GRÁFICOS CON MATPLOTLIB

Instalación de matplotlib

En la ventana de comandos ejecutar: **pip install matplotlib**

Interfaces

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:

```
>>> import matplotlib.pyplot as plt
>>> plt.get_backend()
'TkAgg'
>>>
```

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

```
%matplotlib notebook
import matplotlib.pyplot as plt
plt.get_backend()
'nbAgg'
```

Jupyter Notebook, tiene soporte para matplotlib que se habilita mediante el uso de IPython [Jupyter and the future of IPython — IPython]. 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.

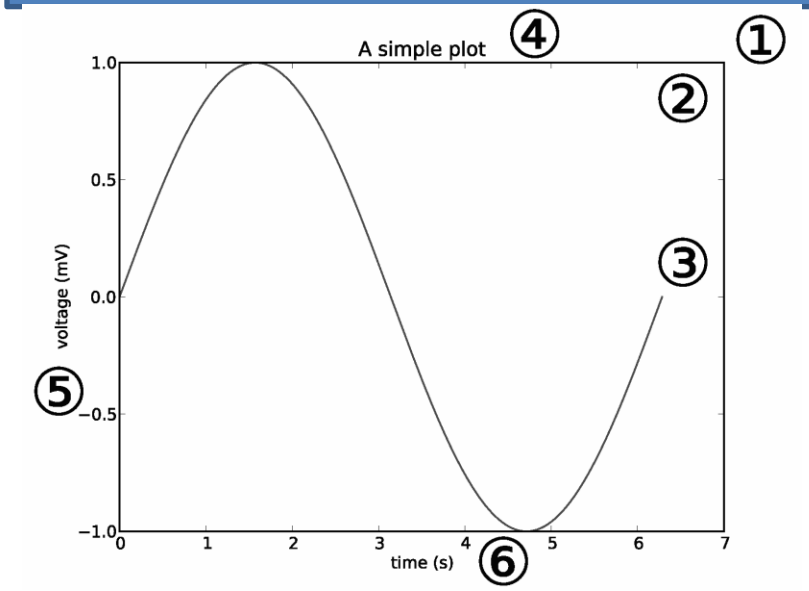
Arquitectura de Matplotlib

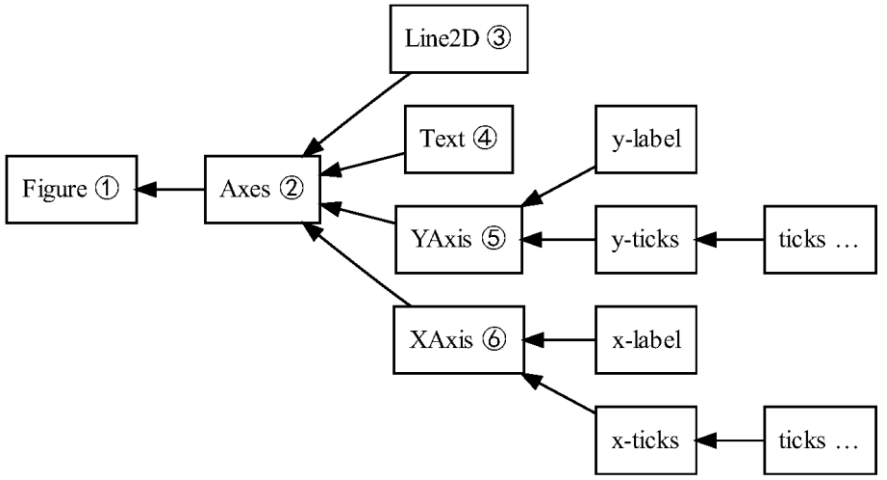
Capa de artista (Artist layer): es el objeto que sabe tomar el Renderer (el pincel) de la capa Back-end para dibujar el gráfico en FigureCanvas, es decir dibujar en el lienzo. Todo lo que se ve en Figure es una instancia de Artist, el título, las líneas, las tick, las imágenes, etc., y corresponden a 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: [matplotlib.artist — Matplotlib 3.4.0 docu mentation]





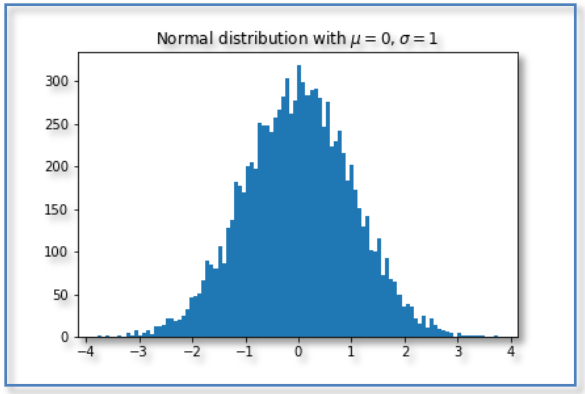
Fuente: **matplotlib,** John Hunter and Michael Droettboom

Por ejemplo, la tabla a continuación brinda una pequeña muestra de métodos Axes que crean objetos de trazado y los almacenan en la instancia Axes.

Axes.imshow	Uno o más: matplotlib.image.AxesImage	Axes.images
Axes.hist	Muchos: matplotlib.patch.Rectangle	Axes.patches
Axes.plot	Uno o más: matplotlib.lines.Line2D	Axes.lines

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.

```
# 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 (bins).
ax.hist(x, 100)

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

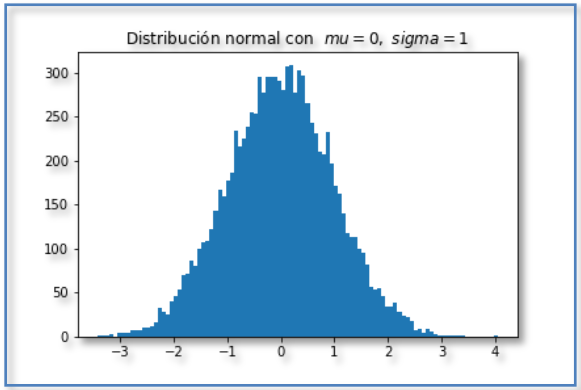


Arquitectura de Matplotlib

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, 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:

