1 Dibujos usando Matplotlib

Esta Librería es una de las varias que existen para realizar gráficos en Python. No es la única, pero es sin duda la más usada.

Ventajas: - Se pueden hacer dibujos con pocos comandos, pero llegado el caso muy sofisticados - Tiene una forma dual, por comandos directos, o bien accediendo al objeto que describe la figura. Esta última opción es mucha más laboriosa pero muy útil en casos de gráficos muy sofisticadas o con detalles complejos.

Muchas información en https://matplotlib.org/

Y una cantidad impresionante de ejemplos (con el código para usarlo en un dibujo similar) en: https://matplotlib.org/stable/plot_types/index.html

Y diagramas para referencias rápidas en: https://matplotlib.org/cheatsheets/

Para usarlo hay que cargar **NumPy** de manera **Obligatoria**

Matplotlib se construyó para su uso sobre NumPy

Algunos de los ejemplos presentados en este capítulo fueron tomados de https://matplotlib.org/stable/gallery/

1.1 Pyplot

pyplot es lainterfase a la biblioteca de matplotlib. Pyplot está diseñada siguiendo el estilo de un lenguaje llamado Matlab que fue muy exitoso en ingeniería y ciencia. Por lo cual a Matplotlib se accede a traves de pyplot con el siguiente comando:

import matplotlib.pyplot as plt

Es decir de matplotlib cargamos pyplot para usarla esta librería y sus funciones, vemos un ejemplo:

[1]:
```
import numpy as np
import matplotlib.pyplot as plt # Esta es la librería para graficar
```

Hay mucha información de como usar esta librería en la página oficial: http://matplotlib.org/

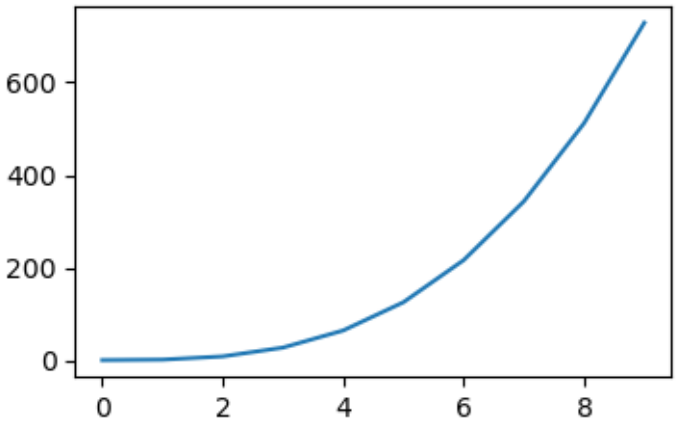
2 Plot simple

Veamos que Matplotlib es muy fácil de usar

[3]:
```
x = np.arange(10) # defino un arreglo para tener un referencia
```

y dibujo f(x) = x^3 de una manera compacta.

[4]:
```
plt.plot(x, x**3); #  El ";" al final es para que no me salga una
                   #  descripción del objeto estilo
                   # [<matplotlib.lines.Line2D at 0x7fbd28b9b590>]
```



Cómo pueden ver en el ejemplo, este gráfico nos llevó sólo una línea de código.

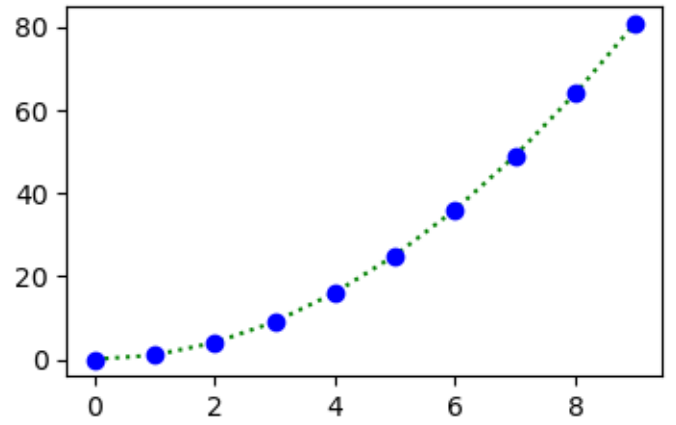
El método plot de matplotlib, necesitó la abscisa que es el arreglo NumPy **x** y la ordenada que es el resultado del cálculo de f(x) = x^3, que se realizó al vuelo mientras se corrió el código. Note que no se hizo indicación alguna del tipo de línea, color o punto en el dibujo. Es decir se cargaron valores "default" pre-programados.

3 Controlando los colores y los símbolos en el dibujo

Hay modos compactos de órdenes para modificar el gráfico, veamos algunas.

Y también podemos superponer gráficos

[5]:
```
plt.plot(x,x**2,':g')  #<--- determina la línea
plt.plot(x,x**2,'ob');  #<--- determina los puntos
```



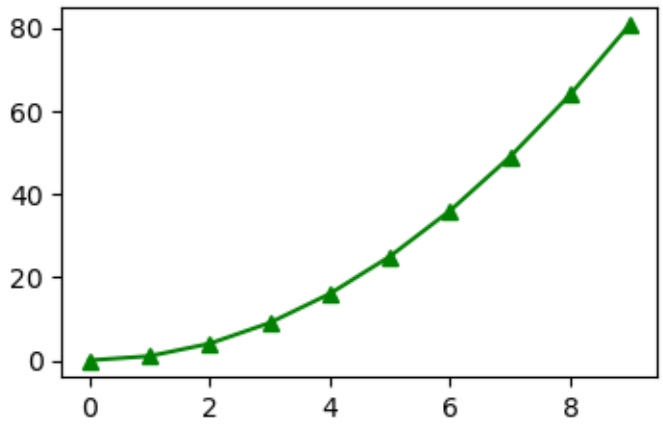
En este ejemplo, se generaron dos gráficos, uno por línea de código. En la primera línea, se agrego '**:g**'

que significa en un modo muy compacto ":" línea de puntos y "g" color verde (la g es de green). En la segundo línea ahora el "o" indica un punto lleno y "b" color azul (la b es de blue)

Note que la segunda orden no une puntos, y es entonces un gráfico de puntos discretos (no conectados o "scatter" en inglés)

También podría haber utilizado una orden no tan compacta y ser más descriptivo de lo que estoy programando.

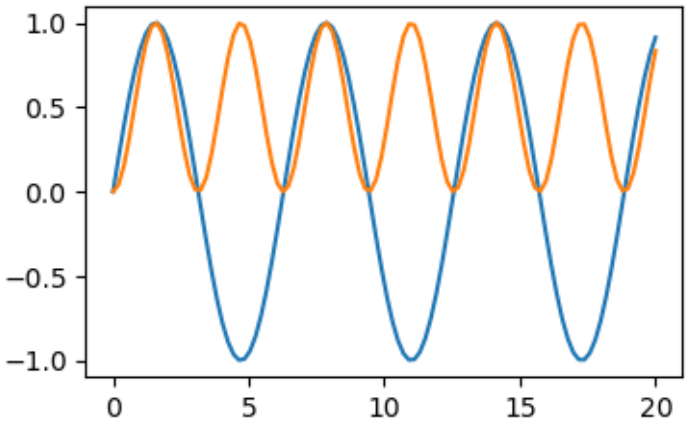
[6]:
```
plt.plot(x, x**2, c='green', marker='^');
```



En este ejemplo pueden ver que el color y el símbolo se pueden poner también modos no tan comprimidos y más claros para el lector del código. "c" o "color" indicará el color y "marker" el símbolo para marcar los puntos en el dibujo.

