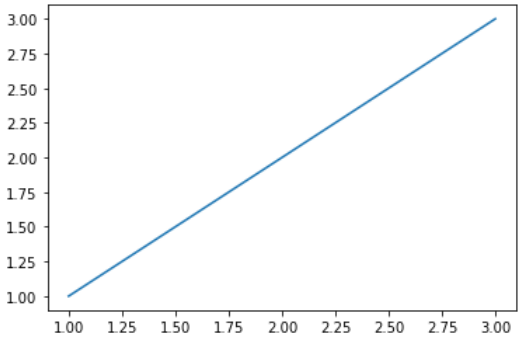


Interfaz Pyplot

La interfaz de Pyplot generará y configurará automáticamente los objetos Figure/ Axes con los que se trabaja. Esto significa que continuará superponiendo los cambios al mismo Figure usando comandos pyplot. Esto es diferente a la interfaz orientada a objetos donde especificamos explícitamente a qué objetos Figure o Axes aplicar nuestros cambios.

Aunque la interfaz de Pyplot es una forma cómoda y rápida de trazar gráficos simples, no es adecuada para hacer gráficos más complicados, como subgráficos.

In [3]:
```
x = [1,2,3]
y = [1,2,3]
plt.plot(x,y)
plt.show()
```



En este caso, la llamada plt.plot(~) crea implícitamente una instancia Figure y una instancia Axes en segundo plano.

¿Cuál es la diferencia?

El código de la interfaz pyplot, convirtió en una FigureCanvas y luego en este se genera implícitamente un área de dibujo en el lienzo para dibujar.

El código de la interfaz orientada a objetos genera dos objetos Figure y axes, luego usa el objeto ax dentro de su área de dibujo.

Más explicaciones: https://stackoverflow.com/questions/37970424/what-is-the-difference-between-drawing-plots-using-plot-axes-or-figure-in-matpl (https://stackoverflow.com/questions/37970424/what-is-the-difference-between-drawing-plots-using-plot-axes-or-figure-in-matpl)

Backends

Matplotlib tiene una serie de backends diferentes disponibles y cuando se utiliza matplotlib para dibujar o trazar datos, **según el entorno de desarrollo o la interfaz de práctica elegida**, el backend puede ser diferente. Para presentar eficazmente la función de dibujo, se debe tener el **back-end** de soporte correspondiente para comunicarse. Por ejemplo, si se utiliza el shell de python para interactuar con matplotlib, escribiendo la siguiente información, se obtiene el back-end:

In [4]:
```
%matplotlib notebook
import matplotlib.pyplot as plt
plt.get_backend()
```

Out[4]:
```
'nbAgg'
```

Si se está interactuando usando en línea en Jupyter Notebook, escribiendo la información se obtiene el back-end:

In [5]:
```
%matplotlib inline
import matplotlib.pyplot as plt
plt.get_backend()
```

Out[5]:
```
'module://matplotlib_inline.backend_inline'
```

In [6]:
```
matplotlib -l
```

```
Available matplotlib backends: ['tk', 'gtk', 'gtk3', 'gtk4', 'wx', 'qt4', 'qt5', 'qt6', 'qt', 'osx', 'nbagg', 'no
tebook', 'agg', 'svg', 'pdf', 'ps', 'inline', 'ipympl', 'widget']
```

Jupyter Notebook, tiene soporte para matplotlib que se habilita mediante el uso de IPython (https://ipython.org/)

Los procesos de IPython son funciones auxiliares que configuran el entorno para que la representación basada en la web se pueda habilitar. Cuando se ejecuta matplotlib con el parámetro en línea: **%matplotlib**, es para que renderice en el navegador.

> **Un backend es una capa de abstracción que sabe cómo interactuar con el entorno operativo, sea un sistema operativo, o un entorno como el navegador, y el back-end sabe cómo renderizar comandos de matplotlib**

Arquitectura de Matplotlib

Se compone de tres capas principales:

Capa back-end (Backend layer):

> **Controla todos los trabajos mediante la comunicación a los kits de herramientas de dibujo, es decir que se ocupa de la representación de gráficos en pantalla o archivos. Es la capa más compleja y consta de 3 clases de interfaz:**

- **FigureCanvas: define el área para dibujar una figura, encapsula el concepto de una superficie sobre la que dibujar (por ejemplo, "el papel").**
- **Renderer: una instancia del representador que sabe cómo dibujar una figura en FigureCanvas, es decir que hace el dibujo (por ejemplo, "el pincel").**
- **Event: maneja entradas de usuario, como el teclado, el mouse o las pantallas táctiles.**

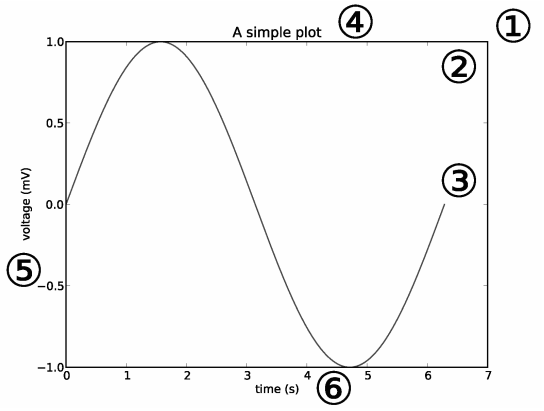
Hay una serie de backends interactivos diferentes y también existen los llamados backends de copia impresa, que admiten renderizado en formatos gráficos, como gráficos vectoriales escalables, SVG o PNG. Por lo tanto, no todos los backends admiten todas las características, especialmente las características interactivas.

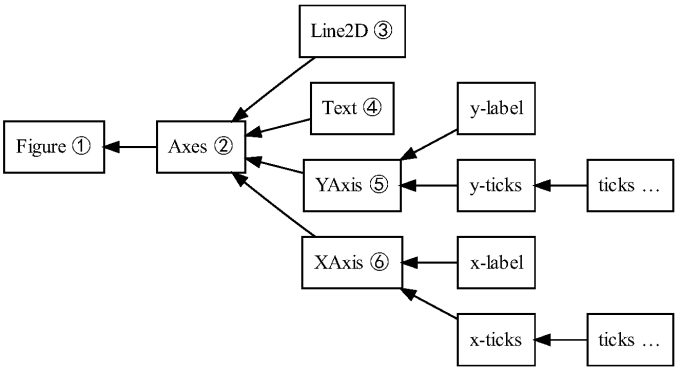
Capa de artista (Artist layer):

Es el objeto que sabe tomar el Renderer (el pincel) de la capa Back-end para dibujar el gráfico en FigureCanvas, es decir dibujar en el lienzo. Todo lo que se ve en Figure es una instancia de Artist, el título, las líneas, las tick, las imágenes, etc., y corresponden a instancias de Artist individuales.

- **Posee contenedores que deben controlar dónde se colocan los objetos, como Figure, Subplots y Axes. Artist Layer sabe exactamente cómo se compone el gráfico de subplots y la posición relativa del objeto en un sistema de ejes determinado (Axes). La base de los objetos visuales es un conjunto de contenedores que incluye un objeto "figure" con uno o más subplots, cada uno con una serie de uno o más ejes. Matplotlib depende en gran medida del objeto de ejes, pero también tiene un objeto eje. Y un eje está hecho de dos objetos eje. Uno para la x, dimensión horizontal y uno para y, o dimensión vertical. Estos últimos, son lo más común con los que se interactúa, cambiando el rango de un eje determinado o trazando formas en él.**
- **Posee la base principal para trabajar con objetos gráficos estándar (o primitivas) que queremos dibujar como: Line2D, Rectangle, Text, AxesImage, etc., y colecciones como PathCollection. Las colecciones saben cómo se componen las figuras de subfiguras y dónde se encuentran los objetos en un sistema de coordenadas de ejes determinado.**

Tipos de artistas están disponibles (https://matplotlib.org/stable/api/artist_api.html)



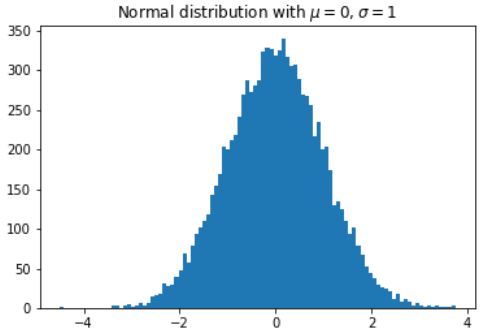


Fuente (https://www.aosabook.org/en/matplotlib.html)

A continuación se muestra una secuencia de comandos de Python simple que ilustra la arquitectura anterior. Define el backend, se conecta Figure a él, usa la biblioteca de matrices numpy para crear 10,000 números aleatorios normalmente distribuidos y traza un histograma.

In [7]:

```
# Importe el FigureCanvas desde el backend de su elección y adjunte el Artista de la figura a él.
from matplotlib.backends.backend_agg import FigureCanvasAgg as FigureCanvas
from matplotlib.figure import Figure
fig = Figure()
canvas = FigureCanvas(fig)

# Importar la biblioteca numpy para generar los números aleatorios.
import numpy as np
x = np.random.randn(10000)

# Ahora usa un método de figura para crear un artista de Axes; el artista Axes se agrega automáticamente al
# contenedor de figuras fig.axes. Aquí "111" es de la convención de MATLAB: cree una cuadrícula con 1 fila
# y 1 columna, y use la primera celda en esa cuadrícula para la ubicación de los nuevos ejes.
ax = fig.add_subplot(111)

# Llame al método Axes hist para generar el histograma; hist crea una secuencia de artistas Rectangle
# para cada  barra de histograma y los agrega al contenedor Axes. Aquí "100" significa crear  100 contenedores (bi
ax.hist(x, 100)

# Decora la figura con un título y guárdala.
ax.set_title('Normal distribution with $\mu=0, \sigma=1$')
fig.savefig('img/matplotlib_histogram.png')
```

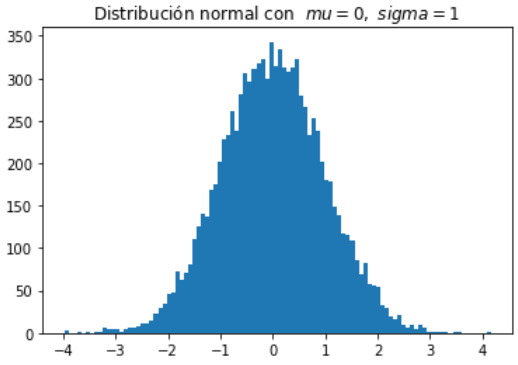


Capa de scripting (Scripting layer):

> **Simplifica el acceso a las capas de artista y backend> La interfaz de scripting es más ligera entre las tres capas, está diseñada para hacer que Matplotlib funcione como el script MATLAB. Permite una fácil generación de variedad de gráficos y ayuda a simplificar y acelerar nuestra interacción con el entorno mediante la interfaz matplotlib.pyplot. Esta capa automatiza el proceso de definición de la instancia de FigureCanvas y Artist, lo que facilita su uso para un análisis exploratorio rápido.**

El mismo código anterior usando **pyplot**:

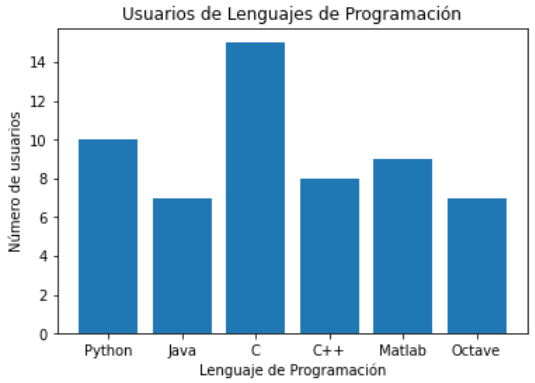
In [8]:
```
import matplotlib.pyplot as plt
import numpy as np
x = np.random.randn (10000)
plt.hist (x, 100)
plt.title (r'Distribución normal con $ \ mu = 0, \ sigma = 1 $ ')
plt.savefig ('img/matplotlib_histogram2.png')
plt.show ()
```



Ejemplo que muestra la funcionalidad de la capa Artist de Matplotlib

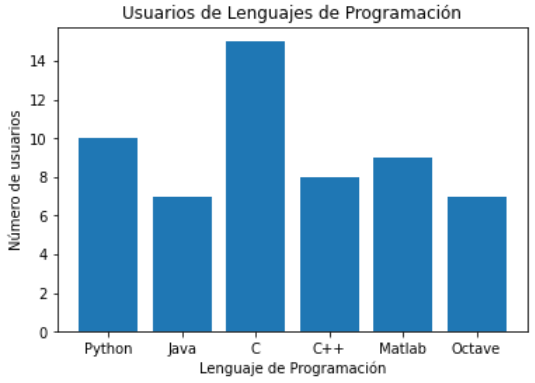
In [9]:
```
import matplotlib.pyplot as plt

# Creamos 2 listas con datos para el gráfico
x = ['Python', 'Java', 'C', 'C++', 'Matlab', 'Octave']
usuarios = [10, 7, 15, 8, 9, 7]

# Uso de la capa de artista
fig = plt.figure() # Se crea una figura
ax = fig.add_subplot(111) # Nuevos ejes creados
#el número 111 significa que sólo queremos una trama


# Interfaz orientada a objetos
# ax es el objeto y los elementos de gráfico se agregan utilizando diferentes métodos de este objeto
ax.bar(x, usuarios)
ax.set_title('Usuarios de Lenguajes de Programación')
ax.set_xlabel('Lenguaje de Programación')
ax.set_ylabel('Número de usuarios')
```

Out[9]:
```
Text(0, 0.5, 'Número de usuarios')
```



Ejemplo que muestra la funcionalidad de la capa Scripting de Matplotlib

In [10]:
```
# Importamos matplotlib
import matplotlib.pyplot as plt

# Creamos 2 listas con datos para el gráfico
x = ['Python', 'Java', 'C', 'C++', 'Matlab', 'Octave']
usuarios = [10, 7, 15, 8, 9, 7]

# Uso de la capa de scripting
# No hay necesidad de crear figura y eje antes de trazar
plt.bar(x, usuarios)
plt.title('Usuarios de Lenguajes de Programación')
plt.xlabel('Lenguaje de Programación')
plt.ylabel('Número de usuarios')
plt.show()
```



De los dos ejemplos dados, las visualizaciones generadas son las mismas. Por lo tanto, el uso de la *interfaz pyplot* o la *interfaz orientada a objetos* es completamente la elección del usuario.

