Matplotlib: visualización gráfica

De Matplotlib, Jupyter y las gráficas mostradas

En todo este capítulo se asumirá que las siguientes líneas de código han sido ejecutadas, previamente, para cada porción de código:

In [2]:
```
import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline
```

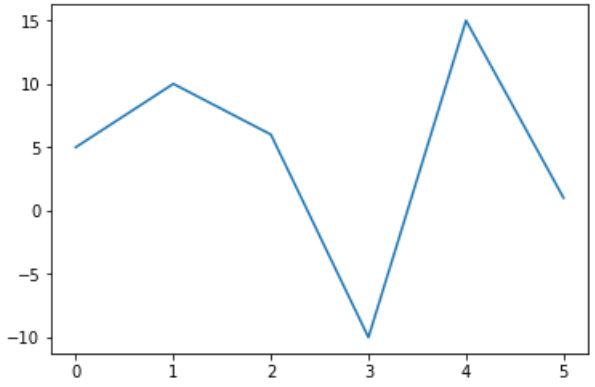
Sí usted ejecuta Python/Matplotlib dentro de un entorno diferente a Jupyter, deberá evitar colocar la instrucción `%matplotlib inline` y en su lugar colocar la instrucción `plt.show()` al final de cada código, para que se le muestren las gráficas correspondientes.

Una primera aproximación

Una de las características de Matplotlib es la facilidad con la que se puede comenzar a trazar gráficas, vea el siguiente código:

In [3]:
```
plt.plot([5,10,6,-10,15,1])
```

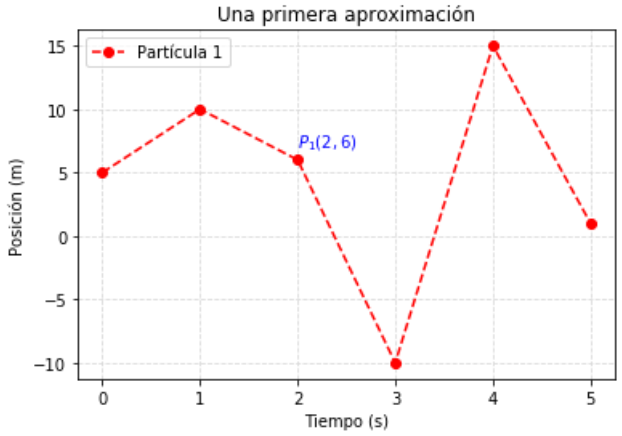
Out[3]:
```
[<matplotlib.lines.Line2D at 0x1e13b8430f0>]
```



El código anterior produce la gráfica mostrada en la figura. Como puede observar son solamente tres líneas de instrucciones, la primera sirve para importar el módulo \code{pyplot} de Matplotlib, el cual contiene muchas de las funciones útiles para el trazo de gráficas; la segunda línea ejecuta la función \code{plot} pasando como argumento una lista de valores numéricos, y finalmente la tercera línea se encarga de mostrar el elemento gráfico resultante en una ventana.

Un resultado un poco más *trabajado* se obtiene con el siguiente código:

In [4]:
```
plt.plot([0,1,2,3,4,5], [5,10,6,-10,15,1], 'r--o', label="Partícula 1")
plt.xlabel("Tiempo (s)")
plt.ylabel("Posición (m)")
plt.title("Una primera aproximación")
plt.text(2,7,"$ P_1 (2,6) $", color="b")
plt.legend()
plt.grid(ls="--", color="#dadada")
```



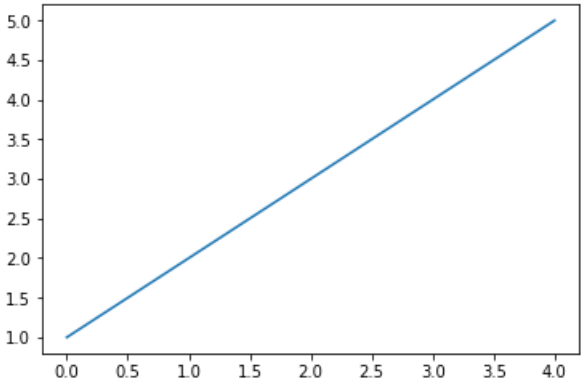
La función `plot`

La función `plot` está contenida en el módulo `pyplot` y básicamente con esta se produce cualquier gráfica bidimensional en coordenadas rectangulares. Esta función soporta varias maneras de ejecutarla dependiendo la cantidad de argumentos que se pasen.

