7 Matplotlib

Matplotlib is probably the most popular Python package for plotting data. It can be used through the procedural interface `pylab` in very quick scripts to produce simple visualizations of data (see Chapter 3) but, as described in this chapter, with care it can also produce high-quality figures for journal articles, books and other publications. Although there is some limited functionality for producing three-dimensional plots (see Section 7.2.3), it is primarily a two-dimensional plotting library.

7.1 Matplotlib basics

Matplotlib is a large package organized in a hierarchy: at the highest level is the `matplotlib.pyplot` module. This provides a "state-machine environment" with a similar interface to MATLAB and allows the user to add plot elements (data points, lines, annotations, etc.) through simple function calls. This is the interface used by `pylab`, which was introduced in Chapter 3.

At a lower level, which allows more advanced and customizable use, Matplotlib has an object-oriented interface that allows one to create a *figure* object to which one or more *axes* objects are attached. Most plotting, annotation and customization then occurs through these axes objects. This is this approach we adopt in this chapter.

To use Matplotlib in this way, we use the following recommended imports:

```
import matplotlib.pyplot as plt
import numpy as np
```

7.1.1 Basic figures

Plotting on a single axes object

The top-level object, containing all the elements of a plot is called `Figure`. To create a figure object, call `plt.figure`. No arguments are necessary, but optional customization can be specified by setting the values described in Table 7.1. For example,

```
In [x]: # a default figure, with title "Figure 1"
In [x]: fig = plt.figure()

In [x]: # a small figure with red background
In [x]: fig = plt.figure('Population density', figsize=(4.5, 2.),
  ....:                  facecolor='red')
```

Table 7.1 Arguments to `plt.figure`

Argument	Description
`num`	An identifier for the figure – if none is provided, an integer, starting at 1, is used and incremented with each figure created. Alternatively using a string will set the window title to that string when the figure is displayed with `plt.show()`.
`figsize`	A tuple of figure `(width, height)`, unfortunately in inches.
`dpi`	Figure resolution in dots-per-inch.
`facecolor`	Figure background color.
`edgecolor`	Figure border color.

To actually plot data, we need to create an `Axes` object – a region of the figure containing the axes, tick-marks, labels, plot lines and markers, and so on. The simplest figure, consisting of a single `Axes` object, is created and returned with

```
In [x]: ax = fig.add_subplot(111)
```

The argument `111` here is a commonly used abbreviation for the tuple `(1,1,1)` specifying subplot `1` of a figure with `1` row and `1` column of subplots (see Section 7.1.3). The `Axes` object, `ax`, is the one on which we can actually plot the data with `ax.plot`. The essential features of this `plot` method were described in Chapter 3. Here, however, we note that the `plot` method actually returns a `list` of objects representing the plotted lines. In its simplest usage, only a single line is plotted, and so this list consists of one `Line2D` object that we may assign to a variable if desired. As a full example, consider the following comparison of the catenary $y = \cosh(x)$ and its parabolic approximation, $y = 1 + x^2/2$.

```
import matplotlib.pyplot as plt
import numpy as np

fig = plt.figure()
ax = fig.add_subplot(111)

x = np.linspace(-2, 2, 1000)
line_cosh, = ax.plot(x, np.cosh(x))
line_quad, = ax.plot(x, x**2 / 2)

plt.show()
```

❶ Note the syntax `line_cosh, = ...` to assign the returned line object to the variable `line_cosh` rather than the `list` containing that object.

The two plotted lines are shown in Figure 7.1.

Plot limits

By default, Matplotlib plots all of the data passed to `plot` and sets the axis limits accordingly. To set the axis limits to something else, use the `ax.set_xlim` and `ax.set_ylim` methods. Either both limits can be set or an individual limit can be set with the arguments `left`, `right` (or `xmin`, `xmax`) and `bottom`, `top` (or `ymin`, `ymax`). Unspecified limits are left unchanged. For example,

Table 7.2 Matplotlib line styles

	(no line)
`-`	solid
`--`	dashed
`:`	dotted
`-.`	dash-dot

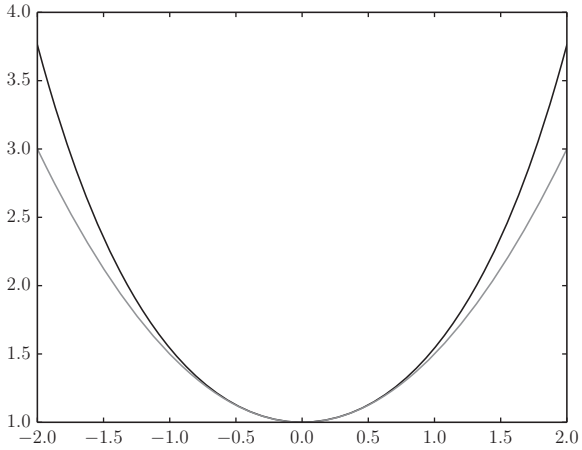

Figure 7.1 A simple plot of two lines on a single `Axes` object.

```
x = np.linspace(-3,3,1000)
y = x**3 + 2 * x**2 - x - 1
fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(x,y)

ax.set_xlim(-1,2)        # x-limits are -1 to 2
ax.set_ylim(bottom=0)    # ymin=0: plot will be "clipped" at the bottom
```

If `bottom` is greater than `top` or `right` less than `left`, the corresponding axis will be reversed; that is, values on this axis will *decrease* from left to right (or from bottom to top) (see Exercise P7.1.5).

If you wish to invert the axis direction without changing the limit values, the method calls `ax.invert_xaxis()` and `ax.invert_yaxis()` will do that for you.

Line styles, markers and colors

As with `pylab`, the plot style can be specified by passing extra arguments to the `plot()` method. The default line style is a solid, 1.0 pt weight line in a color determined by the order in which it is added to the plot.

An alternative line style can be selected from the predefined options with the `linestyle` (or simply `ls`) argument. Possible string values to pass to this argument (including the empty string for plotting no line) are shown in Table 7.2.

Table 7.3 Matplotlib colour code letters

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

Further customization is possible by setting the `dashes` argument to a sequence of values describing the repeated dash pattern in points. For example, `dashes=[2, 4, 8, 4, 2, 4]` represents a pattern of dot (2 pts), space (4 pts), dash (8 pts), space (4 pts), dot (2 pts), space (4 pts) to be repeated as the line style. Equivalently, one can call a plotted line's `set_dashes` method, as in the following code snippet:

```
x = np.linspace(-np.pi, np.pi, 1000)
line, = plt.plot(x, np.sin(x))
line.set_dashes([2, 4, 8, 4, 2, 4])     # dot-dash-dot
```

The line weight is customized by setting the `lineweight` (or simply `lw`) argument to a number of points.

Line colors are specified with the `color` (or simply `c`) argument used in one of several ways:

- *string*: by letter or name, one of the values given in Table 7.3.
- *string*: by HTML 6-digit hex-string preceded by `'#'`, for example `'#ffff00'` is yellow.
- *string*: a string representation of a `float` between 0. and 1. (for example `'0.4'`) gives a gray-scale between black (0.) and white(1.).
- *tuple of floats* between 0. and 1.: RGB components, for example `(0.5, 0., 0.)` is a dark red color.

By default, the `Line2D` object created by calling `plot` on an `Axes` object does not include *markers*: symbols printed at each point on the plot. To add them, specify one of the single-character marker codes given in Table 7.4 using the `marker` argument

```
ax.plot(x, y, marker='v')   # downward pointing triangles
```

Other marker properties can be set with the arguments listed in Table 7.5.

Matplotlib markers can be further customized; see the documentation for details.[1]

Scatterplots

A typical two-dimensional scatterplot depicts the data as points on a Cartesian axes system. Sometimes there is no meaningful or helpful ordering to the data and so no

[1] http://matplotlib.org/api/markers_api.html.

Table 7.4 Some Matplotlib marker styles (single character sting codes)

Code	Marker	Description
`.`	·	point
`o`	○	circle
`+`	+	plus
`x`	×	x
`D`	◇	diamond
`v`	▽	(downward triangle)
`^`	△	(upward triangle)
`s`	□	square
`*`	★	star

Table 7.5 Matplotlib marker properties

Argument	Abbreviation	Description
`markersize`	`ms`	Marker size, in points
`markevery`		Set to a positive integer, N, to print a marker every N points; the default, `None`, prints a marker for every point
`markerfacecolor`	`mfc`	Fill color of the marker
`markeredgecolor`	`mec`	Edge color of the marker
`markeredgewidth`	`mew`	Edge width of the marker, in points

need to join data points by lines. The `pyplot.scatter` function creates a scatterplot. In addition to one-dimensional sequences of $x-$ and $y-$ data, as for `pyplot.plot`, the data point marker colors and sizes can be set individually by passing a sequence of appropriate values of the same length as the data to the arguments `s` and `c` respectively. The marker sizes are in points^2 (*points squared*) so that their *area* is proportional to the values passed to `s`. Manipulating the size of the markers is a common way of indicating a third dimension to the data, as in the following example.

