Johnny Wei-Bing Lin

A Hands-On Introduction to Using Python in the Atmospheric and Oceanic Sciences

http://www.johnny-lin.com/pyintro

2012

 Printed version: ISBN 978-1-300-07616-2. PDF versions: No ISBNs are assigned.

Who would *not* want to pay money for this book?: if you do not need a black-and-white paper copy of the book, a color PDF copy with functional hyperlinks, have limited funds, or are interested in such a small portion of the book that it makes no sense to buy the whole thing. The book's web site (http://www.johnny-lin.com/pyintro) has available, for free, PDFs of every chapter as separate files.

Who would want to pay money for this book?: if you want a black-and-white paper copy of the book, a color PDF copy with functional hyperlinks, or you want to help support the author financially. You can buy a black-and-white paper copy of the book at http://www.johnny-lin.com/pyintro/buypaper.shtml and a hyperlink-enabled color PDF copy of the book at http://www.johnny-lin.com/pyintro/buypdf.shtml.

A special appeal to instructors: Instruction at for-profit institutions, as a commercial use, is not covered under the terms of the CC BY-NC-SA, and so instructors at those institutions should not make copies of the book for students beyond copying permitted under Fair Use. Instruction at not-for-profit institutions is not a commercial use, so instructors may legally make copies of this book for the students in their classes, under the terms of the CC BY-NC-SA, so long as no profit is made through the copy and sale (or Fair Use is not exceeded). However, most instruction at not-for-profit institutions still involves payment of tuition: lots of people are getting paid for their contributions. Please consider also paying the author of this book something for his contribution.

Regardless of whether or not you paid money for your copy of the book, you are free to use any and all parts of the book under the terms of the CC BY-NC-SA.

Chapter 9

Visualization: Basic Line and Contour Plots

With so much of the analysis of AOS problems requiring graphing or visualization of some sort, no scientist's toolkit is complete without a robust visualization suite. Because Python is an open-source programming language, you have not just one visualization suite but several to choose from. For AOS graphics, NCAR's PyNGL, UV-CDAT's Visualization Control System (vcs), and matplotlib are three powerful packages that can handle most AOS visualization tasks. While each has its own strengths and weaknesses, in this chapter we will focus on matplotlib and its 2-D graphics routines to create line and contour plots.[1]

(By the way, just a heads-up that in this chapter, the plots and tables will usually be in figures that float to wherever on the page works best for optimizing the page. Plots and tables may not immediately follow where they are first mentioned.)

9.1 What is matplotlib?

Matplotlib, as its name suggests, emulates the Matlab plotting suite: commands look like Matlab commands. It has a function-centric interface adequate for the needs of most users (especially first-time users), but the entire suite is object-based, so power users have fine-grained control over the de-

[1] PyNGL implements the graphing resources of the NCAR Command Language (NCL) into Python (NCL has more "high-level" functions, but PyNGL can draw everything NCL can). Vcs is the original plotting module for CDAT and UV-CDAT. Its default settings are not always pretty, but they make use of the masks and metadata attached to masked variables, so plotting is fast. Section 10.2 tells you where to go to obtain these packages.

tails of their plots. (In this chapter, I won't talk much about the object-based interface.) Matplotlib's default plots also look uncommonly beautiful, which was the intention of the package's primary author, John Hunter. Finally, matplotlib has a broad user community from many disciplines, so a lot of people contribute to it and templates/examples exist for many different kinds of plots.

The submodule pyplot defines the functional interface for matplotlib. Pyplot is often imported by:

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

Unless otherwise stated, you may assume in the examples in this chapter that the above import has been done prior to any matplotlib calls being run.

Do the online pyplot tutorial. It's very good!

The online pyplot tutorial is very good. In this chapter, we'll cover only a few of the topics found in there; I encourage you to go through it all on your own: http://matplotlib.sourceforge.net/users/pyplot_tutorial.html. The online gallery of examples is also very illuminating: http://matplotlib.sourceforge.net/gallery.html.

9.2 Basic line plots

Plot makes plots and show visualizes them.

Line plots are created by the pyplot `plot` function. Once created, matplotlib keeps track of what plot is the "current" plot. Subsequent commands (e.g., to make a label) are applied to the current plot.

The `show` function visualizes (i.e., displays) the plot to screen. If you have more than one figure, call `show` after all plots are defined to visualize all the plots at once. Consider the following example:

Example 54 (Your first line plot):

Type in this example into a file and run it in the Python interpreter:

```
1 import matplotlib.pyplot as plt
2 plt.plot([1, 2, 3, 4], [1, 2.1, 1.8, 4.3])
3 plt.axis([0, 8, -2, 7])
4 plt.xlabel('Automatic Range')
5 plt.ylabel('Made-up Numbers')
6 plt.show()
```

What did you get? Based on what's output, what do you think each of the commands do?

