



Matplotlib

Es una biblioteca para la generación de gráficos.

[Ver más \(https://python-charts.com/\)](https://python-charts.com/)

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

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**.

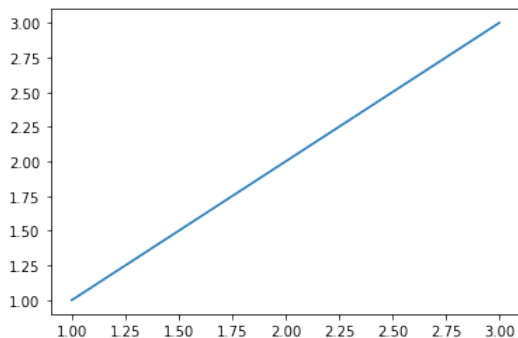
Enfoques para trazar

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]: 1 import matplotlib.pyplot as plt
2 %matplotlib inline
3 # Inicializar un objeto figura
4 fig = plt.figure()
5
6 # Agregue un objeto Axes a La Figura usando add_subplot(1,1,1)
7 # (1,1,1) aquí le dice que agregue una cuadrícula de 1x1, 1ra subtrama
8 ax = fig.add_subplot(1,1,1)
9
10 x = [1,2,3]
11 y = [1,2,3]
12
13 # Trazar los datos y mostrarlos
14 ax.plot(x, y)
15 plt.show()
16
17 # En lugar de generar el objeto Figure y Axes por separado, podemos inicializarlos juntos:
18
19 # fig, ax = plt.subplots()
20 # ax.plot(x, y)
21 # plt.show()
```

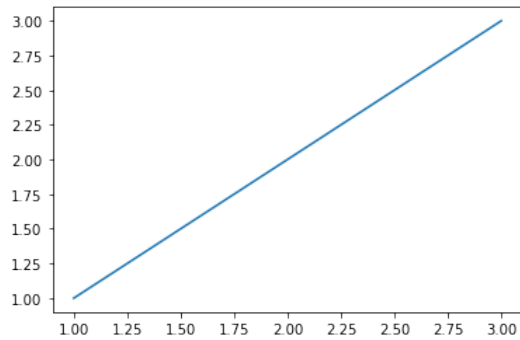


Interfaz Pyplot

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 [2]: 1 x = [1,2,3]
2 y = [1,2,3]
3 plt.plot(x,y)
4 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?

Más info en: <https://stackoverflow.com/questions/37970424/what-is-the-difference-between-drawing-plots-using-plot-axes-or-figure-in-matplotlib>
(<https://stackoverflow.com/questions/37970424/what-is-the-difference-between-drawing-plots-using-plot-axes-or-figure-in-matplotlib>)

Backends

Matplotlib tiene una serie de backends diferentes disponibles y cuando se utiliza 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 [3]: 1 %matplotlib notebook
2 import matplotlib.pyplot as plt
3 plt.get_backend()
```

Out[3]: 'nbAgg'

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

```
In [4]: 1 %matplotlib inline
2 import matplotlib.pyplot as plt
3 plt.get_backend()
```

Out[4]: 'module://matplotlib_inline.backend_inline'

```
In [5]: 1 matplotlib -l
```

Available matplotlib backends: ['tk', 'gtk', 'gtk3', 'gtk4', 'wx', 'qt4', 'qt5', 'qt6', 'qt', 'osx', 'nbagg', 'notebook', 'agg', 'svg', 'pdf', 'ps', 'inline', 'ipympl', 'widget']

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):

El lienzo en el que se representa la figura.

Capa de artista (Artist layer):

Es el objeto que sabe tomar el `Renderer` (el pincel) de la capa `Back-end` para dibujar en el lienzo. (`figure`)

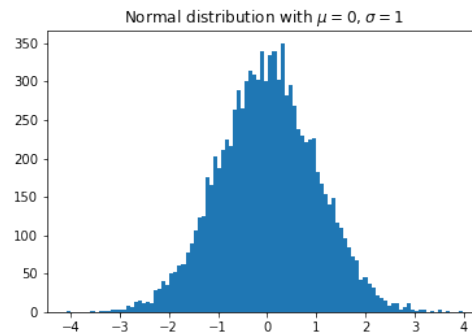
A continuación se muestra una secuencia de comandos de Python simple que ilustra la arquitectura anterior. Define el backend, se conecta `Figure` a él, usa la biblioteca de matrices `numpy` para crear 10,000 números aleatorios normalmente distribuidos y traza un histograma.

```

In [6]: ► 1 # Importe el FigureCanvas desde el backend de su elección y adjunte el Artista de la figura a él.
2 from matplotlib.backends.backend_agg import FigureCanvasAgg as FigureCanvas
3 from matplotlib.figure import Figure
4 fig = Figure()
5 canvas = FigureCanvas(fig)
6
7 # Importar la biblioteca numpy para generar los números aleatorios.
8 import numpy as np
9 x = np.random.randn(10000)
10
11 # Ahora usa un método de figura para crear un artista de Axes; el artista Axes se agrega automáticamente al
12 # contenedor de figuras fig.axes. Aquí "111" es de la convención de MATLAB: cree una cuadrícula con 1 fila
13 # y 1 columna, y use la primera celda en esa cuadrícula para la ubicación de los nuevos ejes.
14 ax = fig.add_subplot(111)
15
16 # Llame al método Axes hist para generar el histograma; hist crea una secuencia de artistas Rectangle
17 # para cada barra de histograma y los agrega al contenedor Axes. Aquí "100" significa crear 100 contenedores (bins).
18 ax.hist(x, 100)
19
20 # Decora la figura con un título y guárdala.
21 ax.set_title('Normal distribution with  $\mu=0$ ,  $\sigma=1$ ')
22 # plt.show()

```

Out[6]: Text(0.5, 1.0, 'Normal distribution with $\mu=0$, $\sigma=1$ ')



Capa de scripting (Scripting layer):

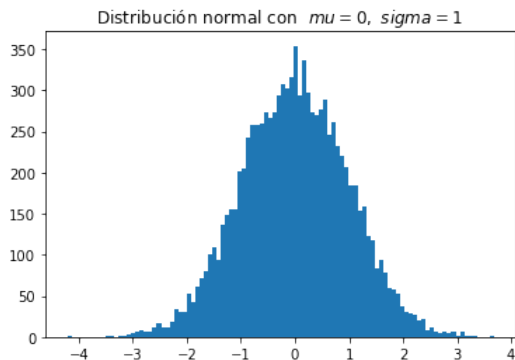
La capa de scripting de Matplotlib es esencialmente la interfaz, que automatiza el proceso de definir un lienzo y definir una instancia de artista de figuras y conectarlos.

