Visualization with Matplotlib

1. Introduction
2. Simple plots - 1
3. Simple plots - 2
4. Advance plots
5. Customizing Plots
6. Multiple Subplots

Introduction

`Matplotlib` is a multiplatform data visualization library built on `NumPy` `arrays`. `Matplotlib` supports numerous backends and output types, which means we can count on it to work regardless of the operating system we are using or the output format we desire. Let's install the package first:

`Matplotlib` is a multiplatform data visualization library built on `NumPy` `arrays`. `Matplotlib` supports numerous backends and output types, which means we can count on it to work regardless of the operating system we are using or the output format we desire. Let's install the package first:

In [2]:
```
package_name = "matplotlib"
package_name2 = "ipympl"

try:
    __import__(package_name)
    print(f"{package_name} is already installed.")
except ImportError:
    print(f"{package_name} not found. Installing...")
    %pip install {package_name}

try:
    __import__(package_name2)
    print(f"{package_name2} is already installed.")
except ImportError:
    print(f"{package_name2} not found. Installing...")
    %pip install {package_name2}
```

```
matplotlib is already installed.
ipympl is already installed.
```

Creating interactive plots within a Jupyter notebook can be accomplished using the `%matplotlib` command. Additionally, we have the option to embed graphics directly in the notebook using `inline` option:

Creating interactive plots within a Jupyter notebook can be accomplished using the `%matplotlib` command. Additionally, we have the option to embed graphics directly in the notebook using `inline` option:

In [3]:
```
#Interactive backend
#%matplotlib widget
#Interactive backend
#%matplotlib ipympl
#Static backend
%matplotlib inline
```

Just as we use the `np` shorthand for `NumPy`, we will use some standard shorthands for `Matplotlib` imports:

Just as we use the `np` shorthand for `NumPy`, we will use some standard shorthands for `Matplotlib` imports:

In [4]:
```
import matplotlib as mpl
import matplotlib.pyplot as plt # a collection of functions that make matplot
import numpy as np
plt.style.use('seaborn-v0_8-whitegrid') #plt.style.use('seaborn-whitegrid')
```

Just as we use the `np` shorthand for `NumPy`, we will use some standard shorthands for `Matplotlib` imports:

In [4]:
```
import matplotlib as mpl
import matplotlib.pyplot as plt # a collection of functions that make matplot
import numpy as np
plt.style.use('seaborn-v0_8-whitegrid') #plt.style.use('seaborn-whitegrid')
```

We can choose the style we would like from the here.

Two interfaces for the `matplotlib`

A feature of `Matplotlib` that may cause confusion is its dual interfaces: a user-friendly functional-style state-based interface and a more powerful object-oriented interface.

Firstly, we create the data we would like to plot. The simplest method, `plot()` accept two `arrays` (`x` and `y`) as inputs. It will plot `y` versus `x` as lines and/or markers.

A feature of `Matplotlib` that may cause confusion is its dual interfaces: a user-friendly functional-style state-based interface and a more powerful object-oriented interface.

Firstly, we create the data we would like to plot. The simplest method, `plot()` accept two `arrays` (`x` and `y`) as inputs. It will plot `y` versus `x` as lines and/or markers.

In [5]:
```
x = np.linspace(-np.pi, np.pi, 256)
C, S = np.cos(x), np.sin(x)
```

A feature of `Matplotlib` that may cause confusion is its dual interfaces: a user-friendly functional-style state-based interface and a more powerful object-oriented interface.

Firstly, we create the data we would like to plot. The simplest method, `plot()` accept two `arrays` (`x` and `y`) as inputs. It will plot `y` versus `x` as lines and/or markers.

In [5]:
```
x = np.linspace(-np.pi, np.pi, 256)
C, S = np.cos(x), np.sin(x)
```

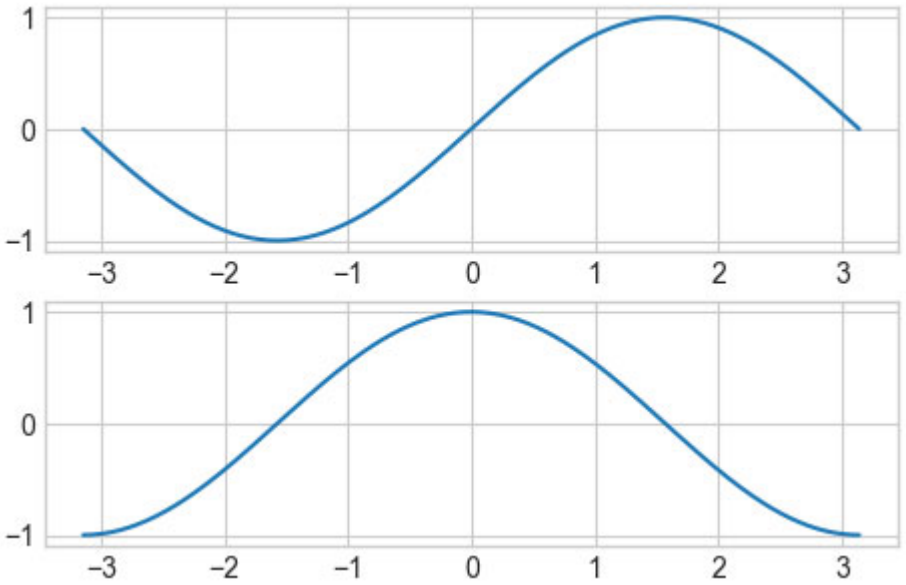
`x` is now a array with 256 values ranging from to (included). `C` is the cosine (256 values) and `S` is the sine (256 values).

Functional Interface

`Matplotlib` was initially developed as a `Python` alternative for `MATLAB` users, and many aspects of its syntax reflect this origin. The `MATLAB`-style tools can be found in the `pyplot` (`plt`) interface.

`Matplotlib` was initially developed as a `Python` alternative for `MATLAB` users, and many aspects of its syntax reflect this origin. The `MATLAB`-style tools can be found in the `pyplot` (`plt`) interface.

In [6]:
```
# 1. create a plot figure
plt.figure(figsize=(5.5, 3.5))
# 2. create the first of two panels and set current axis
plt.subplot(2, 1, 1) # (rows, columns, panel number)
plt.plot(x, S)
# 3. create the second panel and set current axis
plt.subplot(2, 1, 2)
plt.plot(x, C);  # It is stateful!
```

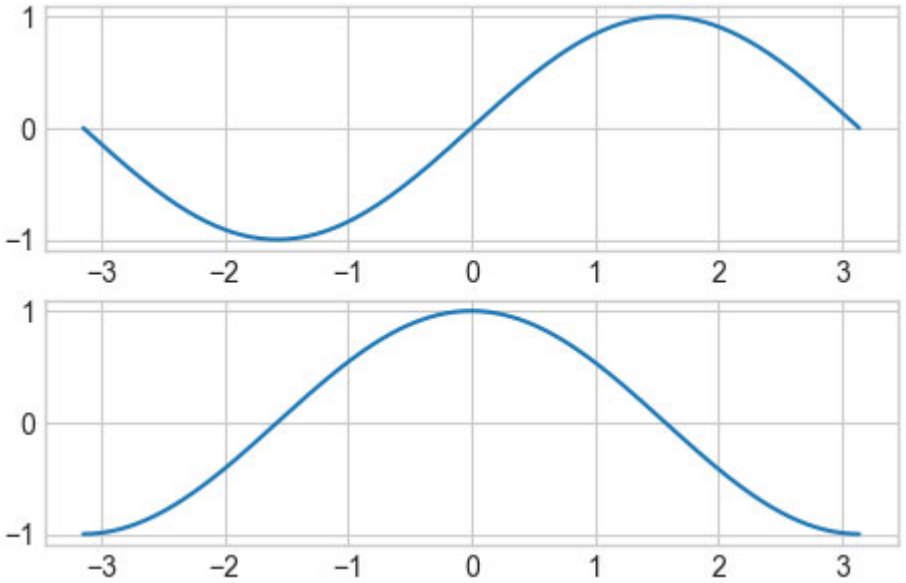


Object-oriented interface

For more complex scenarios or when greater control over the figure is desired, the object-oriented interface comes in handy. Instead of relying on the concept of an "active" figure or axes, the object-oriented interface treats plotting functions as methods of explicit `Figure` and `Axes` objects.

For more complex scenarios or when greater control over the figure is desired, the object-oriented interface comes in handy. Instead of relying on the concept of an "active" figure or axes, the object-oriented interface treats plotting functions as methods of explicit `Figure` and `Axes` objects.