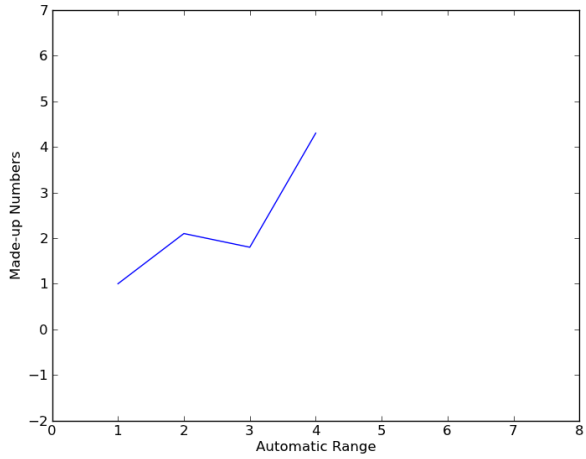


Figure 9.1: Graph created by the code in Example 54.

Solution and discussion: You should have obtained a plot like the one shown in Figure 9.1.

Line 2 of the code creates the plot, and the two list input arguments provide the *x*- and *y*-values, respectively. (I could have used NumPy arrays instead of lists as inputs.) The `axis` function in line 3 gives the range for the *x*- and *y*-axes, with the first two elements of the input parameter list giving the lower and upper bounds of the *x*-axis and the last two elements giving the lower and upper bounds of the *y*-axis. Lines 4 and 5 label the *x*- and *y*-axes, respectively, and the `show` function displays the graph on the screen.

9.2.1 Controlling line and marker formatting

To control line and marker features, you can use the appropriate keyword input parameters with the `plot` function, e.g.:

Controlling linestyle, markers, etc.

```
plt.plot([1, 2, 3, 4], [1, 2.1, 1.8, 4.3],
         linestyle='--', linewidth=5.0,
         marker='*', markersize=20.0,
         markeredgewidth=2.0,
         markerfacecolor='w')
```

Note how `linestyle`, `marker`, and `markerfacecolor` use special string codes to specify the line and marker type and formatting. The `plot` call above uses a dashed line and a white star for the marker. Linewidth, marker size, and marker edge width are in points.

Instead of using keyword input parameters, you can also specify line color and type and marker color and type as a string third argument, e.g.:

```
plt.plot([1, 2, 3, 4], [1, 2.1, 1.8, 4.3], 'r*--')
```

Notice that this third argument contains *all* the codes to specify line color, line type, marker color, and marker type. That is to say, all these codes can be specified in one string. In the above example, the color of the marker and connecting line is set to red, the marker is set to star, and the linestyle is set to dashed. (The marker edge color is still the default, black, however.)

Line and marker property listings.

Tables 9.1 and 9.2 list some of the basic linestyles and marker codes. For a more complete list of linestyles, marker codes, and basically all the line and marker properties that can possibly be set, see the following web pages:

- Linestyles: http://matplotlib.sourceforge.net/api/artist_api.html#matplotlib.lines.Line2D.set_linestyle
- Marker symbol types: http://matplotlib.sourceforge.net/api/artist_api.html#matplotlib.lines.Line2D.set_marker.
- Line and marker properties: http://matplotlib.sourceforge.net/api/artist_api.html#matplotlib.lines.Line2D.

Table 9.3 lists some of the color codes available in pyplot.

9.2.2 Annotation and adjusting the font size of labels

Annotation and font size.

We introduced the `xlabel` and `ylabel` functions in Example 54 to annotate the x- and y-axes, respectively. To place a title at the top of the plot, use the `title` function, whose basic syntax is the same as `xlabel` and `ylabel`. General annotation uses the `text` function, whose syntax is:

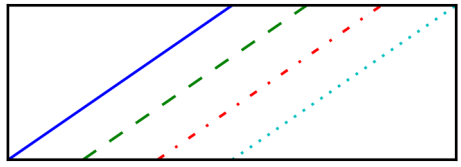
`plt.text(`<*x-location*>`,` <*y-location*>`,` <*string to write*>`)`

The x- and y-locations are, by default, in terms of **data coordinates.** For all four functions (`xlabel`, `ylabel`, `title`, and `text`), font size is controlled by the `size` keyword input parameter. When set to a floating point value, `size` specifies the size of the text in points.

Using LaTeX to annotate plots.

Here's one cool feature: matplotlib gives you the ability to use LaTeX to render text! See http://matplotlib.sourceforge.net/users/usetex.html for details.

