MATPLOTLIB

- Matplotlib (*https://matplotlib.org/*)

El principal propósito de Matplotlib es asistirnos y facilitarnos la tarea de visualización de nuestros datos.

Entre sus puntos fuertes podríamos destacar los siguientes:

- Librería de Python más extendida para realizar gráficas de datos.

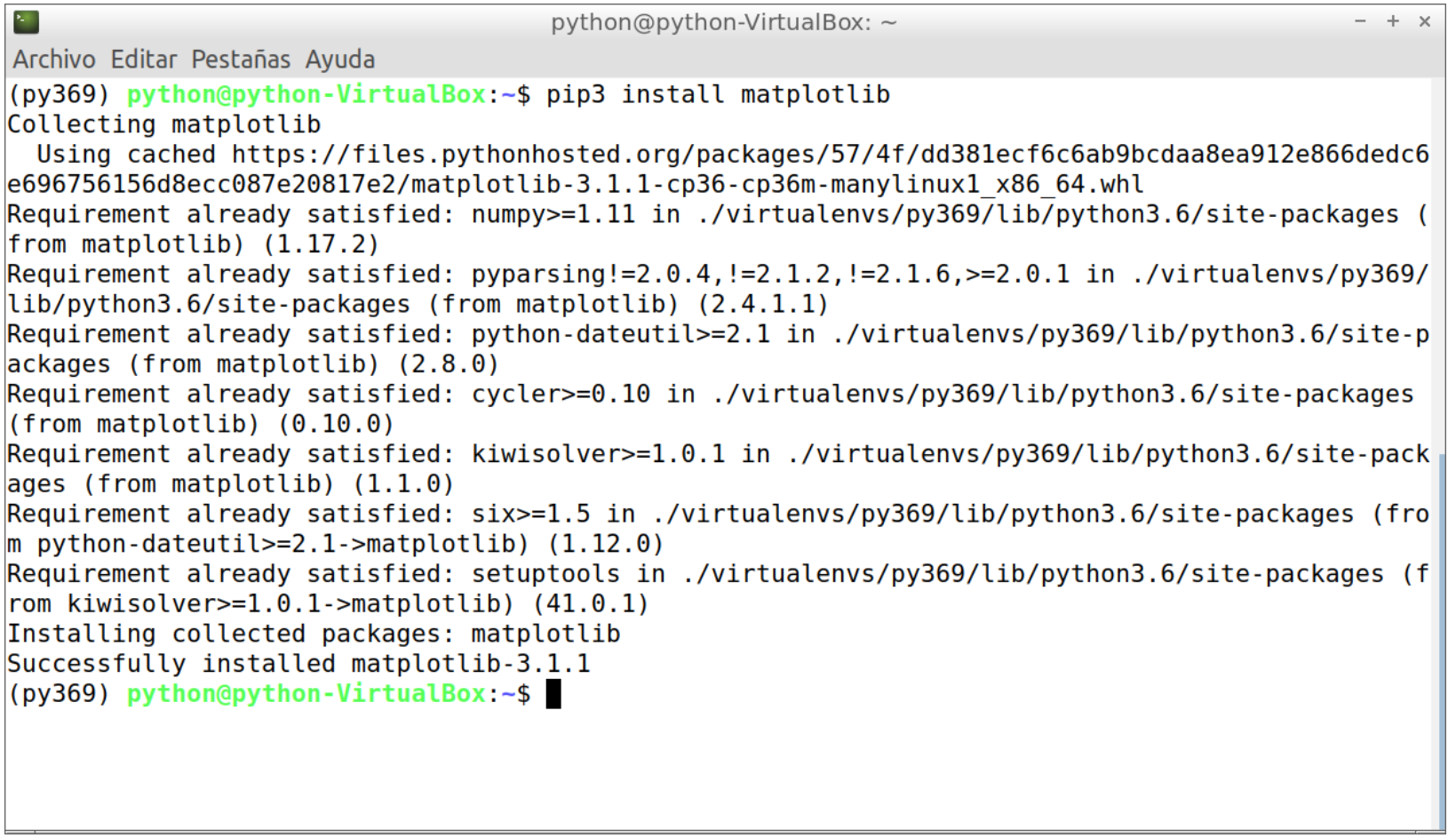
- Permite realizar multitud de gráficos a través de sencillas instrucciones de alto nivel, permitiendo a su vez actuar a más bajo nivel para controlar con detalle todo tipo de aspectos de los mismos.

- Dispone de muchos formatos gráficos de salida para utilizar en artículos, libros, webs, etc.

Format	Description
EPS	Encapsulated PostScript.
JPG	Graphic format with lossy compression method for photographic output.
PDF	**Portable Document Format (PDF).**
PNG	**Portable Network Graphics (PNG)**, a raster graphics format with a lossless compression method (more adaptable to line art than JPG).
PS	Language widely used in publishing and as printers jobs format.
SVG	**Scalable Vector Graphics (SVG)**, XML based.

- Instalación de Matplotlib (i)

Hay diferentes métodos para instalar Matplotlib, pero quizás el más sencillo sea a través del gestor de paquetes de Python denominado *PIP3* (https://piplt.pypa.io/en/stable/).



- Instalación de Matplotlib (ii)

Para asegurarnos que hemos instalado bien la librería Matplotlib nos bastaría con tratar de importarla en un intérprete de Python 3.

```
Python 3.6.9 (default, Jul 16 2019, 21:20:02)
[GCC 7.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import matplotlib
>>>
```

En caso de que Matplotlib no esté correctamente instalada aparecería este mensaje:

ModuleNotFoundError: No module named 'matplotlib'

También podemos consultar la versión que tenemos de Matplotlib:

```
>>> matplotlib.__version__
'3.1.1'
>>>
```

- ¿Qué es un *Backend*?

Python nos ofrece muchas formas de trabajar para elaborar nuestras figuras:

- Desde una Shell de Python, obteniendo las figuras en ventanas generadas a través de los comandos introducidos.

- Realizando scripts de código que generan imágenes directamente como archivos gráficos.

- Embebiendo código Matplotlib en interfaces gráficas de usuario (GUIs) como wxpython o pygtk dentro de aplicaciones.

- Generando gráficos dinámicos en páginas web.

El código que necesita escribir el programador para cada uno de esos posibles casos de uso se conoce como *"frontend"*, mientras que el código subyacente que sirve de soporte y hace que nuestro frontend funcione adecuadamente se denomina *"backend"*.

- Tipos de *Backends* (i)

User interface backends or Interactive backends: permiten renderizar nuestras figuras en ventanas de ploteado que sirven como interfaces de usuario interactivas.

Backend	Description
GTKAgg	Agg rendering to a GTK 2.x canvas (requires PyGTK and pycairo or cairocffi; Python2 only)
GTK3Agg	Agg rendering to a GTK 3.x canvas (requires PyGObject and pycairo or cairocffi)
GTK	GDK rendering to a GTK 2.x canvas (not recommended and d eprecated in 2.0) (requires PyGTK and pycairo or cairocffi; Python2 only)
GTKCairo	Cairo rendering to a GTK 2.x canvas (requires PyGTK and pycairo or cairocffi; Python2 only)
GTK3Cairo	Cairo rendering to a GTK 3.x canvas (requires PyGObject and pycairo or cairocffi)
WXAgg	Agg rendering to to a wxWidgets canvas (requires wxPython)
WX	Native wxWidgets drawing to a wxWidgets Canvas (not recommended and deprecated in 2.0) (requires wxPython)
TkAgg	Agg rendering to a Tk canvas (requires TkInter)
Qt4Agg	Agg rendering to a Qt4 canvas (requires PyQt4 or `pyside`)
Qt5Agg	Agg rendering in a Qt5 canvas (requires PyQt5)
macosx	Cocoa rendering in OSX windows (presently lacks blocking show() behavior when matplotlib is in non-interactive mode)

- Tipos de *Backends* (ii)

Hardcopy backends or Non-interactive backends: permiten almacenar nuestras figuras directamente en ficheros con formato gráfico.

Renderer	Filetypes	Description
AGG	png	raster graphics – high quality images using the Anti-Grain Geometry engine
PS	ps eps	vector graphics – Postscript output
PDF	pdf	vector graphics – Portable Document Format
SVG	svg	vector graphics – Scalable Vector Graphics
Cairo	png ps pdf svg ...	vector graphics – Cairo graphics
GDK	png jpg tiff ...	raster graphics – the Gimp Drawing Kit Deprecated in 2.0

- Configurar el *Backend (i)*

1. A través del parámetro *backend* en el fichero *matplotlibrc.*

   plt.e: *backend : qt5agg # use pyqt5 with antigrain (agg) rendering*

Podemos averiguar donde se encuentra nuestro fichero *matplotlibrc* usando el propio Shell de Python, a través de la función *matplotlib_fname* del paquete *matplotlib*:

```
>>> matplotlib.matplotlib_fname()
'/home/python/virtualenvs/py369/lib/python3.6/site-packages/matplotlib/mpl-data/matplotlibrc'
>>>
```

- Configurar el *Backend (ii)*

2. Estableciendo la variable de entorno *MPLBACKEND*.

```
(py369) python@python-VirtualBox:~$ export MPLBACKEND=qt5agg
(py369) python@python-VirtualBox:~$ echo $MPLBACKEND
qt5agg
(py369) python@python-VirtualBox:~$
```

La declaración de esta variable de entorno tendrá precedencia sobre lo establecido en el fichero de configuración *matplotlibrc.*

Podemos conocer el backend actual con la función *get_backend()* del módulo matplotlib.

```
>>> import matplotlib
>>> matplotlib.get_backend()
'Qt5Agg'
>>>
```

- Configurar el *Backend (iii)*

3. Usando la función *use()* del paquete Matplotlib.

```
>>> import matplotlib
>>> matplotlib.use('ps')
>>> matplotlib.get_backend()
'ps'
>>> import matplotlib.pyplot
>>>
```

Nota: Para que surta efecto, la función *use()* debe ser ejecutada antes de importar el módulo pyplot

- Ploteado interactivo vs no interactivo (i)

Interactivo

- Permite visualizar en una ventana la imagen o figura generada de forma automática al utilizar la función *plot().*
- Podemos ejecutar otros comandos mientras estamos visualizando la imagen en pantalla.

No interactivo

- La imagen no se dibuja automáticamente en pantalla cuando se utiliza la función *plot()* sino que tendremos que utilizar además una llamada a la función *show().*
- Mientras la imagen está dibujada en pantalla, la ejecución de comandos está detenida hasta que la ventana gráfica se cierre de forma manual.

- Ploteado interactivo vs no interactivo (ii)

- Por defecto, el modo interactivo está desactivado.

- Podemos consultar el estado del modo interactivo con la función *is_interactive()* de Matplotlib:

```
>>> import matplotlib as mpl
>>> mpl.is_interactive()
False
>>>
```

- Una forma de activar el modo interactivo por defecto es a través del fichero de configuración *matplotlibrc:*

interactive : True

- Otra manera de activarlo es a través de la función *interactive()* de Matplotlib.

```
>>> mpl.interactive('True')
>>> mpl.is_interactive()
True
>>>
```

- También podemos usar las funciones *ion()* y *ioff()* del módulo *pyplot* de Matplotlib.

```
>>> import matplotlib.pyplot as plt
>>> plt.ion()
>>>
```

- Ploteado interactivo vs no interactivo (iii)

IPython Shell

Está diseñado para trabajar de forma interactiva cuando es necesario. No obstante, si vamos a trabajar con Matplotlib es conveniente indicárselo a través de alguno de los dos siguientes métodos, para aumentar la compatibilidad entre ambos:

1. Al iniciar IPython:

```
(py369) python@python-VirtualBox:~$ ipython --matplotlib
/home/python/virtualenvs/py369/lib/python3.6/site-packages/IPython/core/history.py:226: UserWarning: IPython History requires SQLite,
your history will not be saved
  warn("IPython History requires SQLite, your history will not be saved")
Python 3.6.9 (default, Jul 16 2019, 21:20:02)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.7.0 -- An enhanced Interactive Python. Type '?' for help.
Using matplotlib backend: Qt5Agg

In [1]:
```

2. Desde el Shell de IPython mediante la magic function *%matplotlib*:

```
In [1]: %matplotlib
Using matplotlib backend: Qt5Agg

In [2]:
```

- Sobre los datos a plotear

Todos los datos a plotear deberán ser del tipo *array*.

Numpy es un paquete que sirve para trabajar con arrays de forma eficiente gracias a que nos proporciona una enorme cantidad de métodos y funciones aplicables a los mismos que están diseñados y optimizados para el cálculo científico.

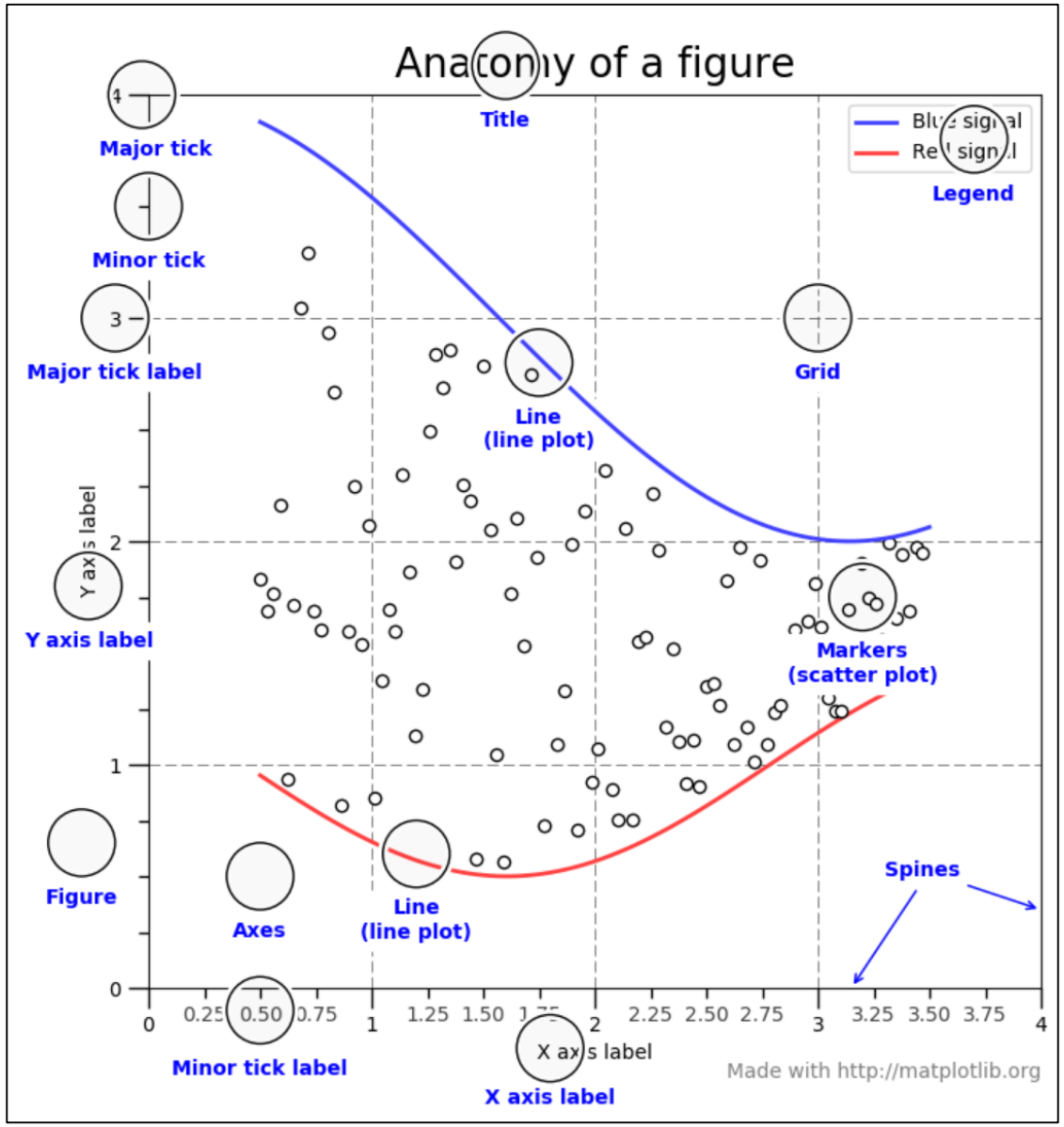
Los arrays en *Numpy* tienen las siguientes características:

- Son estáticos y homogéneos.

- Son más eficientes en el uso y la gestión de memoria.

- La funciones están implementadas en lenguajes más cercanos al hardware, como C o Fortran, para que se ejecuten de una forma más eficiente.

- Partes de una figura



- La parte *Figure* hace referencia a la ventana.
- La parte *Axes* se refiere a la figura en sí, es decir, la zona donde se dibujan todos los objetos del gráfico.

- Interfaces de Matplotlib

A la hora de trabajar con Matplotlib tenemos dos formas de hacerlo:

- *U*sando el estilo de Matlab.

Matplotlib permite a los usuarios realizar el trabajo con figuras de una forma muy similar a como se hace con Matlab. Esta forma de trabajar con Matplotlib permite a aquellos usuarios que hayan utilizado Matlab con anterioridad realizar sus gráficas en Python apenas sin esfuerzo.

- Usando la interfaz de Orientación a Objetos.

Matplotlib también permite realizar gráficas utilizando programación dirigida a objetos, es decir, tratándolas como objetos, y actuando sobre cada objeto de forma independiente a través de sus atributos y métodos.

- El método recomendado es el segundo ya que es mucho más potente e intuitivo.

- El módulo *pyplot*

La mayor parte del ploteado de gráficos en Python se suele hacer a través de las funciones incorporadas en el módulo *pyplot* del paquete *Matplotlib*.

Para usar *pyplot* simplemente tendremos que importarlo:

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

Pyplot en modo interactivo

La figura generada se muestra y actualiza automáticamente si hacemos cambios en la misma. Esto es así debido a que las funciones incluidas en pyplot incorporan en su código una llamada a la función *draw_if_interactive().*

Pyplot en modo no interactivo

La figura generada no se muestra en pantalla hasta que utilicemos la función *show()* de pyplot, que la dibujará y además interrumpirá la ejecución hasta que la ventana sea cerrada.

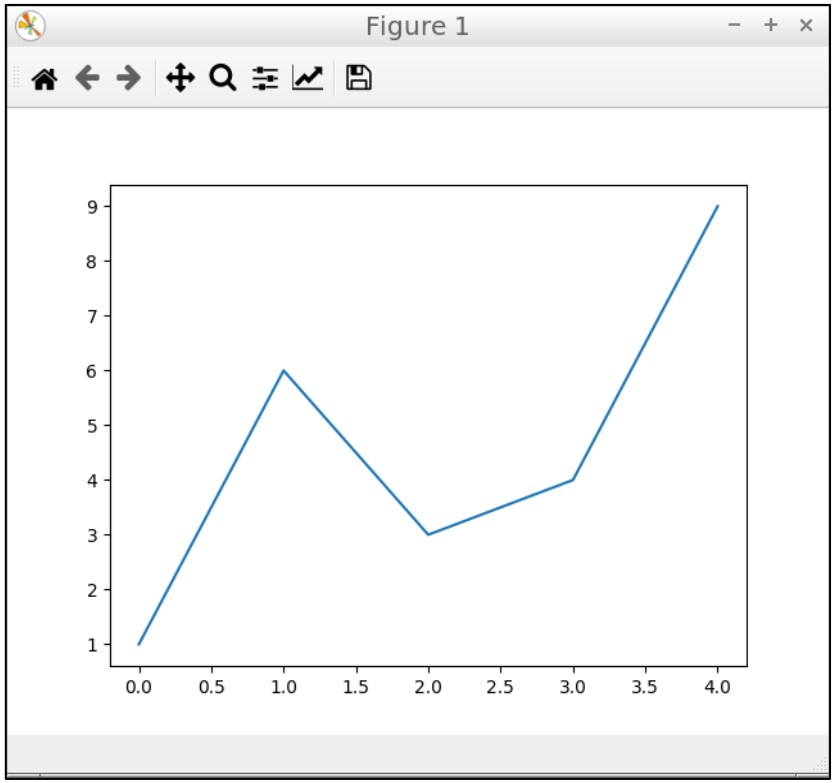
- Ploteado Básico

- Los datos a plotear deberán estar en un array o matriz.

- La función más básica de pyplot que nos permite plotear es *plot().*

- Si estamos en modo interactivo, la salida de *plot()* será una ventana con la figura correspondiente. En caso contrario, para mostrar dicha ventana necesitaremos llamar a la función *show()*.

```
>>> import matplotlib.pyplot as plt
>>> plt.plot([1,6,3,4,9])
[<matplotlib.lines.Line2D object at 0x7f76fc13b860>]
>>> plt.show()
```



- Creación de varias figuras

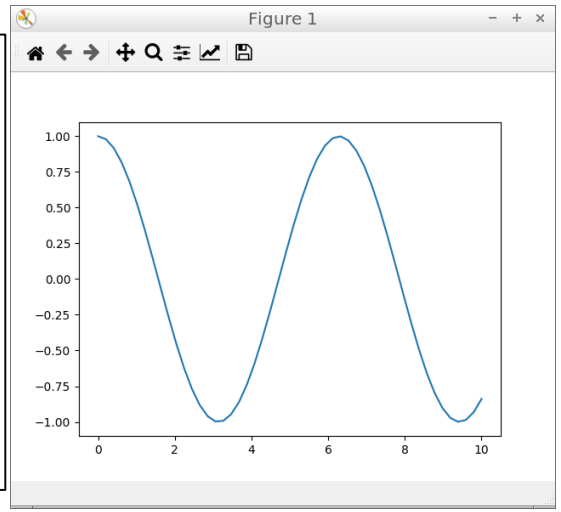
Por defecto, Matpotlib plotea siempre sobre la ventana o figura activa. Si necesitamos crear nuevas ventanas adicionales, lo haremos con la función:

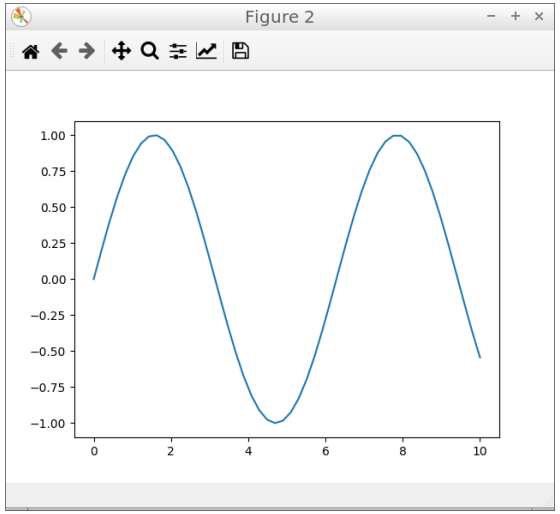
plt.figure(identificador_ventana)

- *Identificador_ventana* será el identificador de la figura, que podrá ser un número entero o un string.

Si la ventana con la etiqueta ya existe, entonces la pone como activa y por tanto, las siguientes instrucciones se aplicarán a ella.

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 x = np.linspace(0,10,50)
5
6 plt.ion()
7
8 fig1 = plt.figure(1)
9 fig2 = plt.figure(2)
10 plt.plot(x, np.sin(x))  # esto se dibuja en la figura 2
11 fig1 = plt.figure(1)    # como fig1 ya existe, se pone como activa esta figura
12 plt.plot(x, np.cos(x))  # esto se dibuja en la figura 1
```





- Cerrar una figura o ventana gráfica

Para cerrar una figura o ventana utilizamos la función *close*() de pyplot:

plt.close(): Cierra la figura activa
plt.close(identificador_figura): Cierra la figura especificada por identificador_figura
plt.close('all'): Cierra todas las ventanas

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 x = np.linspace(0,10,50)
5
6 plt.ion()
7
8 fig1 = plt.figure(1)
9 fig2 = plt.figure(2)
10 plt.plot(x, np.sin(x))  # esto se dibuja en la figura 2
11 fig1 = plt.figure(1)    # como fig1 ya existe, se pone como activa esta figura
12 plt.plot(x, np.cos(x))  # esto se dibuja en la figura 1
13 plt.close(fig = 2)      # cierra la figura 2
```

- Utilizando estilos (i)

https://matplotlib.org/3.1.1/gallery/style_sheets/style_sheets_reference.html

Matplotlib incorpora por defecto una serie de estilos que nos permiten plotear nuestras figuras con diferentes estéticas.