La forma más básica de la función `plot` es pasarle un sólo argumento, por ejemplo:

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

Out[5]:
```
[<matplotlib.lines.Line2D at 0x1e13bb95198>]
```

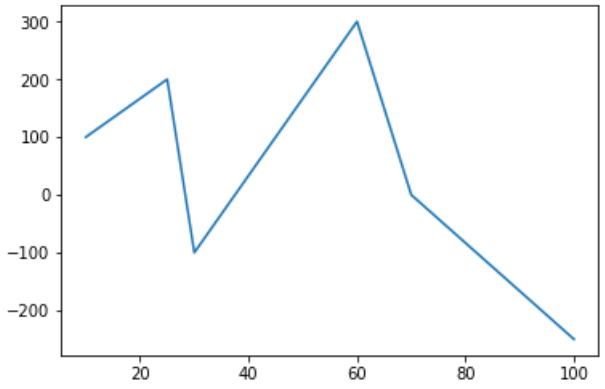


Al pasarle un sólo argumento, este se toma como los valores de la coordenada vertical, y se asume que la horizontal varía de 0 a `N-1`, donde `N` es el número de elementos contenidos en la lista de valores que se introducen.

La sintaxis más habitual es introducir dos argumentos, donde el primero contiene una lista `X` que define los valores de la coordenada horizontal, y el segundo una lista `Y` correspondiente a los valores de la coordenada vertical, por ejemplo:

In [6]:
```
plt.plot([10,25,30,60,70,100], [100,200,-100,300,0,-250])
```

Out[6]:
```
[<matplotlib.lines.Line2D at 0x1e13bc3ab38>]
```



Graficando funciones matemáticas

En matemáticas una función es una relación que asigna elementos de un conjunto de manera unívoca a otro conjunto. Usualmente una función matemática se puede representar mediante una gráfica en coordenadas cartesianas, colocando uno de los conjuntos en el eje horizontal y el otro en el vertical.

Utilizando Python, y de manera específica la librería NumPy, se pueden evaluar las funciones matemáticas en un intervalo determinado y en una cantidad finita de puntos. Por ejemplo, suponga que se requieren calcular todos los pares coordenados correspondientes a la función $y = \cos x$ en el intervalo $0 \leq x \leq 5$, en Python se tendría que definir como:

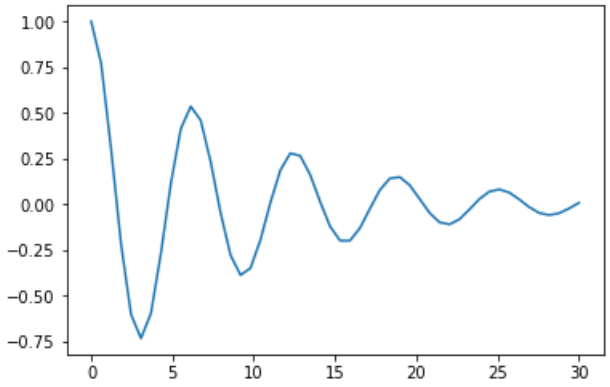
In [7]:
```
x = np.linspace(0,5)
y = np.cos(x)
```

Las variable `x` es un arreglo de NumPy que contiene 50 valores linealmente equiespaciados entre 0 y 5, la variable `y` es también un arreglo de NumPy que resulta de aplicar la función coseno a cada valor de `x`.