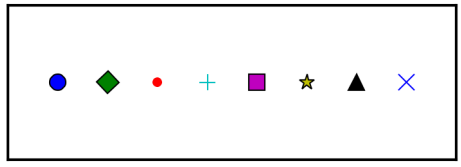
Linestyle	String Code
Solid line	`-`
Single dashed line	`--`
Single dashed-dot line	`-.`
Dotted line	`:`



```
1  import numpy as N
2  import matplotlib.pyplot as plt
3  plt.figure(1, figsize=(3,1))
4  plt.plot( N.arange(4),   N.arange(4), '-',  \
5            N.arange(4)+1, N.arange(4), '--', \
6            N.arange(4)+2, N.arange(4), '-.', \
7            N.arange(4)+3, N.arange(4), ':' )
8  plt.gca().axes.get_xaxis().set_visible(False)
9  plt.gca().axes.get_yaxis().set_visible(False)
10 plt.savefig('pyplot_linestyles.png', dpi=300)
```

Table 9.1: Some linestyle codes in pyplot, a high-resolution line plot showing the lines generated by the linestyle codes, and the code to generate the plot. Lines 8–9 turn-off the x- and y-axis tick marks and labels (see "matplotlib.pyplot.gca," http://matplotlib.sourceforge.net/api/pyplot_api.html and http://stackoverflow.com/a/2176591, both accessed August 13, 2012). A full explanation of these lines is beyond the scope of this book; please see the sources for more information. Note `show` is not called since I only want a file version of the plot.

Marker	String Code
Circle	`o`
Diamond	`D`
Point	`.`
Plus	`+`
Square	`s`
Star	`*`
Up Triangle	`^`
X	`x`



```
1  import numpy as N
2  import matplotlib.pyplot as plt
3  plt.figure(1, figsize=(3,1))
4  plt.plot( 1, 1, 'o',  \
5            2, 1, 'D', \
6            3, 1, '.', \
7            4, 1, '+', \
8            5, 1, 's', \
9            6, 1, '*', \
10           7, 1, '^', \
11           8, 1, 'x' )
12 plt.axis([0, 9, 0, 2])
13 plt.gca().axes.get_xaxis().set_visible(False)
14 plt.gca().axes.get_yaxis().set_visible(False)
15 plt.savefig('pyplot_markers.png', dpi=300)
```

Table 9.2: Some marker codes in pyplot, a high-resolution line plot showing the markers generated by the marker codes, and the code to generate the plot. Lines 12–13 turn-off the x- and y-axis tick marks and labels. See Table 9.1 for sources and more information.

Color	String Code
Black	`k`
Blue	`b`
Green	`g`
Red	`r`
White	`w`

Table 9.3: Some color codes in pyplot. See http://matplotlib.sourceforge.net/api/colors_api.html for a full list of the built-in colors codes as well as for ways to access other colors.

Example 55 (Annotation and font size):
Consider this code:

```
plt.plot([1, 2, 3, 4], [1, 2.1, 1.8, 4.3])
plt.xlabel('Automatic Range')
plt.ylabel('Made-up Numbers')
plt.title('Zeroth Plot', size=36.0)
plt.text(2.5, 2.0, 'My cool label', size=18.0)
plt.show()
```

What does this code do?

Solution and discussion: The above code produces a graph like the one in Figure 9.2. (Note that I resized that graph to fit it nicely on the page, so the text sizes as shown may not be equal to the values given in `size`.)

9.2.3 Plotting multiple figures and curves

Multiple independent figures.

If you have have multiple independent figures (not multiple curves on one plot), call the `figure` function before you call `plot` to label the figure accordingly. A subsequent call to that figure's number makes that figure current. For instance:

Example 56 (Line plots of multiple independent figures):
Consider this code:

```
1 plt.figure(3)
2 plt.plot([5, 6, 7, 8], [1, 1.8, -0.4, 4.3],
3          marker='o')
4 plt.figure(4)
5 plt.plot([0.1, 0.2, 0.3, 0.4], [8, -2, 5.3, 4.2],
6          linestyle='-.')
7 plt.figure(3)
8 plt.title('First Plot')
```

What does this code do?

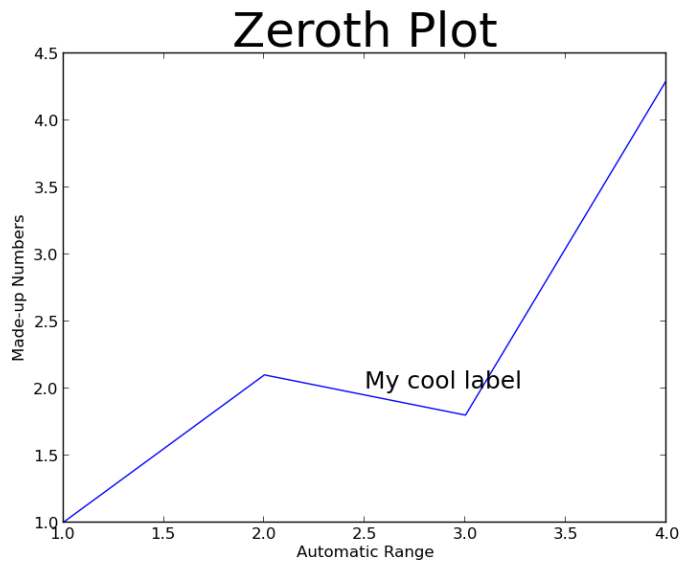


Figure 9.2: Graph created by the code in Example 55.

