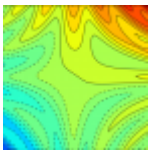


Tutorial de matplotlib.pyplot

Por pybonacci.wordpress.com

Versión 0.1: 31/08/2012

Dedicado a la memoria de John D. Hunter (creador de Matplotlib): 1968-2012

Contenidos:

Acerca de Pybonacci	**4**
Introducción	**5**
1.Primeros pasos	**5**
2.Creando y manejando ventanas y configurando la sesión	**8**
3.Configuración del gráfico	**15**
4.Tipos de gráfico (I)	**21**
5. Tipos de gráfico (II)	**28**
6. Tipos de gráfico (III)	**34**
7. Tipos de gráfico (IV)	**41**
8.Texto y anotaciones	**46**
9.Miscelánea	**51**

Acerca de pybonacci

¡Hola a todos! Somos unos apasionados de Python que nos hemos decidido a rellenar el que pensamos que es un hueco importante en la blogosfera hispana: el uso de Python para aplicaciones científicas.

Nuestra intención es escribir con regularidad sobre cómo podemos utilizar este lenguaje de programación fantástico que es Python para resolver problemas en ciencia e ingeniería, utilizando librerías como NumPy, SciPy, matplotlib, SymPy y muchas más. También traduciremos artículos escritos en otros idiomas que nos resulten interesantes, mostraremos pequeñas recetas…

Vuestra colaboración es muy importante: estaremos deseando que nos indiquéis en los comentarios qué os han parecido los artículos, si habéis tenido algún problema, cómo podríamos mejorar, que nos hagáis sugerencias… Y, por supuesto, si creéis que podéis hacer un aporte de mayor tamaño podéis contactar con nosotros para que publiquemos vuestro artículo.

¡Un saludo, y nos vemos en Pybonacci!


http://pybonacci.wordpress.com/


http://twitter.com/Pybonacci


http://plus.google.com/10462363400652925...


http://www.youtube.com/user/Pybonacci


http://www.facebook.com/Pybonacci

Introducción

Esto pretende ser un tutorial del módulo pyplot de la librería matplotlib. El tutorial lo dividiremos de la siguiente forma:

1. Primeros pasos
2. Creando ventanas, manejando ventanas y configurando la sesión
3. Configuración del gráfico
4. Tipos de gráfico I
5. Tipos de gráfico II
6. Tipos de gráfico III
7. Tipos de gráfico IV
8. Texto y anotaciones (arrow, annotate, table, text…)
9. Miscelánea

[Para este tutorial se ha usado python 2.7.1, ipython 0.11, numpy 1.6.1 y matplotlib 1.1.0, además, en algunos capítulos se ha usado el mpl_toolkit Basemap 1.0.2 y netcdf4-python 0.9.9 para leer ficheros en formato netcdf o grib]

[DISCLAIMER: Muchos de los gráficos que vamos a representar no tienen ningún sentido físico y los resultados solo pretenden mostrar el uso de la librería].

A lo largo de todo el tutorial, todo lo que sea código vendrá escrito con el siguiente formato:

```
Esto es código python
```

En todo momento supondremos que se ha iniciado la sesión y se ha hecho

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

Todos los enlaces en el documento dirigen a la web y no al propio documento.

1. Primeros Pasos

Para empezar diremos que hay tres formas de usar la librería Matplotlib:

- La podemos usar desde python usando el módulo pylab. El módulo pylab pretende mostrar un entorno de trabajo parecido al de matlab mezclando las librerías numpy y matplotlib. Es la forma menos pythónica de usar matplotlib y se obtiene usando

```
from pylab import *
```

Normalmente solo se recomienda para hacer pruebas rápidas desde la línea de comandos.

- Una segunda forma, que es la que veremos en este tutorial, es usando el módulo pyplot.

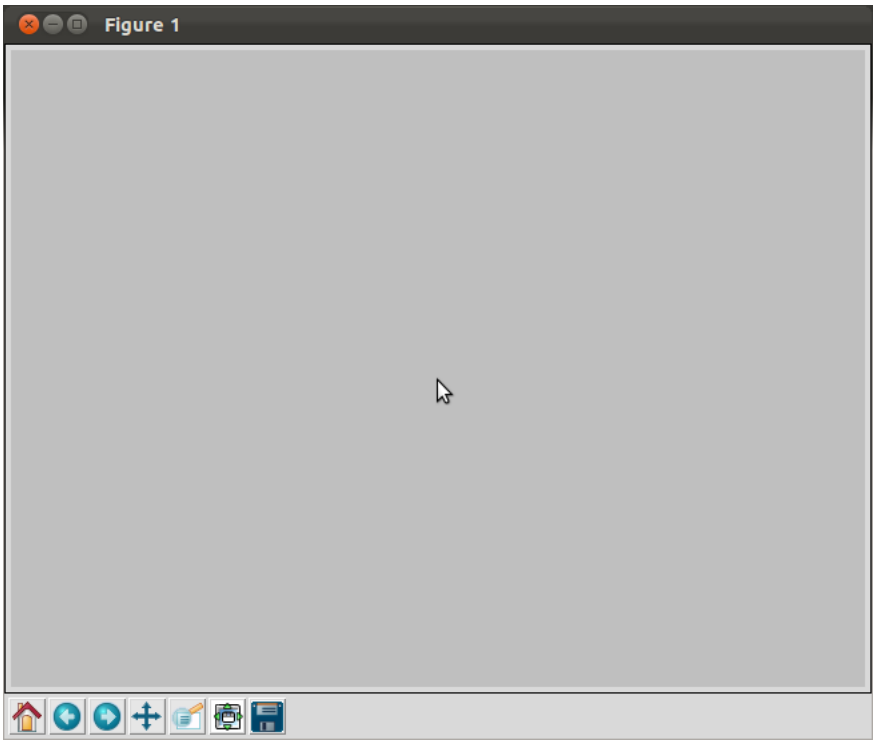
```
import matplotlib.pyplot as plt
```

- Por último, la forma más recomendable y pythónica, pero más compleja, sería usar matplotlib mediante la interfaz orientada a objetos. Cuando se programa con matplotlib, no mientras se trabaja interactivamente, esta es la forma que permite tener más control sobre el código. Quizá veamos esto en el futuro si alguno nos animamos/os animáis a escribir sobre ello.

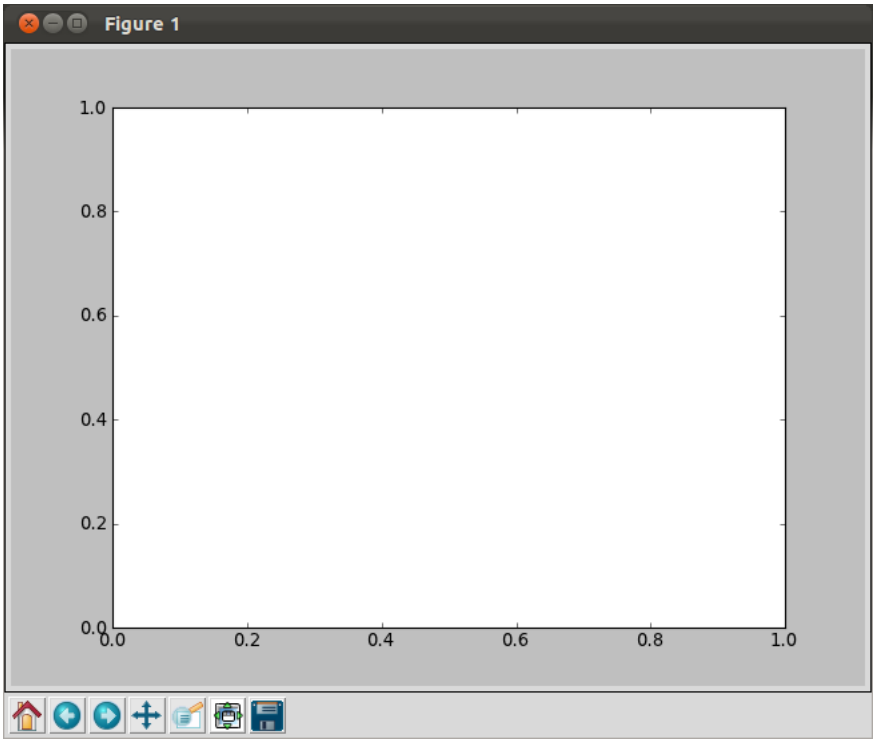
Absolutamente todo lo que vamos a usar en este tutorial y que está relacionado con matplotlib.pyplot lo podréis encontrar documentado y detallado aquí. Como he comentado, todo lo que vamos a ver está en el anterior enlace, pero no todo lo que está en el anterior enlace lo vamos a ver. Por ejemplo, en el índice veréis que he tachado los puntos 9 y 10, las funciones estadísticas y las funciones que permiten meter algo de interactividad en los gráficos dentro de pyplot. Las funciones estadísticas incluidas son pocas, algunas son complejas y muy específicas y las veo poco coherentes como grupo dentro de pyplot, para ello ya tenemos scipy y estas funciones estarían mejor ahí para separar lo que es 'gráficar' (en español de Sudámerica existe la palabra) de lo que es analizar datos. Para interactividad con los gráficos tenemos el módulo matplotlib.widgets, muchísimo más completo.

Para que quede claro desde un principio, las dos zonas principales donde se dibujaran cosas o sobre las que se interactuará serán:

- figure, que es una instancia de matplotlib.figure.Figure. Y es la ventana donde irá el o los gráficos en sí:



- axes, que es una instancia de matplotlib.axes.Axes, que es el gráfico en sí donde se dibujará todo lo que le digamos y está localizada dentro de una figure.



Para lo primero (figure) usaremos la palabra 'ventana' mientras que para lo segundo (axes) usaremos la palabra 'gráfico'.

Si quieres puedes pasar a la siguiente sección.

2. Creando y manejando ventanas y configurando la sesión

Como ya comentamos anteriormente, el módulo pyplot de matplotlib se suele usar para hacer pruebas rápidas desde la línea de comandos, programitas cortos o programas donde los gráficos serán, en general, sencillos.

Normalmente, cuando iniciamos la sesión, esta no está puesta en modo interactivo. En modo interactivo, cada vez que metemos código nuevo relacionado con el gráfico o la ventana (recordad, una instancia de matplotlib.axes.Axes o de matplotlib.figure.Figure, respectivamente), este se actualizará. Cuando no estamos en modo interactivo, el gráfico no se actualiza hasta que llamemos a show() (si no hay una ventana abierta) o draw() (normalmente no lo usaréis para nada) explícitamente. Veamos cómo es esto.

Si acabamos de iniciar sesión deberíamos estar en modo no interactivo. Para comprobarlo hacemos lo siguiente:

```
plt.isinteractive()
```

Si el resultado es *False* significa que estamos en modo no interactivo. Esto significa que si hacemos lo siguiente:

```
plt.plot([1,2,3,4,5])
```

No lanzará una ventana hasta que lo pidamos explícitamente mediante:

```
plt.show()
```

Podemos conmutar a modo interactivo o no usando plt.ion() y plt.ioff(), que lo que hacen es poner el modo interactivo en 'on' o en 'off', respectivamente. Como está en off (recordad que plt.isinteractive() nos ha dado False, lo que significa que está en 'off'), si ahora hacemos lo siguiente (cerrad antes cualquier ventana de gráficos que tengáis abierta):

```
plt.ion()
plt.plot([1,2,3,4])
```

Vemos que directamente se abre una ventana nueva sin necesidad de llamar a plt.show(). Yo suelo usar ipython así para ir probando cosas y cuando ya acierto con como quiero que me salgan los gráficos voy a spyder, donde tengo el programa que esté haciendo, y ya escribo el código que necesito con la interfaz orientada a objetos.

Jugad un poco con plt.isinteractive(), plt.ion(), plt.ioff(), plt.show() y plt.draw() para estar más familiarizados con el funcionamiento.

Lo siguiente que veremos es plt.hold() y plt.ishold(). plt.hold es un conmutador para decir si queremos que los gráficos se sobreescriban, que en el mismo gráfico tengamos diferentes gráficas representadas, o para que el gráfico se limpie y se dibuje la nueva

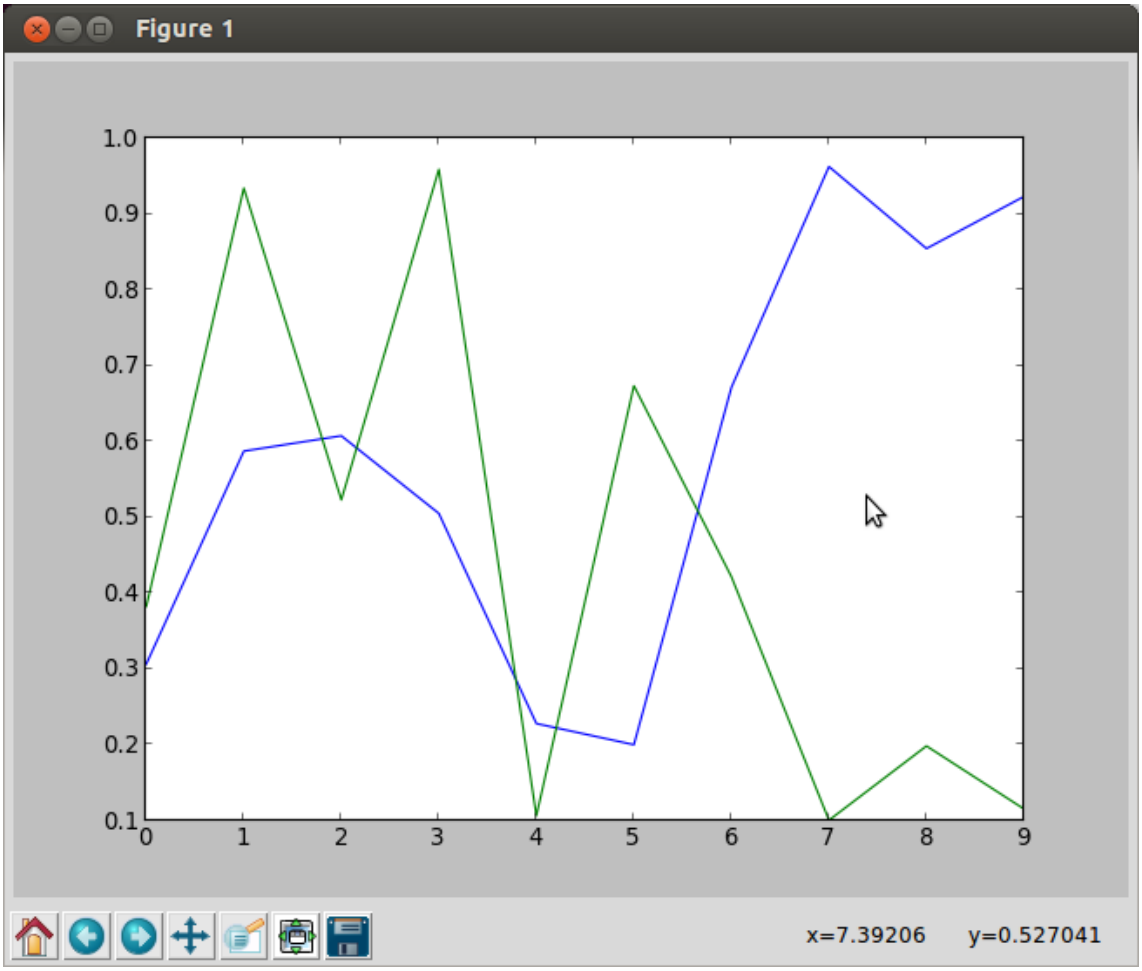
gráfica cada vez. Si usamos plt.ishold() nos 'responderá' True o False. Si acabáis de iniciar sesión, normalmente estará en True.

```
plt.ishold()
```

Si está en *True*, hacemos lo siguiente:

```
plt.plot(np.random.rand(10))
plt.plot(np.random.rand(10))
plt.show()
```

Obtendremos lo siguiente:



Si el modo 'hold' estuviera en *False*, solo se habría conservado el último plot y solo veríamos una línea de las dos (probadlo usando plt.hold() y plt.ishold()).

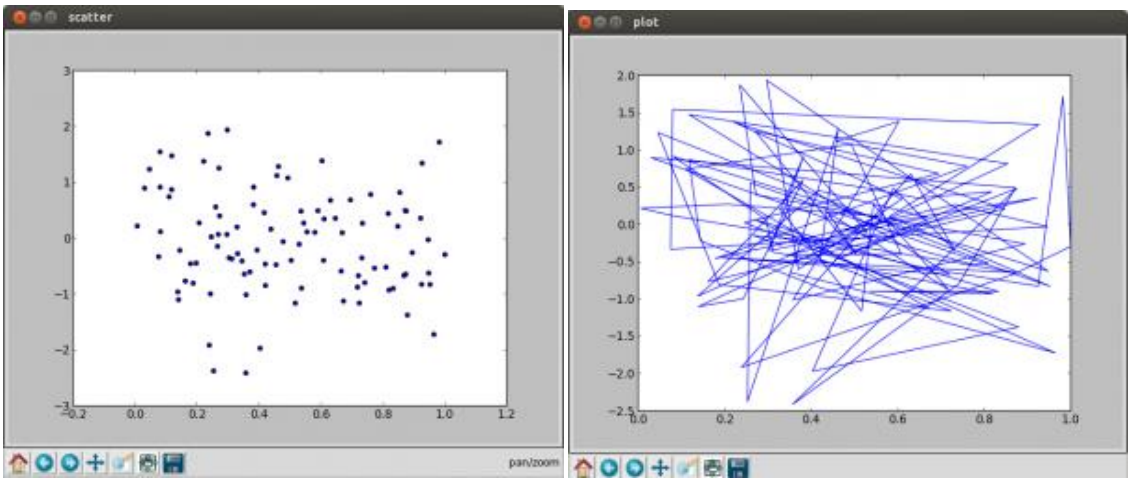
Si estamos en modo interactivo (plt.ion()) y queremos borrar todos los gráficos (matplotlib.axes.Axes), títulos, …, de la ventana (matplotlib.figure.Figure) podemos usar plt.clf() y nos volverá a dejar el 'lienzo' limpio.

Si seguimos en modo interactivo (plt.ion()) y queremos cerrar la ventana podemos usar plt.close().