Resumiendo:

- **Un back-end se ocupa del dibujo real.**
- **Un grupo de artistas, describe cómo se organizan los datos.**
- **Y una capa de scripting, crea esos gráficos**

Configuración de estilos

Usaremos la directiva **plt.style** para elegir los estilos estéticos apropiados para nuestras figuras.

In [11]:
```
plt.style.use('classic')
```

show() o no show ()? Cómo mostrar sus gráficos

La forma en que ve sus gráficos Matplolib depende del contexto. El mejor uso de Matplotlib difiere según cómo lo esté usando; aproximadamente, los tres contextos aplicables usan Matplotlib en un script, en una terminal de IPython o en un cuaderno de IPython.

script

Si está utilizando Matplotlib desde un script, la función es **plt.show()**, esta inicia un ciclo de eventos, busca todos los objetos de figura actualmente activos y abre una o más ventanas interactivas que muestran su figura o figuras.

Esta función debe interactuar con el backend gráfico interactivo del sistema. Los detalles de esta operación pueden variar mucho de un sistema a otro e incluso de una instalación a otra.

Hay que tener en cuenta que plt.show() debe usarse solo una vez por sesión de Python, y se ve con mayor frecuencia al final del script. Múltiples comandos show() pueden conducir a un comportamiento impredecible dependiente del backend.

Trazar desde un shell de IPython

Puede ser muy conveniente utilizar Matplotlib de forma interactiva dentro de un shell de IPython. Este está diseñado para funcionar bien con Matplotlib si especifica el modo Matplotlib. Para habilitar este modo, puede usar el comando mágico %matplotlib después de iniciar ipython

```
%matplotlib
import matplotlib.pyplot as plt
```

En este punto, cualquier comando plt plot hará que se abra una ventana de figura y se pueden ejecutar más comandos para actualizar el gráfico. Algunos cambios (como la modificación de propiedades de líneas que ya están dibujadas) no se dibujarán automáticamente; para forzar una actualización, use plt.draw(). No es necesario usar plt.show() en el modo Matplotlib

Trazado desde un cuaderno de IPython

El cuaderno de IPython es una herramienta de análisis de datos interactiva basada en navegador que puede combinar narrativa, código, gráficos, elementos HTML y mucho más en un solo documento ejecutable. El trazado interactivo dentro de un cuaderno de IPython se puede hacer con el comando **%matplotlib** y funciona de manera similar al shell de IPython. En el cuaderno de IPython, también tiene la opción de incrustar gráficos directamente en el cuaderno, con dos opciones posibles

```
%matplotlib notebook: dará lugar a gráficos interactivos incrustados

%matplotlib inline: generará imágenes estáticas de su gráfico incrustadas
```

Después de ejecutar

```
%matplotlib inline
```

(solo debe hacerlo una vez por núcleo/sesión), cualquier celda dentro del cuaderno que cree un gráfico incrustará una imagen PNG del gráfico resultante

Modos de visualización

Si bien el modo interactivo está desactivado de forma predeterminada, se puede verificar su estado ejecutando:

- **plt.rcParams['interactive'] o, plt.isinteractive(),**

y activarlo y desactivarlo con:

- **plt.ion() y plt.ioff(),**

En algunos ejemplos podemos encontrar la presencia de plt.show() al final de un fragmento de código. El propósito principal de plt.show(), es "mostrar" (abrir) la figura cuando se está ejecutando con el modo interactivo desactivado.

Sintetizando:

- **Si el modo interactivo está activado, no es necesario plt.show(), y las imágenes se visualizarán y se actualizarán a medida que se haga referencia.**
- **Si el modo interactivo está desactivado, plt.show() deberá mostrar una figura y plt.draw() actualizar un gráfico.**

In [12]:
```
plt.rcParams['interactive']
```

Out[12]: True

In [13]:
```
plt.isinteractive()
```

Out[13]: True

In [14]:
```
plt.ioff()
```

Out[14]: <matplotlib.pyplot._IoffContext at 0x2a24a897b20>

In [15]:
```
plt.rcParams['interactive']
```

Out[15]: False

In [16]:
```
plt.ion()
```

Out[16]: <matplotlib.pyplot._IonContext at 0x2a24977f310>

In [17]:
```
plt.isinteractive()
```

Out[17]: True

Función gráfica

Un trazado tiene dos ejes, un eje X, horizontal y un eje Y, vertical. Utilizamos la capa de scripting pyplot como plt. Como se mencionó, todas las funciones que se ejecutarán con el módulo pyplot forman parte de la capa de scripting de la arquitectura.

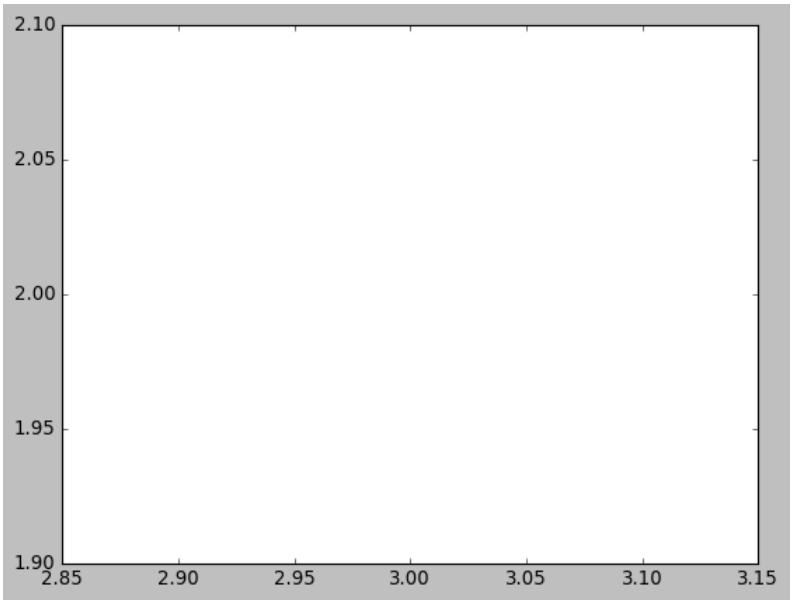
Veamos a la función gráfica mirando el documento [Jupyter Pager.pdf].

Esta declaración de función de Python: plt.plot(*args, **kwargs), significa que en:

- ***args** significa que admite cualquier número de argumentos sin nombre.
- ****kwargs** significa que admite cualquier número de argumentos con nombre.

Esto hace que la declaración de función sea muy flexible ya que puede pasar básicamente cualquier número de argumentos, nombrados o no, pero hace que sea difícil saber qué es un argumento apropiado. Al leer, vemos que los argumentos se interpretarán como pares x, y. Así que intentemos con un solo punto de datos en la posición 3,2.

In [18]:
```
import matplotlib.pyplot as plt

plt.plot(3,2)
plt.show()
```

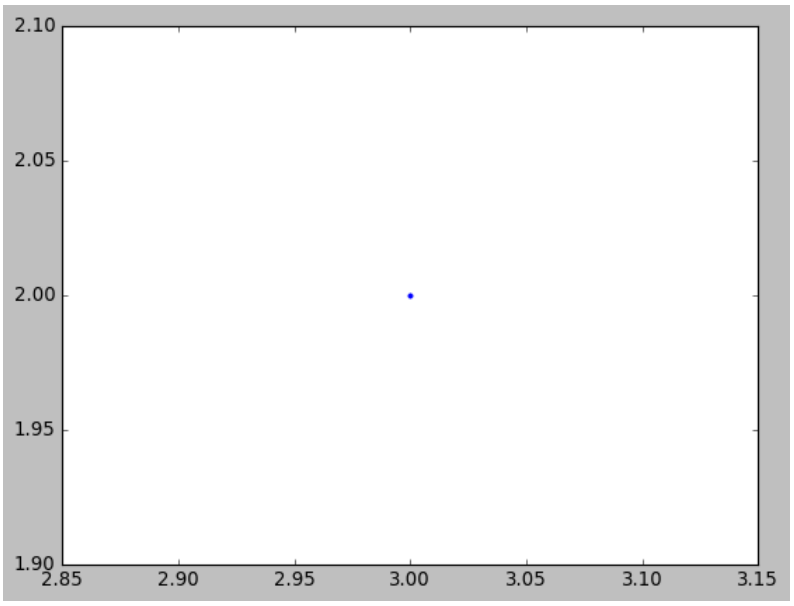


Se visualiza que el valor devuelto está alineado con el objeto, y vemos nuestra primera figura. Sin embargo, no vemos los puntos de datos.

Usemos un punto para un punto, y vemos que el punto de datos aparece.

In [19]:
```
import matplotlib.pyplot as plt
plt.plot(3,2, '.')
```

Out[19]:
```
[<matplotlib.lines.Line2D at 0x2a249447c40>]
```



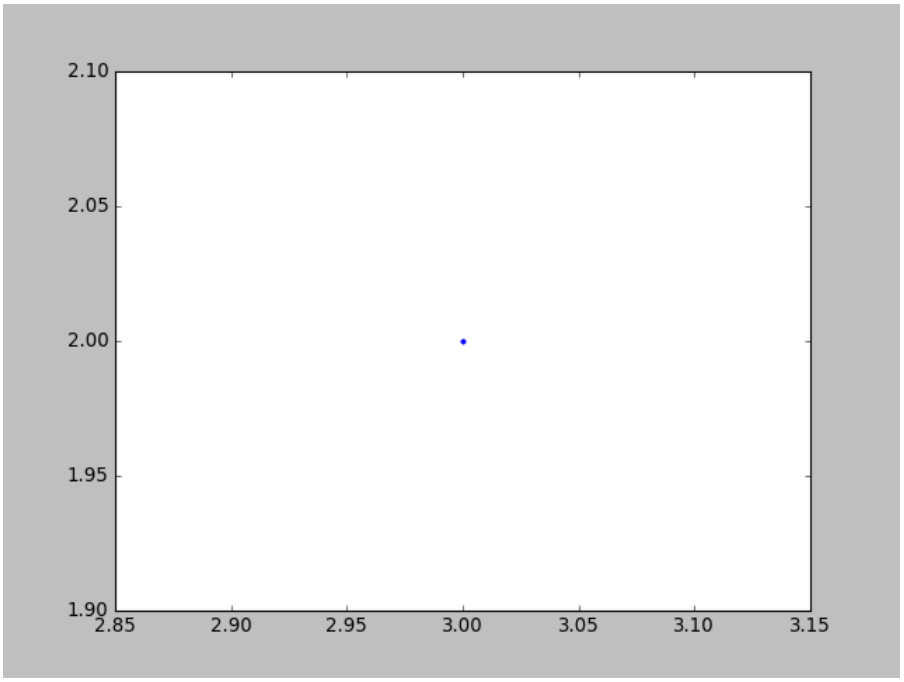
El tercer argumento debe ser una cadena que significa cómo queremos que se renderice ese punto de datos.

El back-end de los Jupyter Notebooks, no es capaz de renderizar esto directamente, ya que espera que la capa de scripting pyplot, haya creado todos los objetos, por eso guardamos la figura en un archivo png:

In [20]:
```
from matplotlib.backends.backend_agg import FigureCanvasAgg
from matplotlib.figure import Figure

fig = Figure()
canvas = FigureCanvasAgg(fig)
ax = fig.add_subplot(111)
ax.plot(3,2, '.')
canvas.print_png('img/test.png')
```

Para visualizar la imagen renderizada en Jupyter notebook debemos escribir lo siguiente:

In [21]:
```
%%html
<img src='img/test.png'/>
```



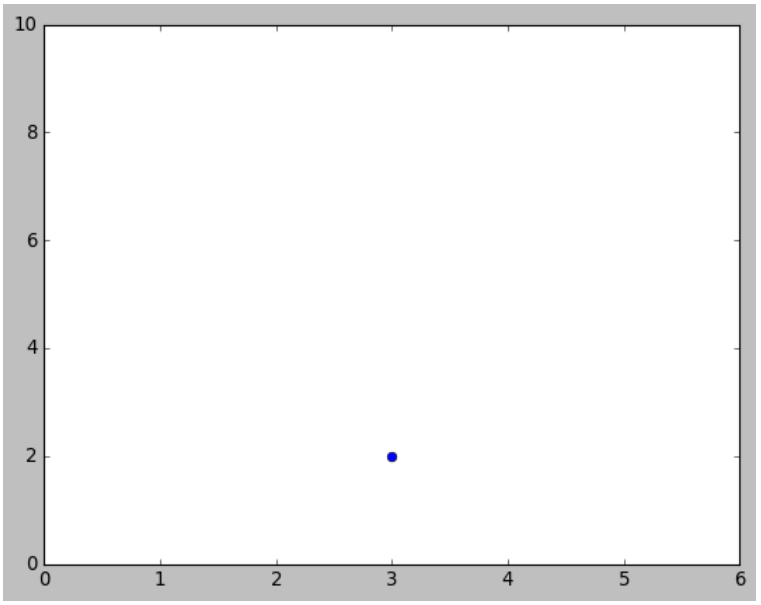
Cuando hacemos una llamada a pyplot con plt.plot, la capa de secuencias de comandos busca ver si hay una figura que existe actualmente, si no la hay, crea una nueva.