Example E7.1 To explore the correlation between birth rate, life expectancy and per capita income, we may use a scatterplot. Note that the marker sizes are set in proportion to the countries' percapita GDP but have to be scaled a little so they don't get too large (see Figure 7.2).

Listing 7.1 Scatterplot of demographic data for eight countries

```
# eg7-scatter.py

import numpy as np
import matplotlib.pyplot as plt

countries = ['Brazil', 'Madagascar', 'S. Korea', 'United States',
             'Ethiopia', 'Pakistan', 'China', 'Belize']
# Birth rate per 1000 population
```

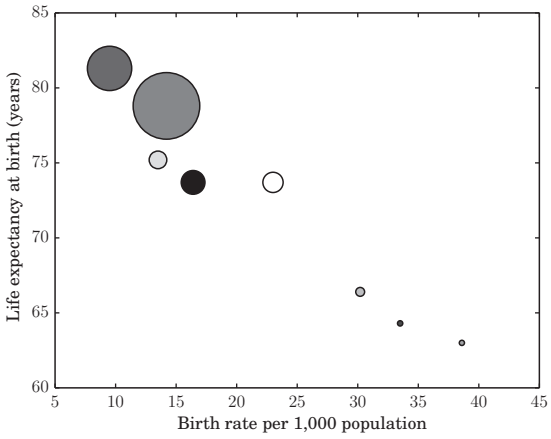

Figure 7.2 A scatterplot with variable marker sizes indicating each country's GDP.

```
birth_rate = [16.4, 33.5, 9.5, 14.2, 38.6, 30.2, 13.5, 23.0]
# Life expectancy at birth, years
life_expectancy = [73.7, 64.3, 81.3, 78.8, 63.0, 66.4, 75.2, 73.7]
# Per person income fixed to US Dollars in 2000
GDP = np.array([4800, 240, 16700, 37700, 230, 670, 2640, 3490])

fig = plt.figure()
ax = fig.add_subplot(111)
# Some arbitrary colors:
colors = range(len(countries))
ax.scatter(birth_rate, life_expectancy, c=colors, s=GDP/10)
ax.set_xlabel('Birth rate per 1000 population')
ax.set_ylabel('Life expectancy at birth (years)')

plt.show()
```

Gridlines

Gridlines are vertical (for the *x*-axis) and horizontal (for the *y*-axis) lines running across the plot to aid with locating the numerical values of data points. By default no gridlines are drawn, but they may be turned on by calling `grid` method on an `Axes` object (to add both horizontal and vertical gridlines) or the `xaxis` or `yaxis` objects of a given `Axes` (to select the gridlines to use). For example,

```
ax.yaxis.grid(True)  # Turn on horizontal gridlines
```

or

```
ax.grid(True)   # Turn on all gridlines
```

The line properties of the gridlines are set with the `linestyle`, `linewidth`, `color`, etc. arguments as for plot lines.

Two sorts of gridlines correspond to the major and minor tick marks (see below): these can be selected with the `which` argument, which takes the values `'major'`, `'minor'` and `'both'`. The default (if not specified) is `which='major'`.

```
ax.xaxis.grid(True, which='minor', c='b')   # Minor x-axis gridlines in blue
```

Log scales

By default, Matplotlib plots data on a linear scale. To set a logarithmic scale, call one or both of the following on your `Axes` object:

```
ax.set_xscale('log')
ax.set_yscale('log')
```

Base-10 logarithms are used by default, but the (integer) base can be set with the optional arguments `basex` or `basey`. Nonpositive values in the data will be masked as invalid by default. If you want negative values to be handled "symmetrically" with positive ones, such that $\log(-|x|) = -\log(|x|)$, then use `'symlog'` instead of `'log'`. See also Question 7.1.1.

Adding titles, labels and legends

Axis labels may be added to the subplot `Axes` object with `ax.set_xlabel` and `ax.set_ylabel`.

Plot line legend labels are defined by adding the `label` attribute to the `plt.plot` function call. However, the legend itself will not appear unless `legend` is called on the plot `Axes` object (e.g., with `ax.legend()`.) The appearance of the legend itself can be customized extensively, but the most common additional argument you may wish to pass to is `legend` is `loc`, defining the location of the legend on the plot (see Table 3.1).

There are two types of title you may want to give your figure: `fig.suptitle` adds a centered title to the entire figure, which may contain more than one subplot; `ax.title` adds a title to a single subplot.[2]

Example E7.2 The data read in from the file `eg7-marriage-ages.txt`, which can be downloaded from scipython.com/eg/aag, giving the median age at first marriage in the United States for 13 decades since 1890 are plotted by the program below. Grid lines are turned on for both axes with `ax.grid()`, and custom markers are used for the data points themselves (see Figure 7.3).

Listing 7.2 The median age at first marriage in the US over time

```
# eg7-marriage-ages.py
import numpy as np
import matplotlib.pyplot as plt

year, age_m, age_f = np.loadtxt('eg7-marriage-ages.txt', unpack=True,skiprows=3)
fig = plt.figure()
ax = fig.add_subplot(111)
```

[2] See the documentation at http://matplotlib.org/api/legend_api.html for more details.

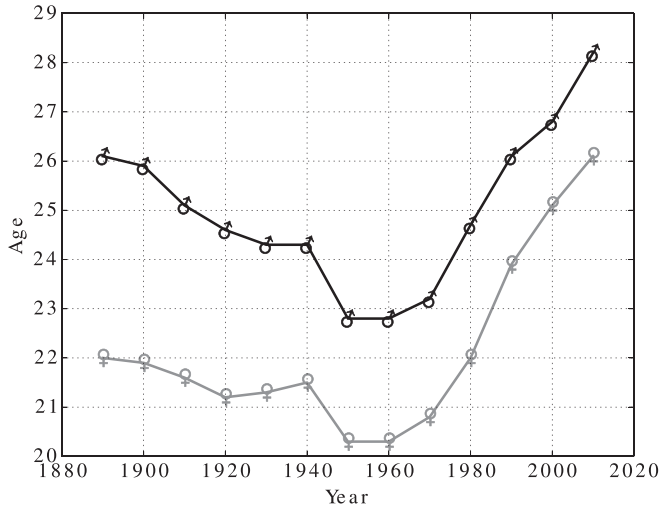

Figure 7.3 Median age at first marriage in the US, 1890–2010.

```
# Plot ages with male or female symbols as markers
ax.plot(year, age_m, marker='$\u2642$', markersize=14, c='blue', lw=2,
        mfc='blue', mec='blue')
ax.plot(year, age_f, marker='$\u2640$', markersize=14, c='magenta', lw=2,
        mfc='magenta', mec='magenta')
ax.grid()

ax.set_xlabel('Year')
ax.set_ylabel('Age')
ax.set_title('Median age at first marriage in the US, 1890 - 2010')

plt.show()
```

Example E7.3 The historical populations of five US cities are given in the files `boston.tsv`, `houston.tsv`, `detroit.tsv`, `san_jose.tsv`, `phoenix.tsv` as tab-separated columns of (year, population). They can be downloaded from scipython.com/eg/aaf.

The following program plots these data on one set of axes with a different line style for each.

Listing 7.3 The populations of five US cities over time

```
# eg7-populations.py
import matplotlib.pyplot as plt
import numpy as np

fig = plt.figure()
ax = fig.add_subplot(111)
```

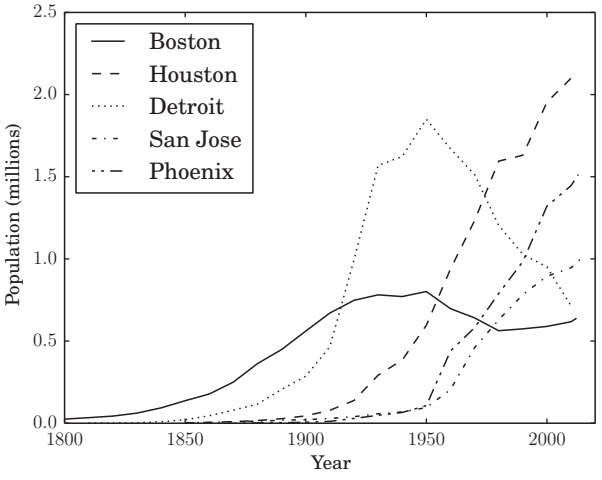

Figure 7.4 Population trends for five US cities.

```
cities = ['Boston', 'Houston', 'Detroit', 'San Jose', 'Phoenix']
# line styles: solid, dashes, dots, dash-dots, and dot-dot-dash
linestyles = [{'ls': '-'}, {'ls': '--'}, {'ls': ':'}, {'ls': '-.'},
              {'dashes': [2, 4, 2, 4, 8, 4]}]

for i, city in enumerate(cities):
    filename = '{}.tsv'.format(city.lower()).replace(' ', '_')
    yr, pop = np.loadtxt(filename, unpack=True)
    line, = ax.plot(yr, pop/1.e6, label=city, c='k', **linestyles[i])
ax.legend(loc='upper left')
ax.set_xlim(1800, 2020)
ax.set_xlabel('Year')
ax.set_ylabel('Population (millions)')
plt.show()
```

❶ Note how the city name is used to deduce the corresponding filename.