El mismo código anterior usando **pyplot**:

```

In [7]: ► 1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 x = np.random.randn (10000)
5 plt.hist (x, 100)
6 plt.title (r'Distribución normal con  $\mu = 0$ ,  $\sigma = 1$  ')
7 plt.show ()

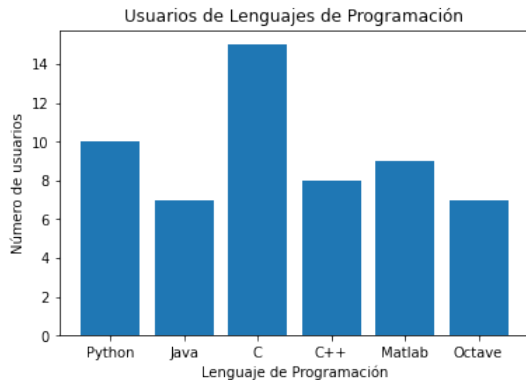
```



Ejemplo que muestra la funcionalidad de la capa Artist de Matplotlib

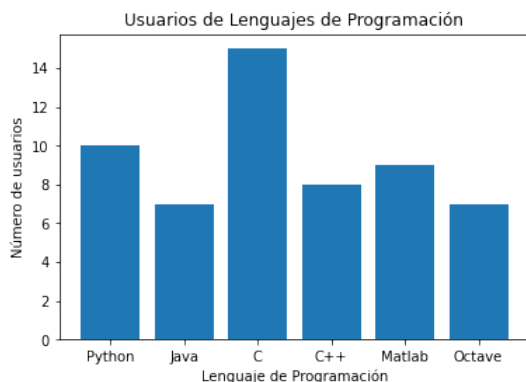
```
In [8]: ▶ 1 import matplotlib.pyplot as plt
2
3 # Creamos 2 Listas con datos para el gráfico
4 x = ['Python', 'Java', 'C', 'C++', 'Matlab', 'Octave']
5 usuarios = [10, 7, 15, 8, 9, 7]
6
7 # Uso de la capa de artista
8 fig = plt.figure() # Se crea una figura
9 ax = fig.add_subplot(111) # Nuevos ejes creados
10 #el número 111 significa que sólo queremos una trama
11
12 # Interfaz orientada a objetos
13 # ax es el objeto y los elementos de gráfico se agregan utilizando diferentes métodos de este objeto
14 ax.bar(x, usuarios)
15 ax.set_title('Usuarios de Lenguajes de Programación')
16 ax.set_xlabel('Lenguaje de Programación')
17 ax.set_ylabel('Número de usuarios')
18 # plt.show()
```

Out[8]: Text(0, 0.5, 'Número de usuarios')



Ejemplo que muestra la funcionalidad de la capa Scripting de Matplotlib

```
In [9]: ▶ 1 import matplotlib.pyplot as plt
2
3 # Creamos 2 Listas con datos para el gráfico
4 x = ['Python', 'Java', 'C', 'C++', 'Matlab', 'Octave']
5 usuarios = [10, 7, 15, 8, 9, 7]
6
7 # Uso de la capa de scripting
8 # No hay necesidad de crear figura y eje antes de trazar
9 plt.bar(x, usuarios)
10 plt.title('Usuarios de Lenguajes de Programación')
11 plt.xlabel('Lenguaje de Programación')
12 plt.ylabel('Número de usuarios')
13 plt.show()
```



En cualquiera de los casos, hay que generar los valores de la variable independiente 'x' (x_values) y los valores de la variable dependiente 'y' (y_values). En primer lugar, genero un gráfico con la opción más sencilla, siguiendo este formato:

librería + tipo de gráfico + (valores de x, valores de y)

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
- Puede utilizarse de dos formas:

Opción 1. Creando una figura que representa al gráfico, indicando el número de Axes y después, mostrando la figura. Esto se conoce como Object-oriented style. #Opción 1 -> fig, ax = plt.subplots() #Creación de una figura con un axis

De esta forma es muy sencillo generar un gráfico, pero la librería genera uno estándar sin configurar el eje x o y, sin incluir nombres, sin etiquetas y sin personalización, que al fin y al cabo es lo que permite que un gráfico potencie la información que se quiere representar.

Personalizar cada gráfico

Para generar un gráfico con matplotlib que represente mejor visualmente todo el potencial de los datos, necesitamos estructurar el gráfico en diferentes partes. El gráfico está formado por:

- La figura
- El área del gráfico. Dentro de una figura puede haber más de un gráfico
- El tipo de representación: línea, barras, histograma, scatter o tarta, entre otros.
- El eje x
- El eje y
- La etiqueta del eje x
- La etiqueta del eje y
- El título del gráfico
- La leyenda

Configuración de estilos

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

```
In [10]: 1 plt.style.use('classic') # Declaración del estilo
```

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

La forma en que ve los gráficos Matplotlib 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

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 [11]: 1 plt.rcParams['interactive']
```

Out[11]: True

```
In [12]: 1 plt.isinteractive()
```

Out[12]: True

```
In [13]: 1 plt.ioff()
```

Out[13]: <matplotlib.pyplot._IoffContext at 0x1dcf3fb1cd0>

```
In [14]: 1 plt.rcParams['interactive']
```

Out[14]: False

```
In [15]: 1 plt.ion()
```

Out[15]: <matplotlib.pyplot._IonContext at 0x1dcf3ff9c70>