4 Sobre escribiendo dibujos (Overplot)

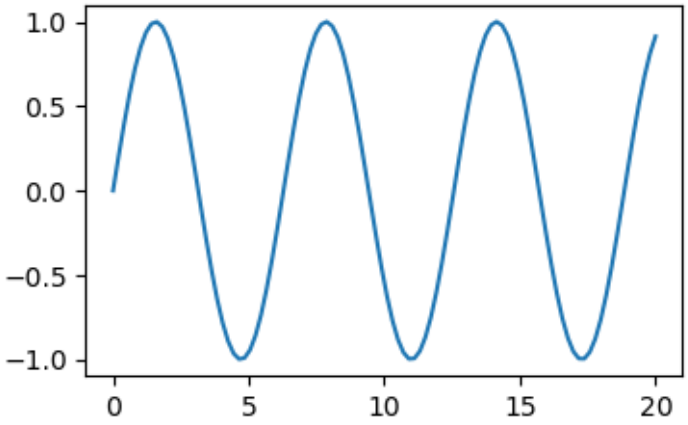
[10]:
```
# Genero dos arreglos NumPy
x = np.linspace(0, 20, 100)  # 100 valores separados de 0 to 20
y = np.sin(x)

# y los dibujo, pero en el segundo realizo otro cálculo en los valores de la␣
 ↪ordenada.
plt.plot(x, y)
plt.plot(x, y**2);
```



5 Modificando ejes y límites

Veamos este dibujo:

[11]:
```
plt.plot(x, y);
```



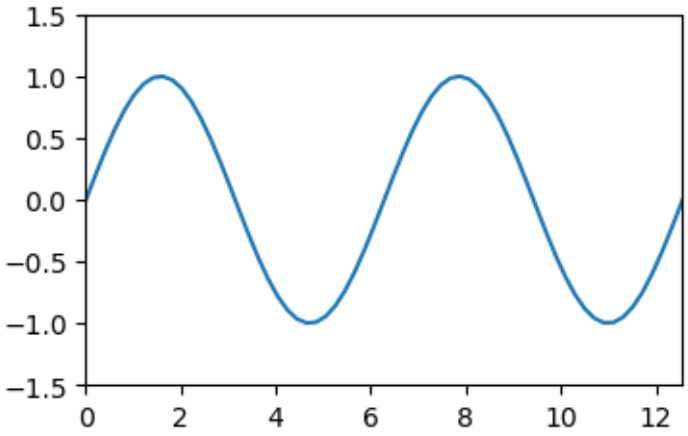
Pero resulta que quiero que sea entre 0, y 4π, o que la ordenada este en el rango (-1.5,1)

Tengo que fijar entonces estos límites en ambos ejes. para ello uso los siguientes comandos: xlim, e ylim.

Si no los uso, el gráfico se realizará con valores “default” determinados por el propio Python

[12]:
```
plt.plot(x, y)
plt.xlim((0., 4*np.pi))
plt.ylim((-1.5,1.5))
```

[12]: (-1.5, 1.5)



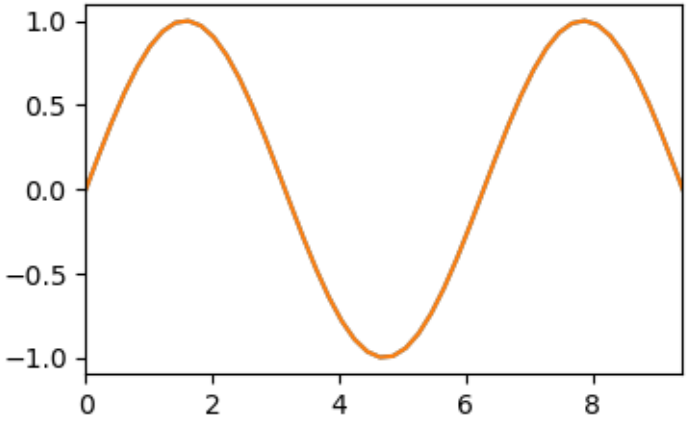
6 Multiples gráficos en una sola celda

¿Puedo poner varios gráficos por celda?

Probemos

[13]:
```
plt.plot(x, y)
plt.xlim((0., np.pi*2))

plt.plot(x, y)
plt.xlim((0., np.pi*3))
```

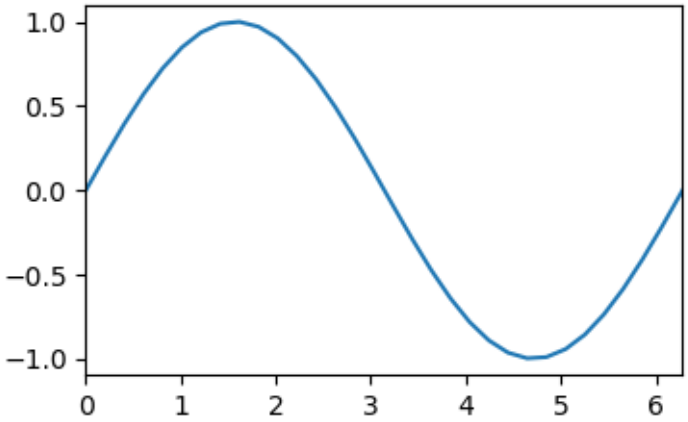
[13]: (0.0, 9.42477796076938)

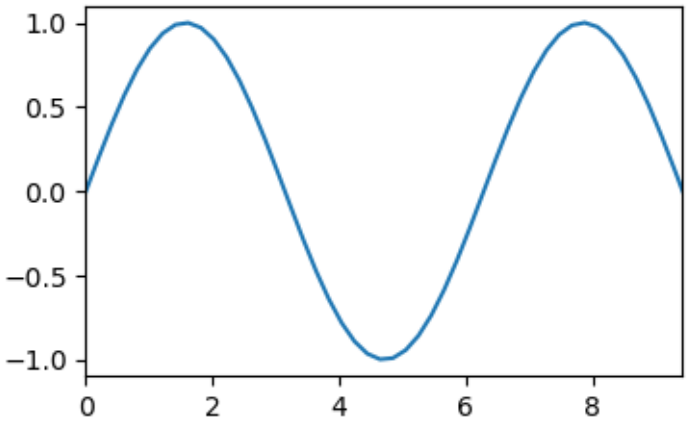


No funcionó, sólo veo el último dibujo. El primero se me perdió.

Si tengo más de una figura en una celda, tengo que "cerrar" el primero para que se grafique en la pantalla. Si no lo hago sólo veré el último de estos. Para cerrar el dibujo tengo que usar el comando **"plt.show( )"**

[14]:
```
plt.plot(x, y)
plt.xlim((0., np.pi*2))
plt.show()

plt.plot(x, y)
plt.xlim((0., np.pi*3))
plt.show()
```

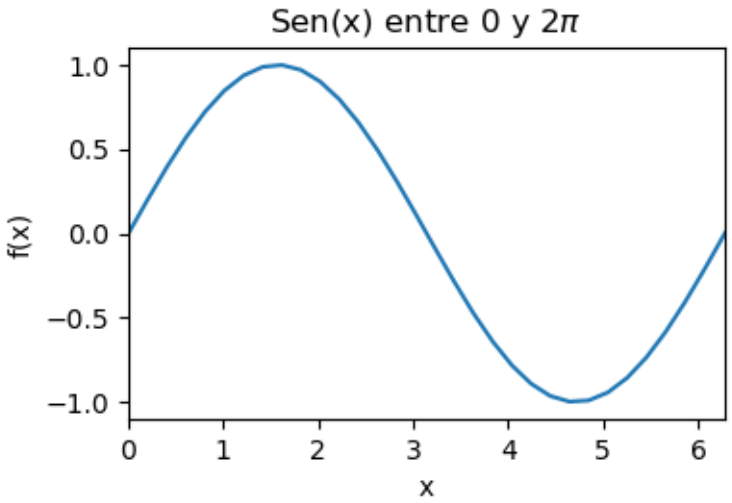




7 Nombre de los ejes y título de la figura

Para ello se utiliza el comando plt.title('texto') donde como string de texto se escribe el título. Los nombres de cada eje se indican con los comandos plt.xlabel('texto') y plt.ylabel('texto') para cada eje respectivamente.