De manera similar a lo anterior se procederá a definir y graficar la función $y = e^{-0.1x}\cos x$ en el intervalo $0 \leq x \leq 30$:

In [8]:
```
x = np.linspace(0, 30)
y = np.exp(-0.1*x)*np.cos(x)
plt.plot(x,y)
```

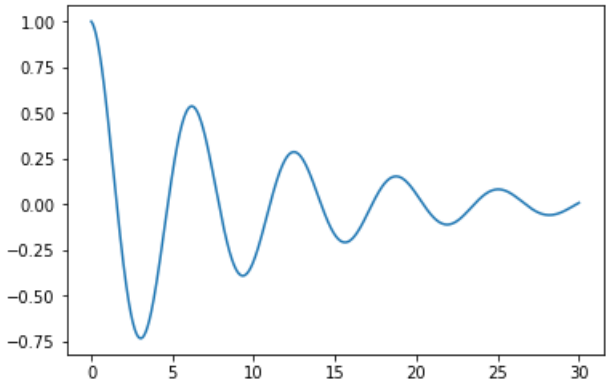
Out[8]:
```
[<matplotlib.lines.Line2D at 0x1e13bc90cf8>]
```



La cantidad de puntos a evaluar es una cuestión muy importante, ya que de esto depende la correcta visualización del comportamiento de una función. Naturalmente, entre más puntos evaluados mejor será la apreciación que se tenga de la curva en cuestión, pero implica un mayor gasto de memoria para guardar y evaluar todos los datos. Enseguida se muestra la misma función graficada en el mismo intervalo pero con 1000 y 5 puntos evaluados de manera respectiva, notará la diferencia entre los casos, es evidente que en el caso de los 5 puntos *se pierde* muchísima información.

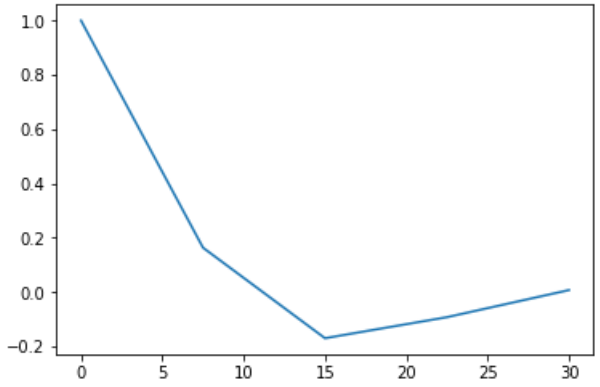
In [9]:
```
# Con 1000 puntos evaluados
x = np.linspace(0, 30, 1000)
y = np.exp(-0.1*x)*np.cos(x)
plt.plot(x,y)
```

Out[9]:
```
[<matplotlib.lines.Line2D at 0x1e13bcefda0>]
```



In [10]:
```
# Con 5 puntos evaluados
x = np.linspace(0, 30, 5)
y = np.exp(-0.1*x)*np.cos(x)
plt.plot(x,y)
```

Out[10]:
```
[<matplotlib.lines.Line2D at 0x1e13bd4ce48>]
```



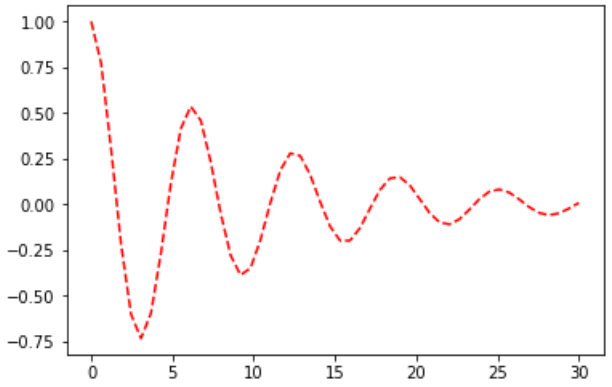
Modificando el color, estilos y grosor de línea

La función `plot` acepta argumentos adicionales que sirven para modificar y controlar características de la línea que se grafica.