Podemos mostrar los estilos disponibles: *plt.style.available*

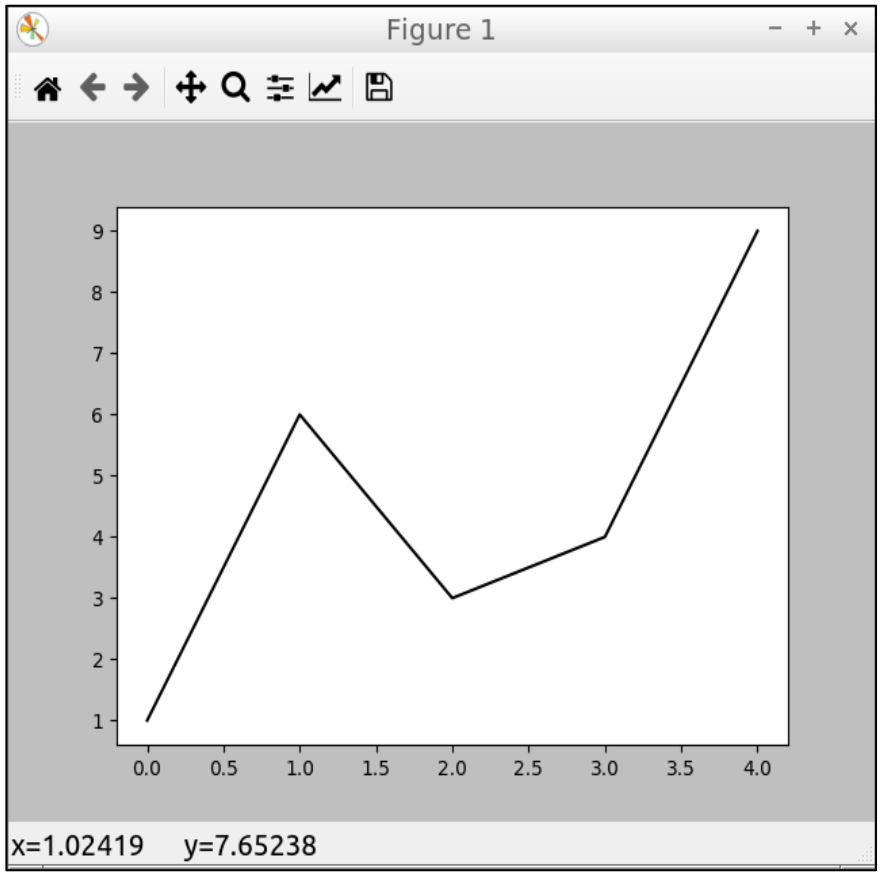
```
>>> plt.style.available
['seaborn-white', 'seaborn-muted', 'seaborn-deep', 'seaborn-notebook', '_classic_test', 'seaborn-talk', 'grayscale', 'seaborn-dark-
palette', 'seaborn', 'fast', 'seaborn-colorblind', 'seaborn-ticks', 'seaborn-paper', 'dark_background', 'ggplot', 'Solarize_Light2'
, 'seaborn-pastel', 'seaborn-dark', 'seaborn-whitegrid', 'classic', 'tableau-colorblind10', 'seaborn-poster', 'seaborn-darkgrid', '
seaborn-bright', 'fivethirtyeight', 'bmh']
```

- Utilizando estilos (ii)

- Establecer o activar un estilo temporalmente:

with plt.style.context('nombre_de_estilo')
plt.plot(x,y)

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

In [2]: with plt.style.context('grayscale'):
   ...:     plt.plot([1,6,3,4,9])
   ...:
```

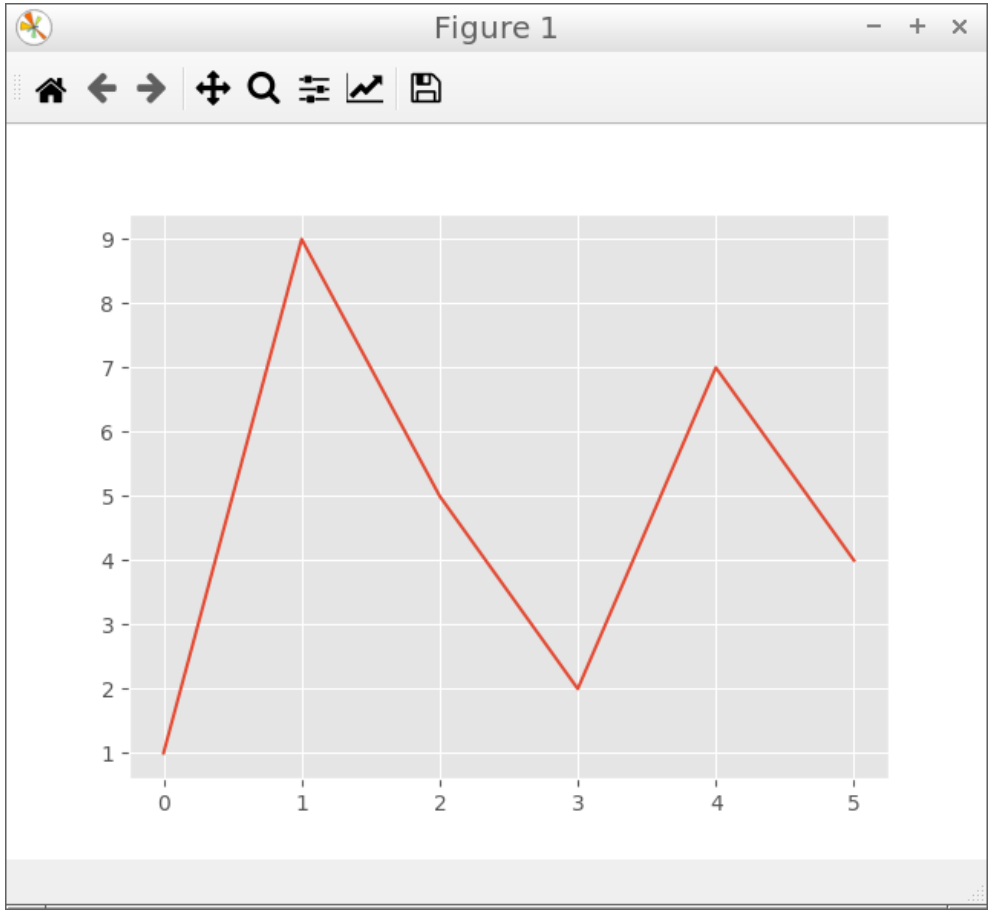


- Establecer o activar un estilo para toda la sesión: *plt.style.use('nombre_de_estilo')*

- Restablecer la Runtime Configuration (rc): *plt.rcdefaults()*

Ejercicio 1: Realizar un script que muestre una gráfica de líneas con los datos 1,9,5,2,7,4 usando el estilo *ggplot*. Después restablecer los ajustes por defecto.

```
 8 import matplotlib.pyplot as plt
 9 datos = [1,9,5,2,7,4]
10 plt.style.use('ggplot')
11 plt.plot(datos)
12 plt.show()
13 plt.rcdefaults()
```



Ejercicio 2: Realizar un script que muestre una figura con los diferentes estilos disponibles en Matplotlib.

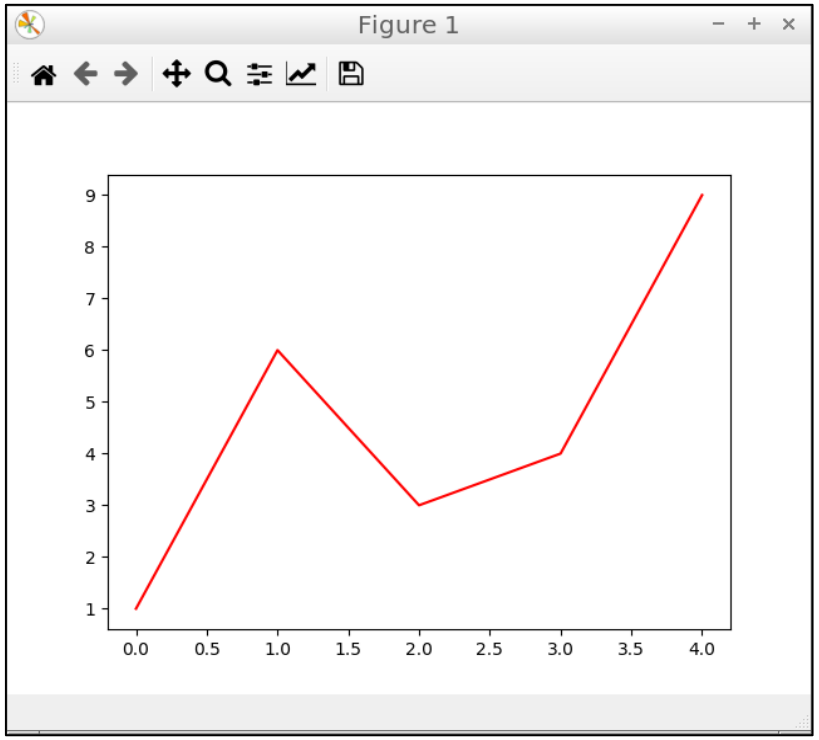
```python
import matplotlib.pyplot as plt
import numpy as np

plt.ion()   # activa el modo interactivo

x = np.linspace(0,1,11)
y = np.random.randint(0,100,11)

estilos = plt.style.available

for e in estilos:
    with plt.style.context(e):
        plt.figure(e)
        plt.plot(x,y)

tecla = input("Pulsa una tecla para continuar ...")
plt.close('all')    # cierra todas las ventanas abiertas
plt.rcdefaults()    # restablezco configuración por defecto
```

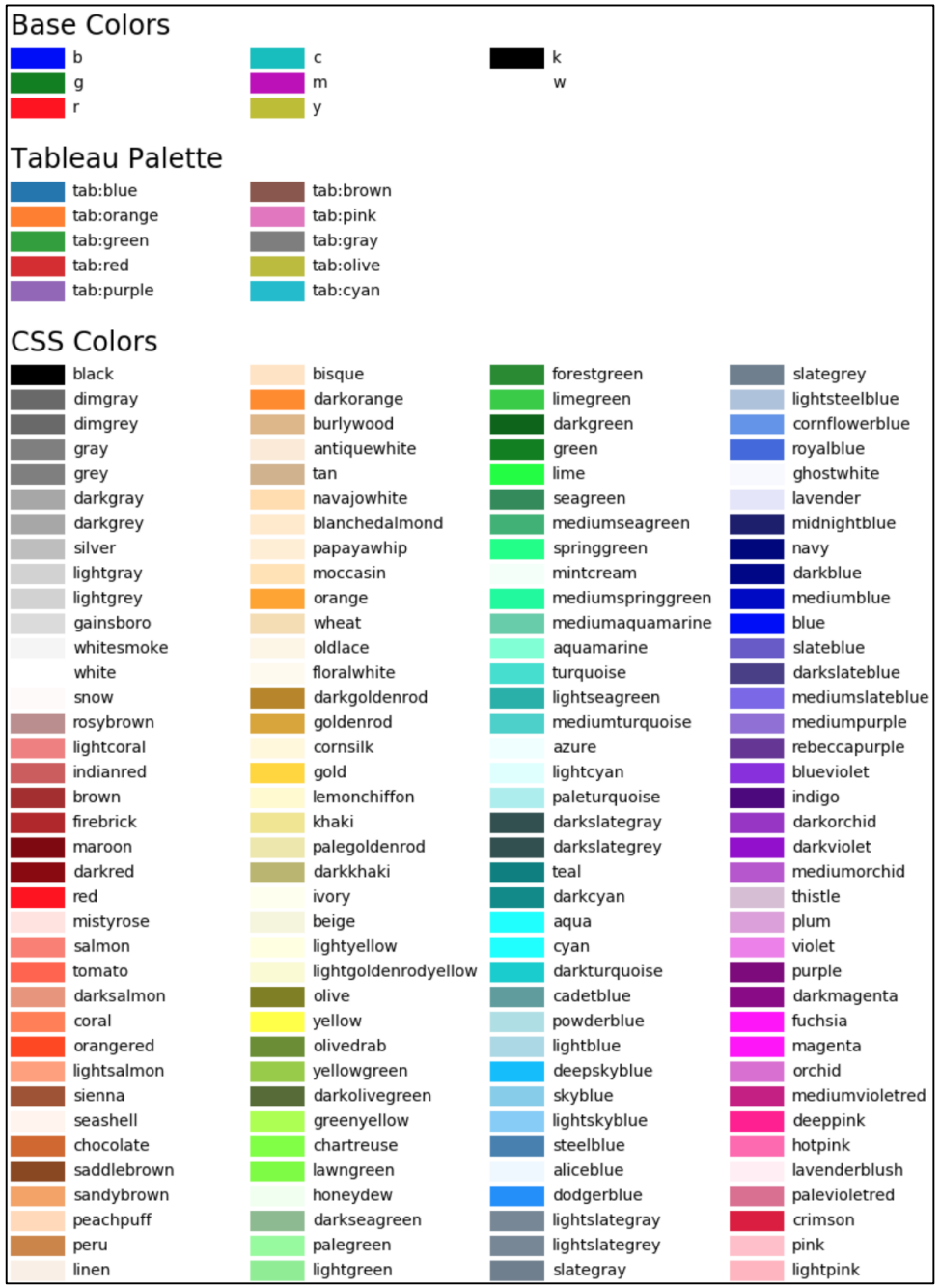
```python
import matplotlib.pyplot as plt
import numpy as np

plt.ion()
plt.rcParams.update({'figure.max_open_warning': 0})  # evita el warning del sistema cuando abrimos más de 20 figuras

x = np.linspace(0,1,11)
y = np.random.randint(0,100,11)

estilos = sorted(plt.style.available)

for e in estilos:
    plt.style.use(e)
    plt.figure(e)
    plt.plot(x,y)
tecla = input("Pulsa una tecla para continuar ...")
plt.close('all')
plt.rcdefaults()
```

- Personalizando el color de las líneas

Podemos establecer el color de la línea a través del parámetro *color* al llamar a la función *plot()*.

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 x = np.arange(0,5,1)
5 y = np.array([1,6,3,4,9])
6 plt.plot(x, y, color = 'red')
7 plt.show()
```





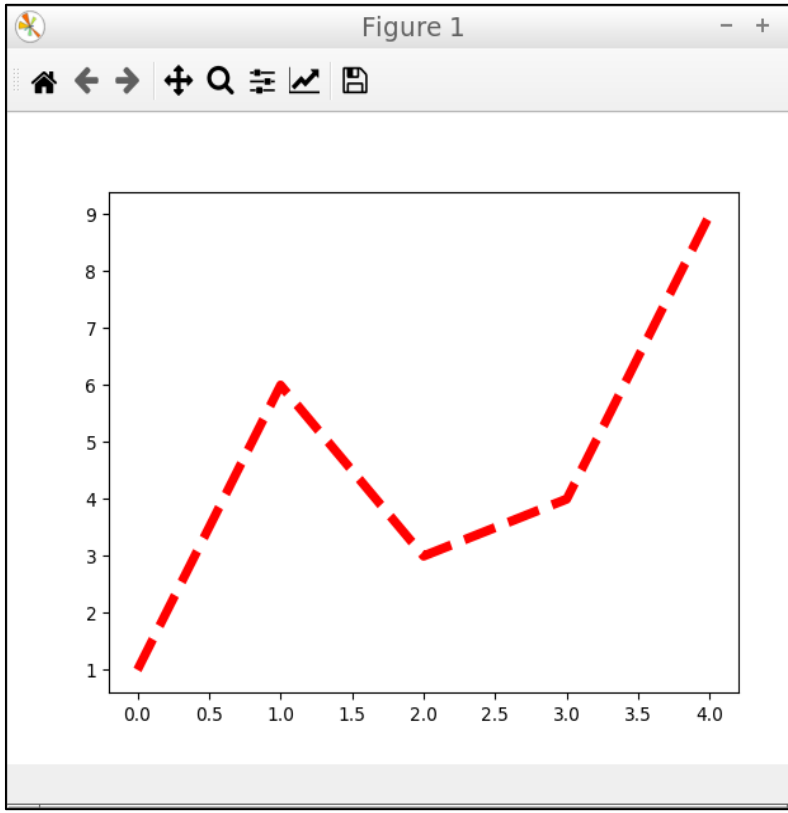
- Personalizando el estilo y el grosor de las líneas

Linestyle	Description
'-' or 'solid'	solid line
'--' or 'dashed'	dashed line
'-.' or 'dashdot'	dash-dotted line
':' or 'dotted'	dotted line
'None' or ' ' or ''	draw nothing

- Podemos establecer el tipo de línea a través del parámetro *linestyle* al llamar a la función plot.

- Podemos establecer el grosor de la línea a través del parámetro *linewidth* al llamar a la función plot.

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 x = np.arange(0,5,1)
5 y = np.array([1,6,3,4,9])
6 plt.plot(x, y, color = 'r', linestyle = '--', linewidth =  5)
7 plt.show()
```

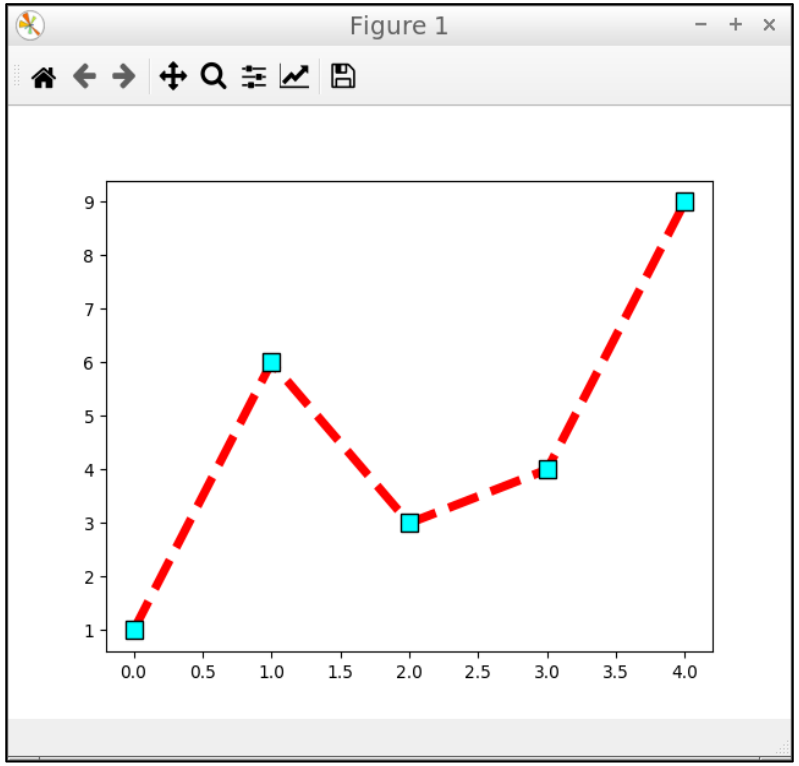


- Marcadores de los datos (i)

Podemos establecer marcadores de los datos en la línea a través del parámetro *marker* al llamar a la función *plot().* También podemos indicar su tamaño con *markersize*, así como su color de relleno y de borde con *markerfacecolor* y *markeredgecolor*

marker	symbol	description
"."		point
","		pixel
"o"		circle
"v"		triangle_down
"^"		triangle_up
"<"		triangle_left
">"		triangle_right
"1"		tri_down
"2"		tri_up
"3"		tri_left
"4"		tri_right
"8"		octagon
"s"		square
"p"		pentagon
"P"		plus (filled)
"*"		star
"h"		hexagon1
"H"		hexagon2
"+"		plus
"x"		x
"X"		x (filled)
"D"		diamond
"d"		thin_diamond
"|"		vline
"_"		hline

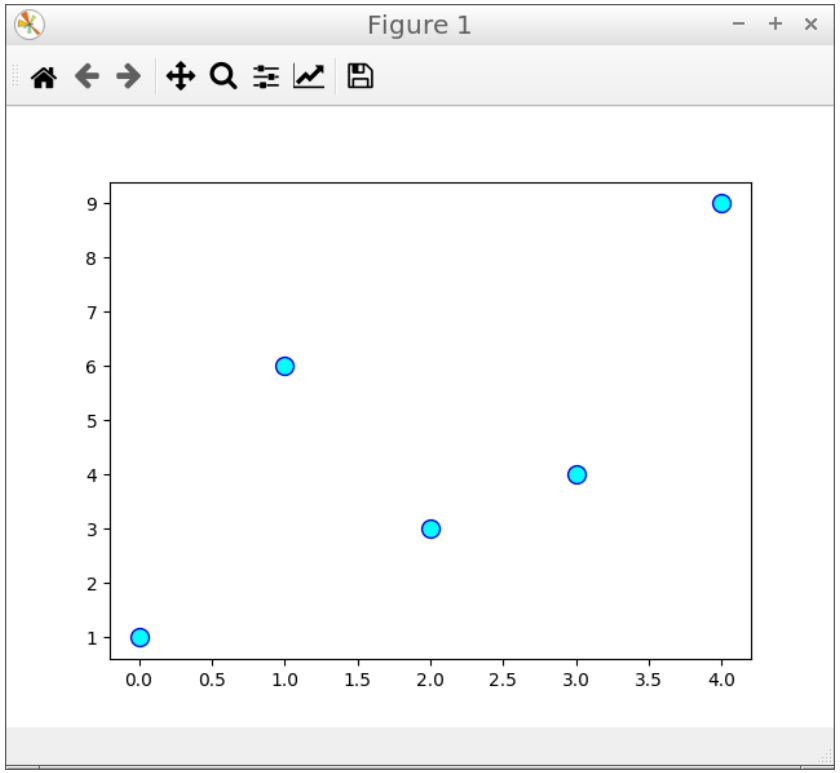
```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 x = np.arange(0,5,1)
5 y = np.array([1,6,3,4,9])
6 plt.plot(x, y, color = 'r', linestyle = '--', linewidth =  5, marker = 's', markersize = 10, markerfacecolor = 'cyan', markeredgecolor = 'black')
7 plt.show()
```



- Marcadores de los datos (ii)

Podemos pintar únicamente los marcadores, sin unirlos mediante líneas a través del atributo *linestyle* de alguna de las siguientes maneras:

linestyle = ' None'
linestyle = ' '
linestyle = ' '




```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 x = np.arange(0, 5, 1)
5 y = np.array([1, 6, 3, 4, 9])
6
7 plt.plot(x, y, linestyle = 'None', marker = 'o', markersize = 10, markerfacecolor = 'cyan', markeredgecolor = 'b')
8 plt.show()
```

- Propiedades del texto

Por norma general, los gráficos también incorporan elementos de texto o etiquetas de texto como el título, los rótulos de los ejes, la leyenda, etc.

Podemos personalizar todos esos elementos que se basan en texto cambiando algunas de sus propiedades o atributos:

- *color = nombre_color*
- *fontfamily o family = nombre_familia_fuente*

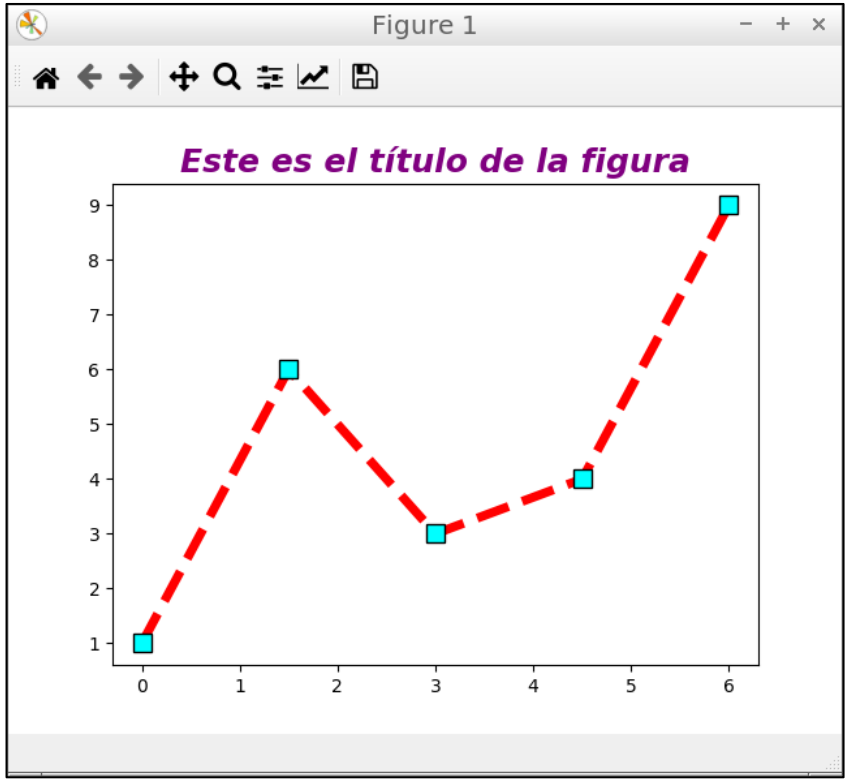
  Tenemos las siguientes familias genéricas: *‘serif’, ‘sans-serif’, ‘cursive’, ‘fantasy’, y ‘monospace’*

  https://matplotlib.org/3.1.1/api/font_manager_api.html#matplotlib.font_manager.FontProperties
- *fontsize o size = size in points, 'xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large'}*
- *fontstyle or style = {'normal', 'italic', 'oblique'}*
- *fontweight or weight = {a numeric value in range 0-1000, 'ultralight', 'light', 'normal', 'regular', 'book', 'medium', 'roman', 'semibold', 'demibold', 'demi', 'bold', 'heavy', 'extra bold', 'black'}*
- *rotation = {angle in degrees, 'vertical', 'horizontal'}*
- *horizontalalignment or ha = {'center', 'right', 'left'}*
- *verticalalignment or va = {'center', 'top', 'bottom', 'baseline', 'center_baseline'}*

- Poner título a la figura

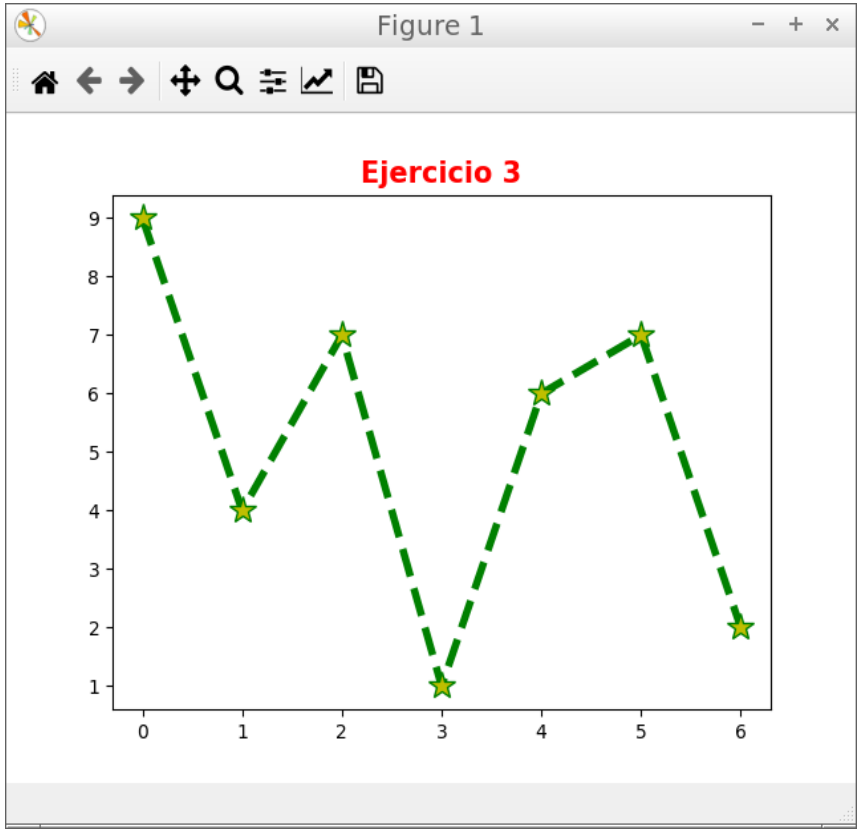
Para establecer el título de nuestra gráfica podremos utilizar la función *title(titulo, atributos_formato).*

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 x = np.arange(0,5,1)
5 y = np.array([1,6,3,4,9])
6 plt.plot(x * 1.5, y, color = 'r', linestyle = '--', linewidth =  5, marker = 's', markersize = 10, markerfacecolor = 'cyan', markeredgecolor = 'black')
7 plt.title('Este es el título de la figura', color = 'purple', fontsize = 18, style = 'italic', weight = 'bold')
8 plt.show()
```



Ejercicio 3: Crear un gráfico de líneas con los datos 9,4,7,1,6,7,2. Poner la línea discontinua, con color verde y con un grosor de tamaño 4. Además, poner estrellas amarillas de tamaño 15 como marcadores de los datos ploteados. Finalmente añadir un título a la figura en negrita, color rojo y tamaño de fuente 15.