```
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 ('matplotlib_histogram2.png')
plt.show ()
```



Partes de una figura

Ejecutar: anatomy.ipynb o anatomy.py

Figure: La figura es el contenedor principal de nuestras graficas. 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.

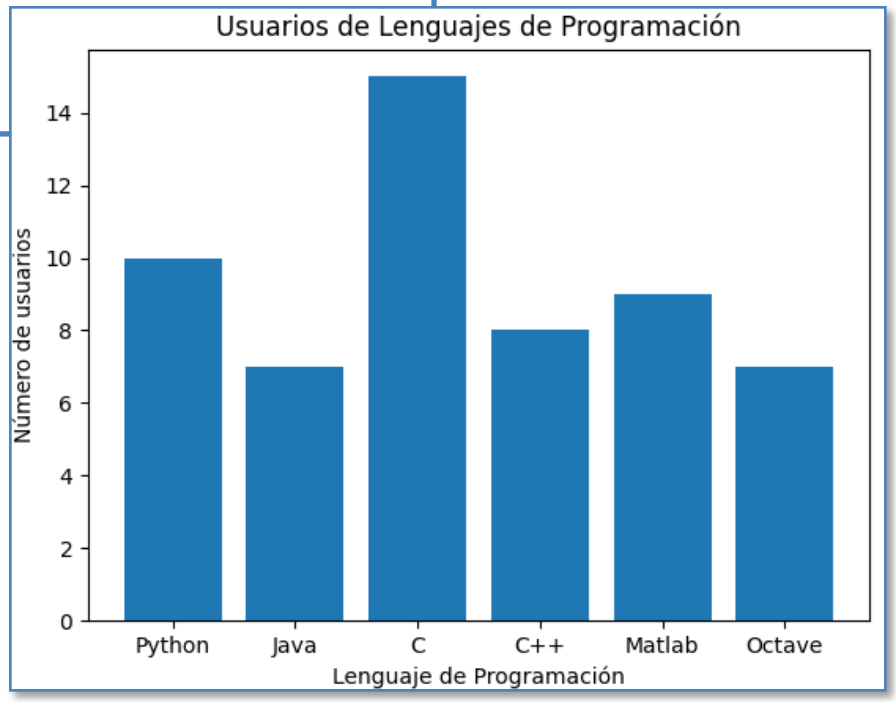
Axes: Es el contenedor de la grafica. Una figura puede contener varios axes, y un axes puede contener varias graficas. El parámetro de axes tiene como miembros los títulos, los limites de los ejes, los incrementos de los ejes, entre otros valores. Aquí se grafican los datos con funciones tales como plot() 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.

Ejemplo que muestra la funcionalidad de la capa Artist de Matplotlib

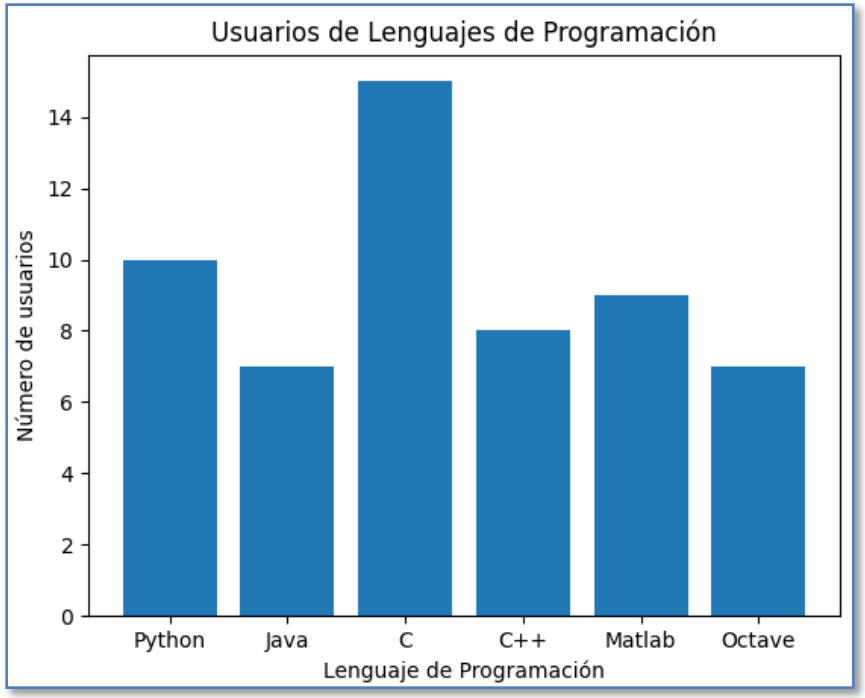
```
# Importar la biblioteca matplotlib
import matplotlib.pyplot as plt

# 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 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')
```



https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.subplot.html

Ejemplo que muestra la funcionalidad de la capa Scripting de Matplotlib

```
# 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')
```



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.

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:

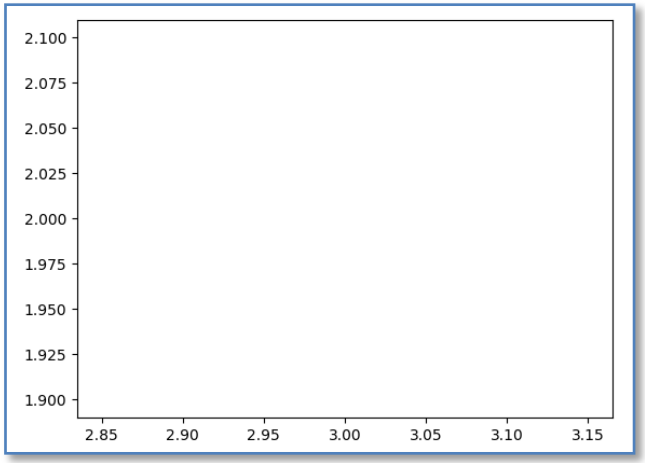
- Si el modo interactivo está activado, no es necesario **plt.show(),** y las imágenes se visualizarán y se actualizarán a medida que se haga referencia.
- Si el modo interactivo está desactivado, **plt.show()** deberá mostrar una figura y **plt.draw()** actualizar un gráfico.

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** , la función admite cualquier número de argumentos sin nombre. La palabra clave ****kwargs** también 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.

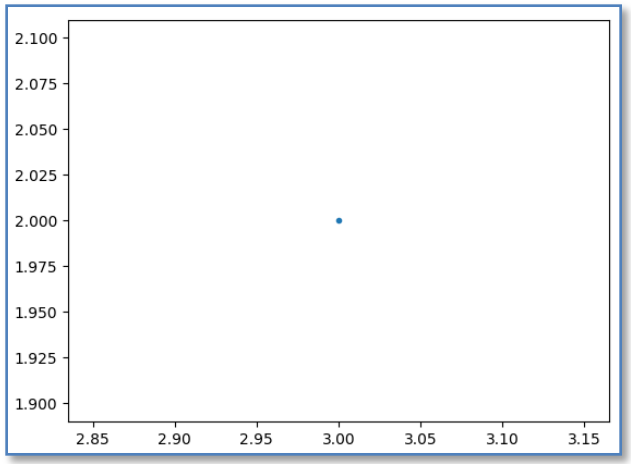
```
import matplotlib.pyplot as plt
plt.plot(3,2)
```



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.

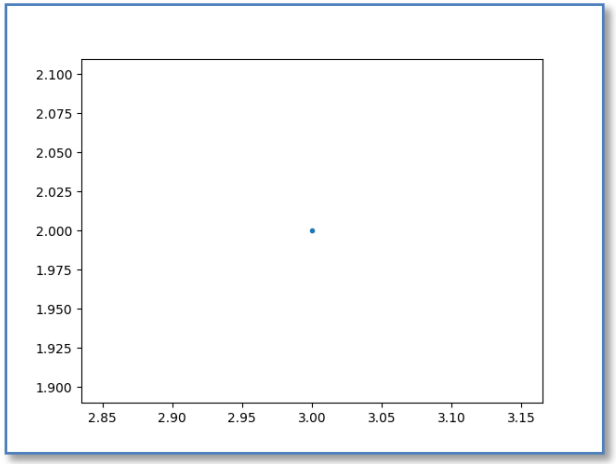
```
import matplotlib.pyplot as plt
plt.plot(3,2, '.')
```



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

Función gráfica

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:



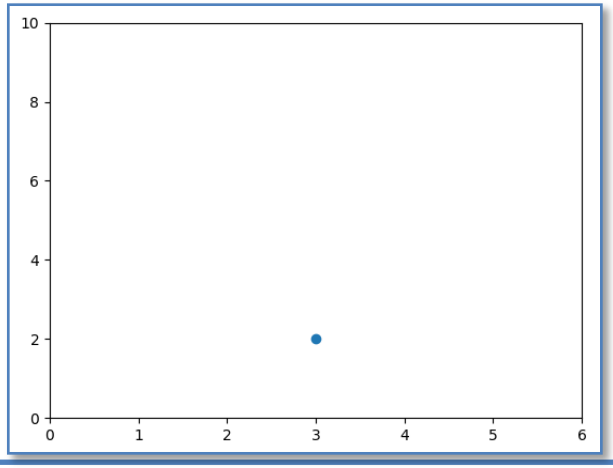

