[*Escuela de código*]

Apuntes de clase:

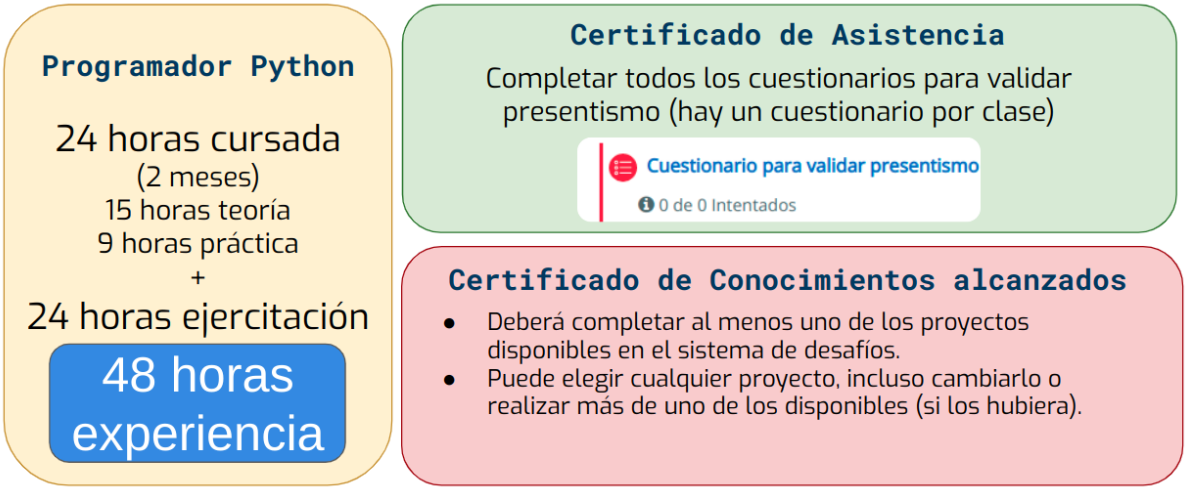
Matplotlib

Tabla de contenido

Curso Programador Python 1
Desafíos 2
Proyecto de Programador Python 4
Sistema de gestión de paquetes (PIP) 4
PIP en Linux-Ubuntu 4
PIP en Windows 5
Instalar paquetes 5
Actualizar o borrar paquetes 5
Matplotlib 6
Instalación y uso 7
Uso de la librería 7
Composición del gráfico 7
Crear nuestro primera figura 8
Múltiples líneas 10
Múltiples gráficos 12
Marcas, colores y tipos de línea 14
Marker 14
Colores y tipo de línea 16
Conclusión 18
Grilla 18
Tipos de gráficos 18
Line Plot 20
Aplicaciones 21
Una sola variable 22
Gridspec 24
Scatter Plot 25
Aplicaciones 28
Bar plot 28
Aplicaciones 30
Múltiples gráficos de barras 31
Bar Plot apilados (stack) 33
Bar plot agrupados (grouped) 35
Pie plot 37
Seaborn 38
Descargar un repositorio de GitHub 38
Hasta la próxima! 40
Links de interés 41

Curso Programador Python

El curso de Programador Python está compuesto por 24 horas de cursada más el mínimo tiempo recomendado por fuera para profundizar en la materia (24 horas de ejercitación). La cursada a su vez se divide en horas enfocadas a contenidos teóricos y de ejercitación. El objetivo de cada clase es que se lleven un concepto nuevo que puedan aplicarlo y despejar sus dudas en el mismo día, para que luego puedan continuar ampliando y fijando la idea en casa con ejercitación. Se espera que dediquen un rato de cada semana a realizar ejercitación que los ayudará a comprender mejor la teoría y el día de mañana serán valiosas horas de experiencia que los ayudaran en su vida personal.



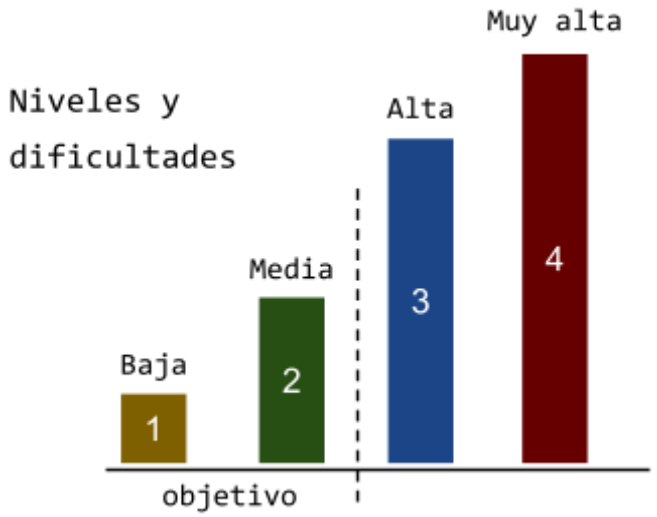
Cada cursada finaliza con un trabajo práctico integrador que deben presentar para poder obtener la certificación del curso, caso contrario recibirán un certificado de asistencia si han completado todos los cuestionarios que se encuentran en el campus.

Cuanto mayor tiempo le dediquen a la ejercitación planteada para hacer durante la semana, más fácil será para ustedes el desarrollo del proyecto final.

Desafíos

Los desafíos se dividen en distintas dificultades:

- **Nivel 1 (dificultad baja)**: Desafíos propuestos para afirmar los conceptos básicos de la clase, requieren muy poca dedicación.
- **Nivel 2 (dificultad media)**: Desafíos que podrá realizar el alumno sin experiencia previa, dedicando un tiempo considerable y realizando las debidas dudas o consultas en los foros de comunidad del curso (¡consulte sus dudas!).
- **Nivel 3 (dificultad alta)**: Desafíos que superan el objetivo mínimo del curso, requieren mucha más investigación y exploración personal. Son desafíos planteados para alumnos con experiencia previa en programación o para aquellos alumnos con mucho tiempo disponible para profundizar lo visto en clase.
- **Nivel 4 (dificultad muy alta)**: Son desafíos similares a los que pueden encontrarse en una entrevista de trabajo, recomendamos realizarlos cuando ya tenga una sólida base de los temas abordados en el curso. Los puede realizar más adelante como calentamiento antes de postularse a una oferta de trabajo.



La realización de los desafíos es un camino que transita el alumno, la prioridad es que pueda realizar los ejercicios de nivel dos ya que son aquellos que garantizan haber entendido el concepto de lo visto en clase y el objetivo mínimo del curso esperado.

Los desafíos de mayor dificultad se deben realizar cuando el alumno no tiene ninguna duda sobre lo realizado en los otros desafíos. Recomendamos siempre priorizar la realización de los desafíos que comprenden el objetivo del curso y haber comprendido la unidad antes de continuar con los desafíos de la siguiente unidad.

La realización del proyecto es la recta final del camino transitado por el alumno, recomendamos haber realizado y comprendido a la perfección los desafíos que comprenden al objetivo mínimo del curso antes de comenzar a desarrollar el proyecto, ya que integra todo lo desarrollado durante la cursada.

A medida que vaya realizando los desafíos obtendrá puntos de experiencia, como recompensa por su esfuerzo.

- Su esfuerzo será recompensado con puntos de experiencia
- Podrá canjear sus puntos por descuentos y beneficios
- Sumando exp puede postularse para trabajar dentro de inove desde el lado académico y de desarrollo
- El sistema lo recompensará si mantiene su objetivo semanal (racha)

Recuerde que hay tiempos de entrega estimados para los desafíos de cada unidad. SI respeta esos tiempos y mantiene su "racha" alcanzando el objetivo mínimo cada semana, el sistema lo recompensará con más puntos aún (puntos de bonus)

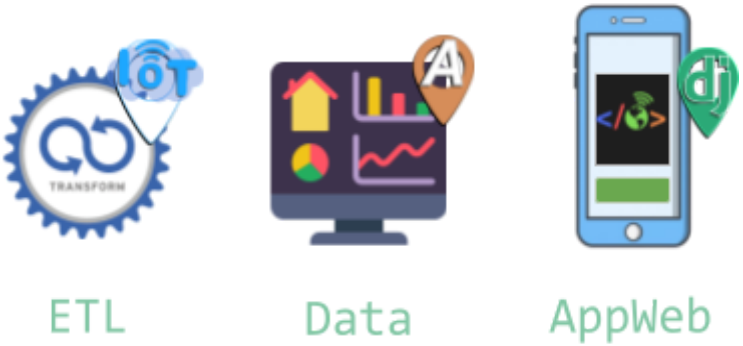
En caso que desee volver a consultar cuales son los tiempo de entrega estipulados para los desafíos, les recomendamos volver a leer los términos y condiciones (sección "Entrega de actividades"):
http://inovecode.com/terminos-y-condiciones/

Proyecto de Programador Python

El proyecto realizado en esta etapa se enfoca en tres pilares, los procesos de transformación de datos (ETL), el proceso de análisis de datos (data) y las aplicaciones web. Al final del curso se le dará a elegir tres opciones de proyecto, cada una relacionada con los pilares mencionados que están fuertemente vinculados con cada especialización de la carrera.

- Transformación de datos → Python IoT
- Análisis de datos → Python Anayltics
- Aplicaciones web → Python Django

Deberán elegir una de las opciones propuestas para alcanzar su certificado de conocimientos alcanzados del curso de Programador Python



Terminado exitosamente un proyecto adquiere el certificado, luego usted si lo desea puede realizar otro proyecto para continuar poniendo en práctica los conceptos del curso.

Sistema de gestión de paquetes (PIP)

"PIP" es una herramienta incluida en Python para gestionar la instalación de paquetes python de terceros por fuera de la librería estándar.
Todos los paquetes o módulos oficiales utilizan este sistema para instalar sus paquetes en nuestro entorno de Python:

https://pypi.org/

Es una herramienta muy práctica y poderosa que con una sola línea en nuestra terminal podremos instalar cualquier paquete de Python sin importar nuestra plataforma o sistema operativo. Resuelve todas las dependencias y las instala, a partir de ahí podremos instanciar ese paquete en nuestro programa con la sentencia **import** colocando el nombre del paquete instalado.

PIP en Linux-Ubuntu

Es probable que para aquellos usuarios de Linux-Ubuntu no tengan instalado por default la versión de **pip3** relativa a **Python3**. Para esos casos instalar PIP mediante el siguiente comando:

```
sudo apt-get install python3-pip
```

PIP en Windows

En Windows **pip3** ya viene instalado con el intérprete de **Python3**, Lo que puede suceder es que si tienen más de un python instalado que el sistema no encuentre el paquete **pip3** acorde. Cuando suceden esos en Windows casos ejecutar:

```
python -m ensurepip
```

Instalar paquetes

Antes de comenzar a utilizar PIP por primera vez es buena práctica actualizarlo, ya que es probable que la versión que descargamos con Python este bastante desactualizada, para ello ejecutamos:

```
pip3 install --upgrade pip
```

IMPORTANTE: Es recomendable siempre que se utilice "PIP" utilizarlo como "**pip3**" para estar seguros de estar descargando el paquete compatible para **Python3**

Luego si quiero instalar cualquier paquete ejecuto:

```
pip3 install <nombre_paquete>
```

Por ejemplo para instalar Numpy:

```
pip3 install numpy
```

En caso de tener varios Python3 instalados (Linux), ejecutaremos PIP como:

```
python3.7 -m pip install numpy
```

Instalar versión específica del paquete:

```
pip3 install <nombre_paquete>==<version>
```

Por ejemplo deseo instalar Numpy 1.18.2:

```
pip3 install numpy==1.18.2
```

Si llegamos a tener problemas de privilegios de usuario administrador o root y deseamos instalar el paquete solo en el usuario actual sumamos la opción "**--user**":

```
pip3 install --user <nombre_paquete>
```

Actualizar o borrar paquetes

Para actualizar un paquete ejecutamos:

```
pip3 install -upgrade <nombre_paquete>
```

Forma simplificada:

```
pip3 install -U <nombre_paquete>
```

Para desinstalar o borrar un paquete instalado:

```
pip3 uninstall <nombre_paquete>
```