Imaginaos que ahora queréis trabajar con varias ventanas de gráficos simultáneamente donde en una dibujáis unos datos y en la otra otro tipo de datos y los queréis ver simultáneamente. Podemos hacer esto dándole nombre (o número) a las ventanas con las que vamos a trabajar. Veamos un ejemplo:

```
plt.figure('scatter') # Crea una ventana titulada 'scatter'

plt.figure('plot') # Crea una ventana titulada 'plot'

a = np.random.rand(100) # Generamos un vector de valores aleatorios

b = np.random.rand(100) # Generamos otro vector de valores aleatorios

# Le decimos que la ventana activa en la que vamos a dibujar
# es la ventana 'scatter'
plt.figure('scatter')

plt.scatter(a,b) # Dibujamos un scatterplot en la ventana 'scatter'

plt.figure('plot') # Ahora cambiamos a la ventana 'plot'

plt.plot(a,b)
```

Y os quedaría algo como lo siguiente:

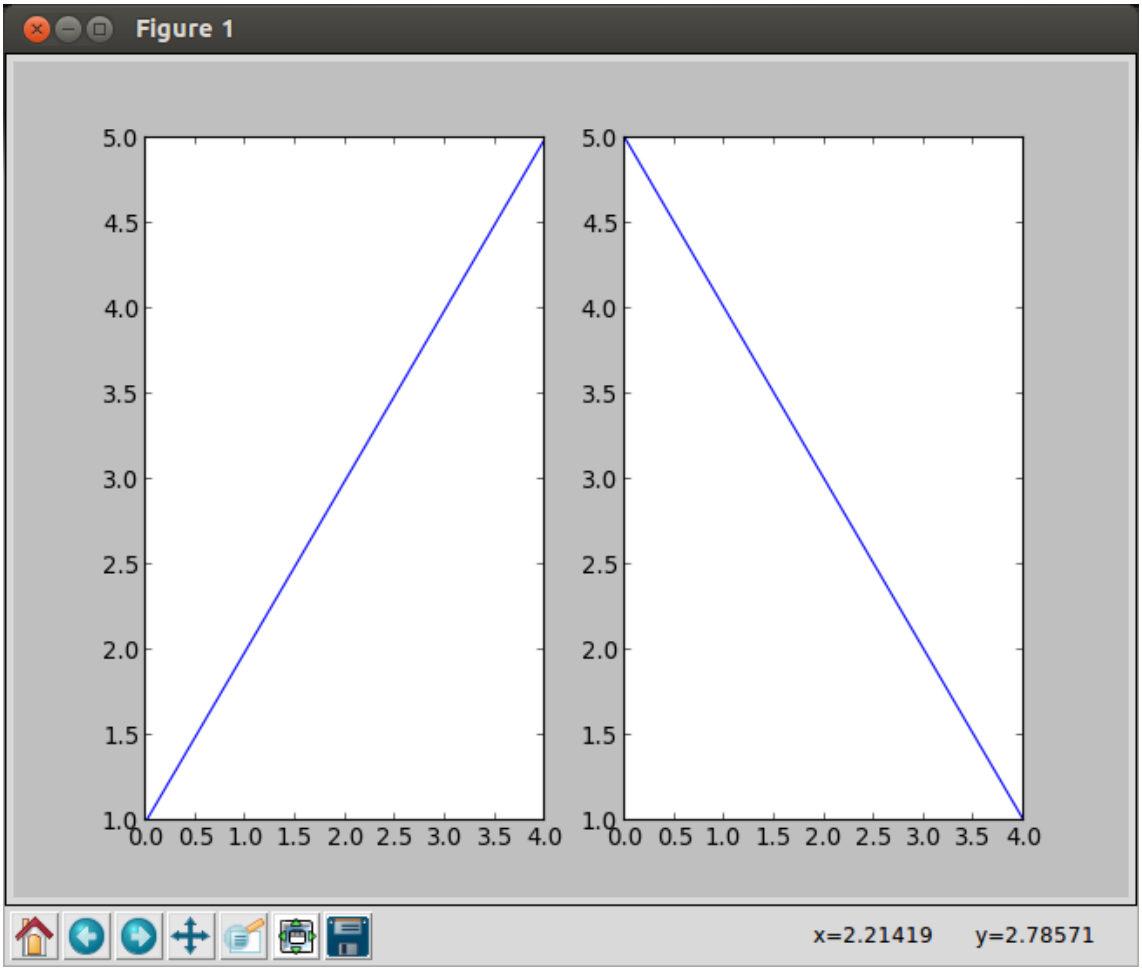


Es decir, podemos ir dibujando en varias ventanas a la vez. Podéis probar a cerrar una de las dos ventanas, limpiar la otra, crear una nueva,… Haciendo una llamada a plt.figure() también podemos definir la resolución del gráfico, el tamaño de la figura,…

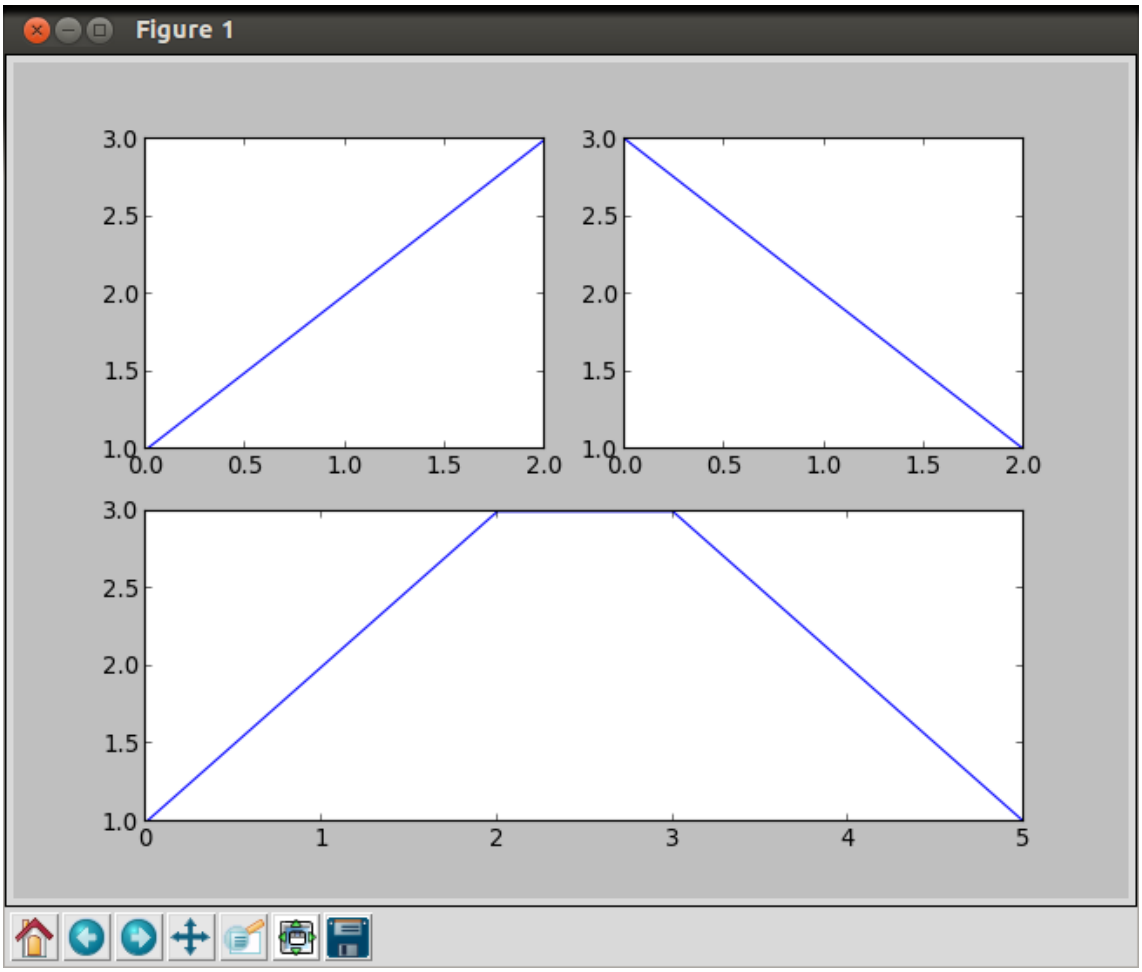
Pero yo no quiero dibujar los gráficos en dos ventanas, yo quiero tener varios gráficos en la misma. Perfecto, también podemos hacer eso sin problemas con la ayuda de plt.subplot(). Con plt.subplot() podemos indicar el número de filas y columnas que corresponderán a como dividimos la ventana. En el siguiente ejemplo se puede ver dos áreas de gráfico en la misma ventana:

```
plt.ion() # Nos ponemos en modo interactivo

# Dividimos la ventana en una fila y dos
# columnas y dibujamos el primer gráfico
plt.subplot(1,2,1)

plt.plot((1,2,3,4,5))

# Dividimos la ventana en una fila y dos
# columnas y dibujamos el segundo gráfico
plt.subplot(1,2,2)

plt.plot((5,4,3,2,1))
```

Obteniendo el siguiente gráfico:

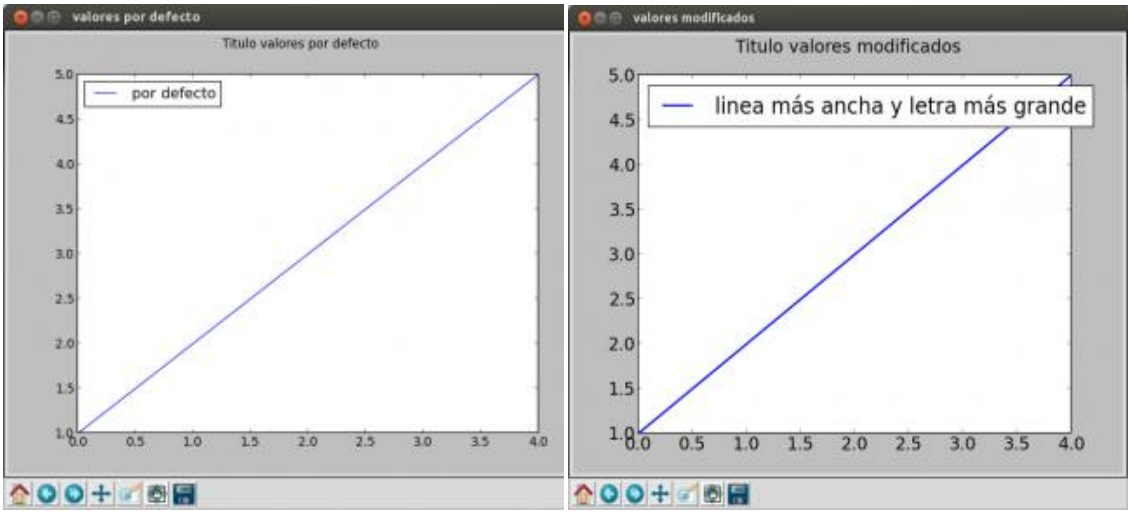


Os dejo como ejercicio ver como podéis conseguir la siguiente gráfica (si no sabéis como dejad un comentario) y con ello creo que habréis entendido perfectamente el uso de plt.subplot():



Por último, vamos a ver como configurar la sesión para ahorrarnos escribir código de más. Por ejemplo, imaginaos que queréis que todas las líneas sean más gruesas por defecto porque os gustan más así, que queréis usar otro tipo de fuente sin escribirlo explícitamente cada vez que hacéis un gráfico, que los gráficos se guarden siempre con una resolución superior a la que viene por defecto,… Para ello podéis usar plt.rc(), plt.rcParams, plt.rcdefaults(). En este caso vamos a usar plt.rc(), podréis encontrar más información sobre como configurar matplotlib en este enlace. Veamos un ejemplo para ver cómo funciona todo esto:

```
plt.ion() # Nos ponemos en modo interactivo

# Creamos una ventana donde dibujamos el gráfico con la
# configuración por defecto
plt.figure('valores por defecto')

# Esto sirve para poner título dentro de la ventana
plt.suptitle('Titulo valores por defecto')

plt.plot((1,2,3,4,5), label = 'por defecto') # Hacemos el plot

# Colocamos la leyenda en la esquina superior izquierda
plt.legend(loc = 2)

# A partir de aquí todas las líneas que dibujemos irán con ancho doble
plt.rc('lines', linewidth = 2)

# A partir de aquí las fuentes que aparezcan en cualquier gráfico en
# la misma sesión tendrán mayor tamaño

plt.rc('font', size = 18)

# Creamos una ventana donde dibujamos el gráfico con la
# configuración por defecto
plt.figure('valores modificados')

# Esto sirve para poner título dentro de la ventana
plt.suptitle('Titulo valores modificados')

# Hacemos el plot
plt.plot((1,2,3,4,5),
         label = u'linea más ancha y letra más grande')

# Colocamos la leyenda en la esquina superior izquierda
plt.legend(loc = 2)
```



Después de usar plt.rc() para modificar un parámetro esa modificación será para toda la sesión a no ser que lo volvamos a modificar explícitamente o a no ser que usemos plt.rcdefaults(), que devolverá todos los parámetros a los valores por defecto.

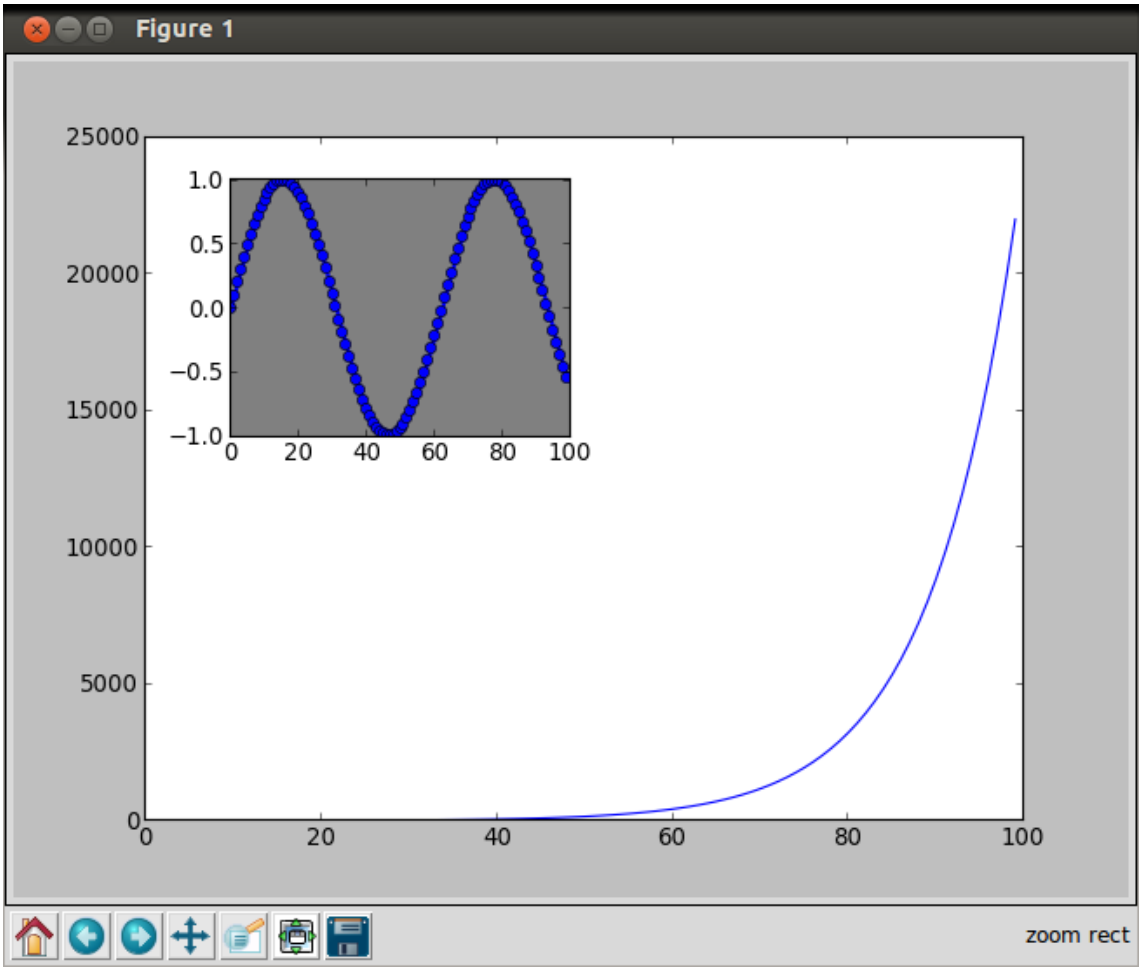
Si no has visto el primer capítulo de esta serie échale un ojo ahora o, si prefieres, puedes pasar a la siguiente parte.

3. Configuración del gráfico

Hasta ahora hemos visto como podemos configurar la ventana y la sesión, en esta ocasión nos vamos a centrar en configurar el área del gráfico. Para ello vamos a empezar con plt.axes(), que sirve para 'llamar' y/o configurar a un área de gráfico. Podemos definir la posición, el tamaño, el color del área del fondo,…:

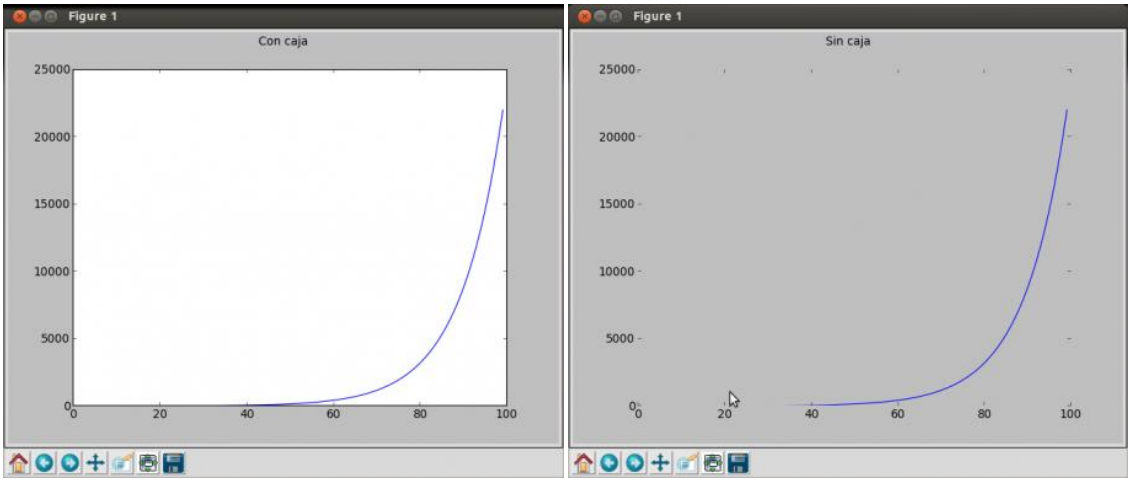
```
plt.ion() # Ponemos la sesión como interactiva si no está como tal

plt.axes() # Coloca un área de gráfico con los valores por defecto

# Dibuja una exponencial de 0 a 10
plt.plot(np.exp(np.linspace(0,10,100)))

# Dibuja una nueva área de gráfica colocada y con ancho y largo
# definido por [0.2,0.55,0.3,0.3] y con gris como color de fondo
plt.axes([0.2,0.55,0.3,0.3], axisbg = 'gray')

plt.plot(np.sin(np.linspace(0,10,100)), 'b-o', linewidth = 2)
```

El resultado es el siguiente:



Como podéis imaginar, podemos usar plt.axes() como sustituto de plt.subplot() si queremos dibujar gráficos que no tengan que tener una forma 'regular' dentro de la

ventana. Si ahora queremos borrar el área del gráfico podemos usar plt.delaxes(), si queremos borrar el contenido que hay en el área del gráfico podemos usar plt.cla() y si queremos que no aparezca la 'caja' donde se dibuja el gráfico podemos usar plt.box() (si no hay 'caja' y queremos que aparezca podemos llamar a plt.box() y volverá a aparecer la 'caja').