The plot produced is shown in Figure 7.4.

Font properties

The text elements of a plot (titles, legend, axis labels, etc.) can be customized with the arguments given in Table 7.6. For example,

```
ax.title('Plot Title', fontsize=18, fontname='Times New Roman', color='blue')
```

To use the same font properties for all text elements, it is easiest to set Matplotlib's `rc` settings using a dictionary of values. This involves a separate import from `pyplot` first:[3]

[3] It is also possible to edit Matplotlib's configuration file, `matplotlibrc`, to set many kinds of plot preferences: see http://matplotlib.org/users/customizing.html.

Table 7.6 Font property arguments for text elements of a plot

Argument	Description
`fontsize`	The size of the font in points (e.g., `12`, `16`)
`fontname`	The font name (e.g., `'Courier'`, `'Arial'`)
`family`	The font family (e.g., `'sans-serif'`, `'cursive'`, `'monospace'`)
`fontweight`	The font weight (e.g., `'normal'`, `'bold'`)
`fontstyle`	The font style (e.g., `'normal'`, `'italic'`)
`color`	Any Matplotlib color specifier (e.g., `'r'`, `'#ff00ff'`)

```
from matplotlib import rc
font_properties = {'family' : 'monospace',
                   'weight' : 'bold',
                   'size'   : 22}
❶ rc('font', **font_properties)
# All text will now be rendered in 22-point, bold monospace in plots
```

❶ Recall that the syntax `**kwargs` passes the (key, value) pairs of dictionary `kwargs` and passes them to a function as keyword arguments (see Section 4.2.2).

Tick marks

Matplotlib does its best to label representative values (*tick marks*) on each axis appropriately, but there are some occasions when you want to customize them, for example, to make the tick marks more or less frequent, or to label them differently.

Most commonly, one simply wants to set the tick mark values to a given sequence of values: this is accomplished by calling `ax.set_xticks` and `ax.set_yticks` on the `Axes` object of the plot. For example,

```
ax.set_xticks([0, 1, 3.5, 6.5, 15])
```

Note that the ticks do not have to be evenly spaced.

To replace the actual numbered labels, pass a sequence of strings of a suitable length to `ax.set_xticklabels` and `ax.set_yticklabels`, as in the following example.[4]

Example E7.4 The following program plots the exponential decay described by $y = Ne^{-t/\tau}$ labeled by lifetimes, ($n\tau$ for $n = 0, 1, \cdots$) such that after each lifetime the value of y falls by a factor of e. The plot is given as Figure 7.5.

Listing 7.4 Exponential decay illustrated in terms of lifetimes

```
# eg7-ticks-exp-decay.py
import numpy as np
import matplotlib.pyplot as plt

# Initial value of y at t=0, lifetime in s
N, tau = 10000, 28
```

[4] Note that setting the tick labels directly in this way decouples your plot from its data to some extent. An entire module, `matplotlib.ticker`, is devoted to the configuration of tick locating and formatting: its API is beyond the scope of this book but is well described at http://matplotlib.org/api/ticker_api.html.

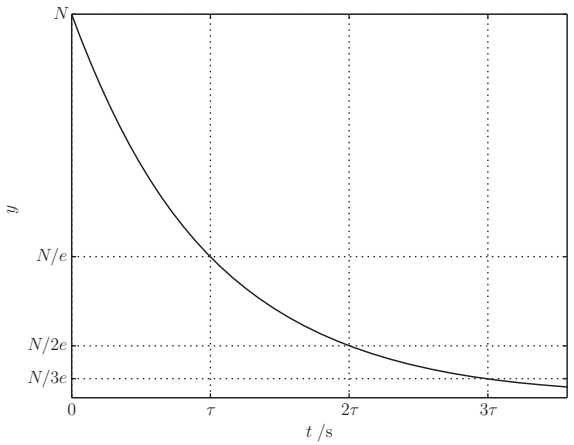

Figure 7.5 An exponential decay with customized tick labels.

```
# Maximum time to consider (s)
tmax = 100
# A suitable grid of time points, and the exponential decay itself
t = np.linspace(0, tmax, 1000)
y = N * np.exp(-t/tau)

fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(t, y)

# The number of lifetimes that fall within the plotted time interval
ntau = tmax // tau + 1
# xticks at 0, tau, 2*tau, ..., ntau*tau; yticks at the corresponding y-values
xticks = [i*tau for i in range(ntau)]
yticks = [N * np.exp(-i) for i in range(ntau)]
ax.set_xticks(xticks)
ax.set_yticks(yticks)

# xtick labels: 0, tau, 2tau, ...
❶ xtick_labels = [r'$0$', r'$\tau$'] + [r'${}\tau$'.format(k) for k in range(2,ntau)]
ax.set_xticklabels(xtick_labels)
# corresponding ytick labels: N, N/e, N/2e, ...
❷ ytick_labels = [r'$N$',r'$N/e$'] + [r'$N/{}e$'.format(k) for k in range(2,ntau)]
ax.set_yticklabels(ytick_labels)

ax.set_xlabel(r'$t\;/\mathrm{s}$')
ax.set_ylabel(r'$y$')
ax.grid()
plt.show()
```

❶ The x-axis tick labels are $0, \tau, 2\tau, ...$

❷ The y-axis tick labels are $N, N/e, N/2e, ...$

Note that the length of the sequence of tick labels must correspond to that of the list of tick values required.

Table 7.7 Common arguments to `ax.tick_params`

Argument	Description
`axis`	Which axis to customize: `'x'`, `'y'`, or `'both'`. Default is `'both'`.
`which`	Which tick mark set to customize: `'major'`, `'minor'`, or `'both'`. Default is `'major'`.
`direction`	Tick mark direction: `'in'`, `'out'`, or `'inout'`. Default is `'in'`.
`length`	Length of the tick marks in points.
`width`	Width of the tick marks in points.
`pad`	Distance between the tick mark and its label in points.
`labelsize`	Tick label size in points.
`color`	Tick mark color (a Matplotlib specifier).
`labelcolor`	Tick mark label color (a Matplotlib specifier).

To remove the tick labels altogether set them to the empty list, for example

```
ax.set_yticklabels([])
```

This retains the tick marks themselves. If you want neither tick marks nor tick labels on the axis use:

```
ax.set_yticks([])
```

There are two kinds of ticks: major ticks and minor ticks. Only major ticks are turned on by default; the smaller and more frequent minor ticks can most easily be enabled with

```
ax.minorticks_on()
```

More advanced customization of tick marks and their labels, including showing minor tick marks for one axis only, can be achieved using the `ax.tick_params` convenience function, which takes the arguments described in Table 7.7.

Finally, `ax.xaxis` and `ax.yaxis` have a method, `set_ticks_position`, which takes a single argument used to determine where the ticks appear: for `ax.xaxis`, `'top'`, `'bottom'`, `'both'` (the default) or `'none'`; for `ax.yaxis`, `'left'`, `'right'`, `'both'` (the default) or `'none'`.

Example E7.5 The following program creates a plot with both major and minor tick marks, customized to be thicker and wider than the default, where the major tick marks point into and out of the plot area.

Listing 7.5 Customized tick marks

```
# eg7-tick-customization.py

import numpy as np
import matplotlib.pyplot as plt

# A selection of functions on rn abcissa points for 0 <= x < 1
rn = 100
rx = np.linspace(0, 1, rn, endpoint=False)
```

```
def tophat(rx):
    """ Top hat function: y = 1 for x < 0.5, y=0 for x >= 0.5 """
    ry = np.ones(rn)
    ry[rx>=0.5]=0
    return ry

# A dictionary of functions to choose from
ry = {'half-sawtooth': lambda rx: rx.copy(),
      'top-hat': tophat,
      'sawtooth': lambda rx: 2 * np.abs(rx-0.5)}

# Repeat the chosen function nrep times
nrep = 4
x = np.linspace(0, nrep, nrep*rn, endpoint=False)
y = np.tile(ry['top-hat'](rx), nrep)

fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(x,y, 'k', lw=2)
# Add a bit of padding around the plotted line to aid visualization
ax.set_ylim(-0.1,1.1)
ax.set_xlim(x[0]-0.5, x[-1]+0.5)
# Customize the tick marks and turn the grid on
ax.minorticks_on()
ax.tick_params(which='major', length=10, width=2, direction='inout')
ax.tick_params(which='minor', length=5, width=2, direction='in')
ax.grid(which='both')
plt.show()
```

❶ This `np.tile` method constructs an array by repeating a given array `nrep` times. To plot a different periodic function, choose `'half-sawtooth'` or `'sawtooth'` here.

The resulting plot is shown in Figure 7.6.