```
In [16]: 1 plt.isinteractive()
```

Out[16]: True

Función gráfica

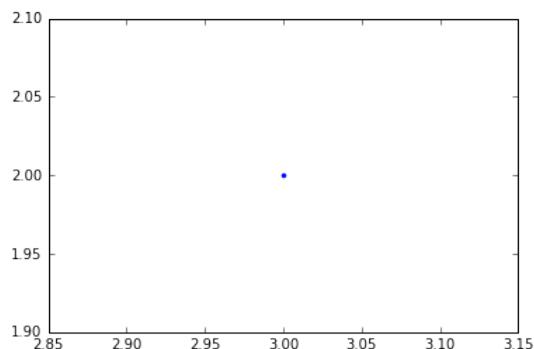
Un trazado tiene dos ejes, un eje X, horizontal y un eje Y, vertical. Utilizamos la capa de scripting pyplot como `plt`. Como se mencionó, todas las funciones que se ejecutarán con el módulo pyplot forman parte de la capa de scripting de la arquitectura.

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 [17]: 1 import matplotlib.pyplot as plt
2 %matplotlib inline
3 plt.plot(3,2, '.')
4 plt.show()
```



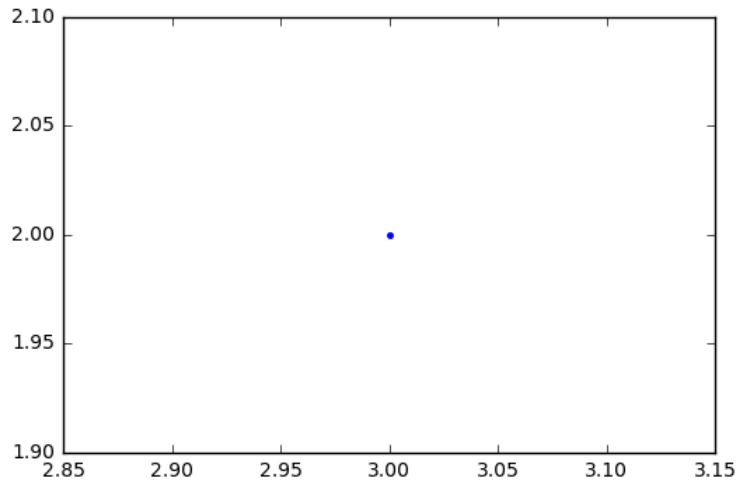
El tercer argumento debe ser una cadena que significa cómo queremos que se renderice ese punto de datos.

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 [18]: 1 from matplotlib.backends.backend_agg import FigureCanvasAgg
2         from matplotlib.figure import Figure
3
4         fig = Figure()
5         # Se debe adjuntar un Lienzo manualmente a la figura (pyplot lo haría automáticamente).
6         # Esto se hace instanciando el lienzo con la figura como argumento.
7         canvas = FigureCanvasAgg(fig)
8         ax = fig.add_subplot(111)
9         ax.plot(3,2, '.')
10        # canvas.print_png('img/test.png')
11        fig.savefig('img/test.png')
```

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

```
In [19]: 1 %%html
2         <img src='img/test.png' />
```



Más información respecto de las imágenes (<https://stackoverflow.com/questions/13714454/specifying-and-saving-a-figure-with-exact-size-in-pixels>).

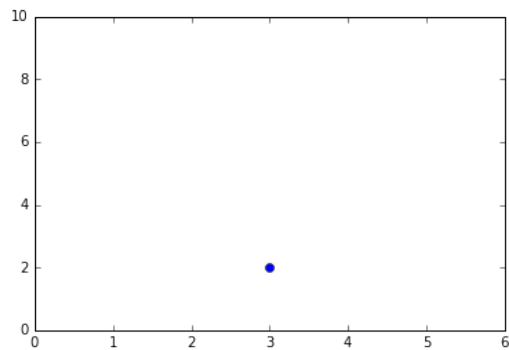
Cuando hacemos una llamada a pyplot con plt.plot, la capa de secuencias de comandos busca ver si hay una figura que existe actualmente, si no la hay, crea una nueva.

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 [20]: 1 import matplotlib.pyplot as plt
2
3         plt.figure()
4         plt.plot(3,2, 'o')
5         ax = plt.gca()
6         ax.axis([0,6,0,10])
```

Out[20]: (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.

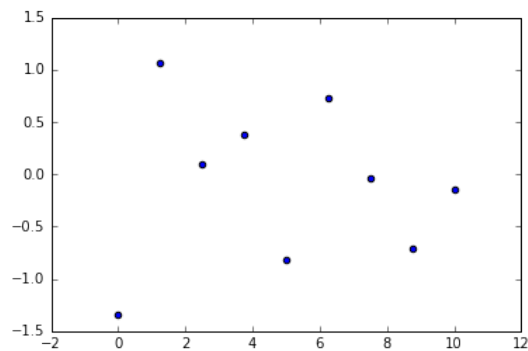
Como estamos haciendo esto con la capa de scripting, si estamos trabajando en Jupyter una vez que ejecutamos la celda, se renderiza con el back-end nbAgg

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 [21]: 1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 x = np.linspace(0,10,9)
5 y = np.random.randn(9)
6
7 plt.scatter(x,y)
8 fig = plt.gcf()
9 fig.set_size_inches(6,4)
```



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

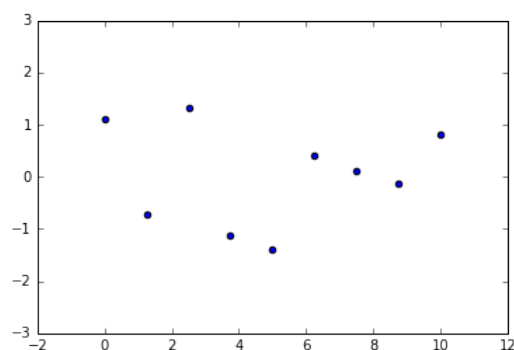
Pyplot: plt.gca ()

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 [22]: 1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 x = np.linspace(0,10,9)
5 y = np.random.randn(9)
6
7 plt.scatter(x,y)
8 axis = plt.gca()
9 axis.set_ylim(-3,3)
```

Out[22]: (-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.