A continuación, devuelve los ejes de la figura. Podemos obtener acceso a la figura usando:

- la función **gcf()**, que significa obtener la figura actual de pyplot, y con
- la función **gca()** obtener acceso a los ejes actuales

In [22]:
```
import matplotlib.pyplot as plt

plt.figure()
plt.plot(3,2, 'o')
ax = plt.gca()
ax.axis([0,6,0,10])
```

Out[22]:
```
(0.0, 6.0, 0.0, 10.0)
```



axis tiene 4 parámetros:

- un valor mínimo para x, al cual le dimos 0,
- un valor máximo para x, que le dimos 6.

Luego,

- los valores mínimos y máximos para y, que le dimos 0 y 10.

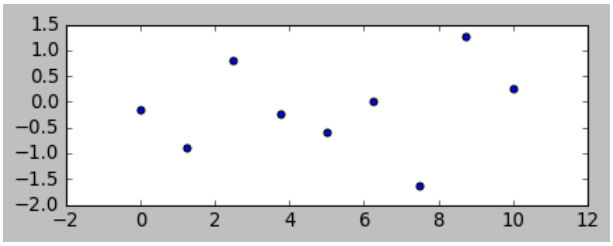
Como estamos haciendo esto con la capa de scripting, si estamos trabajando en Jupyter una vez que ejecutamos la celda, se renderiza con el back-end nbAgg .

Pyplot: plt.gcf ()

GCF son las siglas de Get Current Figure. **plt.gcf()** le permite obtener una referencia a la figura actual cuando usamos pyplot.

Ejemplo: cambiar el tamaño de la imagen usando fig.set_size_inches()

In [23]:
```
import matplotlib.pyplot as plt
import numpy as np #pip install numpy


x = np.linspace(0,10,9)
y = np.random.randn(9)

plt.scatter(x,y)
fig = plt.gcf()
fig.set_size_inches(6,2)
```



Usamos la función plt.gcf () para obtener una referencia a la figura actual y luego llamamos al método set_size_inches () en ella.

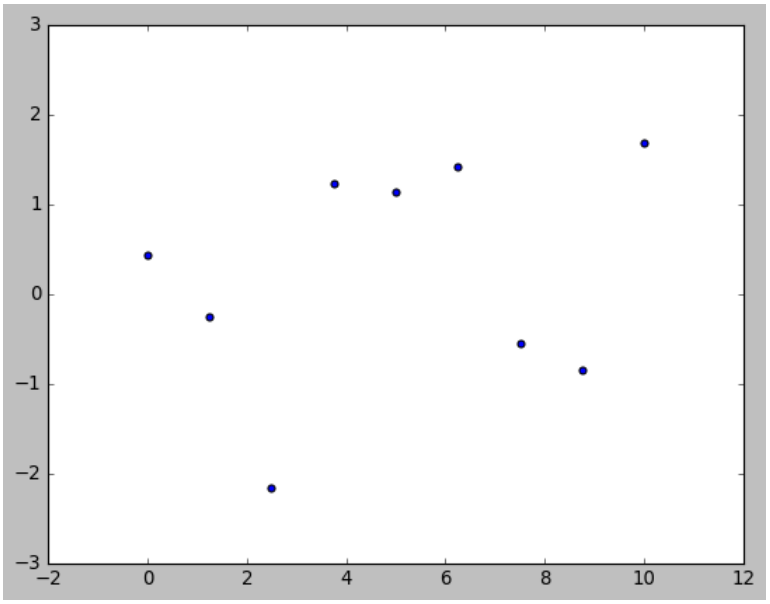
Pyplot: plt.gca ()

GCA son las siglas de Get Current Axes. Al igual que con plt.gcf(), puede usarse **plt.gca()** para obtener una referencia a los ejes actuales, si necesitamos cambiar los límites en el eje y.

Ejemplo:

In [24]:
```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0,10,9)
y = np.random.randn(9)

plt.scatter(x,y)
axis = plt.gca()
axis.set_ylim(-3,3)
```

Out[24]: (-3.0, 3.0)



Tener en cuenta que el eje y, ahora varía de -3 a 3. Obtuvimos una referencia al eje actual llamando a plt.gca(). Luego llamamos al método set_ylim () para ajustar los límites para ese eje.

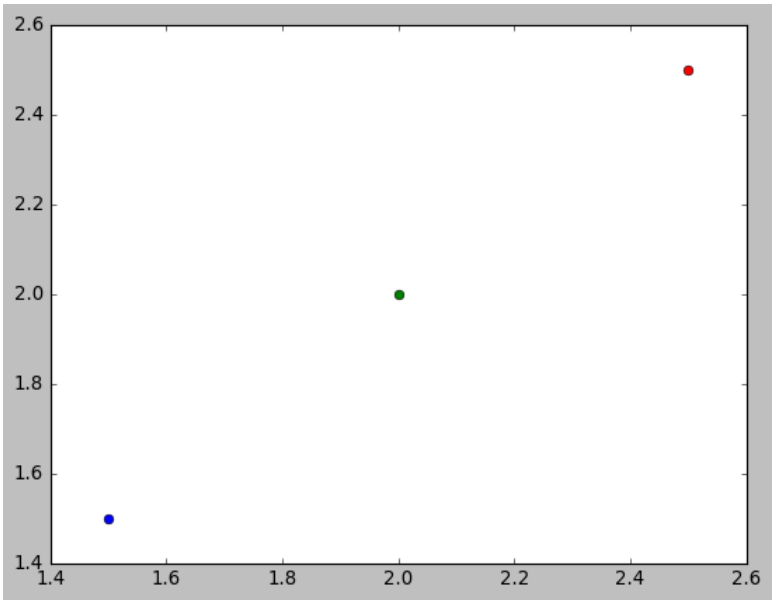
Se puede agregar Artistas a un objeto de ejes en cualquier momento, pyplot hace eso por nosotros cuando llamamos a la función gráfica.

Si hacemos llamadas posteriores a la función gráfica, esto agregará más datos a nuestro gráfico.

Puede verse que cuando se hace esto, los puntos se representan en diferentes colores, ya que los ejes los reconocen como series de datos diferentes:

In [25]:
```
import matplotlib.pyplot as plt

plt.figure()
plt.plot(1.5, 1.5, 'o')
plt.plot(2,2,'o')
plt.plot(2.5,2.5,'o')
```

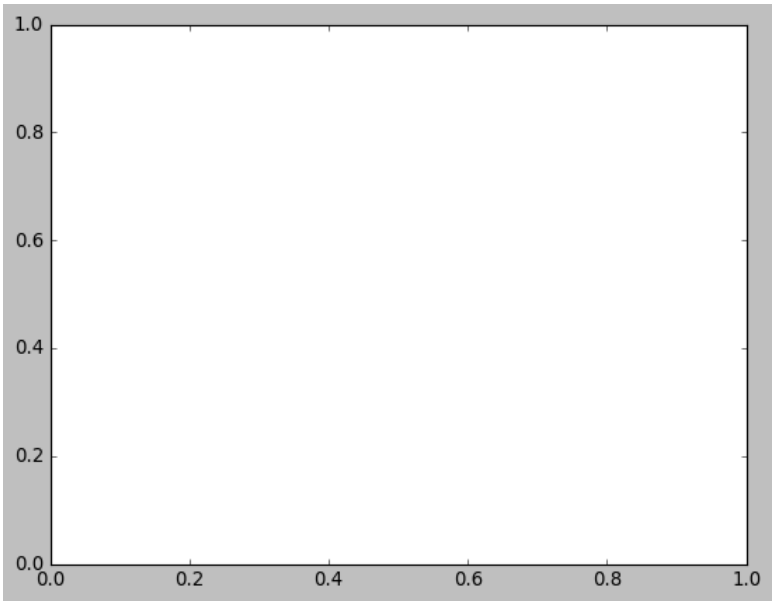
Out[25]:
```
[<matplotlib.lines.Line2D at 0x2a24aa38340>]
```



Podemos profundizar en el objeto de ejes para obtener todos los objetos secundarios que contiene los ejes. Si agregamos el siguiente código al anterior

In [26]:
```
ax = plt.gca()
ax.get_children()

# Eso produce la siguiente salida:
```

Out[26]:
```
[<matplotlib.spines.Spine at 0x2a24aa45220>,
 <matplotlib.spines.Spine at 0x2a24aa4c040>,
 <matplotlib.spines.Spine at 0x2a24aa52e20>,
 <matplotlib.spines.Spine at 0x2a24aa529a0>,
 <matplotlib.axis.XAxis at 0x2a24aa45760>,
 <matplotlib.axis.YAxis at 0x2a24aa6e4f0>,
 Text(0.5, 1.0, ''),
 Text(0.0, 1.0, ''),
 Text(1.0, 1.0, ''),
 <matplotlib.patches.Rectangle at 0x2a24aa81f40>]
```



Tipos gráficos

Scatterplots

Matplotlib tiene una serie de métodos de trazado útiles en la capa de secuencias de comandos que corresponden a diferentes tipos de gráficas. Veamos algunos de los más importantes considerando que:

pyplot recupera la figura actual con la función gcf() y obtiene el eje actual con la función gca(), además realiza un seguimiento de los objetos del eje.

pyplot refleja la API de los objetos del eje, por lo tanto, puede llamarse a la función gráfica con el módulo pyplot, pero esto es llamar a las funciones de trazado del eje por debajo.

La declaración de la mayoría de las funciones en matplotlib termina con un conjunto abierto de argumentos de palabras clave.

Existen propiedades diferentes que puede controlarse a través de estos argumentos.

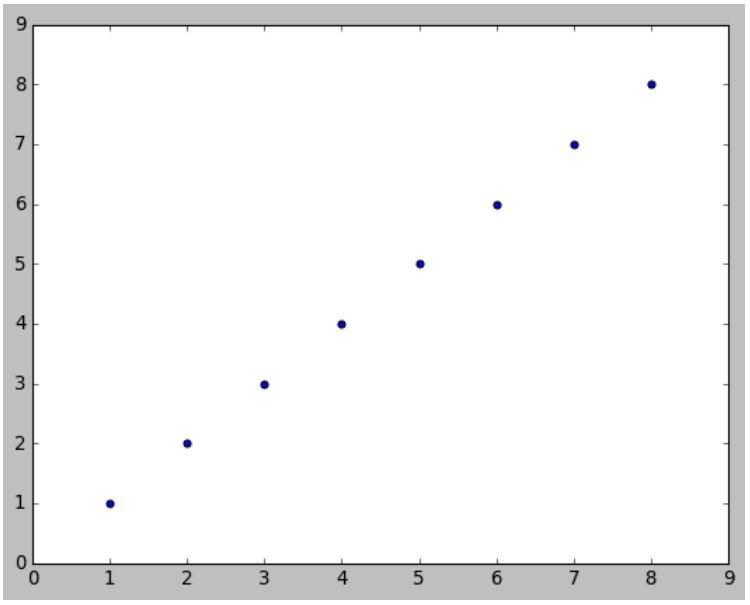
Tramas de dispersión

Una gráfica de dispersión es una gráfica de 2 dimensiones, similar a las gráficas de líneas.

La función de dispersión toma un valor del eje x como primer argumento y el valor del eje y como el segundo. En este ejemplo, ambos argumentos son iguales, por lo tanto obtenemos una buena alineación diagonal de los puntos:

In [27]:
```
import matplotlib.pyplot as plt

x = [1,2,3,4,5,6,7,8]
y = x
plt.figure()
plt.scatter(x, y)
```

Out[27]:
```
<matplotlib.collections.PathCollection at 0x2a24aafa9a0>
```

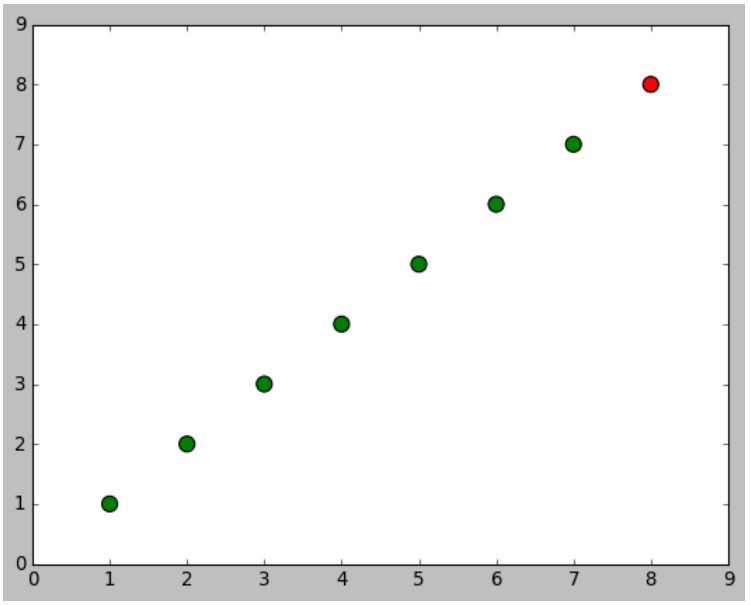


En esta trama puede verse una línea diagonal y matplotlib ha dimensionado nuestros ejes en consecuencia. Pero la dispersión no representa elementos como una serie.