[15]:
```
plt.plot(x, y)
plt.xlim((0., np.pi*2))
plt.title('Sen(x) entre 0 y $2\pi$')
plt.xlabel('x')
plt.ylabel('f(x)');
```



8 Documentación en línea

[16]:
```
help(plt.plot)
```

```
Help on function plot in module matplotlib.pyplot:

plot(*args, scalex=True, scaley=True, data=None, **kwargs)
    Plot y versus x as lines and/or markers.

    Call signatures::

        plot([x], y, [fmt], *, data=None, **kwargs)
        plot([x], y, [fmt], [x2], y2, [fmt2], ..., **kwargs)

    The coordinates of the points or line nodes are given by *x*, *y*.

    The optional parameter *fmt* is a convenient way for defining basic
```

```
formatting like color, marker and linestyle. It's a shortcut string
notation described in the *Notes* section below.

>>> plot(x, y)        # plot x and y using default line style and color
>>> plot(x, y, 'bo')  # plot x and y using blue circle markers
>>> plot(y)           # plot y using x as index array 0..N-1
>>> plot(y, 'r+')     # ditto, but with red plusses

You can use `.Line2D` properties as keyword arguments for more
control on the appearance. Line properties and *fmt* can be mixed.
The following two calls yield identical results:

>>> plot(x, y, 'go--', linewidth=2, markersize=12)
>>> plot(x, y, color='green', marker='o', linestyle='dashed',
...      linewidth=2, markersize=12)

When conflicting with *fmt*, keyword arguments take precedence.


**Plotting labelled data**

There's a convenient way for plotting objects with labelled data (i.e.
data that can be accessed by index ``obj['y']``). Instead of giving
the data in *x* and *y*, you can provide the object in the *data*
parameter and just give the labels for *x* and *y*::

>>> plot('xlabel', 'ylabel', data=obj)

All indexable objects are supported. This could e.g. be a `dict`, a
`pandas.DataFrame` or a structured numpy array.


**Plotting multiple sets of data**

There are various ways to plot multiple sets of data.

- The most straight forward way is just to call `plot` multiple times.
  Example:

  >>> plot(x1, y1, 'bo')
  >>> plot(x2, y2, 'go')

- If *x* and/or *y* are 2D arrays a separate data set will be drawn
  for every column. If both *x* and *y* are 2D, they must have the
  same shape. If only one of them is 2D with shape (N, m) the other
  must have length N and will be used for every data set m.

  Example:
```

```
  >>> x = [1, 2, 3]
  >>> y = np.array([[1, 2], [3, 4], [5, 6]])
  >>> plot(x, y)

  is equivalent to:

  >>> for col in range(y.shape[1]):
  ...     plot(x, y[:, col])

- The third way is to specify multiple sets of *[x]*, *y*, *[fmt]*
  groups::

  >>> plot(x1, y1, 'gˆ', x2, y2, 'g-')

  In this case, any additional keyword argument applies to all
  datasets. Also this syntax cannot be combined with the *data*
  parameter.

By default, each line is assigned a different style specified by a
'style cycle'. The *fmt* and line property parameters are only
necessary if you want explicit deviations from these defaults.
Alternatively, you can also change the style cycle using
:rc:`axes.prop_cycle`.


Parameters
----------
x, y : array-like or scalar
    The horizontal / vertical coordinates of the data points.
    *x* values are optional and default to ``range(len(y))``.

    Commonly, these parameters are 1D arrays.

    They can also be scalars, or two-dimensional (in that case, the
    columns represent separate data sets).

    These arguments cannot be passed as keywords.

fmt : str, optional
    A format string, e.g. 'ro' for red circles. See the *Notes*
    section for a full description of the format strings.

    Format strings are just an abbreviation for quickly setting
    basic line properties. All of these and more can also be
    controlled by keyword arguments.

    This argument cannot be passed as keyword.
```

```
    data : indexable object, optional
        An object with labelled data. If given, provide the label names to
        plot in *x* and *y*.

        .. note::
            Technically there's a slight ambiguity in calls where the
            second label is a valid *fmt*. ``plot('n', 'o', data=obj)``
            could be ``plt(x, y)`` or ``plt(y, fmt)``. In such cases,
            the former interpretation is chosen, but a warning is issued.
            You may suppress the warning by adding an empty format string
            ``plot('n', 'o', '', data=obj)``.

    Returns
    -------
    list of `.Line2D`
        A list of lines representing the plotted data.

    Other Parameters
    ----------------
    scalex, scaley : bool, default: True
        These parameters determine if the view limits are adapted to the
        data limits. The values are passed on to `autoscale_view`.

    **kwargs : `.Line2D` properties, optional
        *kwargs* are used to specify properties like a line label (for
        auto legends), linewidth, antialiasing, marker face color.
        Example::

        >>> plot([1, 2, 3], [1, 2, 3], 'go-', label='line 1', linewidth=2)
        >>> plot([1, 2, 3], [1, 4, 9], 'rs', label='line 2')

        If you specify multiple lines with one plot call, the kwargs apply
        to all those lines. In case the label object is iterable, each
        element is used as labels for each set of data.

        Here is a list of available `.Line2D` properties:

        Properties:
        agg_filter: a filter function, which takes a (m, n, 3) float array and a
dpi value, and returns a (m, n, 3) array
        alpha: scalar or None
        animated: bool
        antialiased or aa: bool
        clip_box: `.Bbox`
        clip_on: bool
        clip_path: Patch or (Path, Transform) or None
        color or c: color
```

```
        dash_capstyle: `.CapStyle` or {'butt', 'projecting', 'round'}
        dash_joinstyle: `.JoinStyle` or {'miter', 'round', 'bevel'}
        dashes: sequence of floats (on/off ink in points) or (None, None)
        data: (2, N) array or two 1D arrays
        drawstyle or ds: {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-
post'}, default: 'default'
        figure: `.Figure`
        fillstyle: {'full', 'left', 'right', 'bottom', 'top', 'none'}
        gid: str
        in_layout: bool
        label: object
        linestyle or ls: {'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
        linewidth or lw: float
        marker: marker style string, `~.path.Path` or `~.markers.MarkerStyle`
        markeredgecolor or mec: color
        markeredgewidth or mew: float
        markerfacecolor or mfc: color
        markerfacecoloralt or mfcalt: color
        markersize or ms: float
        markevery: None or int or (int, int) or slice or list[int] or float or
(float, float) or list[bool]
        path_effects: `.AbstractPathEffect`
        picker: float or callable[[Artist, Event], tuple[bool, dict]]
        pickradius: float
        rasterized: bool
        sketch_params: (scale: float, length: float, randomness: float)
        snap: bool or None
        solid_capstyle: `.CapStyle` or {'butt', 'projecting', 'round'}
        solid_joinstyle: `.JoinStyle` or {'miter', 'round', 'bevel'}
        transform: unknown
        url: str
        visible: bool
        xdata: 1D array
        ydata: 1D array
        zorder: float

    See Also
    --------
    scatter : XY scatter plot with markers of varying size and/or color (
        sometimes also called bubble chart).

    Notes
    -----
    **Format Strings**

    A format string consists of a part for color, marker and line::

        fmt = '[marker][line][color]'
```

Each of them is optional. If not provided, the value from the style
cycle is used. Exception: If ``line`` is given, but no ``marker``,
the data will be a line without markers.