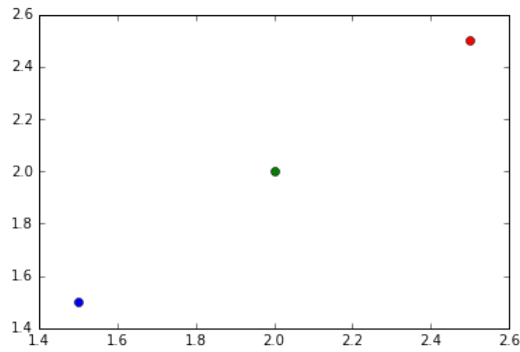
Se puede agregar Artistas a un objeto de ejes en cualquier momento, pyplot hace eso por nosotros cuando llamamos a la función gráfica.

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 [23]: 1 import matplotlib.pyplot as plt
2
3 plt.figure()
4 plt.plot(1.5, 1.5, 'o')
5 plt.plot(2,2,'o')
6 plt.plot(2.5,2.5,'o')
```

Out[23]: [matplotlib.lines.Line2D at 0x1dcf5046340]



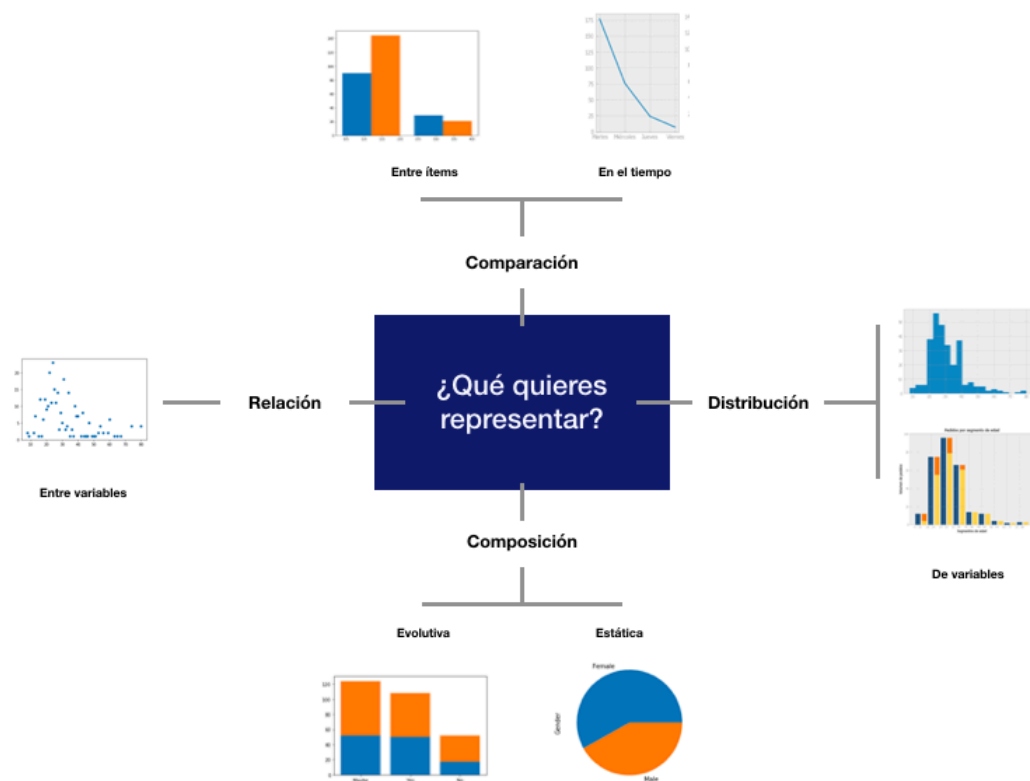
Tipos gráficos

¿Qué tipo de gráfica utilizar en cada caso?

No todas las gráficas sirven para representar visiblemente todos los conjuntos de datos. A la hora de decidir qué gráfico seleccionar tienes que responder a dos preguntas:

- ¿Qué pregunta quieres responder con el gráfico?
- ¿Qué tipo de dato quieres representar?

Los gráficos se dividen en 4 grandes grupos:



Scatterplots

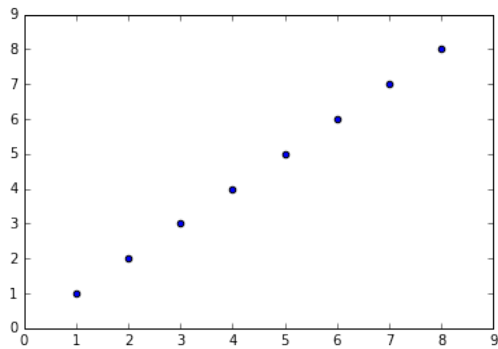
Tramas de dispersión

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 [24]: 1 import matplotlib.pyplot as plt
2
3 x = [1,2,3,4,5,6,7,8]
4 y = x
5 plt.figure()
6 plt.scatter(x, y)
```

Out[24]: <matplotlib.collections.PathCollection at 0x1dcf3e25730>

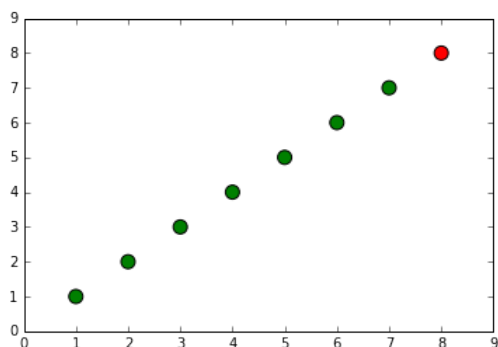


En esta trama puede verse una línea diagonal y matplotlib ha dimensionado nuestros ejes en consecuencia. Pero la dispersión no representa elementos como una serie.