Solution and discussion: Line 1 creates a figure and gives it the name "3". Lines 2–3 (which is a single logical line to the interpreter) makes a line plot with a circle as the marker to the figure named "3". Line 4 creates a figure named "4", and lines 5–6 make a line plot with a dash-dot linestyle to that figure. Line 7 makes figure "3" the current plot again, and the final line adds a title to figure "3".

To plot multiple curves on a single plot, you can string the set of three arguments (*x*-locations, *y*-locations, and line/marker properties) for each plot one right after the other. For instance:

Multiple curves on one plot.

Example 57 (Line plot of multiple curves on one figure):

Consider this code:

```
plt.plot([0, 1, 2, 3], [1, 2, 3, 4], '--o',
         [1, 3, 5, 9], [8, -2, 5.3, 4.2], '-D')
```

What does it do?

Solution and discussion: The first three arguments specify the x- and y-locations of the first curve, which will be plot using a dashed line and a circle as the marker. The second three arguments specify the x- and y-locations of the second curve, which will be plot with a solid line and a diamond as the marker. Both curves will be on the same figure.

9.2.4 Adjusting the plot size

Adjusting plot size.

One easy way of adjusting the plot size is to set the `figsize` and `dpi` keyword input parameters in the `figure` command.[2] For instance, this call to `figure`:

```
plt.figure(1, figsize=(3,1), dpi=300)
```

before the call to the `plot` command, will make figure "1" three inches wide and one inch high, with a resolution of 300 dots per inch (dpi). The plot associated with Table 9.1 shows a code and plot example that explicitly specifies the `figsize` keyword.

9.2.5 Saving figures to a file

Save figure.

To write the plot out to a file, you can use the `savefig` function. For example, to write out the current figure to a PNG file called *testplot.png*, at 300 dpi, type:

```
plt.savefig('testplot.png', dpi=300)
```

Resolution and figure size in `figure` vs. `savefig`.

Here we specify an output resolution using the optional `dpi` keyword parameter; if left out, the matplotlib default resolution will be used. Note that it is not enough for you to set `dpi` in your `figure` command to get an output file at a specific resolution. The `dpi` setting in `figure` will control what resolution `show` displays at while the `dpi` setting in `savefig` will control the output file's resolution; however, the `figsize` parameter in `figure` controls the figure size for both `show` and `savefig`.

You can also save figures to a file using the GUI save button that is part of the plot window displayed on the screen when you execute the `show` function. If you save the plot using the save button, it will save at the default

[2] http://stackoverflow.com/a/638443 (accessed August 13, 2012).

resolution, even if you specify a different resolution in your `figure` command; use `savefig` if you want to write out your file at a specific resolution.

Most of the code for the examples in this section (9.2) are found in the file *example-viz-line.py* in *course_files/code_files*.

9.3 Exercise on basic line plots

▷ **Exercise 27 (Line plot of a surface air temperature timeseries):**

Read in the monthly mean surface/near-surface air temperature and the time axis from the provided NCEP/NCAR Reanalysis 1 netCDF dataset. (The example data is in *course_files/datasets* in the file *air.mon.mean.nc*.) Extract a timeseries at one location (any location) on the Earth and plot the first 100 data points of air temperature vs. time. Annotate appropriately. Write the plot out to a PNG file.

Solution and discussion: Here's my solution. The plotting section using matplotlib starts with line 11:

```
1  import Scientific.IO.NetCDF as S
2  import matplotlib.pyplot as plt
3
4  fileobj = S.NetCDFFile('air.mon.mean.nc', mode='r')
5  T_arctic = fileobj.variables['air'].getValue()[0:100,0,0]
6  T_units = fileobj.variables['air'].units
7  time = fileobj.variables['time'].getValue()[0:100]
8  time_units = fileobj.variables['time'].units
9  fileobj.close()
10
11 plt.plot(time, T_arctic)
12 plt.xlabel('Time [' + time_units + ']')
13 plt.ylabel('Temperature [' + T_units + ']')
14
15 plt.savefig('exercise-T-line.png')
16 plt.show()
```

This code makes a plot like the one in Figure 9.3. Note how string concatenation, coupled with each variable's `units` metadata values in the netCDF file, make it easy to annotate the plot with the units.

On some installations, if you call `show` before `savefig`, things do not always write correctly to the file, so in my code I call `savefig` first, just

Call `savefig` before `show`.