In [8]:
```
# 1. First create a grid of plots
# ax will be an array of two Axes objects
fig, ax = plt.subplots(2, figsize=(5.5, 3.5))
# 2. Call plot() method on the appropriate object
ax[0].plot(x, S)
ax[1].plot(x, C);
```



In [9]:
```
display_quiz(path+"oop.json", max_width=800)
```

Which of the following are examples of the object-oriented interface in Matplotlib? (Select all that apply)

- plt.xlabel('X axis')
- ax.plot(x, y)
- ax.set_xlabel('X axis')
- plt.plot(x, y)

Simple plots - 1

Simple line plots

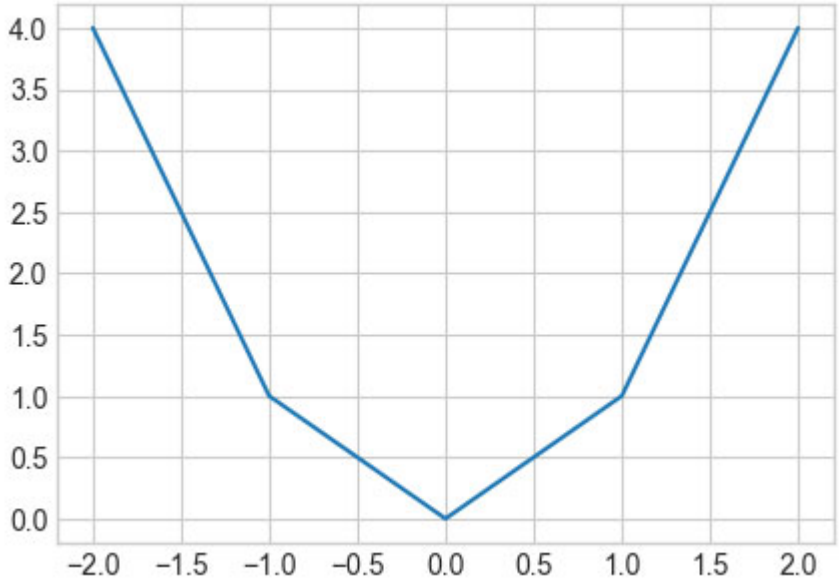
To create a 2D line plot, follow these general steps:

1. Call the `plt.figure()` to create a new figure. (optional for `%matplotlib inline`)
2. Generate a sequence of x values usually using `linspace()`.
3. Generate a sequence of y values usually by substitute the x values into a function.
4. Input `plt.plot(x, y, [format], **kwargs)` where `[format]` is an (optional) format string, and `**kwargs` are (optional) keyword arguments specifying the line properties of the plot.
5. Utilize `plt` functions to enhance the figure with features such as a title, legend, grid lines, etc.
6. Input `plt.show()` to display the resulting figure (this step is optional in a Jupyter notebook).

Let's begin with a basic example where we try plotting the parabola using 5 points:

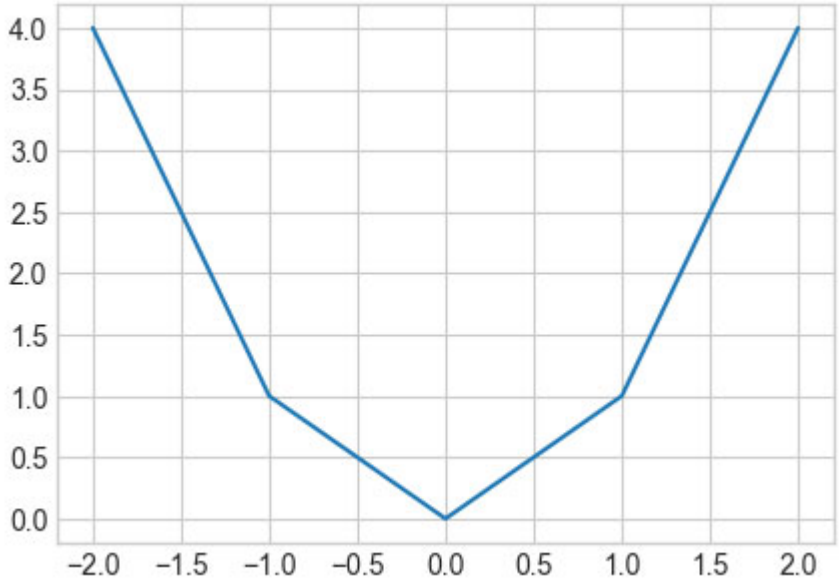
Let's begin with a basic example where we try plotting the parabola using 5 points:

In [10]:
```
plt.figure(figsize=(5, 3.5))
x = [-2,-1,0,1,2]
y = [4,1,0,1,4]

plt.plot(x,y);
```



Let's begin with a basic example where we try plotting the parabola using 5 points:

In [10]:
```
plt.figure(figsize=(5, 3.5))
x = [-2,-1,0,1,2]
y = [4,1,0,1,4]

plt.plot(x,y);
```

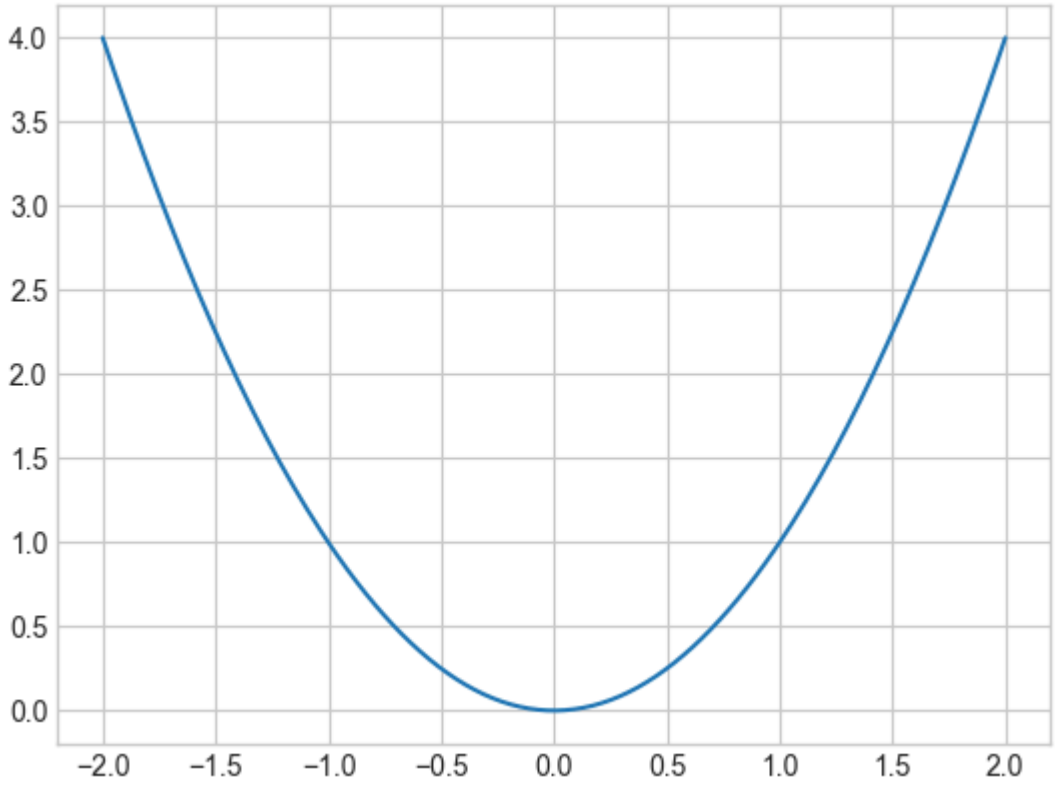


The sequences `x` and `y` determine the coordinates of the points in the plot and the line is formed by connecting these points with straight lines.

The second observation suggests that if we aim to display a smooth curve, we need to plot numerous points; otherwise, the plot will not appear smooth. Let's attempt this again, using the `NumPy` function `np.linspace()` to create 200 points:

The second observation suggests that if we aim to display a smooth curve, we need to plot numerous points; otherwise, the plot will not appear smooth. Let's attempt this again, using the `NumPy` function `np.linspace()` to create 200 points:

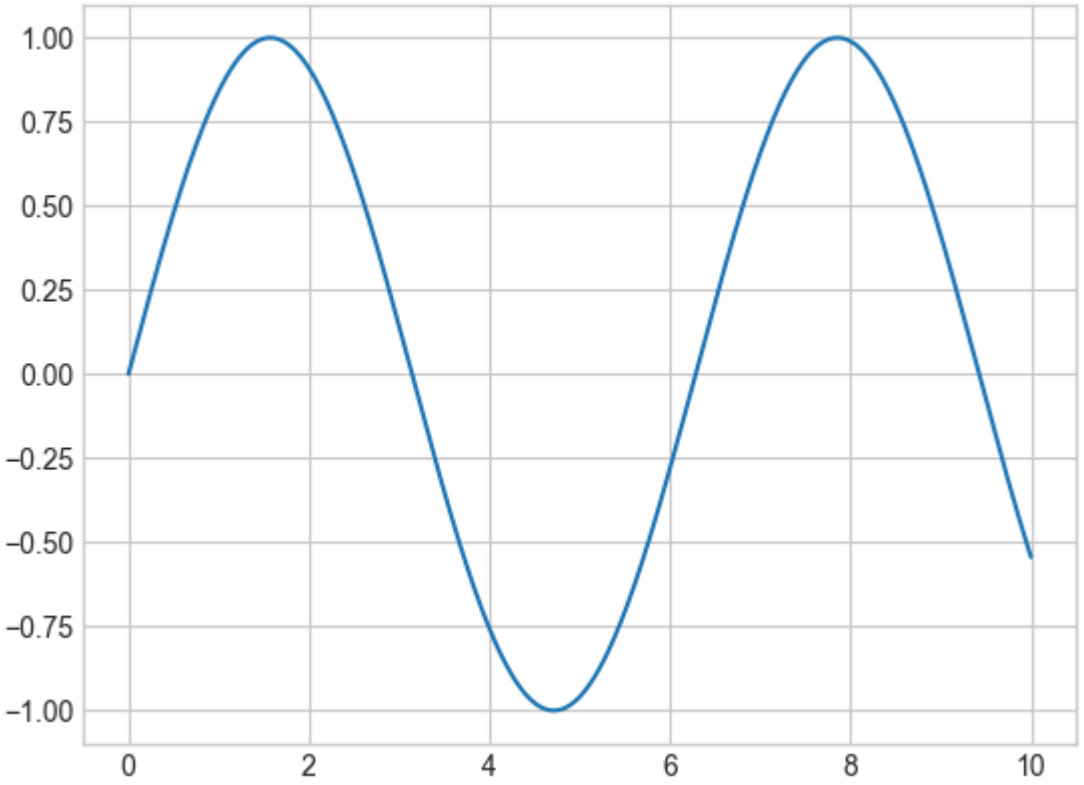
In [11]:
```
x = np.linspace(-2,2,200)
y = x**2
plt.plot(x,y);
```