```
from matplotlib.backends.backend_agg import FigureCanvasAgg
from matplotlib.figure import Figure

fig = Figure()
canvas = FigureCanvasAgg(fig)

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

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

```
%%html
<img src='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 obtener acceso a los ejes actuales usando la función **gca():**

```
import matplotlib.pyplot as plt

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



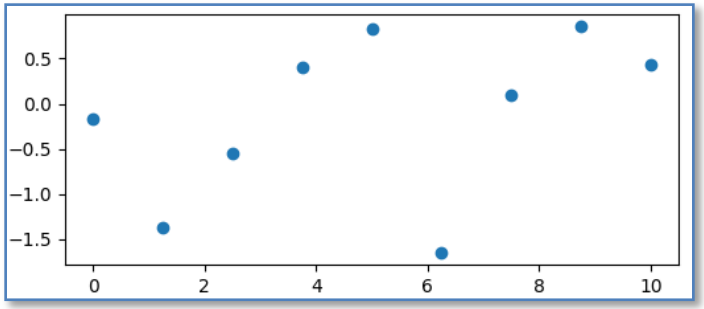
axis tiene 4 parámetros: un valor mínimo para x, al cual le dimos 0, un valor máximo para x, que le dimos 6. Luego, los valores mínimos y máximos para y, que le dimos 0 y 10.

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()

```
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)
plt.show()
```



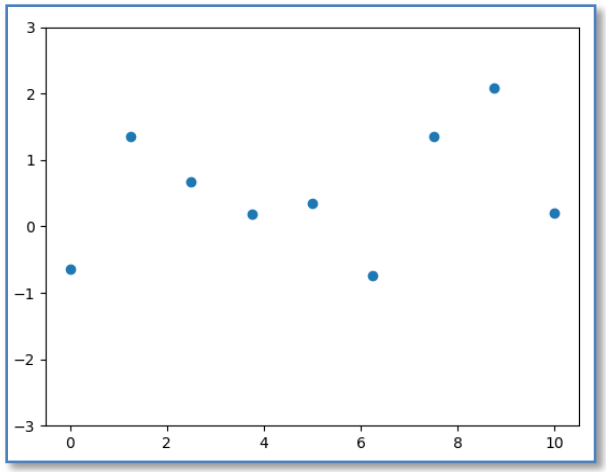

```
>>> print(x)
[ 0.    1.25  2.5   3.75  5.    6.25  7.5   8.75 10.  ]
>>> print(y)
[-0.17203731 -1.37035975 -0.54307496  0.40487676  0.83304518 -1.65481137
  0.09171857  0.85814374  0.4388983 ]
```

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, por ejemplo.

```
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)
plt.show()
```

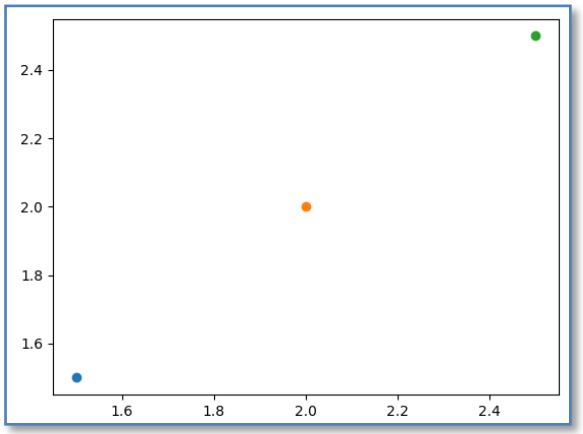


Tener en cuenta que el eje y, ahora varía de -3 a 3. Obtuvimos una referencia al eje actual llamando a plt.gca (). Luego llamamos al método set_ylim () para ajustar los límites para ese eje.

Función gráfica

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:

```
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')
```



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

```
ax = plt.gca()
ax.get_children()
```

Eso produce la siguiente salida:

```
[<matplotlib.lines.Line2D object at 0x000001DCE3E8EF70>,
<matplotlib.lines.Line2D object at 0x000001DCE3E9C2B0>,
<matplotlib.lines.Line2D object at 0x000001DCE3E9C640>,  # estos son nuestros puntos de datos
<matplotlib.spines.Spine object at 0x000001DCE53C17C0>,
<matplotlib.spines.Spine object at 0x000001DCE53C18E0>,
<matplotlib.spines.Spine object at  0x000001DCE53C1A00>,
<matplotlib.spines.Spine object at 0x000001DCE53C1B20>, # renderizaciones reales de los bordes del marco incluyendo marcadores
<matplotlib.axis.XAxis object at 0x000001DCE53C1760>,
<matplotlib.axis.YAxis object at 0x000001DCE53C7070>, # dos objetos de eje
Text(0.5, 1.0, ''), Text(0.0, 1.0, ''), Text(1.0, 1.0, ''), # etiquetas del gráfico.
<matplotlib.patches.Rectangle object at 0x000001DCE3E787F0>]# fondo de los ejes.
```

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.

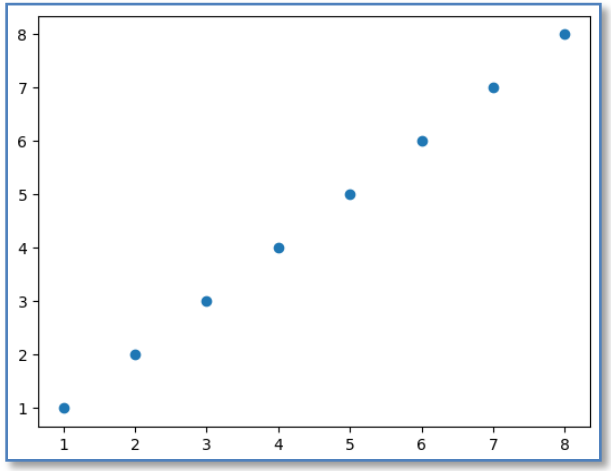
•La declaración de la mayoría de las funciones en matplotlib termina con un conjunto abierto de argumentos de palabras clave. Existen propiedades diferentes que puede controlarse a través de estos argumentos.

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:

```
import matplotlib.pyplot as plt

x = [1,2,3,4,5,6,7,8]
y = x

plt.figure()
plt.scatter(x, y)
```

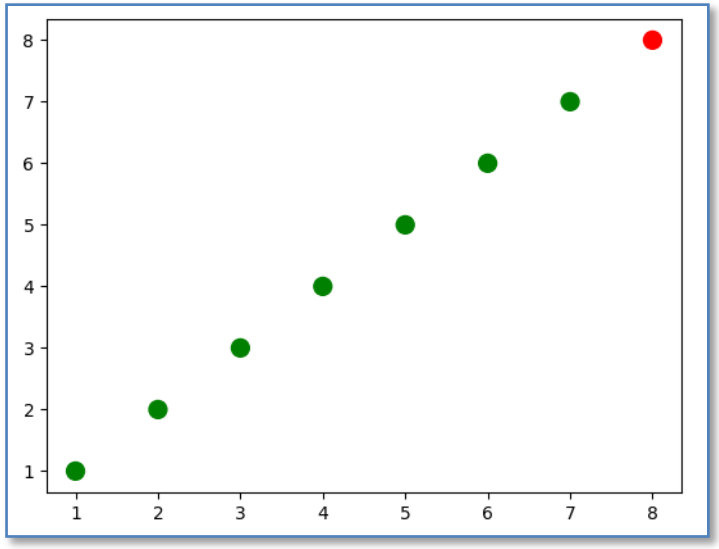


En esta trama puede verse una línea diagonal y matplotlib ha dimensionado nuestros ejes en consecuencia. Pero la dispersión no representa elementos como una serie.