Se puede pasar un tercer argumento que contenga una combinación de color y estilo de línea. Por ejemplo:

In [11]:
```
x = np.linspace(0, 30)
y = np.exp(-0.1*x)*np.cos(x)
plt.plot(x, y, "r--")
```

Out[11]:
```
[<matplotlib.lines.Line2D at 0x1e13bdacbe0>]
```

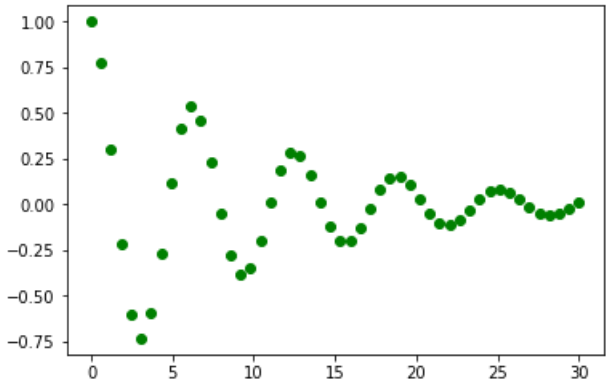


El código anterior genera una gráfica con una línea en color rojo (`r`) y un estilo de línea discontinua (`--`).

Si en lugar del string `--` se coloca `go` , se obtiene una gráfica como la mostrada enseguida, podrá inferir que `g` refiere al color verde (green) y `o` justamente al uso de esta como símbolo para representar cada punto.

In [12]:
```
x = np.linspace(0, 30)
y = np.exp(-0.1*x)*np.cos(x)
plt.plot(x, y, "go")
```

Out[12]:
```
[<matplotlib.lines.Line2D at 0x1e13be0dd68>]
```

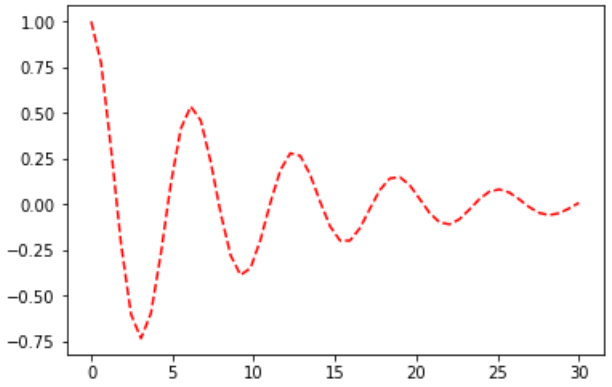


En https://matplotlib.org/api/markers_api.html (https://matplotlib.org/api/markers_api.html) se muestra una tabla con los símbolos (markers) disponibles para utilizar en la función `plot` . En https://matplotlib.org/api/colors_api.html (https://matplotlib.org/api/colors_api.html) puede consultar información respecto a los colores que puede abreviar mediante un sólo caracter.

Además de la forma anterior, también es posible especificar el color y estilo de línea utilizando *keyword arguments*, por ejemplo:

In [14]:
```
x = np.linspace(0, 30)
y = np.exp(-0.1*x)*np.cos(x)
plt.plot(x, y, linestyle="--", color="r")
```

Out[14]:
```
[<matplotlib.lines.Line2D at 0x1e13be7e860>]
```

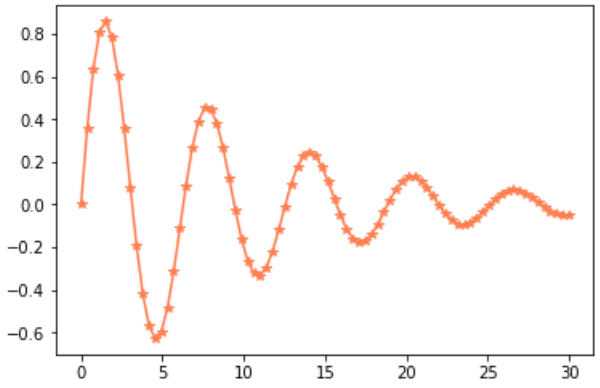


En ambos casos se especifica un cierto estilo de línea y color, con la diferencia notoria de la sintaxis.