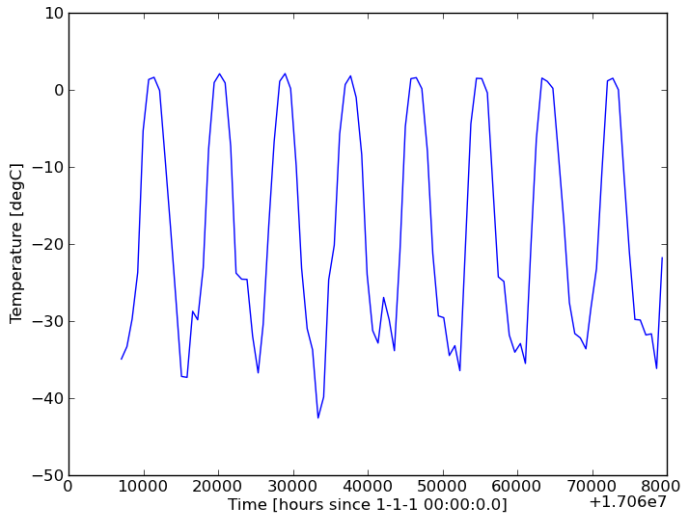


Figure 9.3: Graph created by the solution code to Exercise 27.

to be safe. Of course, if you only want the plot as a file, just use `savefig` without calling `show`.

This code is in the file *exercise-viz-line.py* in the *course_files/code_files* subdirectory.

9.4 Basic contour plots

A number of the aspects of plotting (e.g., saving a figure to a file, etc.) work for contour plots exactly the same as for line plots. In this section, we won't rehash those common aspects.

Contour plots using `contour`.

Contour plots are created by matplotlib's `contour` function. A basic contour plot is generated by:

```
plt.contour(X, Y, Z, nlevels)
```

where `Z` is a 2-D array of the values to contour with and `X` and `Y` are the x- and y-locations, respectively, of the `Z` values (`X` and `Y` can be 2-D arrays or 1-D vectors, the latter if the grid is regular). The optional `nlevels` parameter tells how many automatically generated contour levels to make.

The `nlevels` parameter.

The `levels` keyword controls exactly which levels to draw contours at, e.g.:

```
plt.contour(X, Y, Z, levels=[-2, -1, 0, 1, 2])
```

To make dashed negative contours, set the `colors` keyword to `'k'`:

```
plt.contour(X, Y, Z, colors='k')
```

Making negative contours dashed.

This setting makes the all the contours black. Matplotlib then renders the negative contours using the value of an "rc setting" that defaults to dashed.[3]

While you can do nice default contour maps just by calling the `contour` function, a number of contour map functions take a contour map object as input. Thus, it's better to save the map to a variable:

Saving the contour map to a variable so you can pass it into other formatting functions.

```
mymap = plt.contour(X, Y, Z)
```

Then, to add contour labels, for instance, use the `clabel` function (this is a function that asks for a contour map object as input):

```
mymap = plt.contour(X, Y, Z)
plt.clabel(mymap, fontsize=12)
```

The optional keyword `fontsize` sets the font size (in points).

For filled contours, use the `contourf` function. The color maps available for filled contour maps are attributes of the `pyplot` module attribute `cm`. You specify which color map to use via the `cmap` keyword:

Making filled contour maps.

```
mymap = plt.contourf(X, Y, Z, cmap=plt.cm.RdBu)
```

A list of predefined color maps is located at http://www.scipy.org/Cookbook/Matplotlib/Show_colormaps. To add a color bar that shows the scale of the plot, make a call to `colorbar` that uses the filled contour plot object as input:

List of predefined color maps and adding color bars.

```
plt.colorbar(mymap, orientation='horizontal',
             levels=[-2, -1, 0, 1, 2])
```

The `orientation` keyword specifies the orientation of the color bar, as you'd expect ☺. The `levels` keyword is set to a list that specifies what levels to label on the color bar.

To make a contour map that's both lined and filled, make a filled contour map call then a line contour map call (or vice versa), e.g.:

[3]The rc setting is `contour.negative_linestyle` and can be changed in the *matplotlibrc* file. See http://matplotlib.sourceforge.net/users/customizing.html for details (accessed August 17, 2012).

```
plt.contourf(lonall, latall, T_time0, 10,
             cmap=plt.cm.Reds)
plt.contour(lonall, latall, T_time0, 10,
             colors='k')
```

Both contour maps will be placed on the same figure.

Making wind barbs.

Lastly, atmospheric scientists are often interested in wind barbs: these are generated with the `barbs` method of objects generated by the matplotlib `subplot` function. See http://matplotlib.sourceforge.net/examples/pylab_examples/barb_demo.html for an example.

9.5 Exercise on basic contour plots

▷ **Exercise 28 (Contour plot of surface air temperature):**

Read in the monthly mean surface/near-surface air temperature from the NCEP/NCAR Reanalysis 1 netCDF dataset provided. Also read in the latitude and longitude vectors from the dataset. Extract a single timeslice of the temperature and plot a contour map. Annotate appropriately. Write the plot out to a PNG file. Hint: The NumPy function `meshgrid` can be your friend (see Example 32), though it may not be necessary.