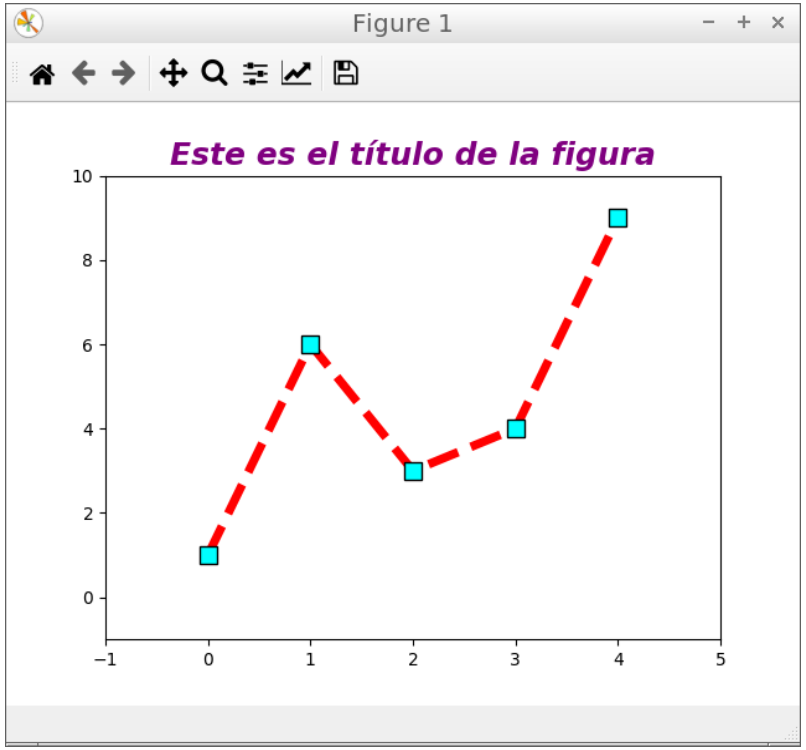
```
 9 import matplotlib.pyplot as plt
10 datos = [9,4,7,1,6,7,2]
11
12 plt.plot(datos, linestyle = "--", color = 'g', linewidth = 4, marker = '*', markersize = 15, markerfacecolor = 'y')
13 plt.title('Ejercicio 3', fontsize = 15, color = 'r', weight = 'bold')
14 plt.show()
```



- Personalizando los ejes (i)

Podemos establecer los límites de los ejes con las funciones *xlim(menor,mayor)* y *ylim(menor,mayor)*.

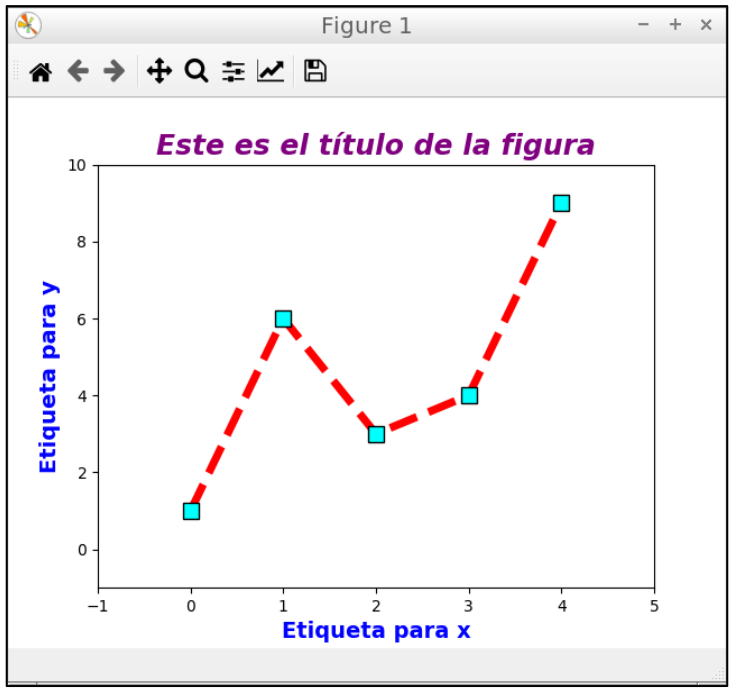
```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 x = np.arange(0,5,1)
5 y = np.array([1,6,3,4,9])
6 plt.plot(x, y, color = 'r', linestyle = '--', linewidth =  5, marker = 's', markersize = 10, markerfacecolor = 'cyan', markeredgecolor = 'black')
7 plt.title('Este es el título de la figura', color = 'purple', fontsize = 18, style = 'italic', weight = 'bold')
8 plt.xlim(-1,5)
9 plt.ylim(-1,10)
10 plt.show()
```



- Personalizando los ejes (ii)

Podemos poner etiquetas a los ejes a través de las funciones *xlabel()* e *ylabel()*.

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 x = np.arange(0,5,1)
5 y = np.array([1,6,3,4,9])
6 plt.plot(x, y, color = 'r', linestyle = '--', linewidth =  5, marker = 's', markersize = 10, markerfacecolor = 'cyan', markeredgecolor = 'black')
7 plt.title('Este es el título de la figura', color = 'purple', fontsize = 18, style = 'italic', weight = 'bold')
8 plt.xlim(-1,5)
9 plt.ylim(-1,10)
10 plt.xlabel('Etiqueta para x', size = 14, color = 'blue', weight = 'semibold')
11 plt.ylabel('Etiqueta para y', size = 14, color = 'blue', weight = 'semibold')
12 plt.show()
```

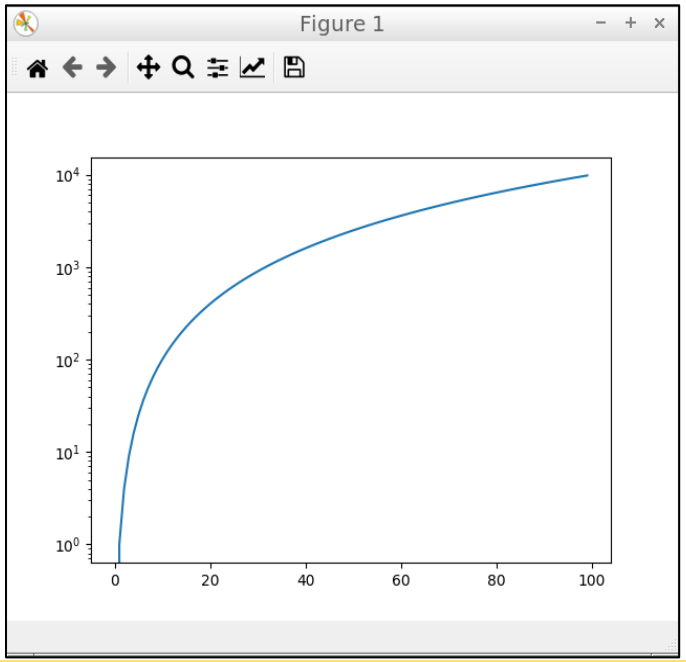


- Personalizando los ejes (iii)

Podemos establecer una de las siguientes escalas con las funciones de pyplot *yscale()* y *xscale():*

- *linear:* escala lineal. Es la escala por defecto.
- *log:* escala logarítmica. Esta escala sólo acepta valores positivos. Los valores negativos se pueden tratar de dos maneras: *mask* = ignorarlos (valor por defecto)
  c*lip* = convertirlos a un valor muy pequeño próximo a 0.
- *symlog (*symmetrical log). Acepta tanto valores positivos como negativos.
- *logit:* escala que se usa para datos comprendidos entre 0 y 1 ambos excluidos. Los datos fuera de dicho rango se tratarán igual que en la escala logarítmica *(log)* indicando *mask* o *clip.*

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 x = np.arange(100)
5 y = x ** 2
6 plt.plot(x,y)
7 plt.yscale('log')
8 plt.show()
```

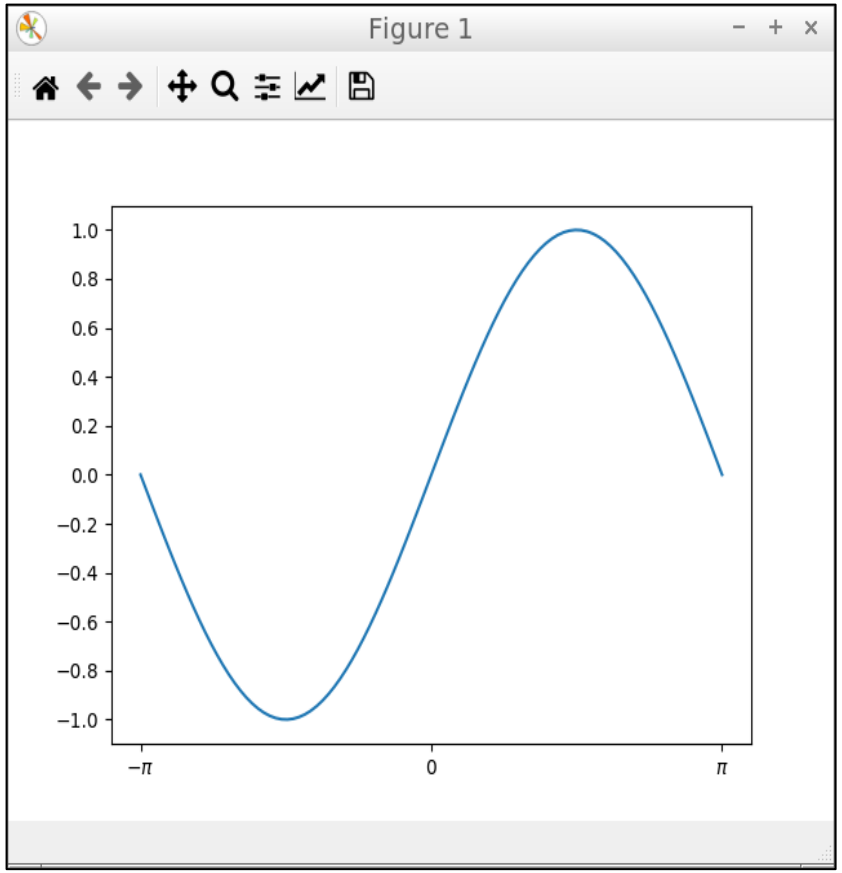


- Personalizando los ejes (iv)

Podemos cambiar las marcas (ticks) en los ejes *x* e *y* con *pyplot* usando las funciones *xticks() e yticks()*, cuyo formato es:

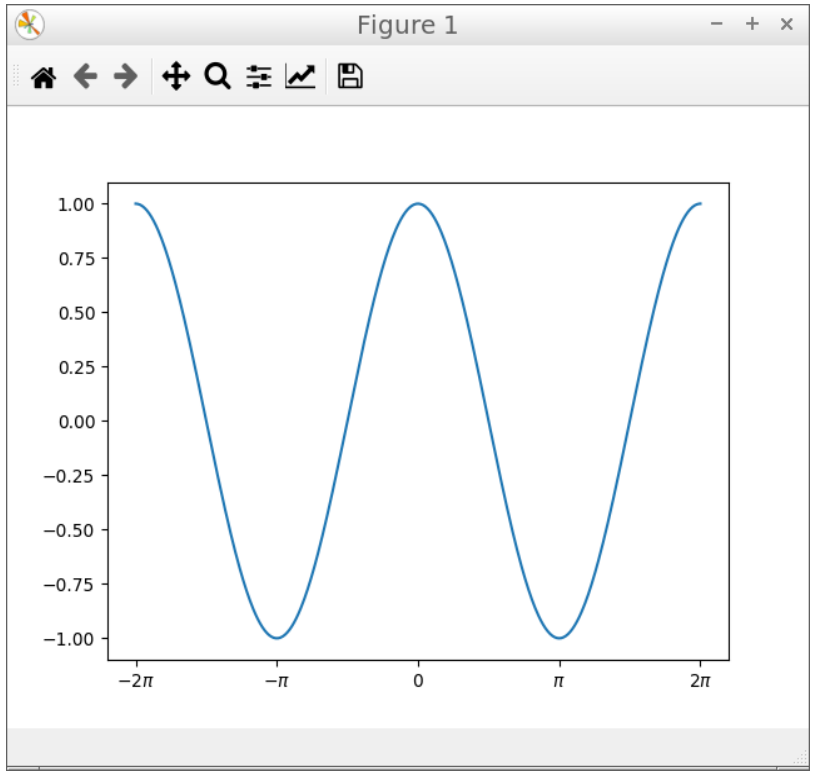
xticks(valores, etiquetas)
yticks(valores, etiquetas)

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4
5 x = np.linspace(-np.pi, np.pi, 100)
6 y = np.sin(x)
7 plt.plot(x,y)
8 plt.yticks(np.arange(-1, 1.1, 0.2))
9 plt.xticks([-np.pi, 0, np.pi],["$-\pi$", '0', '$\pi$'])
10 plt.show()
```



Ejercicio 4: Crear un gráfico de líneas para plotear la función cos(x) entre -2pi y 2pi. Establecer las correspondientes marcas para los ejes x e y, además de ponerles etiquetas.

```
 8 import matplotlib.pyplot as plt
 9 import numpy as np
10
11 x = np.linspace((-2 * np.pi), (2 * np.pi), 500)
12 y = np.cos(x)
13
14 plt.plot(x, y)
15 plt.xticks([(-2 * np.pi), -np.pi, 0, np.pi, (2 * np.pi)], ["$-2\pi$", "$-\pi$", "0", "$\pi$", "$2\pi$"])
16 plt.show()
```

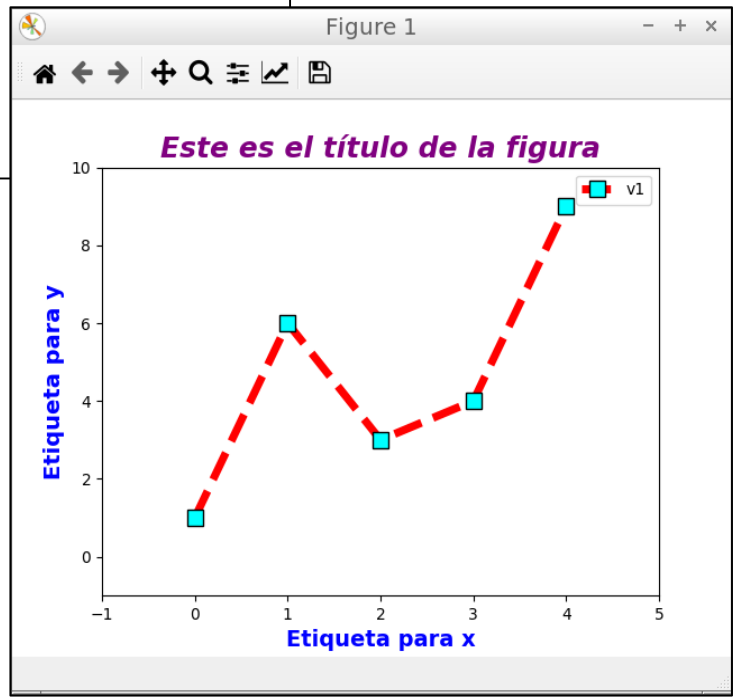


- Mostrar la leyenda

Por defecto, la función *legend()* automáticamente crea una leyenda con las etiquetas de los elementos ploteados.

Si vamos a mostrar leyenda en nuestra gráfica es conveniente poner una etiqueta a cada instrucción *plot(),* ya que en la leyenda se mostrará dicha etiqueta.

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 x = np.arange(0,5,1)
5 y = np.array([1,6,3,4,9])
6 plt.plot(x, y, label = 'v1', color = 'r', linestyle = '--', linewidth =  5, marker = 's', markersize = 10, markerfacecolor = 'cyan', markeredgecolor = 'black')
7 plt.title('Este es el título de la figura', color = 'purple', fontsize = 18, style = 'italic', weight = 'bold')
8 plt.xlim(-1,5)
9 plt.ylim(-1,10)
10 plt.xlabel('Etiqueta para x', size = 14, color = 'blue', weight = 'semibold')
11 plt.ylabel('Etiqueta para y', size = 14, color = 'blue', weight = 'semibold')
12 plt.legend()
13 plt.show()
```



- Personalizar la leyenda (i)

- Podemos establecer la ubicación de la leyenda al lugar más apropiado de nuestra gráfica, con el parámetro *loc* de la función *legend().*

The location of the legend. Possible codes are:

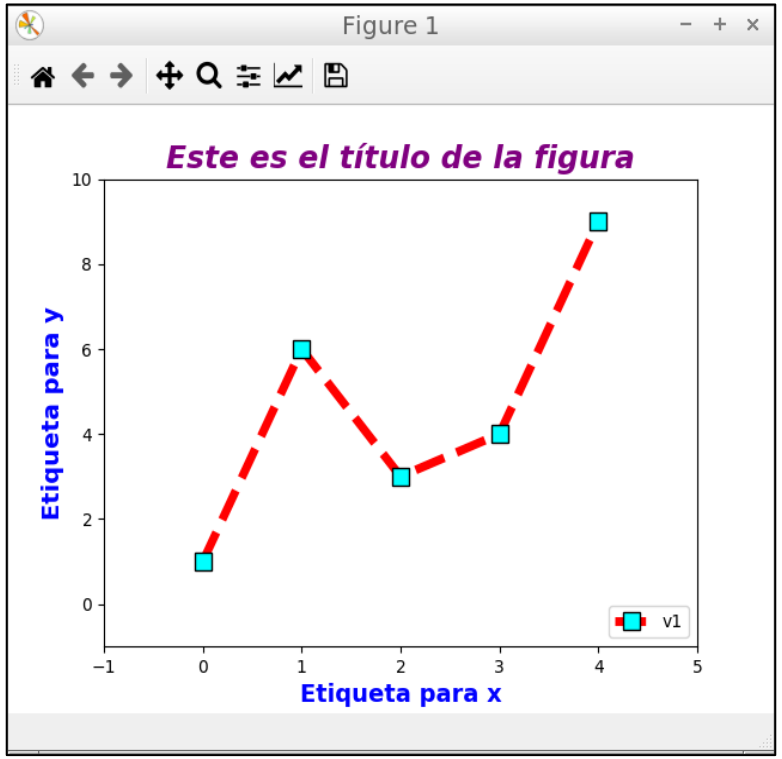
Location String	Location Code
‘best’	0
‘upper right’	1
‘upper left’	2
‘lower left’	3
‘lower right’	4
‘right’	5
‘center left’	6
‘center right’	7
‘lower center’	8
‘upper center’	9
‘center’	10

- También podemos modificar el tamaño de la fuente con el parámetro *fontsize,* indicando un valor numérico para el tamaño en puntos o uno de los siguientes valores:

fontsize : int or float or {‘xx-small’, ‘x-small’, ‘small’, ‘medium’, ‘large’, ‘x-large’, ‘xx-large’}

- Personalizar la leyenda (ii)

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 x = np.arange(0,5,1)
5 y = np.array([1,6,3,4,9])
6 plt.plot(x, y, label = 'v1', color = 'r', linestyle = '--', linewidth =  5, marker = 's', markersize = 10, markerfacecolor = 'cyan', markeredgecolor = 'black')
7 plt.title('Este es el título de la figura', color = 'purple', fontsize = 18, style = 'italic', weight = 'bold')
8 plt.xlim(-1,5)
9 plt.ylim(-1,10)
10 plt.xlabel('Etiqueta para x', size = 14, color = 'blue', weight = 'semibold')
11 plt.ylabel('Etiqueta para y', size = 14, color = 'blue', weight = 'semibold')
12 plt.legend(loc = 4, fontsize = 'medium')
13 plt.show()
```

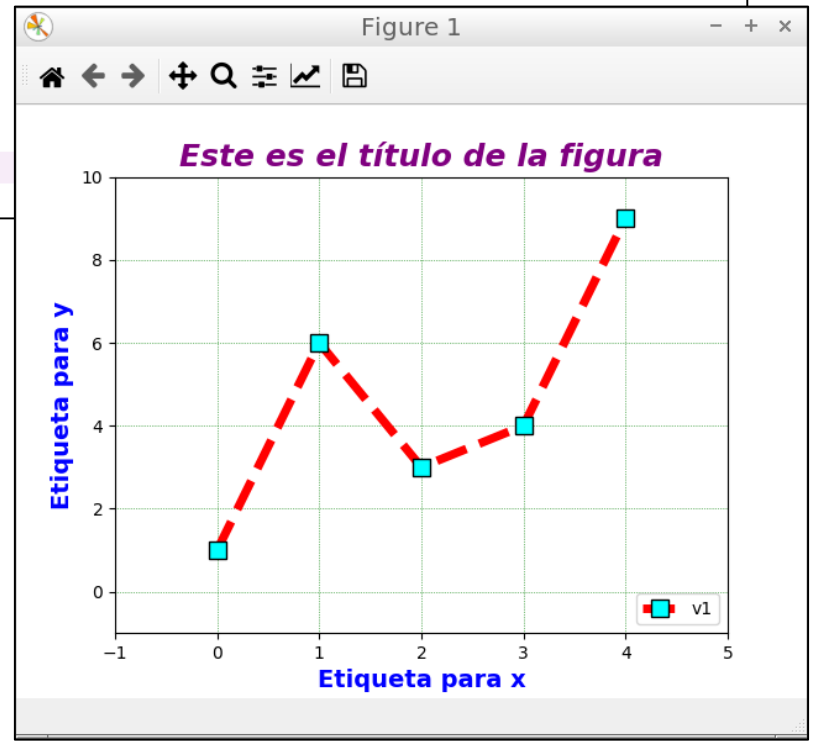


- Grid

Por defecto, Matplotlib no dibuja el grid de la figura. Podemos mostrarlo con la función *grid().*

Algunos de sus parámetros son *color, linestyle* y *linewidth*, que nos permiten establecer el color, el estilo y el grosor de las líneas del grid respectivamente.

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 x = np.arange(0,5,1)
5 y = np.array([1,6,3,4,9])
6 plt.plot(x, y, label = 'v1', color = 'r', linestyle = '--', linewidth =  5, marker = 's', markersize = 10, markerfacecolor = 'cyan', markeredgecolor = 'black')
7 plt.title('Este es el título de la figura', color = 'purple', fontsize = 18, style = 'italic', weight = 'bold')
8 plt.xlim(-1,5)
9 plt.ylim(-1,10)
10 plt.xlabel('Etiqueta para x', size = 14, color = 'blue', weight = 'semibold')
11 plt.ylabel('Etiqueta para y', size = 14, color = 'blue', weight = 'semibold')
12 plt.legend(loc = 4, fontsize = 'medium')
13 plt.grid(color='green', linestyle=':', linewidth=0.5)
14 plt.show()
```

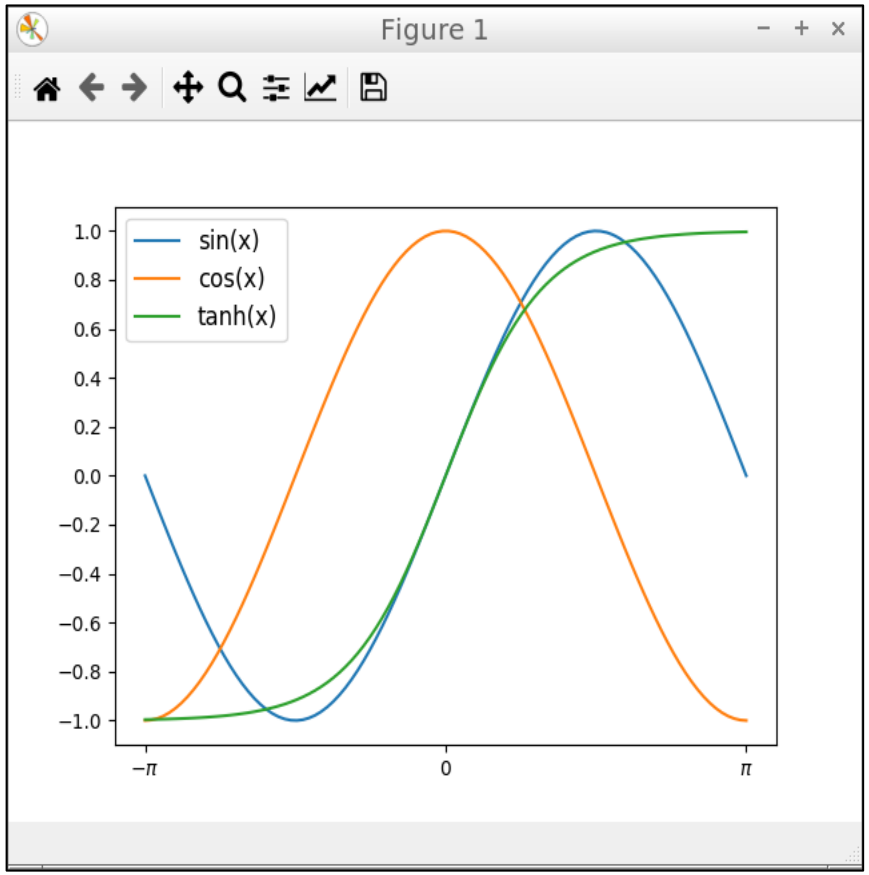


- Una gráfica con múltiples líneas

Podemos plotear varias funciones a la vez en la misma zona (*Axes)*.