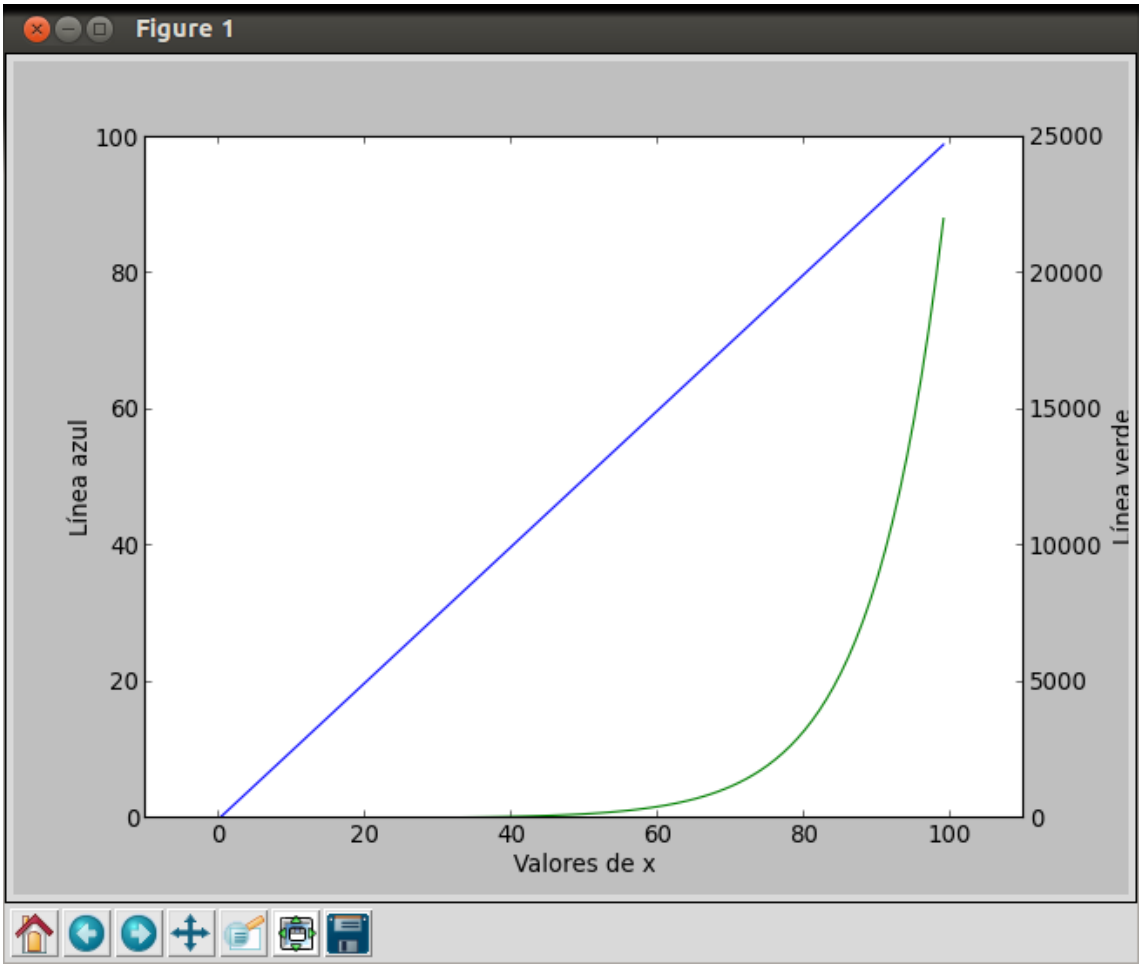


El área del gráfico puede ser un área rectangular o un área para un gráfico polar (ver ejemplo).

Podemos colocar una rejilla que nos ayude a identificar mejor las áreas del gráfico mediante plt.grid() (en un gráfico polar deberemos usar plt.rgrid() y plt.thetagrids()).

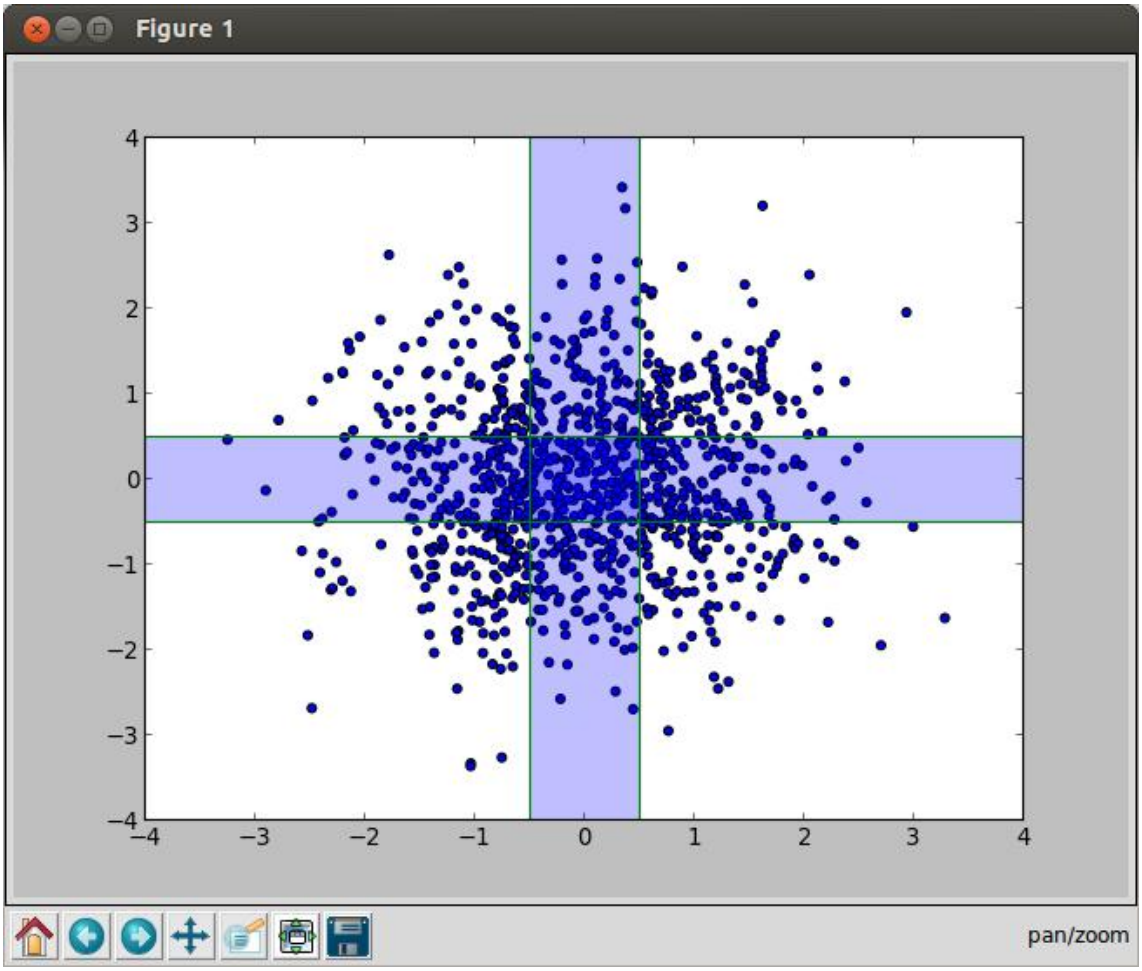
Si os habéis fijado, matplotlib dibuja los ejes de forma que se ajusten al gráfico pero quizá eso no es lo que nos interese en algunos momentos, para ello podemos hacer uso de plt.axis(). Nos permite definir la longitud de los ejes, si queremos que aparezcan los mismos, si queremos que estos estén escalados,… Si solo nos interesa configurar uno de los ejes y dejar que el otro lo maneje matplotlib podemos usar plt.xlim(), plt.xscale(), plt.ylim() y plt.yscale(). Si queremos dejar el eje x o el eje y con escala logarítmica podemos usar, respectivamente, plt.semilogx() o plt.semilogy(). Podemos dibujar un segundo eje x o un segundo eje y usando plt.twinx() o plt.twiny, respectivamente. También podemos establecer unos márgenes alrededor de los límites de los ejes usando plt.margins(). Por último, podemos etiquetar nuestros ejes con plt.xlabel() y plt.ylabel(). Veamos un ejemplo de algunas de estas cosas:

```
plt.ion()

plt.plot(np.arange(100), 'b') # Dibujamos una línea recta azul

plt.xlabel('Valores de x') # Ponemos etiqueta al eje x

plt.ylabel(u'Línea azul') # Ponemos etiqueta al eje y

plt.twinx() # Creamos un segundo eje y

# Dibuja una exponencial de 0 a 5 con la y representada en el
# segundo eje y
plt.plot(np.exp(np.linspace(0,5,100)), 'g')

plt.ylabel(u'Línea verde') # Ponemos etiqueta al segundo eje y

# Limitamos los valores del eje x para que vayan desde -10 a 110
plt.xlim(-10,110)
```



Ahora vamos a ver plt.axvline(), plt.axvspan(), plt.axhline(), plt.axhspan(). ¿Y para qué sirven estas 'cosas'? Pensad que, por ejemplo, queréis resaltar una zona de vuestro gráfico para focalizar la atención en esa área. Eso lo podríamos hacer usando lo anterio. plt.axvline() y plt.axhline() dibujan líneas verticales y horizontales en la x o en la y que le digamos mientras que plt.axvspan y plt.axhspan dibujan recuadros entre las coordenadas x o y que queramos, respectivamente.

```
plt.ion() # Ponemos el modo interactivo

# Dibujamos un scatterplot de valores aleatorios
plt.scatter(np.random.randn(1000),np.random.randn(1000))

# Dibujamos una línea vertical verde centrada en x = -0.5
plt.axvline(-0.5, color = 'g')

# Dibujamos una línea vertical verde centrada en x = 0.5
plt.axvline(0.5, color = 'g')

# Dibujamos una línea horizontal verde centrada en x = -0.5
plt.axhline(-0.5, color = 'g')

# Dibujamos una línea horizontal verde centrada en x = 0.5
plt.axhline(0.5, color = 'g')

# Dibujamos un recuadro azul vertical entre x[-0.5,0.5]
# con transparencia 0.25
plt.axvspan(-0.5,0.5, alpha = 0.25)

# Dibujamos un recuadro azul horizontal entre x[-0.5,0.5]
# con transparencia 0.25
plt.axhspan(-0.5,0.5, alpha = 0.25)
```

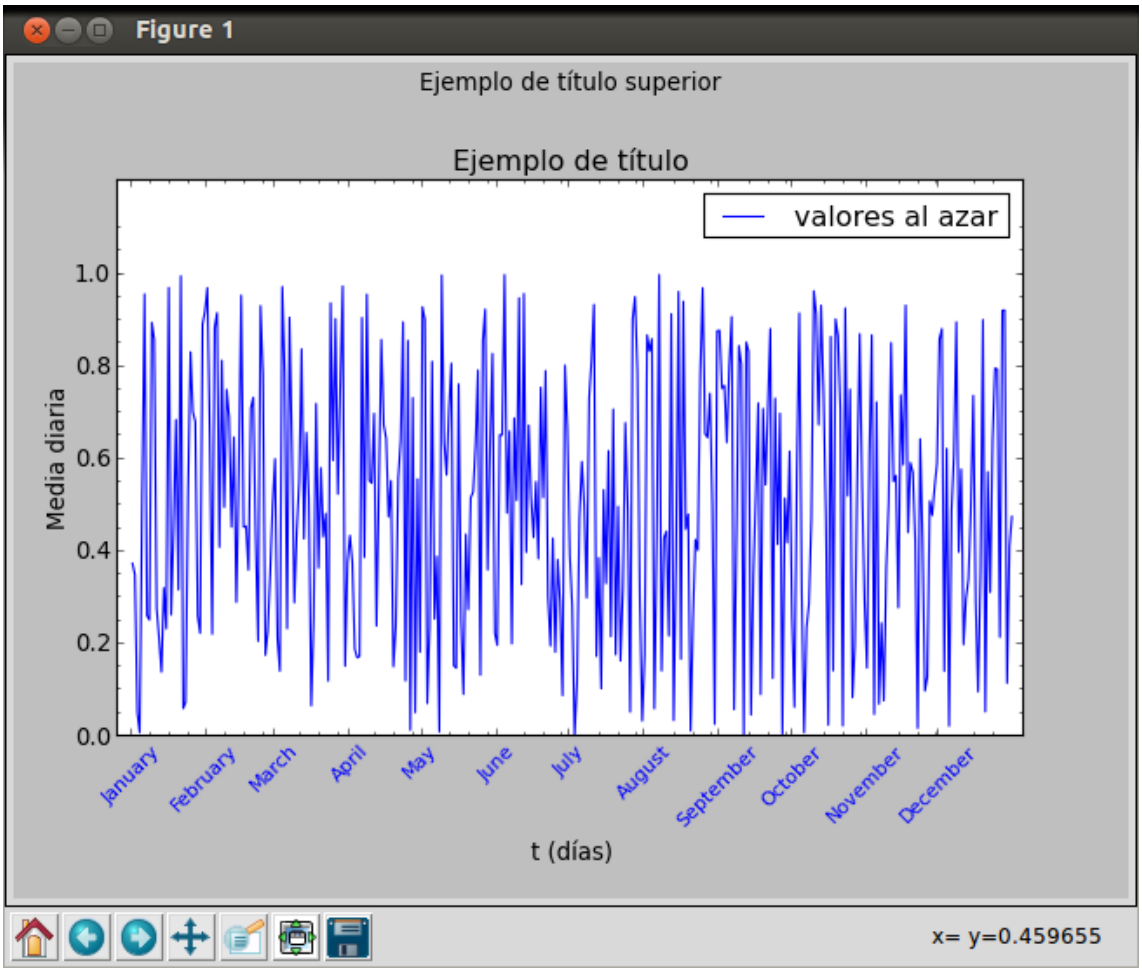


¿Y cómo podemos controlar el texto básico sobre el gráfico? Hay muchas formas de meter texto y controlar las etiquetas de forma básica y sencilla. En algunos momentos

hemos visto plt.legend(), también existe plt.figlegend(). Yo siempre uso plt.legend() el 100% de las veces. Para usos avanzados podéis mirar este enlace y este otro enlace. Si queremos poner un título al gráfico podemos usar plt.title() y plt.suptitle(). Si queremos poner título a los ejes podemos usar plt.xlabel() y plt.ylabel() para los ejes x e y, respectivamente. Por último, para controlar los valores de las etiquetas que se ponen sobre los ejes dispones de plt.locator_params(), plt.minorticks_on(), plt.minorticks_off(), plt.tick_params(), plt.tick_label_format(), plt.xticks() y plt.yticks(). Vamos a manejar la mayor parte de estas funciones mediante un ejemplo para que se vea más claro su uso. Imaginemos que queremos representar el valor medio diario de una variable durante un año, en el eje x queremos que aparezca solo los meses en el día del año en que empieza el mes

```
plt.ion() # Ponemos modo interactivo

import calendar

# Para generar el lugar del primer días de cada mes en un año
dias = [np.array(calendar.mdays)[0:i].sum()+ 1 for i in
np.arange(12)+1]

# Creamos una lista con los nombres de los meses
meses = calendar.month_name[1:13]

plt.axes([0.1,0.2,0.8,0.65])

# Creamos un plot con 365 valores
plt.plot(np.arange(1,366), np.random.rand(365), label = 'valores al
azar')

plt.xlim(-5,370) # Los valores del eje y variarán entre -5 y 370

plt.ylim(0,1.2) # Los valores del eje y variarán entre 0 y 1.2

plt.legend() # Creamos la caja con la leyenda

plt.title(u'Ejemplo de título') # Ponemos un título

# Ponemos un título superior
plt.suptitle(u'Ejemplo de título superior')

# Pedimos que se vean subrrayas de división en los ejes
plt.minorticks_on()

# Colocamos las etiquetas, meses, en las posiciones, días,
# con color azul y rotadas 45°
plt.xticks(dias, meses, size = 'small', color = 'b', rotation = 45)

plt.xlabel(u't (días)')

plt.ylabel('Media diaria')
```

Cuyo resultado será algo parecido a lo siguiente:



Y con esto hemos visto, más o menos, la forma básica de configurar los elementos del gráfico. Si no los habéis visto aún, podéis leer el capítulo 1 y el capítulo 2 de esta serie.

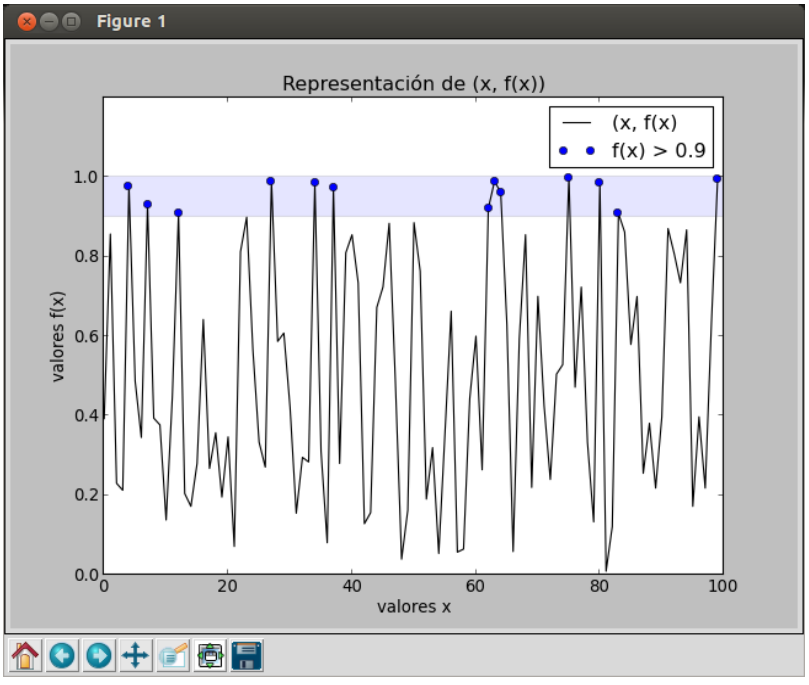
Si quieres puedes pasar a la siguiente parte.