Other combinations such as ``[color][marker][line]`` are also
supported, but note that their parsing may be ambiguous.

Markers

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

Line Styles

=============   ==============================
character   description
=============   ==============================
``'-'``   solid line style
``'--'``   dashed line style
``'-.'``   dash-dot line style

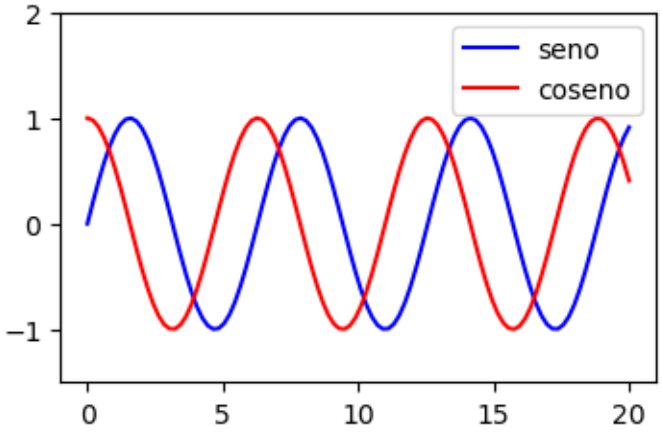
```
``':'``          dotted line style
=============    ================================

Example format strings::

    'b'    # blue markers with default shape
    'or'   # red circles
    '-g'   # green solid line
    '--'   # dashed line with default color
    '^k:'  # black triangle_up markers connected by a dotted line

**Colors**

The supported color abbreviations are the single letter codes

=============    ================================
character        color
=============    ================================
``'b'``          blue
``'g'``          green
``'r'``          red
``'c'``          cyan
``'m'``          magenta
``'y'``          yellow
``'k'``          black
``'w'``          white
=============    ================================

and the ``'CN'`` colors that index into the default property cycle.

If the color is the only part of the format string, you can
additionally use any  `matplotlib.colors` spec, e.g. full names
(``'green'``) or hex strings (``'#008000'``).
```

8.1 Leyendas

A cada curva o puntos específicos dentro de una figura se los puede diferenciar con una leyenda que se escribe con el comando que los dibuja. Y luego al final, indicando el lugar de la figura donde se pondrá esta leyenda.

[17]:
```
x = np.linspace(0, 20, 100)
y1 = np.sin(x)
y2 = np.cos(x)

plt.plot(x, y1, '-b', label='seno')
plt.plot(x, y2, '-r', label='coseno')
plt.legend(loc='upper right')
```

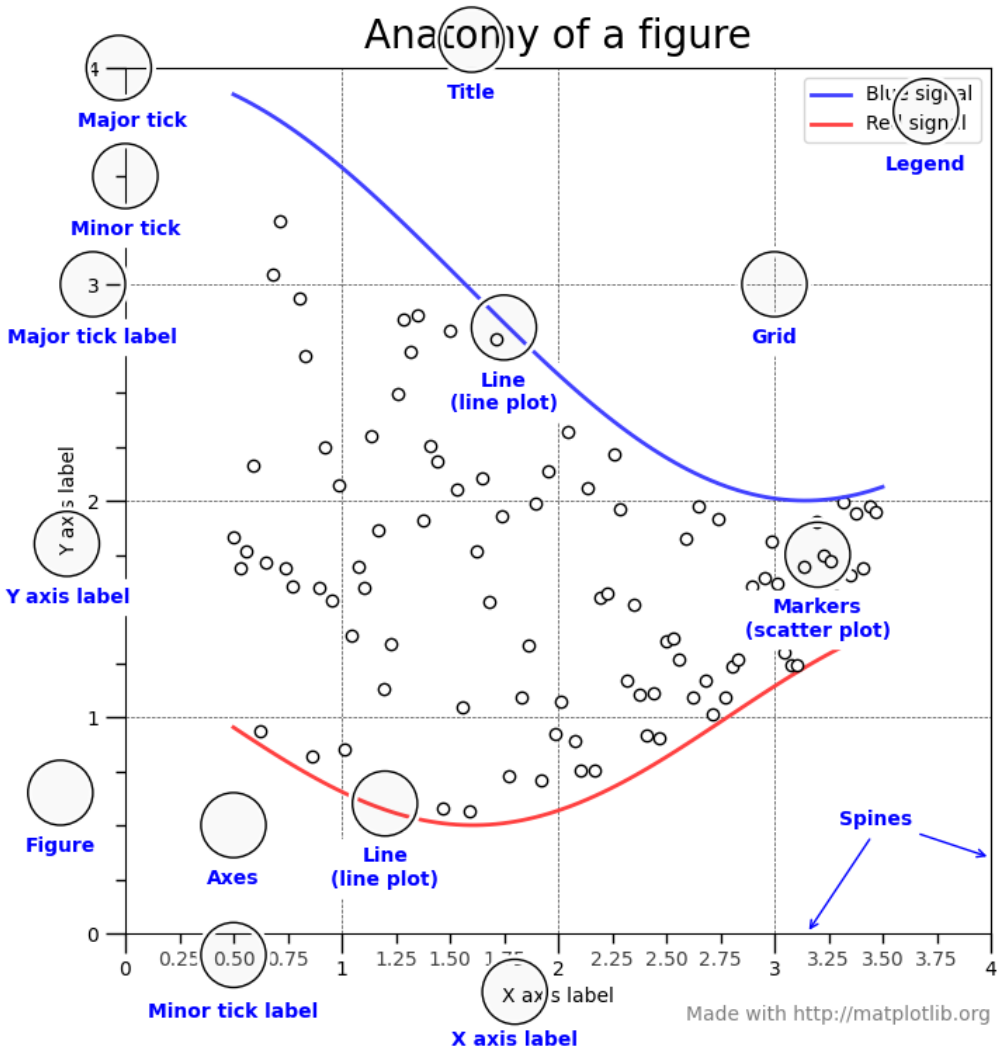
```
plt.ylim((-1.5, 2.0));
```



En este caso indicamos con rojo el sin(x) en azul y el cos(x) en rojo. Luego la leyenda la dibujamos arriba (“upper”) y a la derecha (“right”) con el comando plt.legend( ).

9 Nomenclatura de las distintas partes de un dibujo

Afortunamente la gente de Matplotlib a generado este dibujo que da nombre a las partes de un gráfico del tipo de los que se usan ciencia. Con estos nombres es muy fácil buscar (o adivinar) el comando que se necesita para su modificación para que el dibujo quede a gusto del usuario o bien de los requerimientos de las revistas o congresos.

