Python: Librerie Scientifiche: Matplotlib

Rights & Credits

Questo notebook è stato creato da Agostino Migliore.

Matplotlib è una libreria di grafica per il linguaggio di programmazione Python e la sua estensione numerica/matematica NumPy. Essa fornisce anche tutti gli strumenti per poter inglobare grafica in applicazioni varie, incluso il presente jupyter notebook.
Matplotlib consente di creare grafici di grande qualità in modo semplice. Tale libreria è scritta sostanzialmente in linguaggio Python, ma fa ampio uso di NumPy ed altre estensioni per conseguire una buona performance con grandi arrays di dati.

Ci sono due possibili approcci all'uso di matplotlib: l'interfaccia funzionale Pyplot e la cosiddetta interfaccia "Axes" ovvero Orientata agli Oggetti (OO).

Pyplot Interface

matplotlib.pyplot è una collezione di funzioni che permettono di generare grafici e lavorarvi su con `matplotlib` in modo simile a quanto si farebbe con MATLAB, quindi con lo stesso livello di rapidità e semplicità. Le varie funzioni di `pyplot` consentono di generare una figura e apporatere diverse modifiche e aggiunte successive alla stessa figura: per esempio, crea la figura e la apre sullo schermo, crea l'area in cui plottare all'interno della figura, traccia dei grafici, aggiunge decorazioni come i nomi degli assi, ecc. Infatti, pyplot viene definita come una interfaccia a matplotlib basata sullo stato (*state-based interface*). Lo stato principale è rappresentato dalla figura stessa creata, o comunque attualmente in uso, e dalle su proprietà. Viene preservata memoria dello stato della figura man mano che si usano le varie funzioni di `pyplot` per modificarla. Infatti, nell'invocare una funzione di `pyplot` per effettuare modifiche su una figura, non abbiamo bisogno di specificare esplicitamente a quale figura facciamo riferimento e quali siano le caratteristiche della figura nel momento in cui tale funzione viene chiamata in causa. Iniziamo con l'importare `pyplot`:

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

Abbiamo importato pure `numpy` in quanto diverse cose necessarie per la generazione delle figure, a partire dai set di dati, sono realizzate usando `numpy`.
Definiamo valori di `x` e corrispondenti valori di `y` da plottare:

In [2]:
```
x = np.arange(0, 12, 0.1)
```

```
y = np.sin(x)
```

matplotlib.pyplot.plot (plt.plot)

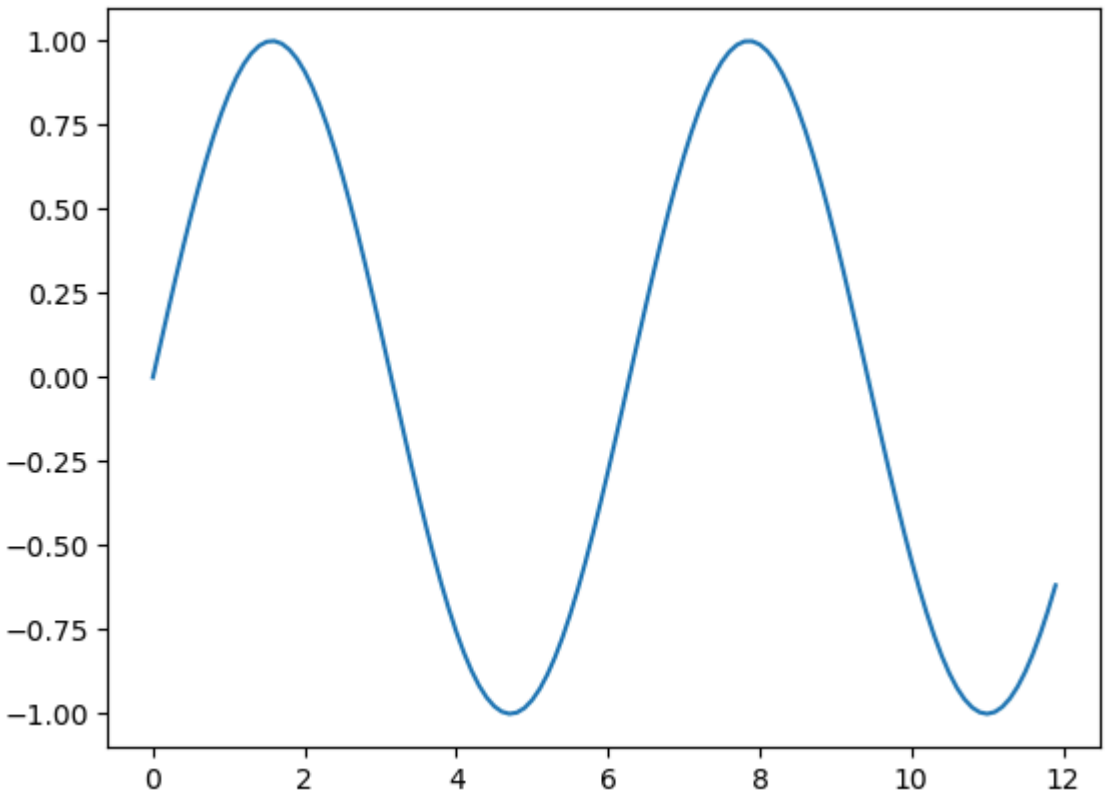
Abbiamo usato la funzione `np.arange` vista precedentemente per generare una ndarray di valori di `x` e la ufunc trigonometrica `np.sin` per ottenere i corrispondenti valori di y in un'altra ndarray. Per plottare il grafico di y in funzione di x, usiamo la funzione `plot` di `matplotlib.pyplot` , cioè matplotlib.pyplot.plot, che possiamo scrivere semplicemente come plt.plot grazie all'alias introdotto prima nell'importare. La sintassi per l'uso di tale funzione è come segue:

```
plt.plot(*args, scalex=True, scaley=True, data=None, **kwargs)
```

Come abbiamo visto `*args` denota un numero arbitrario di argomenti da passare alla funzione. Come minimo, tra trali argomenti dobbiamo fornire i punti `x, y` da plottare. I parametri di input `scalex` e `scaley` dicono alla funzione se i valori minimi e massimi sugli assi devono essere adattati ai ranges di dati. Di default, i valori di tali parametri sono settati a `True` . Poi ci sono altri parametri che non discuteremo qui. Usiamo prima la funzione nel modo più semplice per plottare i punti `(x, y)` creati prima (si noti che non è necessario indicare gli argomenti con valori di default, che quindi rimangono tali):

In [3]:
```
plt.plot(x,y)
```

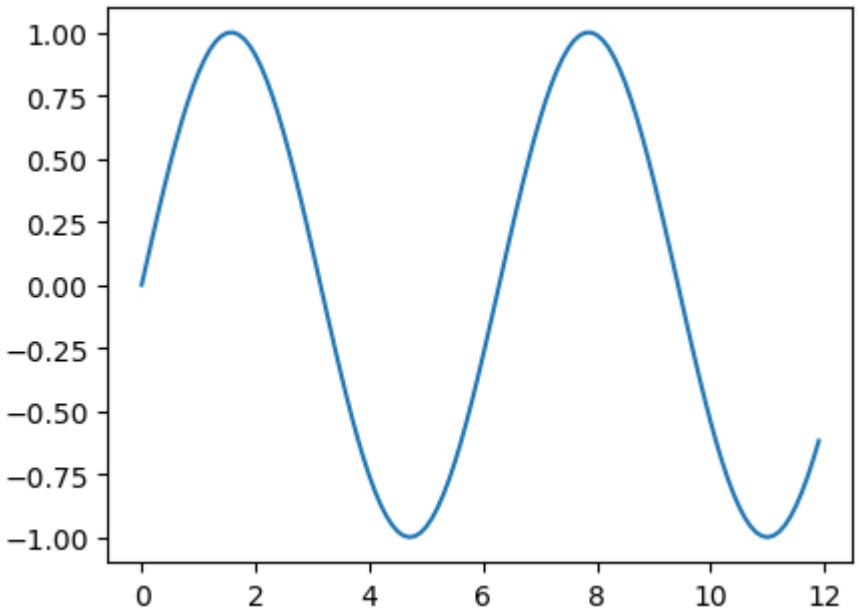
Out[3]:
```
[<matplotlib.lines.Line2D at 0x159def4cf90>]
```