Matplotlib

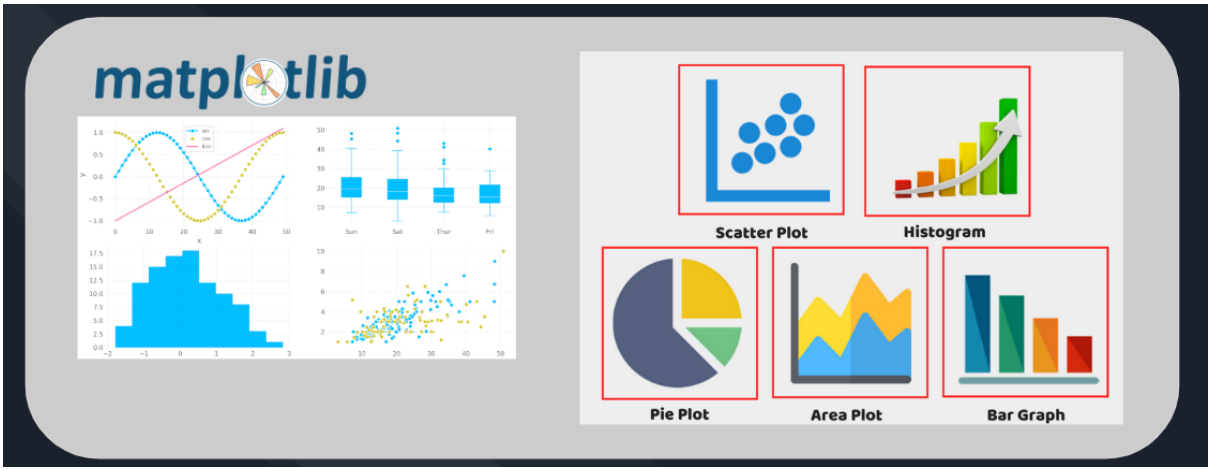
Matplotlib es una librería para la generación de gráficos 2D y 3D open source creada en el 2012. Es la librería más popular en su tipo en Python la cual permite con pocas líneas de código generar gráficos de calidad de producción. Matplotlib nace como necesidad de generar gráficos de estilo similar a "matlab".

De matplotlib derivan otras librerías de terceros que extienden su funcionalidad como **seaborn, HoloViews, ggplot**.

A pesar de que realmente es simple realizar gráficos con Matplotlib la documentación de esta librería es gigante, posee más de 70.000 líneas de código y muchos ejemplos en la nube. Esto genera que veamos, tal vez, 5 formas distintas de hacer el mismo trabajo con matplotlib lo cual al principio puede resultar frustrante porque pareciera que hay que memorizar distintas formas de trabajo según el gráfico que deseemos utilizar o la librería que queramos aprovechar (ej: seaborn). A su vez, la librería crece tan rápido que muchos de los ejemplos que encontrarán de años anteriores son "obsoletos".

El objetivo de este documento es guiarlo en los primeros pasos en la producción de gráficos y darle una orientación de cómo trabajar con matplotlib que sea compatible con:

- Generar uno o múltiples gráficos.
- Modificar el estilo.
- Utilizar otras librerías de terceros que deriven de matplotlib.



Instalación y uso

Para instalar matplotlib en nuestro sistema debemos tener instalado "pip" y ejecutar la siguiente línea:

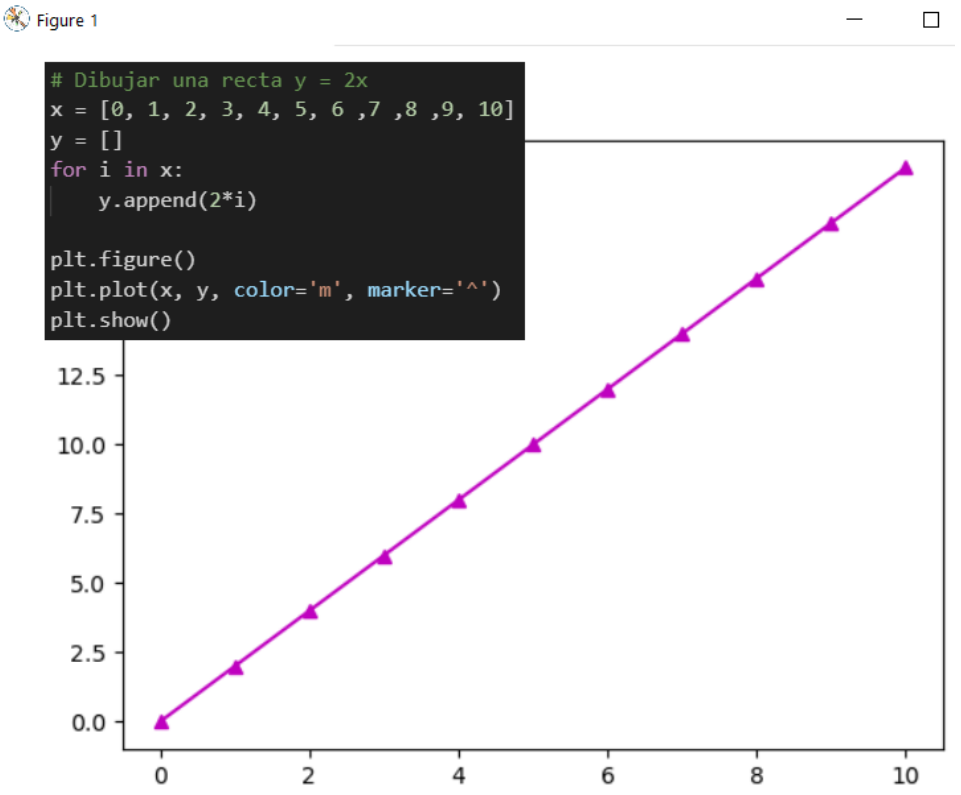
```
pip3 install matplotlib
```

Cuando queramos invocar a "matplotlib" en nuestro programa lo realizaremos de la siguiente forma:

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

En general siempre verán en todos los programas de terceros que de "matplotlib" generalmente se importa "pyplot" con el alias "plt", tal como figura en la documentación oficial.

Con pocas líneas de código podemos tener nuestro primer gráfico:

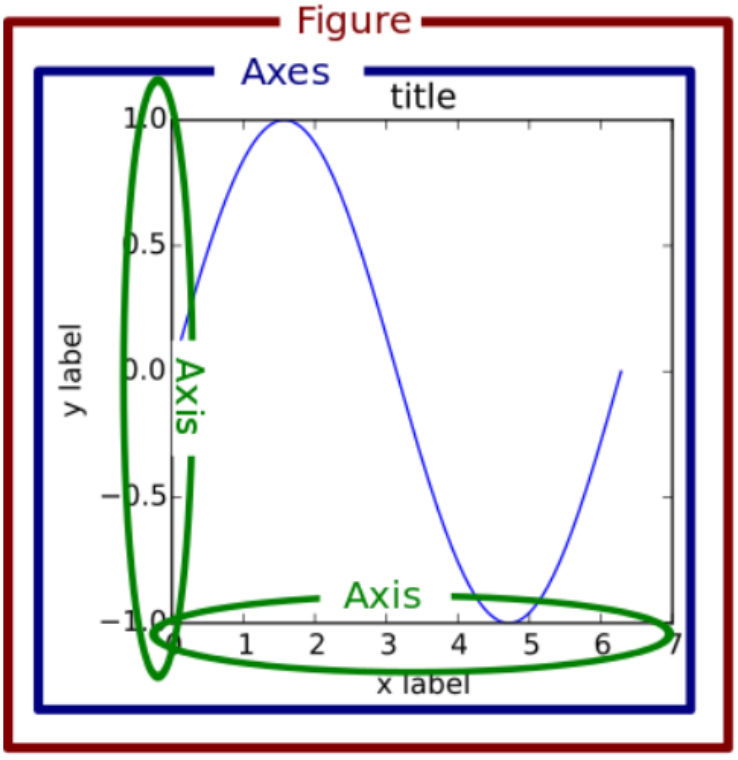


Uso de la librería

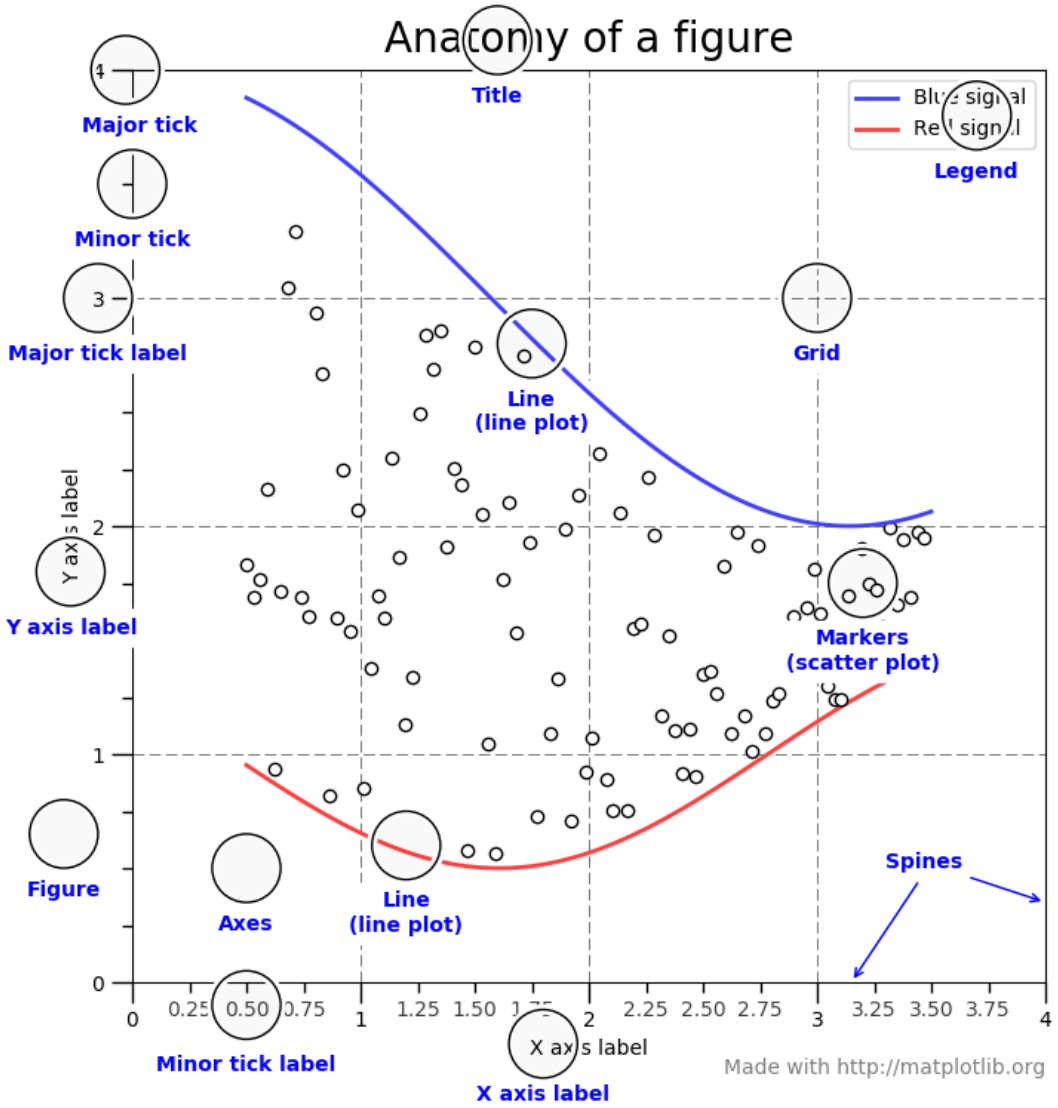
Composición del gráfico

Cuando queremos realizar gráficos en matplotlib debemos construir una "**Figure**", la cual está compuesta de las siguientes partes:

- Figure: Es la figura, el contenedor de todos los gráficos que deseemos mostrar juntos.
- Axes: Es el gráfico, engloba todo lo relativo a este (dibujo, ejes, etc).
- Axis: Son los ejes de un gráfico en cuestión.
- Title: Es el título del gráfico (axes) en cuestión.
- Label: Es el título de los ejes (axis) del gráfico (axes) en cuestión.