Let's try another example with a simple sinusoid:

Let's try another example with a simple sinusoid:

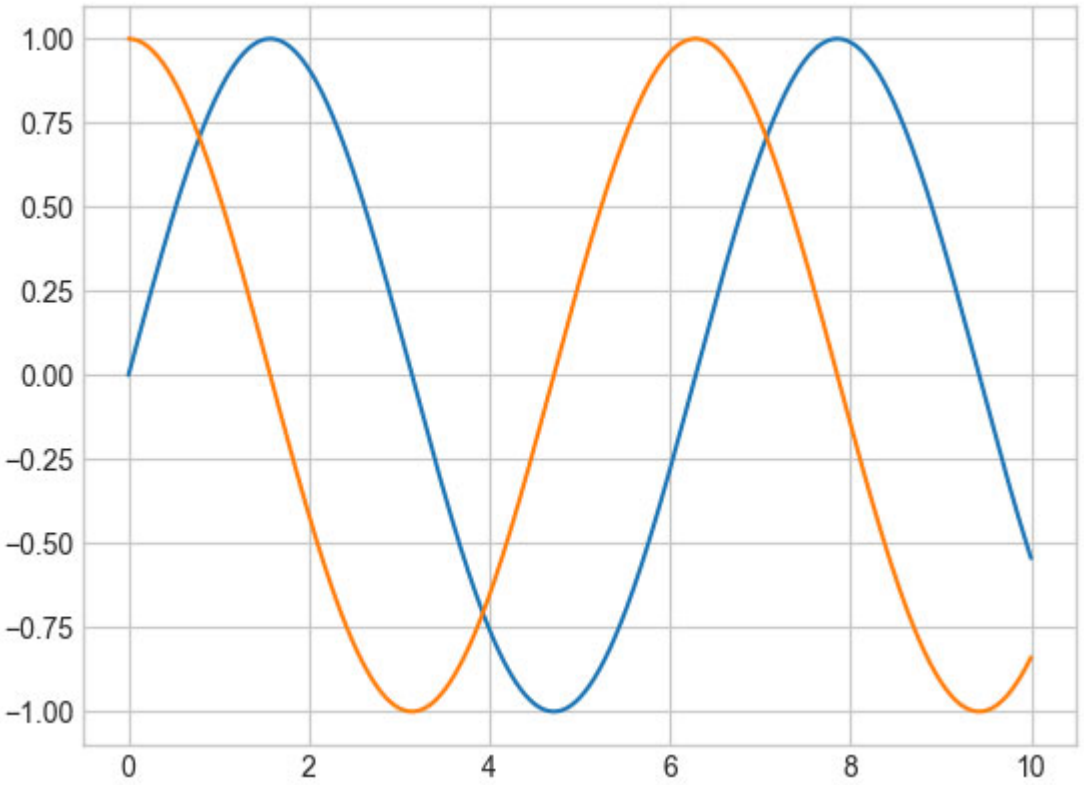
In [12]:
```
x = np.linspace(0, 10, 1000)
plt.plot(x, np.sin(x)); # let the figure and axes be created for us in the ba
```



If we want to create a single figure with multiple lines, we can simply call the `plot()` function multiple times:

If we want to create a single figure with multiple lines, we can simply call the `plot()` function multiple times:

In [13]:
```
plt.plot(x, np.sin(x))
plt.plot(x, np.cos(x));
```

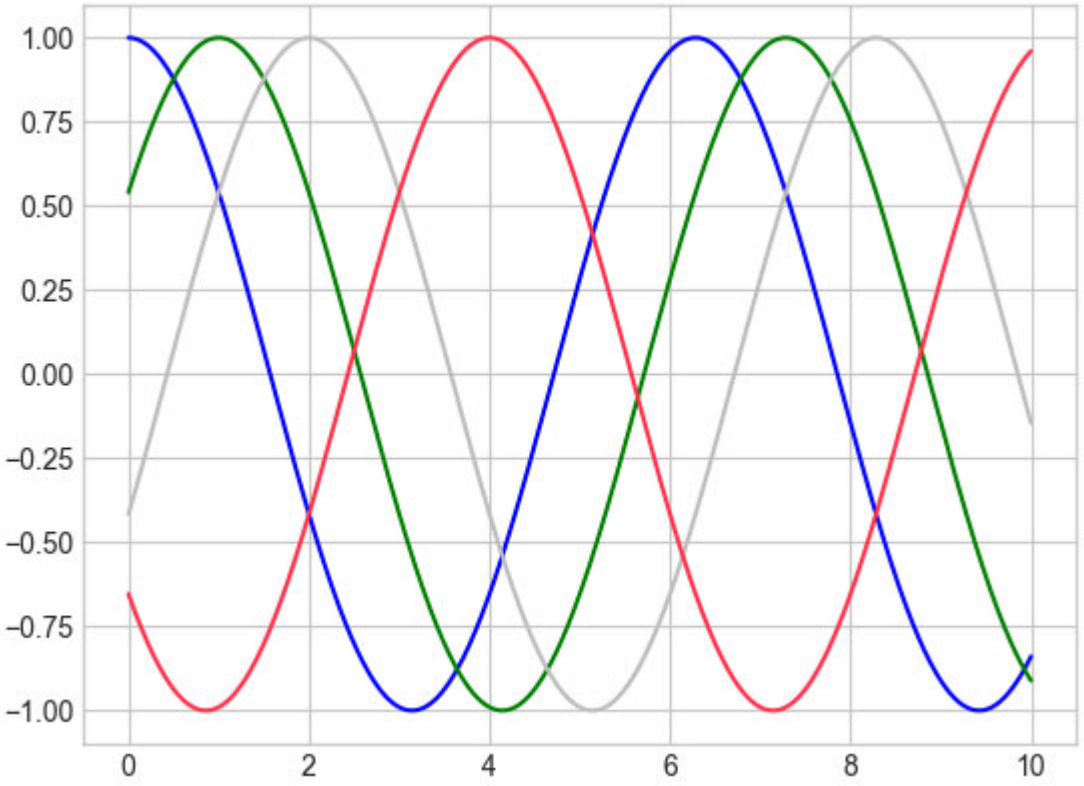


Adjusting the plot: Line colors, styles and widths

One of the first modifications you might want to make to a plot is adjusting the line colors and styles. The `plt.plot()` function accepts additional arguments that can be employed to define these aspects. To change the color, you can use the `color` keyword:

One of the first modifications you might want to make to a plot is adjusting the line colors and styles. The `plt.plot()` function accepts additional arguments that can be employed to define these aspects. To change the color, you can use the `color` keyword:

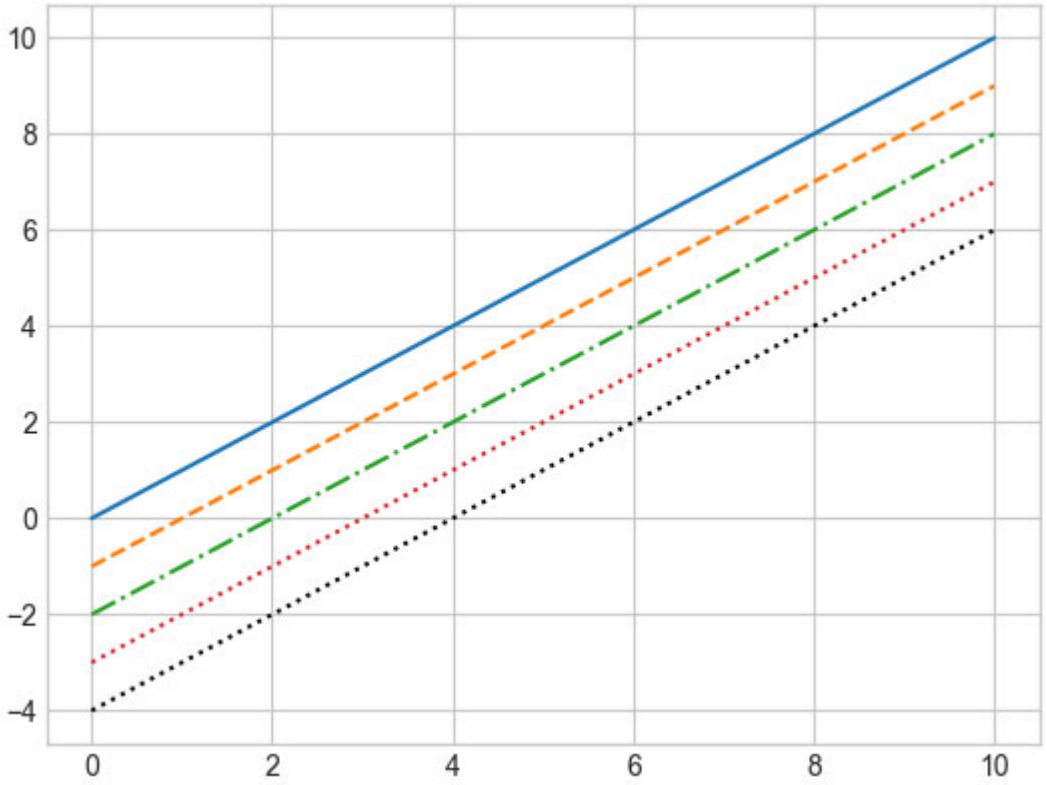
In [14]:
```
plt.plot(x, np.cos(x - 0), color='blue')          # specify color by name
plt.plot(x, np.cos(x - 1), color='g')             # short color code (rgbcmyk)
plt.plot(x, np.cos(x - 2), color='0.75')          # grayscale between 0 and 1
plt.plot(x, np.cos(x - 4), color=(1.0,0.2,0.3)); # RGB tuple, values 0 to 1
```