Scatterplots

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:

```
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)
# s es  el tamaño de los puntos
```



Si queremos observar como quedó la lista de colores con la operación:

```
>>> colors
['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:

```
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.

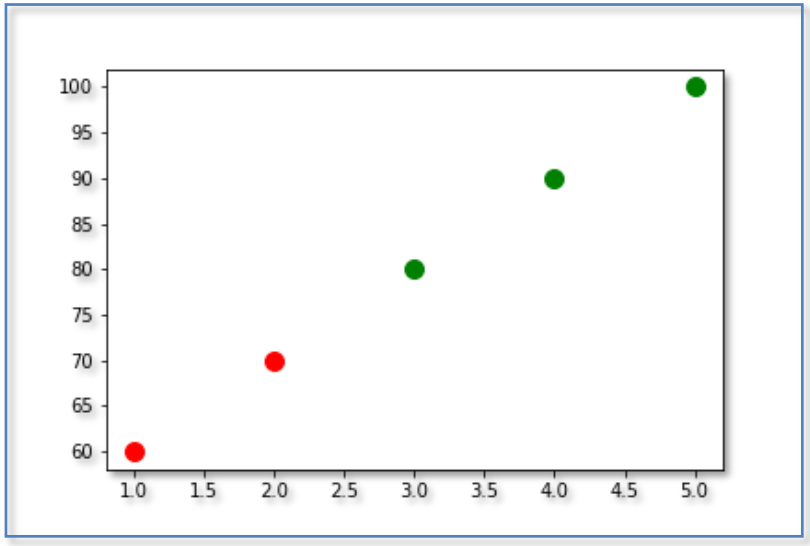
Si queremos volver a convertir los datos en 2 listas, una con el componente x y otra con el componente y, podemos utilizar el parámetro de desempaquetado con zip. **Cuando se pasa una lista o un intervalo, generalmente a una función y se antepone con un *, cada elemento se saca del iterable y se pasa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

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

Scatterplots

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:

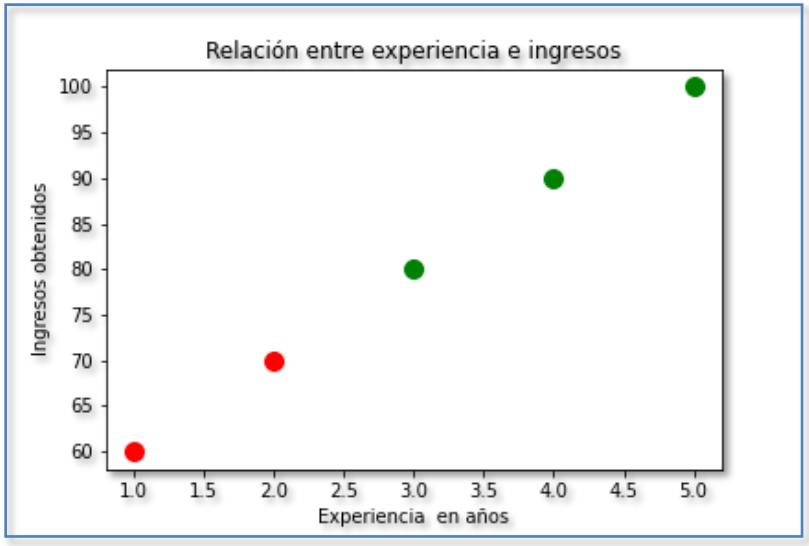
```
import matplotlib.pyplot as plt

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

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))
```



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.

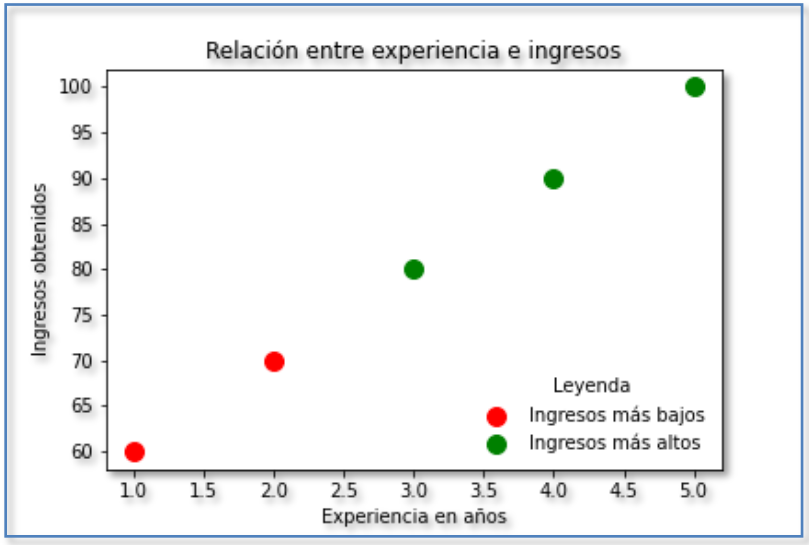
Scatterplots

```
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()
```



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:

```
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‘)
```



Scatterplots

Como la leyenda es un Artist para ver los hijos podemos acceder mediante el código que vimos anteriormente: **plt.gca().get_children():**

[<matplotlib.collections.PathCollection object at 0x0000022C01703A60>, <matplotlib.collections.PathCollection object at 0x0000022C01713AF0>, <matplotlib.spines.Spine object at 0x0000022C016D1AC0>, <matplotlib.spines.Spine object at 0x0000022C016D1BB0>, <matplotlib.spines.Spine object at 0x0000022C016D1CA0>, <matplotlib.spines.Spine object at 0x0000022C016D1D90>, <matplotlib.axis.XAxis object at 0x0000022C016D1A30>, <matplotlib.axis.YAxis object at 0x0000022C016DF160>, Text(0.5, 1.0, 'Relaciónentreexperienciae ingresos'), Text(0.0, 1.0, ''), Text(1.0, 1.0, ''), <matplotlib.legend.Legend object at 0x0000022C016F6A90>, <matplotlib.patches.Rectangle object at 0x0000022C016EB7F0>]

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:

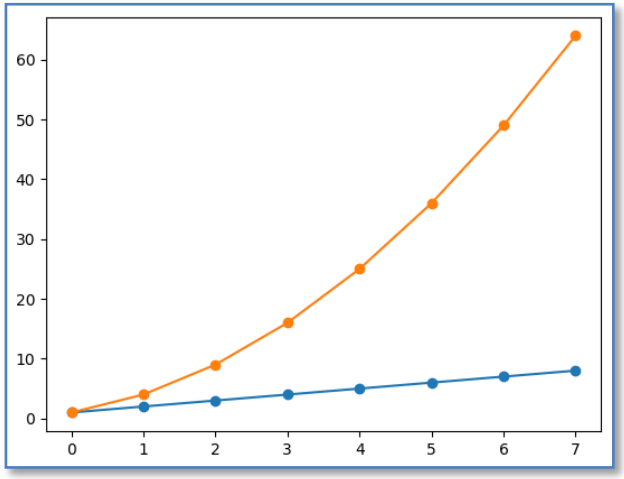
```
leyenda=plt.gca().get_children()[-2]
print(leyenda)

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)
```

```
Legend
 <matplotlib.offsetbox.VPacker object at 0x0000022C0174AA60>
  <matplotlib.offsetbox.TextArea object at 0x0000022C0174A8E0>
   Text(0, 0, 'Leyenda')
  <matplotlib.offsetbox.HPacker object at 0x0000022C0174A8B0>
   <matplotlib.offsetbox.VPacker object at 0x0000022C0174A7F0>
    <matplotlib.offsetbox.HPacker object at 0x0000022C0174A820>
     <matplotlib.offsetbox.DrawingArea object at 0x0000022C0174A190>
     <matplotlib.offsetbox.TextArea object at 0x0000022C0174A160>
      Text(0, 0, 'Ingresos más bajos')
    <matplotlib.offsetbox.HPacker object at 0x0000022C0174A790>
     <matplotlib.offsetbox.DrawingArea object at 0x0000022C0174A4C0>
      <matplotlib.collections.PathCollection object at 0x0000022C0174A760>
     <matplotlib.offsetbox.TextArea object at 0x0000022C0174A490>
      Text(0, 0, 'Ingresos más altos')
 FancyBboxPatch((0, 0), width=1, height=1)
```

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 back-end renderiza en la pantalla o en un archivo.