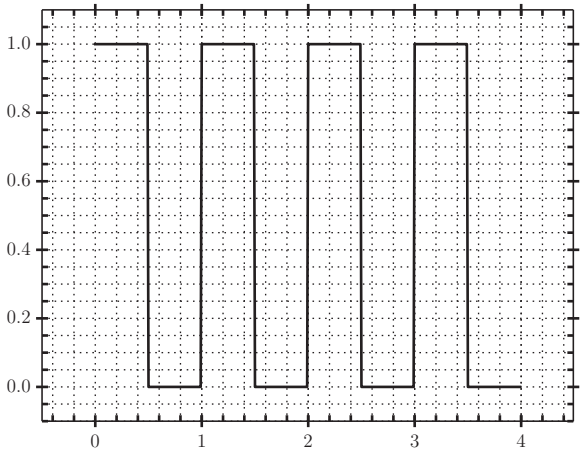

Figure 7.6 A periodic function plotted on a graph with gridlines and customized tick marks.

Table 7.8 Common arguments to `ax.errorbar`

Argument	Description
`x, y`	The data to plot
`yerr, xerr`	Errors on the *x* and *y* data coordinates, as described in the text
`fmt`	The plot format symbol (marker for the data point); set to `None` or the empty string, `''` to display only the error bars
`ecolor`	A Matplotlib color specifier for the error bars; the default, `None`, uses the same color as the connecting line between data markers
`elinewidth`	The width of the error bar lines in points; use `None` to use the same linewidth as the plotted data
`capsize`	The length of the error bar caps, in points
`errorevery`	A positive integer giving the subsampling for the error bars; for example, `errorevery=10` draws error bars on every 10th data point only

Error bars

To produce a plotted line with error bars, use the method `plt.errorbars` instead of `plt.plot`. In addition to the usual arguments of the `plot` function, `errorbars` allows the specification of errors in the $x-$ and $y-$ coordinates by passing the following types of value to the arguments `xerr` and `yerr`:

- `None`: No error bars for this coordinate;
- A scalar (e.g., `xerr=0.2`): all values are associated with symmetric error bars at plus and minus this value (i.e., ± 0.2);
- An array-type object of length `n` or shape `(n, 1)` (e.g., `yerr=[0.1, 0.15, 0.1]`): the symmetric error bars are plotted at plus and minus the values in this sequence for each of the `n` data points (i.e., $\pm 0.1, \pm 0.15, \pm 0.1$);
- An array-type object of shape `(2, n)` (i.e., two rows for each of `n` data points): error bars, which may be asymmetric, are plotted using minus-values from the first row and plus-values from the second.

The appearance of the error bars may be customized using the arguments summarized in Table 7.8. For example,

```
# Some data
x = array([ 0.3,  0.5,  0.7,  0.9])
y = array([ 1. ,  2. ,  2.5,  3.9])
# Constant, symmetric errors of +/- 0.05 on x-data
xerr = 0.05
# Asymmetric, variable errors on y-data
yerr = array([[ 0.1 ,  0.25,  0.5 ,  0.4 ],
              [ 0.1 ,  0.15,  0.2 ,  0.  ]])
ax.errorbar(x, y, yerr, xerr, fmt='o', ls='')
```

Example E7.6 Before fledging, some species of birds lose weight relative to the surface area of their wings to maximize their aerodynamic efficiency. The file

`fledging-data.csv`, available at scipython.com/eg/aad gives wing-loading values (body mass per wing area) as averages for two broods of swifts in the two weeks prior to fledging, with their uncertainties.[5]

In the program below, we perform a weighted fit to the data and plot it, with error bars.

Listing 7.6 Wing loading variation in swifts prior to fledging

```
# eg7-fledging.py
import numpy as np
import matplotlib.pyplot as plt

# Read in the data: day before fledging, wing loading and error for two broods
dt = np.dtype([('day', 'i2'), ('wl1', 'f8'), ('wl1-err', 'f8'),
               ('wl2', 'f8'), ('wl2-err', 'f8')])
data = np.loadtxt('fledging-data.csv', dtype=dt, delimiter=',')

# Weighted fit of exponential decay to the data. This is a linear least squares
# problem because y = Aexp(-Bx) => ln y = ln A - Bx = mx + c
p1_fit = np.poly1d(np.polyfit(data['day'], np.log(data['wl1']), 1,
                   w=np.log(data['wl1'])**-2))
p2_fit = np.poly1d(np.polyfit(data['day'], np.log(data['wl2']), 1,
                   w=np.log(data['wl2'])**-2))
wl1fit = np.exp(p1_fit(data['day']))
wl2fit = np.exp(p2_fit(data['day']))

# Plot the data points with their uncertainties and the fits
fig = plt.figure()
ax = fig.add_subplot(111)

# wl1 data: white circles, black borders, with error bars
ax.errorbar(data['day'], data['wl1'], yerr=data['wl1-err'], ls='', marker='o',
            color='k', mfc='w', mec='k')
ax.plot(data['day'], wl1fit, 'k', lw=1.5)

# wl2 data: black filled circles, with error bars
ax.errorbar(data['day'], data['wl2'], yerr=data['wl2-err'], ls='', marker='o',
            color='k', mfc='k', mec='k')
ax.plot(data['day'], wl2fit, 'k', lw=1.5)

ax.set_xlim(15,0)
ax.set_ylim(0.003, 0.012)
ax.set_xlabel('days pre-fledging')
ax.set_ylabel('wing loading ($\mathrm{g\,mm^{-2}}$)')
plt.show()
```

❶ The data points are weighted in the fit by $1/\sigma^2$ where σ is the estimated one-standard deviation error of the measurement.

Figure 7.7 shows the results of the fit. The broods, initially with different average wing-loading values, are seen to converge prior to fledging.

[5] J. Wright et al., *Proc. R. Soc. B* **273**, 1895 (2006).

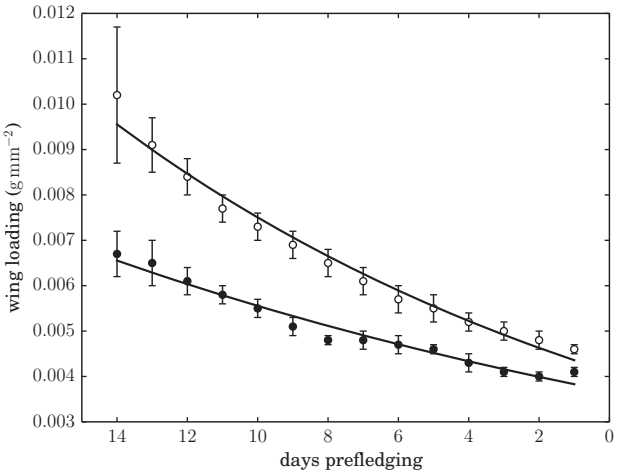

Figure 7.7 Fitted time series for wing-loading values in two cohorts of swift nestlings.

7.1.2 Bar charts and pie charts

Bar charts and histograms

The basic `pyplot` function for plotting a bar chart is `ax.bar`, which makes a plot of rectangular bars defined by their *left* edges and height. For example,

```
ax.bar([0, 1, 2], [40, 80, 20])
```

The width of the rectangles is, by default, `0.8` but can be set with the (third) `width` argument. If you want the bars vertically centered, either set the argument `align` to `'center'` or calculate where their left edges should be:

```
w = 0.5
x, y = np.array([0, 1, 2]), np.array([40, 80, 20])
ax.bar(x, y, w, align='center') # easiest way of centering the bars
ax.bar(x - w/2, y, w)           # or calculate the left edges
```

Additional arguments, including the provision of error bars, are given in Table 7.9.

By default, `ax.bar` produces a vertical bar chart. Horizontal bar charts are catered for either by setting `orientation='horizontal'` or by using the analogous `ax.barh` method.

Example E7.7 The following program produces a bar chart of letter frequencies in the English language, estimated by analysis of the text of *Moby-Dick*.[6] The vertical bars are centered and labeled by letter (Figure 7.8).

Listing 7.7 Letter frequencies in the text of *Moby-Dick*.

```
# eg7-charfreq.py
import numpy as np
import matplotlib.pyplot as plt
```

[6] See, for example, www.gutenberg.org/ebooks/2701 for a free text file of this novel.