Luego cada gráfico (axes) que dibujemos dentro de la figura tiene los siguientes elementos:



- Legend: Es la referencia de que significa cada línea en el gráfico.
- ticks: Son las divisiones (major tick) y subdivisiones (minor tick) del gráfico.

Crear nuestro primera figura

Lo primero que debemos generar es nuestra figura "fig", luego sumarle a nuestras figuras todos los gráficos (axes) que deseamos mostrar:

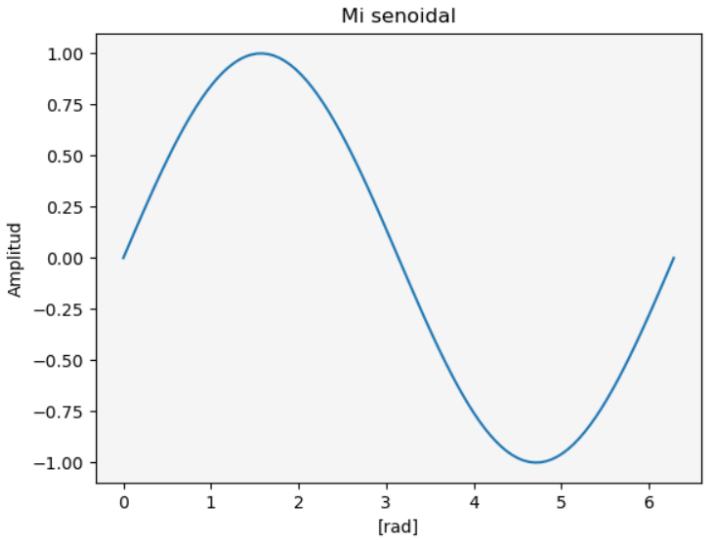
```
# Dibujar una senoidal y = sin(x)
x = np.linspace(start=0, stop=2*np.pi, num=100)
y = np.sin(x)

fig = plt.figure()      # Definir tamaño figura
ax = fig.add_subplot()  # Definir cuantos gráficos tendrá
```

En este caso solo mostraremos un único gráfico, por lo que solo llamaremos una vez a "add_subplot" sin especificar ningún parámetro.

Luego utilizaremos el "ax" que generamos para "plotear" el dibujo de la senoidal, especificando el título (set_title) y los títulos de los ejes (set_xlabel / set_ylabel) (axis):

```
ax.plot(x,y)              # Graficar con plot en mi gráfico "ax"
ax.set_facecolor('whitesmoke')
ax.set_title("Mi senoidal")
ax.set_ylabel("Amplitud")
ax.set_xlabel("[rad]")
plt.show()                # Mostrar el gráfico
```

El resultado final será el siguiente:



NOTA: A modo ilustrativo y para evitar quemarnos la vista, se ha utilizado el método "set_facecolor" para cambiar el color de fondo de blanco a un gris clarito.

Múltiples líneas

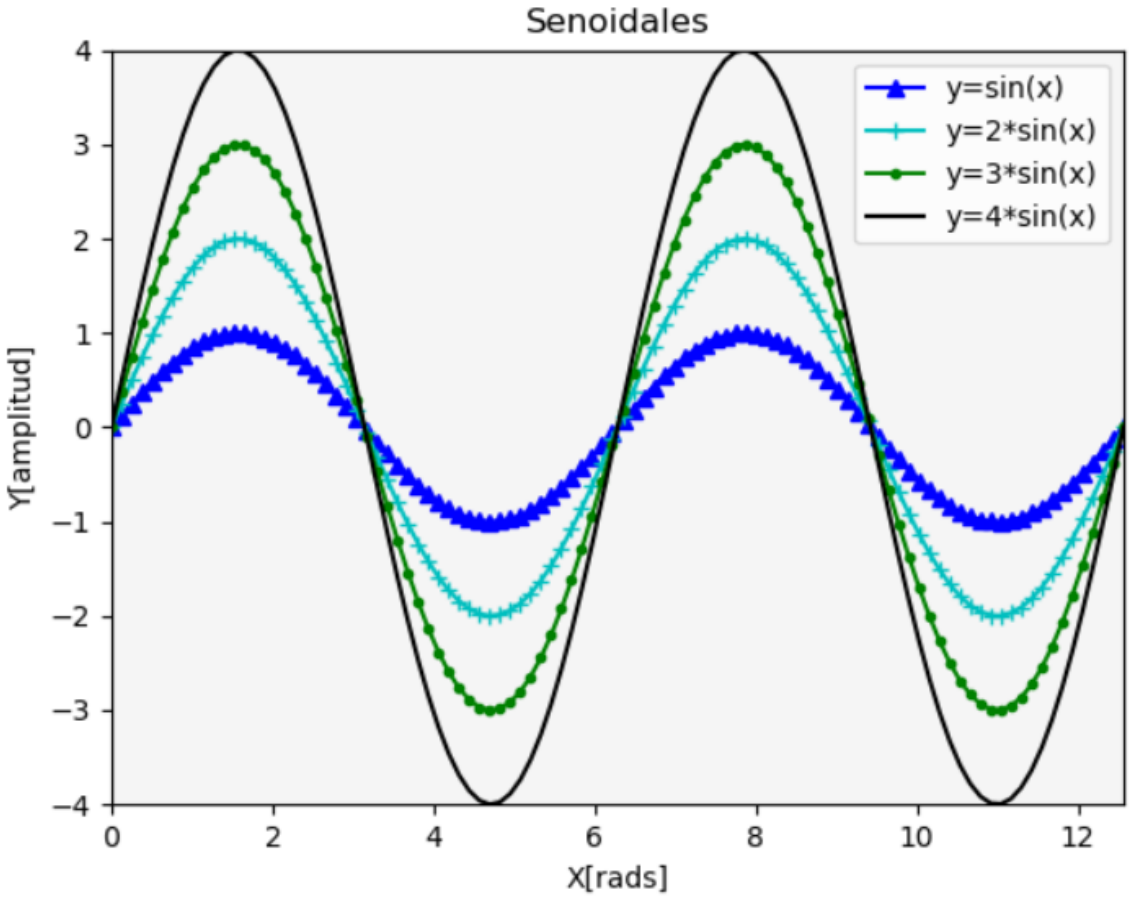
Con múltiples líneas nos referimos a realizar más de un "dibujo" en un mismo gráfico, para ello simplemente llamar al método de ploteo tantas veces como líneas querramos graficar.

Además, cuando hay más de una línea en un gráfico es buena práctica agregar la "leyenda" o "label" a cada línea para que luego el usuario final pueda identificar de qué se trata cada gráfico:

```
# Dibujar múltiples líneas en un mismo gráfico
x = np.linspace(start=0, stop=4*np.pi, num=100)

fig = plt.figure()
ax = fig.add_subplot()

ax.plot(x, np.sin(x), color='b', marker='^', label='y=sin(x)')
ax.plot(x, 2*np.sin(x), color='c', marker='+', label='y=2*sin(x)')
ax.plot(x, 3*np.sin(x), color='g', marker='.', label='y=3*sin(x)')
ax.plot(x, 4*np.sin(x), color='k', label='y=4*sin(x)')
ax.set_facecolor('whitesmoke')
ax.set_title("Senoidales")
ax.set_ylabel("Y[amplitud]")
ax.set_xlabel("X[rads]")
ax.set_xlim([0, 4*np.pi])  # Limito el eje "Y" entre 0 y 4*pi
ax.set_ylim([-4, 4])       #Limito el eje "X" entre -4 y 4
ax.legend()
plt.show(block=False)
```

- Como podemos ver hemos llamado cuatro veces a "ax.plot" por lo que esperamos ver 4 líneas dibujadas.
- En cada "plot" hemos especificado el "label" que luego será identificado por su "color" y "merket" (marcador) en la legenda.
- Hemos limitado lo que se mostrará tanto en el eje "x" como el eje "y" con los métodos "set_xlim" y "set_ylim".
- Activamos el uso de la leyenda con el método "ax.legend()".
- En este ejemplo se puede que a la hora de mostrar la figura, al llamar al método "show" estamos indicando que no queremos que se detenga el programa (block=False). Muy útil cuando queremos generar muchas figuras a la vez.

El gráfico resultado del código mostrado será el siguiente:



Múltiples gráficos

Supongamos que queremos mostrar en una misma figura el ejemplo anterior pero esta vez subdividido en gráficos distintos en vez de mostrar todo en un mismo "axes". Para ello debemos agregar subplot especificando en cada caso cuando deseamos agregar y el "número" del axes que estamos generando.
En este ejemplo deseamos mostrar 4 gráficos ordenados en 2 filas y 2 columnas (2x2), por lo que la definición de mis "axes" será:

```
# Dibujar 4 gráficos en una misma figura
fig = plt.figure()
# Ejemplo de uso --> ax = fig.add_subplot(nrows, ncols, index)
ax1 = fig.add_subplot(2, 2, 1)
ax2 = fig.add_subplot(2, 2, 2)
ax3 = fig.add_subplot(2, 2, 3)
ax4 = fig.add_subplot(2, 2, 4)
```

Por cada gráfico que queramos dibujar utilizaremos respectivamente "ax1", "ax2", "ax3" y "ax4". Veamos el ejemplo de cómo sería el gráfico para ax1:

```
ax1.plot(x, np.sin(x), color='b', marker='^', label='y=sin(x)')
ax1.set_facecolor('whitesmoke')
ax1.set_title("Senoidal1")
ax1.set_ylabel("Y[amplitud]")
ax1.set_xlabel("X[rads]")
ax1.set_xlim([0, 4*np.pi])
ax1.set_ylim([-4, 4])
ax1.legend()
```

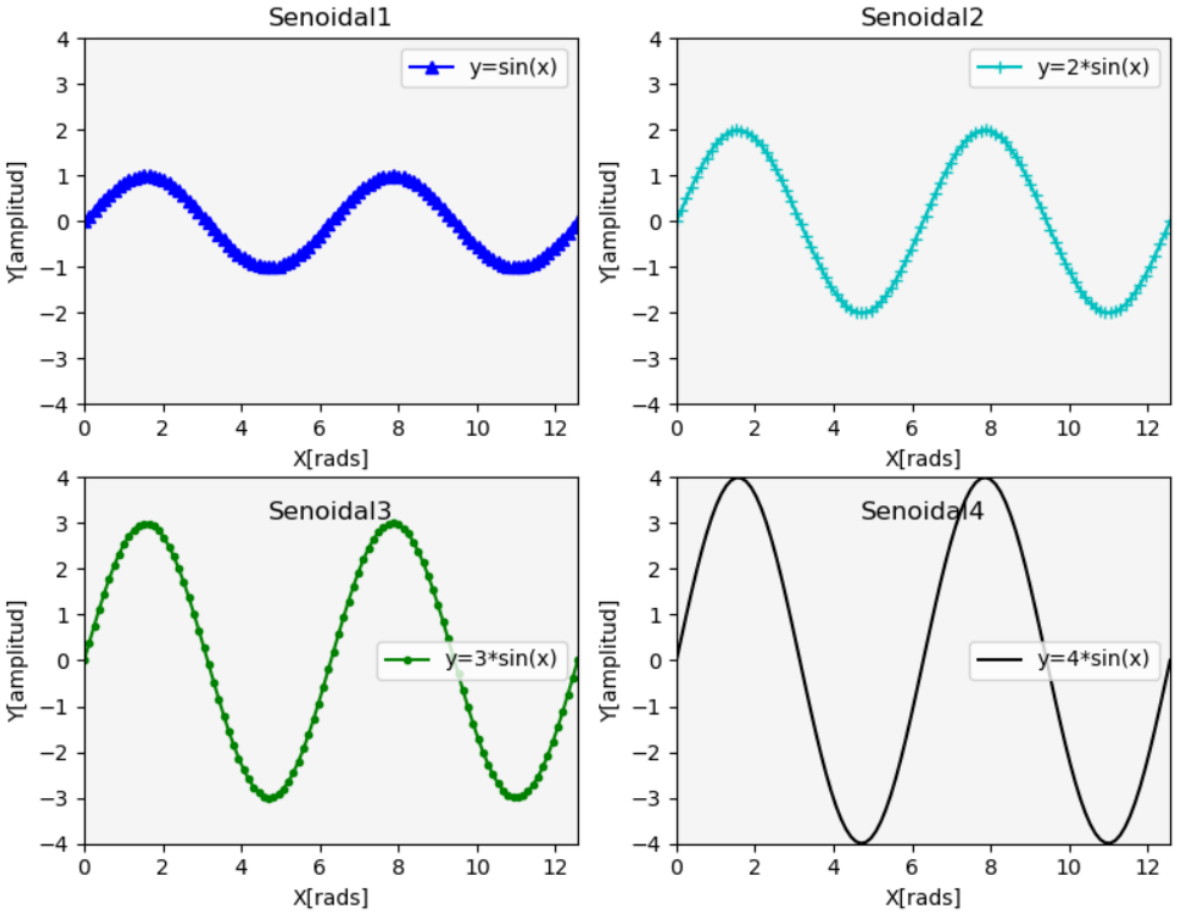
Como pueden ver es exactamente el mismo mecanismo que el utilizado en el caso anterior pero con un solo plot.
Luego el caso para ax2:

```
ax2.plot(x, 2*np.sin(x), color='c', marker='+', label='y=2*sin(x)')
ax2.set_facecolor('whitesmoke')
ax2.set_title("Senoidal2")
ax2.set_ylabel("Y[amplitud]")
ax2.set_xlabel("X[rads]")
ax2.set_xlim([0, 4*np.pi])
ax2.set_ylim([-4, 4])
ax2.legend()
```