Le dimensioni della figura di sopra sono quelle predefinite (di default) di Matplotlib. La configurazione predefinita di Matplotlib può anche essere modificata in fase di esecuzione (ciò viene indicato con l'acronimo *rc*, che sta per *runtime configuration*), sfruttando matplotlib.rcParams, che è un archivio di valori chiave con formato simile a un *dict* per i parametri di configurazione di Matplotlib. Per esempio, fra tali parametri c'è figure.figsize, che stabilisce le dimensioni delle figure in pollici e di cui cambiamo il valore come segue:

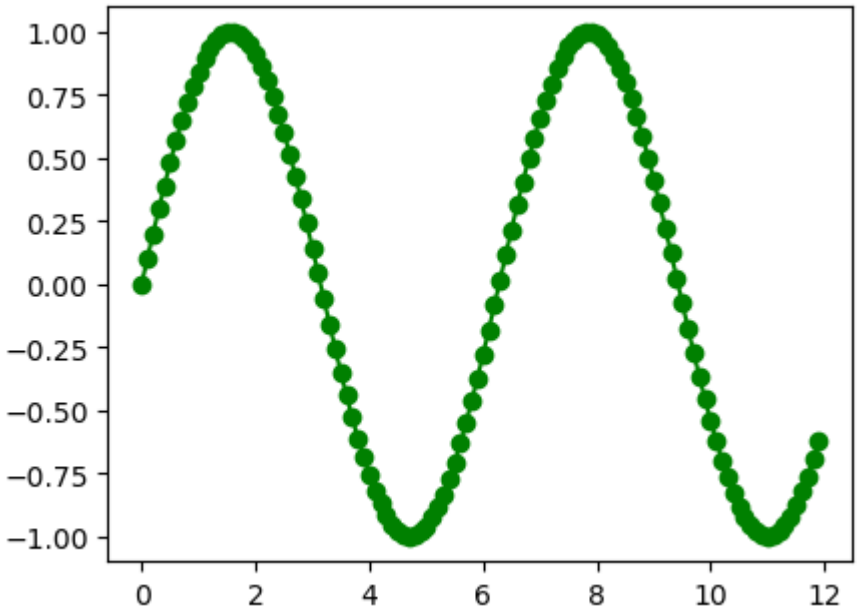
In [4]:
```
import matplotlib
matplotlib.rcParams["figure.figsize"] = (4.8,3.6)
plt.plot(x,y)
```

Out[4]:
```
[<matplotlib.lines.Line2D at 0x159def26a50>]
```



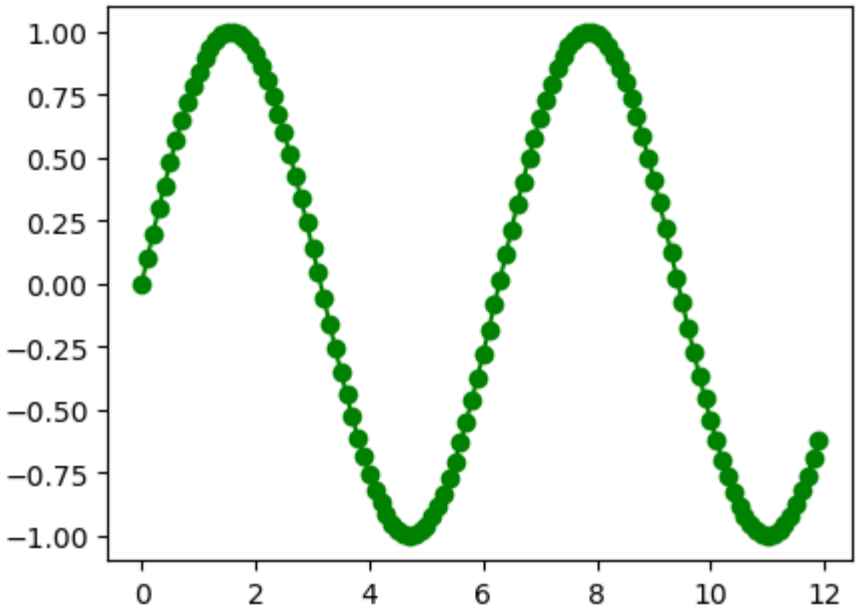
Adesso aggiungiamo un altro dei parametri possibili compresi in `*args`. In particolare, vogliamo aggiungere degli indicatori di formato, `fmt`, che sono come quelli usati da MATLAB per decidere il simbolo (*marker*) per i punti dei dati (*data points*), il tipo di linea (*linestyle*) e il colore (*color*). Tali indicatori sono forniti sotto forma di una stringa. Per esempio, la lettera `o` richiede di mostrare i punti dei dati con tondini; un trattino, `-`, indica di interpolare i punti con una linea continua; e `g` richiede di usare il colore verde:

In [5]:
```
plt.plot(x,y,'o-g')
plt.show()
```



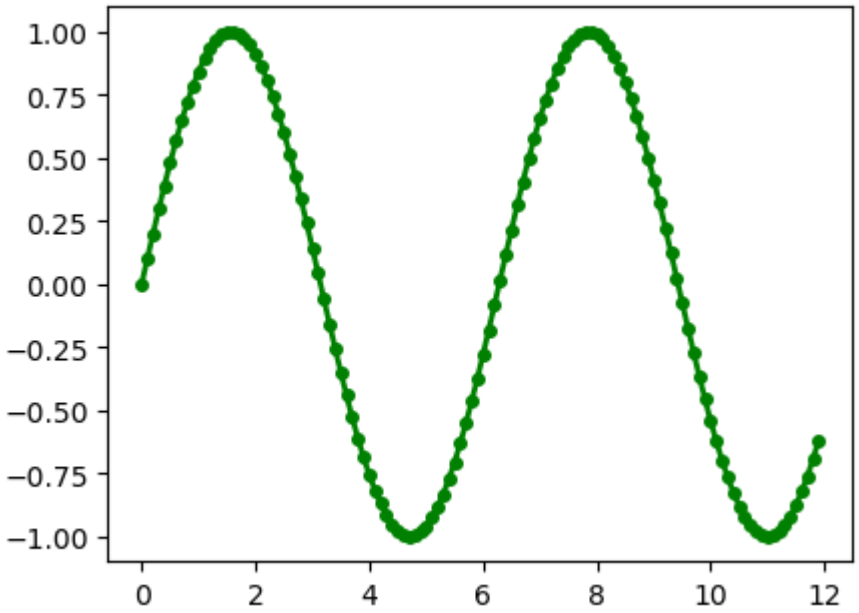
Si noti che abbiamo pure usato la funzione `plt.show()` vista precedentemente per evitare di vedere, sopra il grafico, la linea che indica l'oggetto pronto da plottare e rendere direttamente l'oggetto grafico. Alternativamente, si poteva usare `;`.
Equivalentemente a quanto fatto sopra, i tipi di formato da usare si possono fornire per nome:

In [6]:
```
plt.plot(x, y, marker='o', linestyle='-', color='g');
```



Vediamo che i tondini sono molto grandi. Se ne vogliamo diminuire la dimensione e, per esempio, vogliamo pure decidere lo spessore della linea, possiamo aggiungere altri due argomenti in `*args` come segue:

In [7]:
```
plt.plot(x,y,marker='o', markersize=4, linestyle='-', linewidth=2, color='g');
```



Adesso, creiamo un'altra nadarray per plottare nella stessa figura il $cos(x)$:

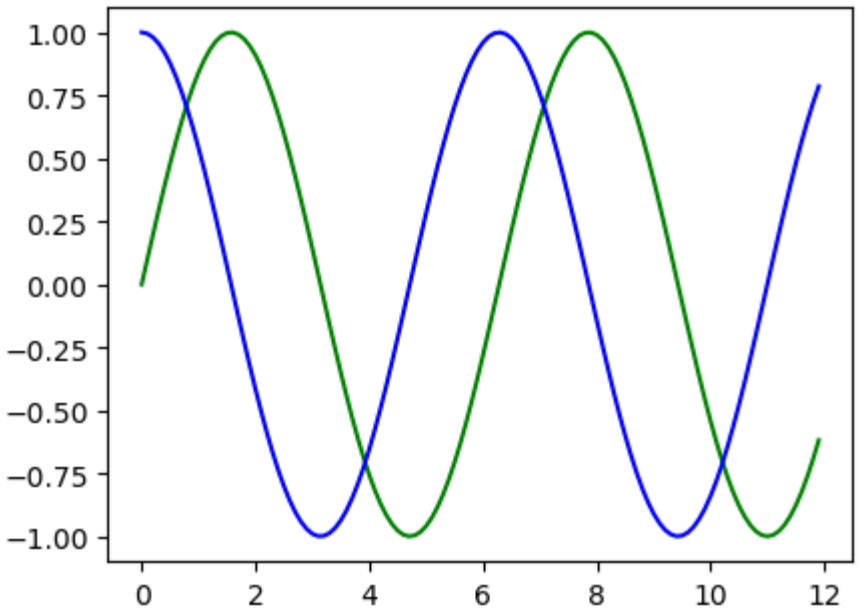
In [8]:
```
y2 = np.cos(x)
```

A questo punto, possiamo usare la sintassi

`plt.plot(x1,y1,fmt1,x2,y2,fmt2)`

per sovrapporre i due grafici nella stessa figura. Volendo solo rappresentare le linee di interpolazione con colore diverso, ed essendo `x1 = x2 = x`, procediamo come segue:

In [9]:
```
plt.plot(x,y,'-g',x,y2,'-b');
```



Riporto sotto i caratteri usati per definire i diversi simboli dei punti di dati, stili di linea e colori.

Markers

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

Stili di Linea e Colori

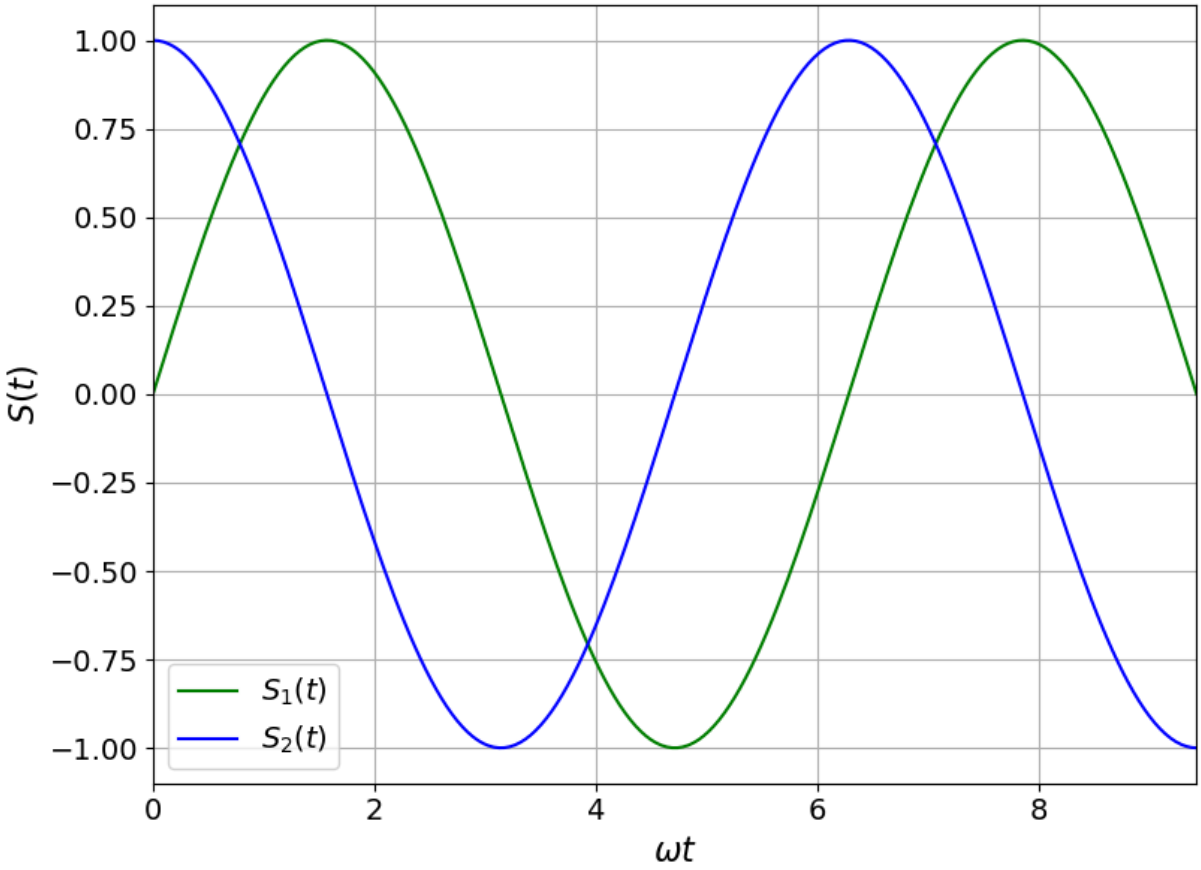
character	description
'-'	solid line
'--'	dashed line
'-.'	dash-dot line
':'	dotted line

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

Infine, vogliamo alterare le dimensioni della figura e aggiungere diversi elementi alla figura come segue, assumendo che l'asse `x` rappresenti ωt (dove ω è una frequenza e t il tempo) e sull'asse `y` siano rappresentati i segnali periodici $S_1(t)$ e $S_2(t)$:

In [10]:
```
x = np.linspace(0, 3*np.pi, 200)
y = np.sin(x)
y2 = np.cos(x)
plt.figure(figsize=(9.00,6.75), dpi=100)
plt.plot(x,y,'-g', x,y2,'-b')
plt.margins(x=0.0)
plt.xticks(fontsize=14)
```

```
plt.yticks(fontsize=14)
fp = {'family':'sans-serif', 'size':16}
plt.xlabel("$\omega t$", fontdict=fp)
plt.ylabel("$S(t)$", fontdict=fp)
plt.legend(["$S_1(t)$", "$S_2(t)$"], loc = "lower left", fontsize=14)
plt.grid(True)
plt.savefig("Figura_1.jpg")
plt.show()
```

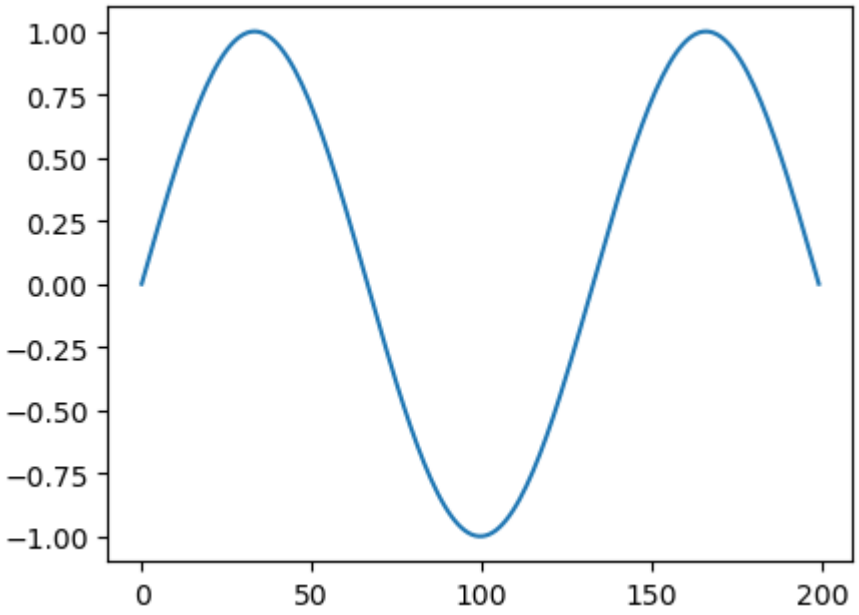


In questo caso abbiamo scelto np.linspace per generare i dati, anche se non era necessario cambiarli. La funzione plt.figure serve per creare un oggetto figura in generale e ci consente di sceglierne la risoluzione e le dimensioni, che abbiamo aumentato in questo caso. La funzione plt.margins è stata usata per imporre zero margine ai lati delle curve lungo l'asse delle ascisse. plt.xticks e plt.yticks ci hanno consentito di variare le dimensioni dei numeri sugli assi `x` e `y` rispettivamente. Le funzioni plt.xlabel e plt.ylabel ci hanno permesso di scegliere i titoli degli assi, il tipo di carattere (*font*, per il quale abbiamo scelto `sans-serif`) da usare nello scrivere tali titoli e la sua grandezza (*size*). Abbiamo usato plt.legend per aggiungere la leggenda, potendone scegliere la collocazione (in inglese, *location*, da cui il nome `loc` del parametro). Abbiamo aggiunto una griglia tramite la funzione plt.grid. Infine, con plt.savefig abbiamo salvato la figura nel file `Figura_1.png` che si trova nella stessa cartella di questo direttorio. Il tipo `png` è scelto di default, per cui non è necessario scrivere tale estensione qualora si voglia un'immagine `png`. Altrimenti, come sopra, bisogna scrivere esplicitamente l'estensione del nome del file.