Podemos pasarle una lista de colores que corresponden a los puntos dados. Vamos a utilizar una **aritmética de lista** para crear una nueva lista sólo por debajo del número de puntos de datos que necesitamos y establecer todos los valores en verde. Luego agregaremos un valor final de rojo:

In [28]:
```
import matplotlib.pyplot as plt

x = [1,2,3,4,5,6,7,8]
y = x
colors = ['green'] * (len(x)-1)
colors.append('red')
plt.figure()
plt.scatter(x, y, s=100, c=colors)
```

Out[28]:
```
<matplotlib.collections.PathCollection at 0x2a24ad37490>
```



s es el tamaño de los puntos. Si queremos observar como quedó la lista de colores con la operación:

In [29]:
```
colors
```

Out[29]:
```
['green', 'green', 'green', 'green', 'green', 'green', 'green', 'red']
```

Función zip

El método **zip** toma una serie de iterables y crea tuplas fuera de ellos, haciendo coincidir elementos basados en el índice.

Aplicaremos la función zip a 2 listas de números. Cuando ejecutamos observamos que hay una lista de tuplas en pares. Es común almacenar datos de puntos como tuplas:

In [30]:
```
zip_generador = zip([1,2,3,4,5],[6,7,8,9,10])
print(list(zip_generador))
```

```
[(1, 6), (2, 7), (3, 8), (4, 9), (5, 10)]
```

zip tomará el primero de cada lista y creará una tupla y así sucesivamente. Como zip tiene una evaluación perezosa porque en realidad es un generador en Python 3, significa que necesitamos utilizar funciones de lista si queremos ver los resultados de iterar sobre zip.

Si queremos volver a convertir los datos en 2 listas, una con el componente x y otra con el componente y, podemos utilizar el parámetro de desempaquetado con zip. Cuando se pasa una lista o un intervalo, generalmente a una función y se antepone con un **(*)**, cada elemento se saca del iterable y se pasa como un argumento separado:

In [31]:
```
zip_generador = zip([1,2,3,4,5],[6,7,8,9,10])
x, y = zip(*zip_generador)
x, y
```

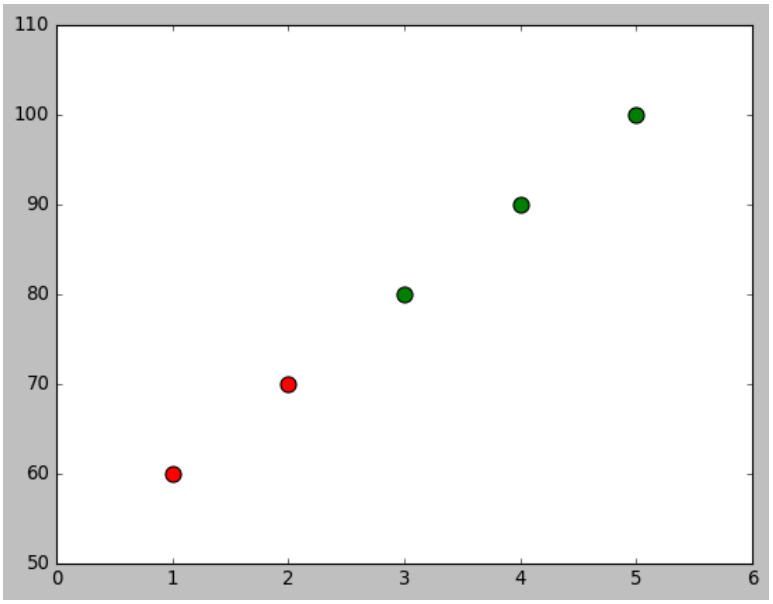
Out[31]:
```
((1, 2, 3, 4, 5), (6, 7, 8, 9, 10))
```

Tomamos 2 listas y trazamos una nueva figura usando dispersión. En lugar de trazarlos como una serie de datos, vamos a cortar las listas y trazarlas como 2 series de datos:

In [32]:

```
import matplotlib.pyplot as plt

zip_generator=zip([1,2,3,4,5],[60,70,80,90,100])
x,y=zip(*zip_generator)
plt.figure()

plt.scatter(x[:2],y[:2],s=100,c='red',label='Ingresos más bajos')
plt.scatter(x[2:],y[2:],s=100,c='green',label='Ingresos más altos')

#x[:2],y[:2] -> ((1, 2), (60, 70))
#x[2:],y[2:] -> ((3, 4, 5), (80, 90, 100))
```

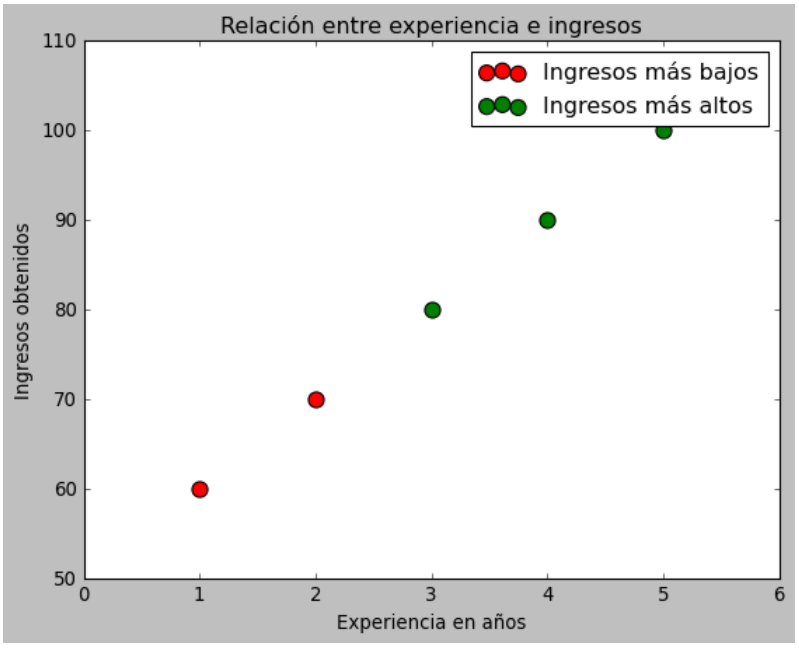
Out[32]: <matplotlib.collections.PathCollection at 0x2a24abf5b50>



Podemos colorear cada serie con un solo valor, o cambiar el color o la transparencia de una serie completa, o puntos de datos individuales y tenemos la capacidad de etiquetar la serie de datos que es útil para construir una leyenda. El eje tiene etiquetas para explicar lo que representan las unidades que describen y los gráficos tienen títulos también.

In [33]:
```
import matplotlib.pyplot as plt

zip_generator=zip([1,2,3,4,5],[60,70,80,90,100])
x,y=zip(*zip_generator)
plt.figure()

plt.scatter(x[:2],y[:2],s=100,c='red',label='Ingresos más bajos')
plt.scatter(x[2:],y[2:],s=100,c='green',label='Ingresos más altos')
plt.xlabel('Experiencia en años')
plt.ylabel('Ingresos obtenidos')
plt.title('Relación entre experiencia e ingresos')
plt.legend()
```

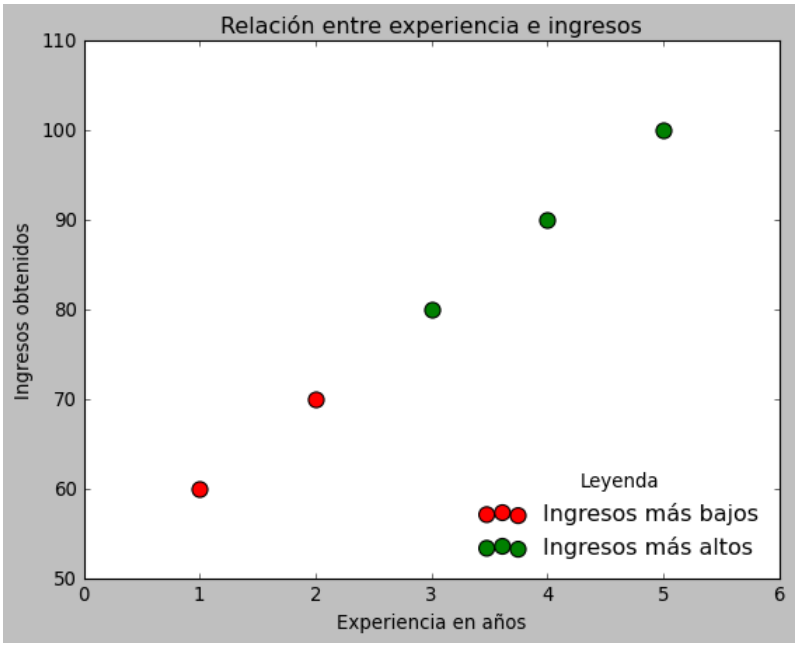
Out[33]:
```
<matplotlib.legend.Legend at 0x2a249388790>
```



Dado que pyplot refleja gran parte de la API del eje, podemos hacer llamadas directamente en pyplot, matplotlib coloca la leyenda en la esquina superior izquierda. En la documentación de la leyenda de matplotlib, puede verse que hay una serie de parámetros diferentes y uno de ellos se llama **loc**. Pondremos la leyenda en la esquina inferior derecha del eje, quitamos el marco y agregamos un título:

In [34]:
```
import matplotlib.pyplot as plt

zip_generator=zip([1,2,3,4,5],[60,70,80,90,100])
x,y=zip(*zip_generator)
plt.figure()

plt.scatter(x[:2],y[:2],s=100,c='red',label='Ingresos más bajos')
plt.scatter(x[2:],y[2:],s=100,c='green',label='Ingresos más altos')
plt.xlabel('Experiencia en años')
plt.ylabel('Ingresos obtenidos')
plt.title('Relación entre experiencia e ingresos')
plt.legend(loc=4,frameon=False,title='Leyenda')
```

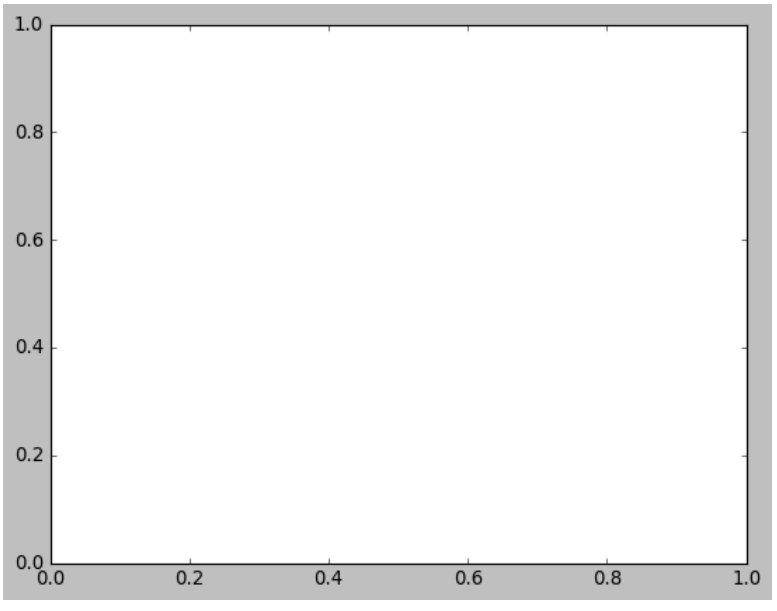
Out[34]:
```
<matplotlib.legend.Legend at 0x2a249871160>
```



Como la leyenda es un Artist para ver los hijos podemos acceder mediante el código que vimos anteriormente:

In [35]:
```
plt.gca().get_children()
```

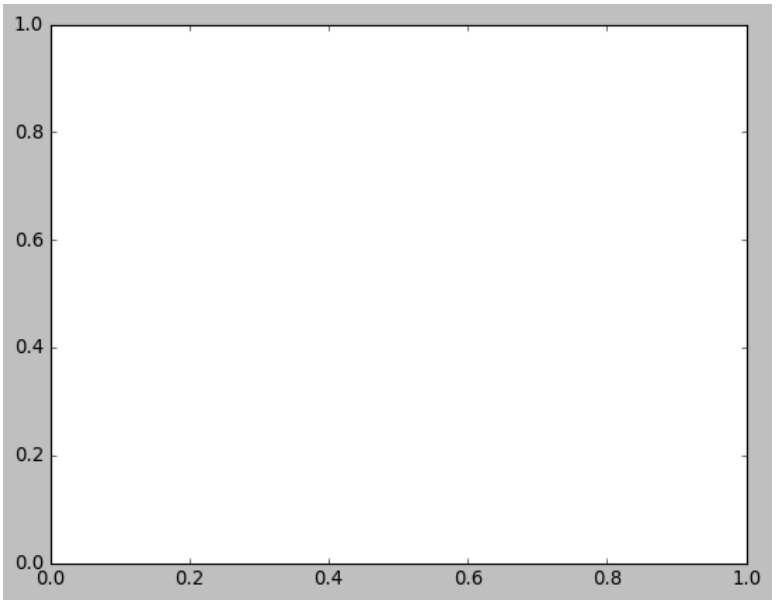
Out[35]:
```
[<matplotlib.spines.Spine at 0x2a24987c610>,
 <matplotlib.spines.Spine at 0x2a24987c6d0>,
 <matplotlib.spines.Spine at 0x2a24987cc10>,
 <matplotlib.spines.Spine at 0x2a24987c6a0>,
 <matplotlib.axis.XAxis at 0x2a24987c190>,
 <matplotlib.axis.YAxis at 0x2a24987cb20>,
 Text(0.5, 1.0, ''),
 Text(0.0, 1.0, ''),
 Text(1.0, 1.0, ''),
 <matplotlib.patches.Rectangle at 0x2a2497d5430>]
```