Table 7.9 Common arguments to `ax.bar` and `barh`

Argument	Description
`left`	A sequence of *x*-coordinates of the left edges of the bars (but see `align`)
`height`	A sequence of heights for the bars
`width`	Width of the bars. If a scalar, all bars have the same width; can be array-like for variable widths
`bottom`	The *y*-coordinates of the bottom of the bars
`height`	A sequence of heights for the bars
`color`	Colors of the bar faces (scalar or array-like)
`edgecolor`	Colors of the bar edges (scalar or array-like)
`linewidth`	Line widths of the bar edges, in points (scalar or array-like)
`xerr, yerr`	Error bar limits, as for `errorbar` (scalar or array-like)
`error_kw`	A dictionary of keyword arguments corresponding to customization the appearance of the errorbars (see Table 7.8)
`align`	The default, `'edge'`, aligns the bars by their left edges (for vertical bars) or bottom edges (for horizontal bars); `'center'` centers the bars on this axis instead
`log`	Set to `True` to use a logarithmic axis scale
`orientation`	`'vertical'` (the default) or `'horizontal'`
`hatch`	Set the hatching pattern for the bars: one of `'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'`. Repeat the character for a denser pattern

```
text_file = 'moby-dick.txt'

letters = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
# Initialize the dictionary of letter counts: {'A': 0, 'B': 0, ...}
lcount = dict([(l, 0) for l in letters])

# Read in the text and count the letter occurrences
for l in open(text_file).read():
    try:
        lcount[l.upper()] += 1
    except KeyError:
        # Ignore characters that are not letters
        pass
# The total number of letters
norm = sum(lcount.values())

fig = plt.figure()
ax = fig.add_subplot(111)
# The bar chart, with letters along the horizontal axis and the calculated
# letter frequencies as percentages as the bar height
x = range(26)
ax.bar(x, [lcount[l]/norm * 100 for l in letters], width=0.8,
       color='g', alpha=0.5, align='center')
ax.set_xticks(x)
ax.set_xticklabels(letters)
ax.tick_params(axis='x', direction='out')
ax.set_xlim(-0.5, 25.5)
```

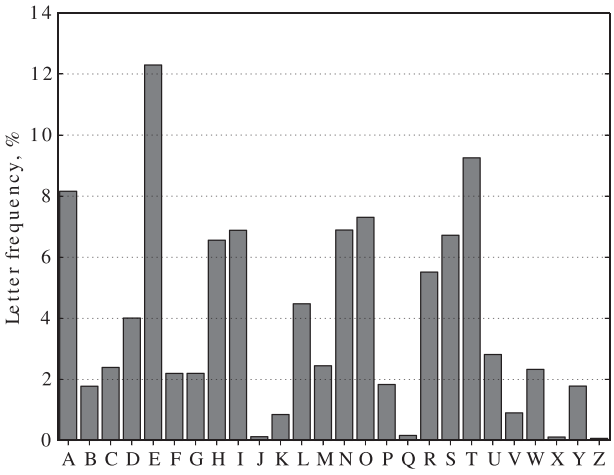

Figure 7.8 Letter frequencies in the novel *Moby-Dick*.

```
ax.yaxis.grid(True)
ax.set_ylabel('Letter frequency, %')
plt.show()
```

For monochrome plots, it is sometimes preferable to distinguish bars by patterns. The `hatch` argument can be used to do this, using any of several predefined patterns (see Table 7.9) as illustrated in the example below.

Example E7.8 The file `germany-energy-sources.txt`, available at scipython.com/eg/aae contains data on the renewable sources of electricity produced in Germany from 1990 to 2013:

```
Renewable electricity generation in Germany in GWh (million kWh)
Year      Hydro    Wind      Biomass  Photovoltaics
2013     21200     49800      47800     29300
2012     21793     50670      43350     26380
2011     17671     48883      37603     19559
...
```

The program below plots these data as a *stacked bar chart*, using Matplotlib's hatch patterns to distinguish between the different sources (Figure 7.9).

Listing 7.8 Visualizing renewable electricity generation in Germany

```
# eg7-germany-alt-energy.py
import numpy as np
import matplotlib.pyplot as plt

data = np.loadtxt('germany-energy-sources.txt', skiprows=2, dtype='i4')
years = data[:,0]
n = len(years)
```

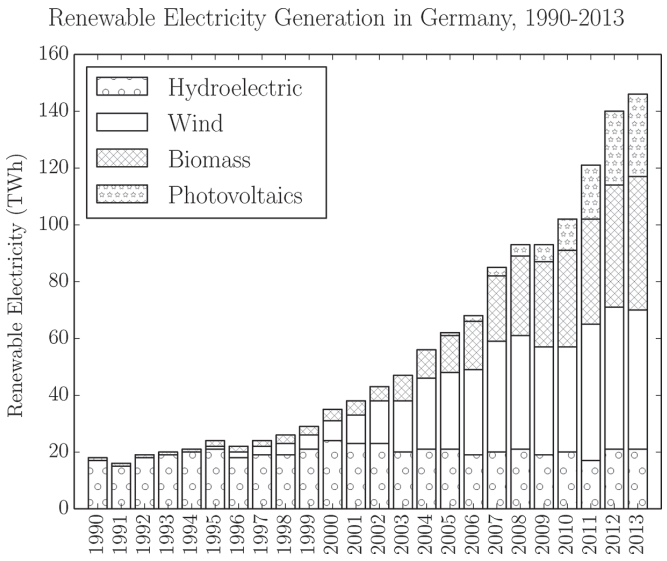

Figure 7.9 Stacked bar chart of renewable energy generation in Germany, 1990–2013.

```
# GWh to TWh
data[:,1:] /= 1000

fig = plt.figure()
ax = fig.add_subplot(111)
sources = ('Hydroelectric', 'Wind', 'Biomass', 'Photovoltaics')
hatch = ['o', '', 'xxxx', '**']
bottom = np.zeros(n)
bars = [None]*n
for i, source in enumerate(sources):
❶     bars[i] = ax.bar(years, bottom=bottom, height=data[:,i+1], color='w',
                       hatch=hatch[i], align='center')
    bottom += data[:,i+1]

ax.set_xticks(years)
plt.xticks(rotation=90)
ax.set_xlim(1989, 2014)
ax.set_ylabel('Renewable Electricity (TWh)')
ax.set_title('Renewable Electricity Generation in Germany, 1990-2013')
❷ plt.legend(bars, sources, loc='best')
plt.show()
```

❶ To include a legend, each bar chart object[7] must be stored in a list, `bars`, which ❷ is passed to the `ax.legend` method with a corresponding sequence of labels, `sources`.

[7] Actually a `Container` of *artists*.

Table 7.10 Common arguments to `ax.pie`

Argument	Description
`colors`	A sequence of Matplotlib color specifiers for coloring the segments
`labels`	A sequence of strings for labeling the segments
`explode`	A sequence of values specifying the fraction of the pie chart radius to offset each wedge by (0 for no explode effect)
`shadow`	`True` or `False`: specifies whether to draw an attractive shadow under the pie
`startangle`	Rotate the "start" of the pie chart by this number of degrees counter-clockwise from the horizontal axis
`autopct`	A format string to label the segments by their percentage fractional value, or a function for generating such a string from the data
`pctdistance`	The radial position of the `autopct` text, relative to the pie radius. The default is 0.6 (i.e., within the pie, which can be awkward for narrow segments)
`labeldistance`	The radial position of the `label` text, relative to the pie radius; the default is 1.1 (just outside the pie)
`radius`	The radius of the pie (the default is 1); this is useful when creating overlapping pie charts with different radii

Pie charts

It is straightforward to draw a pie chart in Matplotlib by passing an array of values to `ax.pie`. The values will be normalized by their sum if this sum is greater than 1, or otherwise treated directly as fractions. Labels, percentages, "exploded" segments and other effects are handled as described in Table 7.10 and illustrated in the following example.

Example E7.9 The following program depicts the emissions of greenhouse gases by mass of "carbon equivalent" (data from the 2007 IPCC report).[8]

Listing 7.9 Pie chart of greenhouse gas emissions

```
# eg7-pie.py
import numpy as np
import matplotlib.pyplot as plt

# Annual greenhouse gas emissions, billion tons carbon equivalent (GtCe)
gas_emissions = np.array([(r'$\mathrm{CO_2}$-d', 2.2),
                          (r'$\mathrm{CO_2}$-f', 8.0),
                          ('Nitrous\nOxide', 1.0),
                          ('Methane', 2.3),
                          ('Halocarbons', 0.1)],
                         dtype=[('source', 'U17'), ('emission', 'f4')])
```

[8] IPCC (2007), *Climate Change 2007: Synthesis Report. Contribution of Working Groups I, II and III to the Fourth Assessment Report of the Intergovernmental Panel on Climate Change* [Core Writing Team, Pachauri, R. K and Reisinger, A. (eds.)]. Geneva, Switzerland: IPCC, 104 pp.

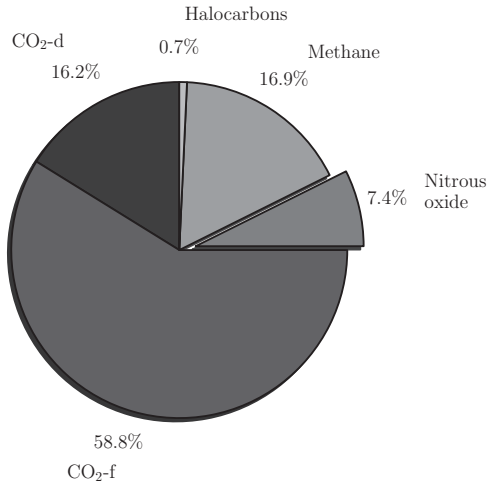

Figure 7.10 Greenhouse gas emissions by percentage for five different sources. CO_2-d denotes CO_2 emissions from deforestation; CO_2-f denotes CO_2 emissions from fossil fuel burning.

```
# 5 colors beige
colors = ['#C7B299', '#A67C52', '#C69C6E', '#754C24', '#534741']

❶ explode = [0, 0, 0.1, 0, 0]

fig, ax = plt.subplots()
ax.axis('equal')        # So our pie looks round!
ax.pie(gas_emissions['emission'], colors=colors, shadow=True, startangle=90,
❷       explode=explode, labels=gas_emissions['source'], autopct='%.1f%%',
       pctdistance=1.15, labeldistance=1.3)

plt.show()
```

❶ The segment corresponding to nitrous oxide has been "exploded" by 10%.