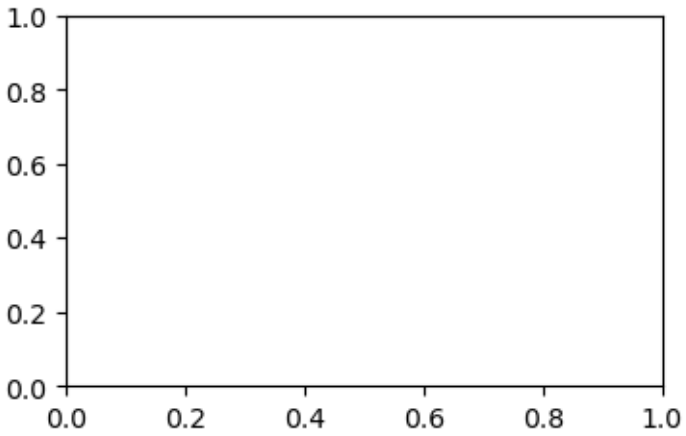


10 La figura como Objeto Python

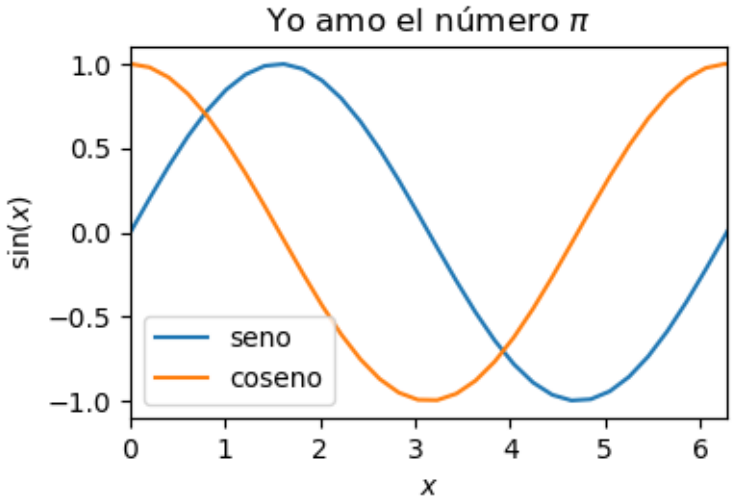
(Todo es un objeto en Python)

Puedo crear el objeto y luego modificar sus parámetros y agregar lo que quiero graficar. Suele ser más laborioso (no mucho) y las órdenes no son las mismas que las que vimos hasta ahora, pero muy parecidas. La forma de trabajo es que por un lado tengo la figura en sí, y por otro los ejes (ya que puede haber varios) y modifico cada uno con lo que necesito

[18]:
```
fig = plt.figure()  # una nueva ventana con una figura
ax = fig.add_subplot(1, 1, 1)  # especifico (número de filas, columnas, número␣
↪de figura (axn))
```



[19]:
```
fig, ax = plt.subplots()  # Creo la figura como dos objetos: figura en si y ejes
ax.plot(x, y1)
ax.plot(x, y2)
ax.set_xlim(0., 2*np.pi)
ax.legend(['seno', 'coseno'], loc='best')
ax.set_xlabel("$x$")
ax.set_ylabel("$\sin(x)$")
ax.set_title("Yo amo el número $\pi$");
```



[20]:
```
# Lo que sigue despliega una cantidad my grande de información, por ahora lo␣
↪tengo comentado
#help(ax)
```

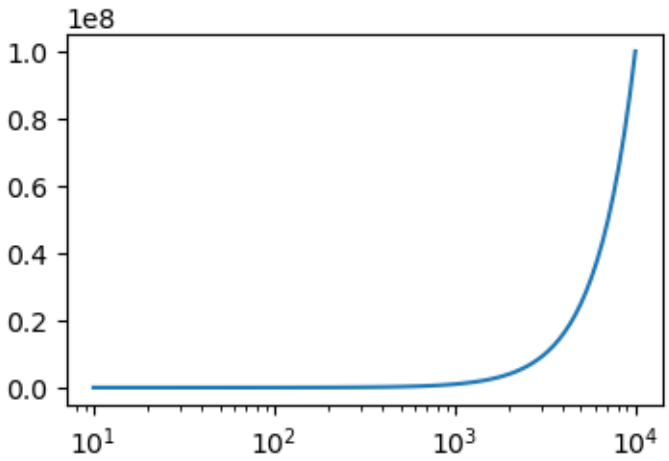
11 Plots Logarítmicos

Los gráficos logarítmicos se pueden hacer con un eje logaritmico y otro lineal o con ambos ejes logarítmicos

- Con un eje logarímtico se lo llama semilog"n"() donde n es x o y según sean las abscisas o las ordenadas
- Con los dos ejes logarítmico es loglog()

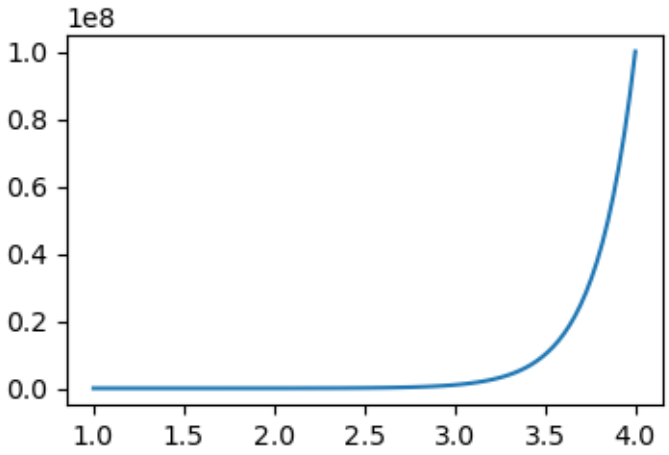
Si el eje X es el logarítmico puede hacer el gráfico así, donde queda muy claro su escala

[21]:
```
xl = np.logspace(1, 4, 100)
fig, ax = plt.subplots()
ax.semilogx(xl, xl**2);
```



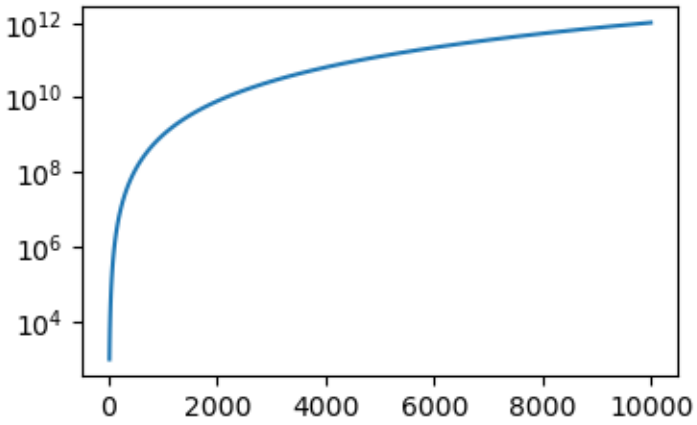
O dejarlo que ponga del valor del logaritmo en el eje directamente, auqnue en esta forma queda menos claro que clase de magnitud es.