Solution and discussion: Here's my solution:

```
1  import numpy as N
2  import Scientific.IO.NetCDF as S
3  import matplotlib.pyplot as plt
4
5  fileobj = S.NetCDFFile('air.mon.mean.nc', mode='r')
6  T_time0 = fileobj.variables['air'].getValue()[0,:,:]
7  T_units = fileobj.variables['air'].units
8  lon = fileobj.variables['lon'].getValue()
9  lon_units = fileobj.variables['lon'].units
10 lat = fileobj.variables['lat'].getValue()
11 lat_units = fileobj.variables['lat'].units
12 fileobj.close()
13
14 [lonall, latall] = N.meshgrid(lon, lat)
15
16 mymapf = plt.contourf(lonall, latall, T_time0, 10,
17                       cmap=plt.cm.Reds)
18 mymap = plt.contour(lonall, latall, T_time0, 10,
19                     colors='k')
20 plt.clabel(mymap, fontsize=12)
21 plt.axis([0, 360, -90, 90])
22 plt.xlabel('Longitude [' + lon_units + ']')
23 plt.ylabel('Latitude [' + lat_units + ']')
24 plt.colorbar(mymapf, orientation='horizontal')
25
26 plt.savefig('exercise-T-contour.png')
27 plt.show()
```

Lines 5–12 read in the data from the netCDF file. In line 6, we obtain the 2-D slab of surface air temperature at time zero and assign it to the array variable `T_time0`. The `lon` and `lat` variables, created in lines 8 and 10, are 1-D vectors. To be on the safe side, we want 2-D versions of these vectors, which we create in line 14 using `meshgrid` and assign as `lonall` and `latall`. Line 16 specifies that we plot the contour plot with 10 contour intervals, and in line 17, we specify a red gradient color map to use for the contour interval filling.

In lines 18–19, we create a contour map of lines only, to superimpose on top of the filled contour plot. We assign the result of the `contour` call to `mymap`, which we'll use with the `clabel` function in line 20 (that generates

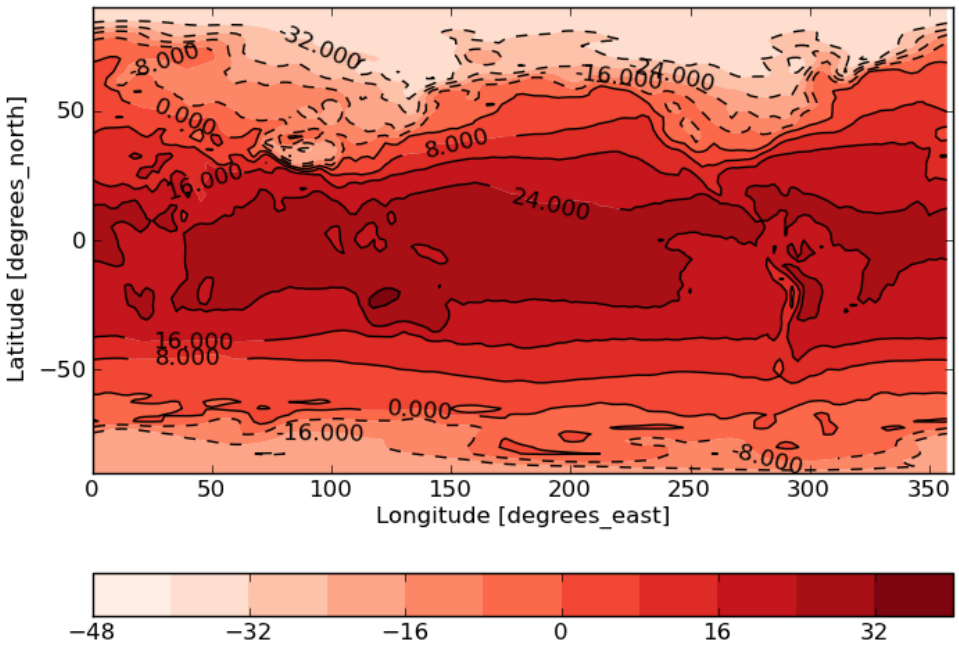


Figure 9.4: Graph created by the solution code to Exercise 28.

the contour labels). Line 21 specifies the axes range using the `axis` function, labeling occurs in lines 22–23, the color map in line 24, and the last two lines save the figure to a file and display the figure on the screen.

Note how the results of both the `contourf` and `contour` calls need to be assigned to objects which are used by the `colorbar` and `clabel` functions as input (in lines 24 and 20, respectively). Also note that on some installations, if you call `show` before `savefig`, things do not always write correctly to the file, so in my code I call `savefig` first, just to be safe.

The code generates a plot like the one in Figure 9.4. This code is in the file *exercise-viz-contour.py* in the *code_files* subdirectory of the *course_files* directory.