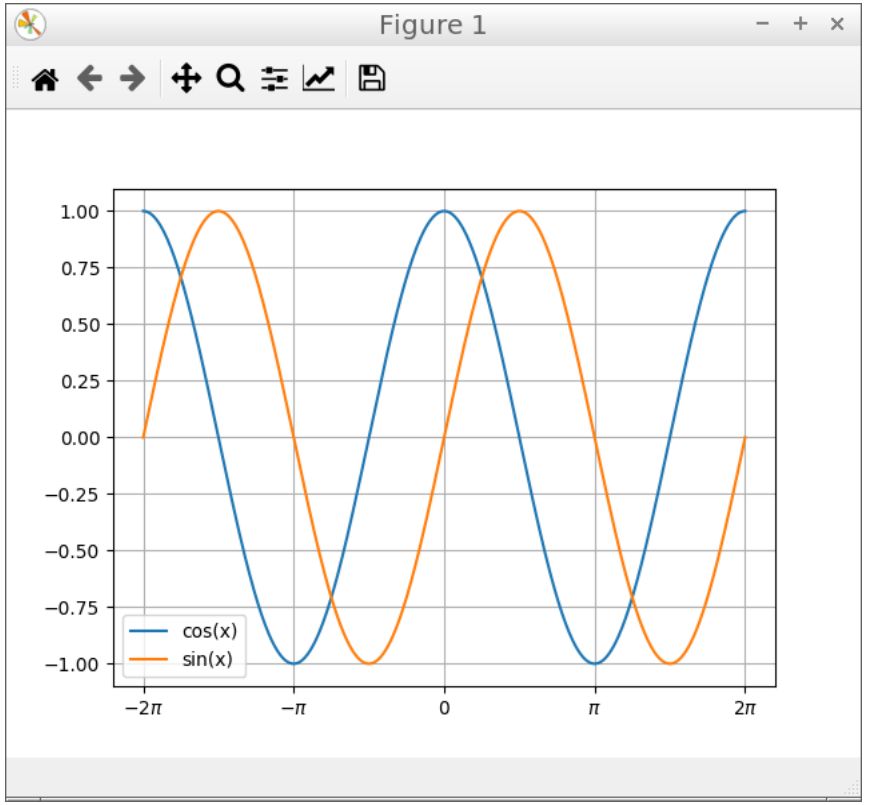
```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(-np.pi, np.pi, 100)
s = np.sin(x)
plt.plot(x, s, label = 'sin(x)')
c = np.cos(x)
plt.plot(x, c, label = 'cos(x)')
t = np.tanh(x)
plt.plot(x, t, label = 'tanh(x)')
plt.xticks([-np.pi, 0, np.pi],["$-\pi$", '0', '$\pi$'])
plt.yticks(np.arange(-1, 1.1, 0.2))
plt.legend(fontsize = 12)
plt.show()
```

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

x = np.linspace(-np.pi, np.pi, 100)
s = np.sin(x)
c = np.cos(x)
t = np.tanh(x)
plt.plot(x, s, x, c, x, t)
plt.legend(fontsize = 12, labels = ['sin(x)', 'cos(x)', 'tanh(x)'])
plt.xticks([-np.pi, 0, np.pi],["$-\pi$", '0', '$\pi$'])
plt.yticks(np.arange(-1, 1.1, 0.2))
plt.show()
```



Ejercicio 5: En la misma gráfica del ejercicio 4 plotear también la función sin(x) en el mismo rango. Añadir al gráfico un grid y también una leyenda en la posición que más os guste.

```
 9 import matplotlib.pyplot as plt
10 import numpy as np
11
12 x = np.linspace((-2 * np.pi), (2 * np.pi), 500)
13 y = np.cos(x)
14 z = np.sin(x)
15
16 plt.plot(x, y, label = 'cos(x)')
17 plt.plot(x, z, label = 'sin(x)')
18 plt.xticks([(-2 * np.pi), -np.pi, 0, np.pi, (2 * np.pi)], ["$-2\pi$", "$-\pi$", "0", "$\pi$", "$2\pi$"])
19 plt.legend(loc = 3)
20 plt.grid()
21
22 plt.show()
```



- Matplotlib Orientado a Objetos

Hasta ahora hemos utilizado Matplotlib de una forma similar a como *Matlab* se comparta a la hora de realizar gráficas. No obstante, podemos ir un paso más allá, aprovechando que Matplotlib nos permite tratar como objetos las diferentes zonas que componen una figura, es decir, podemos utilizar programación dirigida a objetos con Matplotlib.

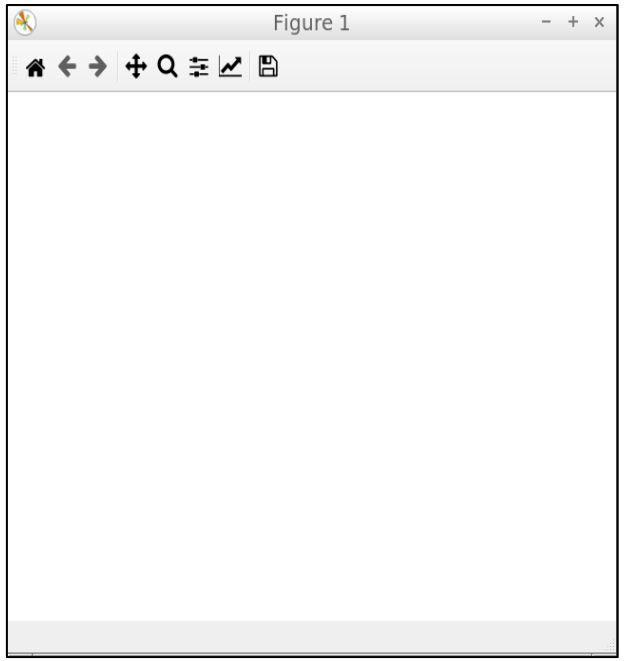
De mayor a menor nivel tenemos la siguiente jerarquía de objetos:

Object	Description
`FigureCanvas`	Container class for the `Figure` instance
`Figure`	Container for one or more `Axes` instances
`Axes`	The rectangular areas to hold the basic elements, such as lines, text, and so on

- Creación de un objeto *Figure y un objeto Axes*

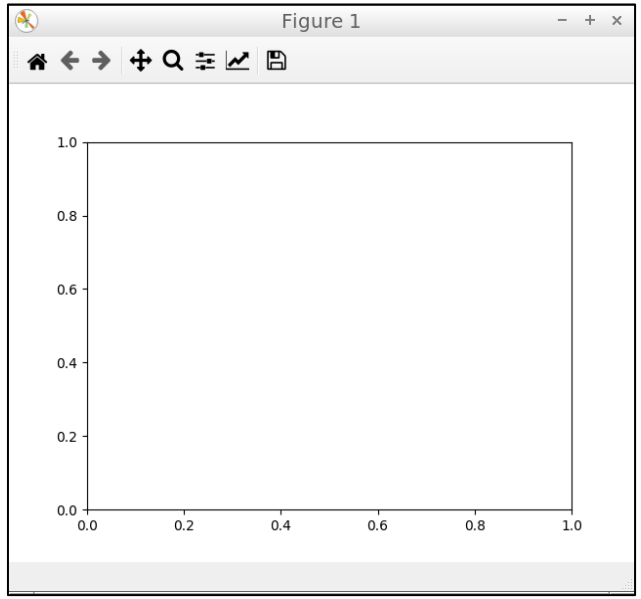
- Para crear una instancia de un objeto *Figure* usaremos la función *figure()* de *Pyplot.*

```
1 import matplotlib.pyplot as plt
2
3 plt.ion()
4 fig1 = plt.figure()
```



- Para crear una instancia de un objeto *Axes* usaremos la función *axes()* de *Pyplot.*

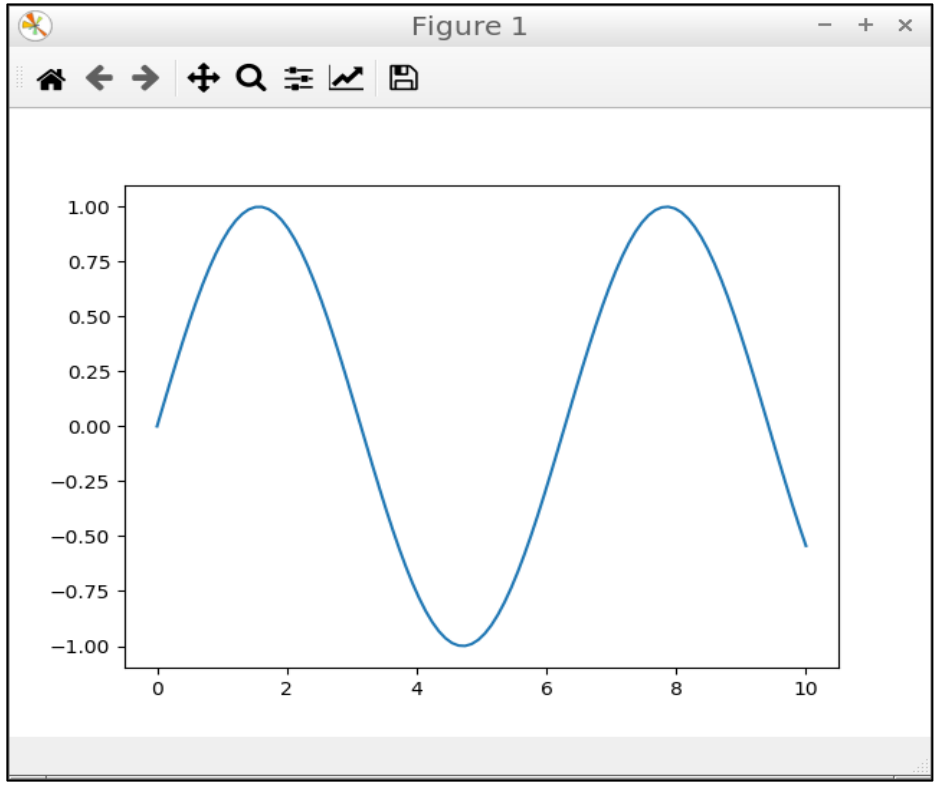
```
1 import matplotlib.pyplot as plt
2
3 plt.ion()
4 fig1 = plt.figure()
5 ax1 = plt.axes()
```



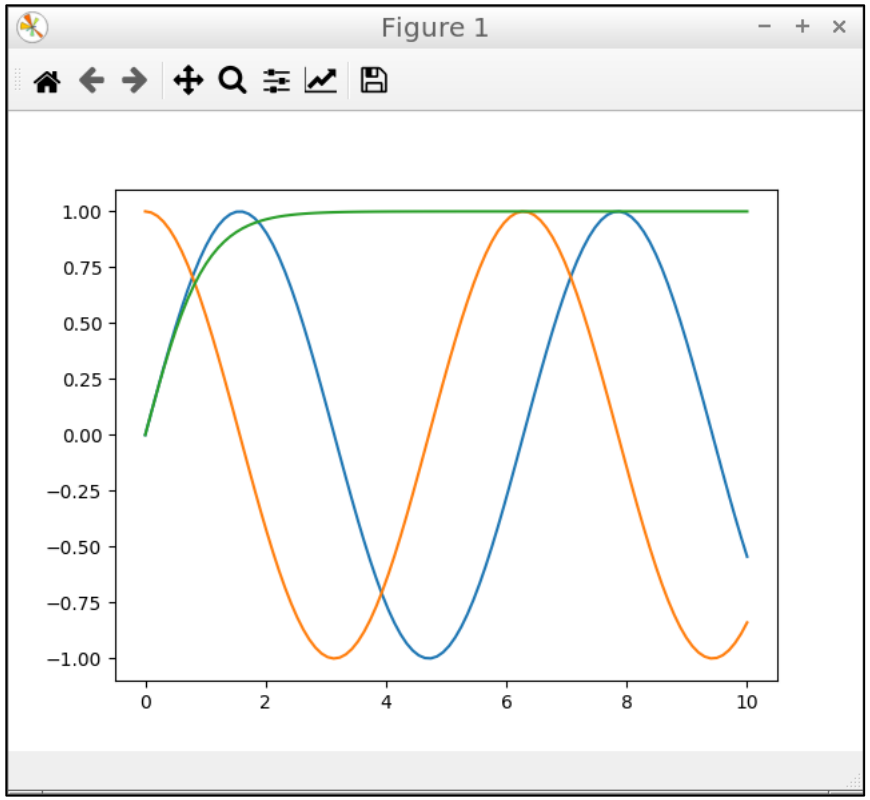
- Ploteado de un gráfico simple de líneas

- Para plotear usaremos el método *plot()* del objeto *Axes.*

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 plt.ion()
5 fig1 = plt.figure()
6 ax1 = plt.axes()
7
8 x = np.linspace(0,10,100)
9 ax1.plot(x, np.sin(x))
```

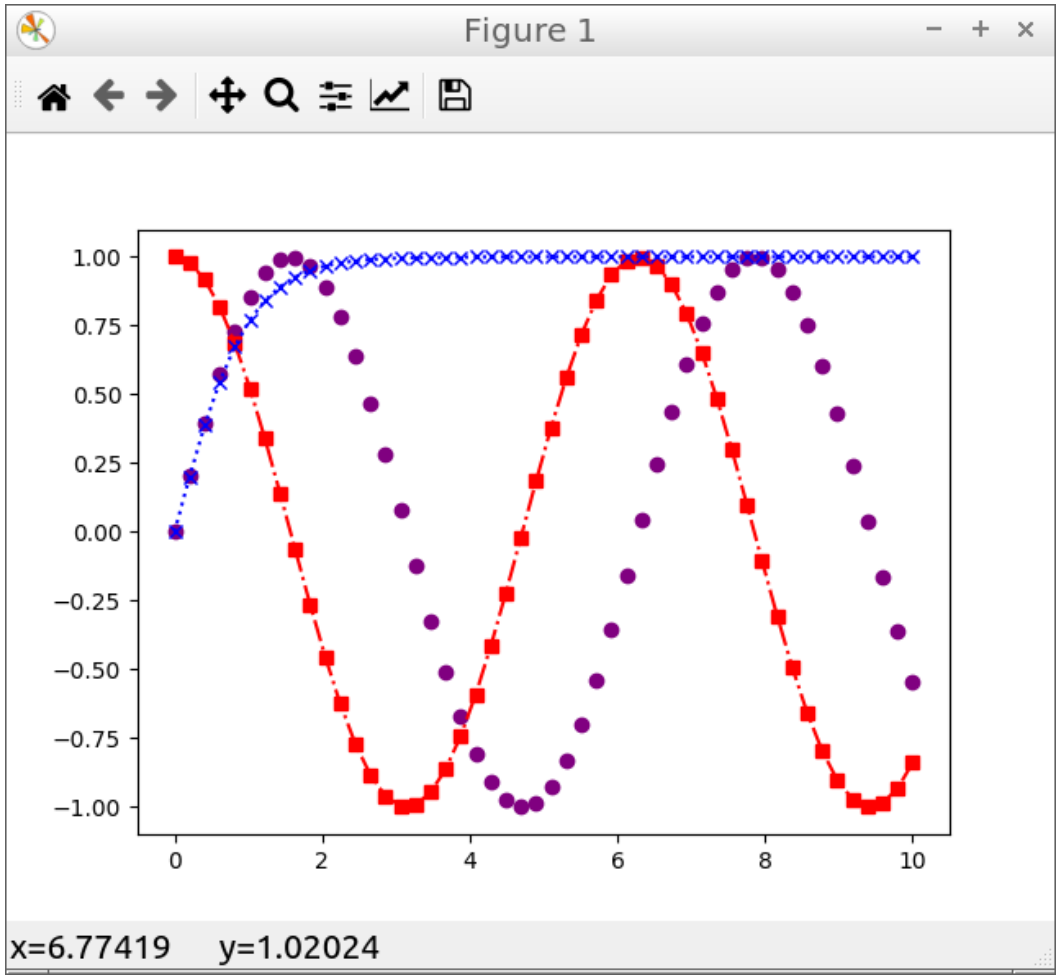



```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 plt.ion()
5 fig1 = plt.figure()
6 ax1 = plt.axes()
7
8 x = np.linspace(0,10,100)
9 ax1.plot(x, np.sin(x))
10 ax1.plot(x, np.cos(x))
11 ax1.plot(x, np.tanh(x))
12 #ax1.plot(x, np.sin(x), x , np.cos(x), x, np.tanh(x))
```



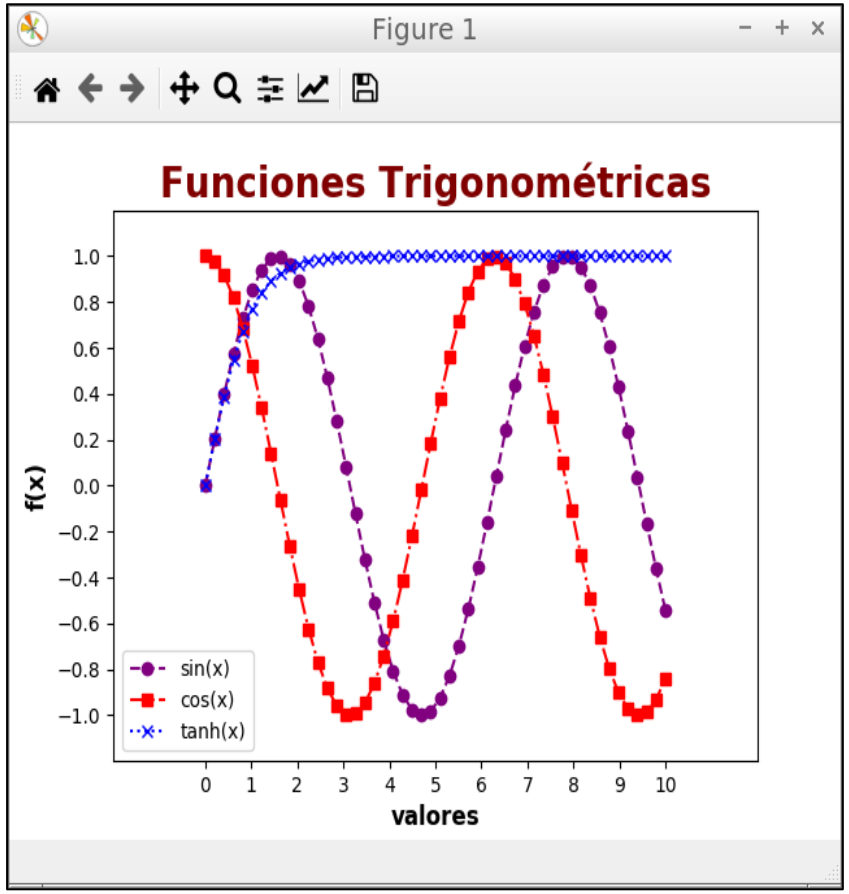
- Personalizando las líneas

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 plt.ion()
5 fig1 = plt.figure()
6 ax1 = plt.axes()
7
8 x = np.linspace(0,10,50)
9 ax1.plot(x, np.sin(x), color = 'purple', linestyle = '', marker = 'o')
10 ax1.plot(x, np.cos(x), color = 'red', linestyle = '-.', marker = 's')
11 ax1.plot(x, np.tanh(x), color = 'blue', linestyle = ':', marker = 'x')
```



- Añadiendo título, grid y leyenda al gráfico.

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

plt.ion()
fig1 = plt.figure()
ax1 = plt.axes()

x = np.linspace(0,10,50)
l1 = ax1.plot(x, np.sin(x), color = 'purple', linestyle = '--', marker = 'o')
l2 = ax1.plot(x, np.cos(x), color = 'red', linestyle = '-.', marker = 's')
l3 = ax1.plot(x, np.tanh(x), color = 'blue', linestyle = ':', marker = 'x')

ax1.set_title('Funciones Trigonométricas', color = 'maroon', size = 20, weight = 'heavy')
##ax1.grid('True')
ax1.legend(['sin(x)', 'cos(x)', 'tanh(x)'])

ax1.set_xlabel('valores', size = 12, weight = 'bold')
ax1.set_ylabel('f(x)', size = 12, weight = 'bold')
ax1.set_xticks(np.arange(0,11,1))
ax1.set_yticks(np.arange(-1.0, 1.1, 0.2))
ax1.set_xlim(-2,12)
ax1.set_ylim(-1.2,1.2)
```

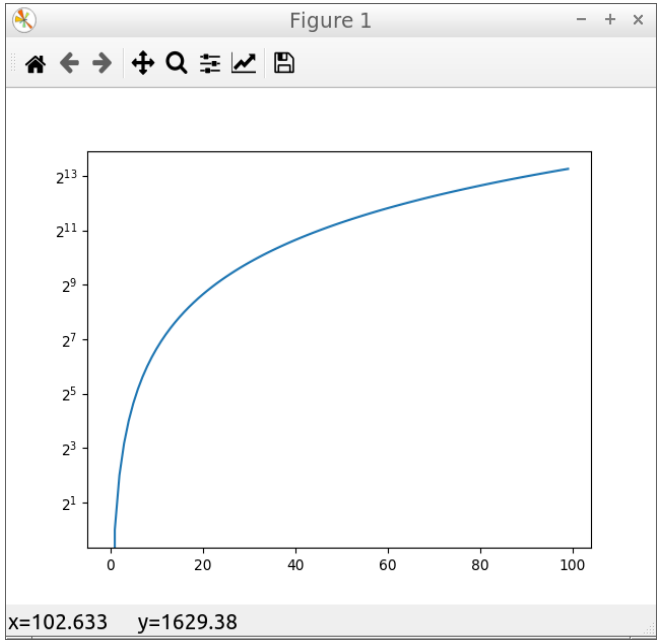


- Cambiando la escala de los ejes

Podemos establecer una de las siguientes escalas con los métodos set_*yscale()* y set_*xscale()* del objeto *Axes:*

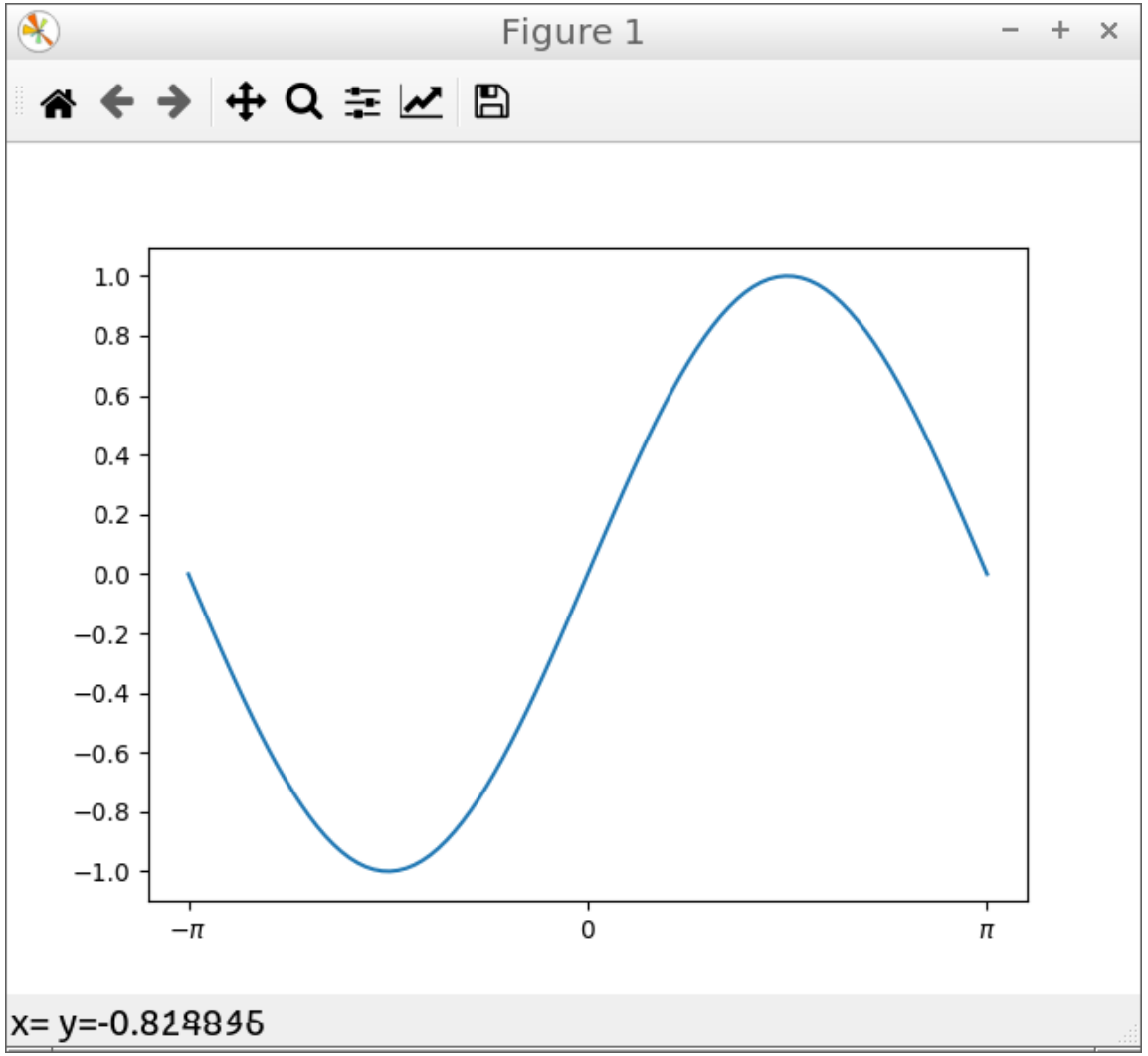
- *linear:* escala lineal. Es la escala por defecto.
- *log:* escala logarítmica. Esta escala sólo acepta valores positivos. Los valores negativos se pueden tratar de dos maneras: *mask* = ignorarlos (valor por defecto)
  c*lip* = convertirlos a un valor muy pequeño próximo a 0.
- *symlog:* symmetrical log. Acepta tanto valores positivos como negativos.
- *logit:* escala que se usa para datos comprendidos entre 0 y 1 ambos excluidos. Los datos fuera de dicho rango se tratarán igual que en la escala logarítmica *(log)* indicando *mask* o *clip.*

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 plt.ion()
5 fig1 = plt.figure()
6 ax1 = plt.axes()
7
8 x = np.arange(100)
9 y = x ** 2
10 ax1.plot(x,y)
11 ax1.set_yscale('log', basey = 2)
```



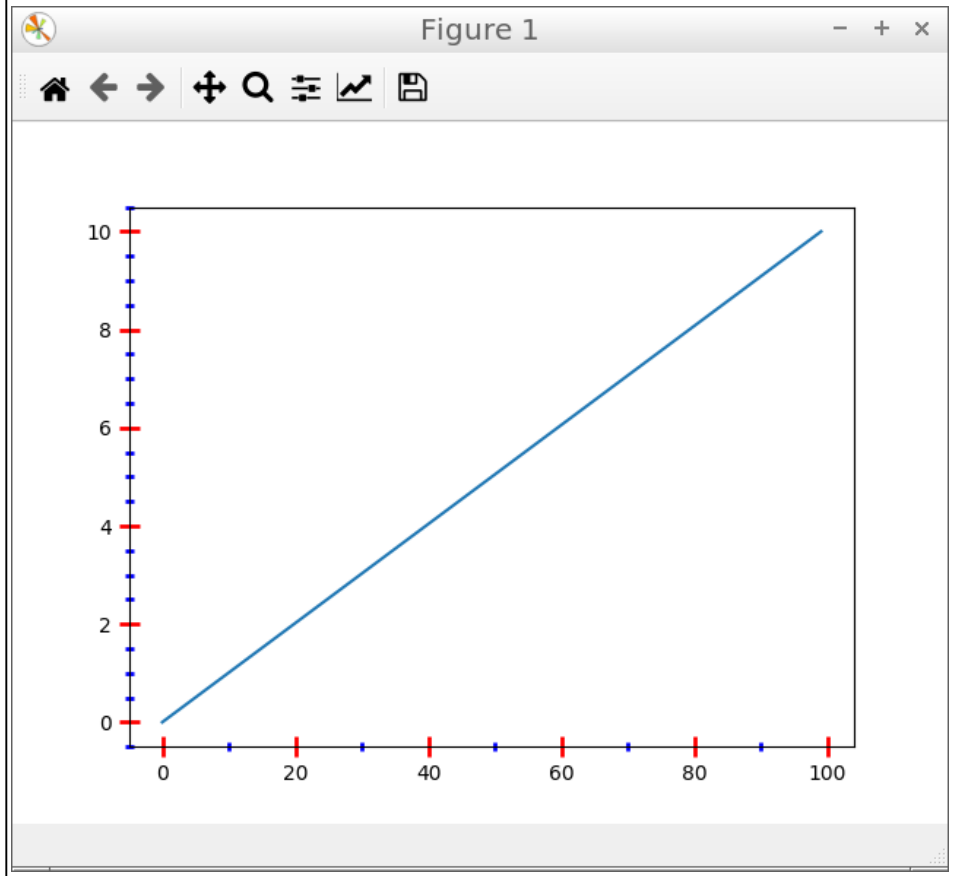
- Estableciendo los valores y sus etiquetas para los ejes

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 plt.ion()
5 x = np.linspace(-np.pi, np.pi, 100)
6 y = np.sin(x)
7
8 fig1 = plt.figure()
9 ax1 = plt.axes()
10 ax1.set_yticks(np.arange(-1, 1.1, 0.2))
11 ax1.set_xticks([-np.pi, 0, np.pi])
12 ax1.set_xticklabels(["$-\pi$", '0', '$\pi$'])
13
14 ax1.plot(x,y)
15
16 plt.show()
```