❷ The percentage values are formatted to one decimal place (`autopct='%.1f%%'`). The resulting pie chart is shown in Figure 7.10.

7.1.3 Multiple subplots

To create a figure with more than one subplot (that is, `Axes`), call `add_subplot` on your `Figure` object, setting its argument to indicate where the subplot should be placed. Each call returns an `Axes` object. Single figures with more than 10 subplots are uncommon, so the usual argument is a three-digit number where each digit indicates the number of rows, number of columns and subplot number. The subplot number increases along the columns in each row and then down the rows. For example, a figure consisting of three rows of two columns of subplots can be constructed by adding `Axes` objects:

```
In [x]: fig = plt.figure()
In [x]: ax1 = fig.add_subplot(321)  # top left subplot
```

```
In [x]: ax2 = fig.add_subplot(322)  # top right subplot
In [x]: ax3 = fig.add_subplot(323)  # middle left subplot
...
In [x]: ax6 = fig.add_subplot(326)  # bottom right subplot
```

Alternatively, to create a figure and add all its subplots to it at the same time, call `plt.subplots`, which takes arguments `nrows` and `ncols` (in addition to those listed in Table 7.1) and returns a `Figure` and an array of `Axes` objects, which can be indexed for each individual axis:

```
In [x]: fig, axes = plt.subplots(nrows=3, ncols=2)
In [x]: axes.shape
Out[x]: (3, 2)

In [x]: ax1 = axes[0,0]      # top left subplot
In [x]: ax2 = axes[2,1]      # bottom right subplot
```

In fact, a useful idiom to create a plot with a single `Axes` object is to call `subplots()` with no arguments:

```
In [x]: fig, ax = plt.subplots()
In [x]: ax.plot(x,y)        # no need to index the single Axes object created
```

Plots with subplots run the risk of their labels, titles and ticks overlapping each other – if this happens, call the method `tight_layout` on the `Figure` object and Matplotlib will do its best to arrange them so that there is sufficient space between them.

Example E7.10 Consider a metal bar of cross-sectional area, A, initially at a uniform temperature, θ_0, which is heated instantaneously at the exact center by the addition of an amount of energy, H. The subsequent temperature of the bar (relative to θ_0) as a function of time, t, and position, x, is governed by the one-dimensional diffusion equation:

$$\theta(x,t) = \frac{H}{c_p A}\frac{1}{\sqrt{Dt}}\frac{1}{\sqrt{4\pi}}\exp\left(-\frac{x^2}{4Dt}\right),$$

where c_p and D are the metal's specific heat capacity per unit volume and thermal diffusivity (which we assume are constant with temperature). The following code plots $\theta(x,t)$ for three specific times and compares the plots between two metals, with different thermal diffusivities but similar heat capacities, copper and iron.

Listing 7.10 The one-dimensional diffusion equation applied to the temperature of two different metal bars

```
# eg7-diffusion1d.py
import numpy as np
import matplotlib.pyplot as plt

# Cross-sectional area of bar in m3, heat added at x=0 in J
A, H = 1.e-4, 1.e3
# Temperature in K at t=0
theta0 = 300

# Metal element symbol, specific heat capacities per unit volume (J.m-3.K-1),
# Thermal diffusivities (m2.s-1) for Cu and Fe
```

```
metals = np.array([('Cu', 3.45e7, 1.11e-4), ('Fe', 3.50e7, 2.3e-5)],
                  dtype=[('symbol', '|S2'), ('cp', 'f8'), ('D', 'f8')])

# The metal bar extends from -xlim to xlim (m)
xlim, nx = 0.05, 1000
x = np.linspace(-xlim, xlim, nx)

# Calculate the temperature distribution at these three times
times = (1e-2, 0.1, 1)
# Create our subplots: three rows of times, one column for each metal
fig, axes = plt.subplots(nrows=3, ncols=2, figsize=(7, 8))
for j, t in enumerate(times):
    for i, metal in enumerate(metals):
        symbol, cp, D = metal
        ax = axes[j, i]
        # The solution to the diffusion equation
        theta = theta0 + H/cp/A/np.sqrt(D*t) / 4/np.pi * np.exp(-x**2/4/D/t)
        # Plot, converting distances to cm and add some labeling
        ax.plot(x*100, theta, 'k')
        ax.set_title('{}, $t={}$ s'.format(symbol.decode('utf8'), t))
        ax.set_xlim(-4, 4)
        ax.set_xlabel('$x\;/\mathrm{cm}$')
        ax.set_ylabel('$\Theta\;/\mathrm{K}$')

# Set up the y axis so that each metal has the same scale at the same t
for j in (0,1,2):
    ymax = max(axes[j,0].get_ylim()[1], axes[j,1].get_ylim()[1])
    print(axes[j,0].get_ylim(), axes[j,1].get_ylim())
    for i in (0,1):
        ax = axes[j,i]
        ax.set_ylim(theta0, ymax)
        # Ensure there are only three y-tick marks
        ax.set_yticks([theta0, (ymax + theta0)/2, ymax])
# We don't want the subplots to bash into each other: tight_layout() fixes this
fig.tight_layout()
plt.show()
```

Because copper is a better conductor, the temperature increase is seen to spread more rapidly for this metal (see Figure 7.11).

To further customize the subplot spacing, call `fig.subplots_adjust()`. This method takes any of the keywords `left`, `bottom`, `right`, `top`, `wspace` and `hspace`, which can be set to fractional values of the figure's height and width as appropriate to determine the positions of the subplots' left side (default 0.125), right side (0.9), bottom (0.1), top (0.9), vertical spacing (0.2) and horizontal spacing (0.2). A practical use of this function is to create "ganged" subplots that share a common axis, as in the following example.

Example E7.11 This code generates a figure of 10 subplots depicting the graph of $\sin(n\pi x)$ for $n = 0, 1, \cdots 9$. The subplot spacing is configured so that they "run into" each other vertically (see Figure 7.12).

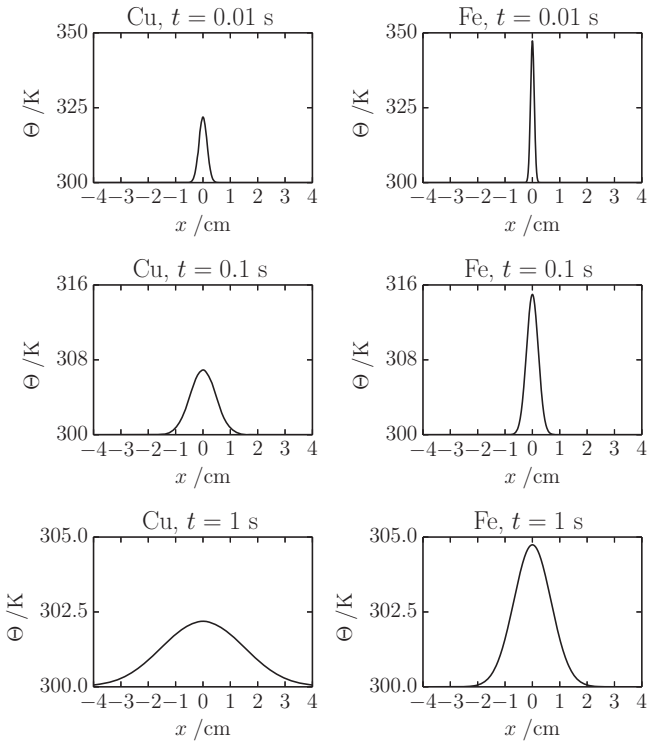

Figure 7.11 Numerical solutions of the one-dimensional diffusion equation for the temperatures of two metal bars.

Listing 7.11 Ten subplots with zero vertical spacing

```
import numpy as np
import matplotlib.pyplot as plt

nrows = 10
fig, axes = plt.subplots(nrows,1)
# Zero vertical space between subplots
fig.subplots_adjust(hspace=0)

x = np.linspace(0,1,1000)
for i in range(nrows):
    # n=nrows for the top subplot, n=0 for the bottom subplot
    n = nrows - i
    axes[i].plot(x, np.sin(n * np.pi * x), 'k', lw=2)
    # We only want ticks on the bottom of each subplot
    axes[i].xaxis.set_ticks_position('bottom')
    if i < nrows-1:
        # Set ticks at the nodes (zeros) of our sine functions
        axes[i].set_xticks(np.arange(0, 1, 1/n))
        # We only want labels on the bottom subplot xaxis
        axes[i].set_xticklabels('')
    axes[i].set_yticklabels('')
plt.show()
```

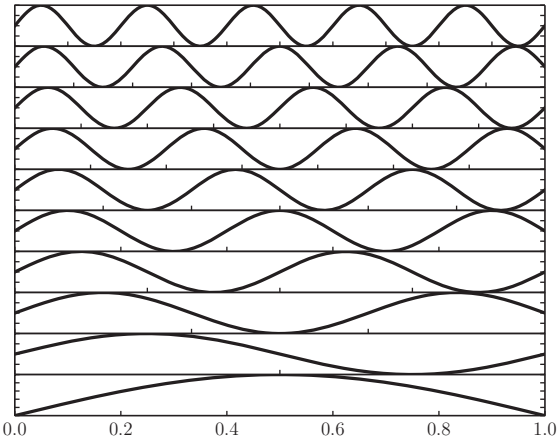

Figure 7.12 Ten subplots of $\sin(n\pi x)$ for $n = 0, 1, \cdots 9$ adjusted to remove vertical space between them.