[22]:
```
xl = np.logspace(1, 4, 100)
fig, ax = plt.subplots()
ax.plot(np.log10(xl), xl**2);
```

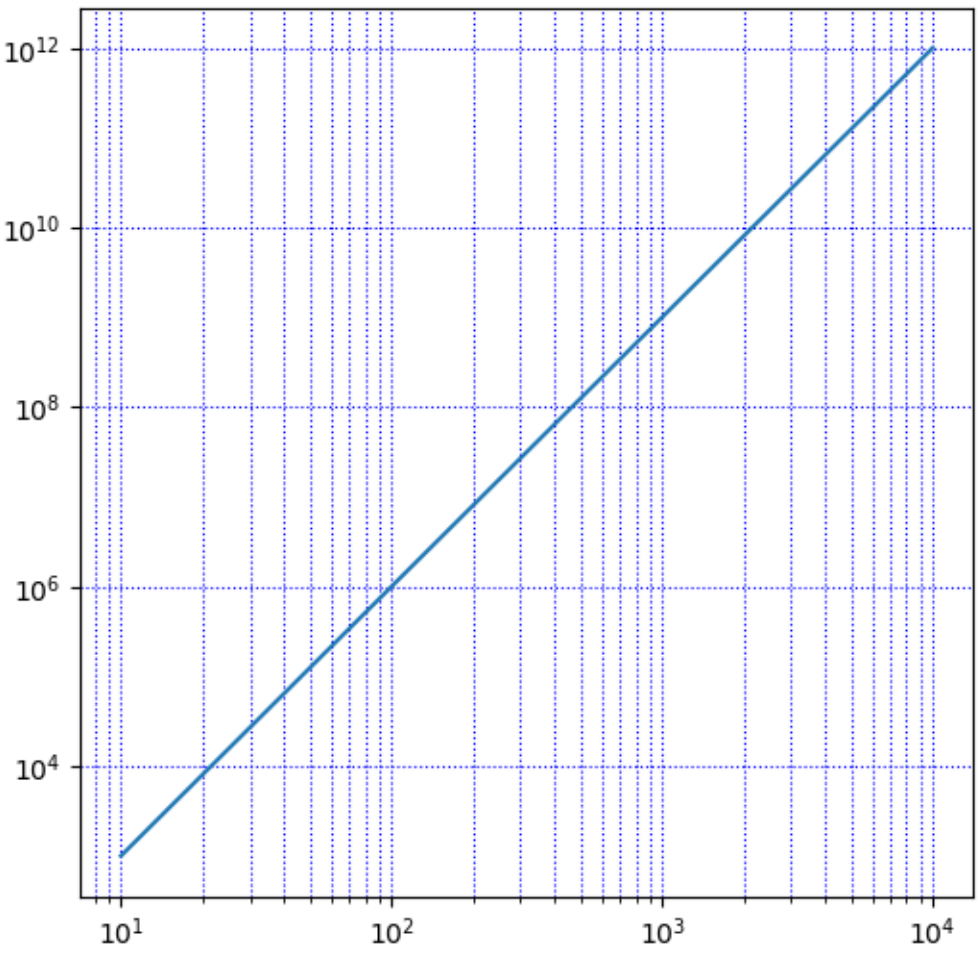


Puedo hacer lo mismo para el eje de las ordenadas (note que cambio entonces "plt.semilog**X**" por "plt.semilog**Y**")

[23]:
```
fig, ax = plt.subplots()
ax.semilogy(xl, xl**3);
```



[24]:
```
fig, ax = plt.subplots(figsize=(6,6))
ax.loglog(xl, xl**3);
ax.grid(True,which="both",ls=":", c='blue')
```

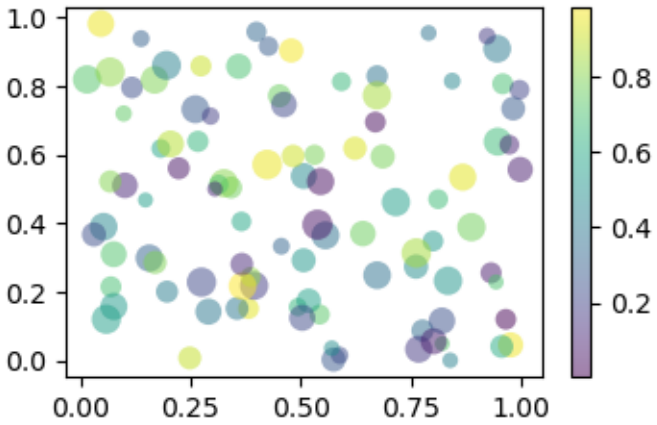


12 Scatter o gráficos con puntos

Hay mucha libertad para elegir el símbolo para el punto, su tamaño y su color veamos un dibujo donde elijo las coordenadas, tamaño y color al azar.

[25]:
```
xr = np.random.rand(100)
yr = np.random.rand(100)
cr = np.random.rand(100)
sr = np.random.rand(100)

fig, ax = plt.subplots()
sc = ax.scatter(xr, yr, c=cr, s=30+sr*100, edgecolor='none', alpha=0.5) #␣
 ↪colores y tamaños dependen
                                                                        # del␣
 ↪valor de las variables
```

```
                                                                    # que en␣
 ↪este caso son números
                                                                    # al azar
fig.colorbar(sc);
```

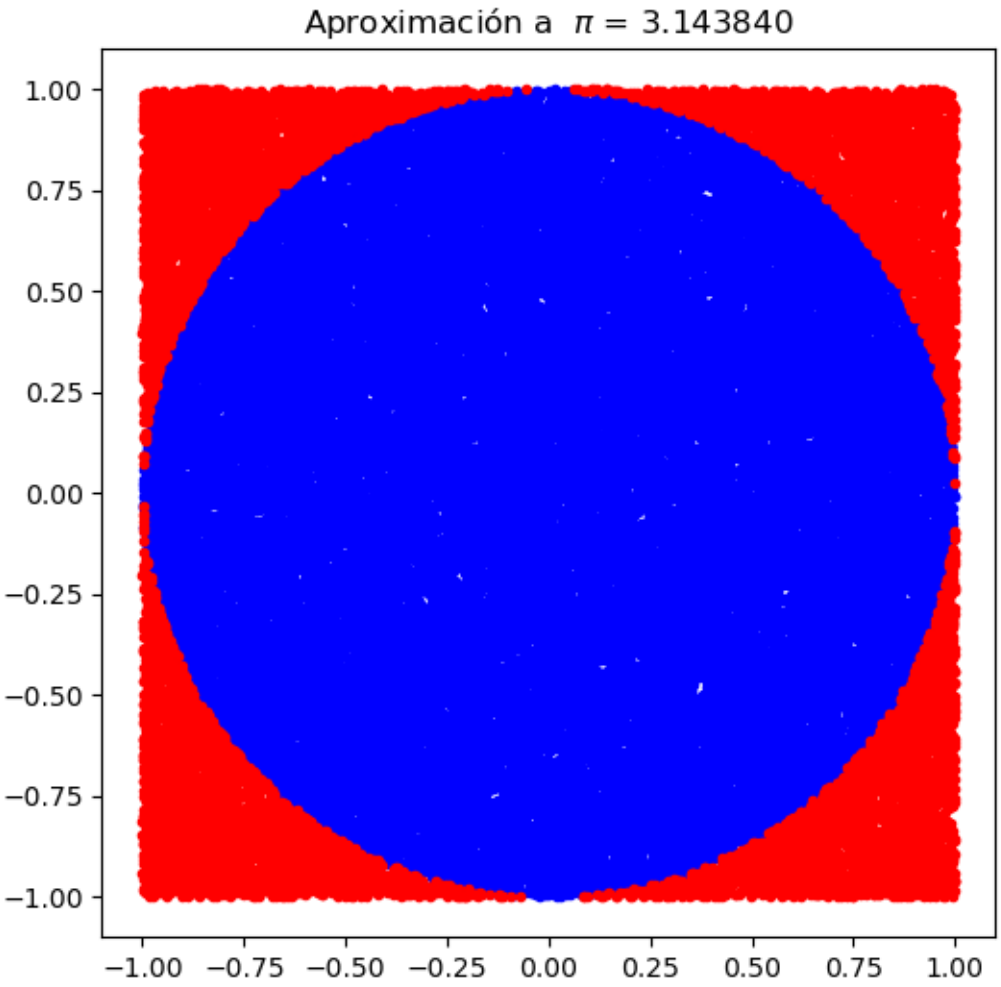


Calculando π

Con el método de las piedras. Tira 5000 de estas y me fijo

[26]:
```
npts = 50000

xs = 2*np.random.rand(npts)-1
ys = 2*np.random.rand(npts)-1

r = xs**2+ys**2

ninside = (r<1).sum()

plt.figure(figsize=(6,6)) # Para que la figura sea cuadrada

plt.title("Aproximación a  $\pi$ = %f" % (4*ninside/float(npts)))
plt.plot(xs[r<1],ys[r<1],'b.')
plt.plot(xs[r>1],ys[r>1],'r.')


print("Pi da:", ninside/npts*4)
```

```
Pi da: 3.14384
```