4. Tipos de gráfico (I)

Hasta ahora hemos visto como configurar las ventanas, manejo de las mismas, definir áreas de gráfico,… Ahora vamos a ir viendo los diferentes tipos de gráficos que existen.

Como habéis podido comprobar, en los ejemplos anteriores hemos estado viendo mucho plt.plot() que es lo que se suele usar para dibujar un gráfico simple de líneas representando los valores (x, f(x)). Ahora vamos a ver un ejemplo explicado para que veáis todas las posibilidades de plt.plot().

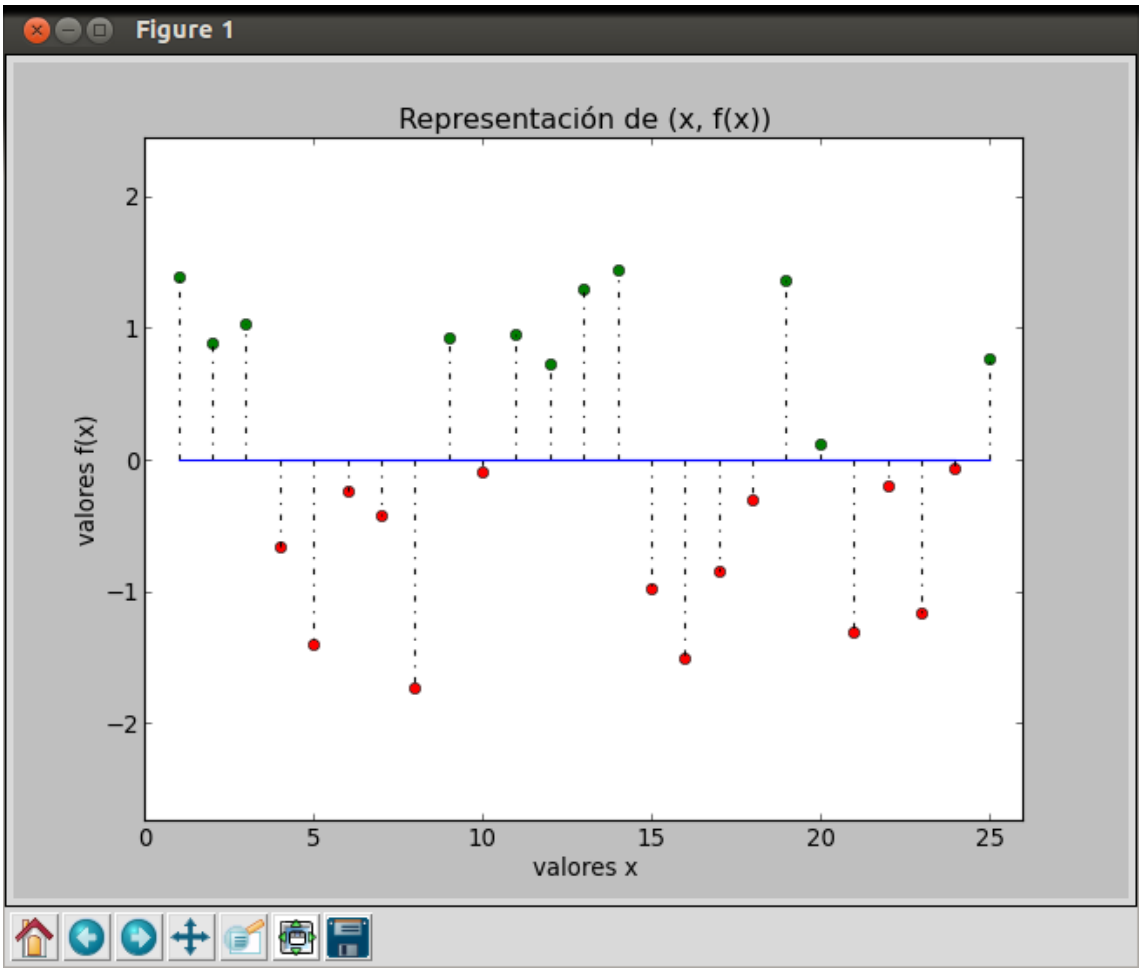
```
plt.ion() # Nos ponemos en modo interactivo

x = np.arange(100) # Valores de x

y = np.random.rand(100) # Valores de y

# Dibujamos la evolución de f(x) frente a x
plt.plot(x,y, color = 'black', label = '(x, f(x)')

# Destacamos los valores por encima de 0.9 colocándoles
# un marcador circular azul
plt.plot(x[y > 0.9], y[y > 0.9],'bo', label = 'f(x) > 0.9')

# Colocamos una banda de color para los valores f(x) > 0.9
plt.axhspan(0.9, 1, alpha = 0.1)

plt.ylim(0,1.2) # Limitamos el eje x
plt.legend() # Colocamos la leyenda

# Colocamos el título del gráfico
plt.title(u'Representación de (x, f(x))')

plt.xlabel('valores x') # Colocamos la etiqueta en el eje x

plt.ylabel('valores f(x)') # Colocamos la etiqueta en el eje y
```



Este es el tipo de gráfico que suelo usar un 75% de las veces. Tipos de gráfico análogos a este son plt.plot_date(), que es similar a plt.plot() considerando uno o ambos ejes como fechas, y plt.plotfile(), que dibuja directamente desde los datos de un fichero.

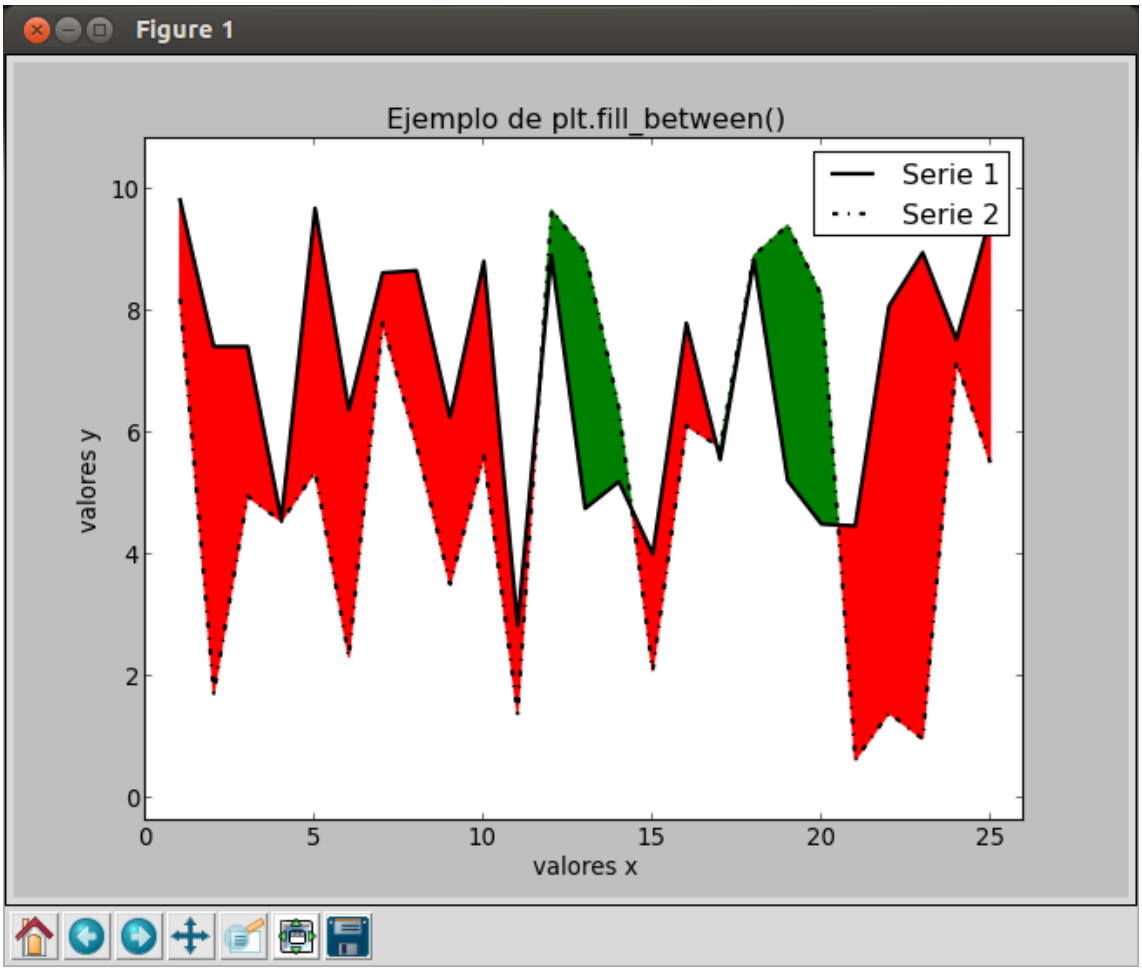
Otro tipo de gráfico sería el que podemos obtener con plt.stem(). Dibuja líneas verticales desde una línea base. Imaginaros, por ejemplo, que tenéis una serie temporal, la normalizamos (restándole su media y dividiendo por su desviación estándar) de forma que nos queda una serie de media 0 y desviación estándar 1. Esta nueva serie la podemos representar con plt.stem() donde la línea horizontal sería el valor medio (en este caso la media sería 0, recuerda que la hemos normalizado la serie) y las líneas verticales sería lo que se desvía el valor individual respecto de la media de la serie. Vamos a ver un ejemplo con los valores por encima de la media en verde y los valores por debajo de la media en rojo.

```
plt.ion() # Nos ponemos en modo interactivo

x = np.arange(25) + 1 # Valores de x

y = np.random.rand(25) * 10. # Valores de y

# Valores de y normalizados. Esta nueva serie tiene media 0
# y desvicación estándar 1 (comprobadlo como ejercicio)
y_norm = (y - y.mean()) / y.std()

# Colocamos los límites del eje x
plt.xlim(np.min(x) - 1, np.max(x) + 1)

# Colocamos los límites del eje y
plt.ylim(np.min(y_norm)-1, np.max(y_norm)+1)

plt.stem(x[y_norm > 0],
         y_norm[y_norm > 0],
         linefmt='k-.',
         markerfmt='go',
         basefmt='b-') # Dibujamos los valores por encima de la media

plt.stem(x[y_norm < 0],
         y_norm[y_norm < 0],
         linefmt='k-.',
         markerfmt='ro',
         basefmt='b-') # Dibujamos los valores por debajo de la media

# Colocamos el título del gráfico
plt.title(u'Representación de (x, f(x))')

plt.xlabel('valores x') # Colocamos la etiqueta en el eje x

plt.ylabel('valores f(x)') # Colocamos la etiqueta en el eje y
```



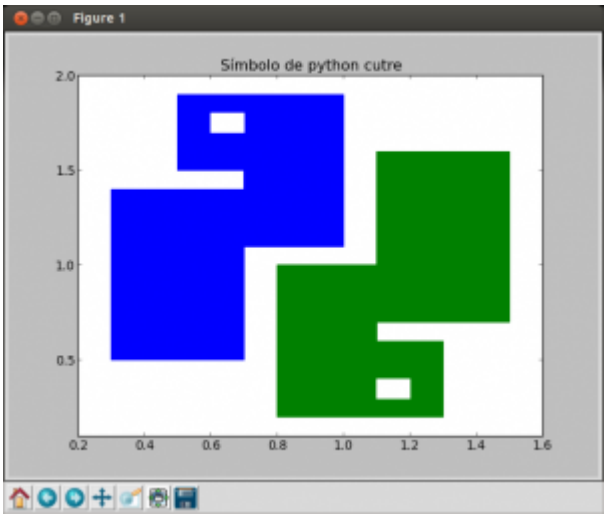
En algunos casos, nos interesa ver cuando una serie está por encima o por debajo de la otra. Eso, con un gráfico tipo plt.plot() lo podemos hacer sin problemas, pero nos gustaría resaltarlo visualmente de forma sencilla. Para ello podemos usar plt.fill_between(). Imaginemos un ejemplo donde tenemos dos series temporales y queremos localizar fácilmente cuando la primera está por encima de la segunda y cuando está por debajo.

```
plt.ion() # Nos ponemos en modo interactivo

x = np.arange(25) + 1 # Valores de x

y1 = np.random.rand(25) * 10. # Valores de y1

y2 = np.random.rand(25) * 10. # Valores de y2

# Colocamos los límites del eje x
plt.xlim(np.min(x) - 1, np.max(x) + 1)

# Colocamos los límites del eje y
plt.ylim(np.min([y1, y2])-1, np.max([y1, y2])+1)

# Dibujamos los valores de (x,y1) con una línea contínua
plt.plot(x, y1, 'k-', linewidth = 2, label = 'Serie 1')

# Dibujamos los valores de (x,y2) con una línea de punto y raya
plt.plot(x, y2, 'k-.', linewidth = 2, label = 'Serie 2')

# Pinta polígonos color verde entre las líneas cuando y1 < y2
plt.fill_between(x, y1, y2,
                 where = (y1 < y2),
                 color = 'g',
                 interpolate = True)

# Pinta polígonos color rojo entre las líneas cuando y1 > y2
plt.fill_between(x, y1, y2,
                 where = (y1 > y2),
                 color = 'r',
                 interpolate = True)

plt.legend()

# Colocamos el título del gráfico
plt.title('Ejemplo de plt.fill_between()')

plt.xlabel('valores x') # Colocamos la etiqueta en el eje x

plt.ylabel('valores y') # Colocamos la etiqueta en el eje y
```



Recordad que usamos valores aleatorios para y1 e y2 por lo que si usáis ese código no os tiene porque dar lo mismo. Como veis, cuando los valores de y2 son mayores que los de y1 dibuja polígonos verdes, en caso contrario dibuja polígonos rojos. Algo parecido pero para el eje y en lugar de para el eje x lo podemos hacer usando plt.fill_betweenx(). También podemos dibujar el polígono que queramos sobre el gráfico usando plt.fill().Veamos una 'cutrez' usando plt.fill():

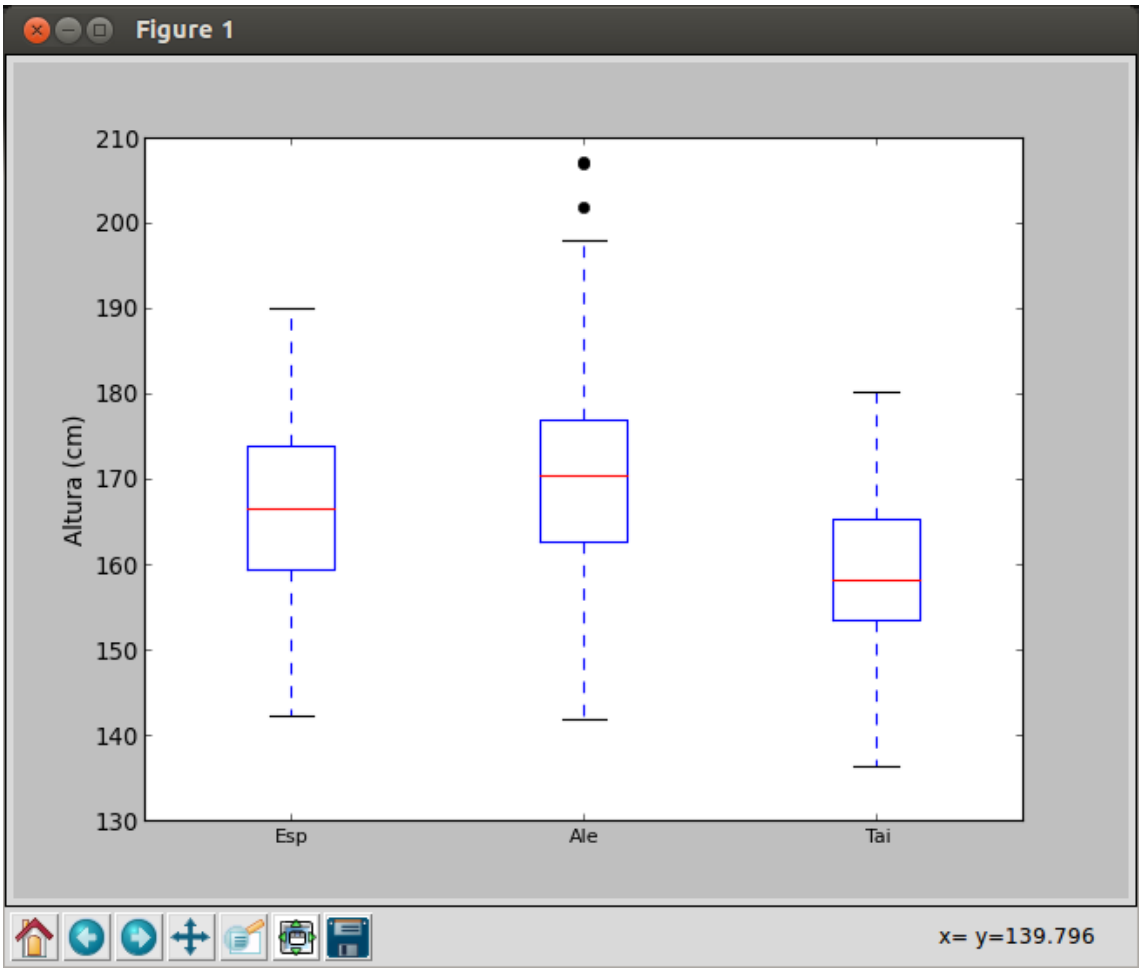
```
plt.ion() # Nos ponemos en modo interactivo

s1x = [0.3,0.3,0.7,0.7,0.5,0.5,1,1,0.7,0.7]

s1y = [0.5,1.4,1.4,1.5,1.5,1.9,1.9,1.1,1.1,0.5]

o1x = [0.6,0.6,0.7,0.7]

o1y = [1.7,1.8,1.8,1.7]

s2x = [0.8,0.8,1.1,1.1,1.5,1.5,1.1,1.1,1.3,1.3]

s2y = [0.2,1,1,1.6,1.6,0.7,0.7,0.6,0.6,0.2]

o2x = [1.1,1.1,1.2,1.2]

o2y = [0.3,0.4,0.4,0.3]

plt.fill(s1x, s1y, color = 'b')

plt.fill(o1x,o1y, color = 'w')

plt.fill(s2x, s2y, color = 'g')

plt.fill(o2x,o2y, color = 'w')

plt.title(u'Símbolo de python cutre')

plt.ylim(0.1,2)
```



Y ya lo último que vamos a ver en este capítulo es un diagrama de caja-bigote (box plot o box-whisker diagram). Este es un diagrama donde se puede ver un resumen de una serie de forma rápida y sencilla. En él se representa el primer cuartil y el tercer cuartil, que son los extremos de la caja, el valor de la mediana (o segundo cuartil), que se representa mediante una línea dentro de la caja, y los extremos de la serie que no se consideran anómalos, los llamados 'bigotes', que son los valores extremos que están dentro del rango de 1.5 veces el rango intercuartílico (IQR por sus siglas en inglés, Inter Quartil Range). Los valores que quedan fuera de este rango que definamos, que como hemos comentado suele ser 1.5xIQR, se consideran valores anómalos u 'outliers' y se representan como puntos fuera de los bigotes. Por tanto, imaginemos que estamos

representando la altura de las mujeres que viven en España, las mujeres que viven en Alemania y las mujeres que viven en Tailandia. Con un diagrama de caja-bigote podemos ver rápidamente como se distribuyen cada uno de estos conjuntos de datos y podemos compararlos visualmente entre ellos.

```
plt.ion() # Nos ponemos en modo interactivo

# Creamos unos valores para la altura de 100 españolas
alt_esp = np.random.randn(100)+165 + np.random.randn(100) * 10

# Creamos unos valores para la altura de 100 alemanas
alt_ale = np.random.randn(100)+172 + np.random.randn(100) * 12

# Creamos unos valores para la altura de 100 tailandesas
alt_tai = np.random.randn(100)+159 + np.random.randn(100) * 9

# El valor por defecto para los bigotes es 1.5*IQR pero lo escribimos
# explícitamente
plt.boxplot([alt_esp, alt_ale, alt_tai], sym = 'ko', whis = 1.5)

# Colocamos las etiquetas para cada distribución
plt.xticks([1,2,3], ['Esp', 'Ale', 'Tai'], size = 'small', color =
'k')

plt.ylabel(u'Altura (cm)')
```



[TODO ESTE COMENTARIO ES PARA COMENTAR EL GRÁFICO, CUALQUIER PARECIDO CON LA REALIDAD SERÍA MUY RARUNO Y HABRÍA QUE LLAMAR A FRIKER JIMÉNEZ] Vemos como las alemanas presentan alturas superiores y las tailandesas son las que, en general, mostrarían alturas inferiores. En las alemanas hay algunas mujeres que quedan por encima de lo que hemos considerado como valores normales llegando a alturas por encima de los 200 cm. Las españolas se encontrarían entre unas alturas de unos 140 cm y unos 190 cm.