7.1.4 Annotations

Matplotlib provides several ways to add different kinds of annotation to your plots. The most important methods for adding text, arrows, lines and shapes are described below.

Adding text

The method `ax.text(x,y,s)` is a basic method used to add a text string `s` at position `(x,y)` (in *data* coordinates) to the axes. The font properties can be determined by additionally passing a dictionary of (keyword, value) pairs to `fontdict` (see Table 7.6). Individual keyword arguments (such as `fontsize=20`) can also be used to customize the font in this way.

If the text annotation refers to a feature of the data, you will usually want the default behavior, placing it using data coordinates so that it maintains the same relative position to the data even if the plot limits are changed. If, instead, you want to place the text in *axis* coordinates (such that (0,0) is the lower left of the axes and (1,1) is the upper right), set the keyword argument `transform=ax.transAxes` where `ax` is the `Axes` object the coordinates refer to.

Arrows and text

The `ax.annotate` method is similar to `ax.text` (although with an annoyingly different syntax) but draws an arrow from the text to a specified point in the plot. The important arguments to `ax.annotate` are

- `s`, the string to output as a text label;
- `xy`, a tuple, `(x,y)` giving the coordinates of the position to annotate (i.e., where the arrow points *to*);
- `xytext`, a tuple, `(x,y)` giving the coordinates of the text label (i.e., where the arrow points *from*);

- `xycoords`, an optional string determining the type of coordinates referred to by the argument `xy`: several options are available,[9] but the most commonly used ones are
  - `'data'`: data coordinates, the default,
  - `'figure fraction'`: fractional coordinates of the *figure size* (`(0,0)` is lower left, `(1,1)` is upper right),
  - `'axes fraction'`: fractional coordinates of the *axes* (`(0,0)` is lower left, `(1,1)` is upper right), and
- `textcoords`: as for `xycoords`, an optional string determining the type of coordinates referred to by `xytext`. An additional value is permitted for this string: `'offset points'` specifies that the tuple `xytext` is an offset *in points* from the `xy` position.
- `arrowprops`: if present, determines the properties and style of the arrow drawn between `xytext` and `xy` (see below).

Additional keyword arguments are interpreted as properties of the `Text` object produced as the label (e.g., `fontsize` and `color`). An important pair is `verticalalignment` (or `va`) and `horizontalalignment` (or `ha`) which determine how the label is aligned relative to its `xytext` position. Valid values are `'center'`, `'right'`, `'left'`, `'top'`, `'bottom'` and `'baseline'` as appropriate.

In its simplest usage, `ax.annotate` just adds a text label to the plot (without an arrow). For example,

```
ax.annotate('My Label', xy=(0.5,0.8), fontsize=16, xycoords='axes fraction',
            ha='center')
```

which adds `'My Label'` at the center, near the top of the axes in 16-point text. Note that if there is no arrow or line, `xytext` is not necessary and the label is placed directly at `xy`.

The argument `arrowprops` is a dictionary determining the style of line or arrow joining the label at `xytext` to the specified `xy` point. There is a somewhat bewildering array of possible items to put in this dictionary, but the important ones are illustrated by the following example.

Example E7.12 The following program produces a plot with eight arrows with different styles (Figure 7.13).

Listing 7.12 Annotations with arrows in Matplotlib

```
# eg7-arrows.py
import numpy as np
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
x = np.linspace(0,1)
ax.plot(x, x, 'o')
```

[9] See the documentation at http://matplotlib.org/api/text_api.html/matplotlib.text.Annotation.

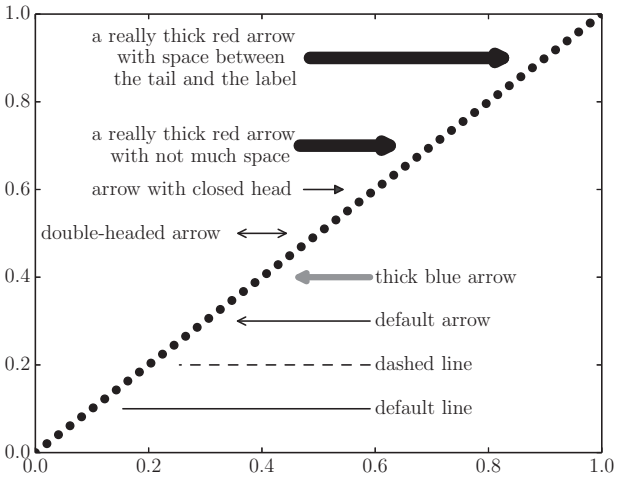

Figure 7.13 An example of different arrow styles.

```
ax.annotate('default line', xy=(0.15,0.1), xytext=(0.6,0.1),
            arrowprops={'arrowstyle': '-'}, va='center')
ax.annotate('dashed line', xy=(0.25,0.2), xytext=(0.6,0.2),
            arrowprops={'arrowstyle': '-', 'ls': 'dashed'}, va='center')
ax.annotate('default arrow', xy=(0.35,0.3), xytext=(0.6,0.3),
            arrowprops={'arrowstyle': '->'}, va='center')
ax.annotate('thick blue arrow', xy=(0.45,0.4), xytext=(0.6,0.4),
            arrowprops={'arrowstyle': '->', 'lw': 4, 'color': 'blue'},
            va='center')
ax.annotate('double-headed arrow', xy=(0.45,0.5), xytext=(0.01,0.5),
            arrowprops={'arrowstyle': '<->'}, va='center')
ax.annotate('arrow with closed head', xy=(0.55,0.6), xytext=(0.1,0.6),
            arrowprops={'arrowstyle': '-|>'}, va='center')
ax.annotate('a really thick red arrow\nwith not much space', xy=(0.65,0.7),
            xytext=(0.1,0.7), va='center', multialignment='right',
            arrowprops={'arrowstyle': '-|>', 'lw': 8, 'ec': 'r'})
ax.annotate('a really thick red arrow\nwith space between\nthe tail and the'
            'label', xy=(0.85,0.9), xytext=(0.1,0.9), va='center',
            multialignment='right',
            arrowprops={'arrowstyle': '-|>', 'lw': 8, 'ec': 'r', 'shrinkA': 10})

plt.show()
```

Example E7.13 Another example of an annotated plot, this time of the share price of BP plc (LSE: BP) with a couple of notable events added to it. The necessary data for this example can be downloaded from Yahoo! Finance.[10]

[10] https://uk.finance.yahoo.com/q/hp?s=BP.L.

Listing 7.13 eg7-share-prices

```
import datetime
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.dates import strpdate2num
from datetime import datetime

❶ def date_to_int(s):
    epoch = datetime(year=1970, month=1, day=1)
    date = datetime.strptime(s, '%Y-%m-%d')
    return (date - epoch).days

def bindate_to_int(bs):
    return date_to_int(bs.decode('ascii'))

dt = np.dtype([('daynum','i8'), ('close', 'f8')])
share_price = np.loadtxt('bp-share-prices.csv', skiprows=1, delimiter=',',
                         usecols=(0,4), converters={0: bindate_to_int},
                         dtype=dt)
fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(share_price['daynum'], share_price['close'], c='g')
❷ ax.fill_between(share_price['daynum'], 0, share_price['close'], facecolor='g',
                  alpha=0.5)

daymin, daymax = share_price['daynum'].min(), share_price['daynum'].max()
ax.set_xlim(daymin, daymax)

price_max = share_price['close'].max()

def get_xy(date):
    """ Return the (x,y) coordinates of the share price on a given date. """
    x = date_to_int(date)
    return share_price[np.where(share_price['daynum']==x)][0]

# A horizontal arrow and label
x,y = get_xy('1999-10-01')
ax.annotate('Share split', (x,y), xytext = (x+1000,y), va='center',
            arrowprops=dict(facecolor='black', shrink=0.05))
# A vertical arrow and label
x,y = get_xy('2010-04-20')
ax.annotate('Deepwater Horizon\noil spill', (x,y), xytext = (x,price_max*0.9),
            arrowprops=dict(facecolor='black', shrink=0.05), ha='center')

years = range(1989,2015,2)
ax.set_xticks([date_to_int('{:4d}-01-01'.format(year)) for year in years])
❸ ax.set_xticklabels(years, rotation=90)

plt.show()
```

❶ We need to do some work to read in the date column: first decode the byte string read in from the file to ASCII (`bindate_to_int`), then use `datetime` (see Section 4.5.3) to convert it into an integer number of days since some reference date (epoch): here we choose the Unix epoch, 1 January 1970 (`date_to_int`).