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 [25]: 1 import matplotlib.pyplot as plt
2
3 x = [1,2,3,4,5,6,7,8]
4 y = x
5 colors = ['green'] * (len(x)-1)
6 colors.append('red')
7 plt.figure()
8 plt.scatter(x, y, s=100, c=colors)
```

Out[25]: <matplotlib.collections.PathCollection at 0x1dcf50738b0>



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

```
In [26]: 1 colors
```

Out[26]: ['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 [27]: 1 zip_generador = zip([1,2,3,4,5],[6,7,8,9,10])
2 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 desempaqueado 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:

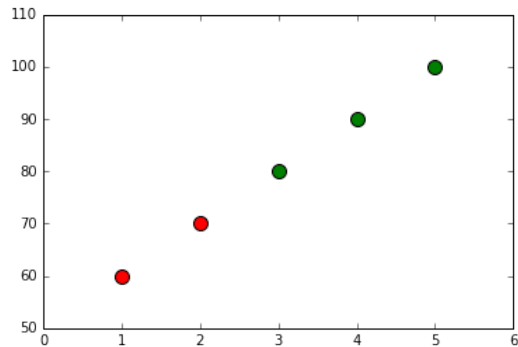
```
In [28]: 1 zip_generator = zip([1,2,3,4,5],[6,7,8,9,10])
2 x, y = zip(*zip_generator)
3 x, y
```

```
Out[28]: ((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 [29]: 1 import matplotlib.pyplot as plt
2
3 zip_generator=zip([1,2,3,4,5],[60,70,80,90,100])
4 x,y=zip(*zip_generator)
5 plt.figure()
6
7 plt.scatter(x[:2],y[:2],s=100,c='red',label='Ingresos más bajos')
8 plt.scatter(x[2:],y[2:],s=100,c='green',label='Ingresos más altos')
9
10 #x[:2],y[:2] -> ((1, 2), (60, 70))
11 #x[2:],y[2:] -> ((3, 4, 5), (80, 90, 100))
```

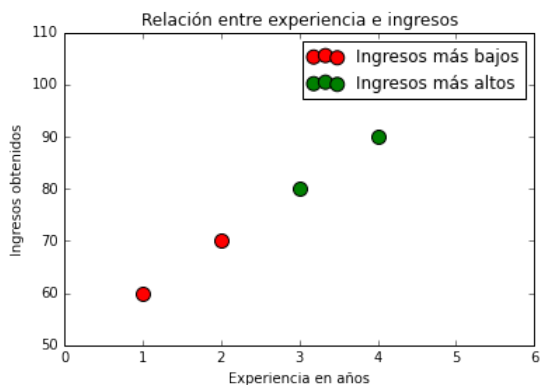
```
Out[29]: <matplotlib.collections.PathCollection at 0x1dcf50fb880>
```



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 [30]: 1 import matplotlib.pyplot as plt
2
3 zip_generator=zip([1,2,3,4,5],[60,70,80,90,100])
4 x,y=zip(*zip_generator)
5 plt.figure()
6
7 plt.scatter(x[:2],y[:2],s=100,c='red',label='Ingresos más bajos')
8 plt.scatter(x[2:],y[2:],s=100,c='green',label='Ingresos más altos')
9 plt.xlabel('Experiencia en años')
10 plt.ylabel('Ingresos obtenidos')
11 plt.title('Relación entre experiencia e ingresos')
12 plt.legend()
```

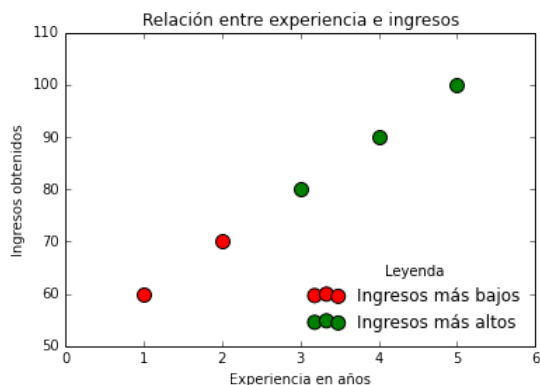
```
Out[30]: <matplotlib.legend.Legend at 0x1dcf514d580>
```



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 [31]: 1 import matplotlib.pyplot as plt
2
3 zip_generator=zip([1,2,3,4,5],[60,70,80,90,100])
4 x,y=zip(*zip_generator)
5 plt.figure()
6
7 plt.scatter(x[:2],y[:2],s=100,c='red',label='Ingresos más bajos')
8 plt.scatter(x[2:],y[2:],s=100,c='green',label='Ingresos más altos')
9 plt.xlabel('Experiencia en años')
10 plt.ylabel('Ingresos obtenidos')
11 plt.title('Relación entre experiencia e ingresos')
12 plt.legend(loc=4,frameon=False,title='Leyenda')
```

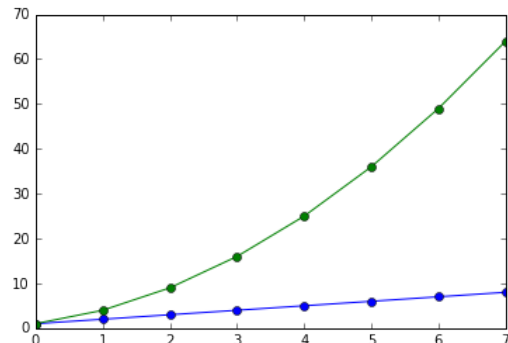
Out[31]: <matplotlib.legend.Legend at 0x1dcf5197c10>



Line plots

```
In [32]: 1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 datos_lins = np.array([1,2,3,4,5,6,7,8])
5 datos_cuads = datos_lins**2
6 plt.figure()
7 plt.plot(datos_lins, '-o', datos_cuads, '-o')
```

Out[32]: [<matplotlib.lines.Line2D at 0x1dcf525e730>, <matplotlib.lines.Line2D at 0x1dcf525e6a0>]



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

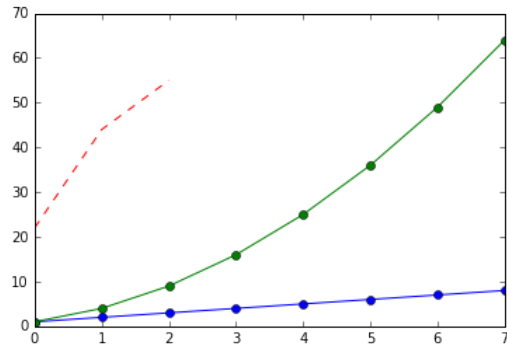
Ambas están usando puntos porque usamos la bandera **"-o"**, elemento nuevo respecto de las tramas de dispersión.

- 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 [33]: 1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 datos_lins = np.array([1,2,3,4,5,6,7,8])
5 datos_cuads = datos_lins**2
6
7 plt.figure()
8
9 plt.plot(datos_lins, '-o', datos_cuads, '-o')
10 plt.plot([22,44,55], '--r')
```