Line plots

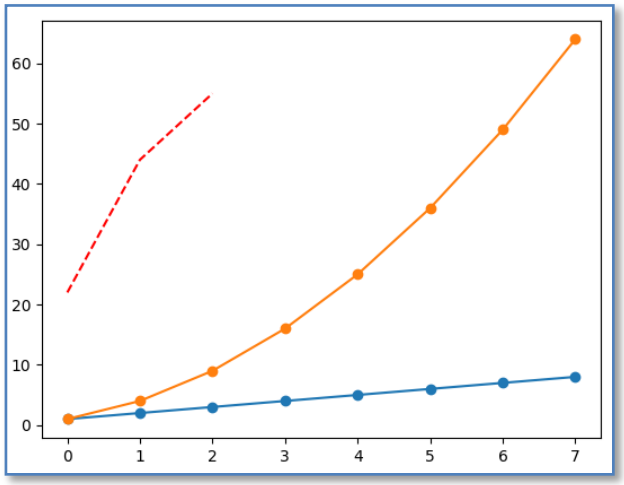
```
import matplotlib.pyplot as plt
import numpy as np

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')
```



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:

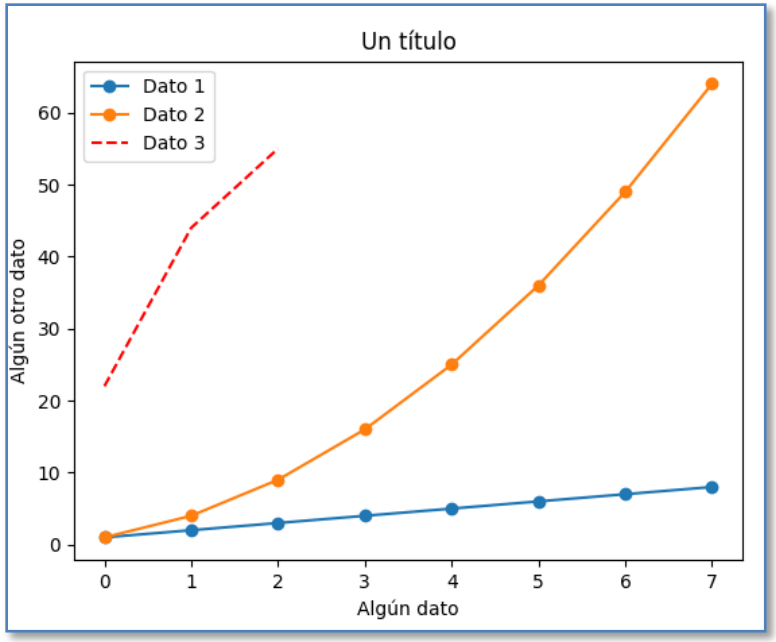
```
import matplotlib.pyplot as plt
import numpy as np

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')
```



Line plots

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:

```
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'])
```

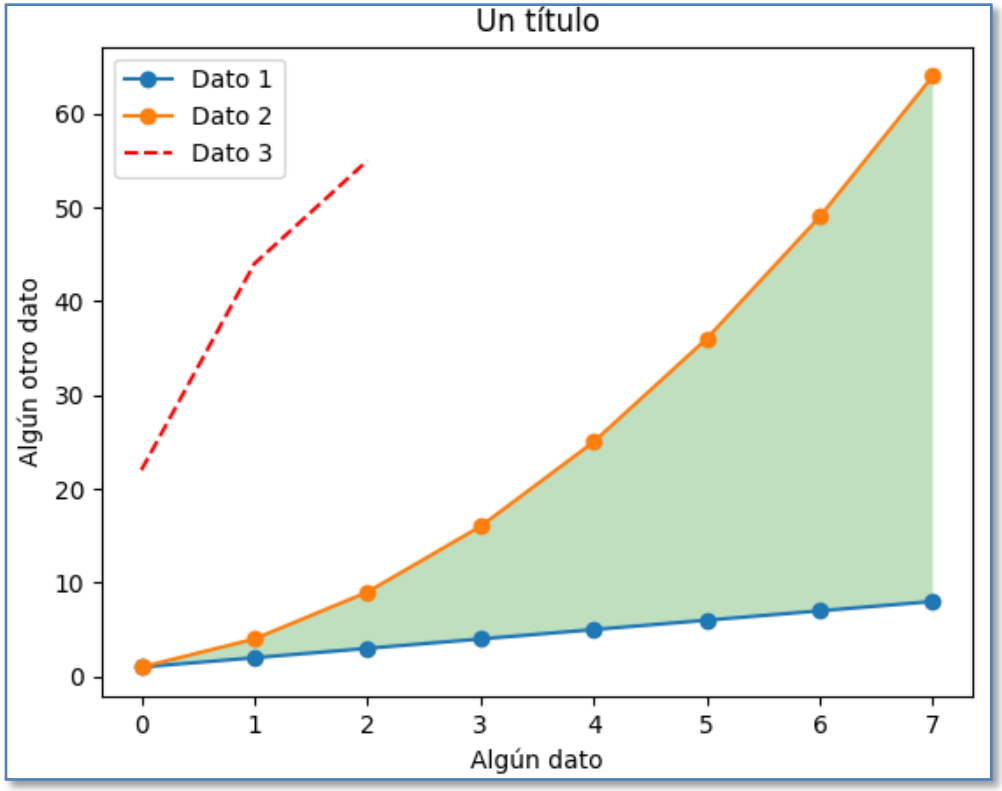


Función de relleno

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.

Line plots

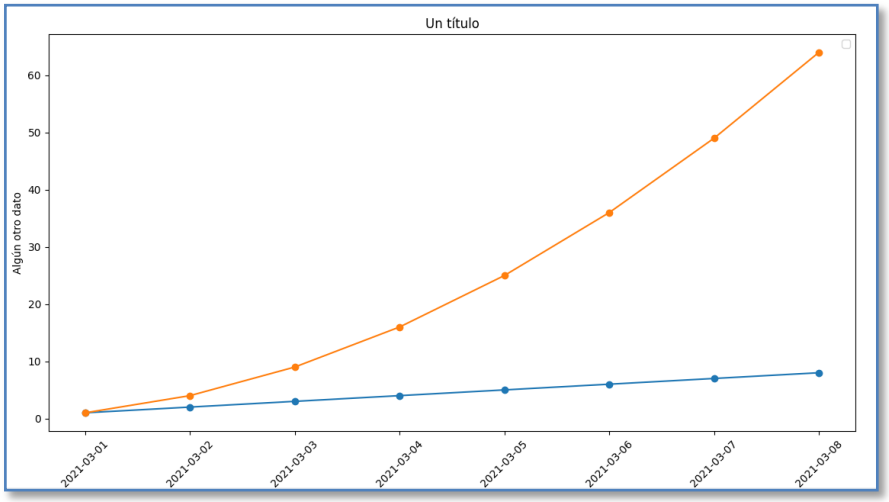
```
import matplotlib.pyplot as plt
import numpy as np

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)
```