Luego de definir todos los "axes" debemos llamar a "plt.show()" para que aparezca la figura con los gráficos deseamos:

```
# Graficar la figura con los 4 axes
plt.show()
```

Finalmente obtendremos como resultado:



Marcas, colores y tipos de línea

Marker

Las "marcas" o "marcadores" son los símbolos que podemos utilizar para identificar donde hay una medición o muestra "real", ya que matplotlib por defecto "interpola" o une las muestras para formar un gráfico continuo. A qué nos referimos:

Supongamos que estoy midiendo la velocidad de un auto en diferentes intervalos de tiempo en segundos, supongamos que realicé 4 mediciones a 0seg, 1seg, 2seg y 3seg distintas velocidades en km/h:

```
t = [0, 1, 2, 3]
vel = [0, 10, 40, 40]
```

Hasta ahora veníamos utilizando Numpy para generar los datos de los gráficos, pero matplotlib acepta cualquier secuencia de datos como ser por ejemplo una lista.

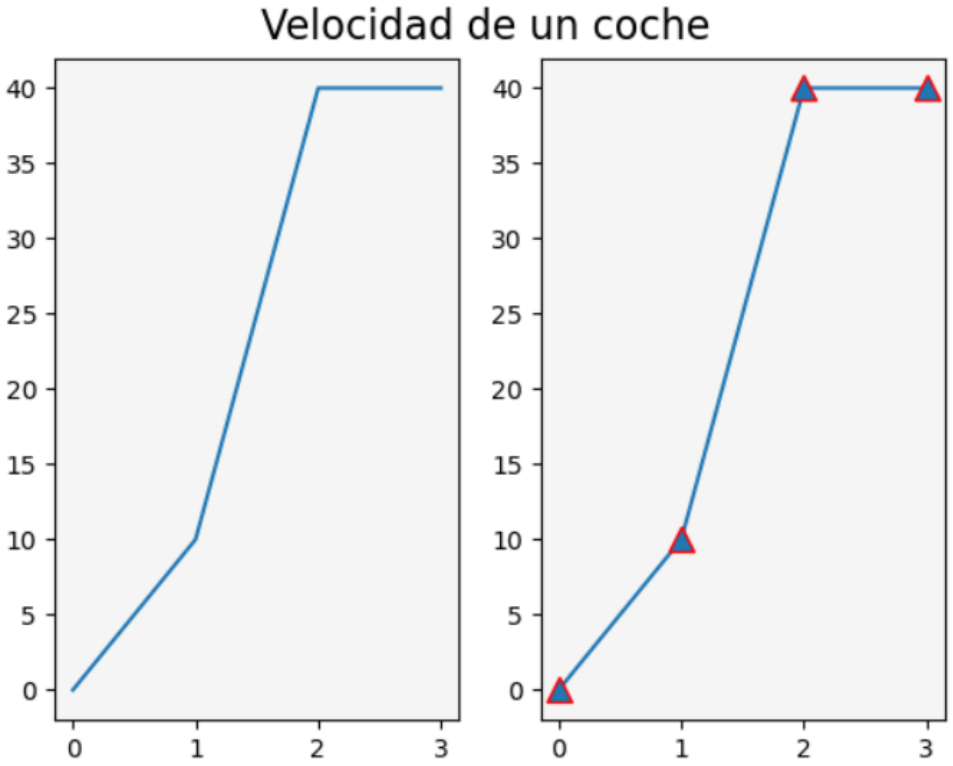
Veamos ahora el ejemplo de construir un gráfico con solo 4 mediciones con y sin marcadores:

```
# Realizaremos dos gráficos, uno sin marker y otro con marker
# Dibujar 2 gráficos en una misma figura
fig = plt.figure()
fig.suptitle('Velocidad de un coche', fontsize=16)
ax1 = fig.add_subplot(1, 2, 1)
ax2 = fig.add_subplot(1, 2, 2)

ax1.plot(t, vel)
ax2.plot(t, vel, marker='^', mec='r', ms=10)
plt.show()
```

Gráfico ax1
Como podemos ver en este gráfico no hemos especificado ni el color de la línea, ni el marcador. Por defecto un gráfico se construye de línea contigua color "azul" sin marcadores.

Gráfico ax2
En este caso hemos especificado el **maker**, si culor con **mer** (abreviatura de markeredgecolor) y su tamaño con **ms** (abreviatura de markersize).

Veamos como se ve el gráfico:



Como podemos ver el hecho de que no estén "marcadas" las muestras es más difícil de comprender en el gráfico que está proyectado o interpolado y que es real. En este gráfico es simple ver los "puntos de quiebre", pero en todo caso tenemos la herramienta de los "marker" cuando queramos tener la certeza de que puntos son los reales en el gráfico.

Colores y tipo de línea

Ya vimos distintas formas de personalizar nuestro gráfico, entraremos más en detalle respecto a los colores y los tipo de línea que son los personalizaciones más comunes:

```
# Realizaremos dos gráficos, con distinto tipo de línea y color
# Dibujar 2 gráficos en una misma figura
fig = plt.figure()
fig.suptitle('Velocidad de un coche', fontsize=16)
ax1 = fig.add_subplot(1, 2, 1)
ax2 = fig.add_subplot(1, 2, 2)

ax1.plot(t, vel, color='r')
ax2.plot(t, vel, c=(0, 0.5, 0.5), ls='--', lw='2')
plt.show()
```

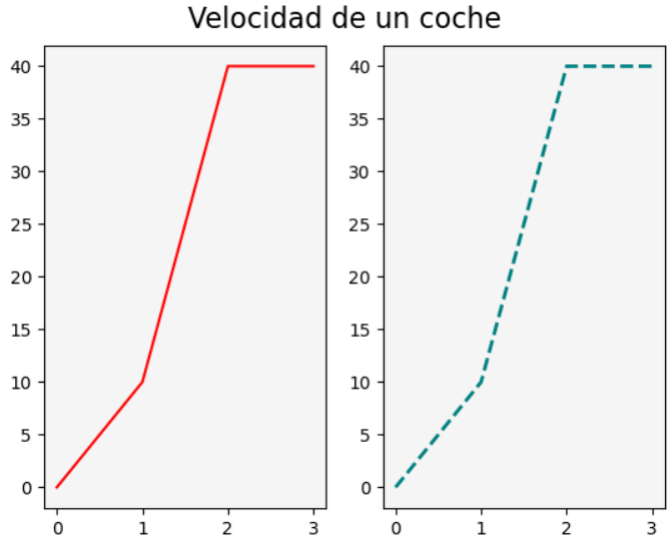
Gráfico ax1

En este gráfico simplemente cambiamos el color de la línea (que por defecto es azul), elegimos el **color** rojo (**r**). Hay distintos colores "standard" que pueden llamarse con una letra, otros se llaman por nombre como "darkred" o simplemente puede especificarse el tipo de color en RGB (R, G, B) → valores a 0 a 1.

Gráfico ax2

En este gráfico nuevamente modificamos el tipo de color llamando al parámetro **color** pero en su versión abreviada **c**, especificando el tipo de color por RGB (red, green, blue). Además, se especifica el tipo de estilo de línea con **ls** (abreviado de linestyle) y el tamaño de línea **lw** (abreviado de line width).

El resultado es el siguiente:



Conclusión

En los links de interés verán bastante información respecto a los distintos tipos de marcadores, colores y tipos de líneas, les dejamos algunos ejemplos de la información que se pueden encontrar:

marker	symbol	description
"."	●	point
","	·	pixel
"o"	●	circle
"v"	▼	triangle_down
"^"	▲	triangle_up
"<"	◀	triangle_left
">"	▶	triangle_right
"1"	Y	tri_down

Line Styles

character	description
'-'	solid line style
'--'	dashed line style
'-.'	dash-dot line style
':'	dotted line style

Para más información ver los “links de interés”

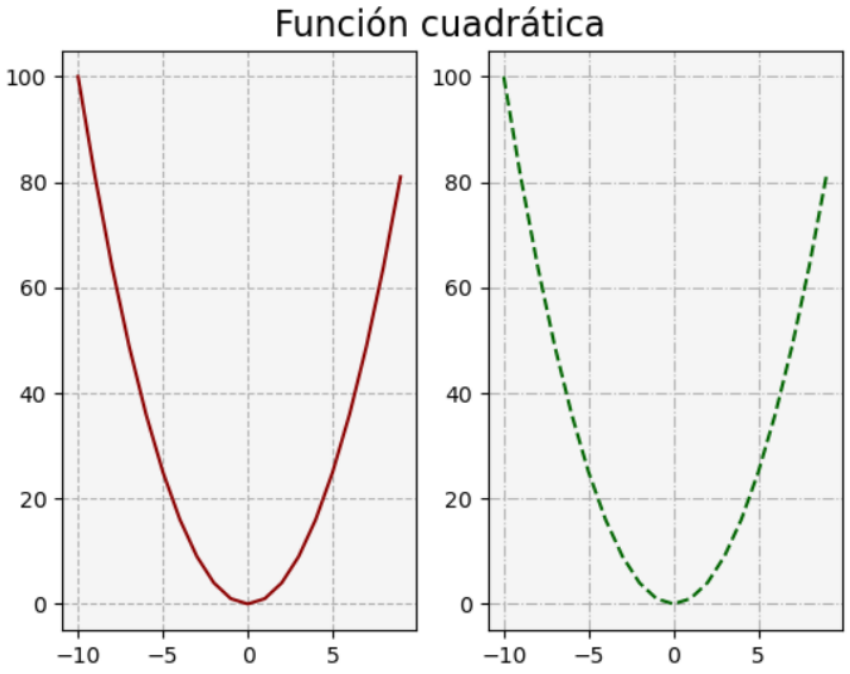
Grilla

La personalización que puede realizarse sobre el gráfico es la "grilla" que podemos utilizar para poder analizar el gráfico.

Las grillas pueden ser punteadas, rayadas, sólidas, etc. Tal como los colores y los marcadores hay un montón de alternativas, les dejamos al final de este documento el link a los distintos tipos de grillas.

EJemplo:

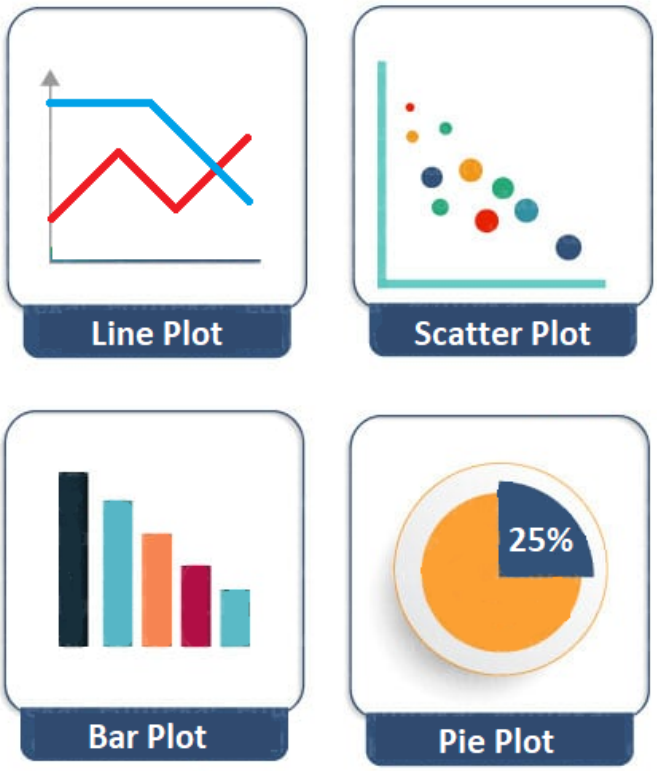
```
# Veremos los distintos tipos de grids
x = range(-10, 10, 1)
y = []
for i in x:
    y.append(i**2)

# Realizaremos dos gráficos, con distinto tipo de línea y color
# Dibujar 2 gráficos en una misma figura
fig = plt.figure()
fig.suptitle('Función cuadrática', fontsize=16)
ax1 = fig.add_subplot(1, 2, 1)
ax2 = fig.add_subplot(1, 2, 2)

ax1.plot(x, y, color='darkred')
ax1.set_facecolor('whitesmoke')
ax1.grid(ls='dashed')
ax2.plot(x, y, c='darkgreen', ls='--')
ax2.set_facecolor('whitesmoke')
ax2.grid(ls='dashdot')
plt.show()
```



Tipos de gráficos

Hay muchos gráficos distintos, algunos relacionados para realizar métricas, ver ensayos o mediciones, otros más enfocados a finanzas, a probabilidad y estadística.

En esta oportunidad veremos 4 de los gráficos más conocidos:



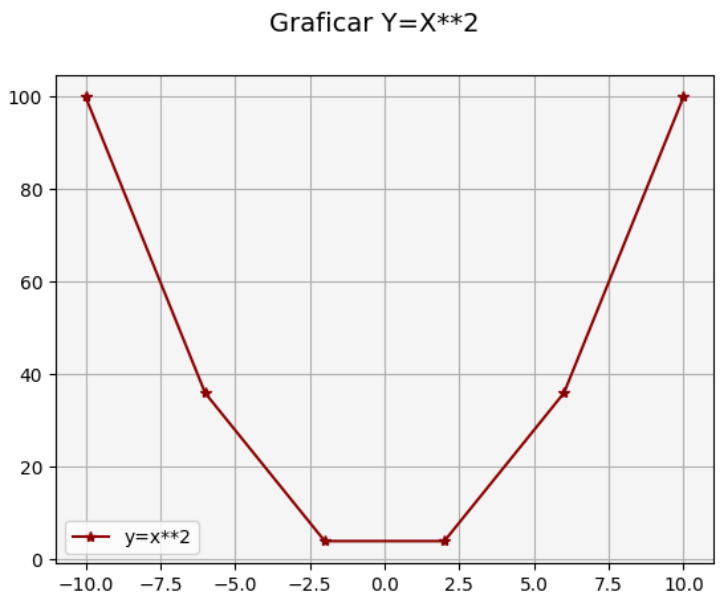
- Line plot: Gráficos de líneas, son los tipos de gráficos en donde los puntos se unen por rectas, tal como todos los ejemplos vistos hasta el momento.
- Scatter plot: Es un gráfico de dispersión, en donde los puntos no se unen como en el gráfico de línea.
- Bar plot: Gráfico de barras, muy utilizado para analizar la evolución de los datos.
- Pie Plot: Gráfico de torta, muy útil para el impacto de ciertos elementos en el total de las muestras.

Line Plot

El gráfico de línea es uno de los más usados y es el gráfico genérico que utilizamos hasta ahora en todos los ejemplos. Este gráfico une con rectas los puntos (datos) que se pasen al gráfico, cuantos más puntos (más muestras) mejor forma tendrá el gráfico.
Ejemplo:

```
# Generamos la función y=X^2 (x al cuadrado)
x = range(-10, 11, 4)
y = []
for i in x:
    y.append(i**2)

fig = plt.figure()
fig.suptitle('Graficar Y=X**2', fontsize=14)
ax = fig.add_subplot()

ax.plot(x, y, c='darkred', marker='*', label='y=x**2')
ax.legend()
ax.grid()
ax.set_facecolor('whitesmoke')
plt.show()
```

Resultado:

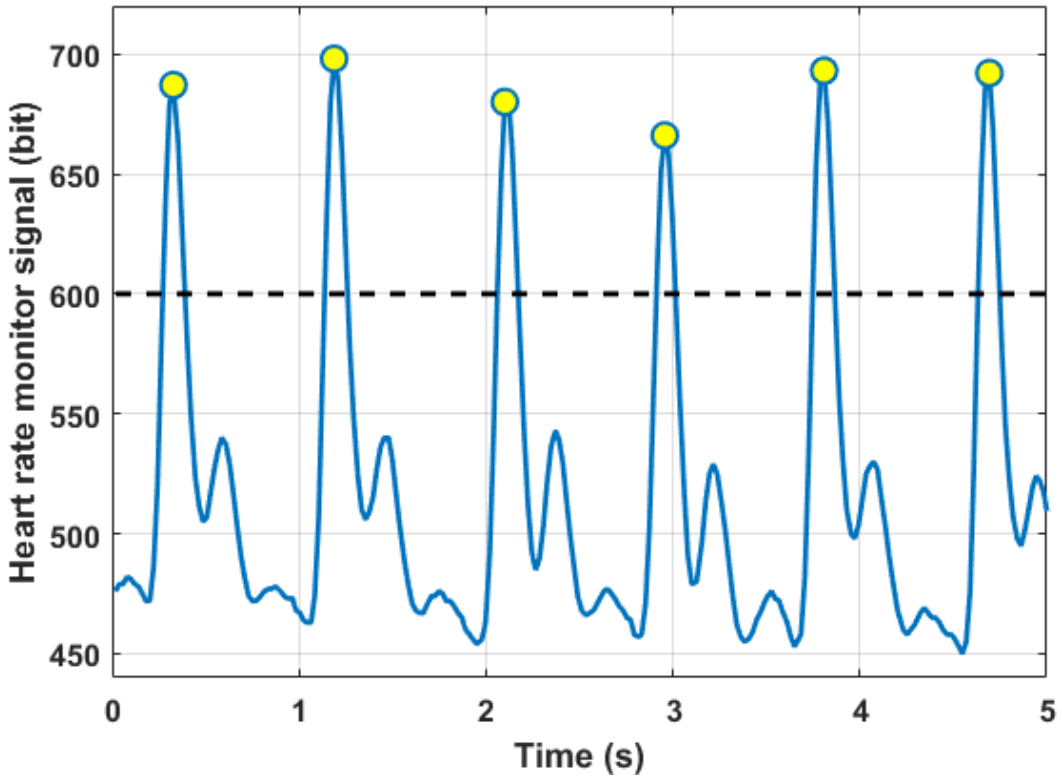


Aplicaciones

Se suele utilizar para visualizar la evolución de un indicador en el tiempo, como por ejemplo:

- Gráficos de acciones, bonos, inversiones.
- Datos provenientes de sensores o dispositivos (temperatura, humedad, etc).
- Datos provenientes de equipos, como por ejemplo signos vitales (ritmo cardíaco, presión, oxígeno).
- Datos como compras, gastos, oferta, demanda.

Ejemplo de las pulsaciones del ritmo cardíaco de una persona:



Una sola variable

Una particularidad que tiene el "Line Plot" es que podemos graficar la evolución o forma de una sola variable. Veamos un ejemplo:

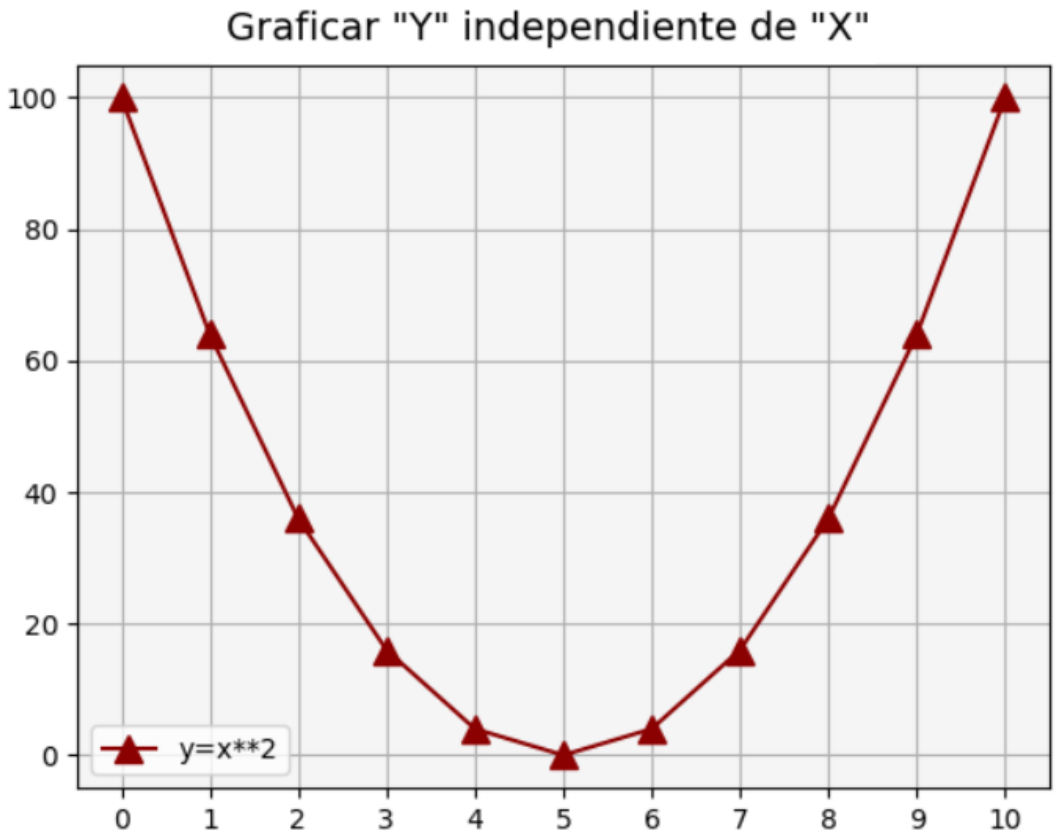
Supongamos que yo tengo la variable "Y" obtenida como resultado de elevar al cuadrado una serie de valores "X":

```
# Demostración de uso de line plot con una sola variable
# Generaremos la función y=X^2 (x al cuadradro)
# pero solo graficaremos los valores de "Y" depedientes de "X"
x = range(-10, 11, 2)
y = []
for i in x:
    y.append(i**2)
```

Supongamos también que por diversas razones no contamos con la variable "X", sólo con los valores de "Y", pero queremos graficar "Y" para tener una idea de la forma de la evolución de esa variable. Con el "**plot**" no hay problema y simplemente graficamos solos "Y":

```
fig = plt.figure()
fig.suptitle('Graficar "Y" independiente de "X"', fontsize=14)
ax = fig.add_subplot()

ax.plot(y, c='darkred', marker='^', ms=10, label='y=x**2')
ax.legend()
ax.grid()
ax.set_facecolor('whitesmoke')
plt.plot(block=False)
```

El resultado obtenido será:



La conclusión es que se pudo graficar solo los valores de "Y", la forma del gráfico es correcto pero no podemos guiarnos por lo que dice el eje de abajo (el eje "X" o de abscisas) ya que no tendrá una relación directa con el eje lateral (eje "Y" o de ordenadas).

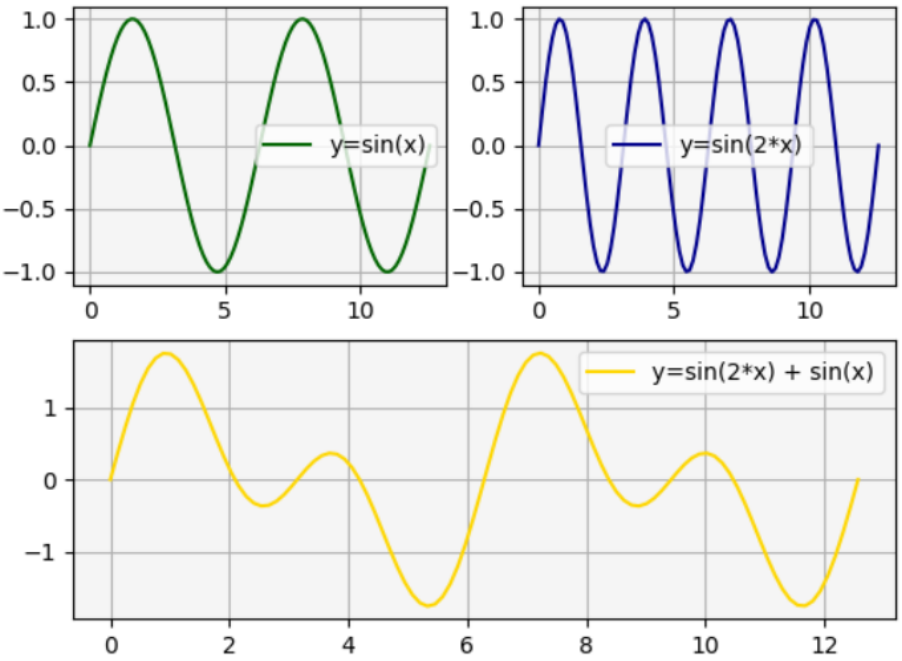
Por ejemplo, podemos ver a simple vista que "5" elevado al cuadrado no es "0". Este gráfico solo sirve para ver la evolución o forma de la variable "Y".

Gridspec

Aprovechando que la introducción a line plot "**plot**" ya fue dada en el resto del material veamos ahora el uso de "gridspec" para definir el layout de nuestra figura y poder por ejemplo una distribución no simétrica de los gráficos:

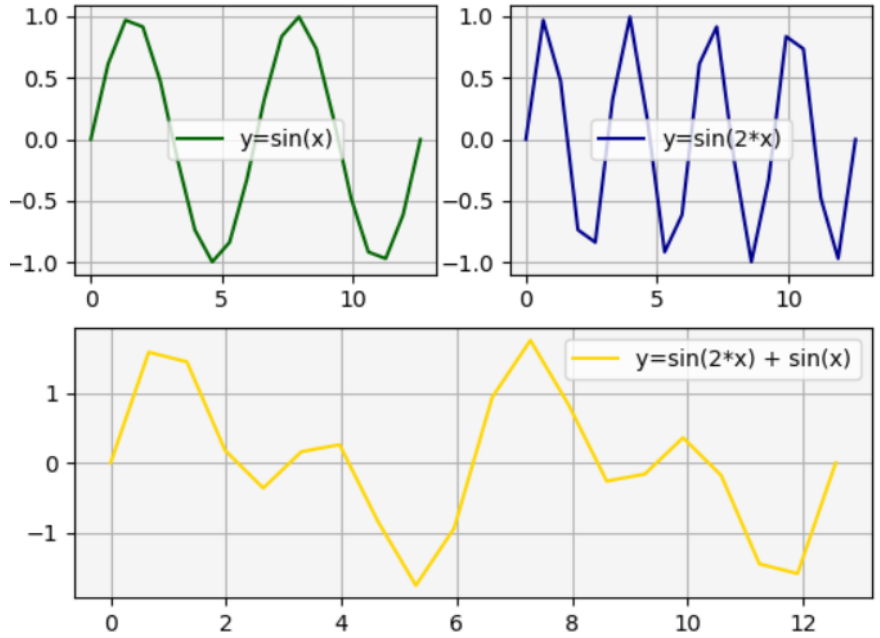
```
# Demostracion de line plot junto con gird layout
gs = gridspec.GridSpec(2, 2)    # (row, col)
fig = plt.figure()
ax1 = fig.add_subplot(gs[0, 0]) # row 0, col 0
ax2 = fig.add_subplot(gs[0, 1]) # row 0, col 1
ax3 = fig.add_subplot(gs[1, :]) # row 1, toma todas las col

x = np.linspace(0, 4*np.pi, 100)

ax1.plot(x, np.sin(x), color='darkgreen', label='y=sin(x)')
ax1.set_facecolor('whitesmoke')
ax1.legend()
ax1.grid('solid')

ax2.plot(x, np.sin(2*x), color='darkblue', label='y=sin(2*x)')
ax2.set_facecolor('whitesmoke')
ax2.legend()
ax2.grid('solid')

ax3.plot(x, np.sin(2*x) + np.sin(x), color='gold',
         label='y=sin(2*x) + sin(x)'
         )
ax3.set_facecolor('whitesmoke')
ax3.legend()
ax3.grid('solid')
plt.show()
```

Así es como se vería el gráfico resultante:



Se puede ver que el tercer gráfico ocupa dos columnas.

¿Qué pasa si bajamos la cantidad de muestras de nuestro gráfico?
Veríamos la señal con menor calidad, como por ejemplo:



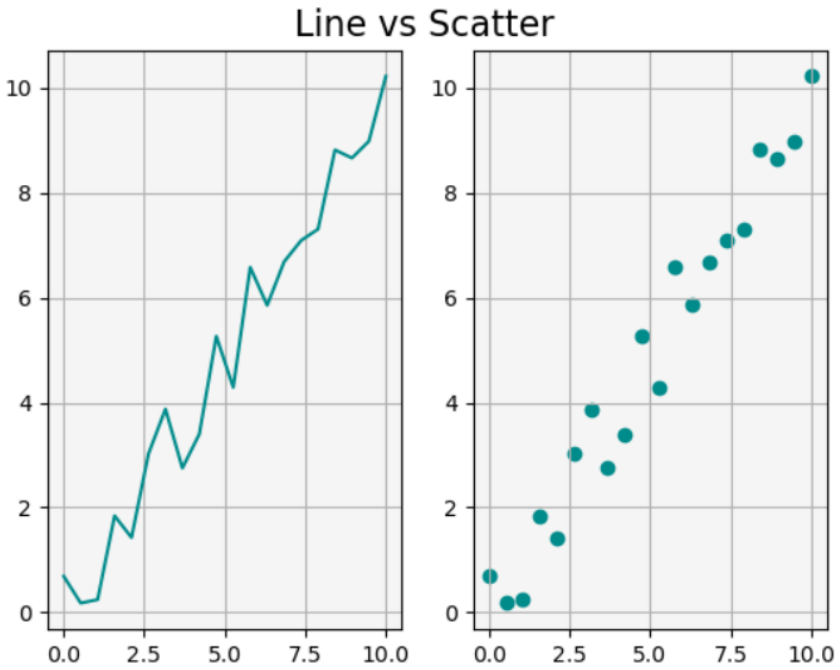
A esto se lo llama "frecuencia de sampleo", básicamente cuanto mayor cantidad de muestras tengamos para la realización del gráfico mayor calidad tendrá.

Scatter Plot

El gráfico de dispersión "**scatter**" es muy parecido al gráfico de línea "**plot**" salvo que no une por líneas las muestras. Es muy útil especialmente cuando hay ruido en las muestras o la cantidad de muestras es baja como para obtener una buena calidad en la interpolación de los puntos.

Para utilizar el "**scatter plot**" debemos llamar al método "**scatter**" en nuestro axes. Toda la personalización vista hasta el momento aplica el 100% en los gráficos de dispersión.

Veamos un ejemplo, tenemos una lista de puntos que corresponden un una línea cuya función es "**y=x**", pero las mediciones de "X" están afectadas por ruido, al unir esos puntos en un "**plot**" no veremos una recta, pero si utilizamos un **scatter** plot podemos construir la recta con nuestra "imaginación":

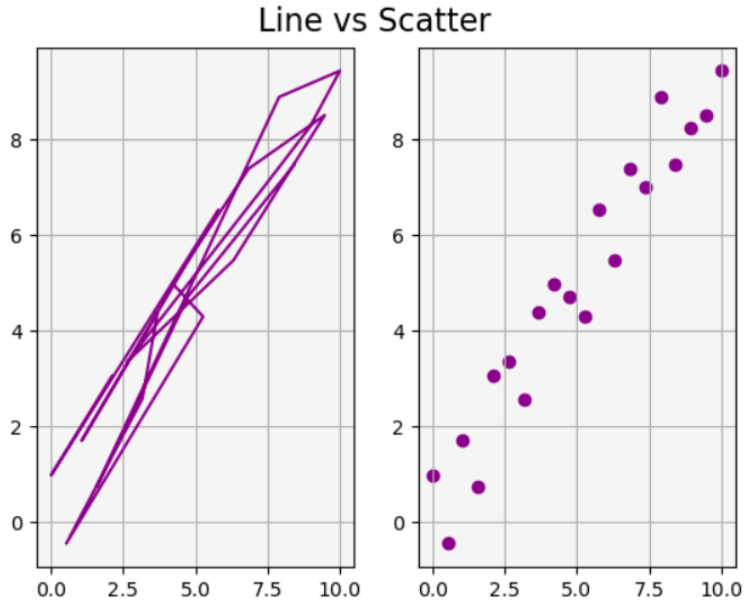


El gráfico de la derecha es el realizado con el método **scatter**, se puede ver a simple vista los puntos, como también, se puede ver que unir esos puntos con rectas no nos dará un buen resultado (tal como se ve en el gráfico de la izquierda realizado con **plot**).

Este gráfico se obtuvo mediante el siguiente código:

```
# Generaremos una linea y=x con ruido sumando
# valores aleatorios uniformes en cada punto
sample_size = 20
x = np.linspace(0, 10, sample_size)
y = x + np.random.uniform(-1, 1, sample_size)

fig = plt.figure()
fig.suptitle('Line vs Scatter', fontsize=16)
ax1 = fig.add_subplot(1, 2, 1)
ax2 = fig.add_subplot(1, 2, 2)

ax1.plot(x, y, c='darkcyan')
ax1.set_facecolor('whitesmoke')
ax1.grid('solid')
ax2.scatter(x, y, c='darkcyan')
ax2.set_facecolor('whitesmoke')
ax2.grid('solid')
plt.show()
```

Otra propiedad importante del **scatter** plot es que no se ve afectado por el orden de las muestras. En el **plot** se unen con líneas las muestras una a una a medida que se van pasando al gráfico, por lo que si las muestras no están ordenadas podría obtener un resultado lejos de lo esperado:



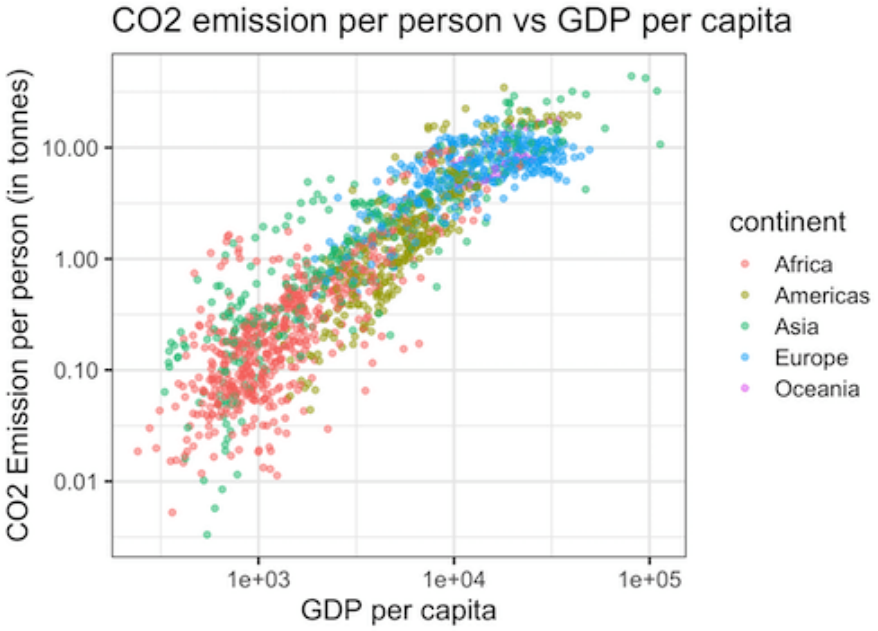
Como podemos observar, el **scatter** plot al no unir las muestras no se ve afectado por el orden en la cual fueron suministradas al gráfico.

Para generar ese gráfico y "desordenar" las muestras se utilizó el método "shuffle":

```
np.random.shuffle(x)
y = x + np.random.uniform(-1, 1, sample_size)

fig = plt.figure()
fig.suptitle('Line vs Scatter', fontsize=16)
ax1 = fig.add_subplot(1, 2, 1)
ax2 = fig.add_subplot(1, 2, 2)

ax1.plot(x, y, c='darkmagenta')
ax1.set_facecolor('whitesmoke')
ax1.grid('solid')
ax2.scatter(x, y, c='darkmagenta')
ax2.set_facecolor('whitesmoke')
ax2.grid('solid')
plt.show()
```

Aplicaciones

Se utiliza los gráficos scatter (dispersión de puntos) cuando deseamos visualizar varios grupos en un mismo gráfico con valores muy dispersos que no permitiría utilizar line plot:



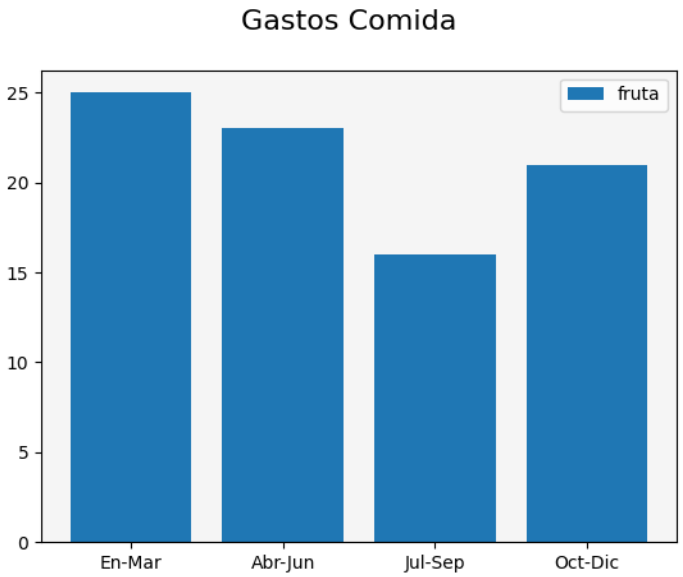
Bar plot

El gráfico de barras es uno de los gráficos más utilizados cuando se desea obtener una "mirada general" de una evolución o comparación de datos. El modo de uso es exactamente el mismo que venimos trabajando, reemplazamos "**plot"** por "**bar**".

El gráfico de barras tiene la particularidad que como datos del eje "X" acepta texto, a fin de representar con nombres cada una de las barras:

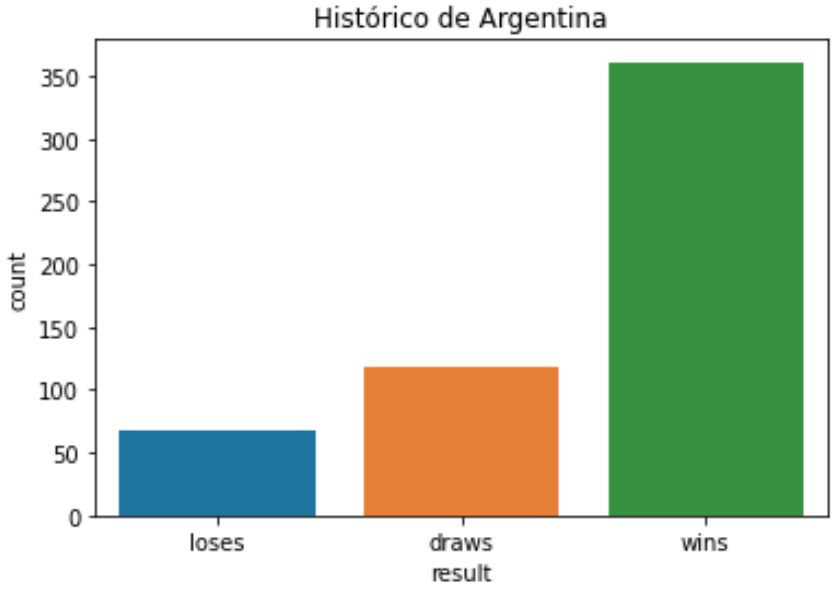
```
# Utilizar el gráfico de barras para comparar el consumo
# de fruta por trimestre
trimestres = ['En-Mar', 'Abr-Jun', 'Jul-Sep', 'Oct-Dic']
fruta = [25, 23, 16, 21]

fig = plt.figure()
fig.suptitle('Gastos Comida', fontsize=16)
ax = fig.add_subplot()

ax.bar(trimestres, fruta, label='fruta')
ax.set_facecolor('whitesmoke')
ax.legend()
plt.show()
```

Resultado:



Aplicaciones

Este tipo de gráfico no se utiliza cuando se desea observar un dato preciso o concreto, sino que cuando se desea comparar sin tanto detalle diferentes grupos. En el siguiente ejemplo se compara cuantos partidos perdio, empato y ganó la selección Argentina de fútbol, en donde el detalle no es importante sino lo que se puede observar a primera vista:



Múltiples gráficos de barras

La letra chica está en que el gráfico de barras para que nos sirva requiere trabajarlo un poco más, utilizando matplotlib a secas (es decir, sin otra librería de tercero) es realmente bastante tedioso.

Veamos un ejemplo en donde tenemos el consumo de carne, fruta y verdura durante los 4 trimestres del año. Queremos ver a grandes rasgos que se consume más de estas categorías en cada trimestre.

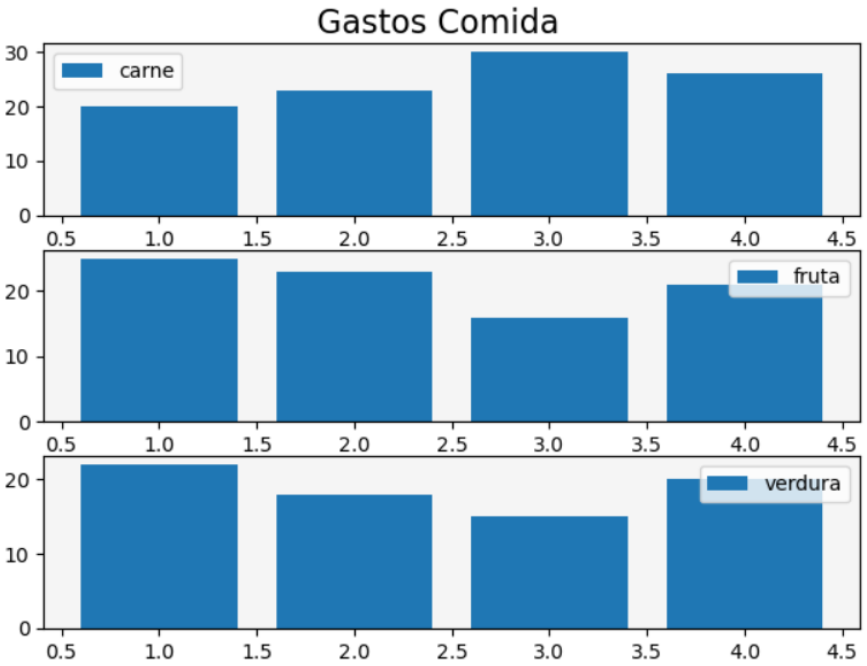
```
# Utilizar el gráfico de barras para comparar el consumo
# de productos por trimestre
trimestres = [1, 2, 3, 4]
trimestres_label = ['En-Mar', 'Abr-Jun', 'Jul-Sep', 'Oct-Dic']
carne = [20, 23, 30, 26]
fruta = [25, 23, 16, 21]
verdura = [22, 18, 15, 20]
```

Por un lado tenemos los trimestres por Nº de trimestre o nombre, y los consumos ordenados de cada categoría.

Para eso comenzaremos por realizar 3 gráficos separados de barras, un gráfico por cada categoría:

```
fig = plt.figure()
fig.suptitle('Gastos Comida', fontsize=16)
ax1 = fig.add_subplot(3, 1, 1)
ax2 = fig.add_subplot(3, 1, 2)
ax3 = fig.add_subplot(3, 1, 3)

ax1.bar(trimestres, carne, label='carne')
ax1.set_facecolor('whitesmoke')
ax1.legend()

ax2.bar(trimestres, fruta, label='fruta')
ax2.set_facecolor('whitesmoke')
ax2.legend()

ax3.bar(trimestres, verdura, label='verdura')
ax3.set_facecolor('whitesmoke')
ax3.legend()
plt.show(block=False)
```

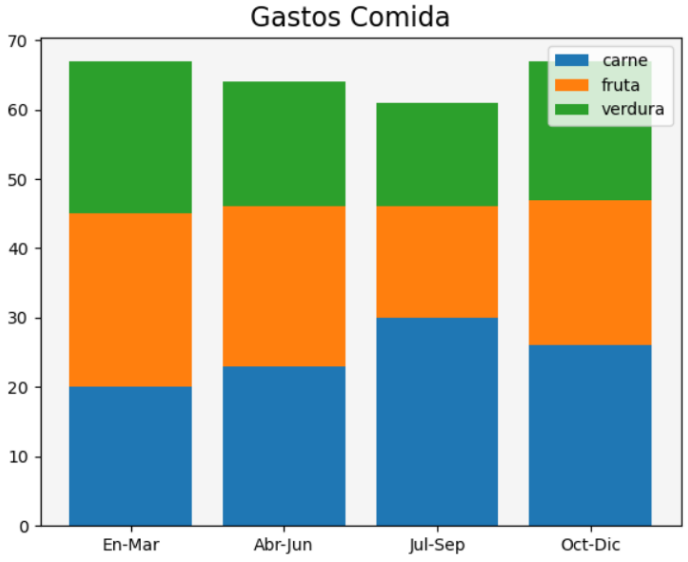
EL gráfico resultante es:



Como podemos observar, por defecto el gráfico de barras por separado no nos aporta mucha información. Habría que trabajar bastante en mejorar la interpretación con colores y ajustar las escalas. Además, es difícil interpretar cuales son los trimestres.

Bar Plot apilados (stack)

Una de las formas de mejorar la interpretación de la información en los gráficos de barras, es ampliar la información generando un único gráfico. Si apilamos esos tres gráficos veremos el siguiente resultado:



Para poder realizar este gráfico se utilizó el siguiente código

```
fig = plt.figure()
fig.suptitle('Gastos Comida', fontsize=16)
ax = fig.add_subplot()

ax.bar(trimestres_label, carne, label='carne')
ax.bar(trimestres_label, fruta, bottom=carne, label='fruta')
ax.bar(trimestres_label, verdura,
       bottom=[sum(x) for x in zip(carne, fruta)], label='verdura'
       )
ax.set_facecolor('whitesmoke')
ax.legend()
plt.show()
```

En principio parece bastante simple, es muy parecido a realizar un gráfico de múltiples líneas en "**plot**", pero hay algo importante a tener en cuenta.

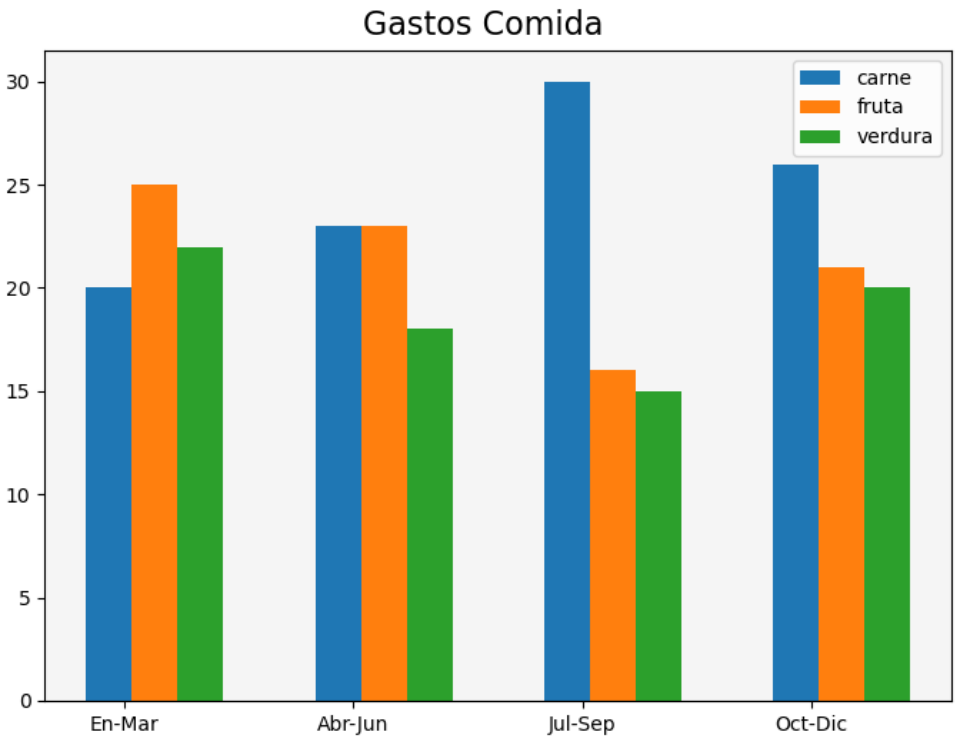
Para poder apilar los gráficos, cada uno nuevo que deseo agregar debo indicarle que gráficos o datos están debajo (**bottom**).

Si realizo solo dos gráficos apilados es bastante simple, en el segundo gráfico que se genera debe colocarse como "**bottom**" la información del primer gráfico. En este caso, el segundo gráfico "fruta" debe indicarse como "bottom" los datos del primer gráfico "carne".

Comienza a complicarse cuando realizo 3 gráficos o más apilados, ya que debo sumar vectorialmente la información de los gráficos anteriores al que sumaré a mi figura. El problema es que para hacer el gráfico "verdura" no puedo colocar como "bottom" la suma de "carne" y "fruta" como → bottom = carne + fruta → **NO**.
Esto no es posible porque en este caso utilizamos listas, y las listas no permiten la suma vectorial. Si hubiéramos utilizado Numpy hubiera sido mucho más simple. En este caso utilizando comprensión de listas con "zip" podemos realizar una "suma vectorial" y salir del paso.

Bar plot agrupados (grouped)

Otra forma de mejorar la interpretación de la información es juntar los 3 gráficos realizados en un solo, pero agrupar sin mezclar los datos de cada trimestre. En otras palabras sería alcanzar el siguiente resultado:



Debería ser simple realizar ese gráfico, pero utilizando "matplotlib" a secas realmente no lo es. Es necesario el siguiente código:

```
# Bar plot "agrupados" (grouped)
trimestres = np.array([1, 2, 3, 4])
width = 0.2 # Tamaño de la barra
fig = plt.figure()
fig.suptitle('Gastos Comida', fontsize=16)
ax = fig.add_subplot()

ax.bar(trimestres, carne, width=width, label='carne')
ax.bar(trimestres + width, fruta, width=width, label='fruta')
ax.bar(trimestres + 2*width, verdura, width=width, label='verdura')
ax.set_facecolor('whitesmoke')
ax.legend()
ax.set_xticks(trimestres + width / 3)
ax.set_xticklabels(trimestres_label)
plt.show()
```

Analicemos las distintas partes:

1 - Fue necesario generar la lista de trimestres en un array de Numpy, ya que lo necesitaremos para realizar una división vectorial más adelante.

2 - Fue necesario darle un tamaño (width) a cada barra para poder esparcirlas, donde el tamaño original es "1.0". Como son tres barras por trimestre podríamos haber definido un width = "0.33" pero para que no queden todas solapas se dejó un poco de espacio y definir width ="0.2".

```
trimestres = np.array([1, 2, 3, 4])
width = 0.2 # Tamaño de la barra
```

3 - Para que no se solapen las barras se debe colocar cada una de ellas espaciadas por el "width". La primera barra va en el "widthx0", la segunda barra en el "widthx1" y la tercera barra en el "widthx2"

```
ax.bar(trimestres, carne, width=width, label='carne')
ax.bar(trimestres + width, fruta, width=width, label='fruta')
ax.bar(trimestres + 2*width, verdura, width=width, label='verdura')
```

4 - Por último debemos indicarle al gráfico cuantos "ticks" (divisiones) tendrá y el nombre que le daremos a cada división:

```
ax.set_xticks(trimestres + width / 3)
ax.set_xticklabels(trimestres_label)
```

Realmente es bastante complejo, otras librerías de terceros como "pandas" lo resuelven mejor.

Pie plot

El gráfico de torta se usa especialmente para ver cómo fue la distribución de los datos sobre la totalidad, por ejemplo utilizado para ver cómo se distribuyeron los votos o como se distribuye los gastos.

Para este ejemplo se volvió sobre la idea de los gastos divididos en las 3 categorías: carne, fruta y verduras. Ahora los datos se agrupan en una lista de diccionarios como si los datos hubieran sido obtenidos de un archivo CSV con una columna por cada categoría, donde cada fila corresponde a un trimestre (los trimestres van del 0 al 3).

```
# Utilizar gráfico de torta para evaluar la distribución
# Segmetar el consumo en una lista de diccionarios tal como
# si hubiera venido de un archivo.
# Ahora los trimestres van del 0 al 3
consumo = [{'carne': 20, 'fruta': 25, 'verdura': 22},
           {'carne': 23, 'fruta': 23, 'verdura': 18},
           {'carne': 30, 'fruta': 16, 'verdura': 15},
           {'carne': 26, 'fruta': 21, 'verdura': 20},
           ]

trimestres = {'En-Mar':0, 'Abr-Jun':1, 'Jul-Sep':2, 'Oct-Dic':3}
```

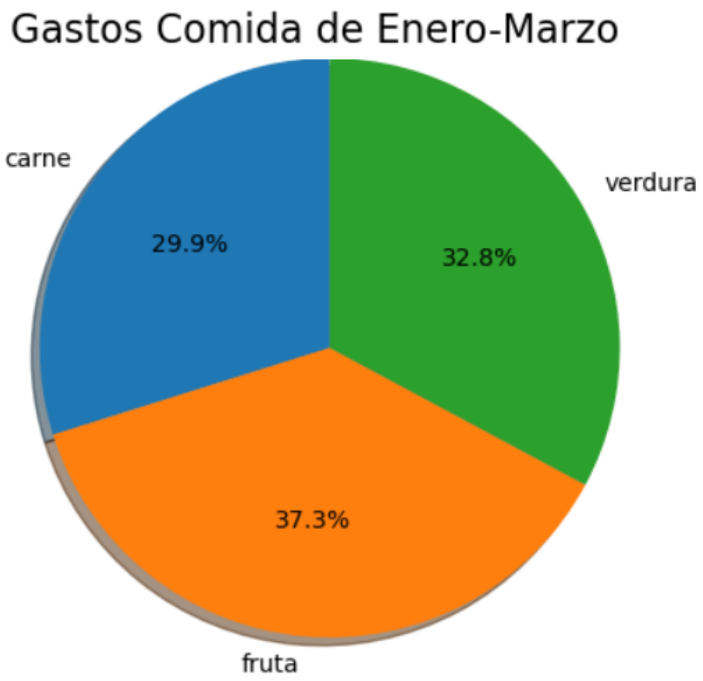
Se utiliza el diccionarios “trimestres” para traducir el “nombre” al número o índice.

Accedemos a los datos del trimestre deseado utilizando el método de “values” y a los nombre de las categorías utilizando el método “keys” de diccionarios:

```
fig = plt.figure()
fig.suptitle('Gastos Comida de Enero-Marzo', fontsize=16)
ax = fig.add_subplot()

index = trimestres['En-Mar']

ax.pie(consumo[index].values(), labels=consumo[index].keys(),
       autopct='%1.1f%%', shadow=True, startangle=90
       )
# Igualo la relacion de aspecto para que se vea como un círculo
ax.axis('equal')
plt.show(block=False)
```

- Para que en cada sección del gráfico cada categoría presente el porcentaje sobre el total se define el parámetro “**autopct**” (auto porcentaje) definiendo el formato.
- Para darle “cuerpo” y “simetría” al gráfico se definen los parámetros “**shadow**” y “**startangle**”.
- Para que el gráfico se vea como un círculo se define “**axis equal**”.

El resultado obtenido es el siguiente:

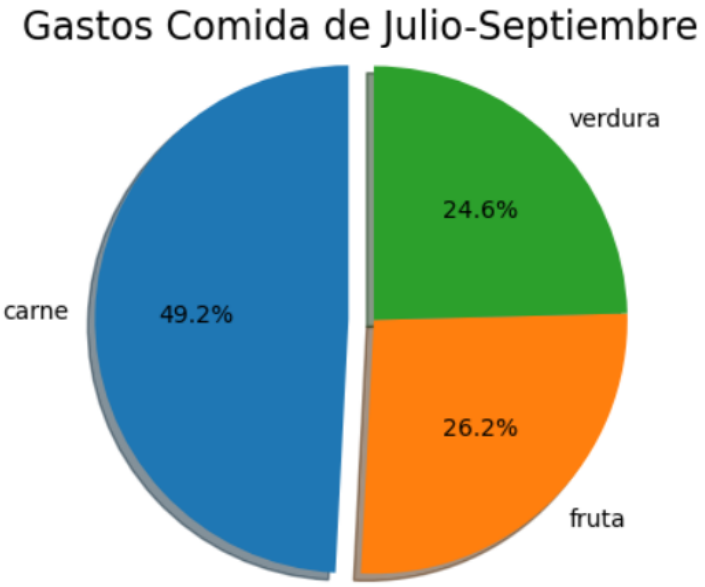


Por otro lado si deseamos resultar una de las categorías en el gráfico podemos usar el parámetro "**explode**" para ello:

```
fig = plt.figure()
fig.suptitle('Gastos Comida de Julio-Septiembre', fontsize=16)
ax = fig.add_subplot()

index = trimestres['Jul-Sep']
explode = (0.1, 0, 0)  # solo resaltar el consumo de carne

ax.pie(consumo[index].values(), labels=consumo[index].keys(),
       explode=explode, autopct='%1.1f%%', shadow=True, startangle=90
       )
ax.axis('equal')
plt.show()
```

En este caso estamos informando que queremos resaltar la primera categoría "carne".

El gráfico resultante es el siguiente:



Seaborn

El que quiera investigar un poco más acerca de esta librería que "aumenta" las funcionalidades de matplotlib (y simplifica algunas cuestiones estéticas) en el repositorio encontrarán ejemplos de cómo utilizarla (básicos). Se usa exactamente igual de la forma que venimos utilizando todos los tipos de gráficos de matplotlib.

Para poder correr los ejemplos primero tienen que instalarse seaborn con pip:

```
pip3 install seaborn
```

Seaborn normalmente se lo importa en Python como:

```
import seaborn as sns
```

Descargar un repositorio de GitHub

Para poder realizar las actividades de aquí en adelante debemos tener instalado y configurado nuestro GitHub. Todos los ejemplos prácticos estarán subidos al repositorio GitHub de **InoveAlumnos**, para aprender como descargar estos ejemplos desde el repositorio referirse al "Instructivo de GitHub: Descargar un repositorio" disponible entre los archivos del campus. De no encontrarse allí, por favor, tenga a bien comunicarse con alumnos@inove.com.ar para su solicitud.

Debemos descargar el repositorio que contiene los ejemplos de clase de ésta unidad:
https://github.com/InoveAlumnos/intro_matplotlib_python

Hasta la próxima!

Con esto finaliza el tema "matplotlib", a partir de ahora tienen las herramientas para poder comenzar a realizar sus gráficos!!.

Si desean conocer más detalles sobre el contenido pueden iniciar un tema de discusión en el foro del campus, o visitar los "Links de interés" que se encuentran al final de este apunte.

Links de interés

- Documentación oficial PIP
- Matplotlib home page
- Matplotlib guía oficial de uso
- Matplotlib ejemplos con Python
- Matplotlib tipos de gráficos
- Matplotlib galería de ejemplos
- Guia matplotlib
- Ejemplo de uso de marcadores
- Colores disponibles
- Resumen de colores, marcadores y tipos de línea
- Marcadores disponibles
- Tipos de grillas
- Introducción a Git
- Fundamentos de Git