Y, de momento, hemos acabado este capítulo. En el próximo capítulo veremos más tipos de gráfico que podemos hacer con matplotlib.pyplot. Si quieres ver las anteriores entregas del tutorial pulsa aquí. Y si quieres ver la nueva entrega pincha aquí.

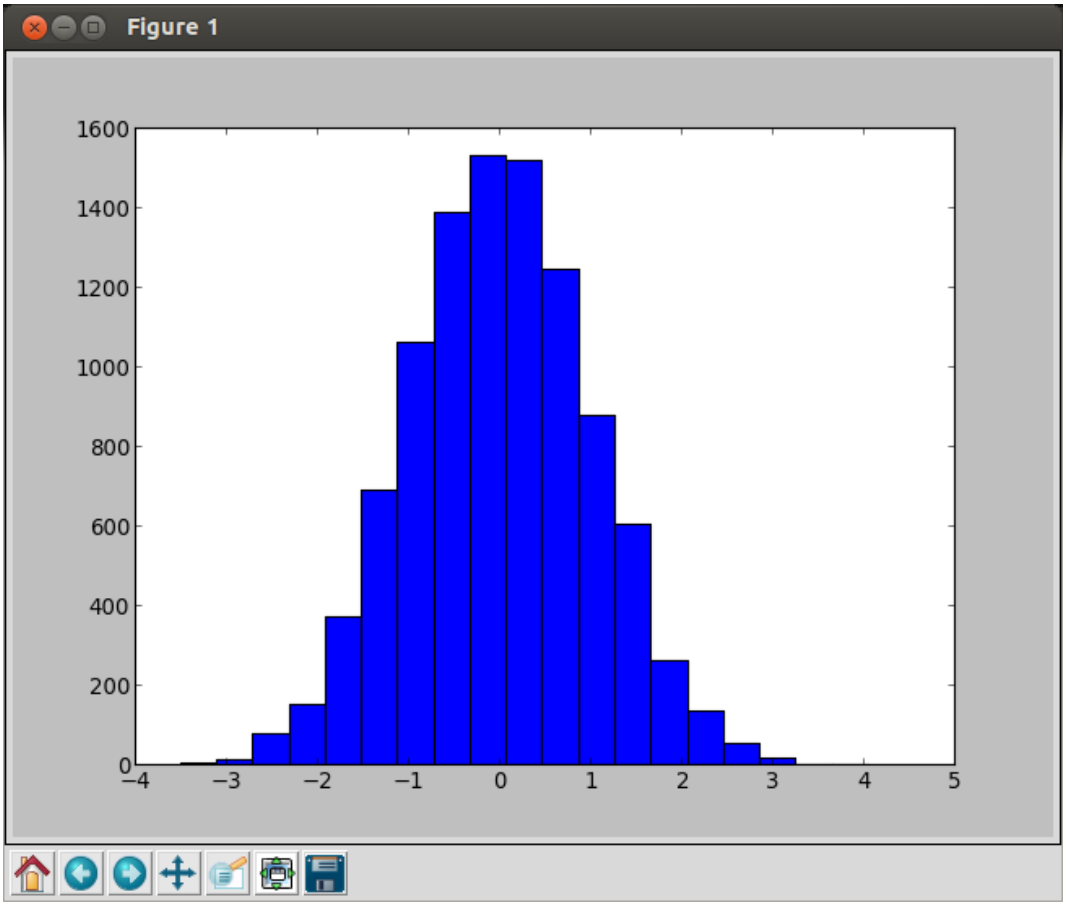
5. Tipos de gráfico (II)

Hasta ahora hemos visto como configurar las ventanas, manejo de las mismas, definir áreas de gráfico, algunos tipos de gráficos… Ahora vamos a continuar viendo tipos de gráficos disponibles desde matplotlib.pyplot. En este caso nos vamos a centrar en los gráficos de barras.

Para dibujar un histograma podemos hacer uso de plt.hist. Un histograma suele ser un gráfico de barras donde se representa la ocurrencia de datos (frecuencia) en intervalos definidos. Lo que hace plt.hist es dibujar el histograma de un vector en función del número de intervalos (bins) que definamos. Como siempre, vamos a ver esto con un ejemplo:

```
plt.ion() # Ponemos el modo interactivo

# Definimos un vector de números
# aleatorios de una distribución normal
x = np.random.randn(10000)

# Dibuja un histograma dividiendo el
# vector x en 20 intervalos del mismo ancho
plt.hist(x, bins = 20)
```

El resultado sería el siguiente, donde se representa el cálculo que haría la función np.histogram gráficamente y en un solo paso:

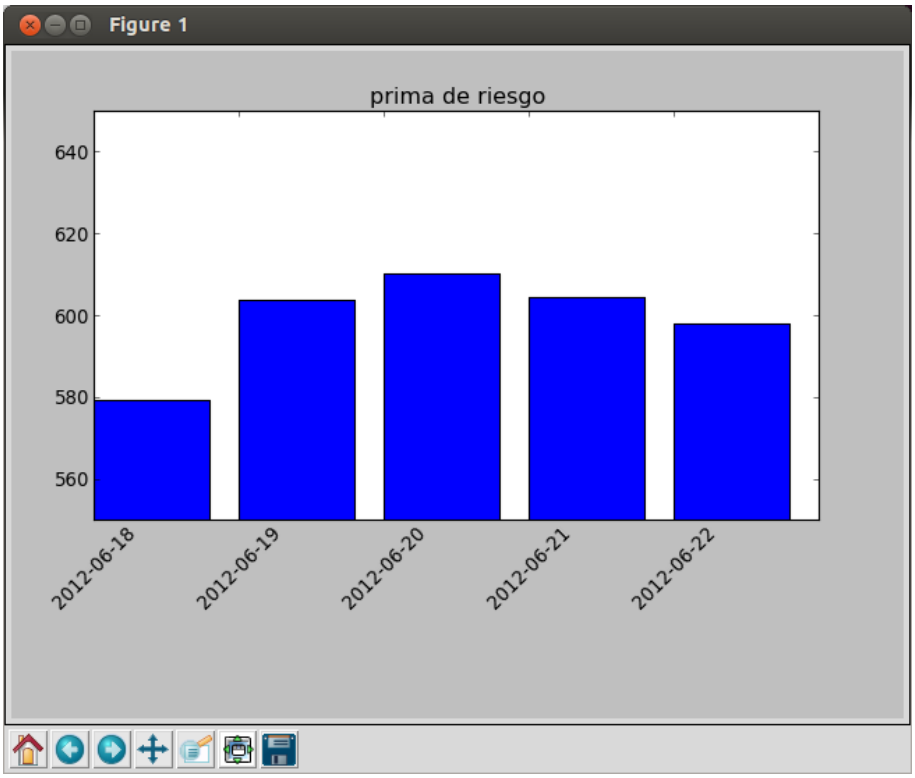


Podéis jugar también con np.histogram2d, np.histogramdd y np. bincount

Si en lugar de dibujar histogramas queremos dibujar gráficos de barras para representar, que se yo, la evolución de la prima de riesgo en los últimos días podemos usar plt.bar

```
import datetime as dt # Importamos el módulo datetime

# Valores inventados para la prima de riesgo
prima = 600 + np.random.randn(5) * 10

# generamos las fechas de los últimos cinco días
fechas = ((dt.date.today() - dt.timedelta(5)) +
          dt.timedelta(1) * np.arange(5))

plt.axes((0.1, 0.3, 0.8, 0.6)) # Definimos la posición de los ejes

plt.bar(np.arange(5), prima) # Dibujamos el gráfico de barras

# Limitamos los valores del eje y al rango definido [450, 550]
plt.ylim(550,650)

plt.title('prima de riesgo') # Colocamos el título

# Colocamos las etiquetas del eje x, en este caso, las fechas
plt.xticks(np.arange(5), fechas, rotation = 45)
```

Obtendríamos un resultado como este:

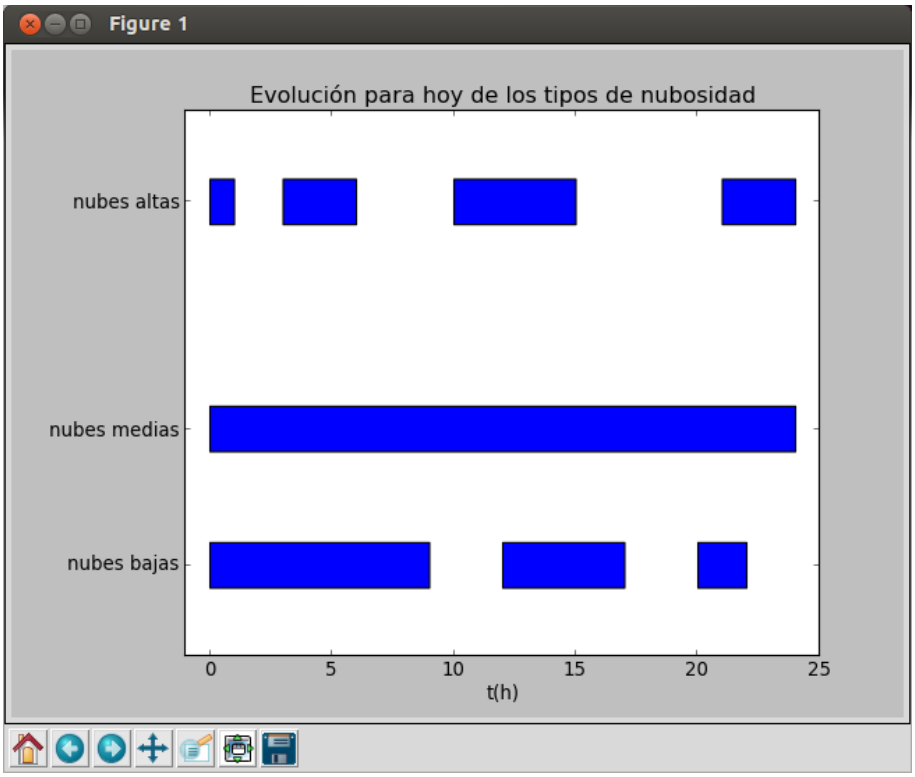


Si las barras las queréis dibujar en dirección horizontal en lugar de vertical podéis echarle un ojo a matplotlib.pyplot.barh. Siguiendo con los gráficos de barras vamos a

ver un caso un poco más especial haciendo uso de matplotlib.pyplot.broken_barh. Queremos representar el tipo de nubosidad que ha habido en un día concreto para saber cuando juanlu ha podido mirar las estrellas con su telescopio. El tipo de nubosidad lo vamos a desglosar en nubes bajas, medias y altas.

```
# Creamos los ejes en la posición que queremos
plt.axes((0.2,0.1,0.7,0.8))

# Ponemos un título al gráfico
plt.title(u'Evolución para hoy de los tipos de nubosidad')

# Dibujamos los momentos en que ha habido nubes altas
plt.broken_barh([(0,1),(3,3), (10,5), (21,3)], (9500, 1000))

# Dibujamos los momentos en que ha habido nubes medias
plt.broken_barh([(0,24)], (4500, 1000))

# Dibujamos los momentos en que ha habido nubes bajas
plt.broken_barh([(0,9), (12,5), (20,2)], (1500, 1000))

plt.xlim(-1,25) # Limitamos el rango de valores del eje x

# Colocamos etiquetas en el eje y
plt.yticks([2000, 5000, 10000],
           ['nubes bajas', 'nubes medias','nubes altas'])

# Y finalmente ponemos un título al eje x, el eje de tiempos
plt.xlabel('t(h)')
```

Además de poder ver que juanlu no ha podido usar su telescopio más que para mirar a la piscina de los vecinos porque el cielo estaba tapado, obtendríamos un resultado como este:



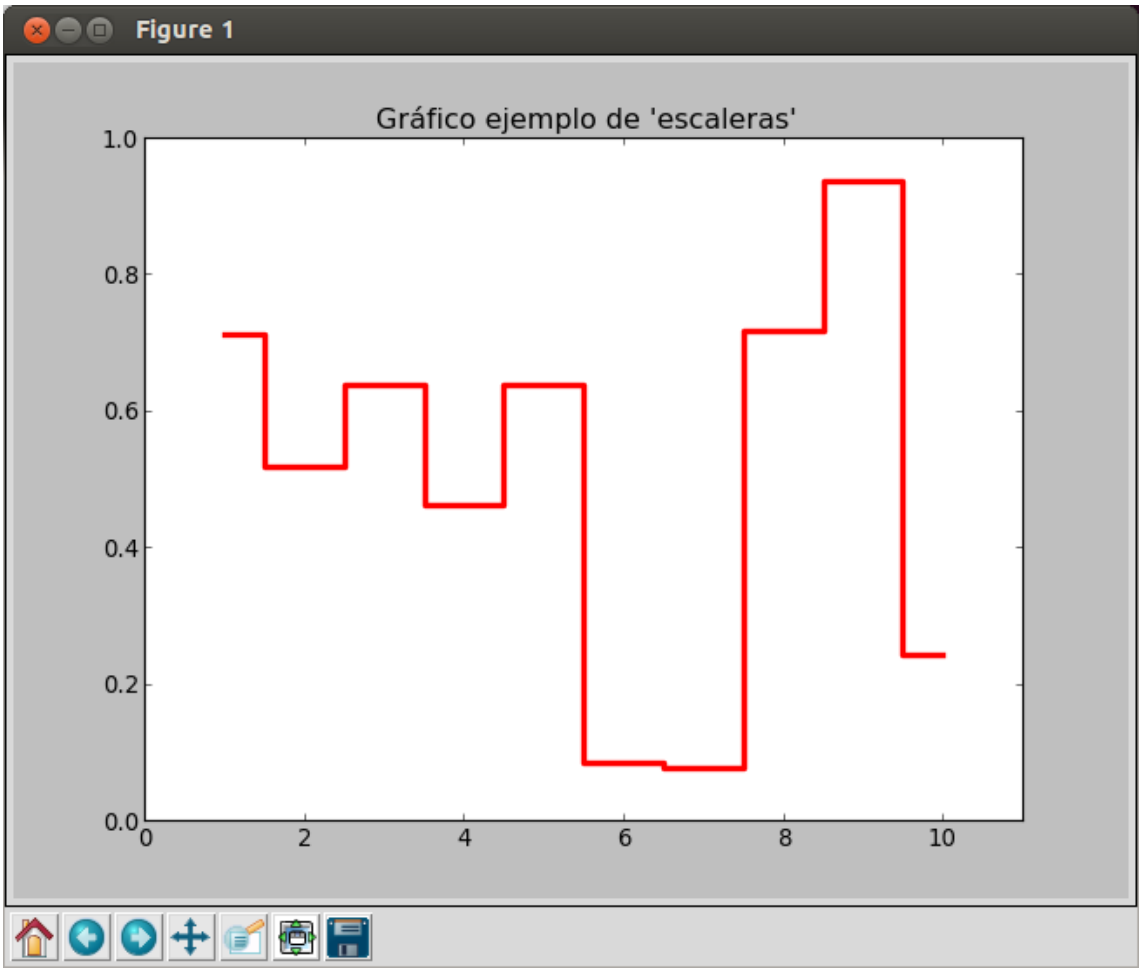
En plt.broken_barh se define primero los valores de x donde irá una barra y la longitud de esta barra y luego se pone el rango de valores de y para todas las barras definidas en x (además de poder cambiar colores y demás de las barras):

plt.broken_barh([(x0+longitud para la barra que empieza en x0), (x1+ longitud para la barra que empieza en x1), ..., (tantos x como queramos)], (valor mínimo del rango para y, longitud del rango de y desde el y mínimo), demás etiquetas que queramos incluir)

Por último para hoy y siguiendo con los gráficos de barras vamos a ver plt.step. Esta función nos permite dibujar un gráfico de 'escaleras'. Viendo esto en acción entenderéis mejor a lo que me refiero:

```
x = np.arange(10) + 1

y = np.random.rand(10)

plt.step(x, y, where = 'mid', color = 'r', linewidth = 3)

plt.title(u"Gráfico ejemplo de \'escaleras\'")

plt.xlim(0,11)
```

El where sirve para situar el centro de la escalera (trastead con ello, que es gratis). El resultado sería:



Y, de momento, hemos acabado por hoy. En el próximo capítulo veremos más tipos de gráfico que podemos hacer con matplotlib.pyplot. Si quieres ver las anteriores entregas del tutorial pulsa aquí. Y si quieres ver la nueva entrega pasa a la siguiente página.