Line plots

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:

```
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'])
```

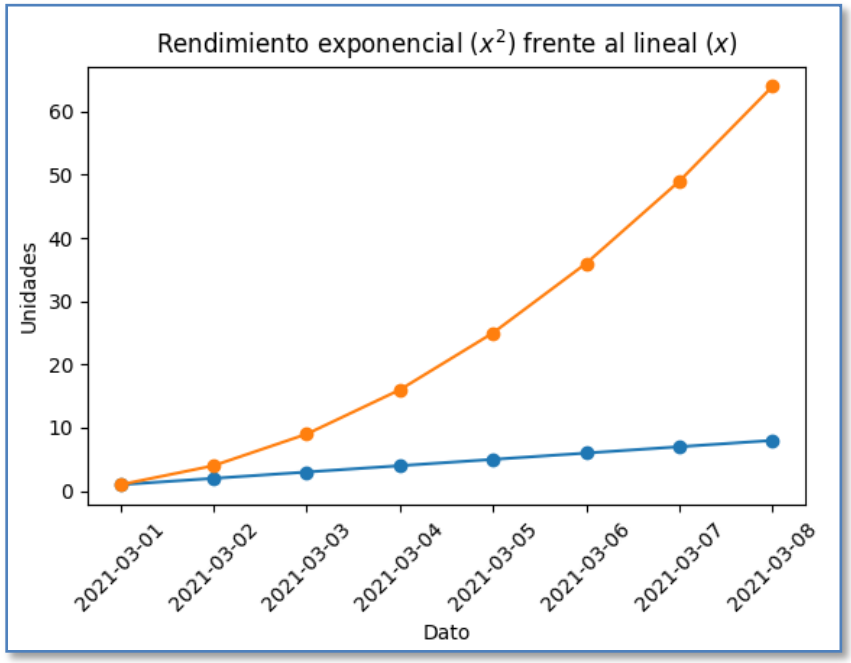



```
datos_observados = np.arange('2021-03-01','2021-03-09',dtype='datetime64[D]')
datos_observados = list(map(pd.to_datetime, datos_observados))

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.

Line plots

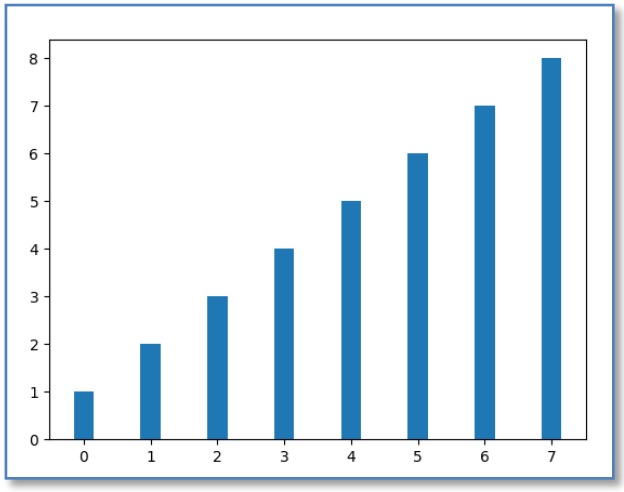
```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

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 ($x$)')
```



Bar charts

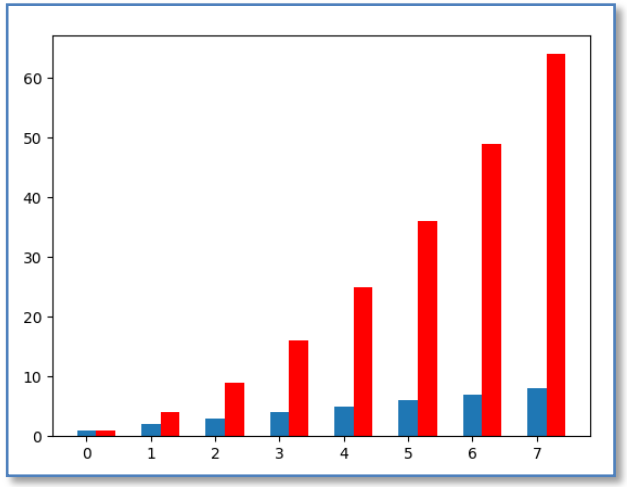
Matplotlib tiene soporte para varios tipos de gráficos de barras. El caso más general, trazamos un gráfico de barras enviando un parámetro de los componentes x, y un parámetro de la altura de la barra:

```
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)
```



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:

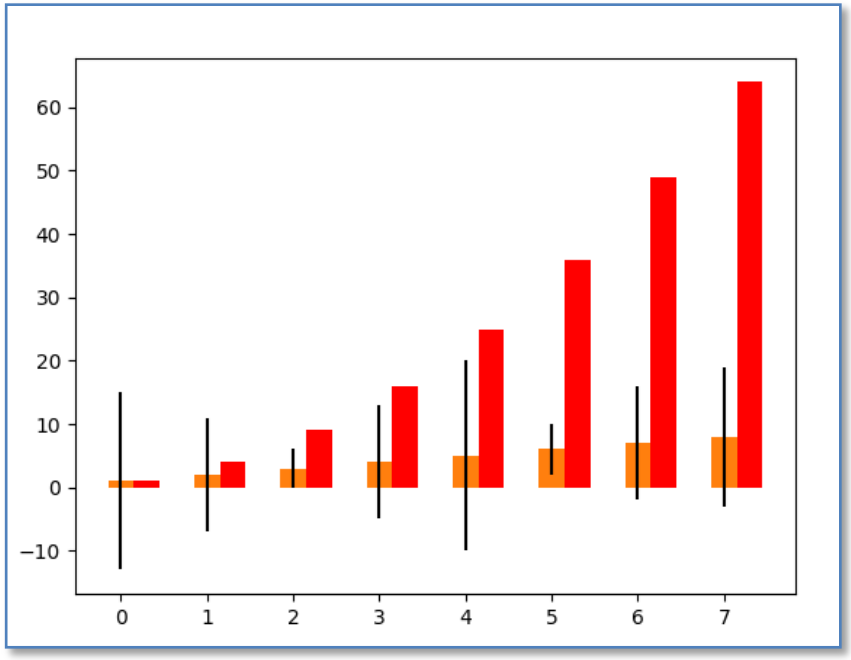
```
import matplotlib.pyplot as plt
import numpy as np

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‘)
```



Bar charts

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:

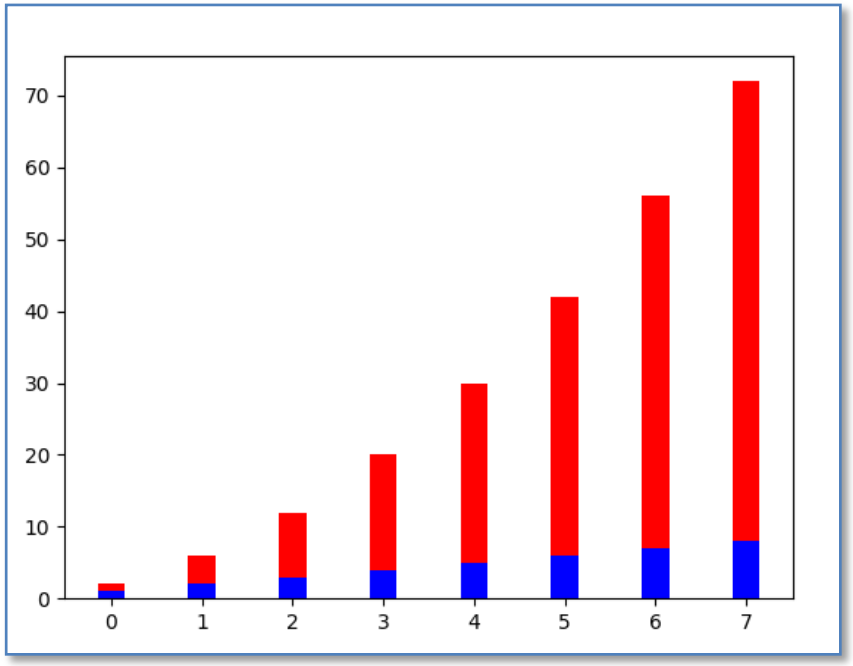
```
import matplotlib.pyplot as plt
import numpy as np