- Personalizando las marcas indicadoras principales y secundarias de los ejes

```python
import matplotlib.pyplot as plt
import numpy as np
from matplotlib.ticker import MultipleLocator # para indicar el número de marcas secundarias

plt.ion()
fig1 =plt.figure()
ax1 = plt.axes()
x = np.linspace(0, 10, 100)

# activamos las marcas secundarias
ax1.minorticks_on()

# parte común a ambos ejes o both
# width = ancho o grosor, direction = [in, out, inout]
ax1.tick_params(which='both', width = 2, direction = 'inout')

# parte aplicable a las marcas principales o major
# length = altura o longitud de la marca, color de la línea del grid y opacidad de la misma
ax1.tick_params(which='major', length = 10,  color = 'r', grid_color='r', grid_alpha=0.25)

# parte aplicable a las marcas secundarias o minor
ax1.tick_params(which='minor', length = 5, color = 'b', grid_color='g', grid_alpha=0.1)

# indicamos el número de marcas secundarias en el eje x y en el eje y
# dicho número se indican en múltiplos de la unidad indicada
# para esto hay que importar MultipleLocator
ax1.xaxis.set_minor_locator(MultipleLocator(10))    # una marca secundaria cada 10 unidades en el eje x
ax1.yaxis.set_minor_locator(MultipleLocator(0.5))   # una marca cada 0.5 unidades en el eje y

ax1.plot(x)
```

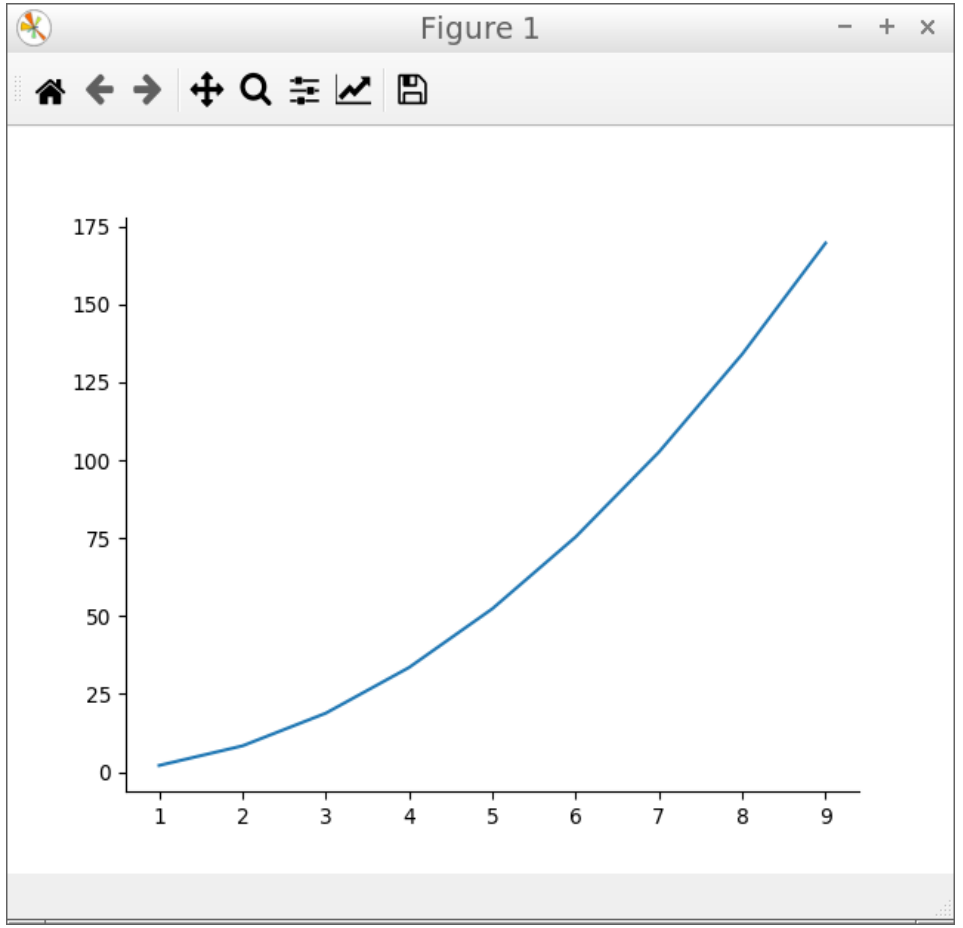


- Los bordes del gráfico (spines)

Podemos ocultar uno o varios bordes del Axes con la función spines():

axes.spines('borde', estado)

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

x = np.arange(1, 10)
f = 2 * np.pi * x / 3 * x

fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(x,f)
ax.spines['right'].set_visible(False)
ax.spines['top'].set_visible(False)
```

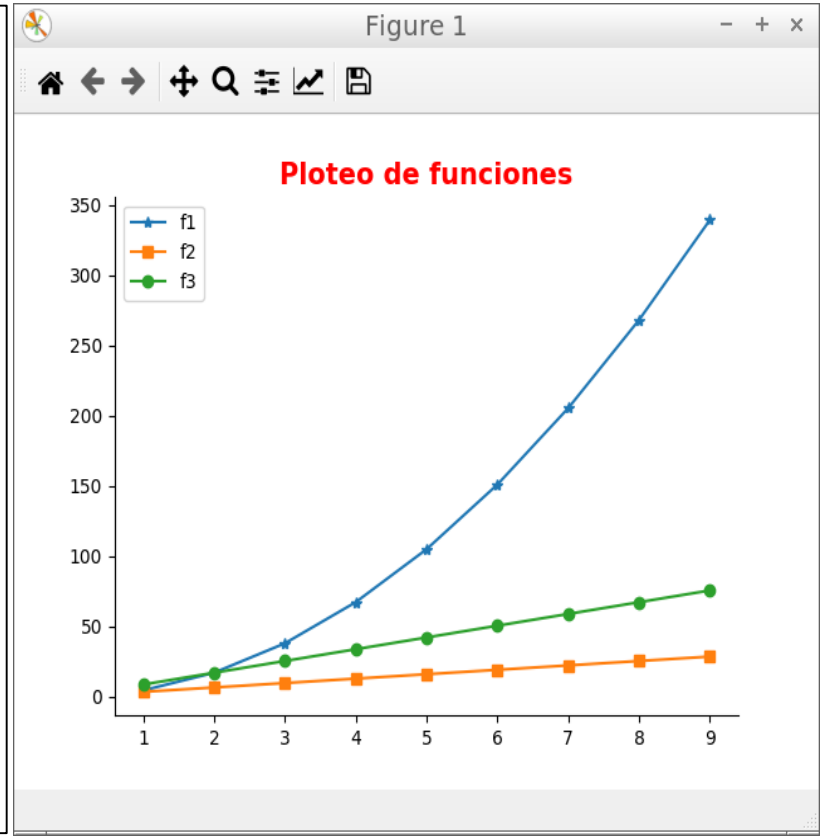


Ejercicio 6: Utilizando notación Orientada a Objetos crear una figura para plotear mediante líneas las siguientes funciones:

f1 = 2pi * $x^2/3$ f2 = pi/2 * x^2 f3 = 4/3 pi * x^2 donde x = np.arange(1,10)

Añadir un título a la figura, una leyenda, establecer diferentes marcadores para cada línea y eliminar los bordes superior y derecho de la gráfica.

```
13 import matplotlib.pyplot as plt
14 import numpy as np
15
16 x = np.arange(1, 10)
17
18 f1 = 2 * np.pi * x * x * 2/3
19 f2 = np.pi / 2 * 2 * x
20 f3 = 4/3 * np.pi * 2 * x
21
22 fig1 = plt.figure()
23 ax1 = plt.axes()
24
25 ax1.plot(x, f1, label = 'f1', marker = '*')
26 ax1.plot(x, f2, label = 'f2', marker = 's')
27 ax1.plot(x, f3, label = 'f3', marker = 'o')
28 ax1.set_title('Ploteo de funciones', weight = 'bold', size = 15, color = 'r')
29 ax1.spines['right'].set_visible(False)
30 ax1.spines['top'].set_visible(False)
31 ax1.legend()
32
33 plt.show()
```



- Subplots (i)

Podemos tener diferentes zonas dentro de una misma figura, lo que nos permite plotear varias gráficas que se pueden tratar de forma independiente dentro de dicha figura.

Una forma de conseguir eso es creando un objeto *Figure* en el que se añadirán tantos objetos *Axes* como sea neccesario.

- *plt.figure():* nos devuelve una instancia del objeto figura, donde podemos añadir una o varias instancias del objeto *Axes.*

  p.e: fig1 = plt.figure() # crea una instancia de un objeto figura en la variable fig1

- *add_subplot():* es un método de un objeto tipo figura que nos devuelve una instancia del objeto *Axes (*área donde poder mostrar la función o los datos a plotear). Esta función tiene el siguiente formato:

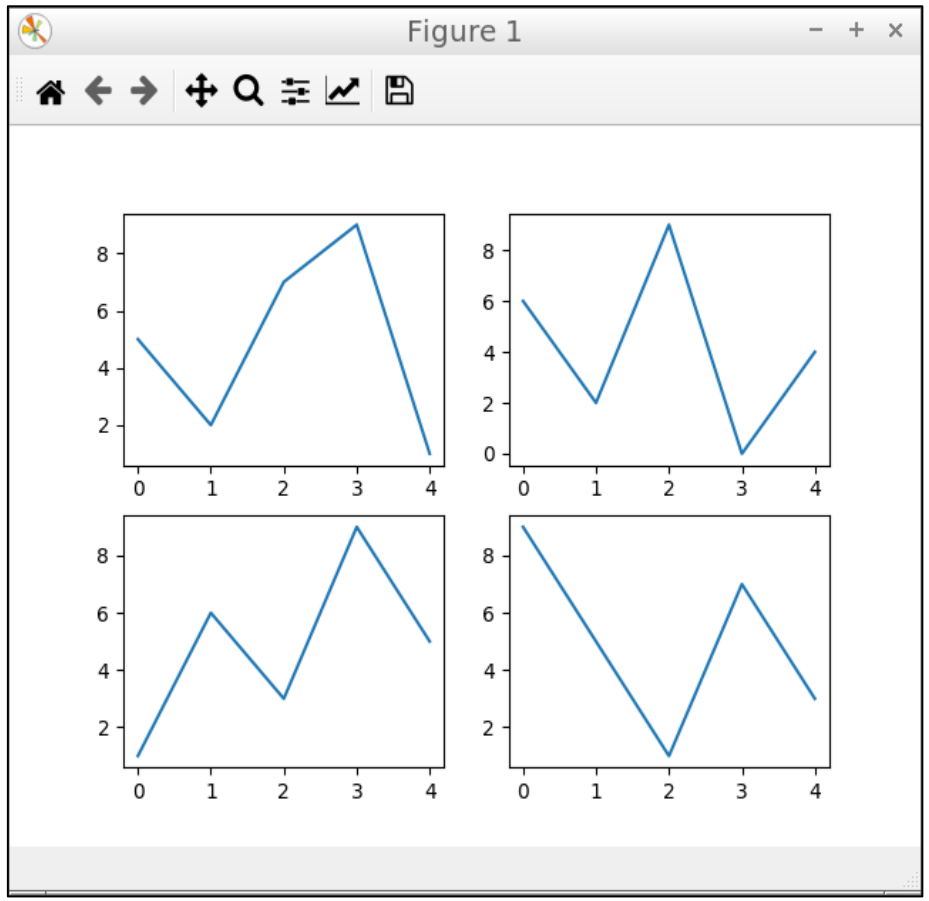
  add_subplot(num_filas, num_columnas, num_ploteado_actual)

  p.e: ax1 = fig1.add_subplot(2, 2, 1) # crea una instancia de un objeto Axes en la variable ax1 para plotear en la primera zona de la figura fig1, que tendrá 4 áreas de ploteado.

- Subplots (ii)

A continuación vemos un ejemplo donde se crea una figura con cuatro gráficas usando los métodos del objeto *Figure*.

```
1 import matplotlib.pyplot as plt
2
3 fig1 = plt.figure()
4
5 ax1 = fig1.add_subplot(2, 2, 1)
6 ax1.plot([5,2,7,9,1])
7
8 ax2 = fig1.add_subplot(2, 2, 2)
9 ax2.plot([6,2,9,0,4])
10
11 ax3 = fig1.add_subplot(2, 2, 3)
12 ax3.plot([1,6,3,9,5])
13
14 ax4 = fig1.add_subplot(2, 2, 4)
15 ax4.plot([9,5,1,7,3])
16
17 plt.show()
```



Nota: si la matriz de gráficas creada con *subplot()* tiene menos de 10 elementos, podemos abreviar la notación concatenando los 3 parámetros de dicha función. Por ejemplo: *fig1.add_subplot(2,2,4)* es equivalente a escribir *fig1.add_subplot(224).*

- Subplots (iii)

Otra forma de tener diferentes zonas dentro de una misma figura es a través de la función *subplots()* de *Pyplot.*

La función *subplots()* sin argumentos nos devuelve a la vez una instancia de un objeto *Figure* y una instancia de un objeto *Axes*.

p.e: fig, ax = plt.subplots()

En este ejemplo la variable fig será una instancia del tipo *Figure* y la variable *ax* será una instancia del tipo *Axes.*

Si indicamos el número de filas y columnas nos devolverá a la vez una instancia de un objeto *Figure* y tantas instancias del objeto *Axes* como resultado de multiplicar el número de filas por el número de columnas.

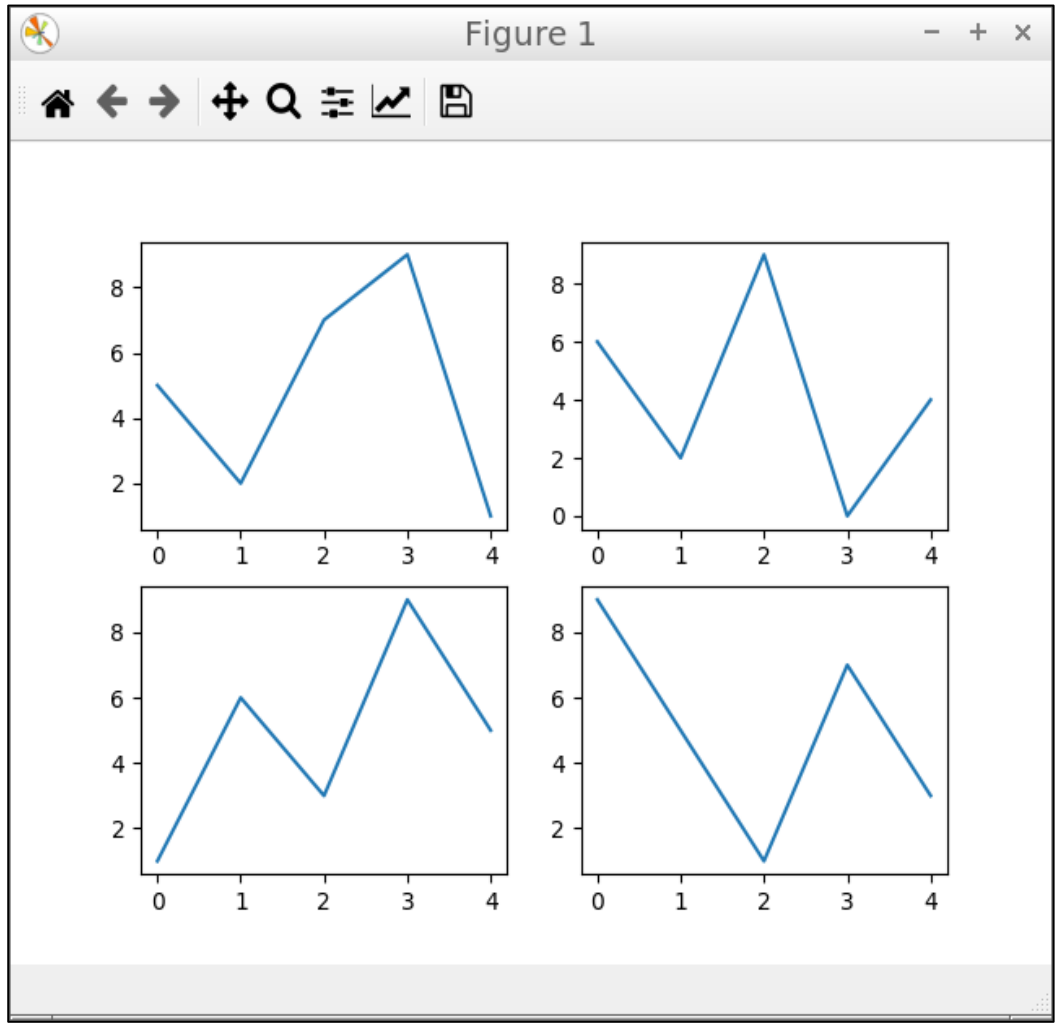
p.e: *fig1, ax = plt.subplots(2,2)*

En este ejemplo la variable *fig1* será una instancia del tipo *Figure* y la variable *ax* será un array bidimensional de 2 x 2 instancias del tipo *Axes.*

- Subplots (iv)

A continuación podemos ver un ejemplo donde se crea una figura con cuatro gráficas usando la función *subplots()* de Pyplot.

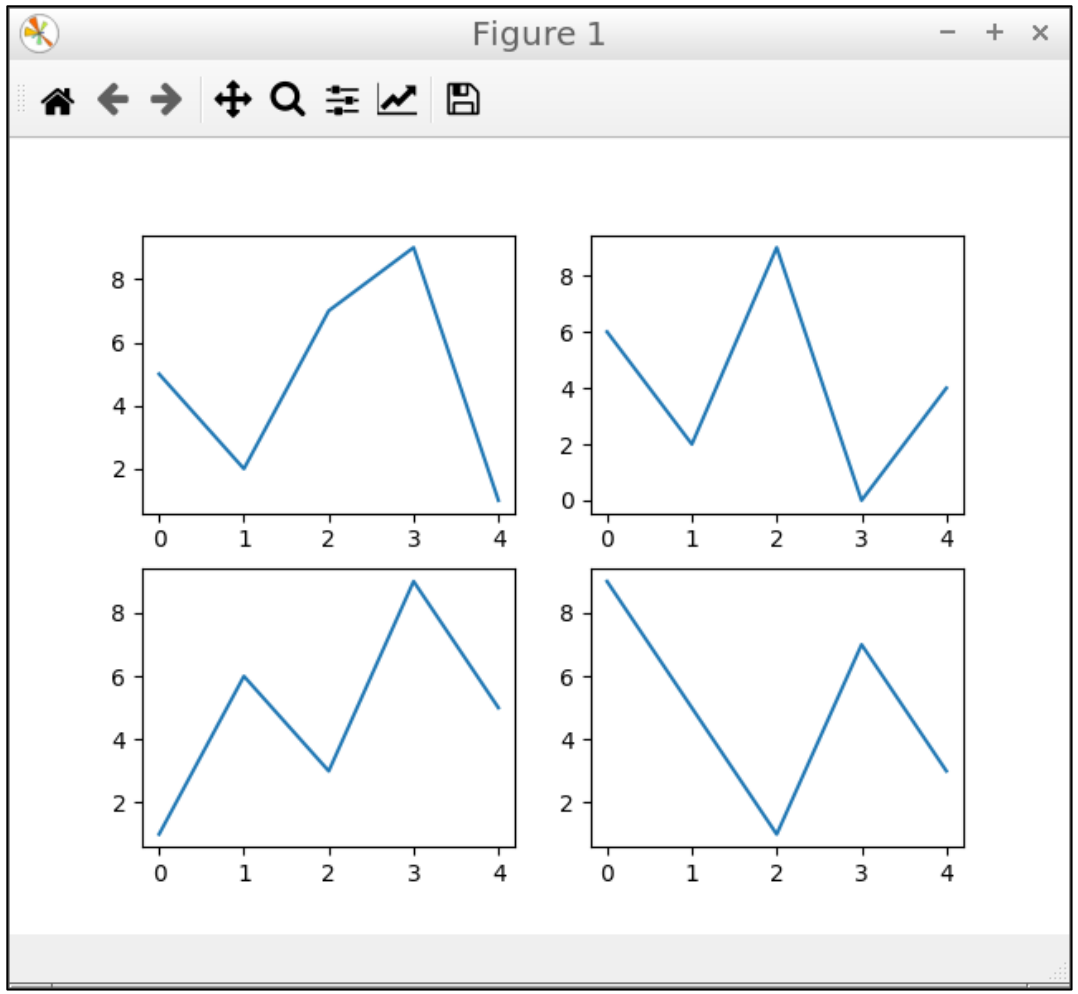
```
1 import matplotlib.pyplot as plt
2
3 fig1, ax = plt.subplots(2,2)
4 ax[0,0].plot([5,2,7,9,1])
5 ax[0,1].plot([6,2,9,0,4])
6 ax[1,0].plot([1,6,3,9,5])
7 ax[1,1].plot([9,5,1,7,3])
```



- Subplots (v)

```python
import matplotlib.pyplot as plt

fig1 = plt.figure()

ax1 = fig1.add_subplot(2, 2, 1)
ax1.plot([5,2,7,9,1])

ax2 = fig1.add_subplot(2, 2, 2)
ax2.plot([6,2,9,0,4])

ax3 = fig1.add_subplot(2, 2, 3)
ax3.plot([1,6,3,9,5])

ax4 = fig1.add_subplot(2, 2, 4)
ax4.plot([9,5,1,7,3])

plt.show()
```

```python
import matplotlib.pyplot as plt

fig1, ax = plt.subplots(2,2)

ax[0,0].plot([5,2,7,9,1])
ax[0,1].plot([6,2,9,0,4])
ax[1,0].plot([1,6,3,9,5])
ax[1,1].plot([9,5,1,7,3])
```

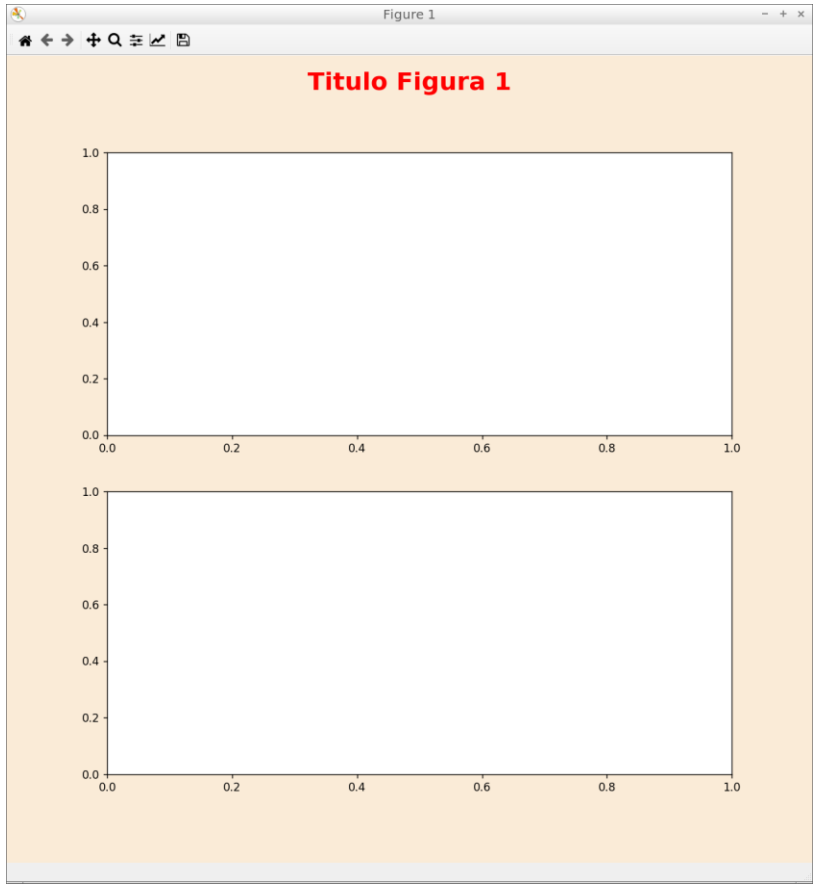


- Subplots (vi)

La parte correspondiente al objeto *Figure* es común a todos y tiene sus propios métodos y atributos.

- figsize(ancho, alto): dimensiones de la figura en pulgadas.
- dpi(resolución): es la resolución de la imagen en puntos por pulgada (dots per inch)
- facecolor(color): color de fondo