Similarly, the line style can be adjusted using the `linestyle` keyword:

Similarly, the line style can be adjusted using the `linestyle` keyword:

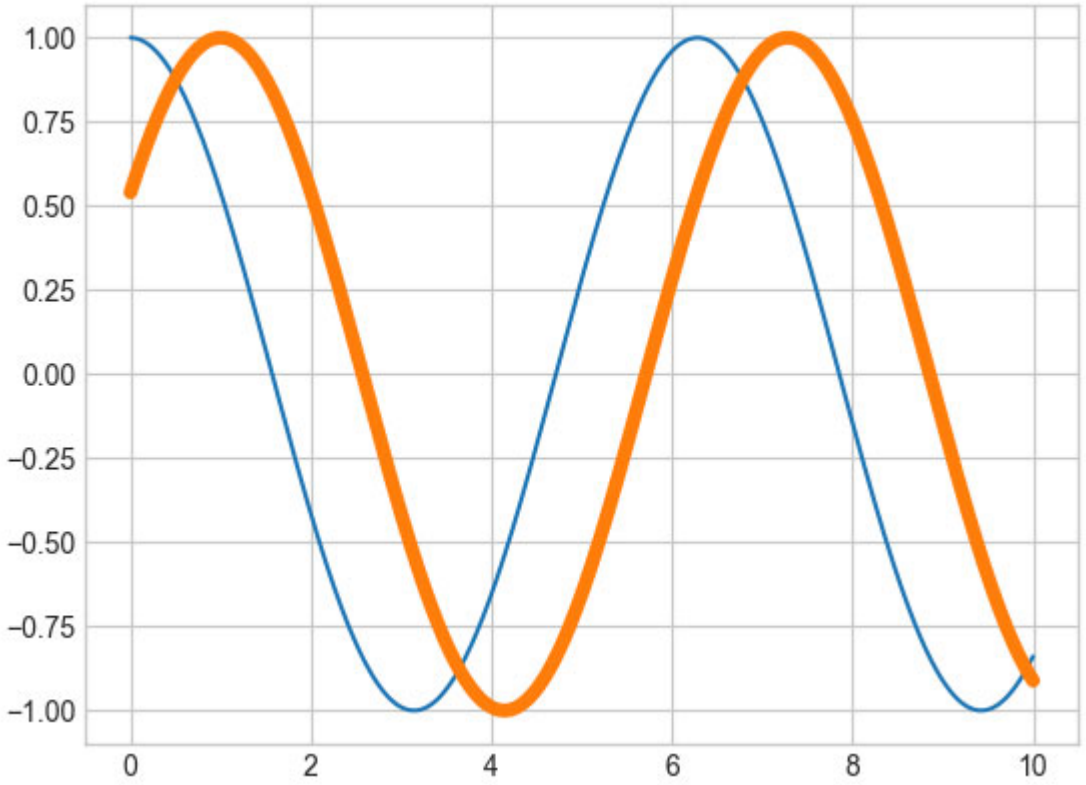
In [15]:
```
plt.plot(x, x - 0, linestyle='-')  # solid
plt.plot(x, x - 1, linestyle='--') # dashed
plt.plot(x, x - 2, linestyle='-.') # dashdot
plt.plot(x, x - 3, linestyle=':')  # dotted
plt.plot(x, x - 4, ':k');          # (use format string here!)
# You can save some keystrokes by combining these linestyle and color codes i
```



Finally, you can also adjust the width using `linewidth` keyword:

Finally, you can also adjust the width using `linewidth` keyword:

In [16]:
```
plt.plot(x, np.cos(x - 0))
plt.plot(x, np.cos(x - 1), linewidth='5');
```

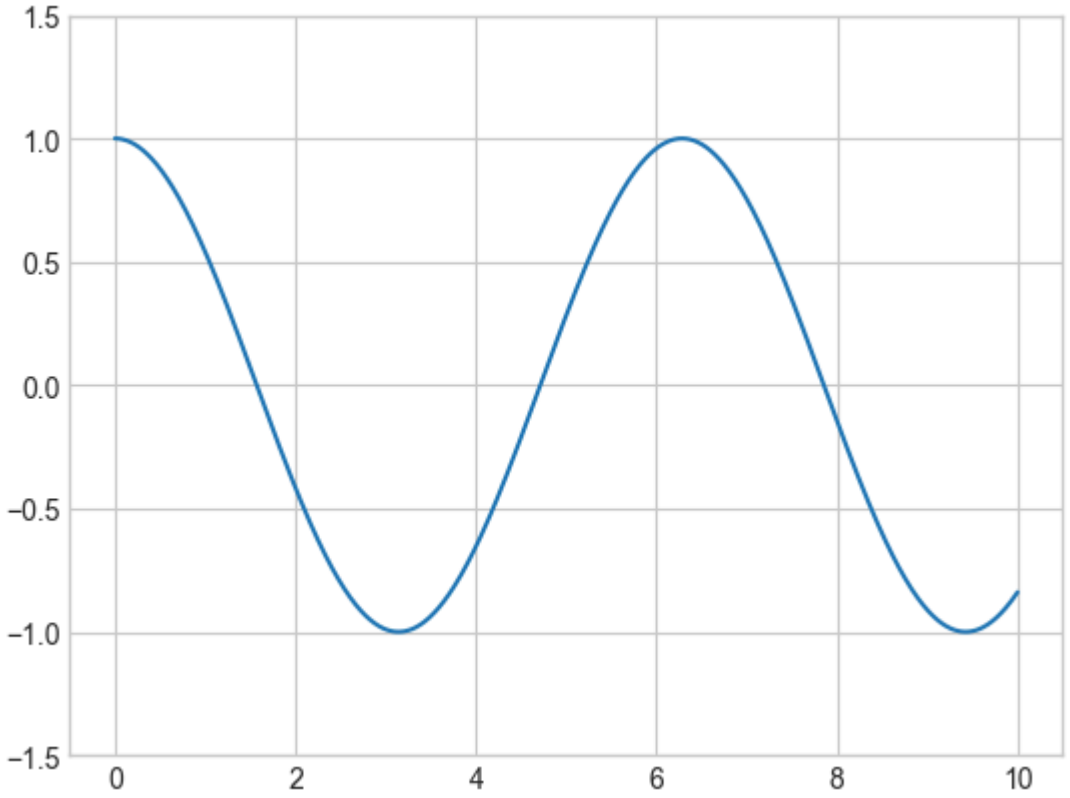


Adjusting the plot: Axes limits

`Matplotlib` generally provides suitable default axes limits for your plot, but in certain cases, having more control can be advantageous. The simplest method to fine-tune the limits is by utilizing the `plt.xlim()` and `plt.ylim()` functions:

`Matplotlib` generally provides suitable default axes limits for your plot, but in certain cases, having more control can be advantageous. The simplest method to fine-tune the limits is by utilizing the `plt.xlim()` and `plt.ylim()` functions:

In [17]:
```
plt.plot(x, np.cos(x))

plt.xlim(-0.5, 10.5)
plt.ylim(-1.5, 1.5);
```

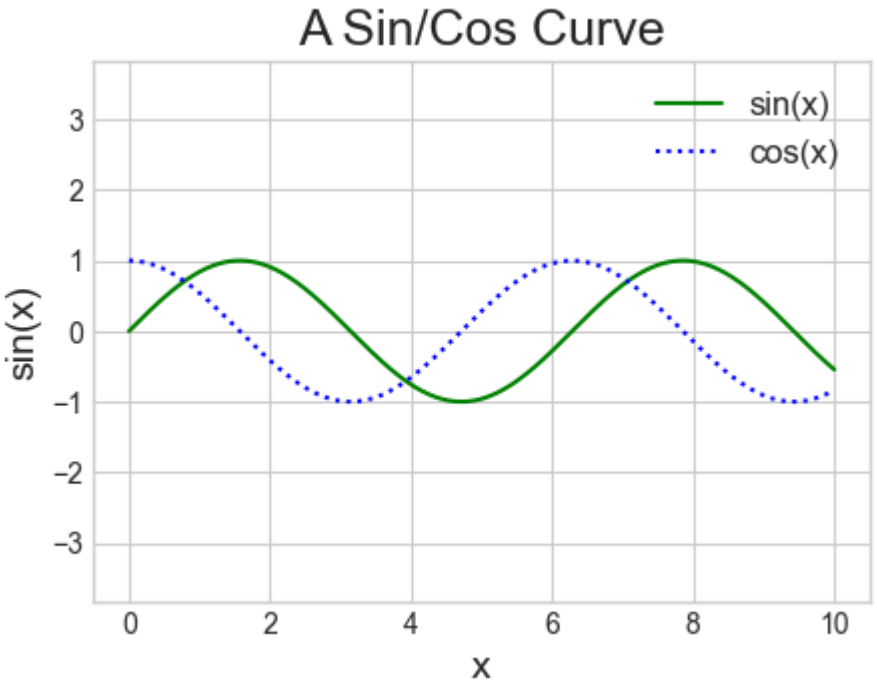


Labeling plots

Let's take a quick look at labeling plots. Titles and axis labels are the most basic types of labels — there are methods available to set them quickly.