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)
```



Bar charts

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:

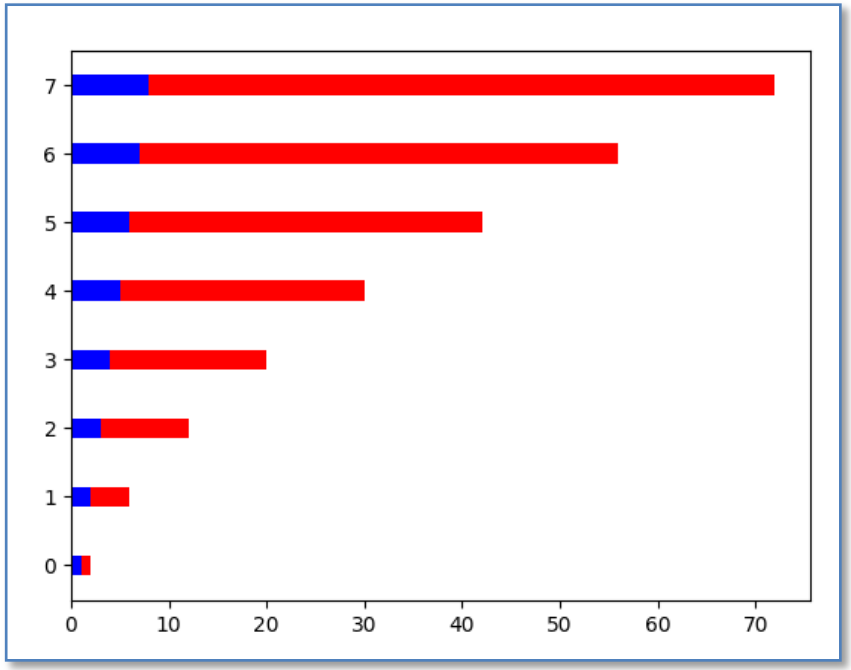
```
import matplotlib.pyplot as plt
import numpy as np

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')
```



Bar charts

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:

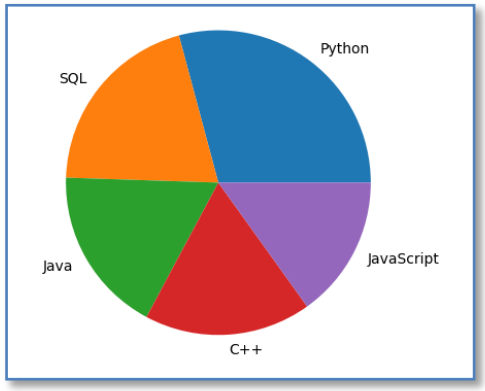
```
import matplotlib.pyplot as plt
import numpy as np

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')
```



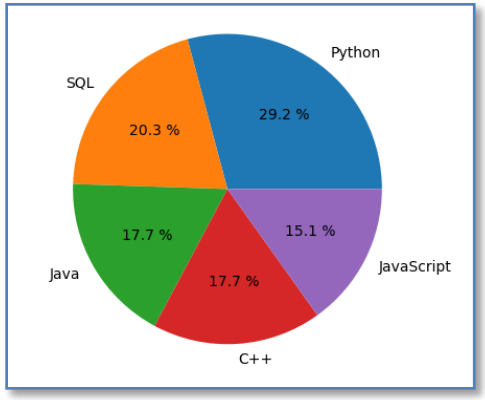
Gráficos circulares

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.

```
import matplotlib.pyplot as plt

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

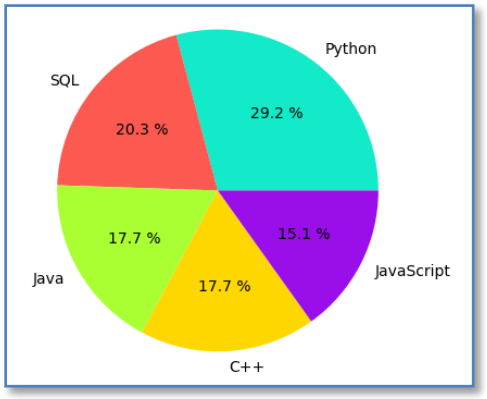



```
import matplotlib.pyplot as plt

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

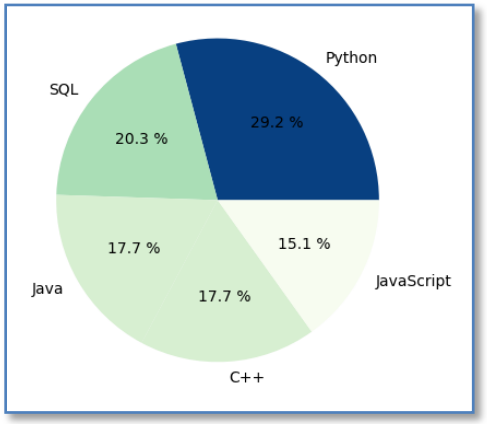


Gráficos circulares