Out[33]: [matplotlib.lines.Line2D at 0x1dcf3fe4c70>]

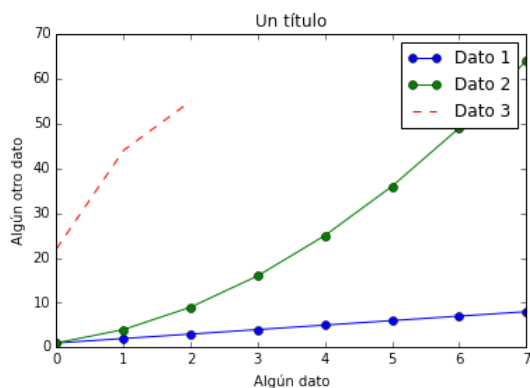


Puede utilizarse las funciones de ejes creando etiquetas para los ejes y para la figura como un todo.

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 misma:

```
In [34]: 1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 datos_lins = np.array([1,2,3,4,5,6,7,8])
5 datos_cuads = datos_lins**2
6 plt.figure()
7
8 plt.plot(datos_lins, '-o', datos_cuads, '-o')
9 plt.plot([22,44,55], '--r')
10 plt.xlabel('Algún dato')
11 plt.ylabel('Algún otro dato')
12 plt.title('Un título')
13 plt.legend(['Dato 1', 'Dato 2', 'Dato 3'])
```

Out[34]: <matplotlib.legend.Legend at 0x1dcf526e850>



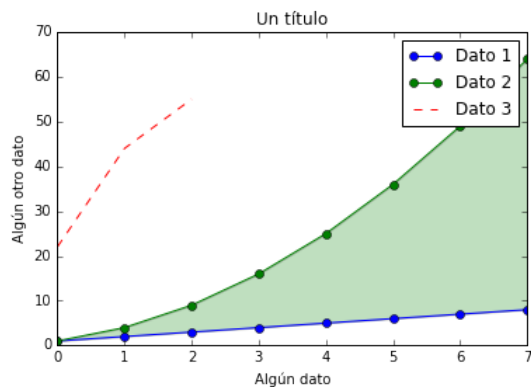
Función de relleno

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 [35]: 1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 datos_lins = np.array([1, 2, 3, 4, 5, 6, 7, 8])
5 datos_cuads = datos_lins**2
6 plt.figure()
7
8 plt.plot(datos_lins, '-o', datos_cuads, '-o')
9 plt.plot([22, 44, 55], '--r')
10 plt.xlabel('Algún dato')
11 plt.ylabel('Algún otro dato')
12 plt.title('Un título')
13 plt.legend(['Dato 1', 'Dato 2', 'Dato 3'])
14
15 plt.gca().fill_between(range(len(datos_lins)),
16 datos_lins, datos_cuads,
17 facecolor='green', alpha=0.25)
```

Out[35]: <matplotlib.collections.PolyCollection at 0x1dcf5364a00>



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.

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().

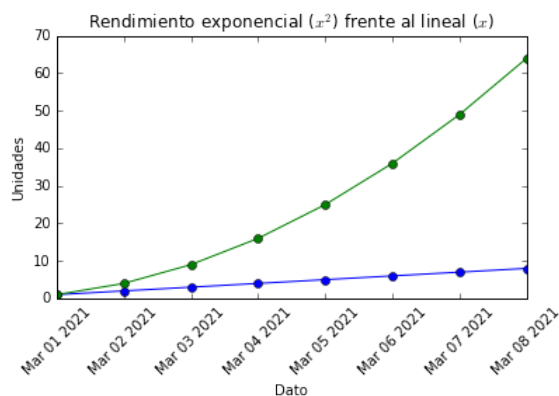
Existen muchas propiedades del objeto de ejes, por ejemplo, se puede obtener las líneas de cuadrícula, las ubicaciones de ticks para las etiquetas principales y secundarias, etc.

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 [36]:

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3 import pandas as pd
4
5 datos_lins = np.array([1, 2, 3, 4, 5, 6, 7, 8])
6 datos_cuads = datos_lins**2
7
8 plt.figure()
9
10 datos_observados = np.arange('2021-03-01', '2021-03-09', dtype='datetime64[D]')
11 datos_observados = list(map(pd.to_datetime, datos_observados))
12
13 plt.plot(datos_observados, datos_lins, '-o', datos_observados, datos_cuads, '-o')
14 x = plt.gca().xaxis
15
16 for item in x.get_ticklabels():
17     item.set_rotation(45)
18
19 plt.subplots_adjust(bottom=0.25)
20 ax = plt.gca()
21 ax.set_xlabel('Dato')
22 ax.set_ylabel('Unidades')
23 ax.set_title('Rendimiento exponencial ( $x^2$ ) frente al lineal ( $x$ )')
```

Out[36]: Text(0.5, 1.0, 'Rendimiento exponencial (x^2) frente al lineal (x)')



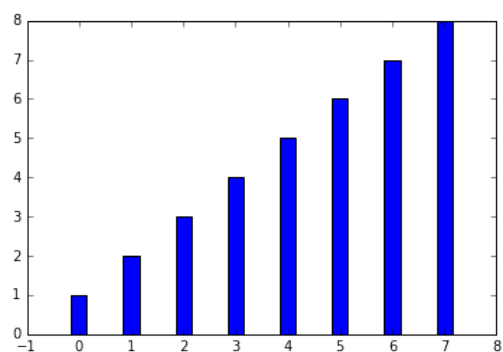
Bar charts

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 [37]:

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 datos_lins = np.array([1, 2, 3, 4, 5, 6, 7, 8])
5 plt.figure()
6
7 xvals = range(len(datos_lins))
8 plt.bar(xvals, datos_lins, width = 0.3)
```

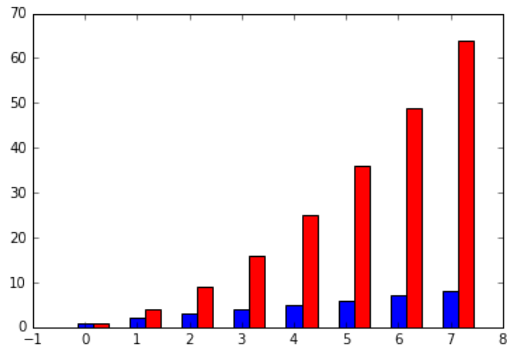
Out[37]: <BarContainer object of 8 artists>



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 [38]: 1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 datos_lins = np.array([1, 2, 3, 4, 5, 6, 7, 8])
5 datos_cuads = datos_lins**2
6
7 plt.figure()
8
9 xvals = range(len(datos_lins))
10 plt.bar(xvals, datos_lins, width = 0.3)
11 new_xvals = []
12
13 for item in xvals:
14     new_xvals.append(item+0.3)
15
16 plt.bar(new_xvals, datos_cuads, width = 0.3 ,color='red')
```

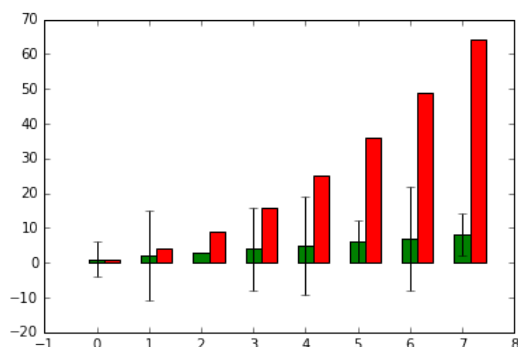
Out[38]: <BarContainer object of 8 artists>



Las etiquetas se pueden centrar usando el parámetro `align`. Pueden agregarse **barras de error** a cada barra, utilizando el parámetro `yerr`. 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 [39]: 1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 datos_lins = np.array([1, 2, 3, 4, 5, 6, 7, 8])
5 datos_cuads = datos_lins**2
6 plt.figure()
7 xvals = range(len(datos_lins))
8 plt.bar(xvals, datos_lins, width = 0.3)
9
10 new_xvals = []
11 for item in xvals:
12     new_xvals.append(item+0.3)
13 plt.bar(new_xvals, datos_cuads, width = 0.3 ,color='red')
14
15 from random import randint
16
17 linear_err = [randint(0,15) for x in range(len(datos_lins))]
18 plt.bar(xvals, datos_lins, width = 0.3, yerr=linear_err)
```

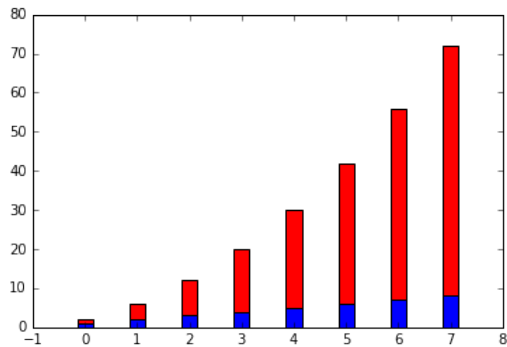
Out[39]: <BarContainer object of 8 artists>



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 [40]: 1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 datos_lins = np.array([1, 2, 3, 4, 5, 6, 7, 8])
5 datos_cuads = datos_lins**2
6
7 xvals = range(len(datos_lins))
8 plt.bar(xvals, datos_lins, width = 0.3, color='b')
9 plt.bar(xvals, datos_cuads, width = 0.3, bottom=datos_lins, color='r')
```

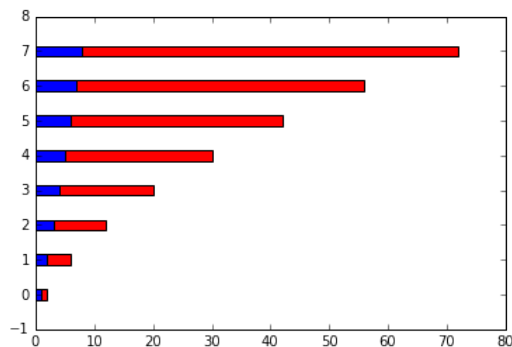
Out[40]: <BarContainer object of 8 artists>



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 [41]: 1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 datos_lins = np.array([1, 2, 3, 4, 5, 6, 7, 8])
5 datos_cuads = datos_lins**2
6
7 xvals = range(len(datos_lins))
8 plt.barh(xvals, datos_lins, height = 0.3, color='b')
9 plt.barh(xvals, datos_cuads, height = 0.3, left=datos_lins, color='r')
```

Out[41]: <BarContainer object of 8 artists>

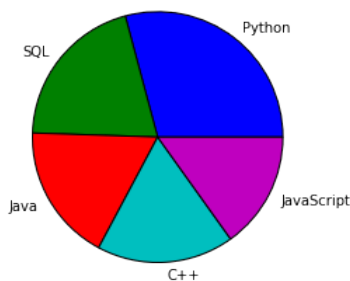


Gráficos circulares

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.

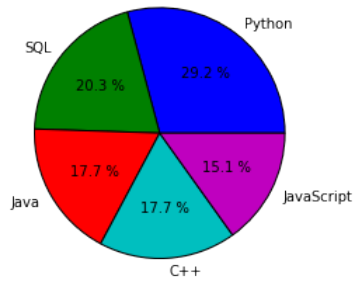
La función que nos permite mostrar un gráfico circular es **matplotlib.pyplot.pie**, existiendo también un método del conjunto de ejes, **matplotlib.axes.Axes.pie** con la misma funcionalidad.