```
2 import matplotlib.pyplot as plt
3
4 fig = plt.figure(figsize = (10,10), dpi = 125, facecolor = 'antiquewhite')
5 fig.suptitle('Titulo Figura 1', color = 'r', weight = 'heavy', size = 22)
6 ax1 = fig.add_subplot(211)
7 ax2 = fig.add_subplot(212)
```



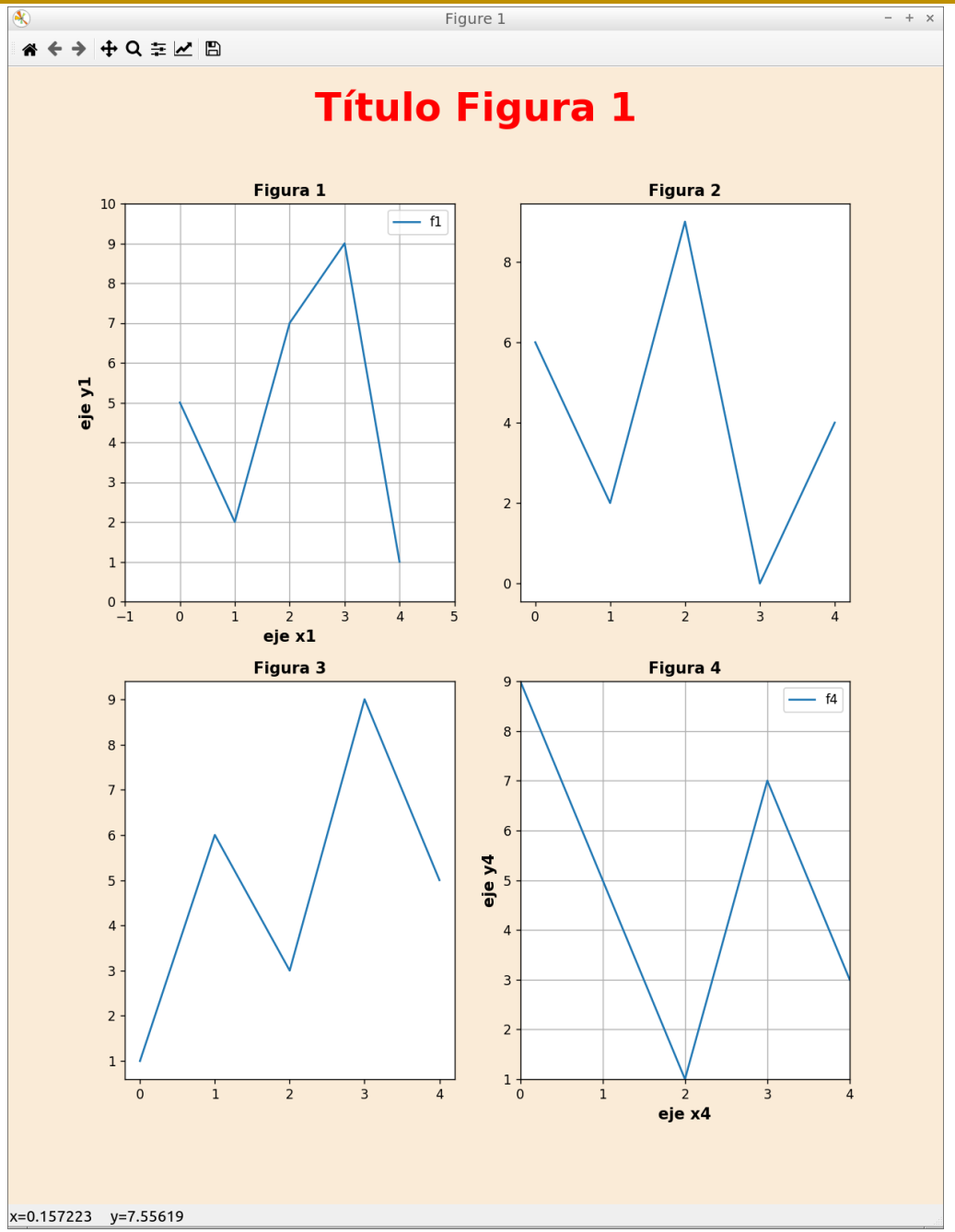
- Subplots (vii)

Cada gráfica es independiente y tiene sus propias características independientes, que podemos establecer con los siguientes métodos:

- set_title(): título para la gráfica
- set_xlabel() y set_ylabel(): etiqueta para el eje x e y respectivamente.
- set_xscale() y set_yscale(): para las escalas del eje x e y respectivamente.
- set_xlim() y set_ylim(): para establecer los límites de cada eje.
- set_xticks() y set_ yticks(): para cambiar las marcas de los ejes.
- grid(): para mostrar el grid en la gráfica.
- legend(): para mostrar la leyenda

- Subplots (viii)

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

fig1 = plt.figure(figsize = (10,12), dpi = 125, facecolor = 'antiquewhite')
fig1.suptitle('Título Figura 1', color = 'red', weight = 'heavy', size = 30)

ax1 = fig1.add_subplot(2, 2, 1)
ax1.plot([5,2,7,9,1], label = 'f1')
ax1.set_title('Figura 1', weight = 'bold')
ax1.set_xlabel('eje x1', size = 12, weight = 'bold')
ax1.set_ylabel('eje y1', size = 12, weight = 'bold')
ax1.grid(True)
ax1.set_ylim(0,10)
ax1.set_xlim(-1,5)
ax1.set_yticks(np.arange(0, 11, 1))
ax1.legend()

ax2 = fig1.add_subplot(2, 2, 2)
ax2.plot([6,2,9,0,4])
ax2.set_title('Figura 2', weight = 'bold')

ax3 = fig1.add_subplot(2, 2, 3)
ax3.plot([1,6,3,9,5])
ax3.set_title('Figura 3', weight = 'bold')

ax4 = fig1.add_subplot(2, 2, 4)
ax4.plot([9,5,1,7,3], label = 'f4')
ax4.set_title('Figura 4', weight = 'bold')
ax4.set_xlabel('eje x4', size = 12, weight = 'bold')
ax4.set_ylabel('eje y4', size = 12, weight = 'bold')
ax4.grid(True)
ax4.set_ylim(1,9)
ax4.set_xlim(0,4)
ax4.set_xticks(np.arange(0,5,1))
ax4.legend()
```



- Subplots (ix)

Podemos ajustar la distribución de las gráficas de la figura a través de la siguiente función de Pyplot*:*

plt.subplots_adjust(bottom, top, left, right, hspace, wspace)

donde:

- *bottom, top, left, right,* indican el espacio que se dejará por cada uno de los lados indicados. Este espacio se mide en porcentajes. Hay que tener en cuenta que los valores se calculan con respecto a los bordes del recuadro que encierra a la gráfica.

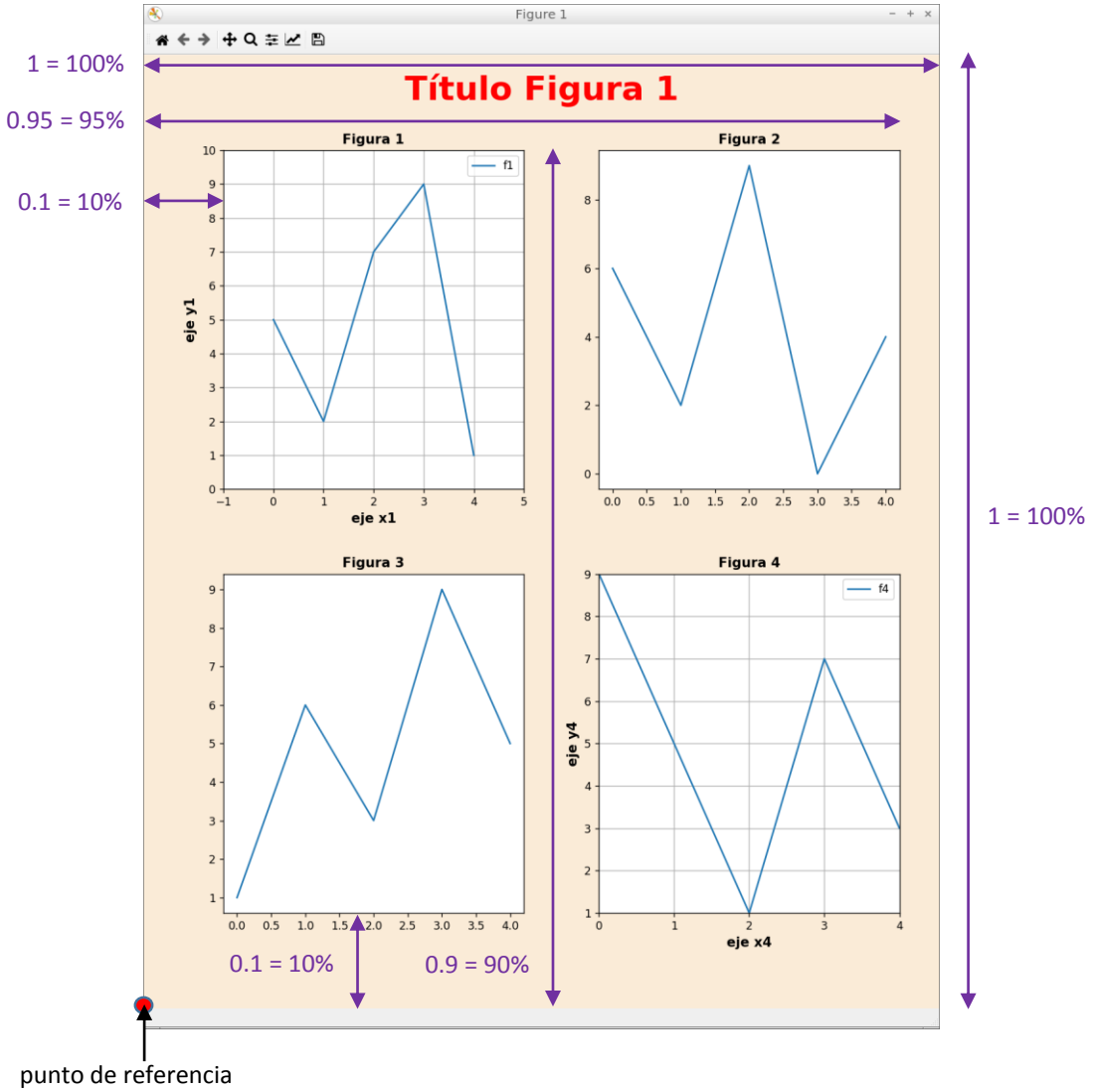
- *hspace, wspace,* indican la separación entre los diferentes gráficos de la figura, medido como porcentaje en relación con el alto y ancho de los gráficos.

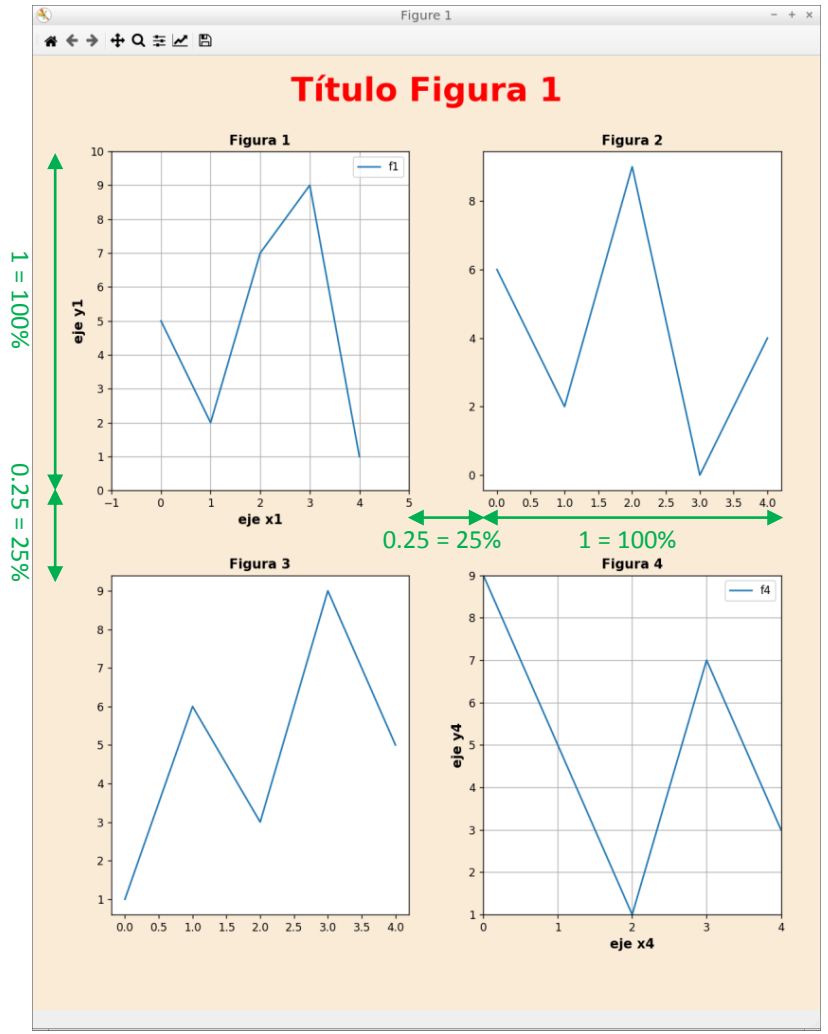
p.e: *plt.subplots_adjust(bottom=0.1, top = 0.9, left = 0.1, right = 0.95, hspace = 0.25, wspace = 0.25)*

En el ejemplo se dejará un 10% del alto de la figura como espacio de separación abajo, el gráfico de la parte superior terminará en el 90% del alto de la figura contado desde abajo, o lo que es lo mismo, empezará por arriba dejando un 10% del alto de la figura, se dejará un espacio de un 10% del ancho de la figura como espacio a la izquierda y la figura más a la derecha terminará al alcanzarse el 95% del ancho total, o lo que es lo mismo, se dejará un 5% de espacio por la derecha. Además, se dejará una separación en altura entre los gráficos de un 25% del alto de un gráfico y un 25% de separación a lo ancho entre los gráficos.

- Subplots (x)

plt.subplots_adjust(bottom=0.1, top = 0.9, left = 0.1, right = 0.95, hspace = 0.25, wspace = 0.25)



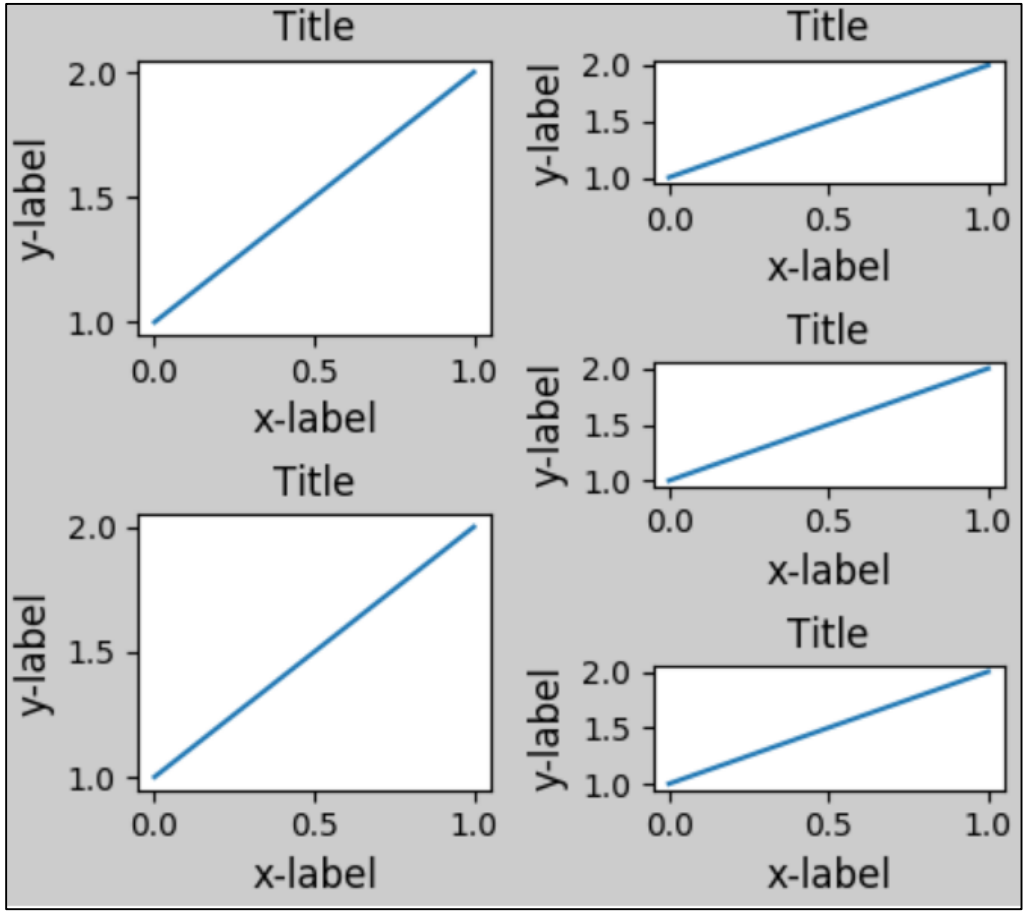


- Subplots (xi)

Podemos crear disposiciones irregulares y más complejas añadiendo grids y subgrids a la figura.

Ver documentación oficial: https://matplotlib.org/3.1.1/tutorials/intermediate/constrainedlayout_guide.html

```
fig = plt.figure()

gs0 = fig.add_gridspec(6, 2)

ax1 = fig.add_subplot(gs0[:3, 0])
ax2 = fig.add_subplot(gs0[3:, 0])

example_plot(ax1)
example_plot(ax2)

ax = fig.add_subplot(gs0[0:2, 1])
example_plot(ax)
ax = fig.add_subplot(gs0[2:4, 1])
example_plot(ax)
ax = fig.add_subplot(gs0[4:, 1])
example_plot(ax)
```

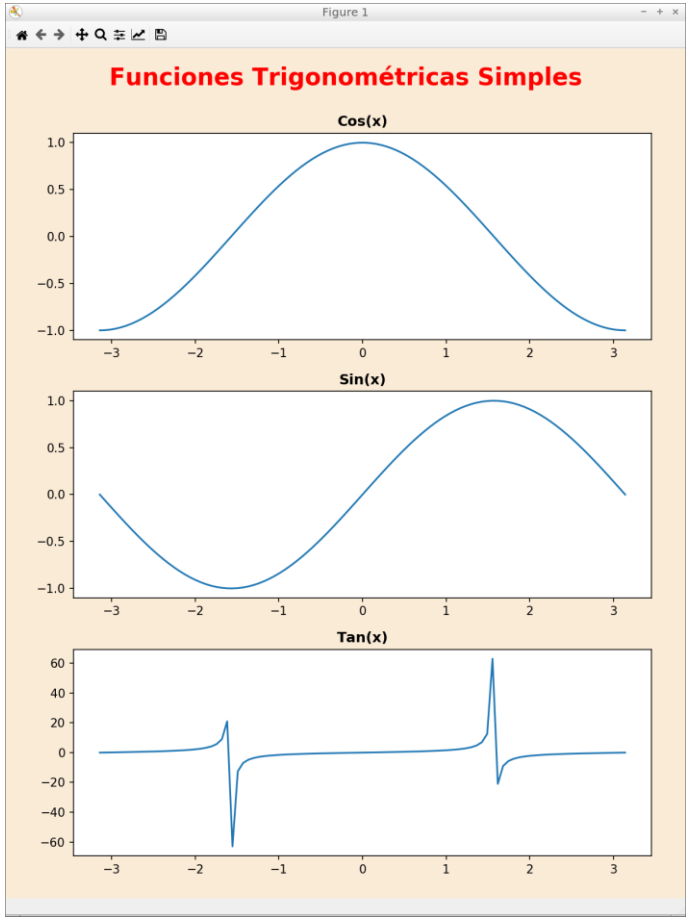


Ejercicio 7: Utilizando notación Orientada a Objetos crear una figura de tamaño 8x10 pulgadas y una resolución de 150 dpi, que esté dividida en 1 columna y 3 filas para plotear mediante líneas las siguientes funciones:

f1 = sin(x) f2 = cos(x) f3 = tan(x) donde x = 100 puntos en el rango [-pi, pi]

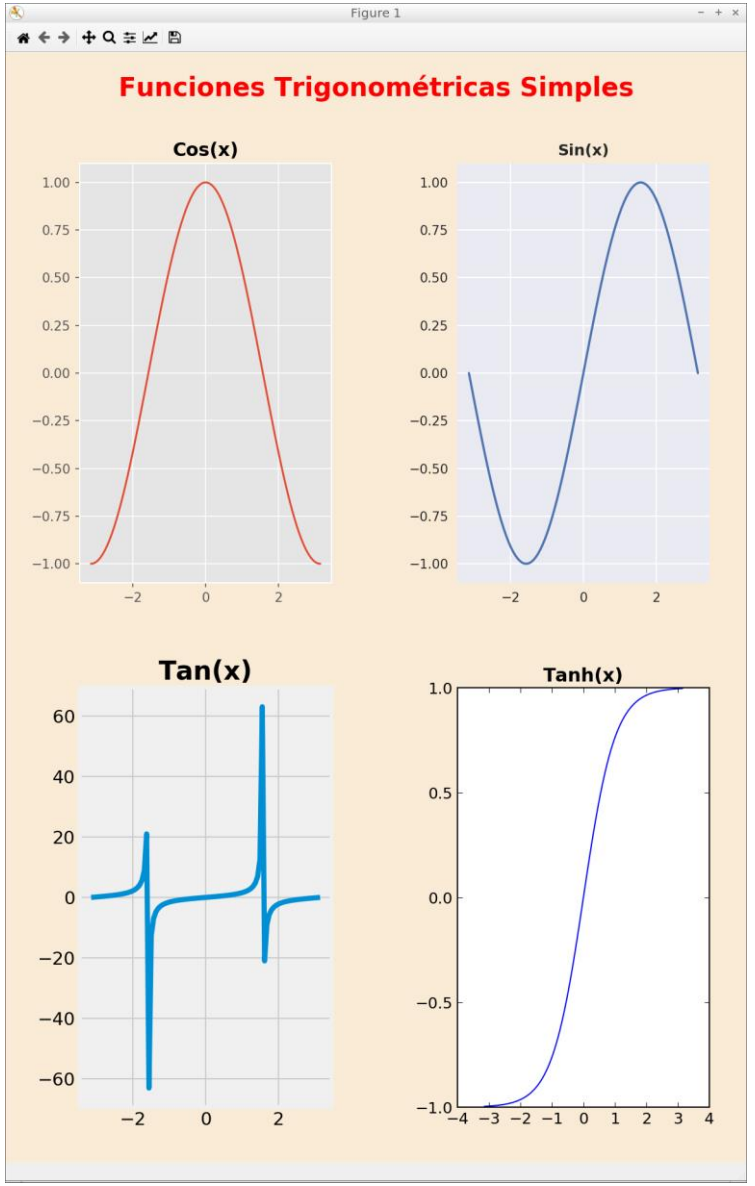
Añadir un título a cada subfigura y uno genérico a la figura. Dejar un espacio de un 5% abajo, un 10% a la izquierda, un 5% a la derecha y un 10% arriba.

```
13 import matplotlib.pyplot as plt
14 import numpy as np
15
16 x = np.linspace(-np.pi,np.pi,100)
17 c = np.cos(x)
18 s = np.sin(x)
19 t = np.tan(x)
20
21 fig = plt.figure(figsize = (8,10), dpi = 150, facecolor = 'antiquewhite')
22 fig.suptitle('Funciones Trigonométricas Simples', color = 'red', size =20, weight = 'bold')
23
24 ax1 = fig.add_subplot(3,1,1)
25 ax1.plot(x, c, label = 'cos(x)')
26 ax1.set_title('Cos(x)', weight = 'bold')
27
28 ax2 = fig.add_subplot(3, 1, 2)
29 ax2.plot(x, s, label = 'sin(x)')
30 ax2.set_title('Sin(x)', weight = 'bold')
31
32 ax3 = fig.add_subplot(3, 1, 3)
33 ax3.plot(x, t, label = 'tan(x)')
34 ax3.set_title('Tan(x)', weight = 'bold')
35
36 plt.subplots_adjust(bottom=0.05,  top = 0.9, left = 0.1, right = 0.95, hspace = 0.25, wspace = 0.5)
37
38 plt.show()
```



Ejercicio 8: crear una matriz de 2x2 gráficos y graficar las funciones cos(x), sin(x), tan(x) y tanh(x) en cada subplot. Poner cada subplot con uno de los siguientes estilos: ggplot, seaborn, fivethirtyeight y _classic_test.