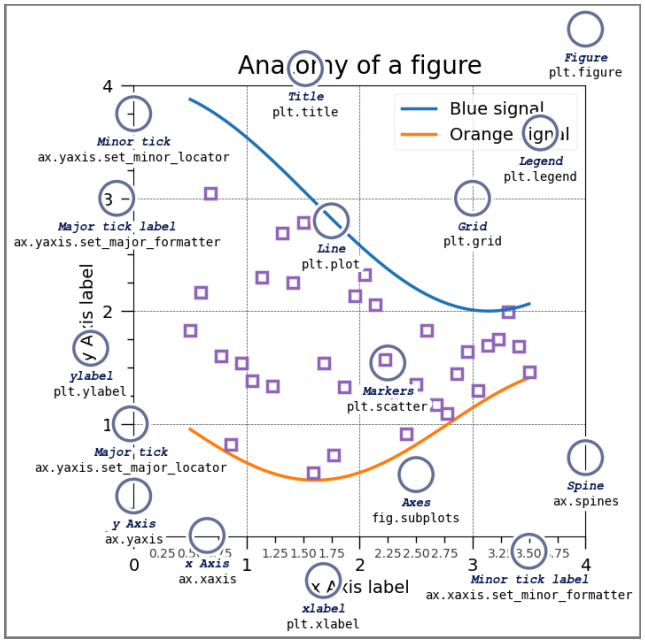
Let's take a quick look at labeling plots. Titles and axis labels are the most basic types of labels — there are methods available to set them quickly.

In [18]:
```
plt.figure(figsize=(5, 3.5))
plt.plot(x, np.sin(x), '-g', label='sin(x)') # solid green line
plt.plot(x, np.cos(x), ':b', label='cos(x)') # dotted blue line
plt.title("A Sin/Cos Curve", fontsize=18)    # we can also specify the font s
plt.xlabel("x", fontsize=14)
plt.ylabel("sin(x)", fontsize=14)
plt.legend(fontsize=12)
plt.axis('equal');
```



For more anatomy of a figure, you can refer to the following figure (which is created using the code available here):

For more anatomy of a figure, you can refer to the following figure (which is created using the code available here):



`Matplotlib` tips

While many `plt` functions (Functional interface) have direct `ax` method (OOP interface) equivalents (`plt.plot()` → `ax.plot()`, `plt.legend()` → `ax.legend()`, etc.), this does not apply to all commands. Specifically, functions for setting limits, labels, and titles undergo slight modifications. To transition between MATLAB-style functions and object-oriented methods, implement the following changes:

Functional	OOP
plt.xlabel()	ax.set_xlabel()
plt.ylabel()	ax.set_ylabel()
plt.xlim()	ax.set_xlim()
plt.ylim()	ax.set_ylim()
plt.title()	ax.set_title()

Exercise 1: Try to plot the function $\frac{1}{x(x-1)}$ within the range -2 to 3 with evenly spaced points. Try to set the point at the discontinuity to `np.nan` so that the point won't be plotted in the figure for better visualization purposes.

Hint: You can use `np.close(x, discontinuity, atol=threshold)` function to find the index of the point closest to the discontinuity. On the other hand `y[y>threshold]; y[y<-threshold]` may also be used.

In [ ]:

```
# Your code here
# ---------------------------------------------------------------
# 1. Generate an evenly-spaced grid of x-values on [-2, 3]
# ---------------------------------------------------------------
num_points = 1_000                      # number of samples to plot
x = np.linspace(____, _____, num_points)     # linearly spaced grid

# ---------------------------------------------------------------
# 2. Evaluate y = 1 / (x * (x - 1)) on that grid
# ---------------------------------------------------------------
y = _____________

# ---------------------------------------------------------------
# 3. Remove the singularities   (x = 0 and x = 1)
#    np.isclose(...) finds the grid points closest to each pole.
# ---------------------------------------------------------------
mask = np.isclose(x,_______, atol=1e-2) | \
            np.isclose(x, _______, atol=1e-2)     # boolean mask for both pol
y[mask_disc] = _______                                  # exclude poles fro

# ---------------------------------------------------------------
# 4. (Optional)  Clip extremely large magnitudes to improve
#    visual readability — anything with |y| > 1e3 is omitted.
# ---------------------------------------------------------------
threshold = 1e3
y[np.abs(y) > threshold] = np.nan
```

In [ ]:

```
plt.figure(figsize=(6, 4))
plt.plot(x, y, label=r'$y = \dfrac{1}{x\,(x-1)}$')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

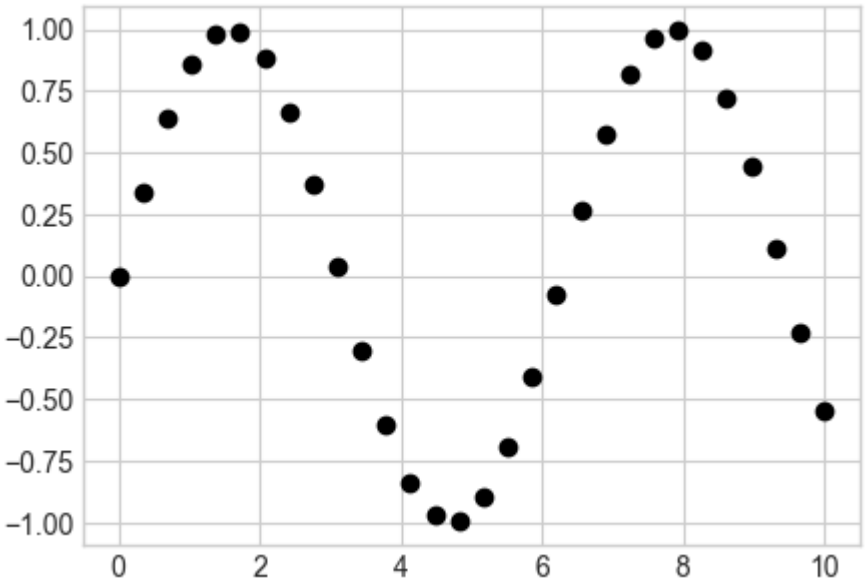
Simple plots - 2

Simple scatter plots

Another frequently used plot type is the basic scatter plot. In this case, points are depicted individually with a dot, circle, or other shape, rather than being connected by line segments. It turns out that the same function can also generate scatter plots:

Another frequently used plot type is the basic scatter plot. In this case, points are depicted individually with a dot, circle, or other shape, rather than being connected by line segments. It turns out that the same function can also generate scatter plots:

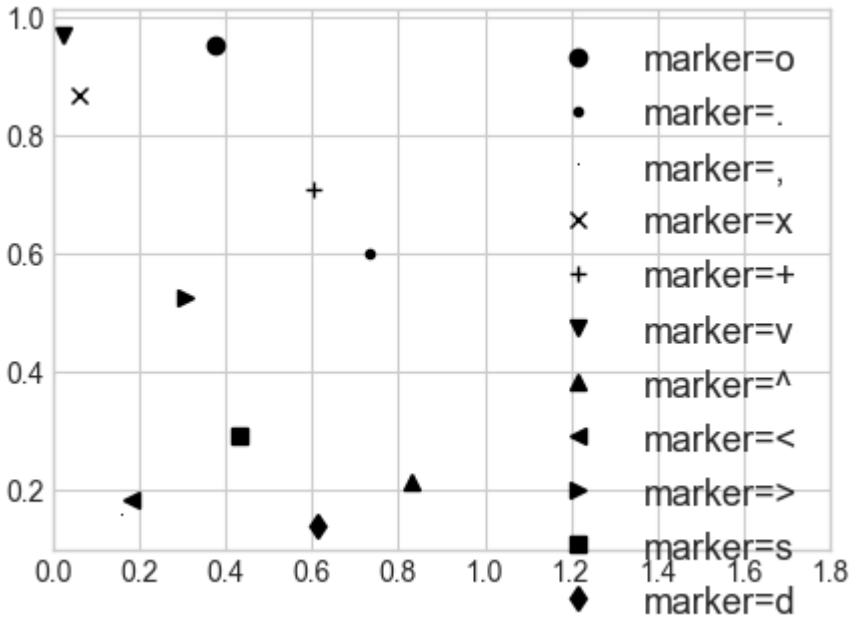
In [19]:
```
plt.figure(figsize=(5, 3.5))
x = np.linspace(0, 10, 30)
y = np.sin(x)
plt.plot(x, y, 'o', color='black');
```



The third argument in the function call is a character representing the type of symbol used for plotting. Similar to specifying options like '-' or '--' to control the line style, marker styles also have their own set of brief string codes:

The third argument in the function call is a character representing the type of symbol used for plotting. Similar to specifying options like '-' or '--' to control the line style, marker styles also have their own set of brief string codes:

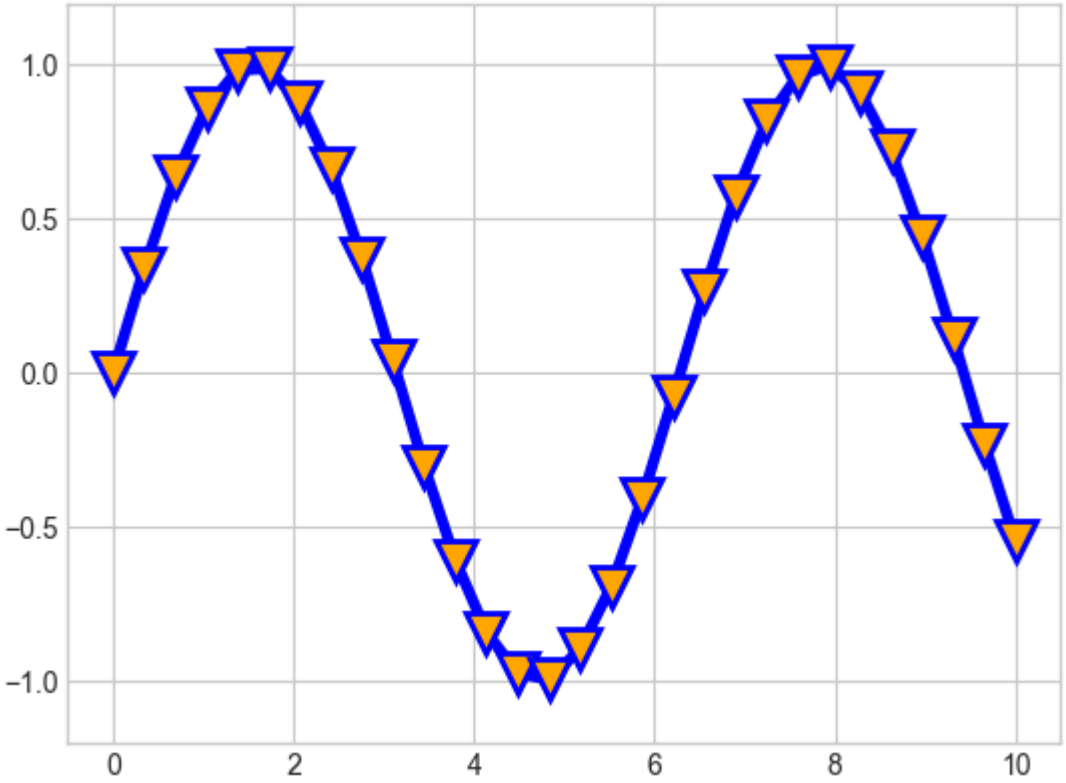
In [20]:
```
np.random.seed(42)
plt.figure(figsize=(5, 3.5))
for marker in ['o', '.', ',', 'x', '+', 'v', '^', '<', '>', 's', 'd']:
    plt.plot(np.random.random(1), np.random.random(1), marker, color='black',