9.6 Superimposing a map

The Basemap package and map projections.

Often, AOS users will want to superimpose a map of some sort (e.g., continental outlines) onto a contour plot. To do so, you need to use the Basemap package, which handles map projection setup for matplotlib. Note, however, that Basemap is a separate package from matplotlib, is distributed under a

different license, and often has to be installed separately.[4] For many operating system environments, you need to build Basemap from source. (It is, however, a Debian package in Ubuntu 12.04.)[5] If you have the full version of the Enthought Python Distribution (EPD), Basemap is installed for you; Basemap, however, is not part of EPD Free.

To create a map and then superimpose a contour plot on the map, follow these steps:

Steps to creating a contour plot on a map.

- Instantiate an instance of the `Basemap` class.
- Use methods of that instance to draw continents, etc.
- Map the 2-D latitude and longitude coordinates of your dataset to the coordinates in the map projection by calling your `Basemap` instance with the dataset coordinates as input arguments.
- Make your contour plot using regular matplotlib commands.

This will become much clearer with an example:

Example 58 (Contour plot on a cylindrical projection map limited to the global Tropics):

Assume you have three 2-D arrays as input: `data`, which is the data being contoured, and `lonall` and `latall`, which give the longitudes and latitudes (in degrees), respectively, of the elements of `data`. The code to create the contour plot and the map is:

```
1  import numpy as N
2  import matplotlib.pyplot as plt
3  import mpl_toolkits.basemap as bm
4  mapproj = bm.Basemap(projection='cyl',
5                       llcrnrlat=-20.0, llcrnrlon=-180.0,
6                       urcrnrlat=20.0, urcrnrlon=180.0)
7  mapproj.drawcoastlines()
8  mapproj.drawparallels(N.array([-20, -10, 0, 10, 20]),
9                        labels=[1,0,0,0])
10 mapproj.drawmeridians(N.array([-180, -90, 0, 90, 180]),
11                       labels=[0,0,0,1])
12 lonproj, latproj = mapproj(lonall, latall)
13 plt.contour(lonproj, latproj, data)
```

[4] See http://sourceforge.net/projects/matplotlib/files/matplotlib-toolkits for the downloads (accessed August 16, 2012).

[5] See http://packages.ubuntu.com/en/precise/python-mpltoolkits.basemap for a description of the package (accessed August 16, 2012).

In lines 4–6, what do you think the keywords do? The `labels` keywords in lines 9 and 11?

Solution and discussion: The first three lines of the code imports the needed packages. Notice that Basemap is normally found as a subpackage of the mpl_toolkits package. Lines 4–6 create `mapproj`, a Basemap instance. The keyword input parameters set the projection (cylindrical) and give the "corner" latitude and longitude values of the map: `llcrnrlat` is the lower-left corner's latitude, `urcrnrlon` is the upper-right corner's longitude, etc.

Basemap map projection parameters.

Once the `mapproj` Basemap instance is created, we use methods attached to the instance to draw coastlines (line 7), latitude lines (lines 8–9), and longitude lines (lines 10–11). The positional input argument for `drawparallels` and `drawmeridians` specifies the locations at which to draw the latitude and longitude lines. The `labels` keyword is set to a 4-element list of integers that specify where to draw the labels. If the first element is set to 1, labels are drawn to the left of the plot, if the second element is set to 1, labels are drawn to the right of the plot, and the third and fourth elements control the top and bottom labels, respectively. Thus, line 9 specifies latitude line labels on the left side of the plot (only) and line 11 specifies longitude line labels at the bottom of the plot (only).

Basemap instance methods create coastlines, etc.

Line 12 calls the `mapproj` instance as if it were a function. The 2-D longitude and latitude arrays are passed into the call. Two 2-D arrays are returned that specify the longitude and latitude values, but altered to account for the projection, that can then be passed into a contour plot call, along with the data to contour, as is done in line 13.

We haven't really talked about calling object instances, but indeed, we can define a special method `__call__` in a class that will be executed when you call an instance (that is, treat the instance like it were a function). That's essentially what is happening in line 12. Note that calling an instance is not the same as **instantiating** the instance!

Calling object instances.

Basemap supports many different types of projections, and the input parameters when instantiating a `Basemap` object will change depending on the projection you specify. The SciPy Cookbook entry for Basemap gives a nice introduction: http://www.scipy.org/Cookbook/Matplotlib/Maps. Also see the Basemap documentation: http://matplotlib.github.com/basemap.

9.7 Exercise on superimposing a map

▷ **Exercise 29 (Contour plot of surface air temperature with continental outlines):**

Redo Exercise 28 but superimpose a map with continental outlines on it.

