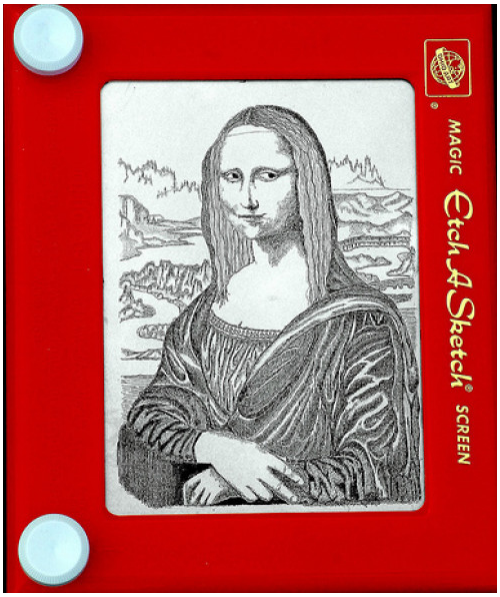
matplotlib
Mathematical Programming with Python

https://people.sc.fsu.edu/~jburkardt/classes/math1800_2023/matplotlib/matplotlib.pdf

---



Data visualization

- *The eye can see patterns that are hidden in large datasets;*
- *To visualize data, we need functions from the* `matplotlib.pyplot` *library;*
- *We will use* `import matplotlib.pyplot as plt` *to simplify our work;*
- *The simplest command is* `plt.plot(x,y)`*, which plots (x,y) data;*
- *Many commands are available to annotate this basic plot;*
- *Multiple* `plt()` *commands can draw on the same plot;*
- *Complicated data, such as z(x,y), can be displayed with a "heat map";*

1 Getting ready to plot

Python plotting requires checking out a plottling library. We will use the most popular plotter, known as `matplotlib`. To accedss this library, we must issue the command

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

Once you have imported `matplotlib`, you can request help on `plot()` or other commands such as `clf()` or `show()` using the `help()` command:

```
help ( plt.clf )
help ( plt.fill )
help ( plt.plot )
help ( plt.scatter )
help ( plt.show )
```

`matplotlib` has many similarities to the MATLAB graphics environment, although this is true only up to a point.

The fundamental command for plotting points and lines given x and y data has the form

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

So to make the simplest plot, we just need to figure out how to make `x` and `y` objects that `plot()` will accept.

2 Plot $y = x^2$

Luckily, `plot()` can accept data in the form of lists. We certainly know how to make a simple, short list. Let's make a simple plot of $y = x^2$, using 9 data points equally spaced in $-2 \leq x \leq 2$. Because this is a small data set and the formula is simple, we can write out our two lists of data by hand:

```
x = [ -3, -2, -1, 0, 1, 2, 3 ]
y = [  9,  4,  1, 0, 1, 4, 9 ]
plt.plot ( x, y )
```

Surprisingly, nothing happens. This is because `matplotlib` waits in case we want to add more information (plots, labels, grids, and so on). Once we think we have got everything specified, we can display the plot by

```
plt.show ( )
```

After the plot is shown, it cannot be further modified.

3 `plt.clf()` and `plt.close()`

If you want to start a fresh plot (particularly after you have already made one), you will usually need to ask that the current figure be cleared away, using the `clf()` or "clear figure" command. If you don't do this, you may find that your new plot is combined with the previous one in a confusing way. It doesn't hurt to always issue a `plt.clf()` command when starting a plot, even if it's the first plot in a series.

Sometimes, especially when you are running a script instead of using Python interactively, the plotting system will be unhappy if the script terminates with you issuing a `plt.close()` command. Usually, there is no real danger, but I always issue such a command after I have completed the definition of a plot.