```
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()
```




```
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()
```




```
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, 0, 0.1)
plt.pie(popularity, labels=languages, autopct="%0.1f %%", colors=colores, explode=d
esfase)
plt.show()
```

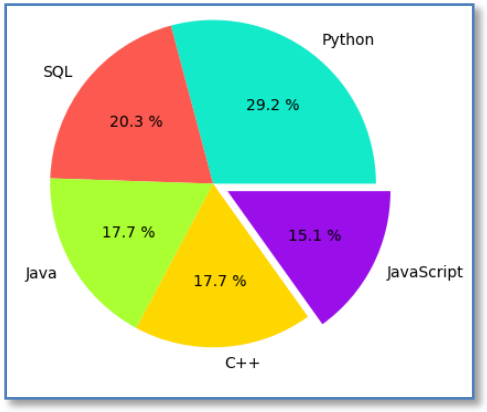
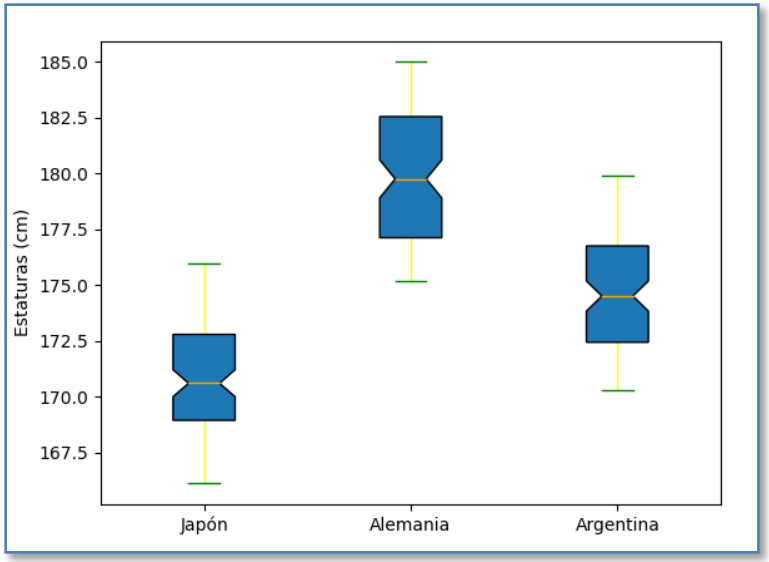


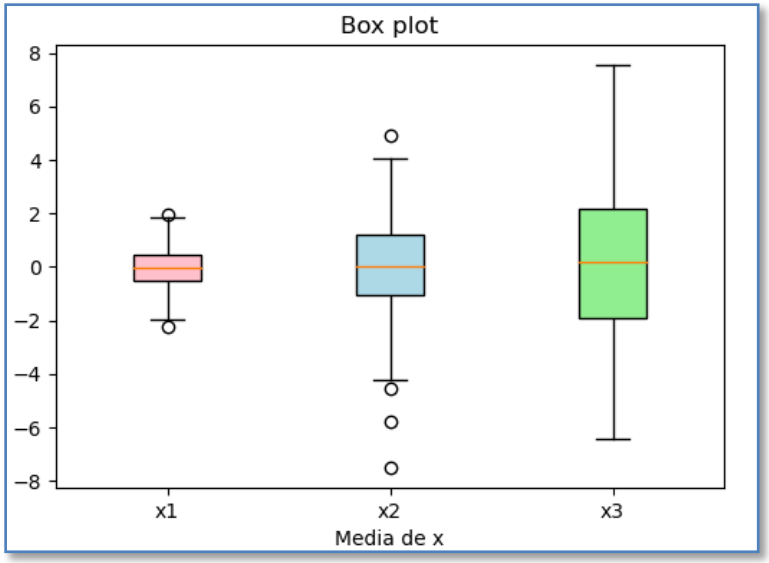
Diagrama de caja

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.

```
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()
```



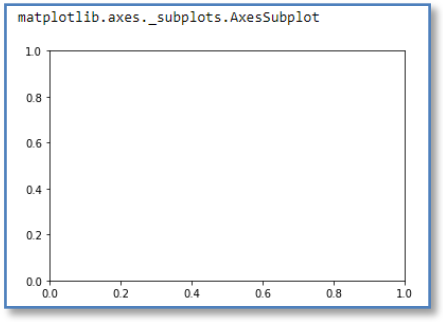

```
import matplotlib.pyplot as plt
import numpy as np

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

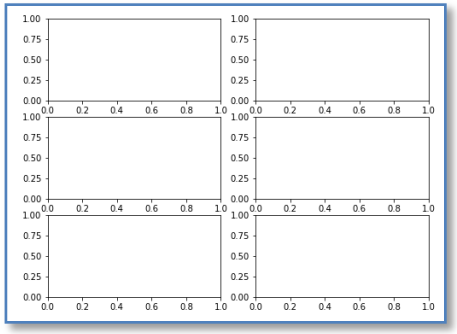
La función matplotlib.pyplot.subplots crea una figura y uno (o varios) conjunto de ejes, devolviendo una referencia a la figura y a los ejes. Por defecto -si no se especifica otra cosa- crea un único conjunto de ejes.

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

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

type(ax)
```



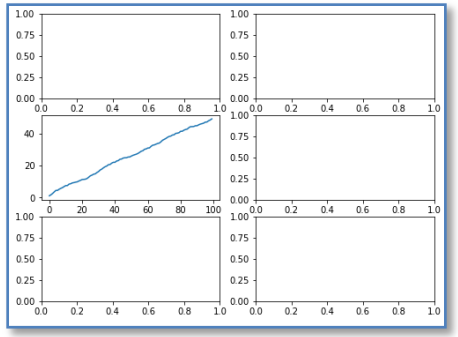
Si queremos crear una matriz de conjuntos de ejes de, por ejemplo, 2 filas y 3 columnas (es decir, 6 conjuntos de ejes repartidos de dicha forma), basta agregar estos valores como primeros argumentos de la función:

```
fig,ax=plt.subplots(3,2)
fig.set_size_inches(8,6)

type(ax)
ax.shape
ax
```

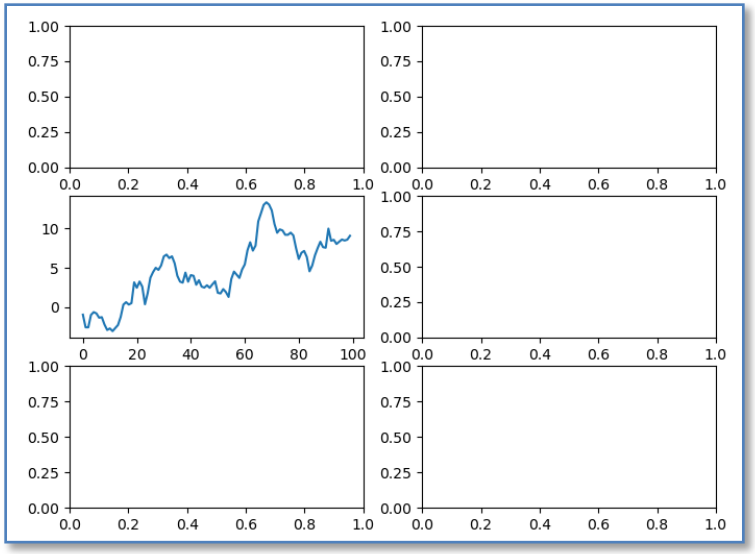


Ahora podríamos ejecutar el método plot asociado a cada uno de estos ejes para mostrar una gráfica. Por ejemplo, si quisiéramos mostrarla en la segunda fila (cuyo índice es 1) y primera columna (cuyo índice es 0), podríamos hacerlo del siguiente modo:

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

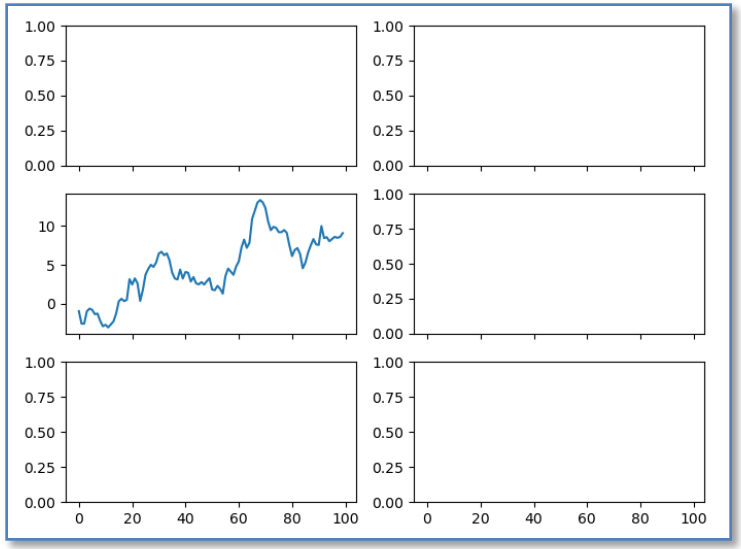


Subplots



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.

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



Otros ejemplos, ejecutar: subplot_demo.ipynb o subplot_demo.py

https://matplotlib.org/stable/tutorials/introductory/usage.html?highlight=usage

https://matplotlib.org/stable/tutorials/introductory/lifecycle.html

https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.ion.html

https://matplotlib.org/stable/tutorials/colors/colors.html

https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.show.html

https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.draw.html

https://matplotlib.org/stable/api/axes_api.html

https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.draw.html

https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.subplots.html

FIN