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

x = np.linspace(-np.pi,np.pi,100)
c = np.cos(x)
s = np.sin(x)
t = np.tan(x)
th = np.tanh(x)

fig = plt.figure(figsize = (8,12), dpi = 150, facecolor = 'antiquewhite')
fig.suptitle('Funciones Trigonométricas Simples', color = 'red', size =20, weight = 'bold')


with plt.style.context("ggplot"):
    ax1 = fig.add_subplot(2,2,1)
    ax1.set_title('Cos(x)', weight = 'bold')
    ax1.plot(x, c, label = 'cos(x)')

with plt.style.context("seaborn"):
    ax2 = fig.add_subplot(2, 2, 2)
    ax2.plot(x, s, label = 'sin(x)')
    ax2.set_title('Sin(x)', weight = 'bold')

with plt.style.context("fivethirtyeight"):
    ax3 = fig.add_subplot(2, 2, 3)
    ax3.plot(x, t, label = 'tan(x)')
    ax3.set_title('Tan(x)', weight = 'bold')

with plt.style.context("_classic_test"):
    ax4 = fig.add_subplot(2, 2, 4)
    ax4.plot(x, th, label = 'tanh(x)')
    ax4.set_title('Tanh(x)', weight = 'bold')

plt.subplots_adjust(bottom=0.05,  top = 0.9, left = 0.1, right = 0.95, hspace = 0.25, wspace = 0.5)

plt.show()

plt.rcdefaults()
```

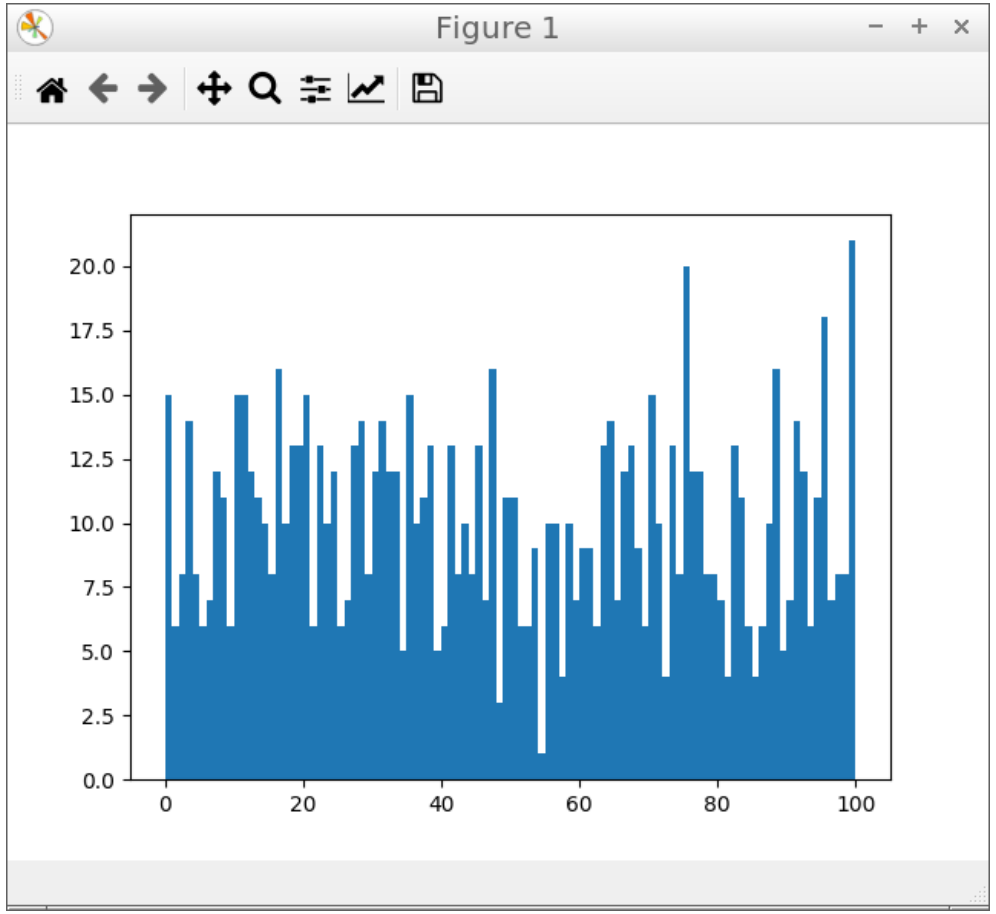


- Histograms

hist(datos, num_categorías, atributos_formato)

- *num_categorías:* indica el número de categorías a mostrar (por defecto 10).

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 y = np.random.randint(0, 100, 1000)
5 plt.hist(y, 100, color = 'r')
6 plt.show()
```



- Bar Charts

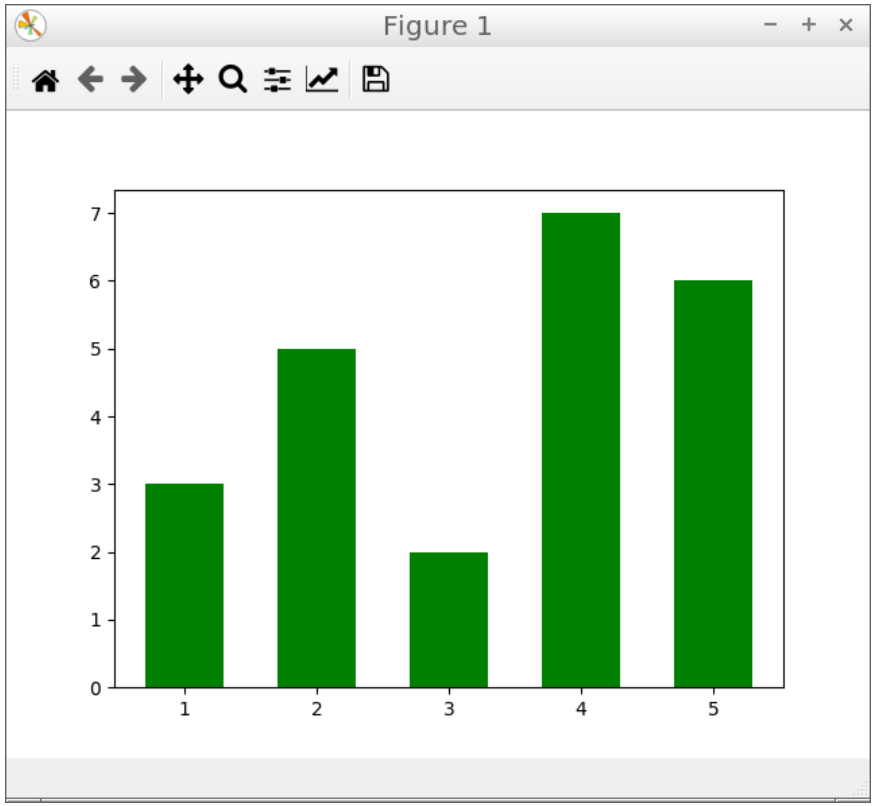
bar(coordenadas_eje_x, alturas, atributos_formato)

- *coordenadas_eje_x:* posición de cada barra en el eje x.

- *alturas:* altura de cada barra.

- el ancho de cada barra o width es de 0.8 por defecto.

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 pos_x = np.arange(1,6)
5 h = [3,5,2,7,6]
6 plt.bar(pos_x, h , width = 0.6, color = 'green', orientation = 'vertical')
7 plt.show()
```



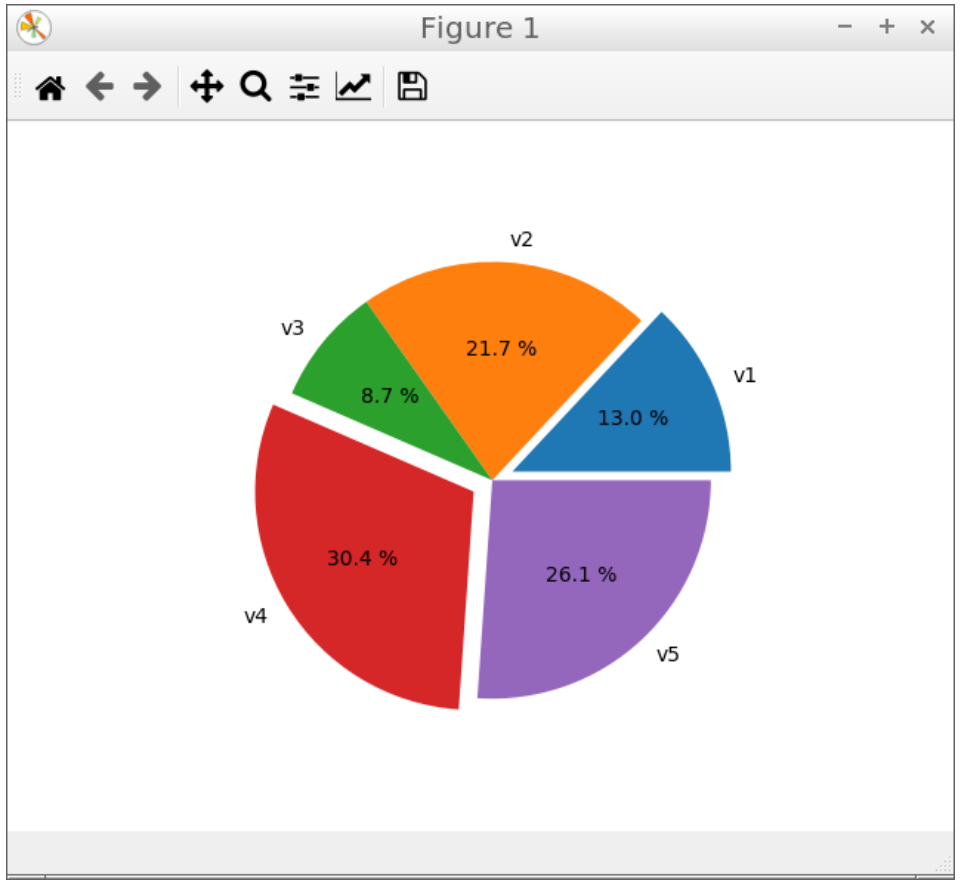
- Pie Charts

pie(valores, etiquetas, atributos_formato)

- *valores:* frecuencias a mostrar en cada sector.

- *etiquetas:* etiqueta de cada valor.

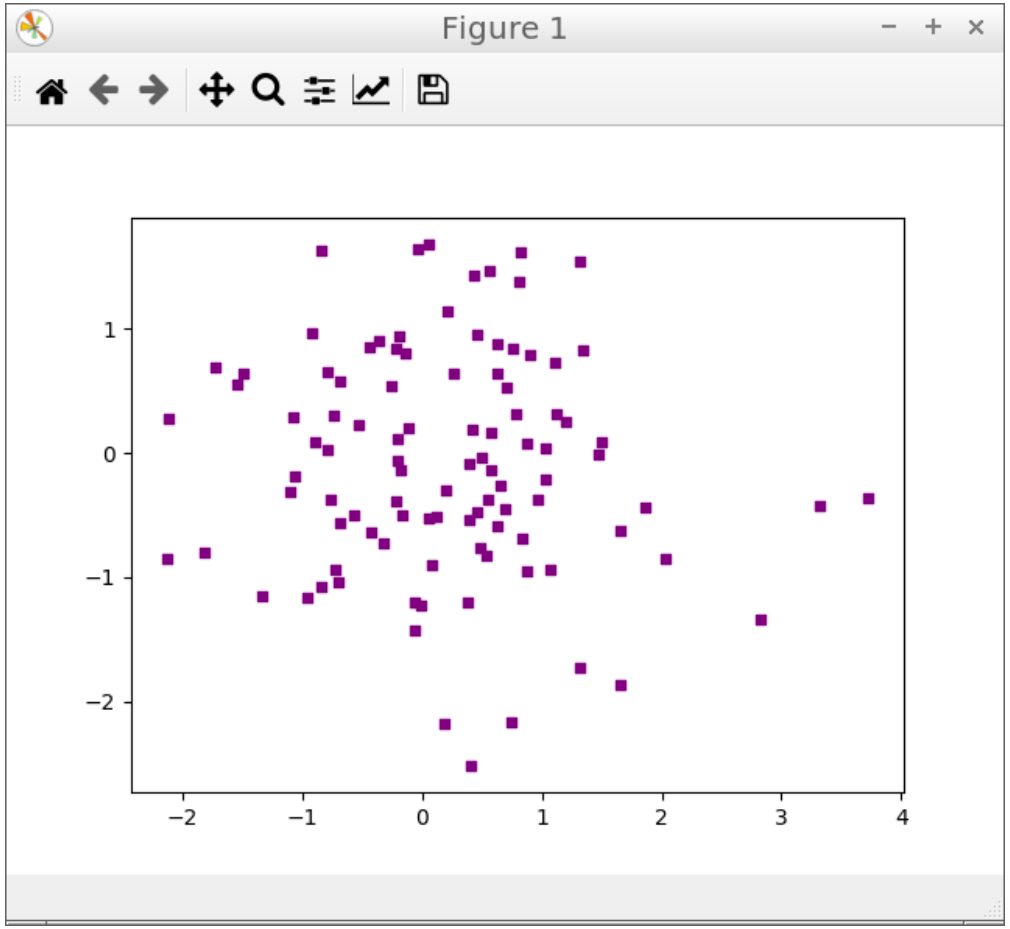
```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 frec = [3, 5 ,2, 7, 6]
5 eti = ['v1', 'v2', 'v3', 'v4', 'v5']
6 plt.pie(frec, labels = eti, autopct = '%1.1f %%', explode = [0.1, 0, 0, 0.1, 0])
7 plt.show()
```



- Scatter plots

scatter(x, y, atributos_formato)

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 x = np.random.randn(100)
5 y = np.random.randn(100)
6 plt.scatter(x, y, s = 25, c = 'purple', marker = 's')
7 plt.show()
```

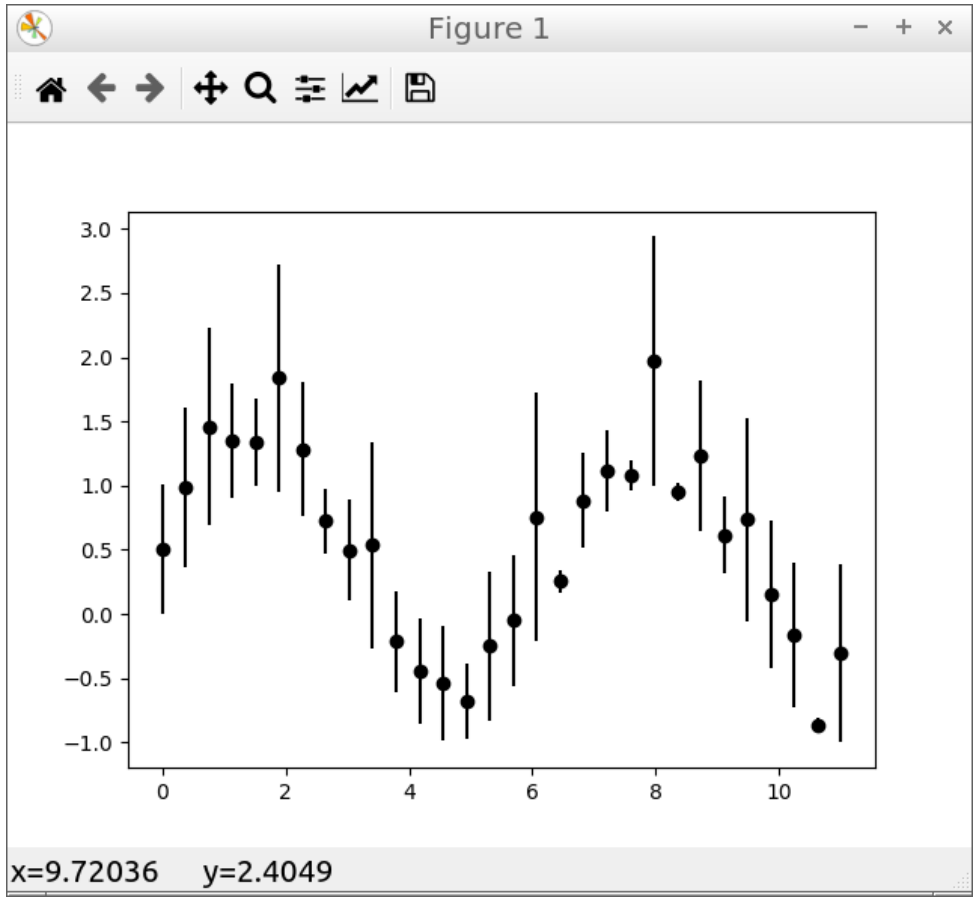


- Errorbars

errorbar(valores, errores, atributos_formato)

- *errores:* array con el error de cada valor.

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 x = np.linspace(0,11,30)
5 dy = np.random.random(30)
6 y = np.sin(x) + dy
7 plt.errorbar(x, y, yerr = dy, fmt = 'ok')
8 plt.show()
```

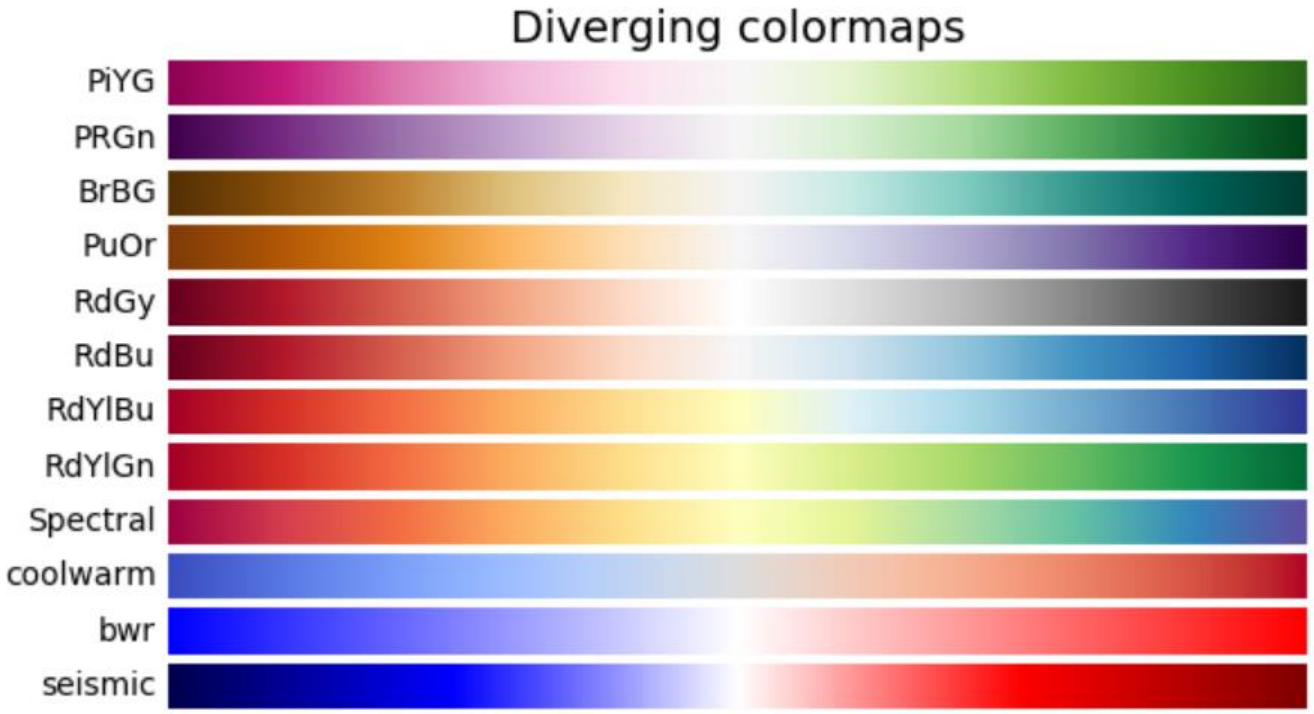


- Colormaps (i)

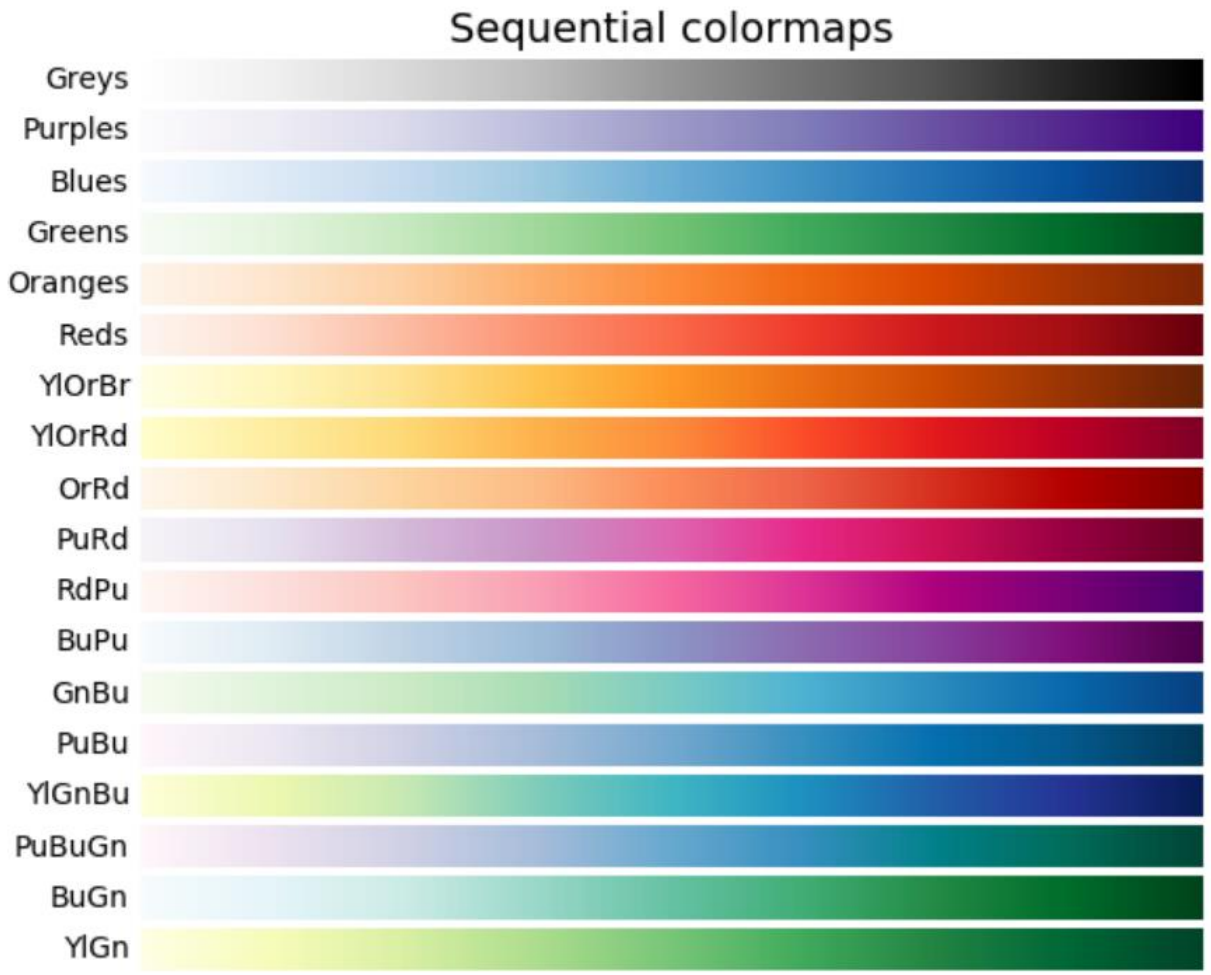
En algunos tipos de gráficos podemos establecer el mapa de color que nos interese. Dichos gráficos o elementos de las figuras que lo permiten, tendrán un atributo *cmap* para ello.

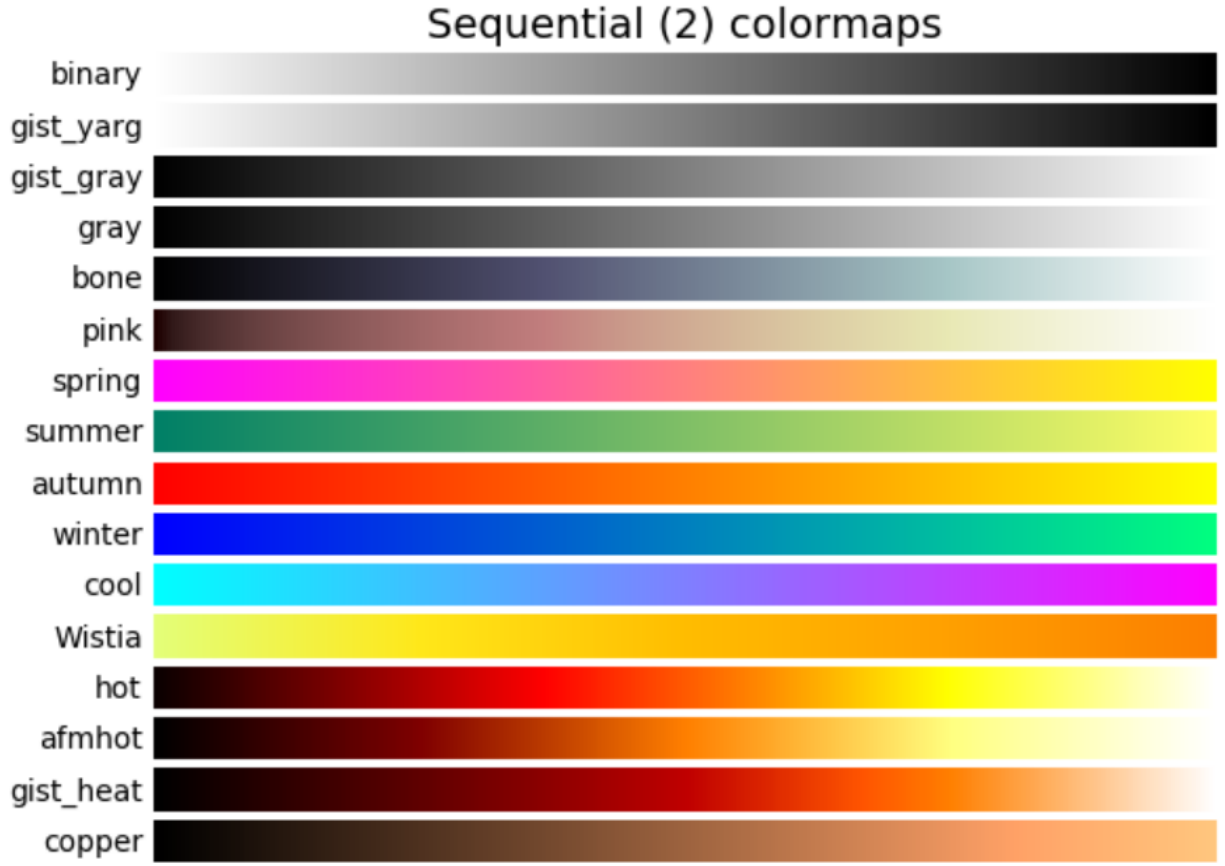
Podemos ver una referencia de todos los tipos de colormaps en la siguiente web:

https://matplotlib.org/3.1.1/gallery/color/colormap_reference.html

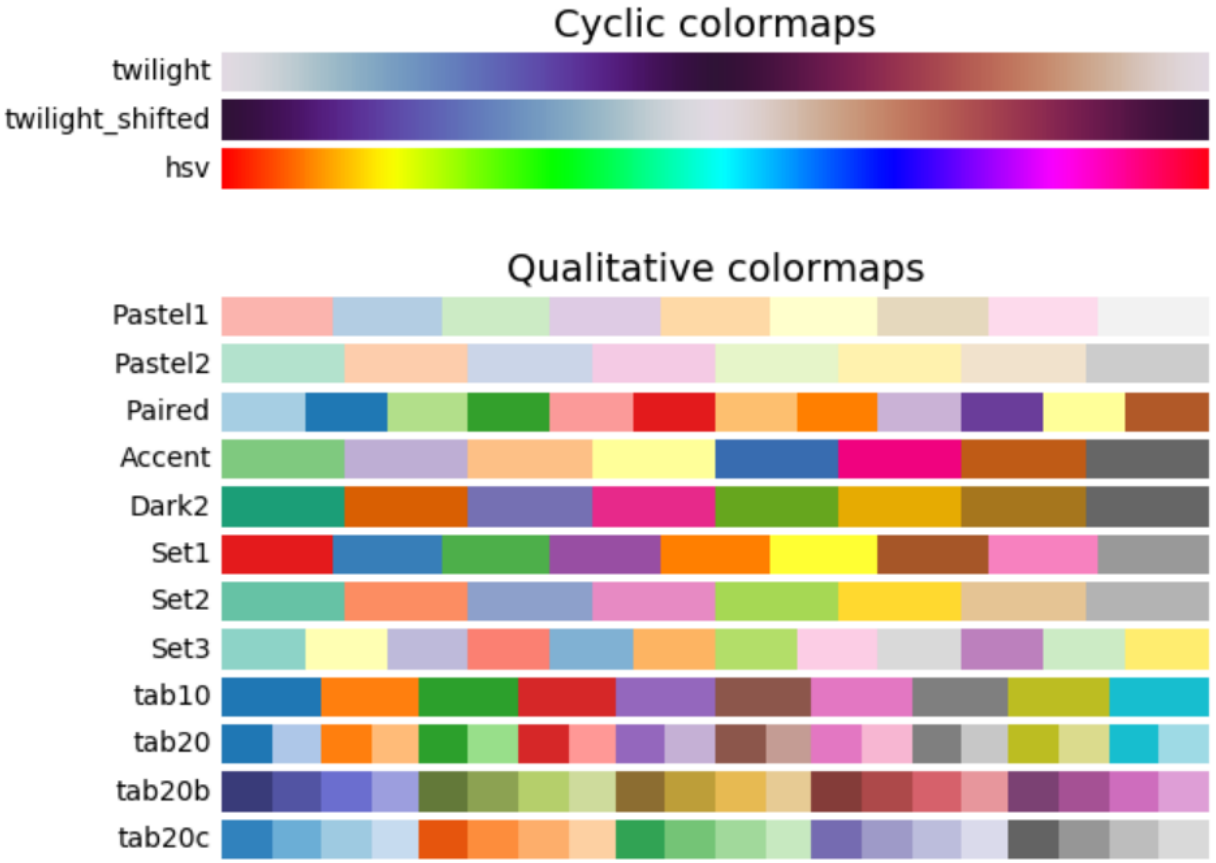


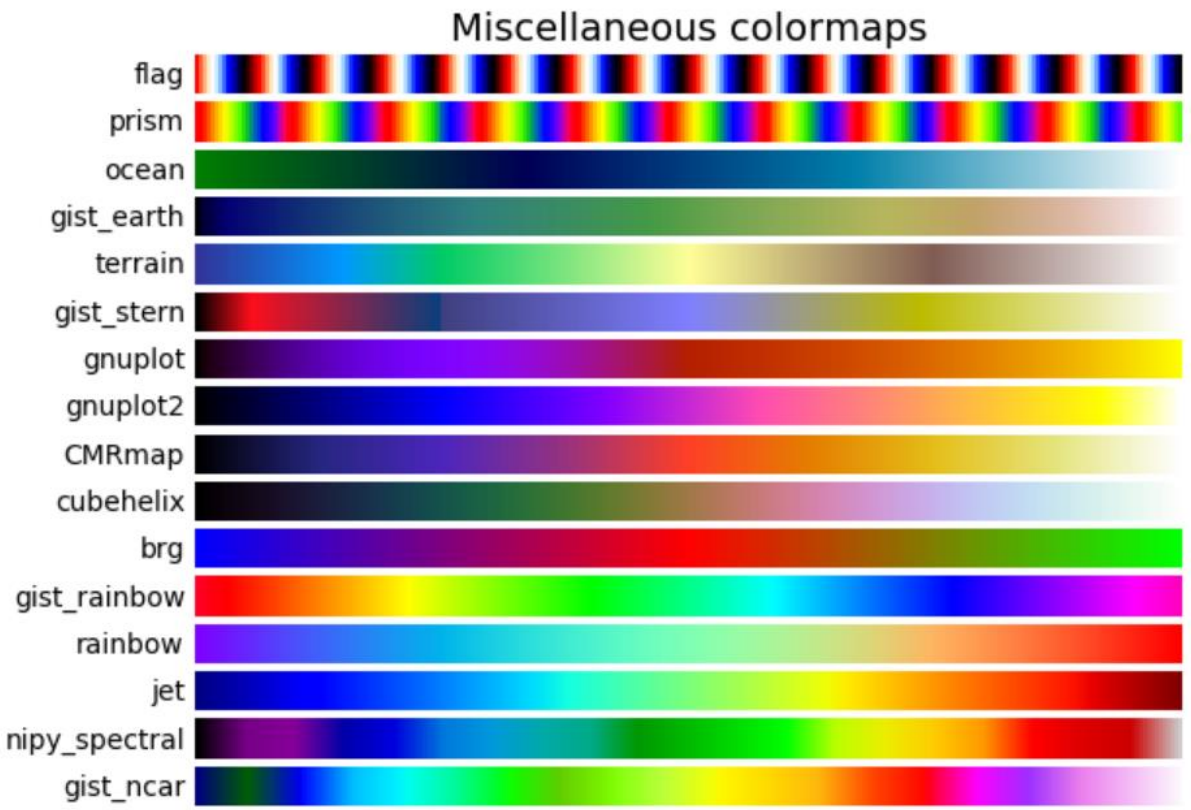
- Colormaps (ii)





- Colormaps (iii)





- Contour Plots and Filled Contour Plots (i)

Se puede usar un campo escalar para visualizar una función de dos variables, *z = f(x,y), tal que z* puede caracterizarse por sus curvas de nivel (o líneas de contorno) a lo largo de las cuales el valor de *f(x,y)* es constante.

Antes de la representación, es necesario crear una malla o matriz de puntos en el plano xy para calcular el valor de z en cada uno de ellos. Para ello, Python dispone de la función *meshgrid():*

[X, Y] = meshgrid(x,y)

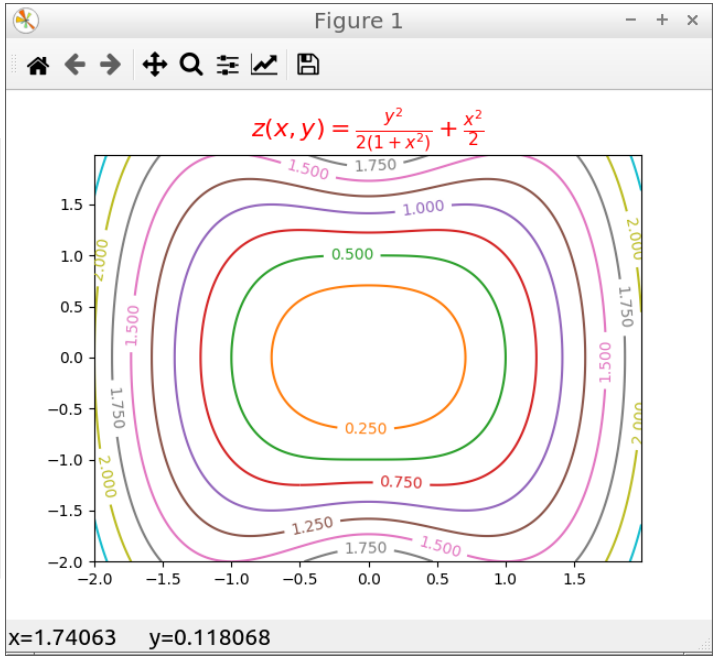
donde x e y son vectores con los valores de esta variables. X es una matriz en la que el vector x se copia en cada una de sus filas, e Y es una matriz en la que el vector y se copia en cada una de sus columnas. De esta forma, podemos trabajar con las matrices X e Y para obtener una matriz Z en términos de la función representada y representarla con una de las siguientes funciones:

contour(X, Y, Z, num_levels, atributos_formato)

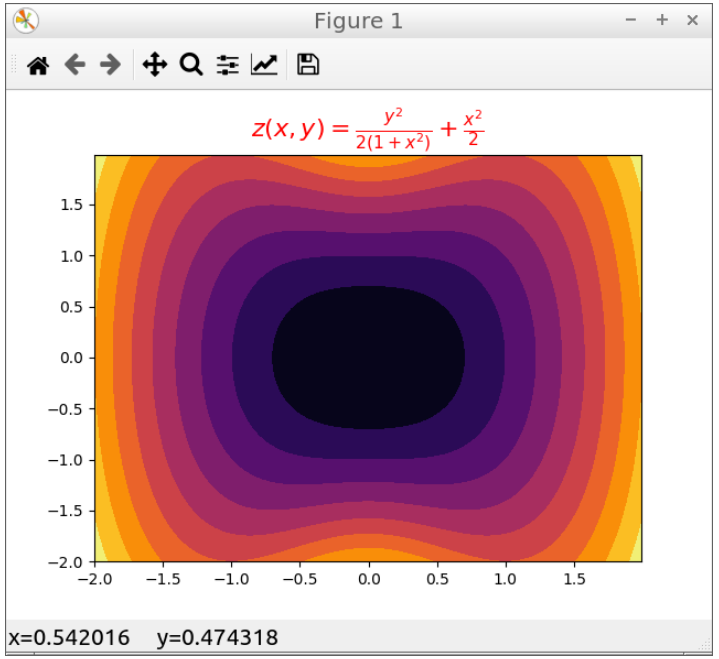
contourf(X, Y, Z, num_levels, atributos_formato)

- Contour Plots and Filled Contour Plots (ii)

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4
5 x=np.arange(-2, 2, 0.01);
6 y=np.arange(-2, 2, 0.01);
7 X,Y=np.meshgrid(x,y)
8 Z = (Y**2) / (2 * (1 + X**2)) + X**2 / 2.0
9
10 fig = plt.figure()
11 ax = fig.add_subplot(111)
12 cp = ax.contour(X, Y, Z, 10, cmap = 'tab10')
13 ax.set_title(r"$ z(x,y) = \frac{y^{2}}{2 (1 + x^{2})} + \frac{x^{2}}{2} $", color = 'r', weight = 'heavy', size = 16)
14 ax.clabel(cp, inline = True, fontsize = 10)
```




```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4
5 x=np.arange(-2, 2, 0.01);
6 y=np.arange(-2, 2, 0.01);
7 X,Y=np.meshgrid(x,y)
8 Z = (Y**2) / (2 * (1 + X**2)) + X**2 / 2.0
9
10 fig = plt.figure()
11 ax = fig.add_subplot(111)
12 cp = ax.contourf(X, Y, Z, 10, cmap = 'inferno')
13 ax.set_title(r"$ z(x,y) = \frac{y^{2}}{2 (1 + x^{2})} + \frac{x^{2}}{2} $", color = 'r', weight = 'heavy', size = 16)
```