Breve nota

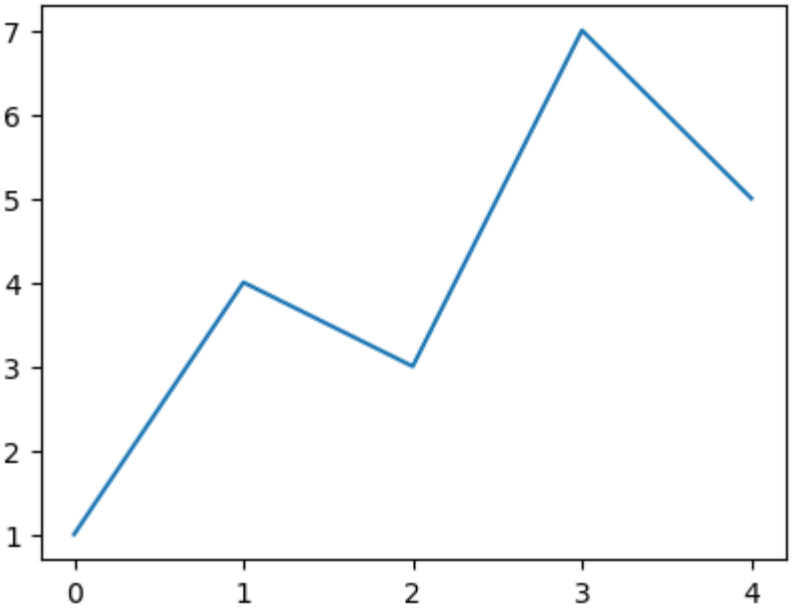
La variabile indipendente (sull'asse delle ascisse) può essere omessa come argomento di `plt.plot`, purché i suoi valori siano stati precedentemente definiti:

In [11]:
```
y = np.sin(x)
plt.plot(y);
```



Se si assegnano `N` valori di una qualsiasi quantità `z` senza fornire i valori di una variabile indipendente, si assume che questi ultimi siano `0, 1, ..., N - 1`, come sotto:

In [12]:
```
z = [1, 4, 3, 7, 5]; plt.plot(z);
```



Usando l'applicazione di interfaccia grafica Qt, si possono far comparire i grafici in finestre separate, dove possono essere laborati (nomi degli assi, ecc.) similmente a quanto si fa in

MATLAB e poi salvati. Per fare ciò, bisogna prima esguire la riga di comando

`%matplotlib qt`

Una volta che ciò viene fatto, se si vuole uscire da tale modalità e ritornare a quella attuale, bisogna eseguire

`%matplotlib inline`

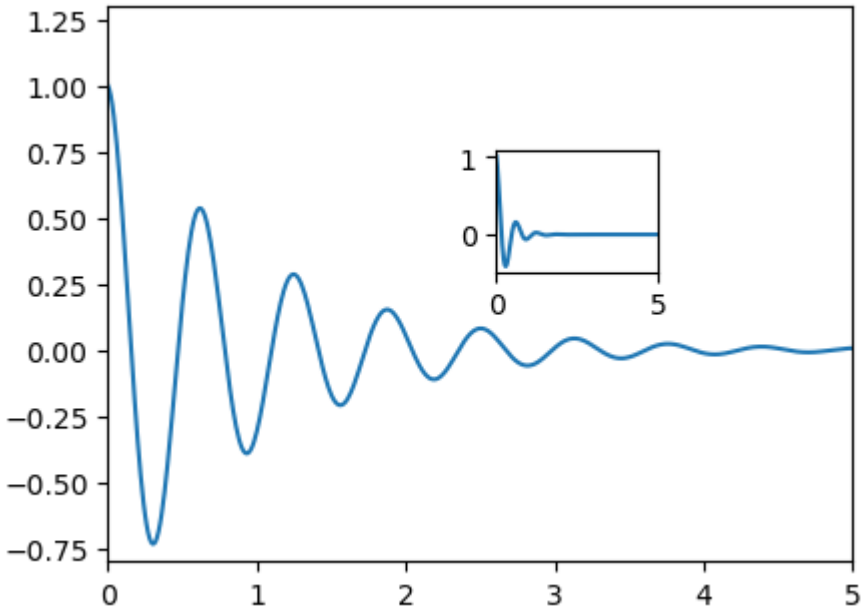
In [13]:
```
%matplotlib qt
```

In [14]:
```
x = np.linspace(0, 3*np.pi, 200)
y = np.sin(x)
y2 = np.cos(x)
plt.figure(figsize=(9.6,7.2), dpi=100)
plt.plot(x,y,'-g', x,y2,'-b')
plt.margins(x=0.0)
plt.xticks(fontsize=14)
plt.yticks(fontsize=14)
fp = {'family':'sans-serif', 'size':16}
plt.xlabel("$\omega t$", fontdict=fp)
plt.ylabel("$S(t)$", fontdict=fp)
plt.legend(["$S_1(t)$", "$S_2(t)$"], loc = "lower left", fontsize=14)
plt.grid(True)
plt.savefig("Figura_1.jpg")
plt.show()
```

In [15]:
```
%matplotlib inline
```

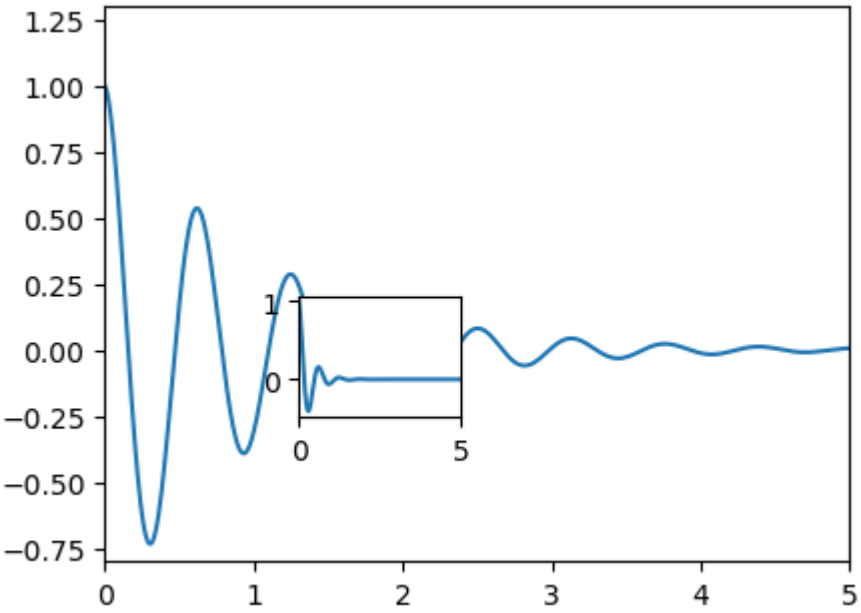
Si può creare un inserto o sottografico (*subplot*) di una figura usando la funzione plt.subtplot. Facciamo un esempio.