Solution and discussion: To save space, I only provide the core of my solution here. The full code is in the file *exercise-viz-basemap.py* in the *code_files* subdirectory of the *course_files* directory:

```
1  mapproj = bm.Basemap(projection='cyl',
2                       llcrnrlat=-90.0, llcrnrlon=0.0,
3                       urcrnrlat=90.0, urcrnrlon=360.0)
4  mapproj.drawcoastlines()
5  mapproj.drawparallels(N.array([-90, -45, 0, 45, 90]),
6                        labels=[1,0,0,0])
7  mapproj.drawmeridians(N.array([0, 90, 180, 270, 360]),
8                        labels=[0,0,0,1])
9  lonall, latall = mapproj(lon2d, lat2d)
10
11 mymapf = plt.contourf(lonall, latall, T_time0, 10,
12                        cmap=plt.cm.Reds)
13 mymap = plt.contour(lonall, latall, T_time0, 10,
14                      colors='k')
15 plt.clabel(mymap, fontsize=12)
16 plt.title('Air Temperature [' + T_units + ']')
17 plt.colorbar(mymapf, orientation='horizontal')
18
19 plt.savefig('exercise-T-basemap.png')
20 plt.show()
```

This code makes a plot like the one in Figure 9.5.

This code is essentially a combination of Exercise 28 and Example 58. The one difference is in lines 2–3 of this exercise, where I specify the longitude corner keywords by the range 0 to 360 degrees instead of −180 to 180 degrees (as in Example 58). Since the data starts with 0 degrees longitude, I decided to put that in the lower-left corner. But referencing longitude by negative longitude values works fine in Basemap.

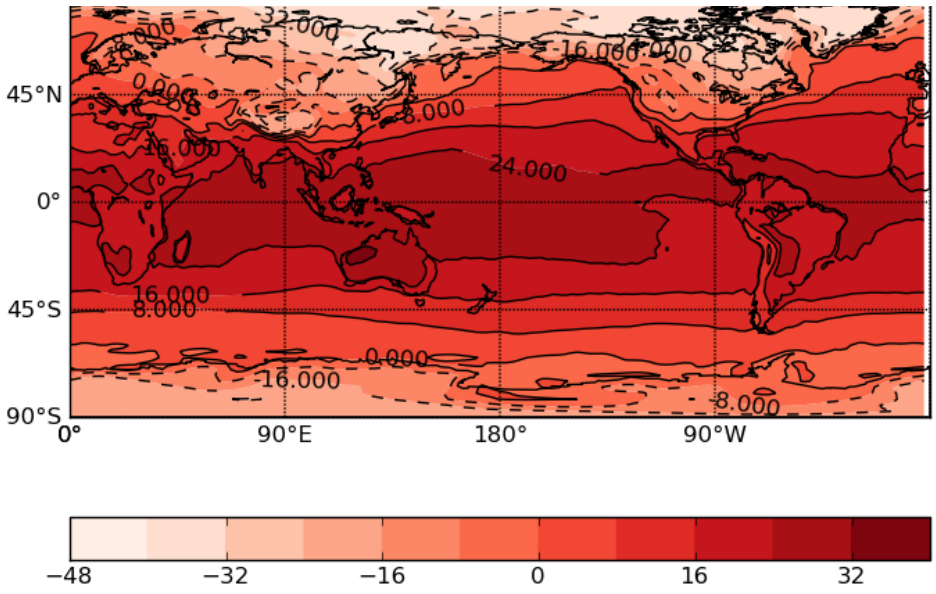


Figure 9.5: Graph created by the solution code to Exercise 29.

9.8 Summary

Basic Python visualization using matplotlib is very much like what you're probably used to using in Matlab and IDL. Coupled with the Basemap module, matplotlib enables you to do the basic line and contour plots that form the bread-and-butter of AOS visualization. Details on matplotlib are found at http://matplotlib.sourceforge.net.

Other Python AOS visualization packages.

This chapter, of course, only scratches the surface regarding Python visualization. The PyAOS website keeps a list of packages that may be of interest to AOS users (http://pyaos.johnny-lin.com/?page_id=117). Some packages of note include:

- ParaView: Analysis and visualization package for very large datasets.
- PyGrADS: Python interface to GrADS.
- PyNGL: All of the basic functionality of NCAR Graphics in a Python interface.
- UV-CDAT: Ultrascale Visualization-Climate Data Analysis Tools.
- VisTrails: Visualization tool with workflow management that tracks the provenance of the visualization and data.

- VPython: An easy-to-use 3-D visualization and animation environment.

Unlike proprietary languages which have only one visualization engine integrated with the language, Python's open-source nature permits radical experimentation with different methods of implementing visualization tools. This does create some confusion, and can make installation a bear, but it also provides you the right visualization tool for your specific needs. Have a very large dataset? Try ParaView. Is workflow provenance integration vital to you? Give VisTrails and UV-CDAT a shot. Want to do really simple 3-D animation for educational modeling? VPython is a snap. But for many everyday visualization tasks, matplotlib works fine.