Es importante entender cómo funciona la biblioteca. Con una función recursiva que toma un artista y un parámetro de profundidad recorremos los hijos de legend:

In [36]:
```
leyenda=plt.gca().get_children()[-2]
leyenda
```

Out[36]:
```
Text(1.0, 1.0, '')
```



In [37]:
```
from matplotlib.artist import Artist

def recorrer(art, depth=0):

    if isinstance(art, Artist):
        print(" " * depth + str(art))
        for hijo in art.get_children():
            recorrer(hijo, depth+2)

recorrer(leyenda)
```

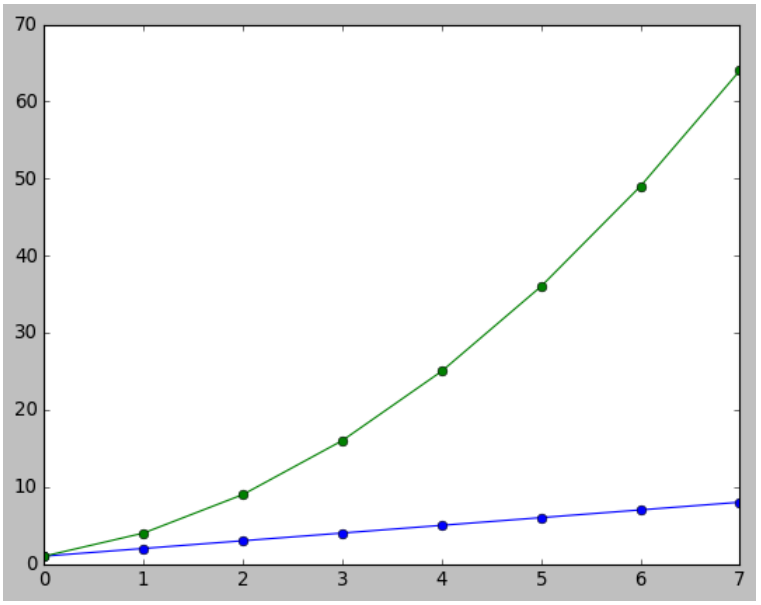
```
Text(1.0, 1.0, '')
```

La función comprueba si el objeto es un artista, imprime su nombre de cadena. Luego repite y aumenta la profundidad. Puede verse que el artista de la leyenda se compone de una serie de diferentes **offsetboxes** para dibujar, así como **TextAreas y PathCollections**. Las llamadas a la interfaz de secuencias de comandos, simplemente crea figuras, subgráficas y ejes. Luego se cargan esos ejes con varios artistas, que el backend renderiza en la pantalla o en un archivo.

Line plots

In [38]:
```
import matplotlib.pyplot as plt
import numpy as np

datos_lins = np.array([1,2,3,4,5,6,7,8])
datos_cuads = datos_lins**2
plt.figure()
plt.plot(datos_lins, '-o', datos_cuads , '-o')
```

Out[38]:
```
[<matplotlib.lines.Line2D at 0x2a2495b44f0>,
 <matplotlib.lines.Line2D at 0x2a2495b4460>]
```



Vemos el resultado como 2 series de datos, en la parte inferior la lineal y en la parte superior la exponencial.

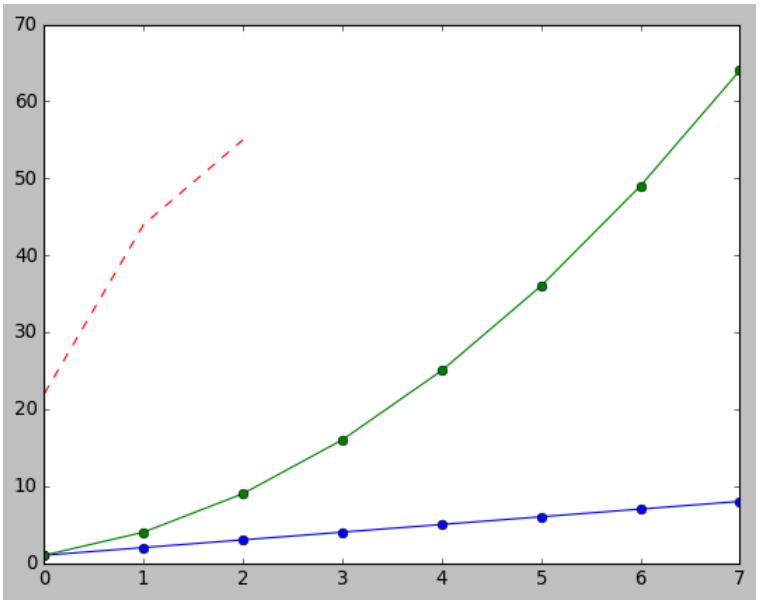
Ambas están usando puntos porque usamos la bandera **"-o"**, elemento nuevo respecto de las tramas de dispersión.

- Primero, solo dimos valores de los ejes a la llamada de trazado, no valores de ejes x. En su lugar, la función de trama sabe que queríamos usar el índice de la serie como el valor x.
- En segundo lugar vemos que la gráfica identifica esto como 2 series de datos y que los colores de los datos de las series son diferentes.

Matplotlib inventa un mini lenguaje basado en cadenas para el formato de uso común. Por ejemplo, podríamos usar una ‘s’ dentro de la cadena de formato que trazaría otro punto usando un marcador cuadrado. O podríamos usar una serie de guiones y puntos para identificar que una línea debe ser discontinua en lugar de sólida:

In [39]:
```
import matplotlib.pyplot as plt
import numpy as np

datos_lins = np.array([1,2,3,4,5,6,7,8])
datos_cuads = datos_lins**2

plt.figure()

plt.plot(datos_lins, '-o', datos_cuads, '-o')
plt.plot([22,44,55],'--r')
```

Out[39]:
```
[<matplotlib.lines.Line2D at 0x2a2495c00a0>]
```

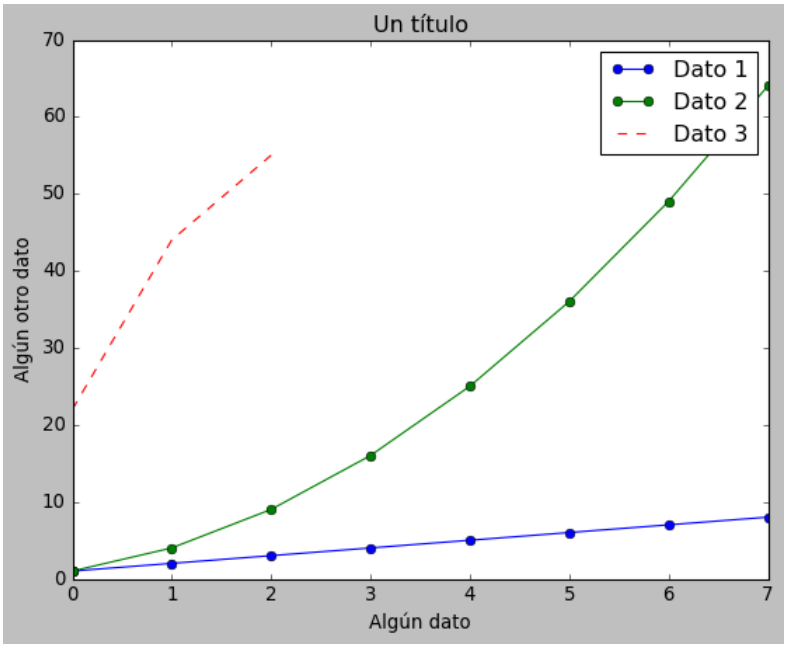


Puede utilizarse las funciones de ejes creando etiquetas para los ejes y para la figura como un todo.

También puede crearse una leyenda, pero hay que tener en cuenta que, dado que no etiquetamos los puntos de datos como lo hicimos con la gráfica de dispersión, necesitamos crear entradas de leyenda cuando agregamos la leyenda misma:

In [40]:
```
import matplotlib.pyplot as plt
import numpy as np

datos_lins = np.array([1,2,3,4,5,6,7,8])
datos_cuads = datos_lins**2
plt.figure()

plt.plot(datos_lins, '-o', datos_cuads , '-o')
plt.plot([22,44,55],'--r')
plt.xlabel('Algún dato')
plt.ylabel('Algún otro dato')
plt.title('Un título')
plt.legend(['Dato 1', 'Dato 2', 'Dato 3'])
```

Out[40]:
```
<matplotlib.legend.Legend at 0x2a2494d5c10>
```



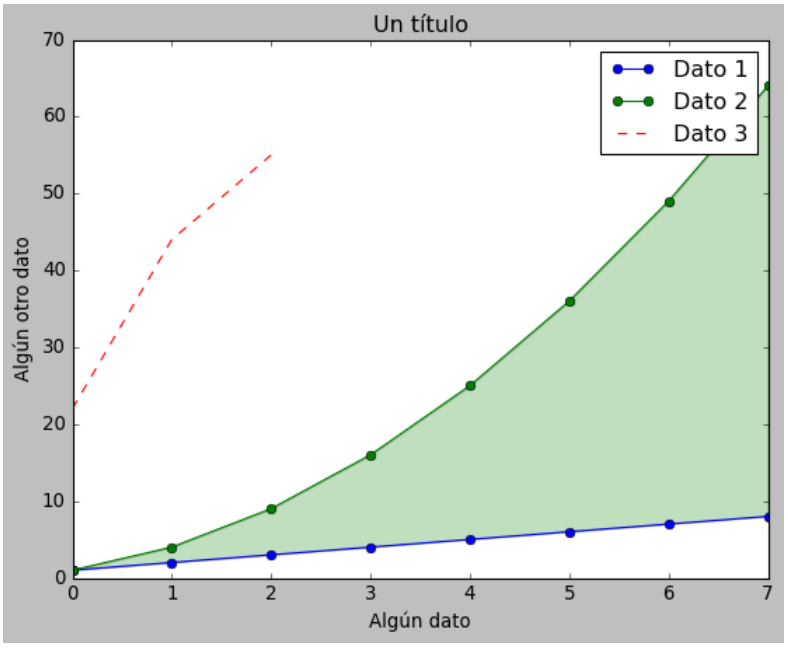
Función de relleno

La función de relleno no es específica para las gráficas de línea, pero se utiliza frecuentemente. Imaginemos que queremos resaltar la diferencia entre las curvas color naranja y celeste. Podríamos hacer que pinte de un color entre estas series usando la función de relleno.

Primero obtenemos los ejes actuales, luego indicamos el rango de valores x que queremos pintar. No especificamos ningún valor x en la llamada a trazar, por lo que solo utilizamos el mismo rango de puntos de datos actual. Luego pondremos nuestros límites inferiores y superiores junto con el color que queremos pintar e incluiremos un valor de transparencia.

In [41]:
```
import matplotlib.pyplot as plt
import numpy as np

datos_lins = np.array([1, 2, 3, 4, 5, 6, 7, 8])
datos_cuads = datos_lins**2
plt.figure()

plt.plot(datos_lins, '-o', datos_cuads, '-o')
plt.plot([22, 44, 55], '--r')
plt.xlabel('Algún dato')
plt.ylabel('Algún otro dato')
plt.title('Un título')
plt.legend(['Dato 1', 'Dato 2', 'Dato 3'])

plt.gca().fill_between(range(len(datos_lins)),
datos_lins, datos_cuads,
facecolor='green', alpha=0.25)
```

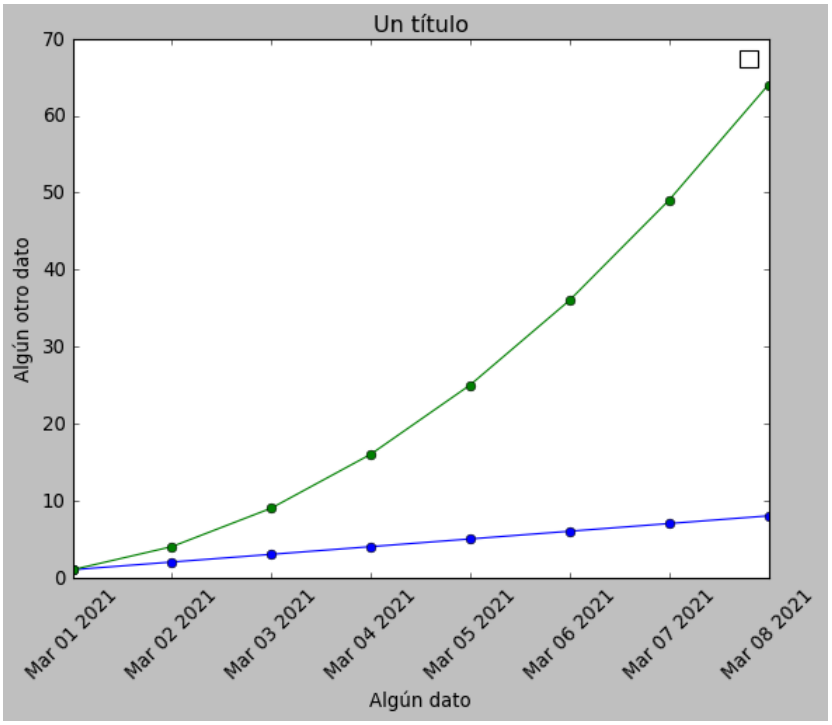
Out[41]:
```
<matplotlib.collections.PolyCollection at 0x2a24ad64910>
```