Utilizar *keyword arguments* es una manera más general, puesto que la definición con strings no funciona para los casos en que se requieren colores que no se pueden especificar con un sólo caracter, por ejemplo, Matplotlib dispone de un color llamado `coral` y este no puede ser invocado mediante un sólo caracter, hace falta escribir todo el nombre.

In [24]:
```
x = np.linspace(0, 30, 80)
y = np.exp(-0.1*x)*np.sin(x)
plt.plot(x, y, linestyle="-", color="coral", marker="*")
```

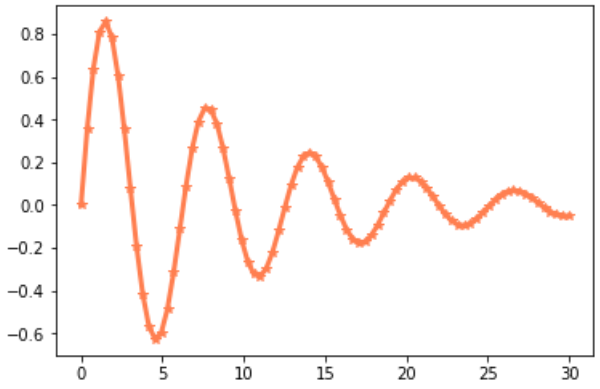
Out[24]:
```
[<matplotlib.lines.Line2D at 0x1e13d17d5c0>]
```



El ancho de línea se puede controlar mediante el *keyword argument* `linewidth` , por ejemplo;

In [27]:
```
plt.plot(x, y, linestyle="-", color="coral", marker="*", linewidth=3)
```

Out[27]:
```
[<matplotlib.lines.Line2D at 0x1e13d29c898>]
```



Título de gráfica, etiquetas de ejes y nombres de curvas

Por su naturaleza las gráficas nos sirven para presentar y/o visualizar información de ciertos datos, para lo cual se hace necesario especificar información descriptiva de lo que se muestra. Es muy común que se agreguen etiquetas a los ejes horizontal y vertical, así como el nombre de gráfica. Además, si se está graficando más de una curva, se hace necesario especificar a qué refiere cada una de ellas.

Por ejemplo, observe el siguiente código y la gráfica producida:

In [29]:
```
x = np.linspace(0, 30, 500)
y1 = np.exp(-0.1*x)*np.cos(x)
y2 = np.exp(-0.2*x)*np.sin(x)
plt.plot(x, y1, "b-", label="Partícula 1")
plt.plot(x, y2, "r-", label="Partícula 2")
plt.xlabel("Tiempo (s)")
plt.ylabel("Posición (mm)")
plt.title("Gráfica de posición")
plt.legend()
```

Out[29]:
```
<matplotlib.legend.Legend at 0x1e13d362390>
```

La instrucción `xlabel` coloca una etiqueta al eje horizontal, de manera similar `ylabel` lo hace para el eje vertical. Con `title` adicionamos un título a la gráfica. La instrucción `legend` sirve para colocar el recuadro con *nombre* asignado a cada curva mediante el *keyword argument* `label` .

Anotaciones

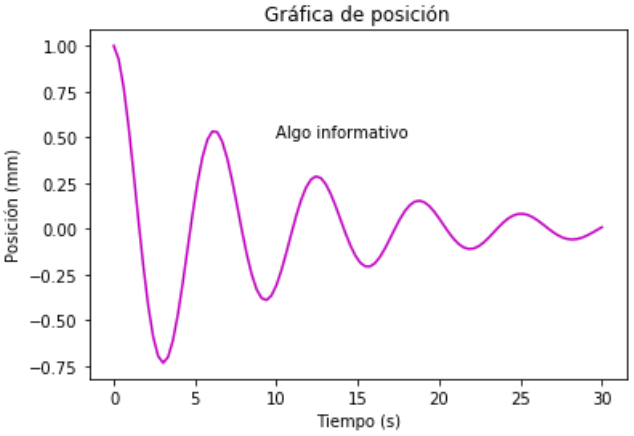
Con anotaciones nos referimos a cualesquiera texto que se coloque dentro del *Axes* de Matplotlib. Usualmente utilizadas para indicar ciertas características partículares en una gráfica, o bien alguna nota informativa al respecto. La función base para realizar este tipo de tareas es `text`. La sintaxis más simple de `text` es:

```
plt.text(px, py, texto)
```

