Lecture 5: Visualization

This Lecture will introduce the module `matplotlib`

We will use `matplotlib` for visualization of data

Data Visualization

- The `matplotlib` **module** is used the basic module for data visualization using Python.
- Other more sophisticated (prettier!) modules have been developed,
    - `seaborn`, **https://seaborn.pydata.org/** (statistical visualization)
    - `plotly`, **https://plotly.com/python/** (interactive and 3D)

but these are always built on top of `matplotlib` and consistent with its framework

- In fact, the seaborn documentation explicitly recommends you understand how `matplotlib` works to get the most out of seaborn.

`pyplot` submodule

- The vast majority of the data visualization make use of a single submodule called `pyplot`

In [23]:

```python
# The line below imports the pyplot submodule of matplotlib
# and gives it the short name plt
from matplotlib import pyplot as plt
### And as always, I'm going to import numpy
import numpy as np
```

In [24]:

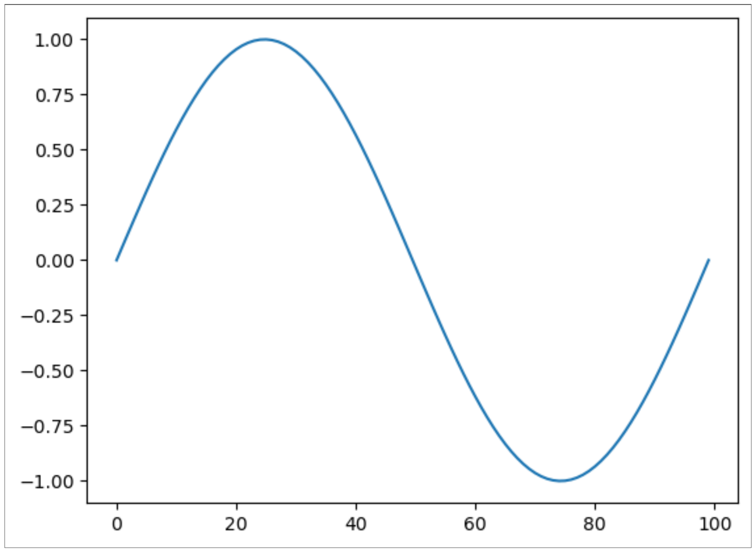
```python
# For the sake of todays exercise, I am going to make a sine and a cosine function
# If you pull up the documentation on numpy, you will find the functions and figure out how to call them.
angle = np.linspace(0,2*np.pi,100) #I make 100 evenly spaced values going from 0 to 2*pi
C = np.cos(angle) #cosine function
S = np.sin(angle) #sine function
```

Simple Line Plots

- Perhaps not so surprisingly, the simplest command to make a plot in `pyplot` is called `plot`.

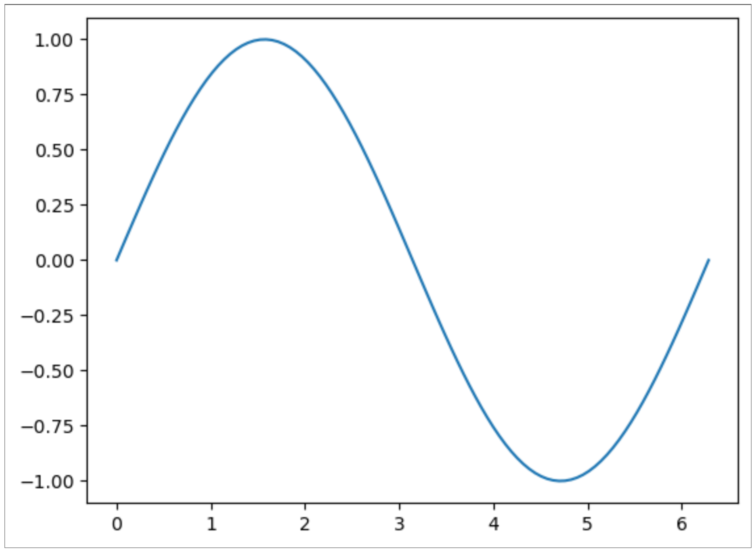
In [25]:

```
plt.plot(S) #This plots the array S and implicitly makes the x axis the index into the array
plt.show() #this makes a clean display of the plot without any code.
```



In [26]:

```
plt.plot(angle,S) # when making plot, the two inputs are x and y.
plt.show() # this makes a clean display of the plot without any code.
```



- `plot` is the simplest, easiest way to invoke a plot in python. And when you just want a quick plot, this is the way to do it. I do it all the time!
- However, when you want to make a high quality plot, you want to make use of a different **"modern"** interface to pyplot, which is how the documentation is written.
- Today's lesson will make use of that interface which is often referred to as the **object oriented** interface.

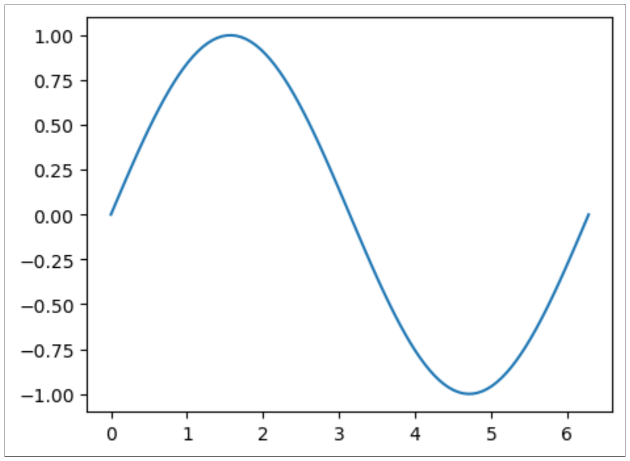
Object-oriented Interface to `pyplot`

1. We create a figure *object* named **fig**. This object has properties which control the figure as a whole
2. We create an axes *object* named **ax**. This object has properties which control the axis of a specific plot.
3. We add a plot to the axes **ax**

- When we use `plot` we are in fact invoking a figure and an axes, then adding a plot. But, by using `plot` we are making use of the default values.
- Our objective here is to make pretty plots. Because the scientist who makes the prettiest plot gets **$$$**.

In [27]:

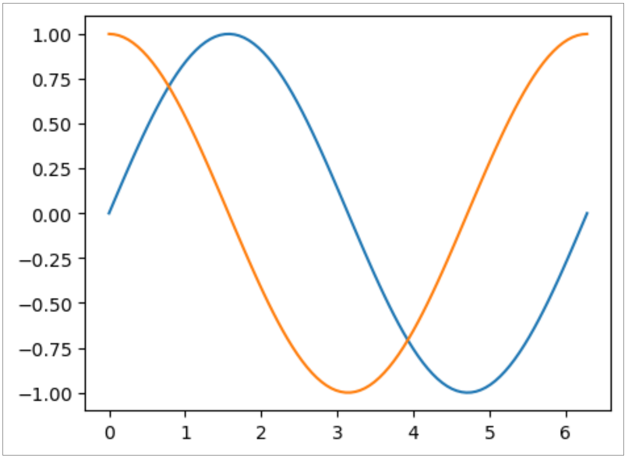
```
#object-oriented interface
fig = plt.figure(figsize = (4,3))   #creates a blank canvas of size (4,3)
#The two size dimension work as width, height
ax = fig.add_axes([0,0,1,1]) #creates an axes
# (starting pt x, starting pt y, fractional size x, fractional size y)
ax.plot(angle,S) #add a plot to the axes.
plt.show()
```



- We can add a second line to the same plot, by calling plot again.

```
In [28]:
```

```
fig = plt.figure(figsize = (4,3))    #creates a blank canvas
ax = fig.add_axes([0,0,1,1]) #creates an axes
# - (starting pt x, starting pt y, fractional size x, fractional size y)
ax.plot(angle,S) #add a plot to the axes.
ax.plot(angle,C) #add a plot to the axes.
plt.show()
```



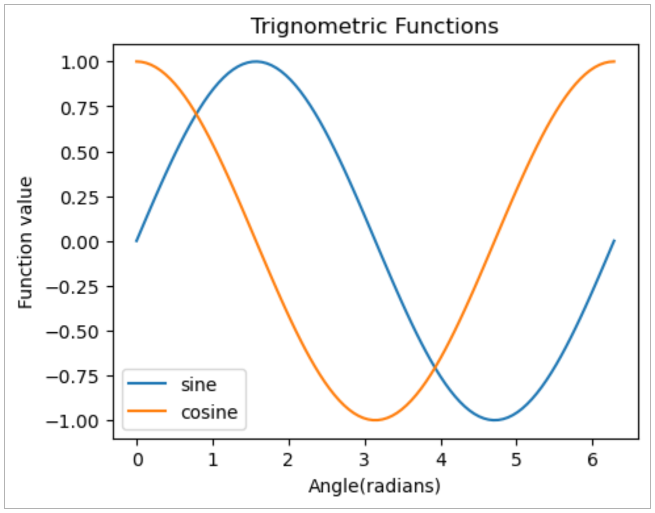
Labeling the plot.

- What's wrong with that plot? Its missing information to be able to interpret the plot.

1. Axes labels - What is the x axis, what is the y axis?
2. Line labels - which one is sine, and which one is cosine?
3. Title (optional) - maybe I need a title.

In [29]:

```
fig = plt.figure(figsize = (4,3))    #creates a blank canvas
ax = fig.add_axes([0,0,1,1]) #creates an axes
# - (starting pt x, starting pt y, fractional size x, fractional size y)
ax.plot(angle,S,label ='sine')  #add a plot to the axes.
                                #I also gave the line a label.
ax.plot(angle,C,label = 'cosine' ) #add a plot to the axes.
                                #I also gave the line a label.
ax.set_xlabel('Angle(radians)') #I added a x axis label
ax.set_ylabel('Function value') #I added a y axis label
ax.legend() #I gave the figure a legend.
ax.set_title('Trignometric Functions') # Let's give the figure a title.
plt.show()
```



Improving the communication of plots

- One might argue that we are done now, and all necessary information to understand the plot is given, but it still *sucks*.
- **Why?**

1. Units
2. Readability
3. Aesthetics

Units should mean something to people

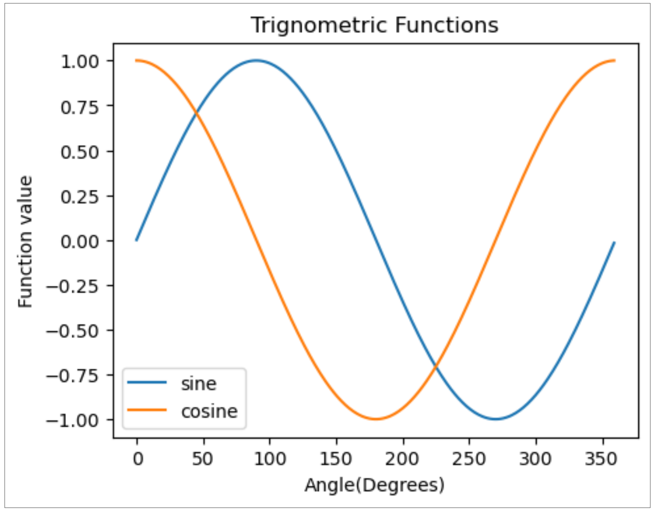
- Here I make 2 equivalent version of the angle variables, one in degrees and the other in radians.
- I will use the version in *radians* to **compute** sine and cosine, but i will use the one in *degrees* to make the **plot**.

In [30]:

```
angle_in_degrees = np.arange(0,360,1) # note here that because i used arange
                        #i do not include 360. I made step size 1 degree.
angle = angle_in_degrees*np.pi/180 # since 180 degrees is  equal to pi
C = np.cos(angle) #cosine function
S = np.sin(angle) #sine function
```

In [31]:

```
fig = plt.figure(figsize = (4,3))   #creates a blank canvas
ax = fig.add_axes([0,0,1,1]) #creates an axes
# - (starting pt x, starting pt y, fractional size x, fractional size y)
ax.plot(angle_in_degrees,S,label ='sine')  #add a plot to the axes.
                                #I also gave the line a label.
ax.plot(angle_in_degrees,C,label = 'cosine' ) #add a plot to the axes.
                                #I also gave the line a label.
ax.set_xlabel('Angle(Degrees)') #I added a x axis label
ax.set_ylabel('Function value') #I added a y axis label
ax.legend() #I gave the figure a legend.
ax.set_title('Trignometric Functions') # Let's give the figure a title.
plt.show()
```

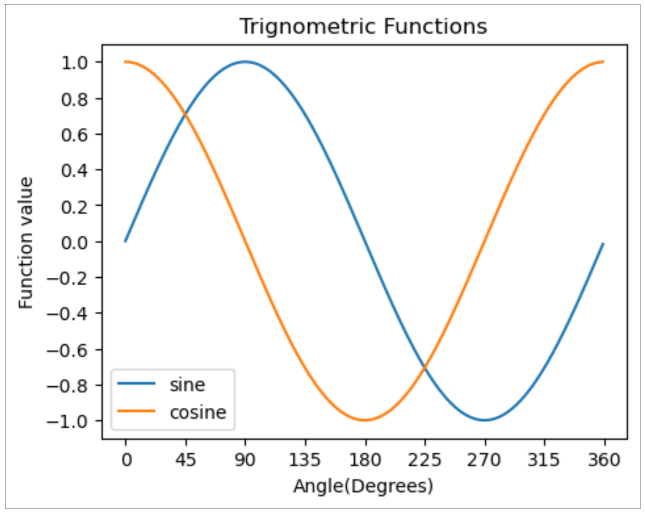


Reading Graphs

- When I look at a plot, I want to be able to *easily read out features of the data*.
- I also want to be able to easily understand the domain and range of the the data and find the minimum and maximum.
- Here the domain is 0 to 360 and the range is -1 to 1.
- The features of the data are clear maxima and minima at function values -1 and 1 of the curves plotted but its not that easy to read out at values of angle at which the maxima occur.
- The solution is to take control of the x and/or y axis values.

In [32]:

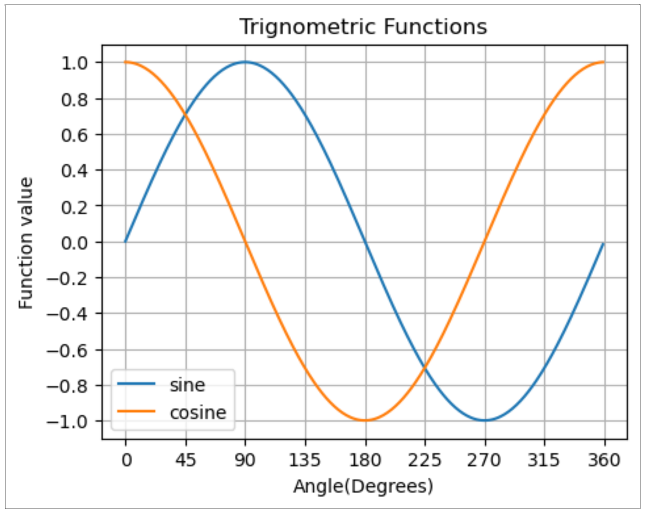
```
fig = plt.figure(figsize = (4,3))   #creates a blank canvas
ax = fig.add_axes([0,0,1,1]) #creates an axes
#- (starting pt x, starting pt y, fractional size x, fractional size y)
ax.plot(angle_in_degrees,S,label ='sine')  #add a plot to the axes.
                                #I also gave the line a label.
ax.plot(angle_in_degrees,C,label = 'cosine' ) #add a plot to the axes.
                                #I also gave the line a label.
ax.set_xlabel('Angle(Degrees)') #I added a x axis label
ax.set_ylabel('Function value') #I added a y axis label
ax.legend() #I gave the figure a legend.
ax.set_title('Trignometric Functions') # Let's give the figure a title.
xticklocations = np.linspace(0,360,9) # I am going to determine to have 9 ticks
                                    #on the x axis between 0 and 360(inclusive)
ax.set_xticks(xticklocations)
yticklocations = np.linspace(-1,1,11) # I am going to determine to have 11 ticks
                                #on the y axis between -1 and 1(inclusive)
ax.set_yticks(yticklocations)
plt.show()
```



- **OK I'm feeling better** But, a plot can always be improved.

In [33]:

```
fig = plt.figure(figsize = (4,3))   #creates a blank canvas
ax = fig.add_axes([0,0,1,1]) #creates an axes
#- (starting pt x, starting pt y, fractional size x, fractional size y)
ax.plot(angle_in_degrees,S,label ='sine')  #add a plot to the axes.
                                #I also gave the line a label.
ax.plot(angle_in_degrees,C,label = 'cosine' ) #add a plot to the axes.
                                #I also gave the line a label.
ax.set_xlabel('Angle(Degrees)') #I added a x axis label
ax.set_ylabel('Function value') #I added a y axis label
ax.legend() #I gave the figure a legend.
ax.set_title('Trignometric Functions') # Let's give the figure a title.
xticklocations = np.linspace(0,360,9) # I am going to determine to have 9 ticks
                                #on the x axis between 0 and 360(inclusive)
ax.set_xticks(xticklocations)
yticklocations = np.linspace(-1,1,11) # I am going to determine to have 11 ticks
                                    #on the y axis between -1 and 1(inclusive)
ax.set_yticks(yticklocations)
plt.grid(True)  #Here I turned on grid lines, to improve readability.
plt.show()
```



Controlling Line Style and Color and Width and/or Marker Type and Color and Size

THERE IS A LOT OF CONTROL AVAILABLE HERE.

Colors -

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

Line styles

- '-' solid line
- '--' dahsed line
- '-." dash dot line
- ':' dotted line

Line width

```
linewidth = 2
```

Marker shapes

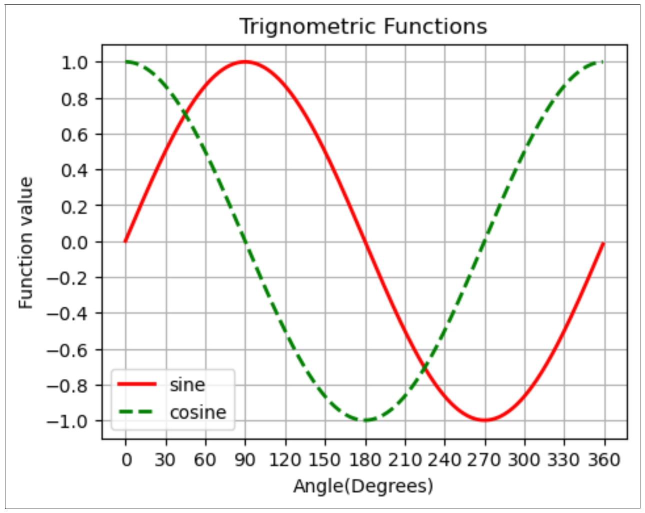
- '.'- point
- 'o' - circle
- 'x' - x marker
- 'D' = diamond marker
- 'H' - hexagon marker
- 's' - square marker
- '+' plus marker

Marker size

- `markersize = 12`

In [34]:

```
fig = plt.figure(figsize = (4,3))   #creates a blank canvas, figsize should be set to a tuple ()
ax = fig.add_axes([0,0,1,1]) #creates an axes
# - (starting pt x, starting pt y, fractional size x, fractional size y)
ax.plot(angle_in_degrees,S,'r-', linewidth = 2, label ='sine')  #add a plot to the axes.
                                #I also gave the line a label.
                                # I selected the color red and a solid line
                                # I set the linewidth to 2, default is 1.
ax.plot(angle_in_degrees,C,'g--', linewidth = 2, label = 'cosine' ) #add a plot to the axes.
                                #I also gave the line a label.
                                # I selected the color green and a dashed line
                                # I set the linewidth to 2, default is 1.
ax.set_xlabel('Angle(Degrees)') #I added a x axis label
ax.set_ylabel('Function value') #I added a y axis label
ax.legend() #I gave the figure a legend.
ax.set_title('Trignometric Functions') # Let's give the figure a title.
xticklocations = np.linspace(0,360,13) # I am going to determine to have 13 ticks
                                    #on the x axis between 0 and 360(inclusive)
ax.set_xticks(xticklocations)
yticklocations = np.linspace(-1,1,11) # I am going to determine to have 11 ticks
                                    #on the y axis between -1 and 1(inclusive)
ax.set_yticks(yticklocations)
plt.grid(True)  #Here I turned on grid lines, to improve readability.
plt.show()
```



Bar plots

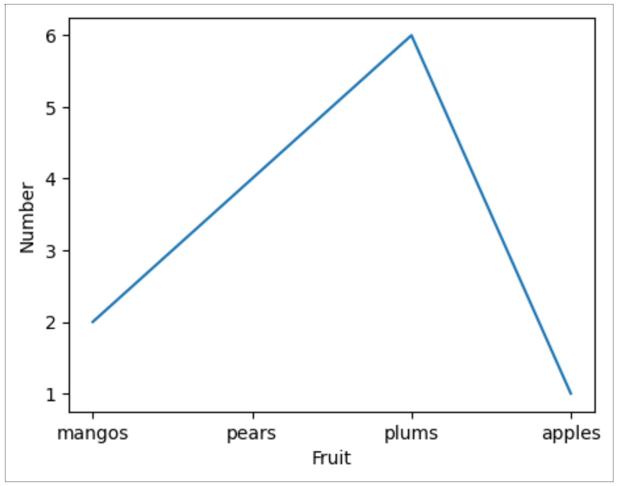
Here I am going to make an example of a bar plot.

In [35]:

```
fruitnames = ['mangos','pears','plums','apples'] #This is a list
fruitnumber = np.array([2,4,6,1]) #I converted a list of fruit counts into a numpy array. Actually not needed.
```

```
In [36]:
```

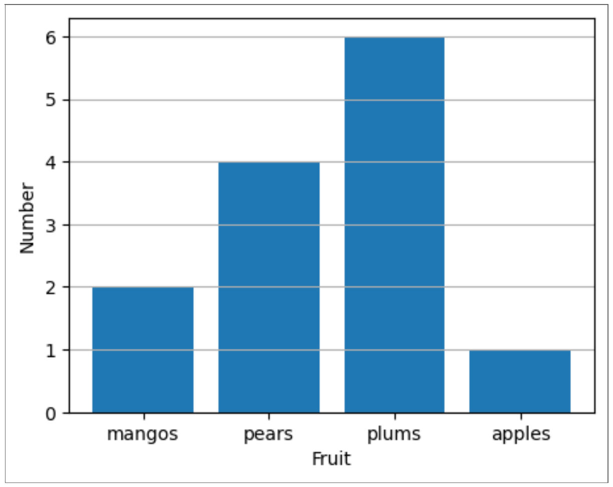
```
fig = plt.figure(figsize = (4,3) )    #creates a blank canvas
ax = fig.add_axes([0,0,1,1]) #creates an axes - (starting pt x, starting pt y, fractional size x, fractional size y)
ax.plot(fruitnames,fruitnumber)
ax.set_xlabel('Fruit')
ax.set_ylabel('Number')
plt.show()
```



We need a bar plot.

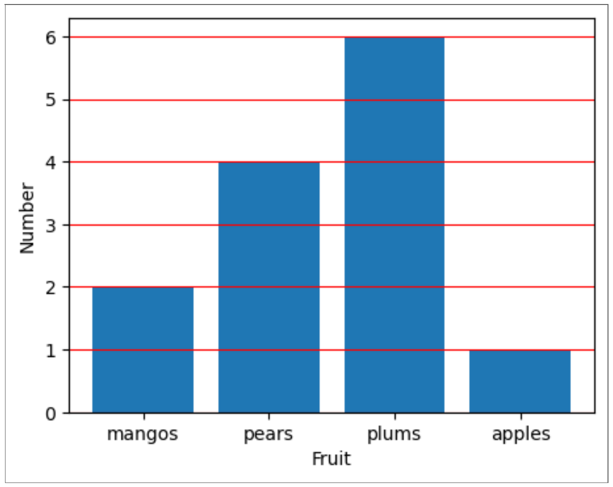
In [37]:

```
fig = plt.figure(figsize = (4,3))   #creates a blank canvas
ax = fig.add_axes([0,0,1,1]) #creates an axes
ax.bar(fruitnames,fruitnumber) #
ax.set_xlabel('Fruit')
ax.set_ylabel('Number')
ax.grid(True,axis='y') #I added a grid and specified it
                #should only be for the y axis
plt.show()
```



In [38]:

```
fig = plt.figure(figsize = (4,3))    #creates a blank canvas
ax = fig.add_axes([0,0,1,1]) #creates an axes
ax.bar(fruitnames,fruitnumber) #
ax.set_xlabel('Fruit')
ax.set_ylabel('Number')
ax.grid(True,axis='y',color='r')#I added a grid and specified it
                #should only be for the y axis and set its color to red
plt.show()
```



Subplot

- Sometimes we want to put more than one graph in a figure. In this case, we can divide the figure into multiple plots.
- The syntax `subplots(n,m)` when I create the figure tells python I want a figure with with n row and m columns each of which can contain a separate **axis**

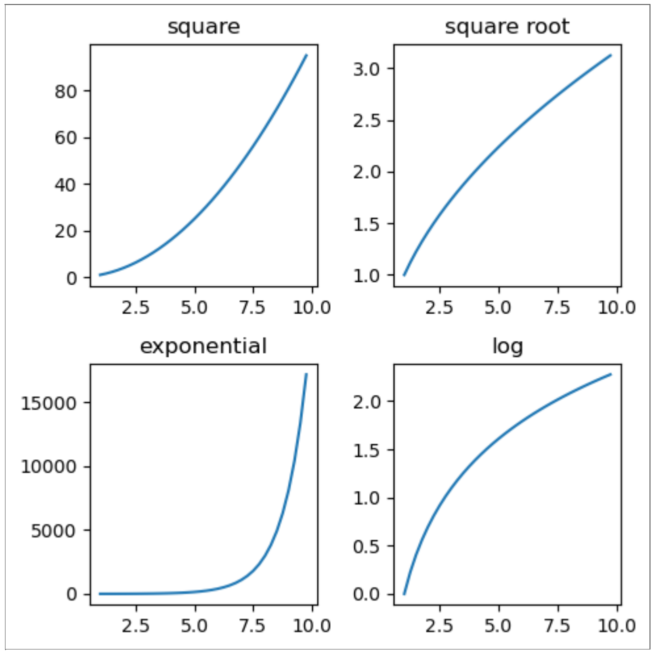
In [39]:

```
#Create some data for four plots
x = np.arange(1,10,0.25)
y1 = x**2
y2 = np.sqrt(x)
y3 = np.exp(x)
y4 = np.log(x) # for the record this is a natural log or ln,
               #base 10 logarithm is log10, and base 2 logarithm is log2
```

In [40]:

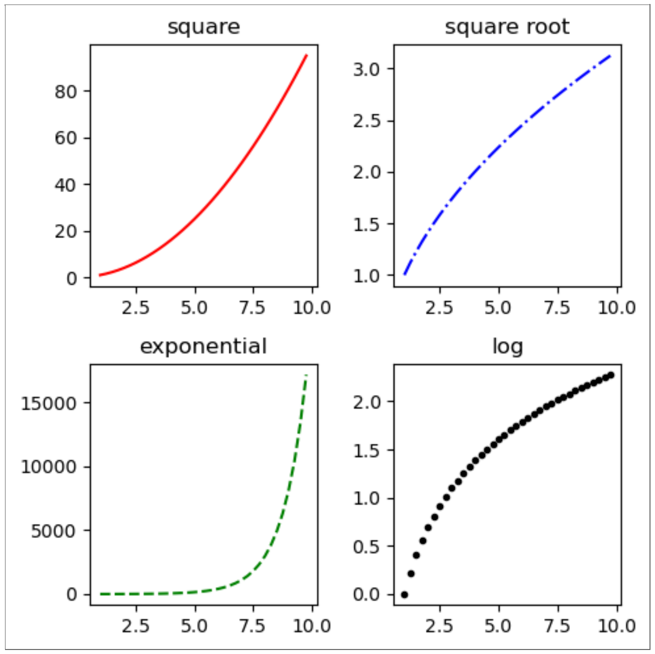
```
fig,a = plt.subplots(2,2,figsize =(5,5))  #Here i create both the figure (fig) and the axes (a) in a single step.
                                       #Any options that you would send into the figure call, you can send to subplots.
                                       # I've asked for 4 subplots in a 2 x 2 grid.

a[0][0].plot(x,y1)   #notice the syntax in dealing with the axis. The axes have a row index and a column index.
a[0][0].set_title('square')
a[0][1].plot(x,y2)
a[0][1].set_title('square root')
a[1][0].plot(x,y3)
a[1][0].set_title('exponential')
a[1][1].plot(x,y4)
a[1][1].set_title('log')
fig.tight_layout()
#this is really cool and fixes problems with overlapping text and figures
plt.show()
```



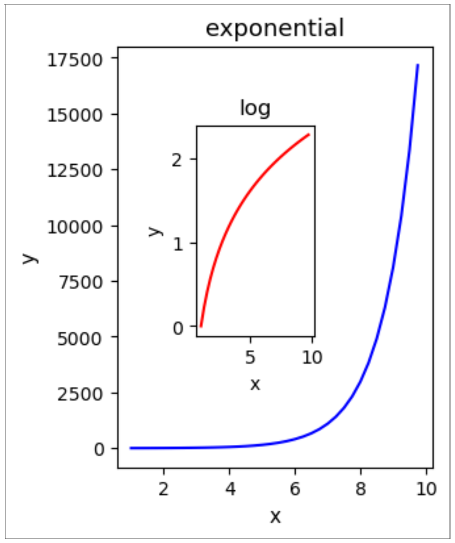
In [41]:

```
fig,a = plt.subplots(2,2, figsize = (5,5))
#note that all of the options in the figure command can be used here.
#I made a square figure window, because doing so made all my plots into nice squares.
a[0][0].plot(x,y1,'r-')
a[0][0].set_title('square')
a[0][1].plot(x,y2,'b-.')
a[0][1].set_title('square root')
a[1][0].plot(x,y3,'g--')
a[1][0].set_title('exponential')
a[1][1].plot(x,y4,'k.')
a[1][1].set_title('log')
plt.tight_layout()
plt.show()
```



In [42]:

```
fig=plt.figure(figsize = (3,4))
axes1 = fig.add_axes([0.1, 0.1, 0.8, 0.8]) # main axes
axes2 = fig.add_axes([0.3, 0.35, 0.3, 0.4]) # inset axes
axes1.plot(x,y3, 'b-')
axes2.plot(x,y4,'r-')
axes1.set_title('exponential', fontsize = 13)
axes1.set_xlabel('x', fontsize = 11)
axes1.set_ylabel('y', fontsize = 11)
axes2.set_title("log", fontsize = 12)
axes2.set_xlabel('x', fontsize = 10)
axes2.set_ylabel('y',fontsize = 10)
plt.show()
```



Logarithmic Scales

- **THIS MAY BE THE MOST IMPORTANT PART OF THE TUTORIAL!**
- By default the plots we make are on linear scales. But many times, the data can be better understood on a *logarithmic* scale.

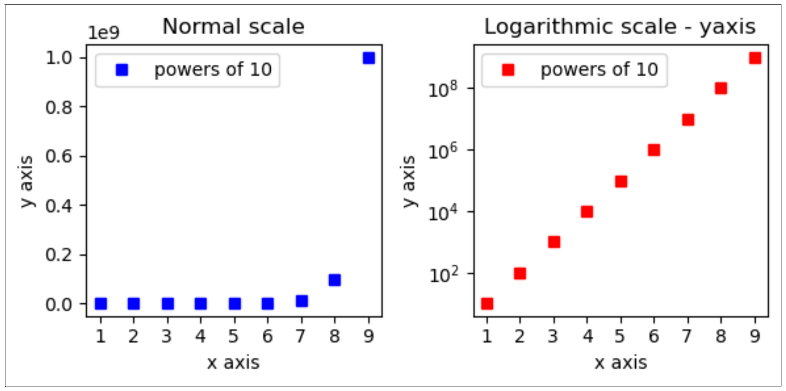
In [43]:

```
# Logarithmic scale

x = np.arange(1,10,1)
z1 = 10**x

fig, axes = plt.subplots(1, 2, figsize=(6,3))
xtickvals = np.linspace(1,9,9)
axes[0].plot(x,z1,'bs',label = 'powers of 10')
axes[0].set_title("Normal scale")
axes[1].plot(x,z1,'rs',label = 'powers of 10')
axes[1].set_title("Logarithmic scale - yaxis")
axes[1].set_yscale("log")  # Here I set the y axis to a logarithmic scale
axes[0].set_xlabel("x axis")
axes[0].set_ylabel("y axis")
axes[1].set_xlabel("x axis")
axes[1].set_ylabel("y axis")

axes[0].set_xticks(xtickvals)
axes[1].set_xticks(xtickvals)
axes[0].legend()
axes[1].legend()
plt.tight_layout()
plt.show()
```



- The plot on the left seems "wrong" but is in fact correct. Although the y axis is labeled 0 to 1 on top of the y axis is the value 1e9. this is to indicate that 1 corresponds to 1e9
- The plot on the right is much better and easier to understand. The y axis has been placed on a logarithmic scale such that each tick mark corresponds to an increase by a **multiplicative** factor of 10^2 or 100. The exponential function now looks linear.
- This makes sense, because every unit increase in the x axis is **multiplying** the y axis by a factor of 10.

In [44]:

```
y = np.array([0.001, 0.01, 0.1, 1, 10, 100, 1000])  # Each element is a factor of 10 larger than the previous element.
z2 = np.log10(y)
```

Rule of Thumb

- If the data looks squished, explore using logarithmic scaling to better visualize the data.
- It is extremely useful when communicating data to manipulate the x and y axis scales to see if we can make the data **look** linear by using logarithmic scales on either or (both) axis. That tells us something useful about the relationship between the variables. When exploring data, we often make one or both of the axis logarithmic.