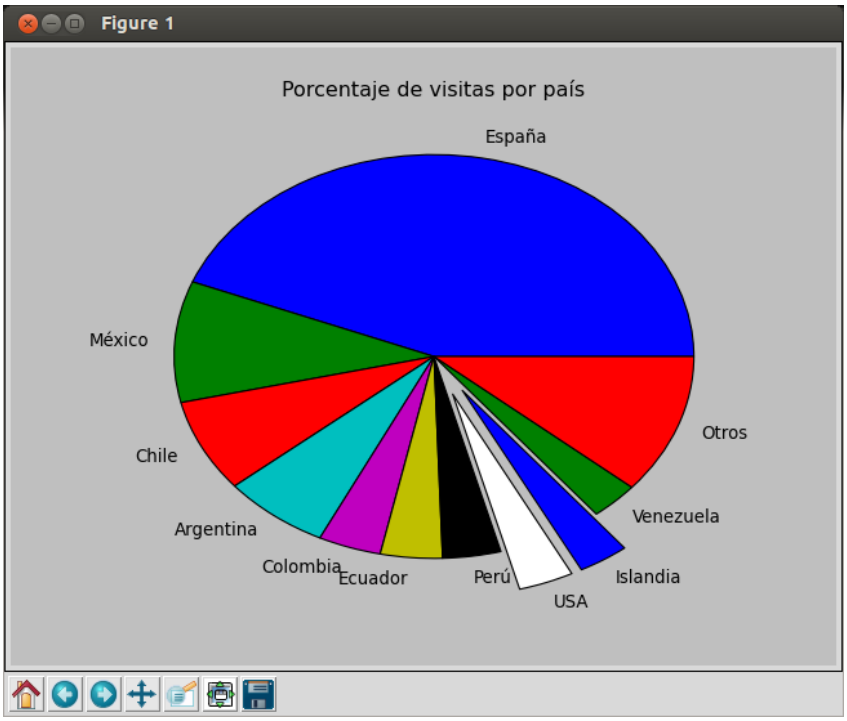
6. Tipos de gráfico (III)

Hasta ahora hemos visto como configurar las ventanas, manejo de las mismas, definir áreas de gráfico, algunos tipos de gráficos… Ahora vamos a continuar viendo tipos de gráficos disponibles desde matplotlib.pyplot. En este caso nos vamos a centrar en otros gráficos que, quizá, sean menos usados que los vistos hasta ahora. Algunos ya los hemos visto en otras entradas, como gráficos polares, gráficos de contornos [1] [2],…

Vamos a empezar por ver un gráfico tipo tarta de quesitos o tipo tarta o como lo queráis traducir (en inglés se llama pie chart). Estos sos los típicos gráficos que ponen en los periódicos con los resultados de elecciones o cosas así. En este caso vamos a ver un ejemplo real a partir de los datos de las visitas por países a este humilde blog:

```
plt.ion() # Ponemos el modo interactivo

# Definimos un vector con el % de visitas del top ten de países
visitas = [43.97, 9.70, 7.42, 6.68, 3.91,
           3.85, 3.62, 3.43, 3.16, 3.04]

# Introducimos un último elemento que recoge el % de visitas de
# otros países fuera del top10
visitas = np.append(visitas, 100. - np.sum(visitas))

# Etiquetas para los quesitos
paises = [u'España', u'México', 'Chile', 'Argentina', 'Colombia',
          'Ecuador', u'Perú', 'USA', 'Islandia', 'Venezuela', 'Otros']

# Esto nos ayudará a destacar algunos quesitos
explode = [0, 0, 0, 0, 0, 0, 0, 0.2, 0.2, 0, 0]

# Dibuja un gráfico de quesitos
plt.pie(visitas, labels = paises, explode = explode)

plt.title(u'Porcentaje de visitas por país')
```

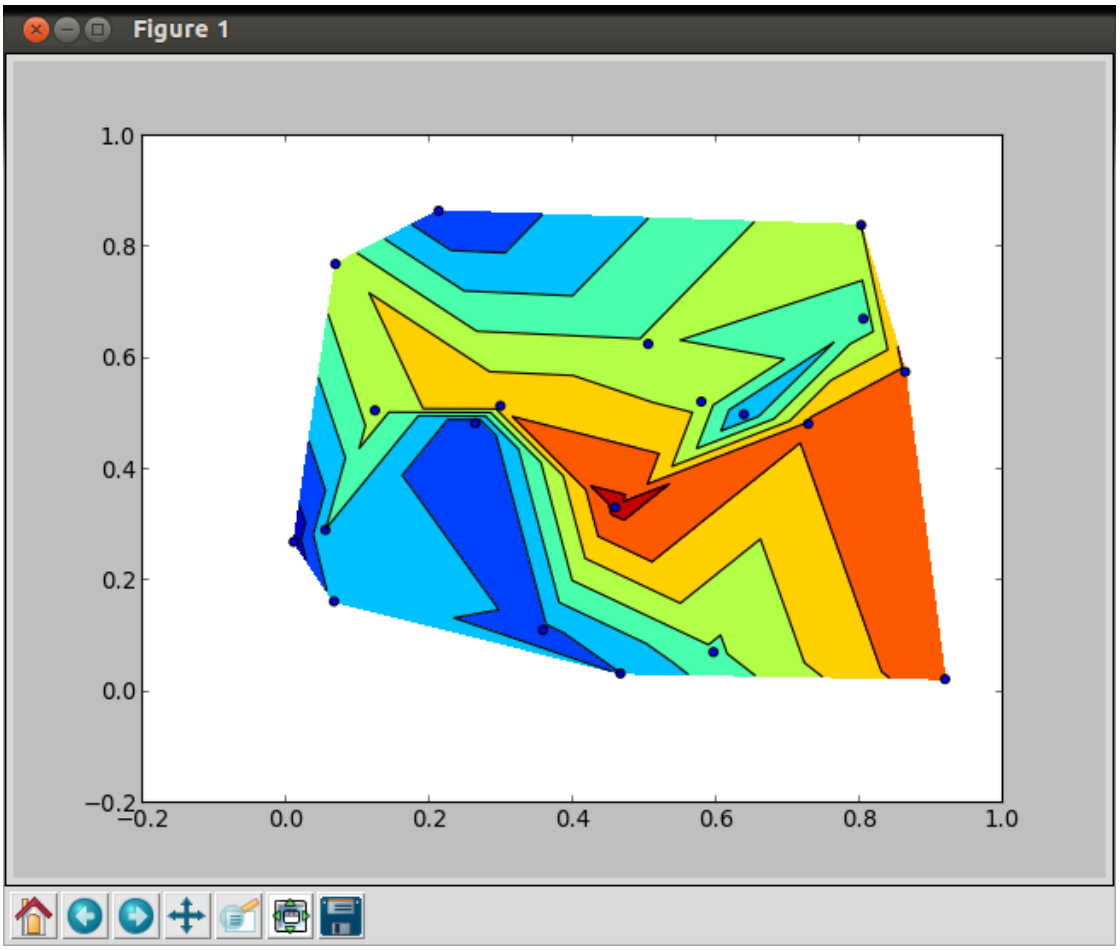
El resultado se puede ver en el gráfico siguiente. Como habréis adivinado, explode sirve para separar quesitos del centro de la tarta. En este caso hemos separado los quesitos de USA e Islandia para destacar los países no hispanohablantes:



Como ya hemos comentado anteriormente, ejemplos de gráficos de contornos ya hemos visto varios. Esos gráficos de contornos se hacen a partir de datos de mallas regulares. Pero, ¿qué sucede si tenemos datos que están repartidos de forma irregular? En este caso podemos hacer uso de plt.tricontour y de plt.tricontourf. Existen unas pocas diferencias de uso con respecto a plt.contour y plt.contourf. En este caso, el valor de Z no tiene que ser 2D. Para ver su funcionamiento pensemos en un caso real. Imaginad que tenéis una red de medidas (por ejemplo, temperaturas) repartidas geográficamente en una zona (AVISO, como siempre, los datos que vamos a representar no tienen ningún sentido físico ni pretender representar una situación real y solo se usan para ver el funcionamiento de tricontour y tricontourf, en este caso).

```
plt.ion() # Ponemos el modo interactivo

x = np.random.rand(20) # posiciones X de nuestra red de medidas

y = np.random.rand(20) # posiciones Y de nuestra red de medidas

# valores de Temperatura (°K) en las posiciones (X, Y)
t = np.random.rand(20)*3000

# Pintamos las triangulaciones con contornos de color
plt.tricontourf(x, y, t)

# Pintamos las líneas de contorno en color negro
plt.tricontour(x, y, t, colors = 'k')

plt.scatter(x, y) # Pintamos la posición de las estaciones de medida.
```

El resultado se puede ver en la siguiente figura. Se ha usado plt.scatter para representar la posición de las estaciones de medida:



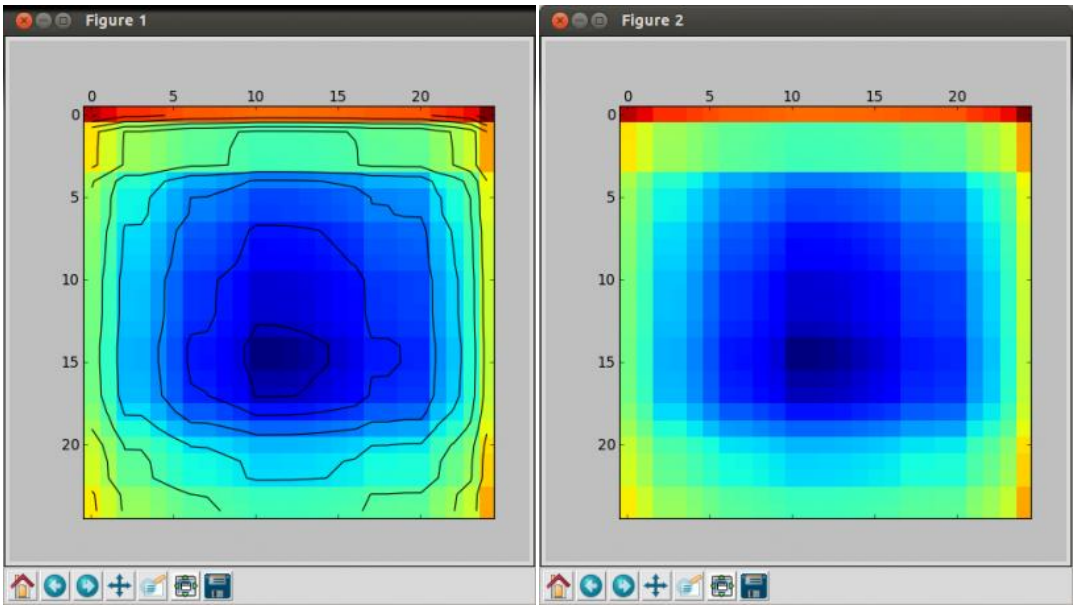
Por defecto usa una triangulación de Delaunay pero se puede definir la triangulación que queramos haciendo uso de matplotlib.tri.triangulation.

Todo esto está metido dentro del paquete matplotlib.tri, donde también podréis encontrar tripcolor y triplot. Probad con todo ello y mandadnos ejemplos para saber como lo usáis y aprender.

También podemos dibujar cuadros de valores que correspondan a una matriz en lugar de interpolar los valores mediante contornos. Puede suceder que, en muchos casos, el número de datos que tengamos en una malla regular sea bajo y una interpolación (usando contour, por ejemplo) dé resultados que pueden quedar feos y no representan fielmente lo que queremos representar. En esos casos podemos usar plt.matshow, que lo que hace es dibujar una matriz con cuadros de colores en función del valor de cada uno de los elementos de la matriz. Vamos a hacer otro ejemplo para que se entienda mejor:

```
plt.ion() # Ponemos el modo interactivo

# Valores de x que vamos a usar posteriormente para crear la matriz
x = np.sort(np.random.randn(25))

# Valores de y que vamos a usar posteriormente para crear la matriz
y = np.sort(np.random.randn(25))

# Creamos dos matrices cuadradas que vamos a cruzar
mat1, mat2 = np.meshgrid(x, y)

# Creamos una matriz final a partir de las dos anteriores
mat = np.sqrt( mat1**2 + mat2 **2)

plt.matshow(mat) # Representamos la última matriz con matshow

# Colocamos líneas de contorno para la matriz mat
plt.contour(np.arange(25), np.arange(25), mat, 10, colors = 'k')
```

El resultado lo podemos ver en el siguiente ejemplo. En la imagen de la izquierda vemos que las líneas de contorno, en este caso, quedan mal en los bordes y la representación solo usando matshow (imagen de la derecha) sería más adecuada (repito, todos los ejemplos no tienen más sentido que el de explicar el uso de matplotlib.pyplot).



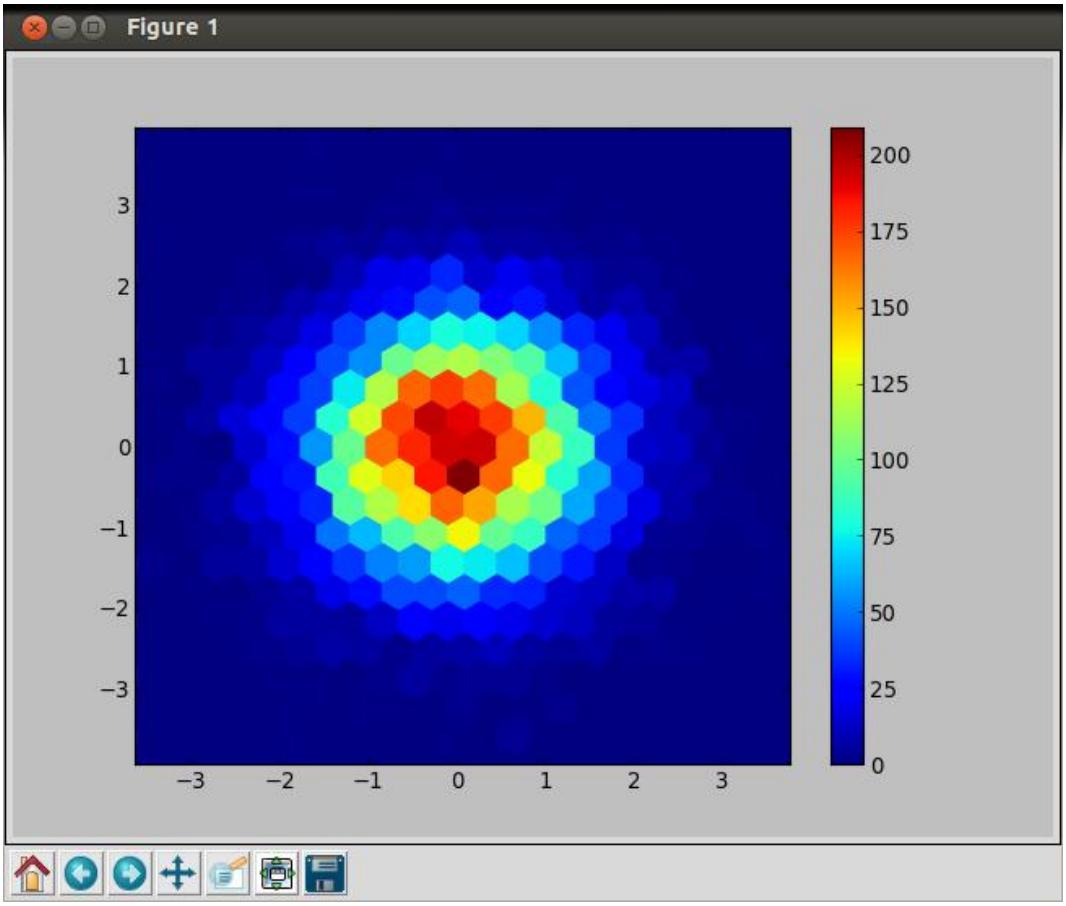
Podéis echarle un ojo a plt.pcolor, plt.pcolomesh y plt.tripcolor que permiten hacer cosas similares a estas.

plt.hexbin hace algo parecido a lo anterior pero teniendo en cuenta la ocurrencia en los intervalos que determinemos (esto mismo lo podemos hacer, por ejemplo, con plt.matshow aunque tendremos que calcular previamente las frecuencias para cada recuadro). Vamos a representar el número de veces que los valores de dos series (x e y)

se encuentran en determinado intervalo de datos. Para ello vamos a recurrir, como siempre, a np.random.randn:

```
plt.ion() # Ponemos el modo interactivo

# Creamos un vector de 10000 elementos distribuidos de forma normal
x = np.random.randn(10000)

# Creamos un vector de 10000 elementos distribuidos de forma normal
y = np.random.randn(10000)

# Representamos como están distribuidos bidimensionalmente con
# ayuda de hexbin, en este caso definimos un tamaño del grid de 20
# (esto se puede elegir como se prefiera)
plt.hexbin(x,y, gridsize = 20)

# Colocamos una barra de colores para saber a qué valor
# corresponden los colores
plt.colorbar()
```

El resultado es el siguiente:

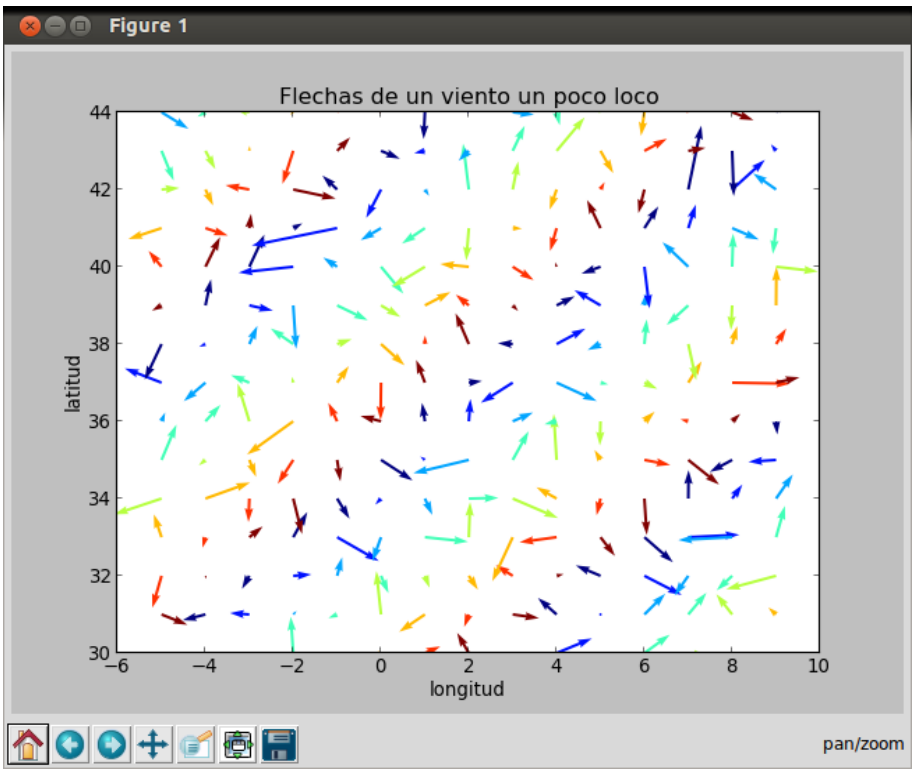


Por último por hoy vamos a dibujar gráficos de flechas. Esto se suele usar para dibujar viento, el movimiento de un fluido, movimiento de partículas, … En este caso vamos usar plt.quiver (echadle un ojo también a plt.quiverkey y a plt.barbs). Vamos a dibujar flechas de un viento un poco loco en las latitudes y longitudes de la Península Ibérica

(no voy a usar un mapa por debajo, si queréis completar el ejemplo usando un mapa podéis echarle un ojo a este ejemplo):

```
plt.ion() # Ponemos el modo interactivo

lon = np.arange(15) - 10. # Creamos un vector de longitudes

lat = np.arange(15) + 30. # Creamos un vector de latitudes

# Creamos un array 2D para las longitudes y latitudes
lon, lat = np.meshgrid(lon, lat)

# Componente x del vector viento que partirá desde una lon y una lat
# determinada
u = np.random.randn(15 * 15)

# Componente y del vector viento que partirá desde una lon y una lat
# determinada
v = np.random.randn(15 * 15)

# Definimos una serie de colores para las flechas
colores = ['k','r','b','g','c','y','gray']

plt.title('Flechas de un viento un poco loco') # Colocamos un título

plt.xlabel('longitud') # Colocamos la etiqueta para el eje x

plt.ylabel('latitud') # Colocamos la etiqueta para el eje y

# Dibujamos las flechas 'locas'
plt.quiver(lon, lat, u, v, color = colores)
```

El resultado es el siguiente:



Los colores los hemos colocado de forma aleatoria solo definiendo ocho colores.