Donde `px` y `py` denotan las coordenadas en donde se colocará la anotación indicado en `texto`. Veamos un ejemplo:

In [36]:
```
x = np.linspace(0, 30, 100)
y = np.exp(-0.1*x)*np.cos(x)
plt.plot(x, y, "m")
plt.xlabel("Tiempo (s)")
plt.ylabel("Posición (mm)")
plt.title("Gráfica de posición")
plt.text(10, 0.5, "Algo informativo")
```

Out[36]:
```
Text(10,0.5,'Algo informativo')
```



Note que únicamente colocamos el texto *Algo informativo* dentro del gráfico, de manera más específica en las coordenadas (10,0.5).

Al texto colocado podemos darle formato y ajustarlo a nuestros requerimientos, para ello a la función `text` se le pueden incluir los *keyword arguments* descritos en https://matplotlib.org/users/text_props.html (https://matplotlib.org/users/text_props.html). Por ejemplo:

In [41]:
```
x = np.linspace(0, 30, 200)
y = np.exp(-0.1*x)*np.cos(x)
plt.plot(x, y, "m")
plt.xlabel("Tiempo (s)")
plt.ylabel("Posición (mm)")
plt.title("Gráfica de posición")
plt.text(10, 0.5, "Algo informativo", fontsize=16, color="r",
         name="Times New Roman")
```

Out[41]:
```
Text(10,0.5,'Algo informativo')
```

Observe que lo único que se cambió fueron algunas propiedades del texto, tales como el tamaño de la fuente con `fontsize`, el color de fuente con `color` y el tipo de fuente con `name`, con este último se debe tener cuidado, dado que el nombre de la fuente indicada debe estar instalada en la PC que se ejecuta.

Gráficas en coordenadas polares

Las coordenadas polares o sistema de coordenadas polares son un sistema de coordenadas bidimensional en el que cada punto del plano se determina por una distancia y un ángulo [https://es.wikipedia.org/wiki/Coordenadas_polares] (https://es.wikipedia.org/wiki/Coordenadas_polares]). Habitualmente las funciones en coordenadas polares tienen la forma $r = f(\theta)$.

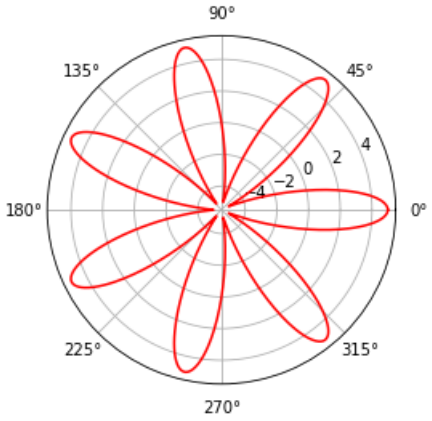
En Matplotlib se dispone de la función `polar`, la cual traza una gráfica en coordenadas polares, dados como argumentos tanto la variable independiente θ como la función r. Enseguida vamos a ver cómo graficar la tan conocida rosa polar, cuya ecuación general está dada por:

$$r = a\cos(k\theta + \phi_0)$$

Implementando esto en Python, se tiene:

In [42]:
```
theta = np.linspace(0, 2*np.pi, 1000)
a,k,phi0 = 5,7,0
r = a*np.cos(k*theta + phi0)
plt.polar(theta, r, "r")
```

Out[42]:
```
[<matplotlib.lines.Line2D at 0x1e13e86aa58>]
```



Observe que la función `polar` funciona de manera bastante similar a `plot` , de hecho se le pueden pasar los mismos *keyword arguments* para personalizar el gráfico resultante.