plt.legend(fontsize=13)
plt.xlim(0, 1.8);
```



For even greater versatility, these character codes can be combined with line and color codes to plot points accompanied by a connecting line. Furthermore, the size or color of the markers can be customized:

For even greater versatility, these character codes can be combined with line and color codes to plot points accompanied by a connecting line. Furthermore, the size or color of the markers can be customized:

In [21]:
```
plt.plot(x, y, '-vb', markersize=15, linewidth=4, markerfacecolor='orange', m
plt.ylim(-1.2, 1.2);
```

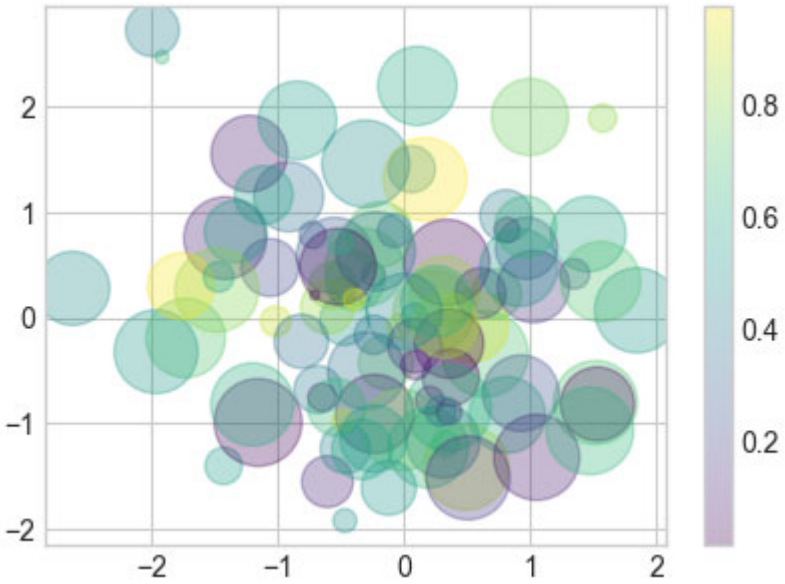


Scatter Plots with `plt.scatter()`

The main advantage of `plt.scatter()` over `plt.plot()` is its ability to generate scatter plots **where the properties of each individual point (size, face color, edge color, etc.) can be individually controlled or mapped to data**.

The main advantage of `plt.scatter()` over `plt.plot()` is its ability to generate scatter plots **where the properties of each individual point (size, face color, edge color, etc.) can be individually controlled or mapped to data**.

In [22]:
```
np.random.seed(42)
plt.figure(figsize=(5, 3.5))
x = np.random.randn(100)
y = np.random.randn(100)
colors = np.random.rand(100)
sizes = 1000 * np.random.rand(100)
plt.scatter(x, y, c=colors, s=sizes, alpha=0.3, cmap='viridis')
plt.colorbar(); # show color scale
```



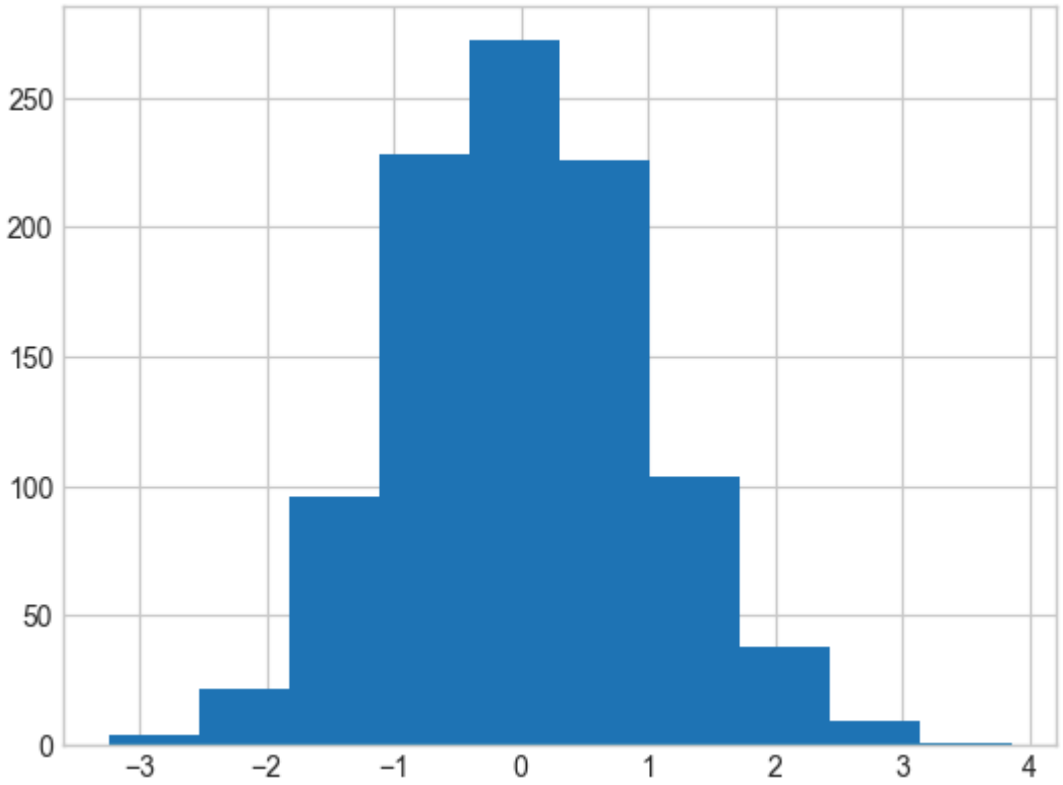
Density plots

Histograms, binnings, and density plots

A basic histogram can be an excellent initial step in comprehending a dataset. We can use `plt.hist()` to calculate and generate a histogram of sample data:

A basic histogram can be an excellent initial step in comprehending a dataset. We can use `plt.hist()` to calculate and generate a histogram of sample data:

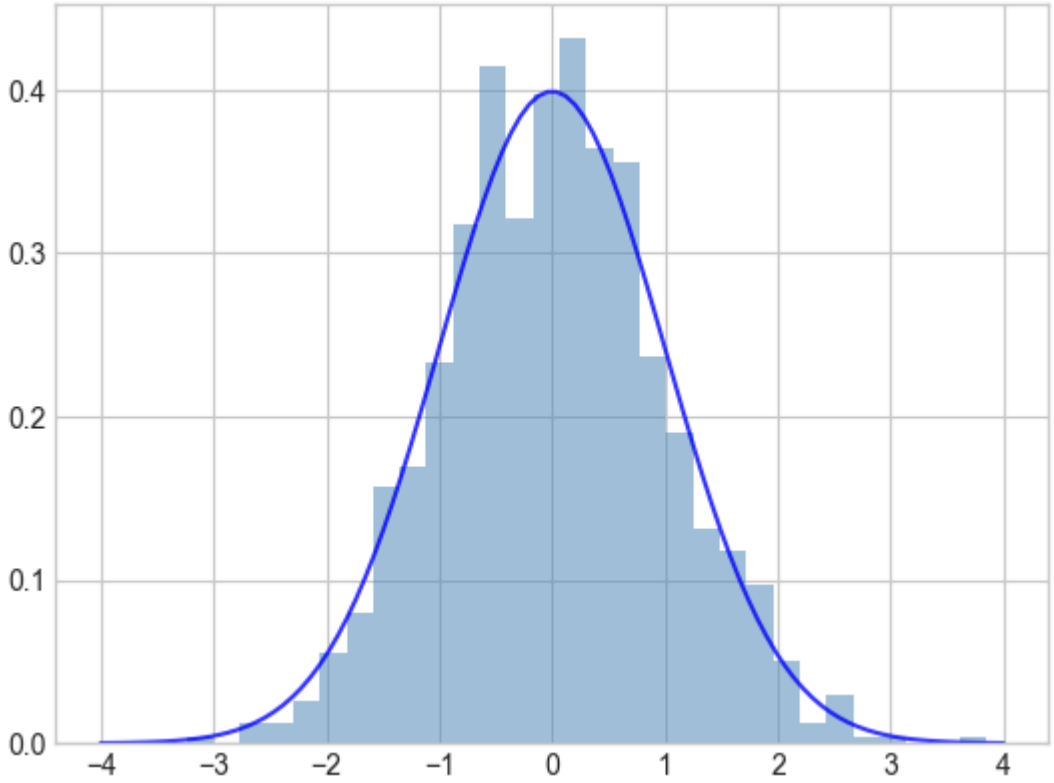
In [23]:
```
np.random.seed(42)
data = np.random.normal(size=1000)
plt.hist(data);
```



The `hist()` function provides numerous options for fine-tuning both the computation and display. Here's an example of a more customized histogram:

The `hist()` function provides numerous options for fine-tuning both the computation and display. Here's an example of a more customized histogram:

In [24]:
```
plt.hist(data, bins=30, density=True, alpha=0.5, color='steelblue', edgecolor
x = np.linspace(-4,4,100)
y = 1/(2*np.pi)**0.5 * np.exp(-x**2/2)
plt.plot(x,y,'b',alpha=0.8);
```



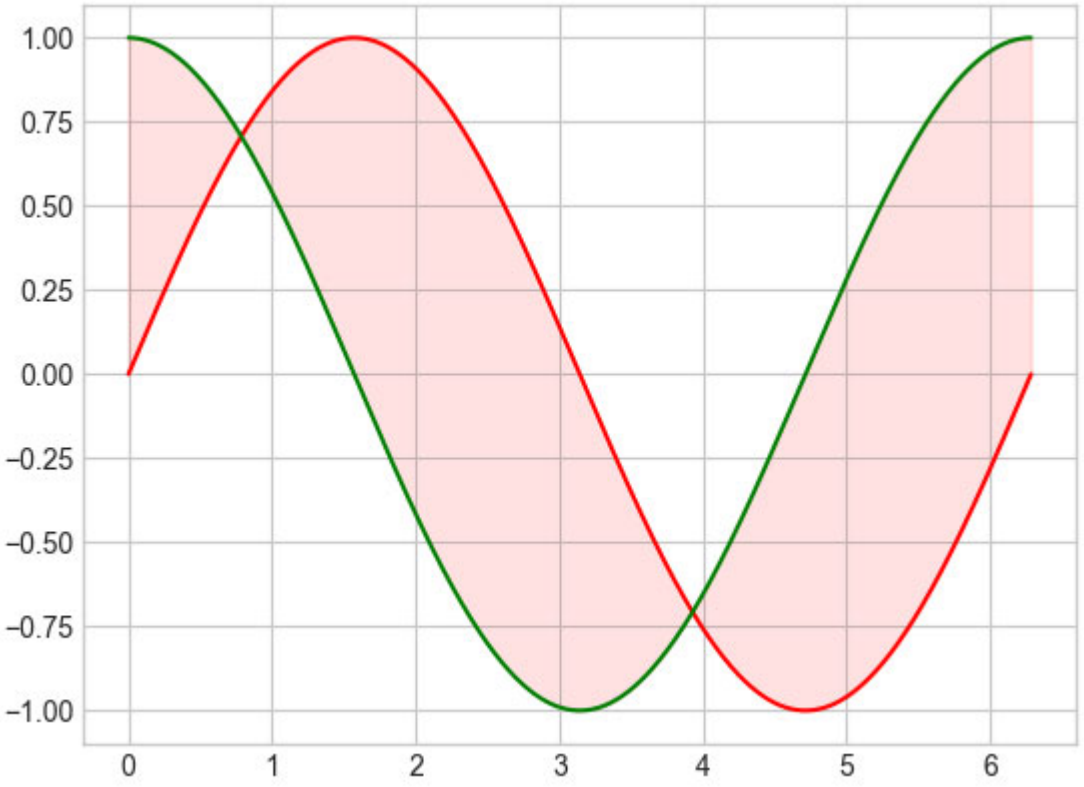
Advance plots

Filling the area between lines

Sometimes, it may be useful to fill areas between plots using `plt.fill_between()`:

Sometimes, it may be useful to fill areas between plots using `plt.fill_between()`:

In [25]:
```
x = np.linspace(0, 2*np.pi, 1000)