Y, de momento, hemos acabado este capítulo. En el próximo capítulo veremos algún gráfico que no ha entrado dentro de los anteriores capítulos. Si quieres ver las anteriores entregas del tutorial pulsa aquí. Y si quieres ver la nueva entrega pasa a la siguiente página.

7. Tipos de gráfico (IV)

Para este capítulo, en todo momento supondremos que se ha iniciado la sesión y se ha hecho

```
import matplotlib.pyplot as plt

import numpy as np

import netCDF4 as nc

from mpl_toolkits.basemap import Basemap as Bm
```

Hasta ahora hemos visto como configurar las ventanas, manejo de las mismas, definir áreas de gráfico, algunos tipos de gráficos… Ahora vamos a ver un último capítulo sobre tipos de gráficos. En esta última entrada sobre los tipos de gráfico hemos metido gráficos que quizá no estén muy relacionados entre sí por lo que quizá este capítulo puede parecer un poco cajón desastre.

Anteriormente ya hemos usado Basemap, un toolkit que da capacidades de generar mapas a matplotlib. En este momento vamos a volver a recurrir a esta librería para mostrar datos sobre mapas. En el primer caso vamos a dibujar gráficos de barbas de viento (wind barbs) sobre un mapa. También vamos a usar la librería netcdf4-python, que permite leer, modificar y crear ficheros netcdf así como descargarlos desde diferentes TDS (servidores de datos THREDDS) de forma muy sencilla. Los datos que usaremos serán datos de viento obtenidos del reanálisis CFSR de la atmósfera pero podéis usar cualquier otro tipo de datos que se os ocurra:

```
plt.ion() # Ponemos el modo interactivo

url1 = 'http://nomads.ncdc.noaa.gov/thredds/dodsC/cfsr1hr/'
url2 = `200912/wnd10m.1.gdas.200912.grb2'

url = url1 + url2 # Ruta al fichero que usaremos

datos = nc.Dataset(url) # Accedemos a los datos
```

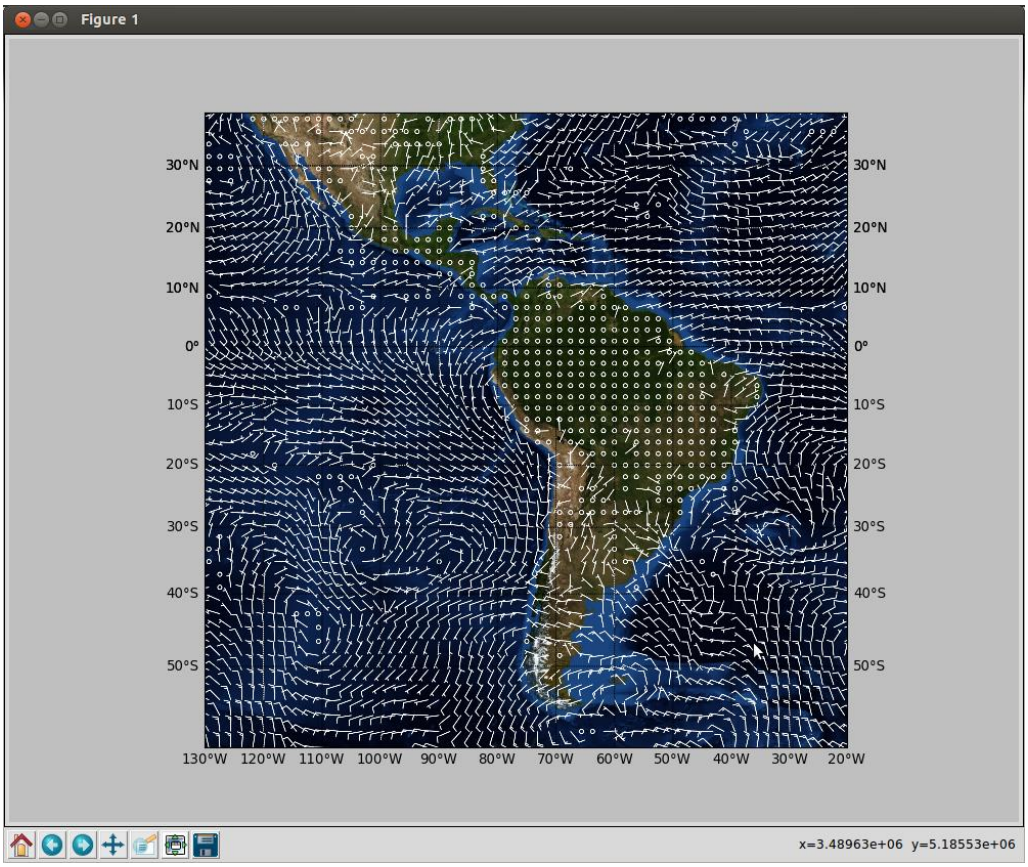
El fichero al que hemos accedido no es un fichero netcdf, sino que tiene formato grib2, pero netcdf4-python es 'todoterreno' y nos permite también acceder a este tipo de ficheros. Los datos a los que corresponden a diciembre de 2009 con pasos temporales de 6 horas. Usaremos solo el campo de vientos que corresponde a las 00:00 UTC del 01/12/2009. Si ponéis en el prompt datos.variables veréis un diccionario con las variables disponibles. Vamos a usar 'U-component_of_wind', 'V-component_of_wind', 'lon' y 'lat'.

```
# Guardamos en memoria el valor u del vector de viento para
# las 00:00 UTC del 01/12/2009
u = datos.variables['U-component_of_wind'][0,:,:,:]

# Guardamos en memoria el valor v del vector de viento para
# las 00:00 UTC del 01/12/2009
v = datos.variables['V-component_of_wind'][0,:,:,:]

lon = datos.variables['lon'][:] # Guardamos los valores de longitud

lat = datos.variables['lat'][:] # Guardamos los valores de latitud

# Hacemos una malla regular 2D para las latitudes y las longitudes
lons, lats = np.meshgrid(lon, lat)
```

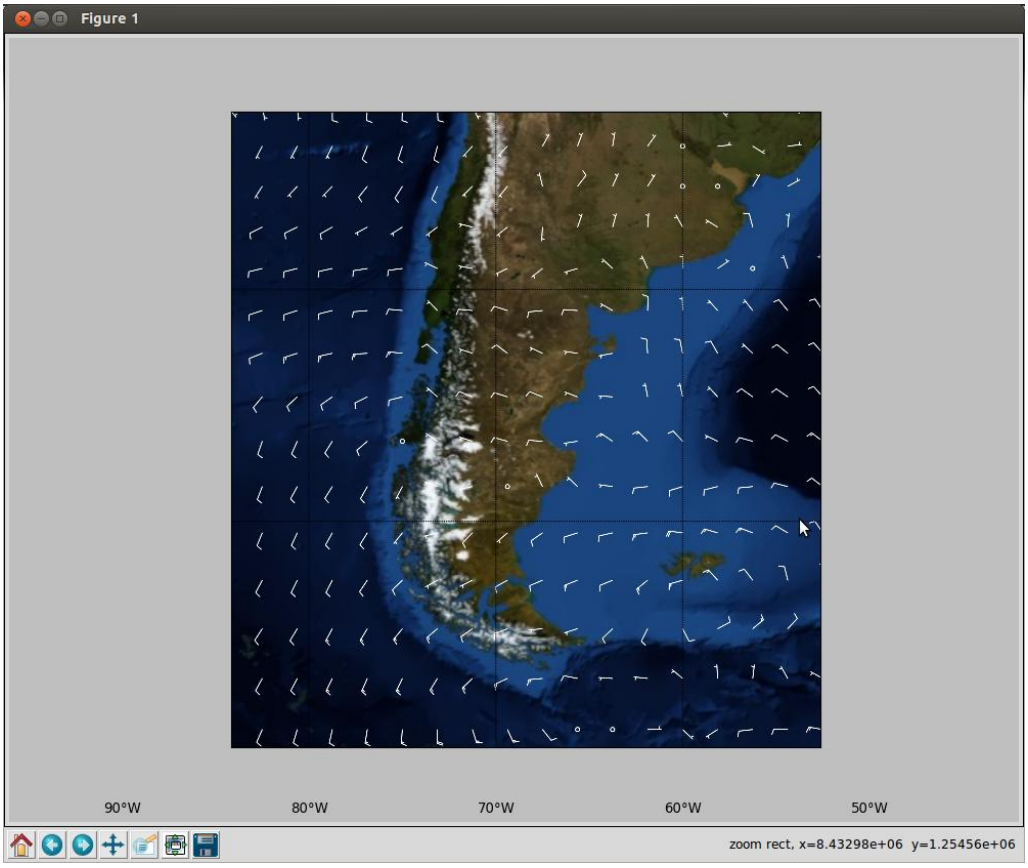
Una vez que disponemos de todas las variables pasamos a hacer el gráfico representando las barbas de viento. Como en diciembre hace mucho frío por Europa vamos a ver los vientos veraniegos que tuvimos en América Central y en América del sur durante en esa fecha.

```
# Definimos el área del gráfico y la proyección
m = Bm(llcrnrlon = 230,llcrnrlat = -60, urcrnrlon = 340,
       urcrnrlat = 38, projection = 'mill')

# Dibujamos los paralelos
m.drawparallels(np.arange(-180,180,10), labels=[1,1,0,0])

# Dibujamos los meridianos
m.drawmeridians(np.arange(0,360,10), labels=[0,0,0,1])

m.bluemarble() # Ponemos un mapa 'bonito' de fondo

# Y dibujamos los valores del vector viento
m.barbs(x, y, u[0,:,:], v[0,:,:], length=5, barbcolor='w',
        flagcolor='w',linewidth=0.5)
```

El resultado quedaría de la siguiente forma:



Y un detalle de la parte sur:



El código completo del ejemplo:

```
import matplotlib.pyplot as plt

import numpy as np

import netCDF4 as nc

from mpl_toolkits.basemap import Basemap as Bm

plt.ion() # Ponemos el modo interactivo

url1 = 'http://nomads.ncdc.noaa.gov/thredds/dodsC/cfsr1hr/'
url2 = '200912/wnd10m.1.gdas.200912.grb2'

url = url1 + url2 # Ruta al fichero que usaremos

datos = nc.Dataset(url) # Accedemos a los datos

# Guardamos en memoria el valor u del vector de viento para
# las 00:00 UTC del 01/12/2009
u = datos.variables['U-component_of_wind'][0,:,:,:]

# Guardamos en memoria el valor v del vector de viento para
# las 00:00 UTC del 01/12/2009
v = datos.variables['V-component_of_wind'][0,:,:,:]

lon = datos.variables['lon'][:] # Guardamos los valores de longitud

lat = datos.variables['lat'][:] # Guardamos los valores de latitud

# Hacemos una malla regular 2D para las latitudes y las longitudes
lons, lats = np.meshgrid(lon, lat)

# Definimos el área del gráfico y la proyección
m = Bm(llcrnrlon = 230,llcrnrlat = -60, urcrnrlon = 340,
       urcrnrlat = 38, projection = 'mill')

# Dibujamos los paralelos
m.drawparallels(np.arange(-180,180,10), labels=[1,1,0,0])

# Dibujamos los meridianos
m.drawmeridians(np.arange(0,360,10), labels=[0,0,0,1])

m.bluemarble() # Ponemos un mapa 'bonito' de fondo

# Y dibujamos los valores del vector viento
m.barbs(x, y, u[0,:,:], v[0,:,:], length=5, barbcolor='w',
        flagcolor='w',linewidth=0.5)
```

En realidad no hemos usado 'matplotlib.pyplot.barbs' sino que hemos hecho uso de 'barbs' dentro de 'basemap' pero su uso es similar. Si no queréis dibujar barbas y queréis dibujar flechas echadle un ojo a matplotlib.pyplot.quiver y a matplotlib.pyplot.quiverkey (aquí tenéis ejemplos que os pueden ayudar http://matplotlib.github.com/basemap/users/examples.html).

Podéis encontrar muchos ejemplos de como usar matplotlib.pyplot (en algunos casos usando pylab) en http://matplotlib.sourceforge.net/examples/pylab_examples/index.html.

Y, después de este breve entrada, hemos acabado por hoy y hemos acabado los tipos de gráfico que vamos a ver. Esto ha sido solo una muestra de las cosas que se suelen usar más. En el próximo capítulo veremos cómo hacer anotaciones en un gráfico. Si quieres ver las anteriores entregas del tutorial pulsa aquí.

8. Texto y Anotaciones

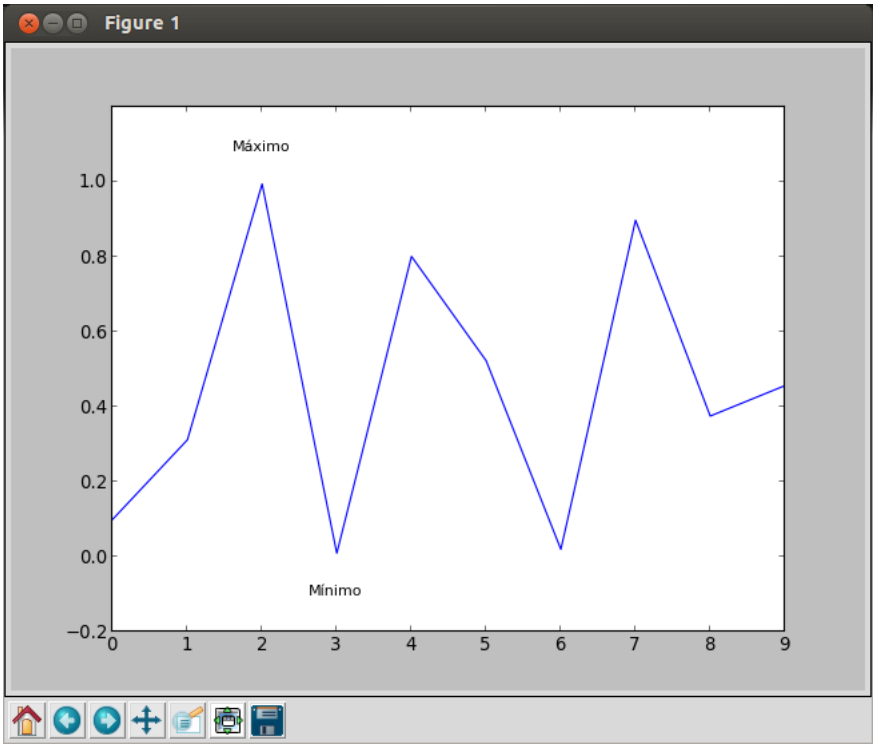
Hasta ahora hemos visto como configurar las ventanas, manejo de las mismas, definir áreas de gráfico, algunos tipos de gráficos… En esta ocasión nos interesa ver cómo podemos meter anotaciones, tablas,…, en nuestros gráficos.

A lo largo de las anteriores entregas del tutorial hemos podido ver algunas formas de tener anotaciones típicas para el título, los ejes, leyenda,… (title, suptitle, xlabel, ylabel, figtext, legend,…). En este caso vamos a revisar las posibilidades de escribir texto personalizado mediante el uso de plt.text, plt.arrow, plt.annotate y plt.table.