Frecuentemente, con gráficos de línea, se trabaja en forma de **fecha y hora** para los ejes x. Entonces cambiaremos nuestro eje x a una serie de 8 instancias de fecha y hora, en intervalos de un día. Primero creamos una nueva imagen:

In [42]:

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd #pip install pandas

datos_lins = np.array([1, 2, 3, 4, 5, 6, 7, 8])
datos_cuads = datos_lins**2
plt.figure()

plt.xlabel('Algún dato')
plt.ylabel('Algún otro dato')
plt.title('Un título')
plt.legend(['Dato 1', 'Dato 2', 'Dato 3'])

datos_observados = np.arange('2021-03-01','2021-03-09',dtype='datetime64[D]')
datos_observados = list(map(pd.to_datetime, datos_observados))

plt.plot(datos_observados, datos_lins, '-o', datos_observados, datos_cuads, '-o')
x = plt.gca().xaxis

for item in x.get_ticklabels():
    item.set_rotation(45)
```



NumPy es útil para trabajar operaciones con fechas y con Pandas existe una función llamada a **datetime** que permite convertir las fechas NumPy en fechas de biblioteca estándar, que es lo que matplotlib espera.

Usamos la función **map** de la biblioteca estándar que devuelve un iterador, matplotlib no puede manejar el iterador, por lo que necesitamos convertirlo en una lista.

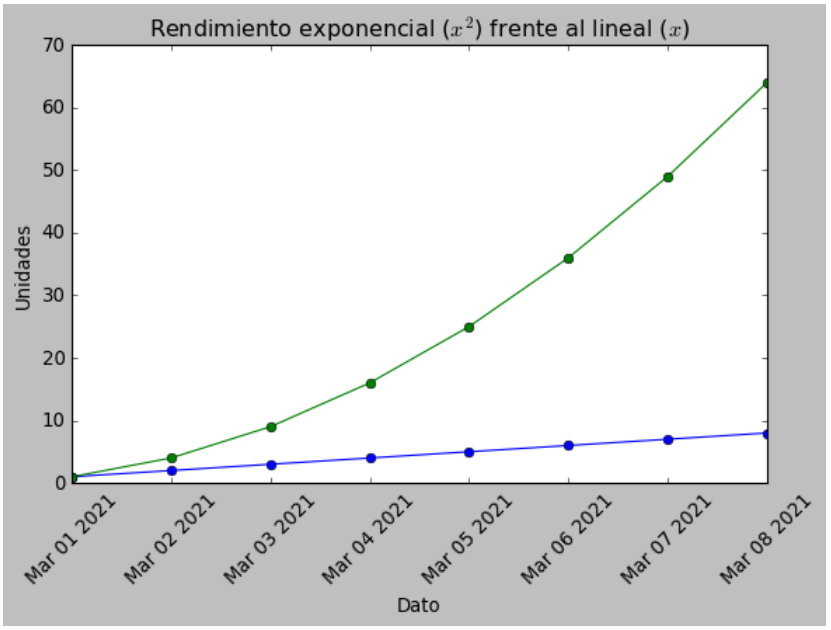
Para que las fechas no se apilen, podemos obtener un solo eje usando las propiedades del eje x o del eje y del objeto de ejes que podemos obtener con gca().

Existen muchas propiedades del objeto de ejes, por ejemplo, se puede obtener las líneas de cuadrícula, las ubicaciones de ticks para las etiquetas principales y secundarias, etc.

Cada una de las etiquetas es un objeto de texto que a su vez es un artista. Esto significa que se pueden utilizar una serie de funciones de artista diferentes. Una función específica del texto es la de rotación que puede cambiarse en función de grados.

In [43]:
```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

datos_lins = np.array([1, 2, 3, 4, 5, 6, 7, 8])
datos_cuads = datos_lins**2

plt.figure()

datos_observados = np.arange('2021-03-01','2021-03-09',dtype='datetime64[D]')
datos_observados = list(map(pd.to_datetime, datos_observados))

plt.plot(datos_observados, datos_lins, '-o',datos_observados, datos_cuads, '-o')
x = plt.gca().xaxis

for item in x.get_ticklabels():
    item.set_rotation(45)

plt.subplots_adjust(bottom=0.25)
ax = plt.gca()
ax.set_xlabel('Dato')
ax.set_ylabel('Unidades')
ax.set_title('Rendimiento exponencial ($x^2$) frente al lineal ($x$)')
```

Out[43]:
```
Text(0.5, 1.0, 'Rendimiento exponencial ($x^2$) frente al lineal ($x$)')
```

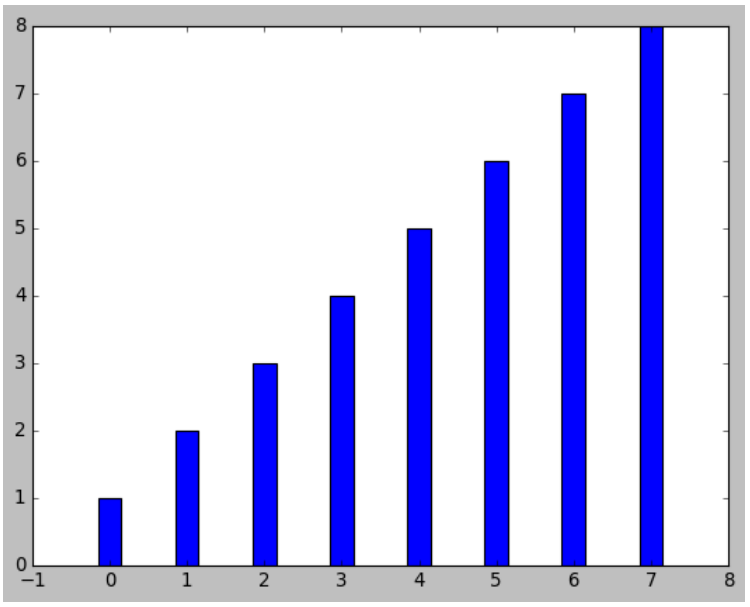


Bar charts

Matplotlib tiene soporte para varios tipos de gráficos de barras. El caso más general, trazamos un gráfico de barras enviando un parámetro de los componentes x, y un parámetro de la altura de la barra:

In [44]:
```
import matplotlib.pyplot as plt
import numpy as np

datos_lins = np.array([1, 2, 3, 4, 5, 6, 7, 8])
plt.figure()

xvals = range(len(datos_lins))
plt.bar(xvals, datos_lins, width = 0.3)
```

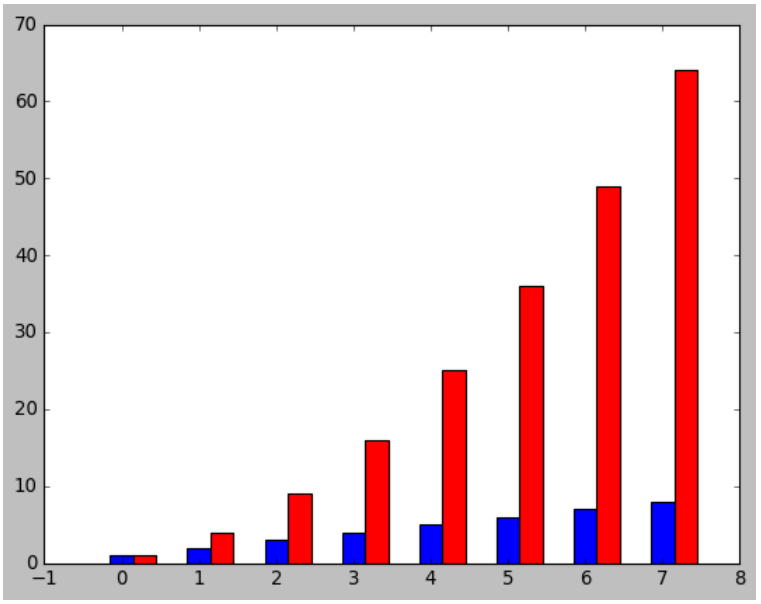
Out[44]:
```
<BarContainer object of 8 artists>
```



Para agregar una segunda barra, llamamos a la gráfica de barras nuevamente con nuevos datos, teniendo en cuenta que necesitamos ajustar el componente x para compensar la primera barra que trazamos:

In [45]:
```
import matplotlib.pyplot as plt
import numpy as np

datos_lins = np.array([1, 2, 3, 4, 5, 6, 7, 8])
datos_cuads = datos_lins**2

plt.figure()

xvals = range(len(datos_lins))
plt.bar(xvals, datos_lins, width = 0.3)
new_xvals = []

for item in xvals:
    new_xvals.append(item+0.3)

plt.bar(new_xvals, datos_cuads, width = 0.3 ,color='red')
```

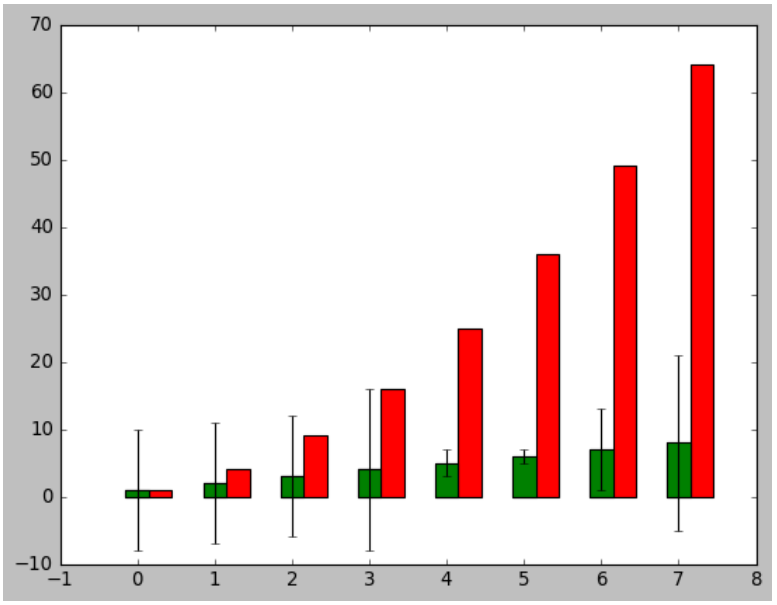
Out[45]:
```
<BarContainer object of 8 artists>
```



Las etiquetas se pueden centrar usando el parámetro **align**. Pueden agregarse barras de error a cada barra, utilizando el parámetro y error. Por ejemplo, cada uno de nuestros datos, en los datos lineales, podría ser en realidad un valor medio, calculado a partir de muchas observaciones diferentes. Creamos una lista de valores de error importando una función aleatoria:

In [46]:
```
import matplotlib.pyplot as plt
import numpy as np

datos_lins = np.array([1, 2, 3, 4, 5, 6, 7, 8])
datos_cuads = datos_lins**2
plt.figure()
xvals = range(len(datos_lins))
plt.bar(xvals, datos_lins, width = 0.3)

new_xvals = []
for item in xvals:
    new_xvals.append(item+0.3)
plt.bar(new_xvals, datos_cuads, width = 0.3 ,color='red')

from random import randint

linear_err = [randint(0,15) for x in range(len(datos_lins))]
plt.bar(xvals, datos_lins, width = 0.3, yerr=linear_err)
```

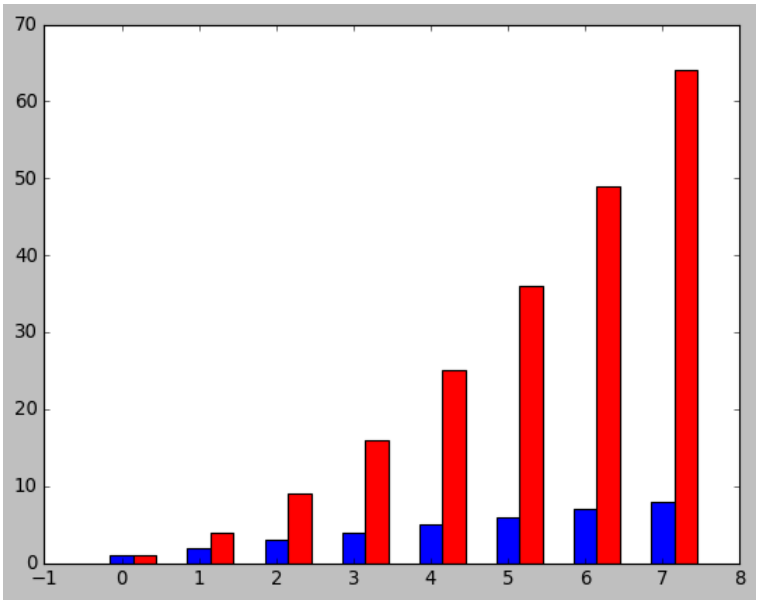
Out[46]: <BarContainer object of 8 artists>

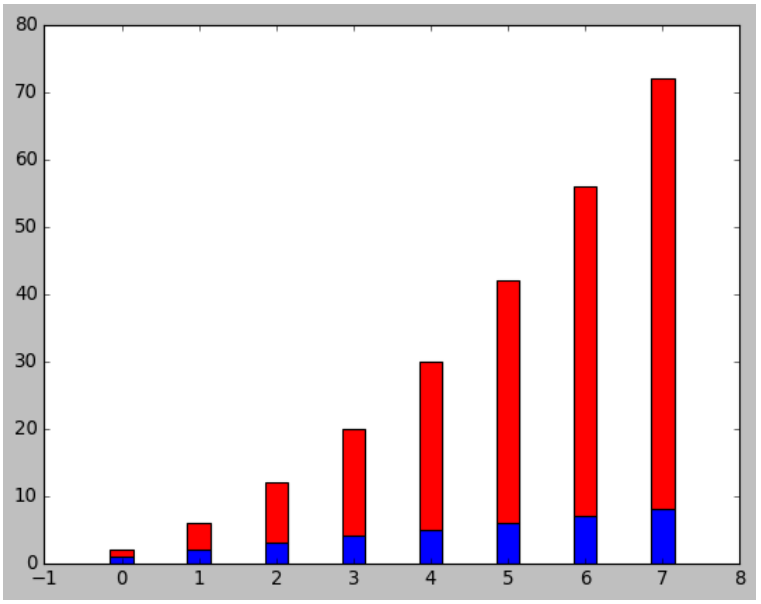


También podemos construir gráficos de barras apilados. Por ejemplo, si quisiéramos mostrar valores acumulativos mientras mantenemos la serie independiente, podríamos hacer esto estableciendo el parámetro inferior y nuestro segundo gráfico para que sea igual al primer conjunto de datos a trazar:

In [47]:
```
import matplotlib.pyplot as plt
import numpy as np