Gráficas de barras

Gráficas de curvas paramétricas en el espacio

Una función vectorial de la forma:

$$\vec{r}(t) = \begin{bmatrix} x(t) \\ y(t) \\ z(t) \end{bmatrix}$$

Se dice que es una función parámetrica, siendo t en este caso el parámetro correspondiente. Una función vectorial de este tipo tiene una curva en el espacio asociada como representación gráfica. Es muy común trabajar con este tipo de expresiones en el análisis cinemático de partículas.

Supongamos que queremos graficar la función vectorial:

$$\vec{r}(t) = \begin{bmatrix} \cos(t) \\ \sin(t) \\ t \end{bmatrix}$$

En el intervalo $0 \leq t \leq 4\pi$. Para ello en Python haríamos lo siguiente:

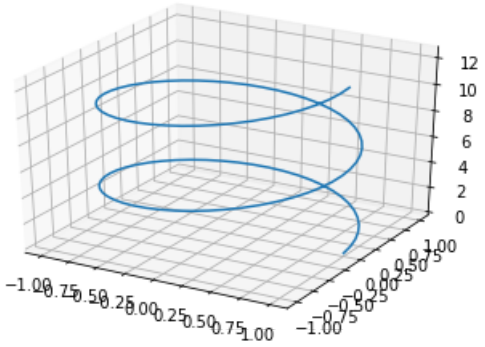
In [44]:
```
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure()
ax = fig.add_subplot(111, projection="3d")

t = np.linspace(0, 4*np.pi, 100)
x = np.cos(t)
y = np.sin(t)
z = t

ax.plot(x, y, z)
```

Out[44]:
```
[<mpl_toolkits.mplot3d.art3d.Line3D at 0x1e13e8f2f28>]
```



Ahora explicamos lo referente al código anterior. Observe que en la primera línea importamos la clase `Axes3D` del módulo `mpl_toolkits.maplot3d`, esto nos sirve para poder trabajar con gráficas tridimensionales. Luego, definimos un objeto de la clase `Figure` y lo asignamos a la variable `fig`, al objeto `fig` le añadimos un *Axes* mediante el método `add_subplot`, indicando que en dicho *axes* se utilizarán las proyecciones espaciales mediante el *keyword argument* `projection`. Las siguientes cuatro líneas definen las ecuaciones parámetricas. Y finalmente, con el método `plot` del objeto `ax` trazamos la gráfica de la curva tridimensional, note que en este caso el método `plot`, recibe al menos tres argumentos: las coordenadas en x, y, z.

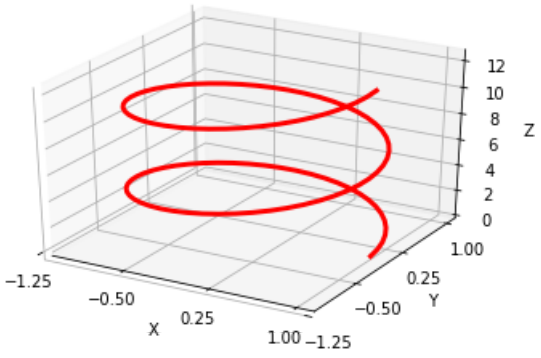
Al igual que en las otros tipos de gráficos, podemos también manipular las características. Vea por ejemplo el siguiente código:

In [45]:
```
fig = plt.figure()
ax = fig.add_subplot(111, projection="3d")

t = np.linspace(0, 4*np.pi, 100)
x = np.cos(t)
y = np.sin(t)
z = t

ax.plot(x, y, z, color="r", linewidth=3)
xticks = ax.get_xticks()
yticks = ax.get_yticks()
ax.set_xticks(xticks[::3])
ax.set_yticks(yticks[::3])
ax.set_xlabel("X")
ax.set_ylabel("Y")
ax.set_zlabel("Z")
```

Out[45]: Text(0.5,0,'Z')



Gráficas de superficies

In [ ]:
```

```