13 Parámetros del gráfico

Para dejar fijos algunos valores "domésticos" del dibujo tenemos dos formas de establecerlos:

Con la orden **plc.rc('Lo que queremos cambiar', nuevo valor)**, donde nuevo valor puede ser una variable en la que está el valor del nuevo parámetro. O la orden:

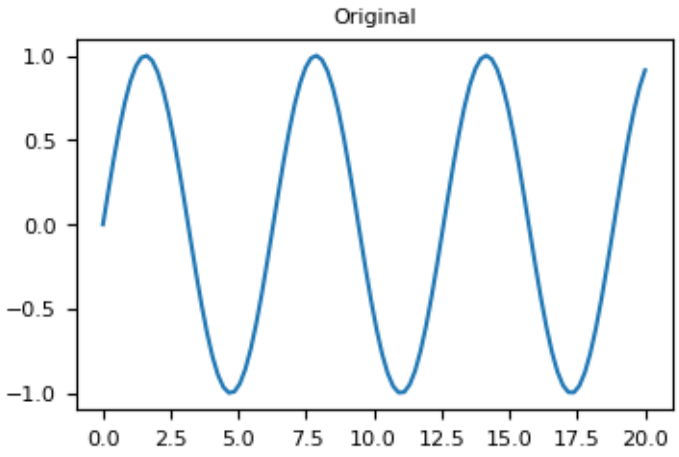
import matplotlib # cargo TODA la matplotlib

matplotlib.rc('Lo que queremos cambiar', nuevo valor) # modifico un parámetro

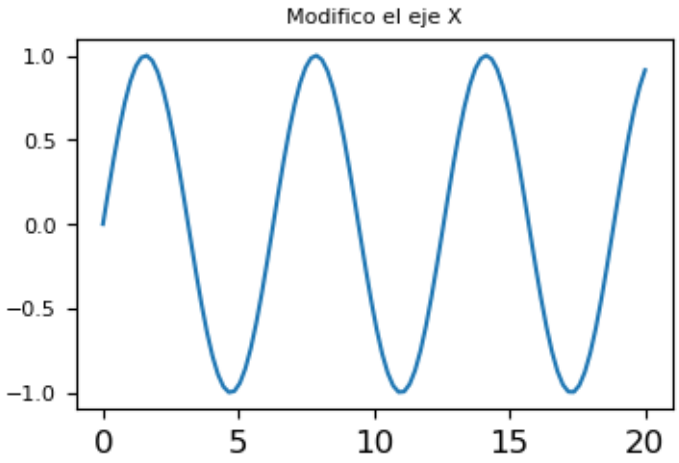
Veamos algunos ejemplos:

[27]:
```
SMALL_SIZE = 8
MEDIUM_SIZE = 10
BIGGER_SIZE = 12
```

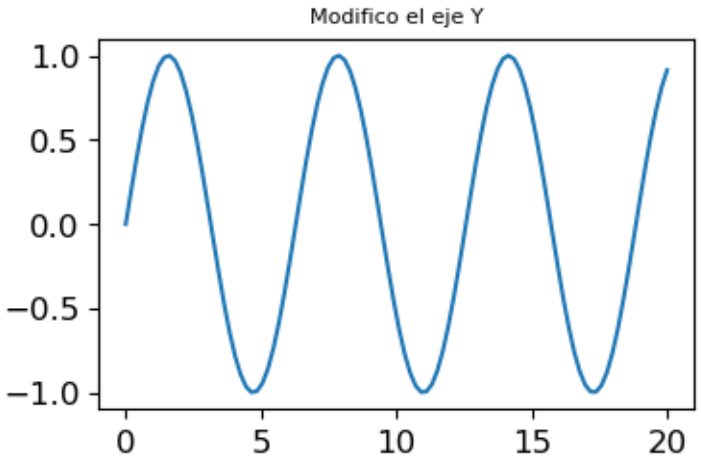
```
plt.rc('font', size=SMALL_SIZE)          # controls default text sizes
plt.rc('axes', titlesize=SMALL_SIZE)     # fontsize of the axes title
plt.rc('axes', labelsize=MEDIUM_SIZE)    # fontsize of the x and y labels
plt.rc('xtick', labelsize=SMALL_SIZE)    # fontsize of the tick labels
plt.rc('ytick', labelsize=SMALL_SIZE)    # fontsize of the tick labels
plt.rc('legend', fontsize=SMALL_SIZE)    # legend fontsize
plt.rc('figure', titlesize=BIGGER_SIZE)  # fontsize of the figure title

# y  lo mismo se puede hacar para los demás parámetros del
# gráfico

## Ejemplo:
plt.plot(x,np.sin(x))
plt.title('Original')
plt.show()

print("")
print("Hago un cambio en los números de los ticks del eje X")
plt.rc('xtick', labelsize=BIGGER_SIZE)
plt.title('Modifico el eje X')
plt.plot(x,np.sin(x))
plt.show()

print("")
print("y si ahora cambio el eje Y")
plt.rc('ytick', labelsize=BIGGER_SIZE)
plt.title('Modifico el eje Y')
plt.plot(x,np.sin(x))
plt.show()

# O también lo podría hacer de la forma

import matplotlib
SMALL_SIZE = 8
matplotlib.rc('font', size=SMALL_SIZE)
matplotlib.rc('axes', titlesize=SMALL_SIZE)
```



Hago un cambio en los números de los ticks del eje X



y si ahora cambio el eje Y

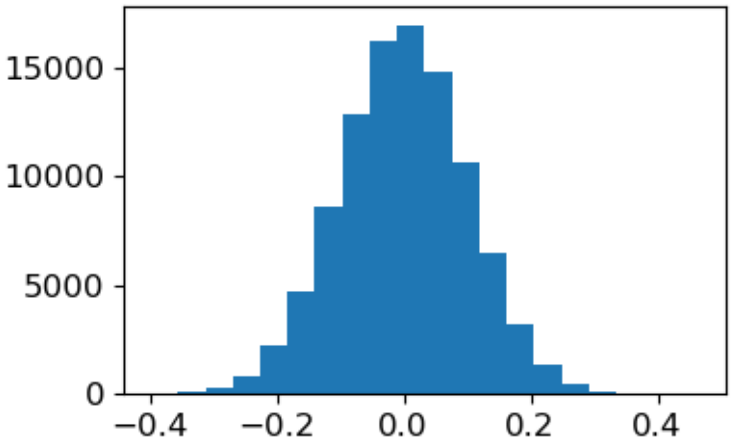


14 Cálculos y dibujos en Matplolib

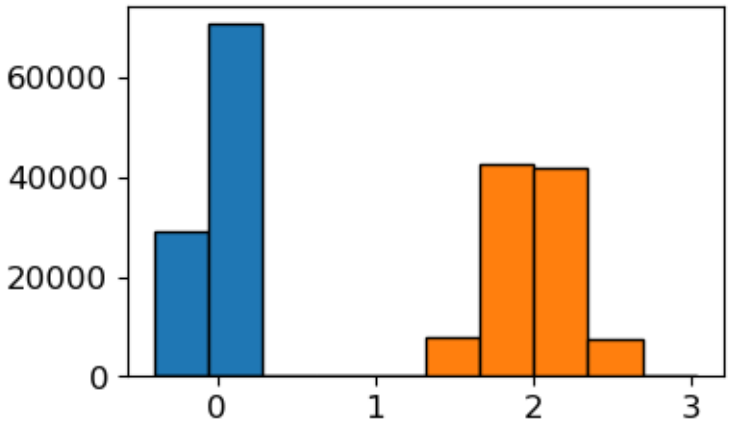
Matplotlib tiene también capacidad de hacer cálculos para dibujos específicos. Por ejemplo en el caso de calcular un histograma. Los histogramas son graficos de barras, donde cada barra es la cuenta de cuentos eventos se producen en un cierto intervalo.