datos_lins = np.array([1, 2, 3, 4, 5, 6, 7, 8])
datos_cuads = datos_lins**2
# plt.ion()
plt.figure()
xvals = range(len(datos_lins))
plt.bar(xvals, datos_lins, width = 0.3)

new_xvals = []
for item in xvals:
    new_xvals.append(item+0.3)

plt.bar(new_xvals, datos_cuads, width = 0.3 ,color='red')
plt.figure()
xvals = range(len(datos_lins))
plt.bar(xvals, datos_lins, width = 0.3, color='b')
plt.bar(xvals, datos_cuads, width = 0.3,
bottom=datos_lins, color='r')
```

Out[47]: <BarContainer object of 8 artists>



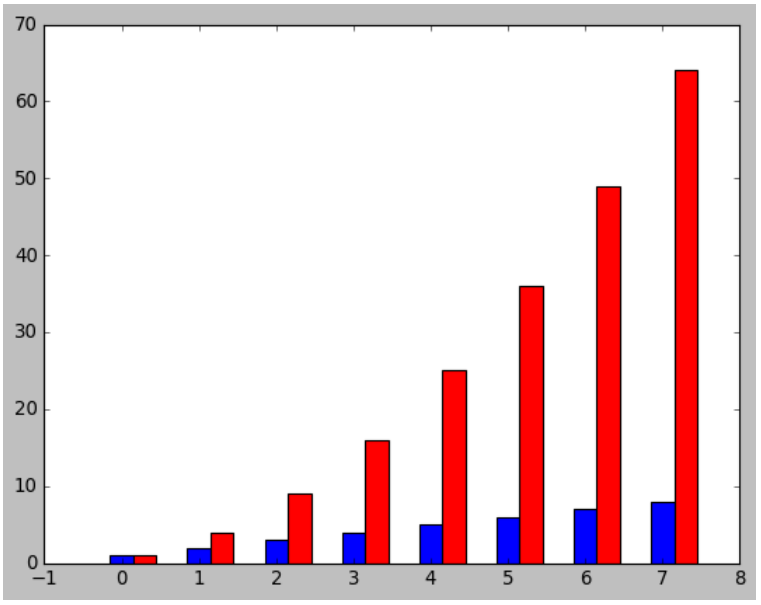


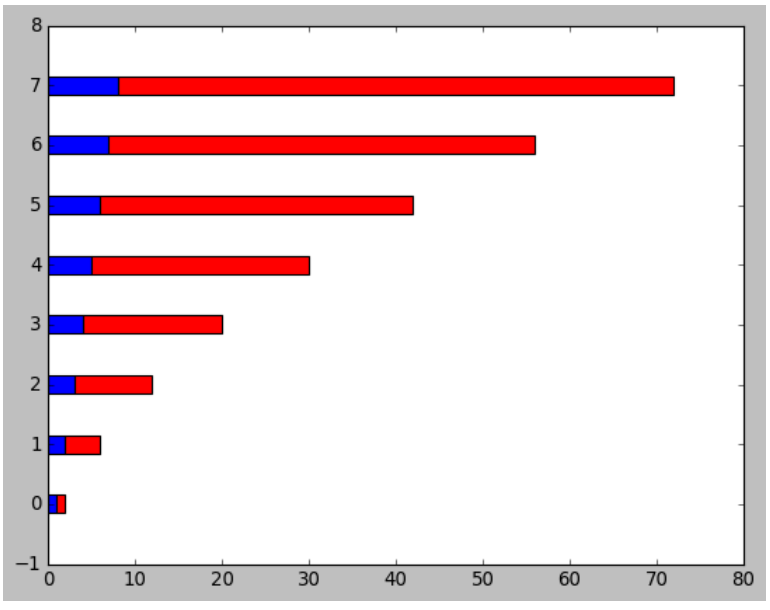
Finalmente, podemos girar este gráfico de barras verticales en un gráfico de barras horizontal llamando a la función barh, pero teniendo en cuenta que tenemos que cambiar la parte inferior a la izquierda y la altura sobre el ancho:

In [48]:

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

datos_lins = np.array([1, 2, 3, 4, 5, 6, 7, 8])
datos_cuads = datos_lins**2
plt.ion()
plt.figure()
xvals = range(len(datos_lins))
plt.bar(xvals, datos_lins, width = 0.3)

new_xvals = []
for item in xvals:
    new_xvals.append(item+0.3)

plt.bar(new_xvals, datos_cuads, width = 0.3 ,color='red')
plt.figure()
xvals = range(len(datos_lins))
plt.barh(xvals, datos_lins, height = 0.3, color='b')
plt.barh(xvals, datos_cuads, height = 0.3, left=datos_lins, color='r')
```

Out[48]: <BarContainer object of 8 artists>



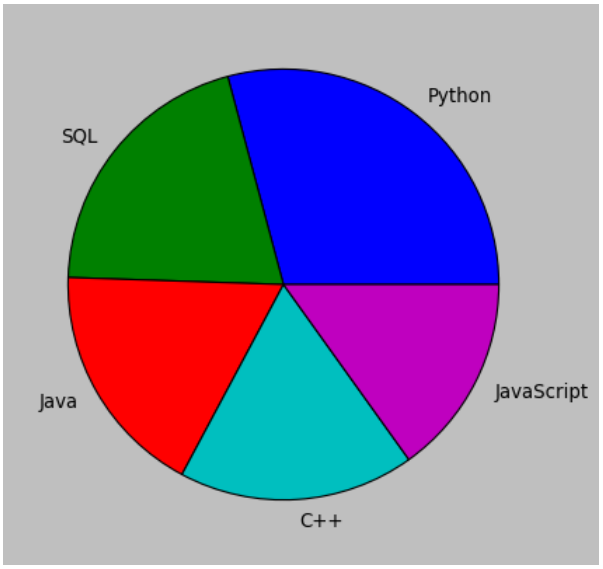


Gráficos circulares

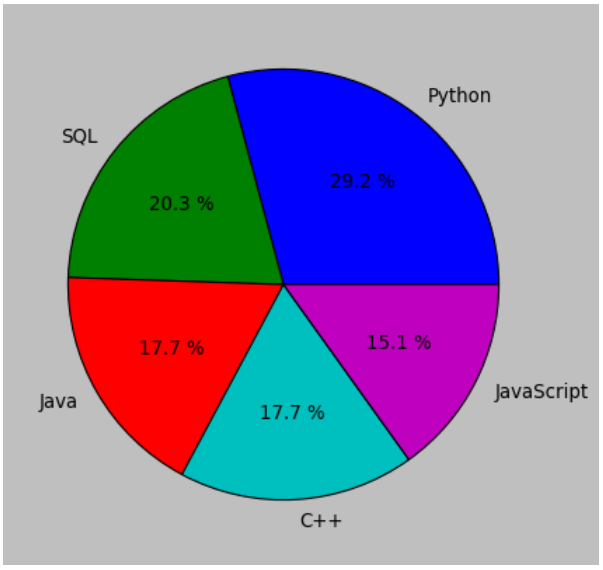
El gráfico circular, también llamado gráfico de torta o, en inglés, **pie chart**, es adecuado para mostrar proporciones en un conjunto, y es especialmente aconsejable cuando el número de sectores (de valores a mostrar) no es demasiado elevado pues, de otra forma, los sectores del gráfico resultan más difíciles de apreciar.

La función que nos permite mostrar un gráfico circular es **matplotlib.pyplot.pie**, existiendo también un método del conjunto de ejes, **matplotlib.axes.Axes.pie** con la misma funcionalidad.

In [49]:
```
import matplotlib.pyplot as plt

popularity = [56, 39, 34, 34, 29]
languages =['Python', 'SQL', 'Java', 'C++', 'JavaScript']
plt.pie(popularity, labels=languages)
plt.show()
```

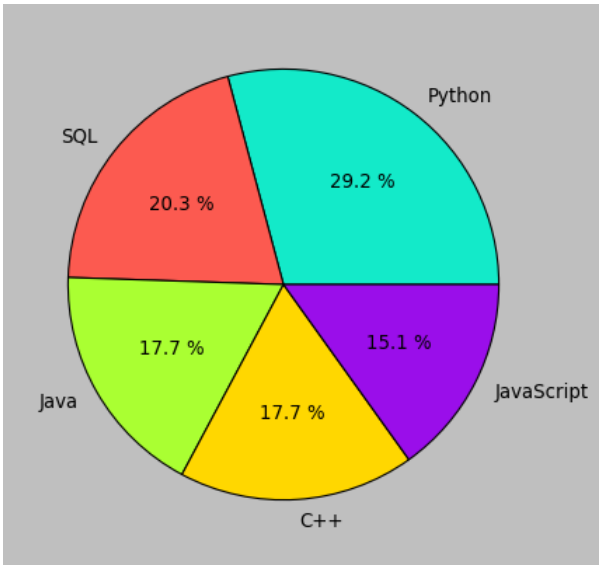


In [50]:
```
import matplotlib.pyplot as plt

popularity = [56, 39, 34, 34, 29]
languages =['Python', 'SQL', 'Java', 'C++', 'JavaScript']
plt.pie(popularity, labels=languages, autopct="%0.1f %%")
plt.show()
```



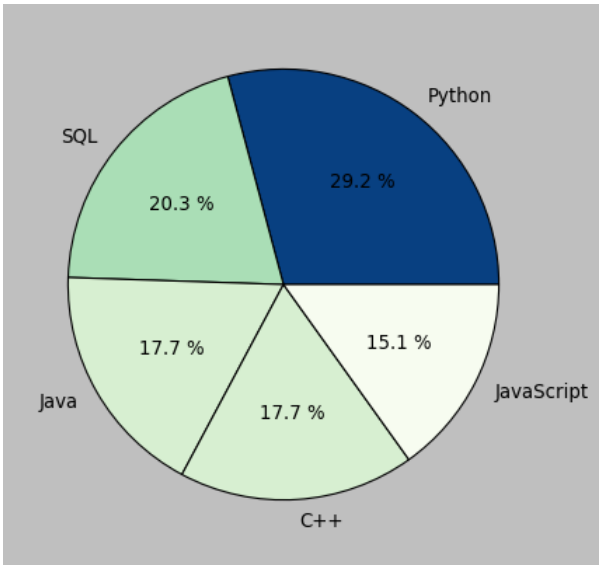
In [51]:

```
import matplotlib.pyplot as plt

popularity = [56, 39, 34, 34, 29]
languages =['Python', 'SQL', 'Java', 'C++', 'JavaScript']
colores = ["#13EAC9","#FC5A50","#AAFF32","#FFD700","#9A0EEA"]
plt.pie(popularity, labels=languages, autopct="%0.1f %%", colors=colores)
plt.show()
```



In [52]:

```
import matplotlib.pyplot as plt
from matplotlib import cm
from matplotlib import colors

popularity = [56, 39, 34, 34, 29]
languages =['Python', 'SQL', 'Java', 'C++', 'JavaScript']
normdata = colors.Normalize(min(popularity), max(popularity))
colormap = cm.get_cmap("GnBu")
colores =colormap(normdata(popularity))
plt.pie(popularity, labels=languages, autopct="%0.1f %%", colors=colores)
plt.show()
```



In [53]:

```
import matplotlib.pyplot as plt

popularity = [56, 39, 34, 34, 29]
languages =['Python', 'SQL', 'Java', 'C++', 'JavaScript']
colores = ["#13EAC9","#FC5A50","#AAFF32","#FFD700","#9A0EEA"]
desfase = (0, 0, 0.5, 0, 0.1)
plt.pie(popularity, labels=languages, autopct="%0.1f %%", colors=colores, explode=desfase)
plt.show()
```