Thus, at this point, our simple little program might begin to look a little more complicated:

```
import matplotlib.pyplot as plt
plt.clf()
x = [ -3, -2, -1, 0, 1, 2, 3 ]
y = [  9,  4,  1, 0, 1, 4, 9 ]
plt.plot ( x, y )
plt.show ( )
plt.close ( )
```

4 Plot $y = x$ and $y = \cos(x)$

In an earlier discussion, we plotted two curves on one display. To show the cosine curve, we need lots of points, and we need to use a function to get the values. So we don't plan to simply type lists of x and y data. We will instead use a loop and the `append()` function. Let's guess we need 101 points over $0 \leq x \leq 1$.

```
from math import cos
n = 101
xlist = []
ylist = []
for i in range ( 0, n + 1 ):
  x = i / ( n - 1 )    #  x = 0, 0.01, 0.02, ..., 1.0
  y = cos ( x )
  xlist.append ( x )
  ylist.append ( y )

plt.plot ( xlist, xlist, 'k-' )  # Plots y = x, black line
plt.plot ( xlist, ylist, 'r-')   # Plots y = cos(x), red line
plt.show ( )
```

Instead of the formula for x, we could instead have written

```
x = 0.0
dx = 0.1
for i in range ( 0, n + 1 ):
  y = cos ( x )
  xlist.append ( x )
  ylist.append ( y )
  x = x + dx
```

but even this formulation is easy to get slightly wrong. Also, the the last value of x in our list will probably not be exactly 1, because of roundoff.

The simple tasks of defining `xlist` and `ylist` seem to take up more of our program than the plotting statements do. This will become much easier when we can use the `numpy` library.

5 Plot a star

Now let's take some data that does **not** represent a graph of some function $y = f(x)$. We will suppose that this data in the form of a list of lists:

```
data = [
 [ 0.95,  0.31 ],
 [ 0.22,  0.31 ],
 [-0.95,  0.31 ],
 [-0.36, -0.12 ],
 [ 0.59, -0.81 ],
 [ 0.36, -0.12 ],
 [ 0.00,  1.00 ],
 [-0.22,  0.31 ],
 [-0.59, -0.81 ],
 [ 0.00, -0.38 ] ]
```

If you don't want to type this information, but you want to try this example, you can copy/paste this data, or else copy the file *star_data.py* and set

```
from star_data import star_data
data = star_data ( )
```

Our information in `data` is a list of lists. To call `plt.plot()`, we need to extract the x and y values as separate lists. It would seem possible that we could do this as follows:

```
plt.plot ( data[:][0], data[:][1] )  # NO, THIS DOESN'T WORK!
```

Our logical guess for the right thing to do has failed. We could always do this by using a `for()` loop to locate the values from each row and append them to our new lists. Can we be sure we remember how to do that?

As an alternative, there is a technique called *list comprehenseion* which can help. Here, it will let us consider each row of the list of lists, and extract the first or second element into a new list:

```
xlist = [ row[0] for row in data ]
ylist = [ row[1] for row in data ]
```

Although a little mysterious, if you can get familiar with these one-line statements, they can be very helpful when working with lists and other Python datatypes.

Once we have managed to get arrays `x` and `y` defined, we can simply say

```
plt.plot ( xlist, ylist )
plt.show ( )
```

We get a figure that is "almost" a star. What are we seeing and why did part of the star not get drawn? What simple "surgery" on our `xlist` and `ylist` data will make a new plot that draws the entire star?

6 Modifying the color, thickness, or type of a line

Let's draw the star in red:

```
plt.plot ( xlist, ylist, 'r' )
```

Other one-letter color codes include 'g', 'b', 'c', 'y', 'm', and 'k' for black.

Let's draw the star with lines 5 times thicker than the default:

```
plt.plot ( xlist, ylist, 'g', linewidth = 5 )
```

The input `linewidth = 3` is an example of keyword input. Many Python functions allow you to specify an input quantity by a phrase of the form `input_name = input_value`.

Let's draw the star using a dotted line style:

```
plt.plot ( x, y, 'b:' )
```

Let's draw the star with no lines, but dots at the data.

```
plt.plot ( x, y, 'c.', markersize = 10 )
```

A similar plot can be made with the `plt.scatter()` function.

```
plt.scatter ( x, y )
```

which allows you to add one or more colors, and one or more sizes for the dots, if you wish.

7 A "filled" plot

Let's use `plt.fill()` to fill in our star shape with color.

```
plt.fill ( x, y, 'm' )
```

This plot suggests a lot of new things we can do, unrelated to the usual $y = f(x)$ plots. For instance, how would we go about plotting a red/black checkerboard?

Who can help me here? Python Jeopardy Time!

```
for row in range ( 0, 8 ):
  for col in range ( 0, 8 ):
    x = [ row, row, row+1, row+1, row ]
    y = [ col, col+1, col+1, col, col ]
    if ( ( row + col ) % 2 == 0 ):
      plt.fill ( x, y, 'r' )
    else
      plt.fill ( x, y, 'k' )

plt.show )
```

Later we will see how to make the squares show up as squares rather than rectangles, and how to hide the axis lines!

8 Plotting n points over $xmin \leq x \leq xmax$

Suppose we need to plot n points over an interval $[xmin, xmax]$. We may suppose that n might be large, or might change several times as we adjust our plot. We probably want to choose equally spaced points in the interval, but we may realize that the spacing is not going to be some simple fraction like `dx=0.1`. So we need to be comfortable with the idea that we can handle such a problem in cases where we are using lists, so that we have to do the assignments by hand.

For n equally spaced points in $xmin \leq x \leq xmax$, the spacing can be written as $dx = \frac{xmax-xmin}{n-1}$. This is why I frequently choose values of n like 11, 51, or 101, or 501, so that the value of dx is a relatively simple quantity. Realize that 10 equally spaced points in the unit interval defines a spacing of $dx = 1/9 = 0.1111...$, and not the spacing $dx = 0.1$ that you might mistakenly imagine.

So here are three ways to generate the list of x values. You may find one makes more sense to you than the others.

We can recursively increment x:

```
x = xmin
dx = ( xmax - xmin ) / ( n - 1 )
for i in range ( 0, n ):
  y = ?
  xlist.append ( x )
  ylist.append ( y )
  x = x + dx
```

or we can add i increments to the starting point:

```
dx = ( xmax - xmin ) / ( n - 1 )
for i in range ( 0, n ):
  x = xmin + i * dx
  y = ?
  xlist.append ( x )
  ylist.append ( y )
```

or we can use the convex combination formula:

```
for i in range ( 0, n ):
  x = ( ( n - 1 - i ) * xmin + i * xmax ) / ( n - 1 )
  y = ?
  xlist.append ( x )
  ylist.append ( y )
```

Let's try one of these approaches that will allow us to easily adjust the value of n as we try to make a sensible plot of the function $y = x^2 + \sin(53x)$ over the interval $[\frac{-\pi}{2}, \frac{\pi}{2}]$.

```
from math import pi
n = 11
xlist = []
ylist = []
xmin = -pi / 2
xmax =  pi / 2
dx = ( xmax - xmin ) / ( n - 1 )
for i in range ( 0, n ):
  x = xmin + i * dx
  y = x**2 + sin ( 53.0 * x )
  xlist.append ( x )
  ylist.append ( y )

plt.clf ( )
plt.plot ( xlist, ylist )
plt.show ( )
plt.close ( )
```

9 Extras

We have concentrated on the basic technique of getting something to show up on the plotting screen. After we've gotten the picture we want, it's often important to be able to adjust and label and highlight our plot. We will come back to this idea in a later discussion, but I'd just like to suggest a few items that you may quickly find useful.

When presenting at a plot, it is often helpful to add gridlines, so that an observer can better determine the locations of various interesting points. This is easily done by issuing, before the `plt.show()` command, the following request:

```
plt.grid ( True )
```

You may also want to preserve a copy of your plot as a graphics file. Again, before issuing the `plt.show()` command, you can save a snapshot of your plot by

```
plt.savefig ( filename )
```

where `filename` includes an extension that indicates the desired format, such as

```
'graph.eps'
'picture.pdf'
'plot.png'
'my_file.svg'
```

Note that the *jpg* format is **not** supported by `savefig()`.