Veamos como es esto:

[28]:
```
N_points = 100000
n_bins = 20

# Generemos dos distribuciones de números la azar
# gaussianas (Son los que tiene forma de campana)

dist1 = np.random.normal(0,0.1,N_points)
dist2 = 0.4 * np.random.normal(0,0.6,N_points) + 2

# Con esos datos hago el dibujo de un histograma

fig, axs = plt.subplots(1, 1, sharey=True, tight_layout=True)

axs.hist(dist1, bins=n_bins);
```



[29]:
```
bins = np.histogram(np.hstack((dist1, dist2)), bins=10)[1]
plt.hist(dist1, bins, edgecolor='black')
plt.hist(dist2, bins, edgecolor='black');
```

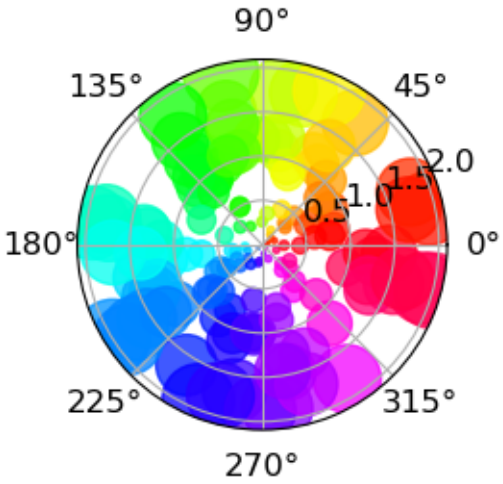


15 Gráficos con distribución de puntos en coordenadas Polares

Este es otro de los gráficos que podemos hacer Para este dibujo enero dos coordenadas r y tita con 150 números al azar cada una

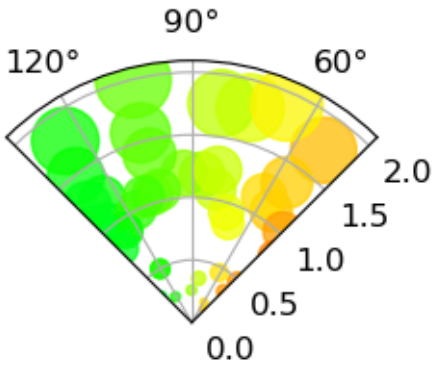
[30]:
```
N=150
r = 2* np.random.rand(N)
tita = 2 * np.pi * np.random.rand(N)

# Genero un área para cada punto en función de la coordenada r
area = 200* r**2
```

```
# Genero colores que los hago depender del la coordenada anfgular
color = tita

# Creo la figura
fig= plt.figure()

# Y en la figura creo un plot polar
ax = fig.add_subplot(projection='polar') #<-- Notar que uso la proyección polar
                                         # para activar esta propiedad

# la dibujo

ax.scatter(tita, r, c=color, s=area, cmap='hsv', alpha=0.75);
```



Incluso podría tomar una región determinado por el ángulo

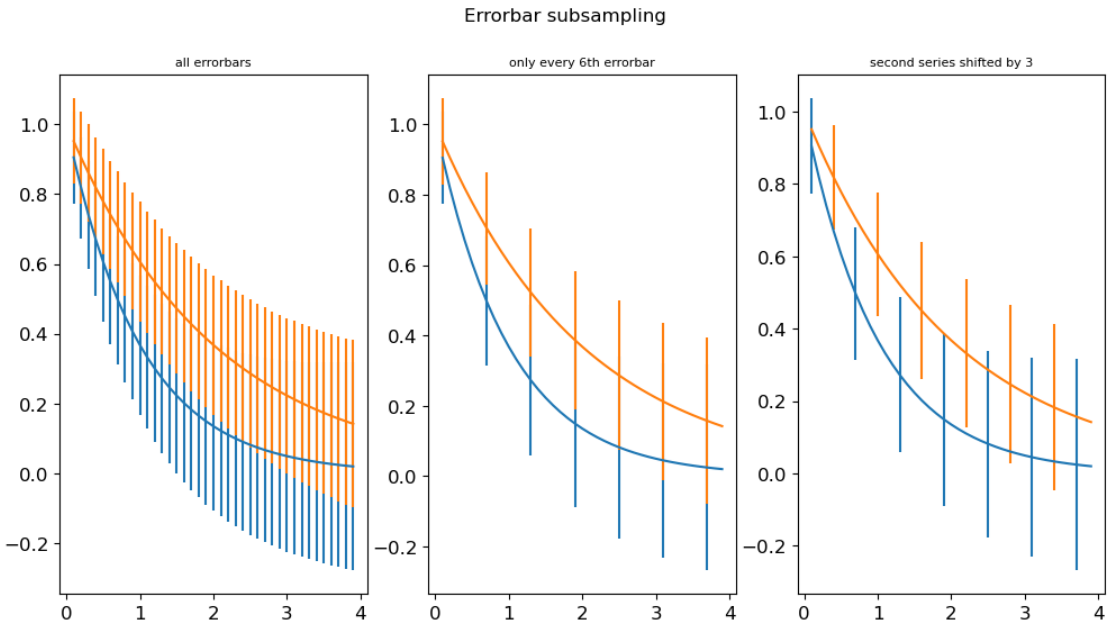
[31]:
```
fig= plt.figure()
ax = fig.add_subplot(projection='polar')
ax.set_thetamin(45)
ax.set_thetamax(135)
ax.scatter(tita, r, c=color, s=area, cmap='hsv', alpha=0.75);
```



16 Gráficos múltiples usando un sólo Objeto

[32]:
```
import numpy as np
import matplotlib.pyplot as plt

# example data
x = np.arange(0.1, 4, 0.1)
y1 = np.exp(-1.0 * x)
y2 = np.exp(-0.5 * x)

# example variable error bar values
y1err = 0.1 + 0.1 * np.sqrt(x)
y2err = 0.1 + 0.1 * np.sqrt(x/2)


fig, (ax0, ax1, ax2) = plt.subplots(nrows=1, ncols=3, sharex=True,
                                    figsize=(12, 6))

ax0.set_title('all errorbars')
ax0.errorbar(x, y1, yerr=y1err)
ax0.errorbar(x, y2, yerr=y2err)

ax1.set_title('only every 6th errorbar')
ax1.errorbar(x, y1, yerr=y1err, errorevery=6)
ax1.errorbar(x, y2, yerr=y2err, errorevery=6)

ax2.set_title('second series shifted by 3')
ax2.errorbar(x, y1, yerr=y1err, errorevery=(0, 6))
ax2.errorbar(x, y2, yerr=y2err, errorevery=(3, 6))

fig.suptitle('Errorbar subsampling')
```

```
plt.show()
```


Errorbar subsampling
all errorbars
only every 6th errorbar
second series shifted by 3
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
0
1
2
3
4