Como caso sencillo para anotar texto en nuestro gráfico podemos usar plt.text. En el siguiente ejemplo vamos a resaltar donde está el valor máximo y el valor mínimo de una serie de datos:

```
# Creamos una serie de 10 valores pseudo-aleatorios entre 0 y 1
y = np.random.rand(10)

plt.plot(y) # Los dibujamos

plt.ylim(-0.2, 1.2) # Definimos el rango de valores para el eje y

# Colocamos texto cerca del valor donde se encuentra el mínimo
plt.text(np.argmin(y), np.min(y) - 0.1, u'Mínimo', fontsize = 10,
         horizontalalignment='center', verticalalignment='center')

# Colocamos texto cerca del valor donde se encuentra el máximo
plt.text(np.argmax(y), np.max(y) + 0.1, u'Máximo', fontsize = 10,
         horizontalalignment='center', verticalalignment='center')
```

El resultado es el siguiente:

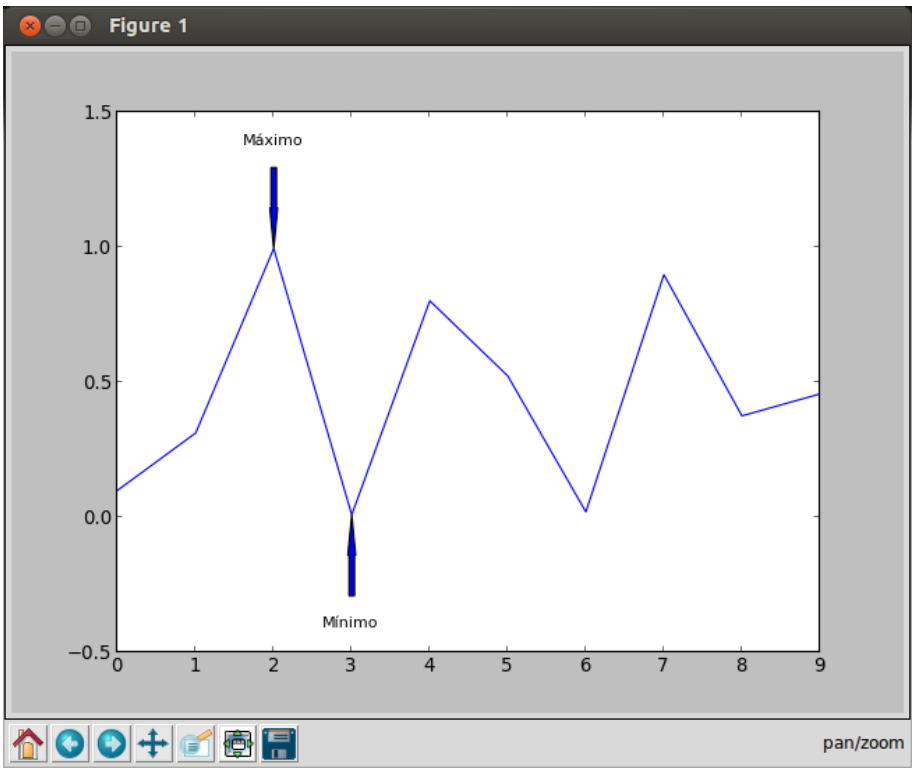


Lo que hemos hecho en plt. text es definir la posición del texto con un valor para la x y un valor para la y (en el sistema de referencia de los datos), la cadena de texto a mostrar, como queremos que sea la fuente, donde queremos que vaya colocado, si la queremos rotar, si la queremos en negrita,…

Al anterior ejemplo le podemos incluir una flecha que una el texto con la representación del valor máximo y del valor mínimo. Para ello podemos usar plt.arrow modificando ligeramente el anterior código:

```
plt.plot(y)

plt.ylim(-0.5, 1.5) # Extendemos un poco el rango del eje y

# Recolocamos el texto del máximo
plt.text(np.argmax(y), np.max(y) + 0.4, u'Máximo', fontsize = 10,
         horizontalalignment='center', verticalalignment='center')

# Recolocamos el texto del mínimo
plt.text(np.argmin(y), np.min(y) - 0.4, u'Mínimo', fontsize = 10,
         horizontalalignment='center', verticalalignment='center')

# Unimos el texto al valor representado
plt.arrow(np.argmax(y), np.max(y) + 0.3, 0, -0.3,
          length_includes_head = "True", shape = "full",
          width=0.07, head_width=0.1)

# Unimos el texto al valor representado
plt.arrow(np.argmin(y), np.min(y) - 0.3, 0, 0.3,
          length_includes_head = "True", shape = "full",
          width=0.07, head_width=0.1)
```

El resultado obtenido es el siguiente:

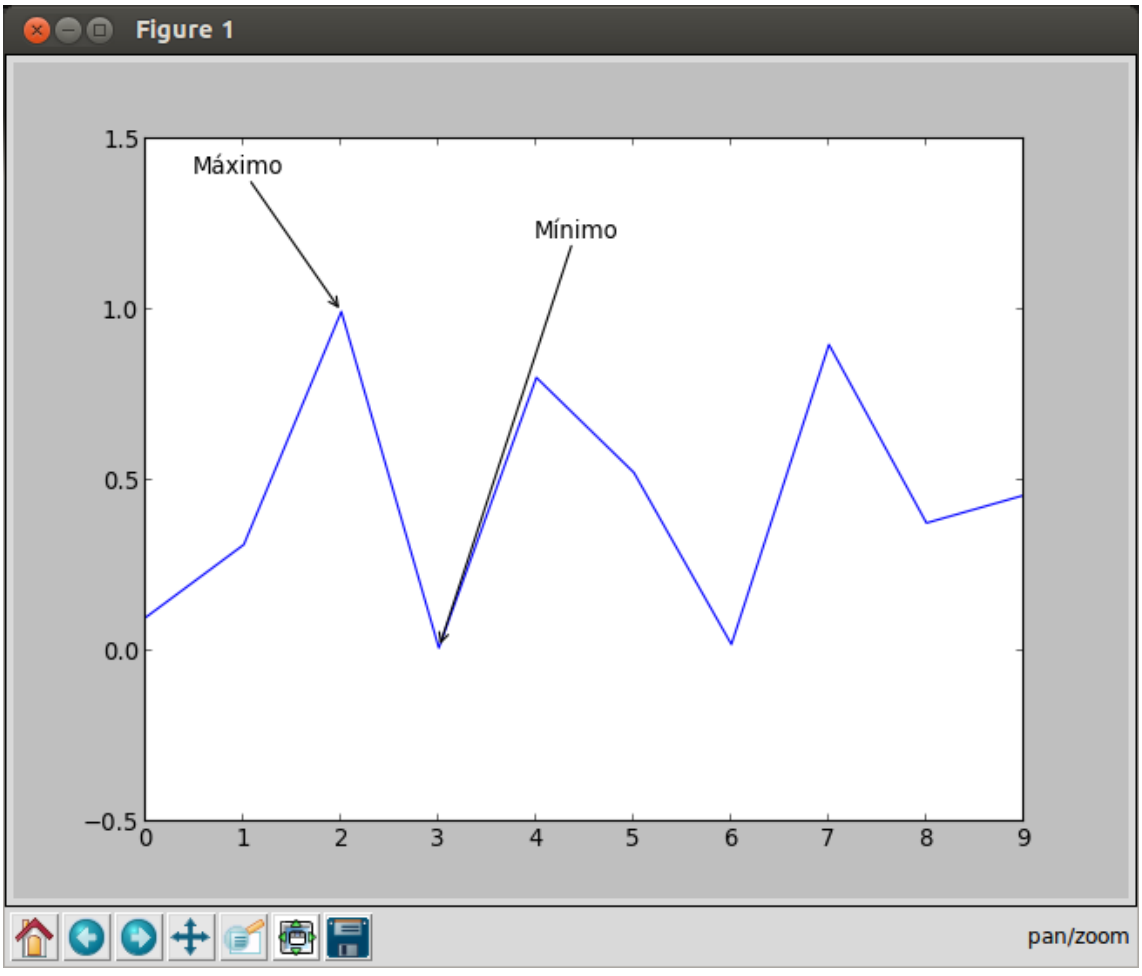


En plt.arrow hemos de definir el origen de la flecha, la distancia desde ese origen hasta el otro extremo de la flecha, si queremos que tenga cabecera, si queremos que la cabecera esté en el origen, el color de la flecha,…

Lo que hemos hecho con plt.text y con plt.arrow lo podemos hacer de forma más compacta y elegante con plt.annotate. Como anteriormente, hacemos uso de un ejemplo y vamos viendo las partes a modificar de plt.annotate:

```
plt.plot(y)

plt.ylim(-0.5, 1.5) # Extendemos un poco el rango del eje y

plt.annotate(u'Máximo', xy = (np.argmax(y), np.max(y)),
             xycoords = 'data',
             xytext = (np.argmax(y) - 1.5, np.max(y) + 0.4),
             textcoords = 'data',
             arrowprops = dict(arrowstyle = "->"))

plt.annotate(u'Mínimo', xy = (np.argmin(y), np.min(y)),
             xycoords = 'data',
             xytext = (np.argmin(y) + 1, np.min(y) + 1.2),
             textcoords = 'data',
             arrowprops = dict(arrowstyle = "->"))
```

Siendo el resultado el siguiente:

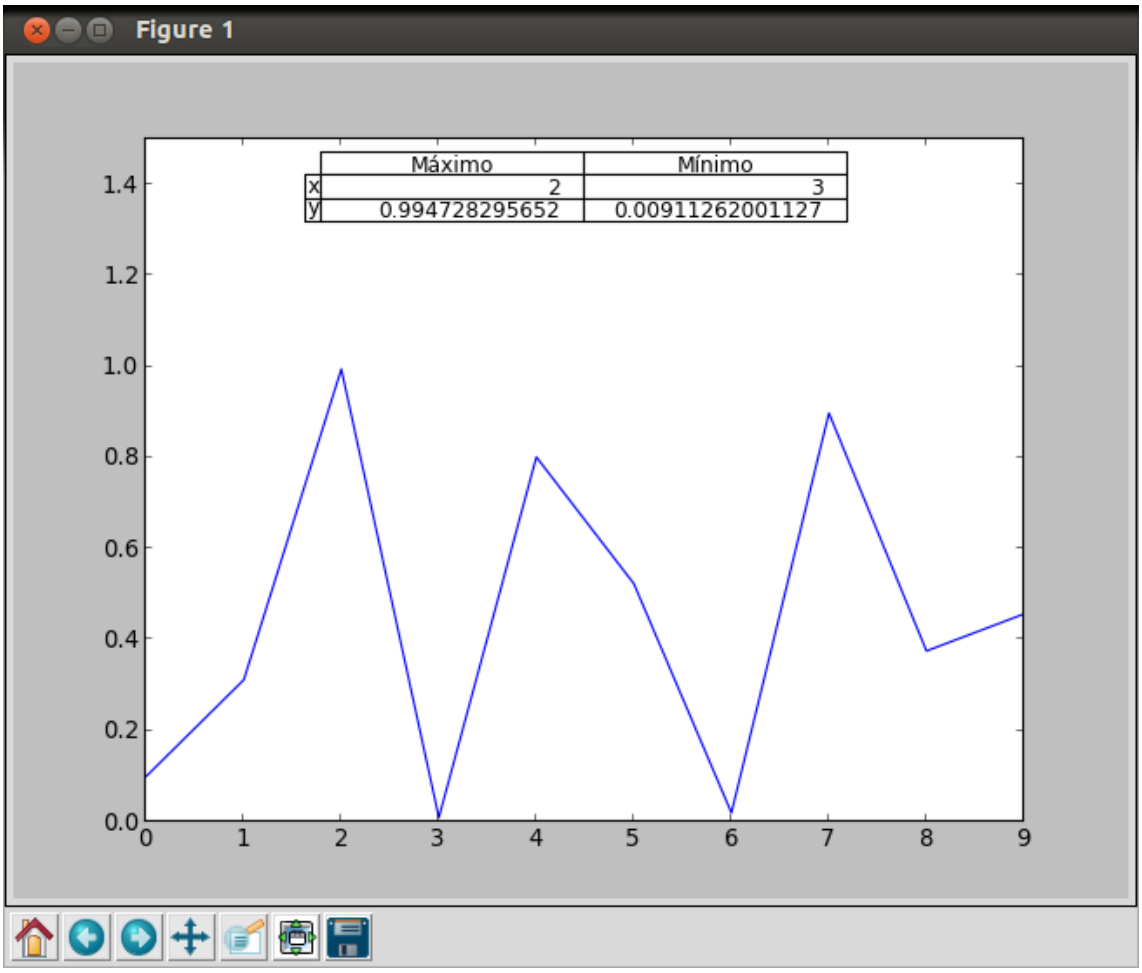


En plt.annotate introducimos la cadena de caracteres a mostrar, indicamos hacia donde apuntará esa cadena de caracteres (xy, en este caso estamos usando el sistema de referencia de los datos, 'data', pero podemos usar píxeles, puntos,…), la posición del texto (xytext), y como se representará la flecha. Con plt.annotate podemos tener anotaciones elegantes de forma sencilla como puedes ver en estos enlaces [1], [2].

Por último, vamos a ver cómo podemos dibujar una tabla de forma sencilla. Con plt.table podemos meter rápidamente una tabla pero por defecto la mete debajo del eje x. Vamos a ver un ejemplo que he encontrado en SO donde metemos la tabla dentro de los ejes.

```
valores = [[np.argmax(y), np.argmin(y)], [np.max(y), np.min(y)]]

etiquetas_fil = ('x', 'y')

etiquetas_col = (u'Máximo', u'Mínimo')

plt.plot(y)

plt.table(cellText=valores, rowLabels=etiquetas_fil,
          colLabels = etiquetas_col, colWidths = [0.3]*len(y),
          loc='upper center')
```

Cuyo resultado es el siguiente:



Donde hemos definido los valores de las celdas internas (cellText), Las etiquetas de filas y columnas (rowLabels y colLabels), el ancho de las celdas y la localización de la tabla.

Y, después de este breve entrada, hemos acabado el capítulo haciendo un montón de anotaciones. Si quieres ver las anteriores entregas del tutorial pulsa aquí. Y si quieres ver la nueva entrega pasa al siguiente capítulo.

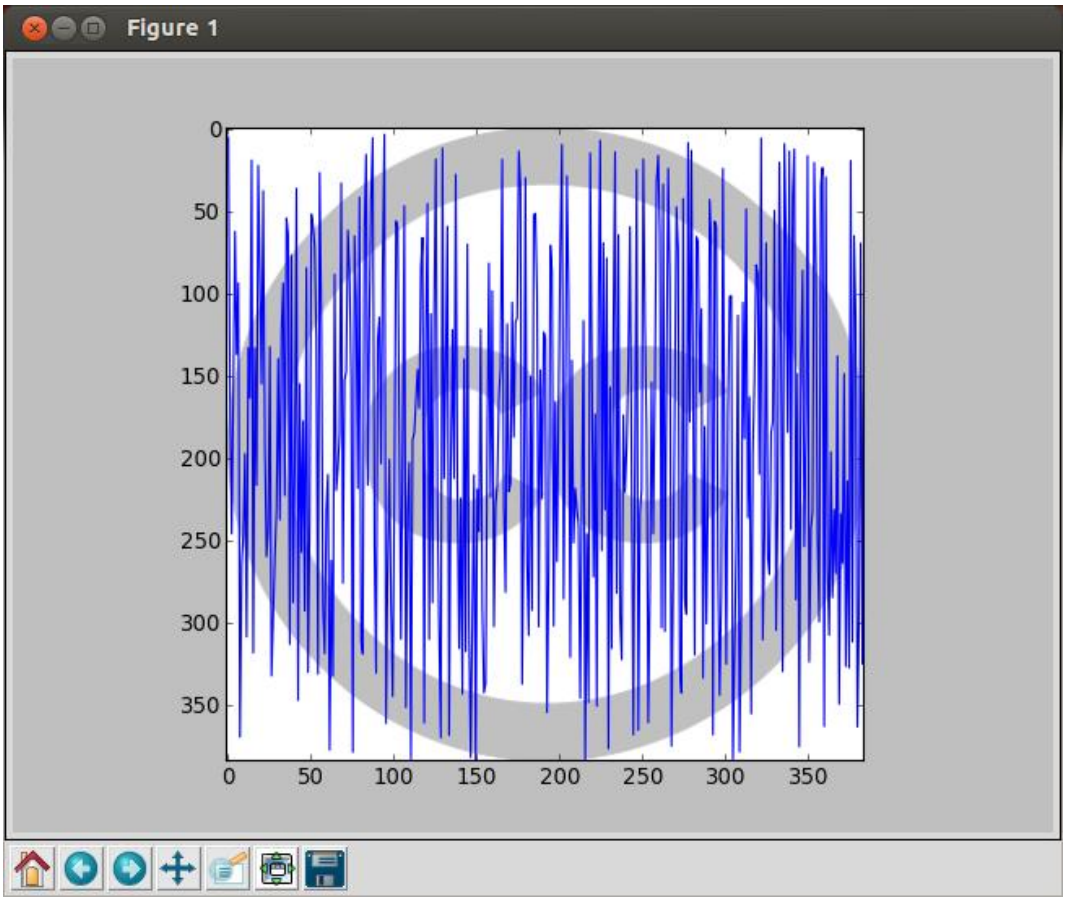
9.Miscelánea

Después de dar un repaso por toda la librería, obviando algunas funciones estadísticas y eventos, vamos a acabar este tutorial viendo algunas funciones que sirven para leer y guardar imágenes.

Imaginad que queréis usar una imagen de fondo, por ejemplo vuestro nombre, o las siglas de creative commons o una foto,..., en vuestros gráficos. Para el ejemplo que vamos a ver a continuación vamos a usar la imagen que está en el siguiente enlace como fondo (guardadala en local para poder leerla).

```
# Leemos la imagen que queremos usar de fondo, lo que escribáis entre
# comillas es la ruta a la imagen
background = plt.imread('Cc.large.png')

x = np.arange(background.shape[1]) # Definimos valores de x

# Definimos valores de y
y = np.random.rand(background.shape[0]) * background.shape[0]

plt.plot(x, y) # Dibujamos la serie

# Creamos el fondo con una transparencia del 0.25 (1 es opaco y 0 es
# transparente)
plt.imshow(background, alpha = 0.25)
```

El resultado es el siguiente:



Con plt.imread lo que hacemos es leer una imagen y convertirla en un numpy array que más tarde podemos utilizar como queramos (en este caso, como fondo para la imagen). Con plt.imshow lo que hemos hecho es mostrar la imagen en pantalla. Por último, que sepáis que también existe plt.imsave, que permite guardar un numpy array como una imagen.

Por último, pero no por ello menos importante, quedaría el uso plt.savefig, que nos permite guardar cualquiera de las figuras que hemos ido creando a lo largo de todo el tutorial. Para el anterior caso, solo tenemos que añadir lo siguiente al código de más arriba:

```
plt.savefig('imagen_con_fondo_cc.png')
```

La función plt.savefig permite definir la resolución de la imagen, el formato de salida (por defecto, matplotlib solo permite los formatos png, eps, ps, png y svg, si queremos usar otros formatos haría falta instalar otras librerías (o bibliotecas) como PIL), la orientación de la figura,...

Y esto es todo, de momento, espero que os haya resultado útil por lo menos alguna cosa. Lástima no disponer de más tiempo para poder ver todo lo anterior con más profundidad pero espero que, por lo menos, si no conocíais esta maravillosa librería/biblioteca :-) os haya servido para adentraros un poco en ella y poder profundizar más por vuestra cuenta.

He 'limpiado' y resumido todo el tutorial en un documento pdf que podéis descargar a continuación.

Hasta la próxima.

P.D.: Por favor, si veis errores, conceptos erróneos, faltas ortográficas o tenéis alguna crítica, podéis usar los comentarios para hacérnosla llegar y que la podamos corregir.