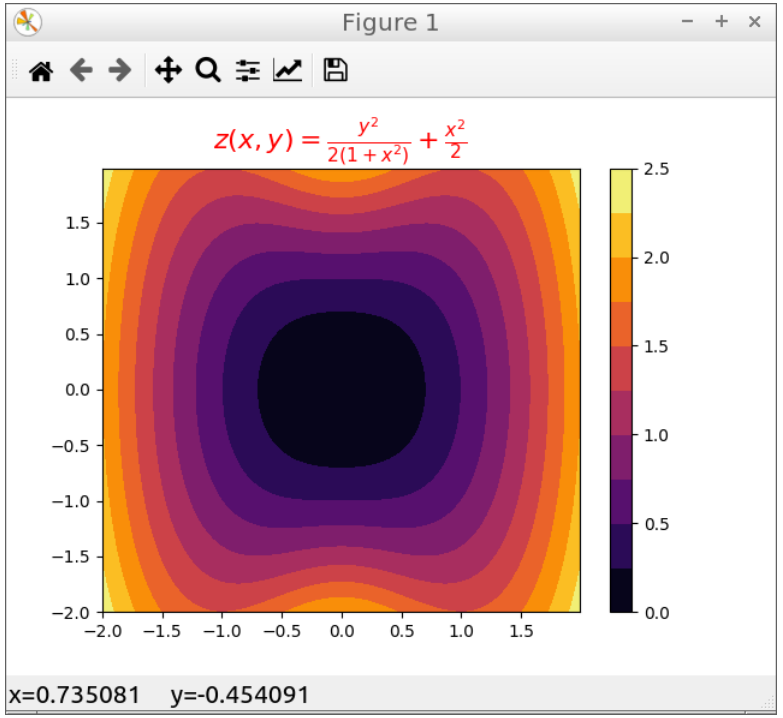
- Colorbars

También podemos añadir a la figura una colorbar a la figura a través de la siguiente función:

colorbar(objeto_origen, atributos_formato)

- *objeto_origen:* es el gráfico a partir del cual obtiene la información de los colores a trasladar a la barra de color.

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4
5 x=np.arange(-2, 2, 0.01);
6 y=np.arange(-2, 2, 0.01);
7 X,Y=np.meshgrid(x,y)
8 Z = (Y**2) / (2 * (1 + X**2)) + X**2 / 2.0
9
10 fig = plt.figure()
11 ax = fig.add_subplot(111)
12 cp = ax.contourf(X, Y, Z, 10, cmap = 'inferno')
13 ax.set_title(r"$ z(x,y) = \frac{y^{2}}{2 (1 + x^{2})} + \frac{x^{2}}{2} $", color = 'r', weight = 'heavy', size = 16)
14 fig.colorbar(cp, orientation = 'vertical')
```

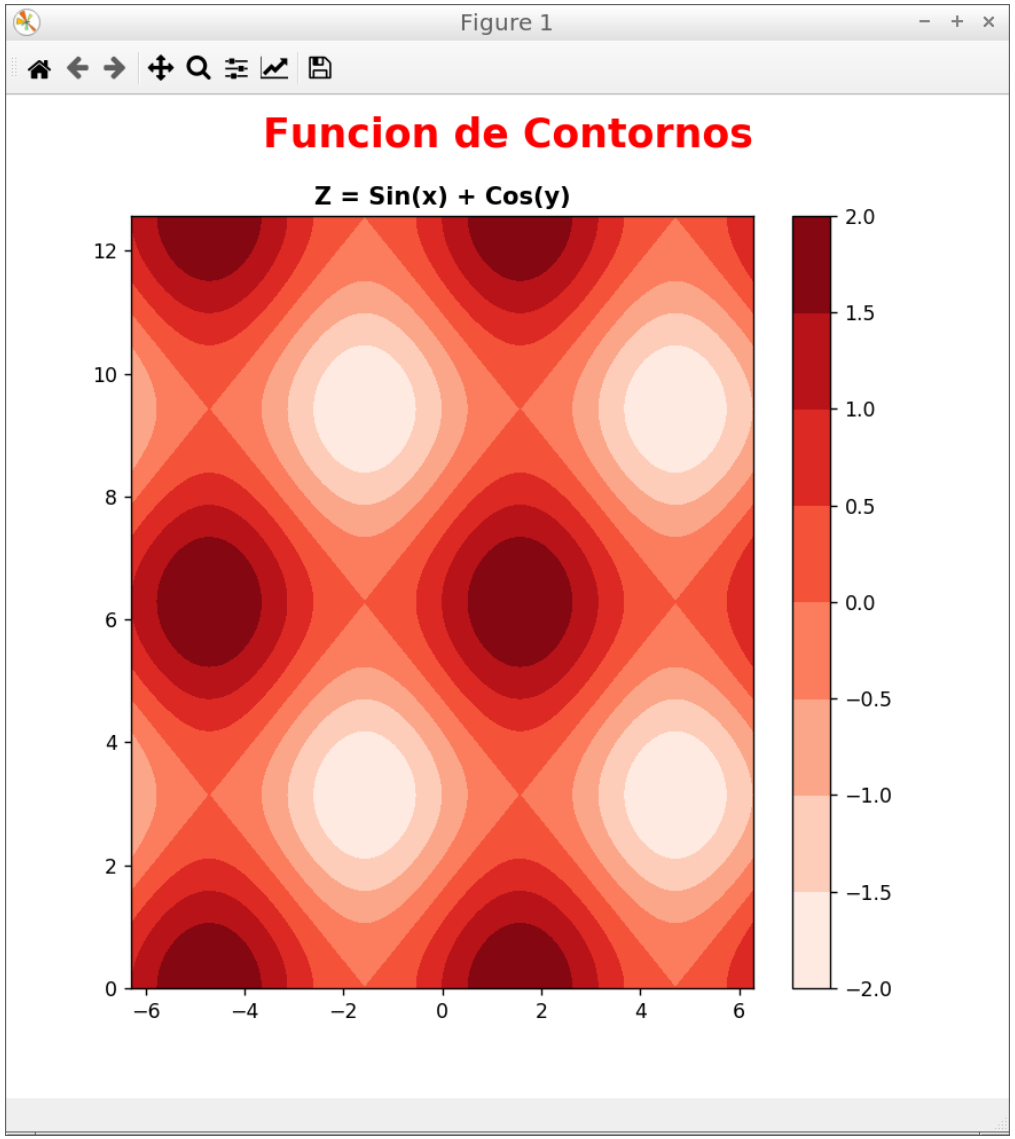


Ejercicio 9: Utilizando notación Orientada a Objetos crear una figura de tamaño 7x7 pulgadas y una resolución de 125 dpi, para representar un gráfico de contornos con relleno con los siguientes datos:

x = 1000 puntos equidistantes en el rango [-2pi, 2pi]
y = 1000 puntos equidistantes en el rango [0, 4pi]
z = sin(x) + cos(y)

El número de líneas de contorno a representar será 8.
El mapa de color será 'Reds'
Añadir una barra de color a la figura.

```
16 import matplotlib.pyplot as plt
17 import numpy as np
18
19 num_contornos = 8
20 x = np.linspace(-2 * np.pi, 2 * np.pi, 1000)
21 y = np.linspace(0, 4 * np.pi, 1000)
22 X, Y = np.meshgrid(x, y)
23 Z = np.sin(X) + np.cos(Y)
24
25 fig = plt.figure(figsize = (7,7), dpi = 125)
26 fig.suptitle('Funcion de Contornos', color = 'red', size =20, weight = 'bold')
27
28 ax = fig.add_subplot(1,1,1)
29 cp = ax.contourf(X, Y, Z, num_contornos, cmap = 'Reds')
30 ax.set_title('Z = Sin(x) + Cos(y)', weight = 'bold')
31
32 fig.colorbar(cp)
33
34 plt.show()
```



- Otros gráficos

https://matplotlib.org/3.1.1/gallery/

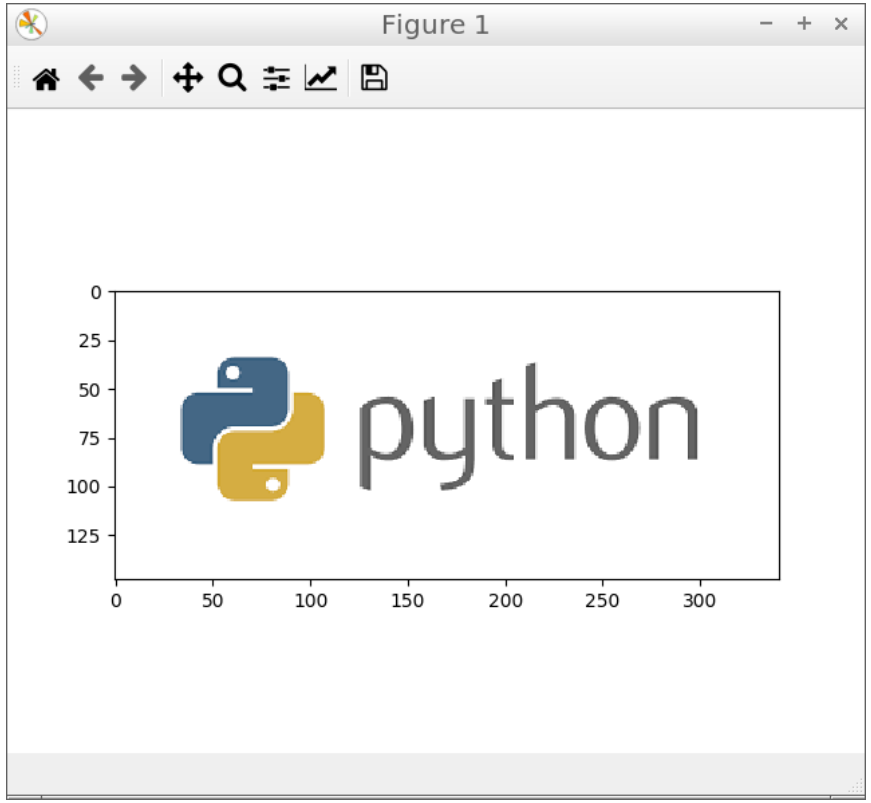
https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.pyplot.figure.html#examples-using-matplotlib-pyplot-figure

- Plotear imágenes

Pytplot nos permite mostrar imágenes de disco que tenemos guardadas en los formatos gráficos compatibles. Para ello utiliza las siguientes funciones:

- *imread(nombre_imagen):* lee la imagen indicada (incluyendo la ruta) y devuelve un array de numpy con su representación.
- *imshow(array_imagen):* plotea el array que tiene la representación numérica de la imagen.

```
1 import matplotlib.pyplot as plt
2
3 fig = plt.figure()
4 sp1 = fig.add_subplot(111)
5
6 img = plt.imread('/home/python/Descargas/python.png')
7 sp1.imshow(img)
```



- Salvar la figura en un fichero (i)

Podemos consultar la lista de formatos en los que *Pyplot* nos permite guardar nuestra figura:

- *plt.gcf().canvas.get_supported_filetypes().*
  Muestra los formatos de salida disponibles para almacenamiento de figuras.

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

In [2]: plt.gcf().canvas.get_supported_filetypes()
Out[2]:
{'ps': 'Postscript',
 'eps': 'Encapsulated Postscript',
 'pdf': 'Portable Document Format',
 'pgf': 'PGF code for LaTeX',
 'png': 'Portable Network Graphics',
 'raw': 'Raw RGBA bitmap',
 'rgba': 'Raw RGBA bitmap',
 'svg': 'Scalable Vector Graphics',
 'svgz': 'Scalable Vector Graphics',
 'jpg': 'Joint Photographic Experts Group',
 'jpeg': 'Joint Photographic Experts Group',
 'tif': 'Tagged Image File Format',
 'tiff': 'Tagged Image File Format'}
```

- *plt.gcf().canvas.get_supported_filetypes_grouped().*
  Igual que el anterior pero de forma agrupada.

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

In [2]: plt.gcf().canvas.get_supported_filetypes_grouped()
Out[2]:
{'Postscript': ['ps'],
 'Encapsulated Postscript': ['eps'],
 'Portable Document Format': ['pdf'],
 'PGF code for LaTeX': ['pgf'],
 'Portable Network Graphics': ['png'],
 'Raw RGBA bitmap': ['raw', 'rgba'],
 'Scalable Vector Graphics': ['svg', 'svgz'],
 'Joint Photographic Experts Group': ['jpeg', 'jpg'],
 'Tagged Image File Format': ['tif', 'tiff']}
```

- Salvar la figura en un fichero (ii)

La función de *Pyplot* que nos permite almacenar la figura en disco es *savefig()*. Entre sus múltiples parámetros, los más utilizados son los siguientes:

- *fname:* nombre del fichero donde almacenar la figura.

- *dpi (dots per inch):* se corresponde con la resolución que le vamos a dar.

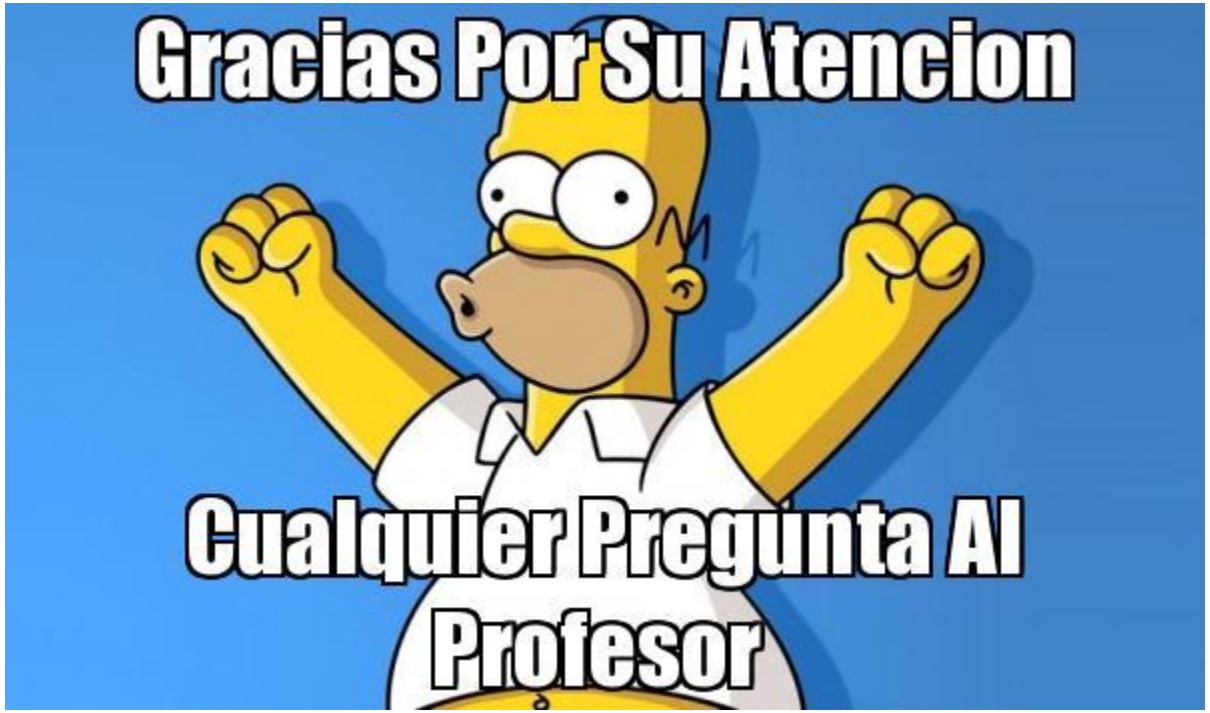
- *format:* cadena de texto con el formato de salida del archivo

- *transparent:* es útil si la figura la queremos poner sobre un objeto con color de fondo (p.e. en páginas web).

```
1  import matplotlib.pyplot as plt
2  import numpy as np
3
4  x=np.arange(-2, 2, 0.01);
5  y=np.arange(-2, 2, 0.01);
6  X,Y=np.meshgrid(x,y)
7  Z = (Y**2) / (2 * (1 + X**2)) + X**2 / 2.0
8
9  plt.ioff()
10
11 fig = plt.figure()
12 ax = fig.add_subplot(111)
13 cp = ax.contourf(X, Y, Z, 10, cmap = 'inferno')
14 ax.set_title(r"$ z(x,y) = \frac{y^{2}}{2 (1 + x^{2})} + \frac{x^{2}}{2} $", color = 'r', weight = 'heavy', size = 16)
15 fig.colorbar(cp, orientation = 'vertical')
16
17 plt.savefig('contourf.png', dpi = 200, format = 'png')
```

Ejercicio 10: Modificar el código del ejercicio 9 para que en lugar de plotear la imagen en pantalla se almacene en un fichero denominado *ej10.png* en formato png con color de fondo transparente.

```
 8 import matplotlib.pyplot as plt
 9 import numpy as np
10
11 plt.ioff()
12
13 num_contornos = 8
14 x = np.linspace(-2 * np.pi, 2 * np.pi, 1000)
15 y = np.linspace(0, 4 * np.pi, 1000)
16 X, Y = np.meshgrid(x, y)
17 Z = np.sin(X) + np.cos(Y)
18
19 fig = plt.figure(figsize = (7,7), dpi = 125)
20 fig.suptitle('Funcion de Contornos', color = 'red', size =20, weight = 'bold')
21
22 ax = fig.add_subplot(1,1,1)
23 cp = ax.contourf(X, Y, Z, num_contornos, cmap = 'Reds')
24 ax.set_title('Z = Sin(x) + Cos(y)', weight = 'bold')
25
26 fig.colorbar(cp)
27
28 plt.savefig('ej10.png', format = 'png', transparent = True)
```


Gracias Por Su Atencion
Cualquier Pregunta Al Profesor