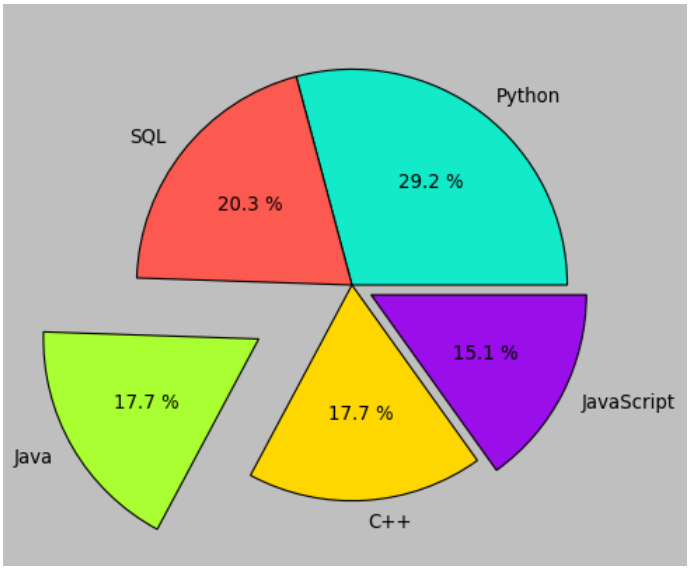


Diagrama de caja

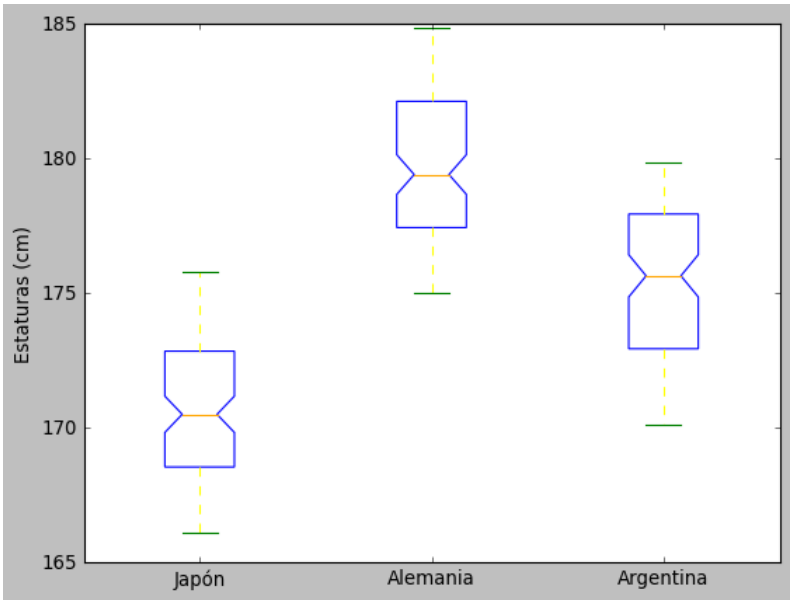
También se conoce como diagrama de bigotes y se crea para mostrar el resumen del conjunto de valores de datos que tienen propiedades como mínimo, primer cuartil, mediana, tercer cuartil y máximo.

En el diagrama de caja, se crea una caja desde el primer cuartil hasta el tercer cuartil, también hay una línea vertical que pasa por la caja en la mediana.

Aquí el eje x denota los datos que se trazarán, mientras que el eje y muestra la distribución de frecuencia.

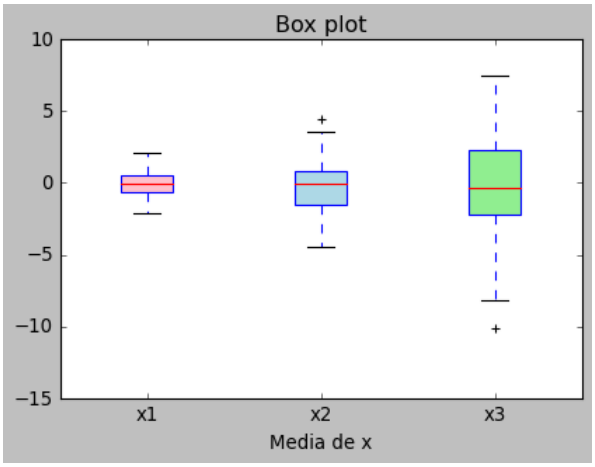
In [54]:

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

jap = np.random.uniform(166, 176, 100)
ale = np.random.uniform(175, 185, 100)
arg = np.random.uniform(170, 180, 100)

plt.boxplot([jap, ale, arg],
notch=True, patch_artist=True,
    capprops=dict(color="green"),
    medianprops=dict(color="orange"),
    whiskerprops=dict(color="yellow"))
plt.xticks([1, 2, 3], ['Japón', 'Alemania', 'Argentina'])
plt.ylabel('Estaturas (cm)')
plt.show()
```



In [55]:

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

all_data = [np.random.normal(0, std, 100) for std in range(1, 4)]
fig = plt.figure(figsize=(6,4))
bplot = plt.boxplot(all_data,
    notch=False,
    vert=True,
    patch_artist=True)
colors = ['pink', 'lightblue', 'lightgreen']

for patch, color in zip(bplot['boxes'], colors):
    patch.set_facecolor(color)

plt.xticks([y+1 for y in range(len(all_data))], ['x1', 'x2', 'x3'])
plt.xlabel('Media de x')
t = plt.title('Box plot')
plt.show()
```

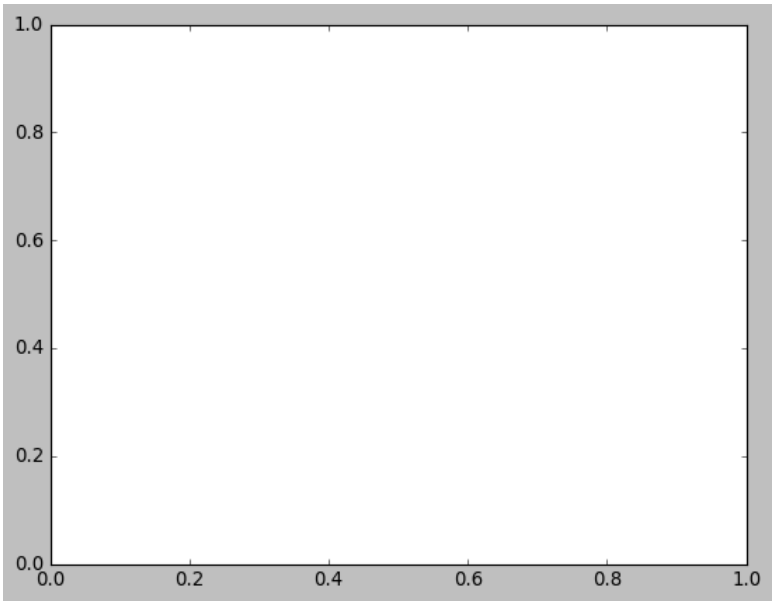


Subplots

La función **matplotlib.pyplot.subplots** crea una figura y uno (o varios) conjunto de ejes, devolviendo una referencia a la figura y a los ejes. Por defecto -si no se especifica otra cosa- crea un único conjunto de ejes.

In [56]:
```
import matplotlib.pyplot as plt
import numpy as np

y = np.random.randn(100).cumsum()
fig,ax=plt.subplots()
type(ax)
```

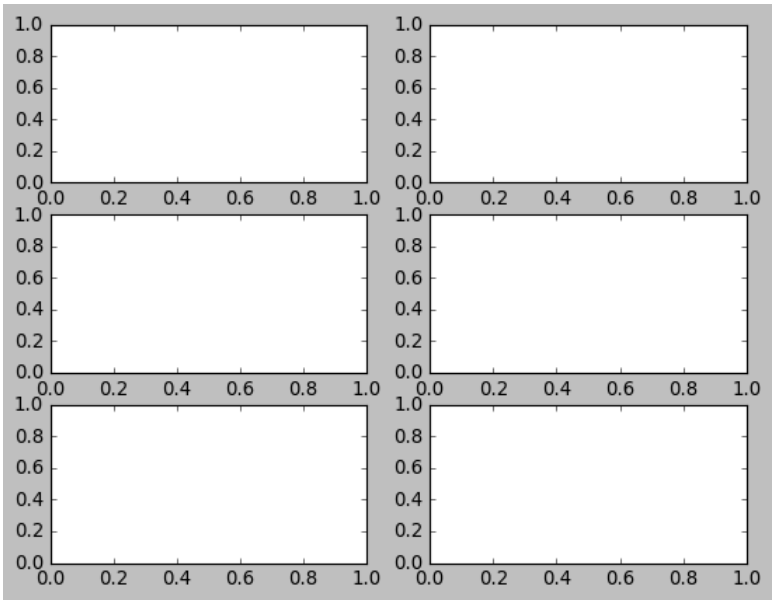
Out[56]:
```
matplotlib.axes._subplots.AxesSubplot
```



Si queremos crear una matriz de conjuntos de ejes de, por ejemplo, 3 filas y 2 columnas (es decir, 6 conjuntos de ejes repartidos de dicha forma), basta agregar estos valores como primeros argumentos de la función:

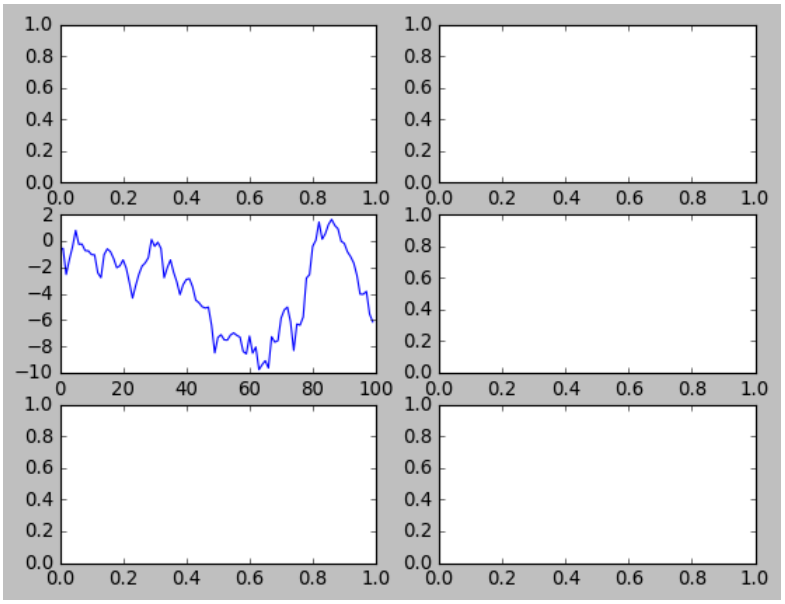
In [57]:
```
fig,ax=plt.subplots(3,2)
fig.set_size_inches(8,6)
type(ax)
ax.shape
ax
```

Out[57]:
```
array([[<AxesSubplot:>, <AxesSubplot:>],
       [<AxesSubplot:>, <AxesSubplot:>],
       [<AxesSubplot:>, <AxesSubplot:>]], dtype=object)
```



Ahora podríamos ejecutar el método plot asociado a cada uno de estos ejes para mostrar una gráfica. Por ejemplo, si quisiéramos mostrarla en la segunda fila (cuyo índice es 1) y primera columna (cuyo índice es 0), podríamos hacerlo del siguiente modo:

In [58]:
```
fig,ax=plt.subplots(3,2)
fig.set_size_inches(8,6)
ax[1,0].plot(y)
plt.show()
```

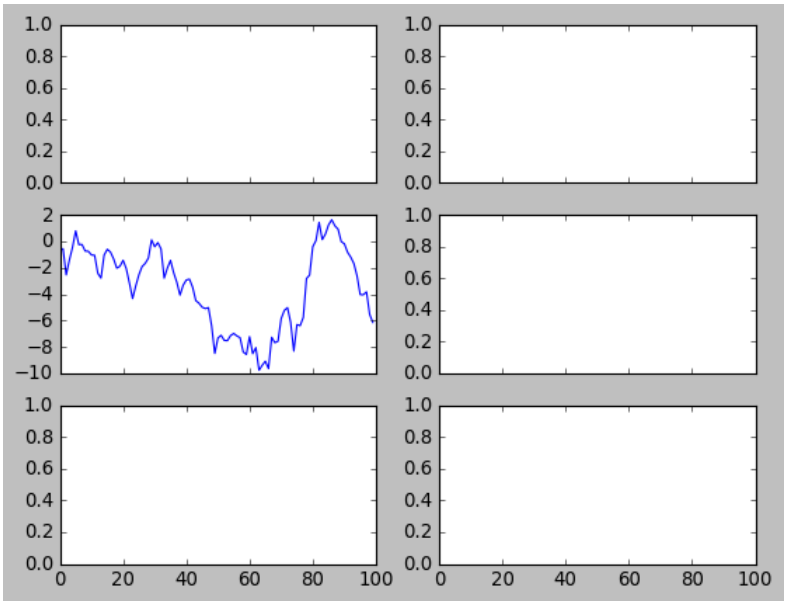


Los parámetros de la función subplots **sharex y sharey** controlan las propiedades de los ejes compartidas.

Por defecto toman el valor False, lo que supone que cada conjunto de ejes es independiente.

Si, por ejemplo, el argumento sharex se fija a True, todos los ejes x de los diferentes conjuntos de ejes compartirán las mismas propiedades.

In [59]:
```
fig,ax=plt.subplots(3,2, sharex = True)
fig.set_size_inches(8,6)
ax[1,0].plot(y)
plt.show()
```



https://pbpython.com/effective-matplotlib.html (https://pbpython.com/effective-matplotlib.html)