```
In [42]: 1 import matplotlib.pyplot as plt
2
3 popularity = [56, 39, 34, 34, 29]
4 languages = ['Python', 'SQL', 'Java', 'C++', 'JavaScript']
5 plt.pie(popularity, labels=languages)
6 plt.show()
```



In [43]:

```
1 import matplotlib.pyplot as plt
2
3 popularity = [56, 39, 34, 34, 29]
4 languages = ['Python', 'SQL', 'Java', 'C++', 'JavaScript']
5 plt.pie(popularity, labels=languages, autopct="%0.1f %")
6 plt.show()
```



In [44]:

```
1 import matplotlib.pyplot as plt
2
3 popularity = [56, 39, 34, 34, 29]
4 languages = ['Python', 'SQL', 'Java', 'C++', 'JavaScript']
5 colores = ["#13EAC9", "#FC5A50", "#AAFF32", "#FFD700", "#9A0EEA"]
6 desfase = (0, 0, 0.5, 0, 0.1)
7 plt.pie(popularity, labels=languages, autopct="%0.1f %", colors=colores, explode=desfase)
8 plt.show()
```

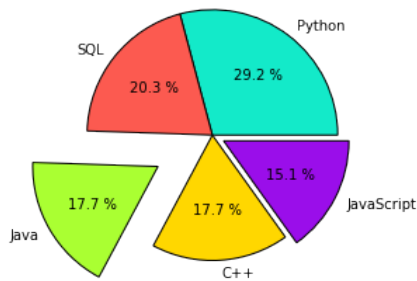


Diagrama de caja

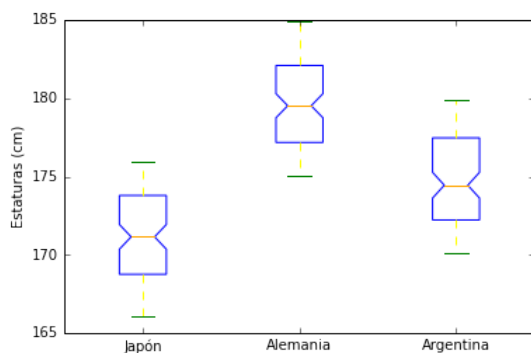
También se conoce como diagrama de bigotes y se crea para mostrar el resumen del conjunto de valores de datos que tienen propiedades como mínimo, primer cuartil, mediana, tercer cuartil y máximo.

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.

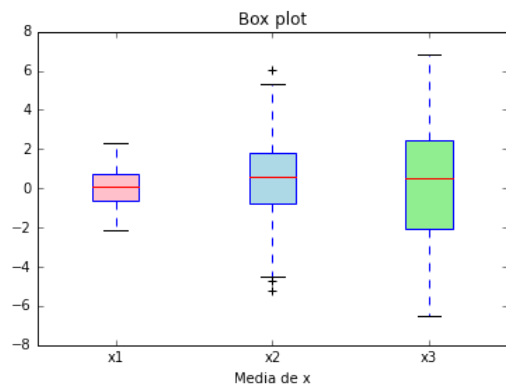
In [45]:

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 jap = np.random.uniform(166, 176, 100)
5 ale = np.random.uniform(175, 185, 100)
6 arg = np.random.uniform(170, 180, 100)
7
8 plt.boxplot([jap, ale, arg],
9 notch=True, patch_artist=True,
10 capprops=dict(color="green"),
11 medianprops=dict(color="orange"),
12 whiskerprops=dict(color="yellow"))
13 plt.xticks([1, 2, 3], ['Japón', 'Alemania', 'Argentina'])
14 plt.ylabel('Estaturas (cm)')
15 plt.show()
```



In [46]:

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 all_data = [np.random.normal(0, std, 100) for std in range(1, 4)]
5 fig = plt.figure(figsize=(6,4))
6 bplot = plt.boxplot(all_data,
7     notch=False,
8     vert=True,
9     patch_artist=True)
10 colors = ['pink', 'lightblue', 'lightgreen']
11
12 for patch, color in zip(bplot['boxes'], colors):
13     patch.set_facecolor(color)
14
15 plt.xticks([y+1 for y in range(len(all_data))], ['x1', 'x2', 'x3'])
16 plt.xlabel('Media de x')
17 t = plt.title('Box plot')
18 plt.show()
```

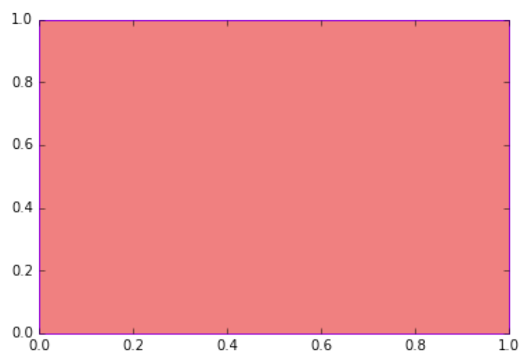


Subplots

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.

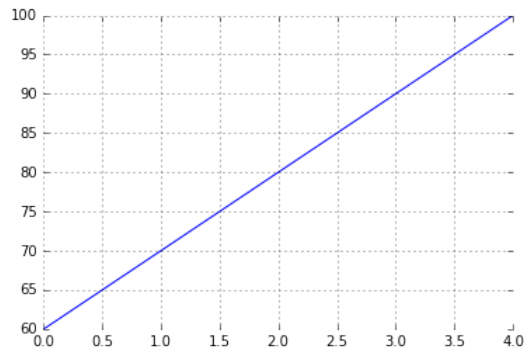
In [47]:

```
1 import matplotlib.pyplot as plt
2
3 fig,ax=plt.subplots()
4 # type(ax)
5 # Color de fondo
6 ax.set_facecolor('lightcoral')
7 # Color de Los bordes:
8 for axis in ['top', 'bottom', 'left', 'right']:
9     ax.spines[axis].set_color('darkviolet')
```



In [48]:

```
1 ax = plt.axes()
2 plt.plot(y)
3 ax.spines['right'].set_visible(False)
4 ax.spines['top'].set_visible(False)
5 ax.spines['bottom'].set_visible(False)
6 ax.spines['left'].set_visible(False)
7 plt.grid()
8 plt.show()
```



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.

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.

Por ejemplo, si quisiéramos mostrar una gráfico en la segunda fila (cuyo índice es 1) y primera columna (cuyo índice es 0), podríamos hacerlo del siguiente modo:

In [49]:

```
1 import numpy as np
2
3 y = np.random.randn(100).cumsum()
4
5 fig,ax=plt.subplots(3,2, sharex = True)
6 fig.set_size_inches(8,6)
7 ax[1,0].plot(y)
8 plt.show()
```