plt.plot(x, np.sin(x), 'r')
plt.plot(x, np.cos(x), 'g')
plt.fill_between(x, np.cos(x), np.sin(x), color='red', alpha=0.1);
```

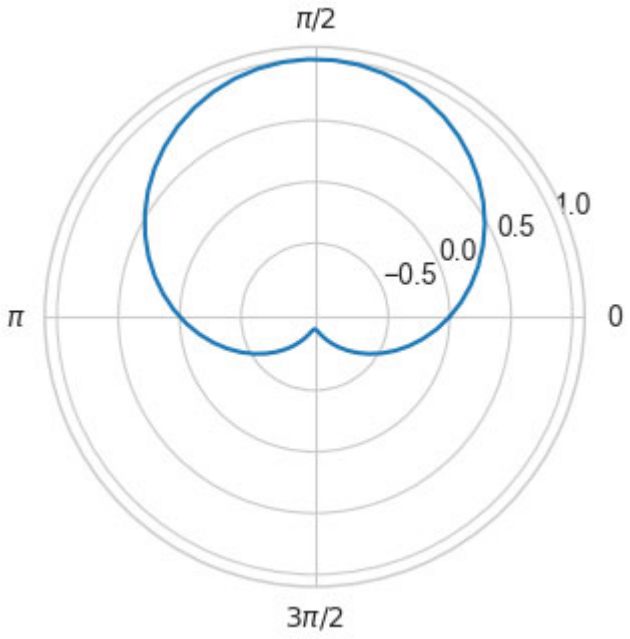


Plot in polar coordinate

To plot the figure in different coordinate system, we can use `projection` option of the `plt.axes()` method:

To plot the figure in different coordinate system, we can use `projection` option of the `plt.axes()` method:

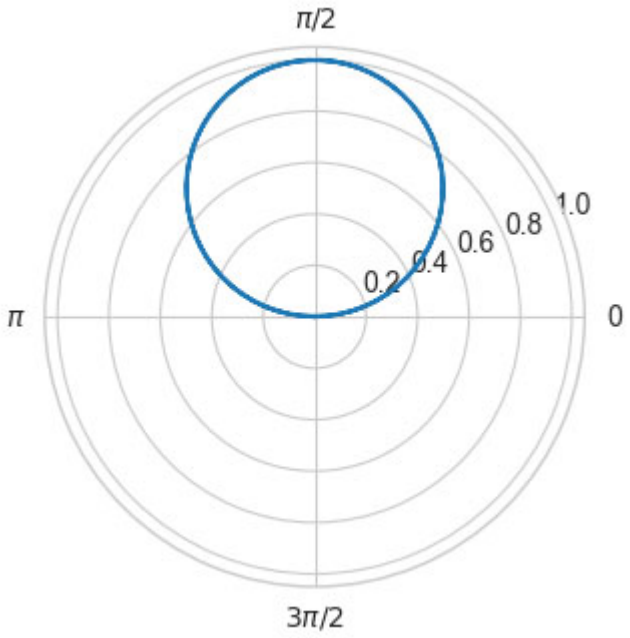
In [26]:
```
t = np.linspace(0, 2*np.pi, 64)
plt.figure(figsize=(5, 3.5))
# plot in polar coordinates
plt.axes(projection='polar')
plt.plot(t, np.sin(t), '-');
# Set ticks for polar coordinate
plt.xticks([0, np.pi/2, np.pi, 3*np.pi/2], ['0', '$\pi/2$', '$\pi$', '$3\pi/2
plt.yticks([-0.5,0,0.5,1]);
```



Note that we would expect that a radius of 0 designates the origin, and a negative radius is reflected across the origin; Specifically, the polar coordinates and should represent the same point. To implement this behavior, use the code below:

Note that we would expect that a radius of 0 designates the origin, and a negative radius is reflected across the origin; Specifically, the polar coordinates and should represent the same point. To implement this behavior, use the code below:

In [27]:
```
t = np.linspace(0, 2*np.pi, 64)
r = np.sin(t)
plt.figure(figsize=(5, 3.5))
# plot in polar coordinates
plt.axes(projection='polar')
plt.plot(t+(r<0)*np.pi, np.abs(r), '-')
# Set ticks for polar coordinate
plt.xticks([0, np.pi/2, np.pi, 3*np.pi/2], ['0', '$\pi/2$', '$\pi$', '$3\pi/2
```



Exercise 2: Try to plot the $\sin(2x)$ function in the range $x = [-\pi, \pi]$ and fill the area between the curve and the x-axis with the color `blue` and `alpha=0.25` as follows

You can use the following code to set the ticks:

```
radian_multiples = [-1, -1/2, 0, 1/2, 1]
radians = [n * np.pi for n in radian_multiples]
radian_labels = ['$\pi$', '$-\pi/2$', '0', '$\pi/2$', '$\pi$']
plt.xticks(radians, radian_labels);
```

In [ ]:

```
# Your code here
# ----------------------------------------------------------------
# 1. Generate x-values from -π to π
# ----------------------------------------------------------------
x = np.linspace(-np.pi, np.pi, 1_000)      # dense grid for smooth curve

# ----------------------------------------------------------------
# 2. Evaluate y = sin(2x)
# ----------------------------------------------------------------
y = _________

# ----------------------------------------------------------------
# 3. Plot the curve and shade the area between y and the x-axis
# ----------------------------------------------------------------
plt.figure(figsize=(6, 4))
plt.plot(___,____)                        # curve
plt.fill_between(____, ____, _____, color='blue', alpha=0.25)  # shaded regio
```

In [ ]:
```
# Your code here
# ---------------------------------------------------------------------
# 1. Generate x-values from -π to π
# ---------------------------------------------------------------------
x = np.linspace(-np.pi, np.pi, 1_000)      # dense grid for smooth curve

# ---------------------------------------------------------------------
# 2. Evaluate y = sin(2x)
# ---------------------------------------------------------------------
y = _________

# ---------------------------------------------------------------------
# 3. Plot the curve and shade the area between y and the x-axis
# ---------------------------------------------------------------------
plt.figure(figsize=(6, 4))
plt.plot(___,____)                      # curve
plt.fill_between(____, ____, _____, color='blue', alpha=0.25)  # shaded regio
```

In [ ]:
```
radian_multiples = [-1, -1/2, 0, 1/2, 1]
radians = [n * np.pi for n in radian_multiples]
radian_labels = ['$-\\pi$', '$-\\pi/2$', '0', '$\\pi/2$', '$\\pi$']
plt.xticks(radians, radian_labels)

plt.xlabel('x')
plt.ylabel('sin(2x)')
plt.title(r'$y = \sin(2x)$ on $[-\pi, \pi]$')
plt.axhline(0, color='black', linewidth=0.8)  # x-axis
plt.grid(True)
plt.tight_layout()
plt.show()
```

Multiple Subplots

Sometimes, it's helpful to look at different pieces of data next to each other. To do this, `Matplotlib` uses something called **subplots**. Subplots are basically smaller graphs that can live together in one bigger graph. These smaller graphs could be little graphs placed inside a larger one, a grid of many graphs, or they could be arranged in other more complicated ways.

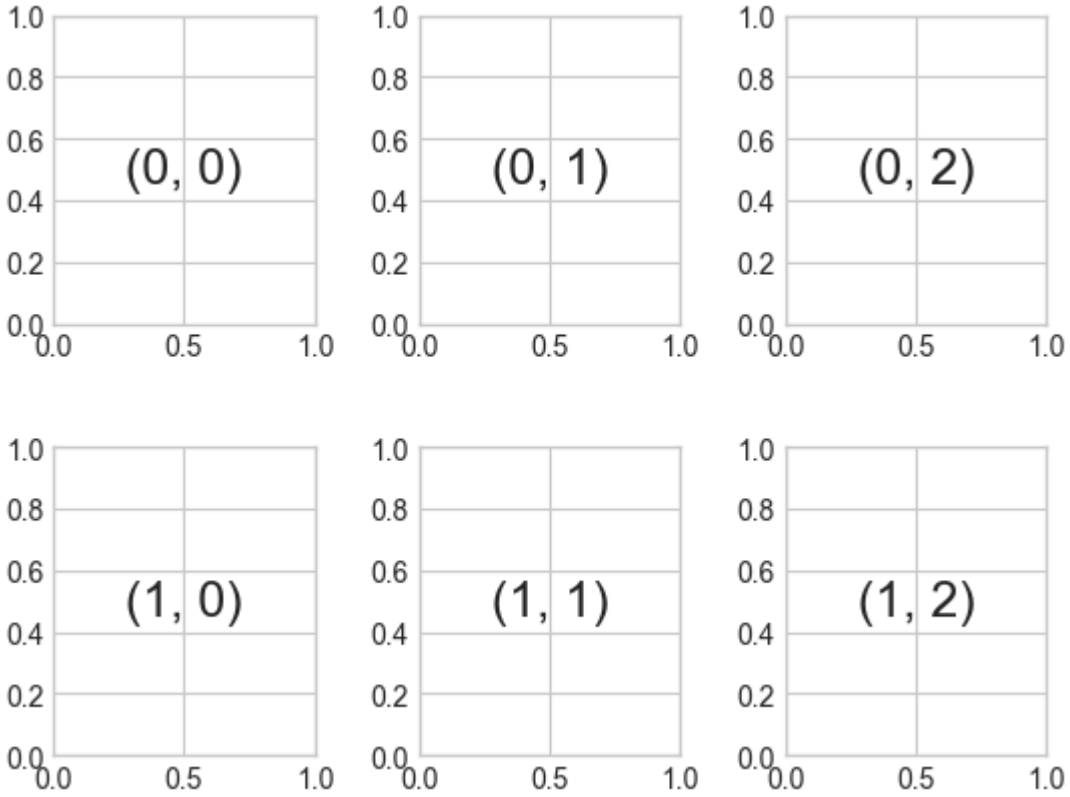
```
plt.subplots()
```

Aligned rows or columns of subplots are a common enough requirement that `Matplotlib` has several convenience routines that make it easy to create them. `plt.subplots()` is the easiest tool to use. Instead of creating a single subplot, **this function creates a complete grid of subplots in one line, and returns them as a `NumPy` array**. The arguments are the number of rows and the number of columns.

Let’s create a grid of subplots, and adjust the spacing between them:

Let's create a grid of subplots, and adjust the spacing between them:

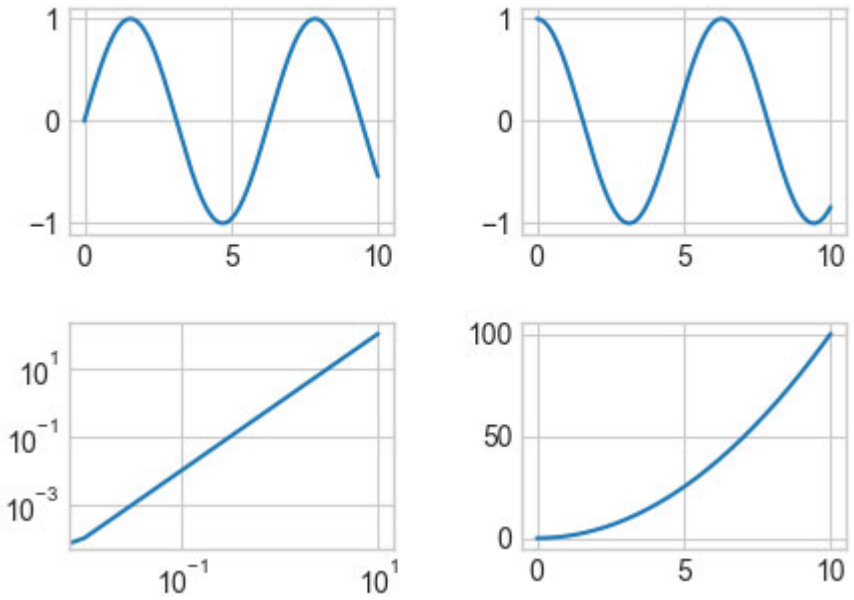
In [32]:
```
fig, ax = plt.subplots(2, 3)
fig.subplots_adjust(hspace=0.4, wspace=0.4)
for i in range(2):
    for j in range(3):
        ax[i, j].text(0.5, 0.5, str((i, j)), fontsize=18, ha='center', va='ce
```



The command `plt.subplots_adjust()` can be used to adjust the spacing between subplots. We can then use the subplots to plot different figures:

The command `plt.subplots_adjust()` can be used to adjust the spacing between subplots. We can then use the subplots to plot different figures:

In [33]:
```
fig, ax = plt.subplots(2, 2, figsize=(5, 3.5))
fig.subplots_adjust(hspace=0.4, wspace=0.4)
x = np.linspace(0, 10, 1000)
ax[0,0].plot(x, np.sin(x))
ax[0,1].plot(x, np.cos(x))
ax[1,0].plot(x, x**2)
ax[1,0].set_xscale('log') # Set the scale to log scale
ax[1,0].set_yscale('log')
ax[1,1].plot(x, x**2);
```



In summary, `Matplotlib` is a data visualization library for creating visualizations in `Python`. It provides a wide variety of customizable plots, charts, and graphs, making it a powerful tool for data analysis and communication. With `Matplotlib`, we can create line plots, scatter plots, histograms, and many other types of visualizations. You can customize the appearance of your plots with a wide range of options, including color schemes, fonts, axes labels, and annotations. Refer to https://matplotlib.org/cheatsheets/ for more details.

In [34]:
```
from jupytercards import display_flashcards
fpath= "https://raw.githubusercontent.com/phonchi/nsysu-math106A/refs/heads/m
display_flashcards(fpath + 'ch10.json')
```

Functional-style Interface

Next

>

In [ ]: