Part 29
-
From CSV file to Graphs

Version: 2018-03-18

Using Mathplot

Matplotlib is a powerful two-dimensional plotting library for the Python language. Matplotlib is capable of creating all manner of graphs, plots, charts, histograms, and much more.
In most cases, matplotlib will simply output the chart to your viewport when the `.show()` method is invoked, but we'll briefly explore how to save a matplotlib creation to an actual file on disk.

https://matplotlib.org/api/pyplot_summary.html

Installing matplotlib
Like all Python libraries, you'll need to begin by installing matplotlib

```
sudo apt-get install -y python-matplotlib
```

Using matplotlib
While the feature-list of matplotlib is nearly limitless, we'll quickly go over how to use the library to generate a basic chart for your own testing purposes.
Import the matplotlib library. You'll likely also want to import the pyplot sub-library, which is what you'll generally be using to generate your charts and plots when using matplotlib.

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

Now to create and display a simple chart, we'll first use the `.plot()` method and pass in a few arrays of numbers for our values. For this example, we'll plot the number of books read over the span of a few months.

```
plt.plot([0, 1, 2, 3, 4], [0, 3, 5, 9, 11])
```

We can also add a few axis labels:

```
plt.xlabel('Months')
plt.ylabel('Books Read')
```

Finally, we can display the chart by calling `.show()`:

```
plt.show()
```

The `savefig` Method
With a simple chart under our belts, now we can opt to output the chart to a file instead of displaying it (or both if desired), by using the `.savefig()` method.

```
plt.savefig('books_read.png')
```

The `.savefig()` method requires a filename be specified as the first argument. This filename can be a full path and as seen above, can also include a particular file extension if desired. If no extension is provided, the configuration value of `savefig` format is used instead.

Additional `savefig` **Options**
In addition to the basic functionality of saving the chart to a file, `.savefig()` also has a number of useful optional arguments.

```
savefig(fname, dpi=None, facecolor='w', edgecolor='w',
        orientation='portrait', papertype=None, format=None,
        transparent=False, bbox_inches=None, pad_inches=0.1,
        frameon=None)
```

- dpi: [None | scalar > 0 | 'figure']
  The resolution in dots per inch. If None it will default to the value savefig.dpi in the matplotlibrc file. If 'figure' it will set the dpi to be the value of the figure.

- facecolor, edgecolor:
  the colors of the figure rectangle

- orientation: ['landscape' | 'portrait']
  not supported on all backends; currently only on postscript output

- papertype:
  One of 'letter', 'legal', 'executive', 'ledger', 'a0' through 'a10', 'b0' through 'b10'. Only supported for postscript output.

- format:
  One of the file extensions supported by the active backend. Most backends support png, pdf, ps, eps and svg.

- transparent:
  If True, the axes patches will all be transparent; the figure patch will also be transparent unless facecolor and/or edgecolor are specified via kwargs. This is useful, for example, for displaying a plot on top of a colored background on a web page. The transparency of these patches will be restored to their original values upon exit of this function.

- frameon:
  If True, the figure patch will be colored, if False, the figure background will be transparent. If not provided, the rcParam 'savefig.frameon' will be used.

- bbox_inches:
  Bbox in inches. Only the given portion of the figure is saved. If 'tight', try to figure out the tight bbox of the figure.

- pad_inches:
  Amount of padding around the figure when bbox_inches is 'tight'.

- bbox_extra_artists:
  A list of extra artists that will be considered when the tight bbox is calculated.

From CSV file to graphs

Assume we have this sample csv file '`sample.csv`' containing 960 lines of data and a header line.

```
Date,Time,Temperature,Humidity,Pressure
2017-08-01,00:02:43,16.9,39.2,1012.6
2017-08-01,00:07:56,17.0,46.7,1012.8
2017-08-01,00:38:46,15.8,37.2,1013.0
2017-08-01,00:43:58,15.6,45.5,1013.1
2017-08-01,00:49:08,15.6,53.2,1013.2
2017-08-01,00:54:17,15.4,55.8,1013.2
.
.
.
2017-08-06,14:29:23,25.0,38.3,1020.4
2017-08-06,14:30:52,29.4,36.1,1020.3
2017-08-06,14:36:05,29.6,37.2,1020.3
2017-08-06,14:37:35,28.2,36.0,1020.3
```

As you can see, it consists of a 5 colomns being `date, time, temperature humidity` and `pressure`.

Read the csv file
First you'll need to import the reader function from the csv module.

```
from csv import reader
```

Next you can open the csv sheet and store the data in a list.

```
with open('sample.csv', 'r') as f:
    data = list(reader(f))
```

Have a look at the contents of the data list that you have created. This will show you the headers of the csv sheet.

```
data[0]
```

If you want to have a look at the first set of values in the data, then you can use:

```
data[1]
```

You can see the last items in the list, by typing:

```
data[-1]
```

Getting specific data sets.
At the moment the values are stored as a *list of lists*. To graph the data, you'll want specific data sets. For instance, you might want to get the temperature.

```
Date,Time,Temperature,Humidity,Pressure
```

This is the *2nd* item in the list, as lists are indexed from 0.
You can use a *list comprehension* to extract the temperatures from the *list of lists*. This line of code, takes the third value from every item in data.

```
temp = [i[2] for i in data]
```

You can look at the contents of some of the temp data, by typing into the interpreter. This will show you the start of the list, which is the header:

```
temp[0]
```

This will show you the first 10 items in the list:

```
temp[0:10]
```

You don't actually want the header, so you can alter the *list comprehension* so that the first item of the data list is ignored.

```
temp = [i[3] for i in data[1::]]
```

You can do the same again, to get the date-time stamps from data.

```
time = [i[19] for i in data[1::]]
```

Your first graph
The Matplotlib module (along with *numpy* and *scipy*) is one of the reasons so many mathematicians and scientists use Python. It's an excellent way of drawing graphs.

You'll need to import the module, to begin with:

```
from matplotlib import pyplot
```

You need only two lines to graph your data:

```
pyplot.plot(range(len(temp)), temp)
pyplot.show()
```

`pyplot.plot()` needs to be given two *iterables*. In the above, you have given it `range(len(temp))` which is all numbers from 0 up to the length of the `temp` list. You've also given it `temp` which is the set of temperatures. `pyploy.show()` draws the graph.

Adding dates and times
At the moment, the date-time isn't being used in the graph. The reason you can't really use it at the moment is because the data is not in a format that matplotlib can recognise '2015-08-20 23:58:30'

This string needs changing to a datetime object, which is fortunately quite easy.
You need to import the parser function from the dateutil module to begin with.

```
from dateutil import parser
```

To convert a date in *string* format, to a *datetime* object, the syntax is fairly simple. For instance:

```
parser.parse('2015-08-20 23:58:30')
```

You want to convert each date that is added to the time list, so you can edit your list comprehension to read:

```
time = [parser.parse(i[19]) for i in data[1::]]
```

Now you can change your `pyplot.plot()` call so it looks like this

```
pyplot.plot(time, temp)
pyplot.show()
```

Your complete code should look like:

```
from matplotlib import pyplot
from csv import reader
from dateutil import parser

with open('sample.csv', 'r') as f:
    data = list(reader(f))

print data[0]
print data[1]
print data[2]
print data[-1]

temp = [i[2] for i in data[1::]]

print temp[0]
print temp[0:10]

pyplot.plot(range(len(temp)), temp)
pyplot.xlabel('range')
pyplot.ylabel('temp')
pyplot.savefig('range-temp.png')

date = [i[0] for i in data[1::]]
time = [i[1] for i in data[1::]]
dt = []
for x in range(0,len(date)):
   dt.append(parser.parse(date[x] + " " + time[x]))

pyplot.xlabel('datetime')
pyplot.ylabel('temp')

fig, ax = pyplot.subplots()
ax.plot_date(dt, temp, 'b-', 'CET', True)
pyplot.savefig('dt-temp.png')
```

Adding Titles and Axis
Graphs should always be titles and have labelled axis. Again, this is trivial with matplotlib. First you can add a title:

```
pyplot.title('Temperature changes over Time')
```

Then the *x axis*:

```
pyplot.xlabel('Time/hours')
```

And lastly the *y axis*:

```
pyplot.ylabel('Temperature/$^\circ$C')
```

Rendering a CSV File as a Graph Using Python and Matplotlib

Step 1
Create a simple CSV file for testing. A sample might look like this:

```
1,2
2,3
3,8
4,13
5,18
6,21
7,13
7.5,4
2.5,4.3
```

Step 2
Import the necessary python libraries into your code file:

```
import matplotlib.pyplot as plt
import csv import sys
```

Step 3
Open the CSV file and create a reader object from it. Declare variables to define the upper and lower bounds for the x and y axis values of the graph:

```
csv_reader = csv.reader(open('test.csv'))
bigx   = float(-sys.maxint -1)
bigy   = float(-sys.maxint -1)
smallx = float(sys.maxint)
smally = float(sys.maxint)
```

Step 4
Iterate over each row contained in the reader object storing each row as a vertex in a vertex array. In the same loop compare the x and y values in order to store their upper and lower bounds. Sort the vertex array and then loop through it again. This time store the sorted x and y values in separate arrays:

```
verts = []
for row in csv_reader:
    verts.append(row)
    if float(row[0]) > bigx:
        bigx = float(row[0])
    if float(row[1]) > bigy:
        bigy = float(row[1])
    if float(row[0]) < smallx:
        smallx = float(row[0])
    if float(row[1]) < smally:
        smally = float(row[1])
verts.sort()
x_arr = []
y_arr = []
for vert in verts:
    x_arr.append(vert[0])
    y_arr.append(vert[1])
```

Step 5
Create a FigureCanvas object using the imported matplotlib pyplot object. Add the graph's axes to the FigureCanvas by calling the function add_axes and passing it an array of values in the form of: left, bottom, width, height. These values define where the graph is placed on the canvas —they can range from 0.0 to 1.0:

```
fig = plt.figure()
ax = fig.add_axes([0.1, 0.1, 0.8, 0.8])
```

Step 6
Format the graph adding labels and defining the minimum and maximum values for each axis:

```
ax.set_xlabel('x data')
ax.set_ylabel('y data')
ax.set_xlim(smallx,bigx)
ax.set_ylim(smally,bigy)
```

Step 7
Plot the graph by passing in the two arrays containing the x and y values retrieved from the CSV file. Customize the line plot by passing in optional values such as line color (color) or line width (lw). Display the finished graph by calling the show method to open a window and store the image by calling savefig to create a bitmap file on disk:

```
ax.plot(x_arr,y_arr, color='blue', lw=2)
plt.show()
fig.savefig('test.png')
```