In [16]:
```
x = np.linspace(0, 10, 1000)
y = np.cos(10*x)*np.exp(-x)
plt.plot(x,y)
plt.xlim(0,5)
plt.ylim(-0.8,1.3)
x2 = np.linspace(0, 10, 1000)
y2 = np.cos(10*x)*np.exp(-3*x)
plt.subplot(447) # notazione alternativa: plt.subplot(4,4,7)
plt.xlim(0,5)
plt.plot(x2,y2)
plt.show()
```



Comunemente, si passano esplicitamente tre numeri alla funzione `plt.subtplot` che rappresentano il numero di righe, il numero di colonne e l'indice. Sopra, abbiamo rispettivamente usato i valori 4, 4 e 7. Questo significa che dividiamo la superficie della figura in quattro righe e quattro colonne e l'inserto può essere inserito in 16 posizioni diverse. L'indice varia dunque da 1 a 16. La posizione 1 corrisponde all'angolo in alto a sinistra e l'indice varia scorrendo riga per riga.
Si noti che, non appena viene creato l'inserto, esso diventa la figura "attuale", per cui dopo `plt.plot` agisce su di esso senza necessità di specificarlo.
Attenzione: plt.subplot può dar luogo a sovrapposizioni indesiderate (e la funzione plt.axes si potrebbe, per esempio, usare in modo opportuno come alternativa), come si vede sotto.

In [17]:
```
x = np.linspace(0, 10, 1000)
y = np.cos(10*x)*np.exp(-x)
plt.plot(x,y)
plt.xlim(0,5)
plt.ylim(-0.8,1.3)
x2 = np.linspace(0, 10, 1000)
y2 = np.cos(10*x)*np.exp(-3*x)
plt.subplot(4,4,10) # scelta non buona
plt.xlim(0,5)
plt.plot(x2,y2)
plt.show()
```



Tali sovrapposizioni sono certamente evitate se non si usa `plt.subplot` per creare inserti in una figura principale, ma grafici diversi opportunamente collocati l'uno rispetto all'altro in una figura composta (vedi sezione seguente).
Come vedremo nella prossima sezione, la funzione `subplots` di `matplotlib.pyplot`, cioè plt.subplots, può creare un oggetto figura con un ben organizzato set di grafici.

Interfaccia OO

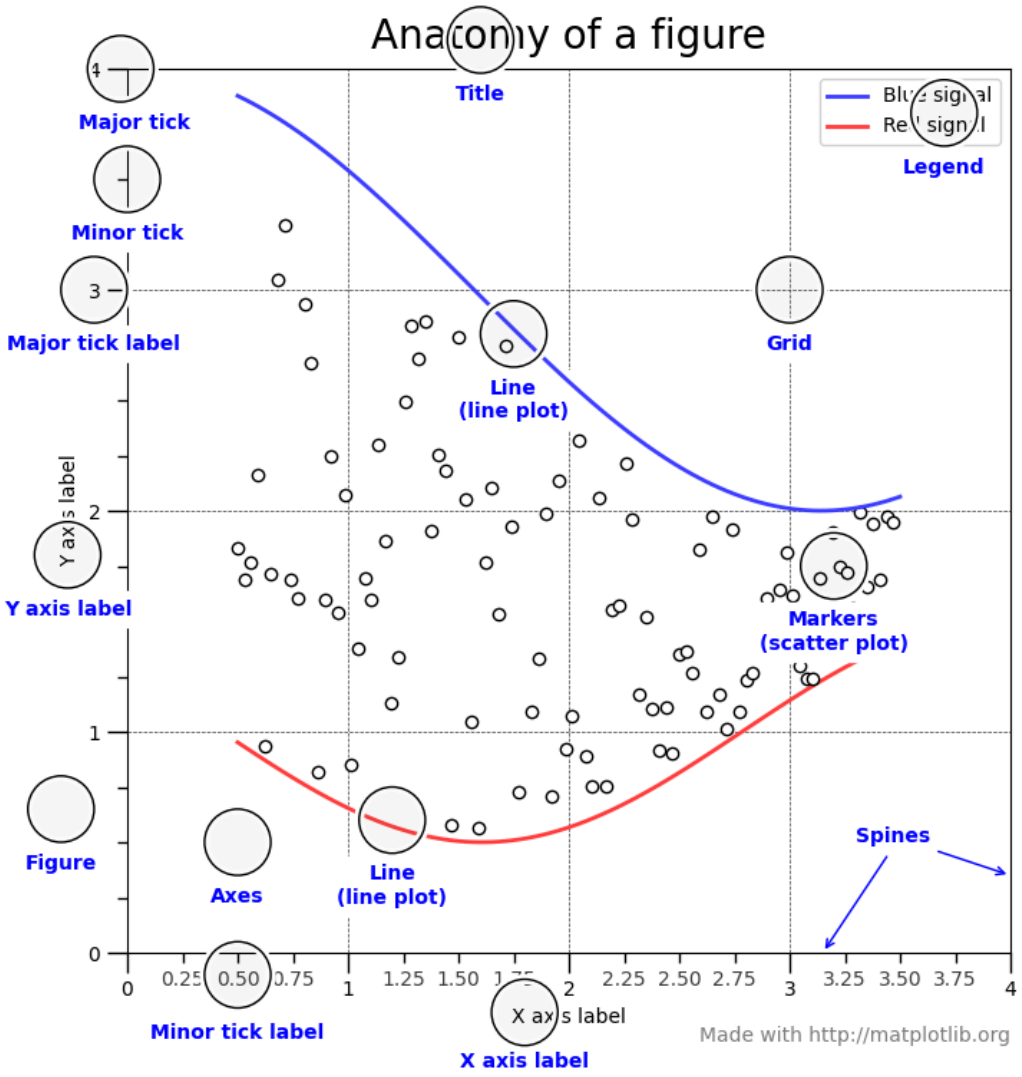
L'approccio che fa uso di `matplotlib.pyplot` (qui chiamato `plt`) è molto intuitivo e comodo per lavorare interattivamente. Come abbiamo visto, esso consente di fare una figura, di decorarla in vari modi e di disegnare diversi grafici nella stessa figura. L'oggetto Python che ingloba tutti gli elementi di un singolo plot all'interno di una figura (punti dei dati in tale plot, curva, eventuale leggenda, ecc.) viene chiamato Axes, da non confondersi con Axis che è invece l'oggetto Python che rappresenta gli assi del sistema di riferimento.

L'uso di `plt` diventa problematico se, per esempio, vogliamo rappresentare simultaneamente due scale diverse sull'asse delle ordinate (una a sinistra e una a destra) o se, in generale, vogliamo realizzare figure molto complesse. Diventa allora più conveniente o necessario usare la `interfaccia OO` (in inglese, la *object-oriented Application Programming Interface* o *object-oriented API*) di Matplotlib. Tuttavia, anche nel caso di tali figure complesse, è molto spesso conveniente continuare ad usare le funzioni `plt.figure`, `plt.subplot`, `plt.subplots` e `plt.savefig`, perché semplificano molto la programmazione delle figure. Spesso è conveniente procedere come segue.

1. Si realizzano le strutture della figura (ove sia necessario costruire un oggetto figura complessivo che contiene tutti i plot) con `plt.figure` e dei suoi vari `Axes` con `plt.subplot` o `plt.subplots` a seconda del caso (vedremo esempi di entrambi gli

utilizzi). Qui stiamo parlando del modello dell'oggetto figura, della sua struttura, del suo template (come il template di un file), che poi va riempito con i contenuti desiderati, e delle strutture degli oggetti `Axes`, che poi riempiremo con grafici e decorazioni varie.
2. Dopo il punto 1, si lavora sui vari oggetti usando esclusivamente la `interfaccia OO` per creare i grafici e aggiungere tutti gli elementi desiderati.
3. Alla fine, si può sempre usare `plt.savefig` per salvare la figura.

Usando la `interfaccia OO`, le varie componenti della figura sono oggetti dotati di metodi (come abbiamo visto finora per tanti altri oggetti) e si usano tali metodi per fare tutto ciò che serve nel rappresentare la figura. A questo punto, è dunque necessario conoscere gli oggetti grafici essenziali, che sono ben esemplificati dalla seguente figura derivata da una pagina web di Matplotlib.



L'oggetto di livello più alto è l'oggetto Figure, cioè la figura complessiva, il contenitore di tutti gli elementi grafici.
Tutti gli elementi grafici resi in una figura si chiamano Artists e sono indicati in blu nella

figura di sopra. Essi includono testo sulla figura, oggetti come per esempio frecce e forme (tali oggetti sono chiamati *patches* in inglese e sono caratterizzati dall'avere un colore di riempimento o di faccia, in inglese detto *face color*, e un colore del bordo, *edge color*, come le forme in powerpoint), gli stessi `Axes`, ecc. Gli `Axes` sono quindi un sottoinsieme dell'insieme degli `Artists`.
Un `Axes` contiene un singolo set di dati che si mostra nella figura e gli elementi connessi, inclusi due o più oggetti di tipo `Axis`. L'oggetto `Axis` può essere `XAxis` (asse delle ascisse) o `YAxis` (asse delle ordinate) e ciascuno di tali oggetti contiene gli ementi che rappresentano le tacche e i numeri, nonché i nomi degli assi.
In ultima analisi, la gerarchia degli oggeti grafici è

Figure ⊃ Artist ⊃ Axes ⊃ Axis.

Ulteriori informazioni si possono trovare nelle pagine web al link 1 e al link 2. Vediamo l'applicazione di tali punti mediante alcuni esempi.

Esempio 1: uso di plt.plot con l'interfaccia OO

Il codice di sotto è derivato dalla galleria di esempi di `Matplotlib`. La figura risultante, mostrata sotto, presenta l'andamento di una funzione oscillante la cui ampiezza decade esponenzialmente nel tempo (dove avete visto una funzione di questo tipo?), unitamente ad una sua immagine rimpicciolita (visione d'insieme da lontano) e l'ingrandimento di una sua parte. Si noti che in tale immagine non viene rappresentato un grafico principale con i vari plot come inserti. Non vi è, quindi, il rischio di sovrapposizione visto prima ed è sufficiente usare ripetutamente la funzione `plt.subplot` per creare i vari grafici in opportune posizioni diverese, in modo tale che non si sovrappongano tra di loro. Per costruire tale figura composita procediamo come segue:

- Definiamo una funzione $f(t)$ che fornisce i valori della variabile dipendente.
- Usiamo la funzione `np.arange` per creare il set, denominato `t1`, dei valori della variabile indipenendete, cioè il tempo.
- Usando la funzione `plt.subplot`, si crea la struttura per un primo grafico, richiedendo di utilizzare un'area suddivisa in due righe ed una colonna e di introdurre il grafico nella seconda riga. Essendovi solo due righe e una colonna, l'indice di posizione può assumere solo due valori, per cui la seconda riga corrisponde al valore 2 dell'indice (posizione del pannnelo: `212`). Andando a guardare la descrizione di tale funzione, si vede che l'output di tale funzione è un oggetto `Axes`. Assegniamo tale valore di output ad `ax1`. Per essere più precisi, tale oggetto è una *instance* della classe `matplotlib.axes._axes.Axes`, come potete vedere eseguendo `type(ax1)` dopo aver eseguito la cella sottostante (perché dovete prima creare `ax1`).

- A questo punto, usiamo metodi dell'oggetto denominato `ax1`. Il primo è margins. Infatti, eseguendo `help(ax1.margins)`, potete vedere che `margins` è un metodo della classe `matplotlib.axes._axes.Axes` che consente di aggiungere spazio oltre a quello richiesto dalla curva graficata nelle direzioni degli assi `x` e `y`.

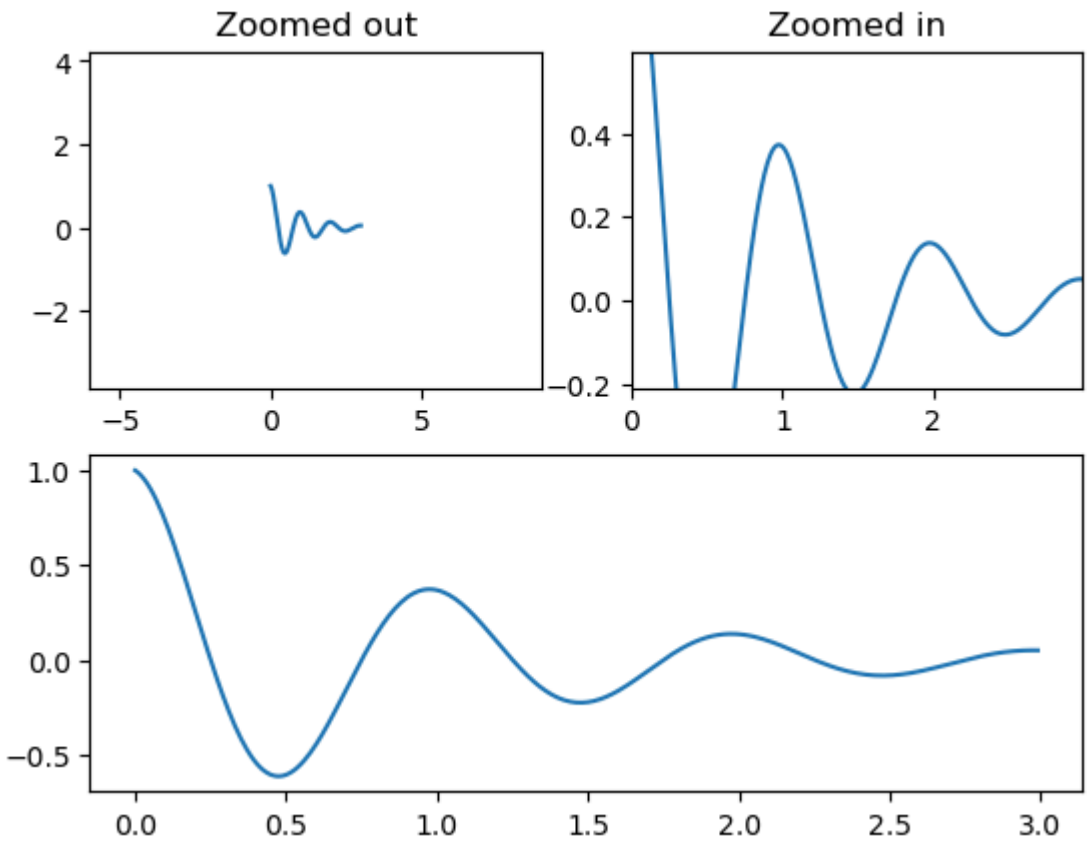
- Tale classe ha anche un metodo plot (prima usavamo la funzione `plt.plot`, adesso utilizziamo una funzione di `matplotlib.axes._axes.Axes` e, nello specifico, la usiamo come metodo applicato all'*instance* `ax1`, cioè `ax1.plot`). Tale funzione crea il grafico di $f(t)$ usando i valori di t nella ndarray `t1`.

- Procediamo analogamente per l'oggetto grafico `ax2`, usando margini più grandi, in modo tale da ridurre la parte di area dedicata al grafico, che quindi appare rimpicciolito. L'area grafica si considera, adesso, divisa in 2 righe e due colonne e il pannello viene messo nella posizione 1, cioè in alto a sinistra (posizione del pannello: `221`).

- Aggiungiamo un titolo al pannello `ax2`, tramite la funzione `matplotlib.axes.Axes.set_title`.

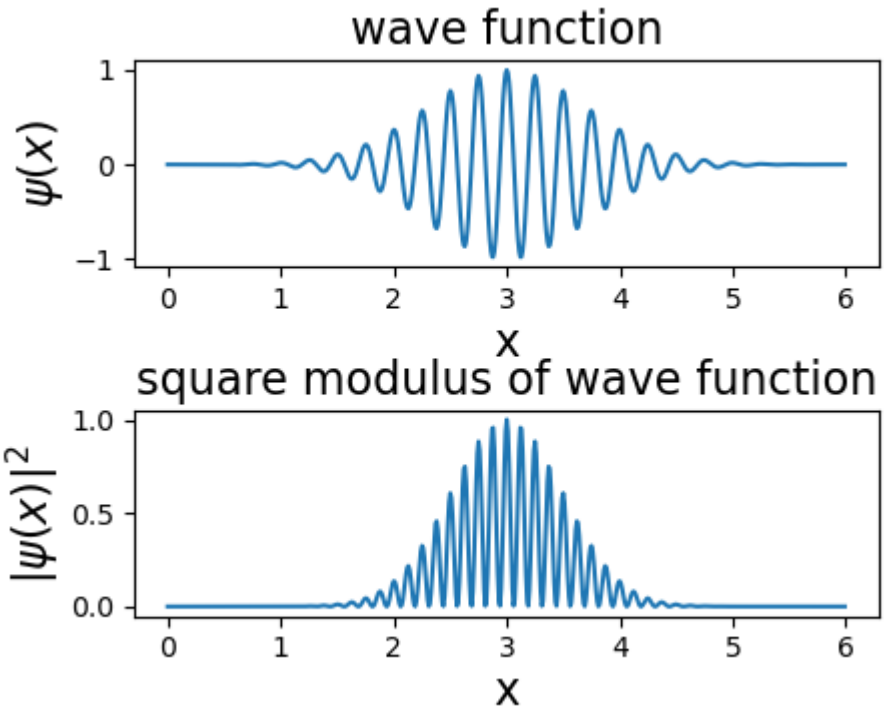
- Procediamo analogamente per `ax3`, a cui assegniamo la posizione di indice 2 in una griglia con quattro posizioni (generate da due righe e due colonne), quindi in alto a destra (posizione del pannello: `222`).

In [18]:
```
def f(t):
    return np.exp(-t) * np.cos(2*np.pi*t)
t1 = np.arange(0.0, 3.0, 0.01)

plt.figure(figsize=(6.4,4.8))

ax1 = plt.subplot(212)
ax1.margins(0.05)
ax1.plot(t1, f(t1))

ax2 = plt.subplot(221)
ax2.margins(2, 2)   # margini grandi: la figura si vede piccola
ax2.plot(t1, f(t1))
ax2.set_title('Zoomed out')

ax3 = plt.subplot(222)
ax3.margins(x=0, y=-0.25) # margini nel range (-0.5, 0.0) ingrandiscono al centro
ax3.plot(t1, f(t1))
ax3.set_title('Zoomed in')

plt.show()
```



Riporto sotto un ulteriore esempio dell'uso di `plt.plot` (non in Matplotlib). Suggerimento: studiate gli elementi nuovi ivi presenti.

In [19]:
```
import matplotlib.pyplot as plt
import numpy as np
# %matplotlib qt

def f(t):
    return np.exp(-(t-3.0)**2) * np.cos(8*np.pi*t)
def g(t):
    return np.exp(-2*(t-3.0)**2) * (np.cos(8*np.pi*t))**2
t1 = np.arange(0.0, 6.0, 0.01)

ax1 = plt.subplot(211)
ax1.margins(0.05)
ax1.plot(t1, f(t1))
ax1.set_title('wave function', fontsize=16)
plt.xlabel('x', fontsize=16, labelpad=2)
plt.ylabel('$\psi(x)$', fontsize=16, labelpad=2)

ax2 = plt.subplot(212)
ax2.margins(0.05)
ax2.plot(t1, g(t1))
ax2.set_title('square modulus of wave function', fontsize=16)
plt.xlabel('x', fontsize=16, labelpad=2)
plt.ylabel('$|\psi(x)|^2$', fontsize=16, labelpad=2)

plt.subplots_adjust(hspace=0.7)
```



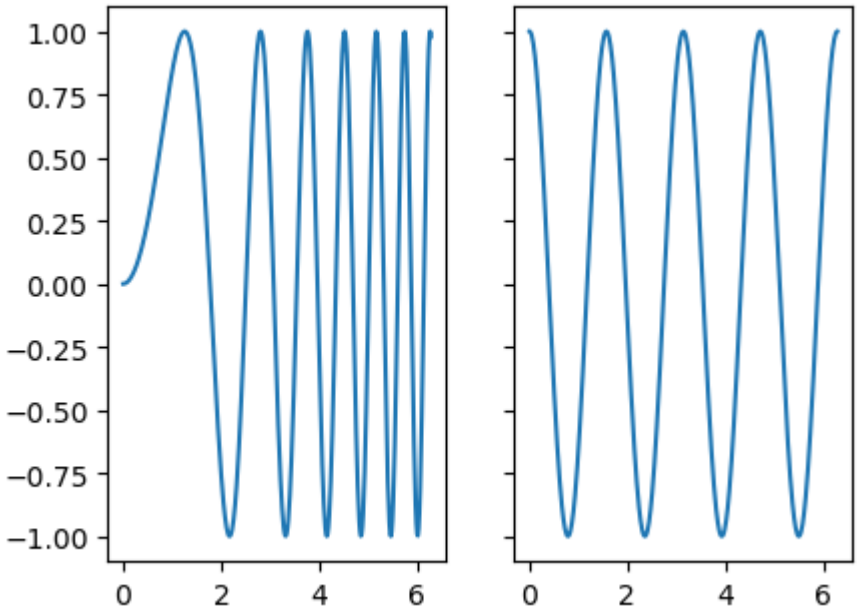
Esempio 2: uso di plt.plot con l'interfaccia OO

Uno strumento molto potente nella creazione di grafici è la funzione plt.subplots, che consente di creare assieme una figura e un insieme di "sottofigure". Infatti, come si può vedere a questa pagina web, l'output di tale funzione consiste di

- una `Figure` (di solito chiamata `fig`)
- un oggetto `Axes` (di solito denotato con `ax`) o una `array di Axes`, a seconda del numero di pannelli creati.

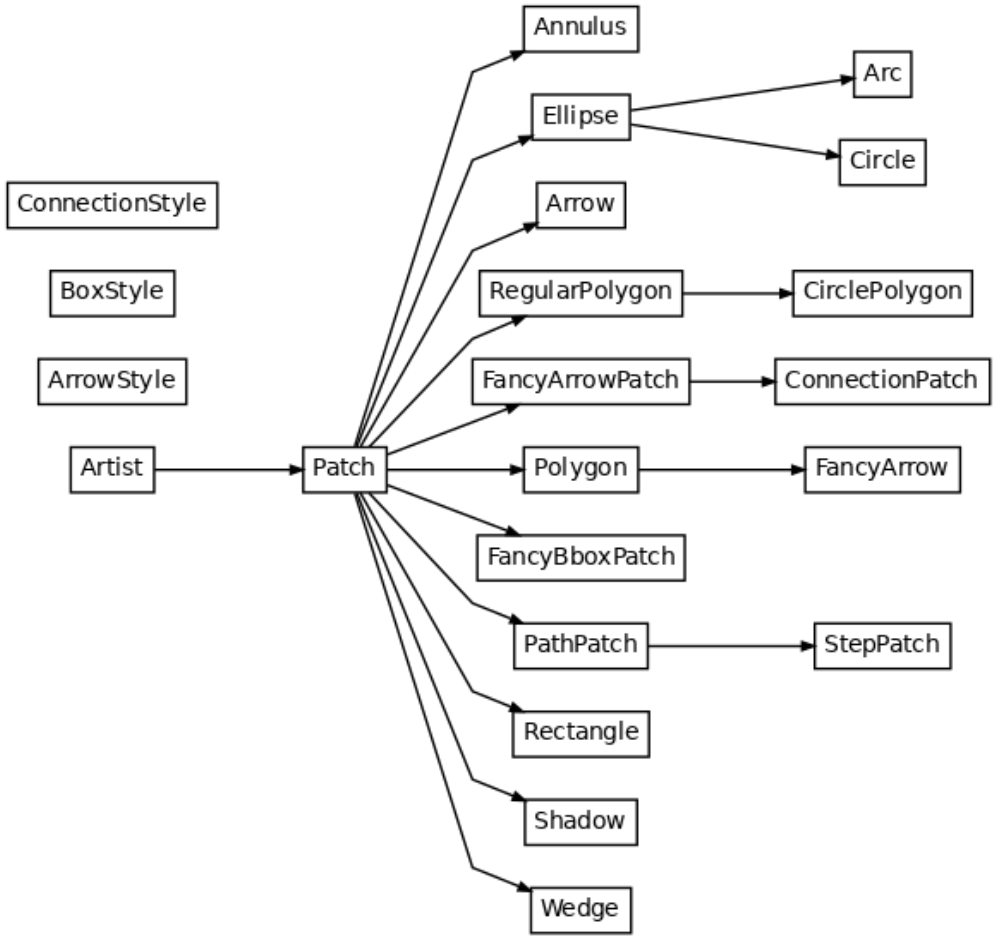
Di conseguenza, l'esito dell'applicazione della funzione, `plt.subplots(...)`, va assegnato ad una tupla come `fig, ax` o, in generale, `fig, (ax1, ax2, ..., axn)`, dove `n` è il numero di plots. Nel semplice esempio di sotto creiamo una figura con due plots.

In [20]:
```
x = np.linspace(0, 2*np.pi, 400)
y1 = np.sin(x**2)
y2 = np.cos(4*x)
fig, (ax1, ax2) = plt.subplots(1, 2, sharey=True)
ax1.plot(x, y1)
ax2.plot(x, y2)
plt.show()
```



Esempio 3: uso di patches and ipywidgets

Le classi che definiscono i *patches* sono contenute nel modulo matplotlib.patches. I *patches* disponibili sono mostrati sotto (figura da https://matplotlib.org/stable/api/patches_api.html).



Adesso vogliamo creare un set di rettangoli che approssimano una data curva, cosa che ci sarà utile nello studio degli integrali. A tal fine, useremo il *patch* matplotlib.patches.Rectangle, che ha la sintassi

`matplotlib.patches.Rectangle(xy, width, height, *, angle=0.0, rotation_point='xy', **kwargs)`

dove xy denota la tupla delle coordinate del "punto di ancoraggio" (in inglese, *anchor point*), diciamo `(x0, y0)`; `width` e `height` sono, rispettivamente, la larghezza e l'altezza del rettangolo; ecc. Tra i parametri `**kwargs` si possono definire, per esempio, il colore di riempimento dei rettangoli `facecolor`, il colore degli spigoli `edgecolor` e lo spessore di questi ultimi `linewidth`.
Per aggiungere un `Rectangle` ad un oggetto `Axes` (come, per esempio, `ax1` e `ax2` sopra), useremo il metodo add_patch.
Per semplificare la scrittura seguente, importiamo il *patch* come segue:

In [21]:
```
from matplotlib.patches import Rectangle
```

Inoltre, importiamo le ipywidgets:

In [22]:
```
import ipywidgets as widgets
```

Scegliendo una funzione Gaussiana, procediamo come segue:

In [23]:
```
def rettangoli(n):
    d = 8/float(n)
    x = np.linspace(-4,4,10**3)
    y = np.exp(-x**2/2)/np.sqrt(2*np.pi)
    fig, ax = plt.subplots()
    ax.plot(x,y)
    xr = np.ndarray(n)
    h = np.ndarray(n)
    for j in range(0,n):
        xr[j] = -4 + float(j)*d
        h[j] = np.exp(-(xr[j]+0.5*d)**2/2)/np.sqrt(2*np.pi)
    for j in range(0,n):
        ax.add_patch(Rectangle((xr[j], h[j]), d, -h[j], facecolor='none', edgecolor
    return plt.show()
```

In [24]:
```
widgets.interactive(rettangoli, n=(1,160,1))
```

Out[24]:
```
interactive(children=(IntSlider(value=80, description='n', max=160, min=1), Output
()), _dom_classes=('widget-i…
```

Esploriamo l'uso delle *widgets* ulteriormente tramite questo stesso esempio, presentando un modo diverso di interagire col grafico prodotto dal codice di sopra (altre informazioni sull'uso di widgets si possono trovare nella documentazione di Jupyter Widgets 8.1.2). Per esempio, vogliamo creare un riquadro (*box*) in cui possiamo inserire il valore di `n` da usare, con un titolo che ci indica cosa dobbiamo inserire. Inoltre, vogliamo pure creare un bottone

da pigiare per avviare la presentazione della figura.
Per raggiungere tali obiettivi, usiamo innanzitutto un *widget* (in italiano si può tradurre con *aggeggio*), widgets.Textarea, che ci consente di mostrare il valore di una stringa. La sua sintassi è

```
widgets.Textarea(value=None, **kwargs)
```

`value` è il valore da inserire nel riquadro (cioè, nel nostro caso, il valore di `n`), al quale assegniamo un valore di default, per esempio 30. `**kwargs` includono vari descrittori di dati. Tra questi, usiamo `description`, il nome o titolo del riquadro, a cui diamo il valore "n. rettangoli", e `rows`, il numero di righe da visualizzare nel riquadro: ne basta una nel nostro caso. Vi è anche un parametro `disabled` che può essere usato per disabilitare l'interazione dell'utente, per esempio in attesa che si verifichi una certa condizione. Non siamo qui interessati a fornire un argomento per tale parametro.
Un altro *widget* che ci accingiamo ad utilizzare è widgets.Button, che crea il pulsante da cliccare per avviare la presentazione della figura. Tra i parametri disponibili, usiamo solo `description`, a cui assegniamo l'argomento "run", che apparirà come il nome del pulsante.
Il widget widgets.Box ci serve per poter mostrare tutto il gruppo di widgets (in questo caso le due di sopra). I widgets sono disposti orizzontamente. La sintassi è

```
widgets.Box(children=[], **kwargs)
```

dove `children` contiene la lista di widgets da mostrare.
Infine, usiamo un widget widgets.Output, un cosiddetto gestore di contesto (*context manager*) per mostrare l'output, svuotare (*clear*) l'output (per mezzo del suo metodo `clear_output`) per poi presentarne uno nuovo e altro. In particolare, usiamo un'istruzione del tipo

```
widgets.Output(layout={'border': 'spessore_linea_in_pixel
tipo_colore_linea'})
```

per raccogliere tutto l'output in un riquadro con una linea del dato spessore e tipo.
Tutti i widgets possono essere mostrati nel notebook usando la funzione intrinseca di IPython display. Quanto detto finora, si traduce nelle righe di codice e risultato seguenti:

In [25]:
```
txta = widgets.Textarea(value="30", description='n. rettangoli', rows=1)
button = widgets.Button(description='run')
out = widgets.Output(layout={'border': '2px solid blue'})
box = widgets.Box(children=[txta, button])
display(box, out)
```

```
Box(children=(Textarea(value='30', description='n. rettangoli', rows=1), Button(desc
ription='run', style=Butto…
Output(layout=Layout(border_bottom='2px solid blue', border_left='2px solid blue', b
order_right='2px solid blu…
```

Adesso dobbiamo dare le istruzioni per mostrare la figura al click del pulsante. A tal fine usiamo le seguenti linee di codice:

In [26]:
```
def mostra_al_clic(out):
    out.clear_output()
    with out:
        print("Questa è la suddivisione dell'area sottesa in rettangoli:")
        rettangoli(int(txta.value))

button.on_click(mostra_al_clic)
```

Nella funzione definita sopra, il blocco `with` crea il contesto per indirizzare la scrittura della frase tramite `print` e l'output della funzione `rettangoli` al framework di `widgets.Output`.
Il metodo `on_click` di `button`, cioè di `widgets.Button()` è utilizzato per registrare la funzione da chiamare quando si fa clic sul pulsante.
Si noti che, anche se è necessario dotare la funzione di un parametro di input, non abbiamo bisogno di usare tale parametro, che quindi può avere un qualsiasi nome non utilizzato già per altri oggetti nella cella. A questo punto inseriamo le istruzioni nei due blocchi precedenti in un'unica cella per ottenere il risultato seguente.

In [27]:
```
txta = widgets.Textarea(value="30", description='n. rettangoli', rows=1)
button = widgets.Button(description='run')
out = widgets.Output(layout={'border': '2px solid blue'})
box = widgets.Box(children=[txta, button])
display(box, out)

def mostra_al_clic(z):
    out.clear_output();
    with out:
        print("Questa è la suddivisione dell'area sottesa in rettangoli:")
        rettangoli(int(txta.value))

button.on_click(mostra_al_clic)
```

```
Box(children=(Textarea(value='30', description='n. rettangoli', rows=1), Button(desc
ription='run', style=Butto…
Output(layout=Layout(border_bottom='2px solid blue', border_left='2px solid blue', b
order_right='2px solid blu…
```