❷ `ax.fill_between` fills the region below the plotted line with a single color.

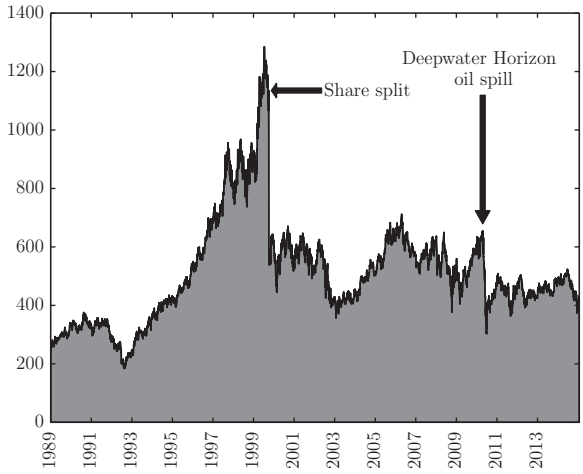

Figure 7.14 BP plc's share price on an annotated chart.

❸ We rotate the year labels so there's enough room for them (reading bottom to top). Figure 7.14 shows the plotted chart.

Lines and span rectangles

Adding an arbitrary straight line to a Matplotlib plot can be achieved by simply plotting the data corresponding to its start and end points with `ax.plot`; for example,

```
ax.plot([x1, x2], [y1, y2], color='k', lw=2)
```

draws a line between `(x1, y1)` and `(x2, y2)`. Of course, this approach would be tedious for a drawing a large number of disconnected lines, so for horizontal and vertical lines there are a pair of convenient methods, `ax.hlines` and `ax.vlines`. `ax.hlines` takes mandatory arguments `y`, `xmin`, `xmax` and draws horizontal lines with *y*-coordinates at each of the values given by the sequence `y` (if `y` is passed as a scalar, a single line is drawn). `xmin` and `xmax` specify the start and end of each line; they can be scalars (in which case all the lines will have the same start and end *x*-coordinates) or a sequence (with one value for each of the *y*-coordinates specified by `y`). `ax.vlines` draws vertical lines; its mandatory arguments, `x`, `ymin` and `ymax` are entirely analogous.

Example E7.14 The code below illustrates some different uses of `ax.vlines` and `ax.hlines` (see Figure 7.15).

Listing 7.14 Some different ways to use `ax.vlines` and `ax.hlines`

```
# eg7-circle-lines.py
import numpy as np
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
ax.axis('equal')
```

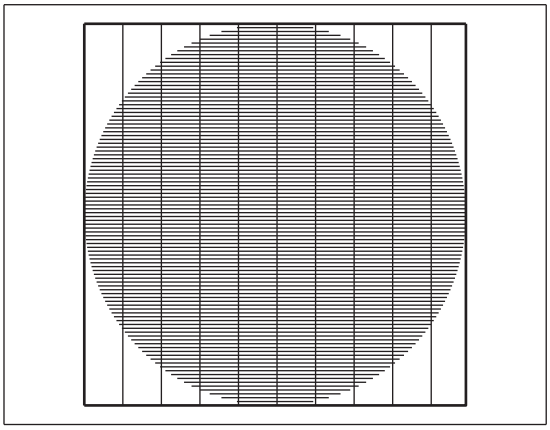

Figure 7.15 A figure generated from vertical and horizontal lines.

```
# A circle made of horizontal lines
y = np.linspace(-1,1,100)
xmax = np.sqrt(1 - y**2)
ax.hlines(y, -xmax, xmax, color='g')

# Draw a box of thicker lines around the circle
ax.vlines(-1, -1, 1, lw=2, color='r')
ax.vlines(1, -1, 1, lw=2, color='r')
ax.hlines(-1, -1, 1, lw=2, color='r')
ax.hlines(1, -1, 1, lw=2, color='r')
# Some evenly spaced vertical lines
ax.vlines(y[::10], -1, 1, color='b')

# Remove tick marks and labels
ax.xaxis.set_visible(False)
ax.yaxis.set_visible(False)
# A bit of padding around the outside of the box
ax.set_xlim(-1.1,1.1)
ax.set_ylim(-1.1,1.1)

plt.show()
```

On static plots such as figures for printing, `ax.hlines` and `ax.vlines` work well, but note that the line limits don't change upon changing the axes' limits in an interactive plot. There are two further methods, `ax.axhline` and `ax.axvline`, which simply plot a horizontal or vertical line across the axis, whatever its current limits. `axhline` takes arguments `y`, `xmin`, `xmax`, but these must be scalar values (so multiple lines require repeated calls) and `xmin`, `xmax` are given in *fractional* coordinates such that 0 is the far left of the plot and 1 the far right. Again, the `axvline` arguments: `x`, `ymin`, `ymax` are analogous. Some examples:

```
ax.axhline(100, 0, 1)   # Horizontal line across whole of x-axis at y = 100.
ax.axhline(100)         # Same thing: xmin and xmax default to 0 and 1
```

Table 7.11 Keyword arguments for styling patches

Argument	Description
`alpha`	Set the alpha transparency (0-1)
`color`	Set both the `facecolor` and the `edgecolor` of the patch
`edgecolor`, `ec`	Set the edge (border) color
`facecolor`, `fc`	Set the patch face color
`fill`	Indicate whether to fill the patch or not (`True` or `False`)
`hatch`	Set the hatching pattern for the patch: one of `'/'`, `'\'`, `'|'`, `'-'`, `'+'`, `'x'`, `'o'`, `'O'`, `'.'`, `'*'`. Repeat the character for a denser pattern
`linestyle`, `ls`	Set the patch line style: `'solid'`,`'dashed'`,`'dashdot'`,`'dotted'`
`linewidth`, `lw`	Set the patch line width, in points

```
# A thick, blue, dashed vertical line at x = 5. around the center of the y-axis
ax.axvline(5, 0.4, 0.6, c='b', lw=4, ls='--')
```

The methods `ax.axhspan` and `ax.axvspan` are similar but produce a horizontal or vertical *spanning rectangle* across the axis. `ax.axhspan` is passed arguments `ymin`, `ymax` (in *data* coordinates), and `xmin`, `xmax` (in *fractional axes* units). `ax.axvspan` takes the arguments `xmin`, `xmax`, `ymin` and `ymax` analogously. Extra keywords can be used to style the spanning rectangle (see Table 7.11).

Example E7.15 The program below annotates a simple wave plot to indicate the different regions of the electromagnetic spectrum, using `text`, `axvline`, `axhline` and `axvspan` (see Figure 7.16).

Listing 7.15 A representation of the electromagnetic spectrum, 250–1,000 nm

```
# eg7-annotate.py
import numpy as np
import matplotlib.pyplot as plt

# wavelength range, nm
lmin, lmax = 250, 1000
x = np.linspace(lmin, lmax, 1000)
# A wave with a smoothly increasing wavelength
wv = (np.sin(10 * np.pi * x / (lmax+lmin-x)))[::-1]

fig = plt.figure()
ax = fig.add_subplot(111, axisbg='k')
ax.plot(x, wv, c='w', lw=2)
ax.set_xlim(250,1000)
ax.set_ylim(-2,2)

# Label and delimit the different regions of the electromagnetic spectrum
ax.text(310, 1.5, 'UV', color='w', fontdict={'fontsize': 20})
ax.text(530, 1.5, 'Visible', color='k', fontdict={'fontsize': 20})
ax.annotate('', (400, 1.3), (750, 1.3), arrowprops={'arrowstyle': '<|-|>',
                                                    'color': 'w', 'lw': 2})
ax.text(860, 1.5, 'IR', color='w', fontdict={'fontsize': 20})
ax.axvline(400, -2, 2, c='w', ls='--')
```

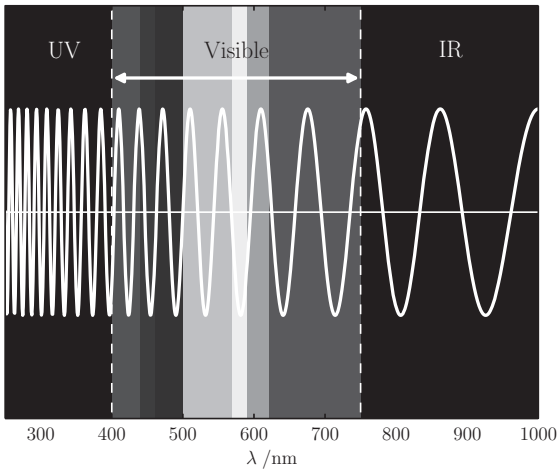

Figure 7.16 A representation of the electromagnetic spectrum.

```
ax.axvline(750, -2, 2, c='w', ls='--')
# Horizontal "axis" across the center of the wave
ax.axhline(c='w')
# Ditch the y-axis ticks and labels; label the x-axis
ax.yaxis.set_visible(False)
ax.set_xlabel(r'$\lambda\;/\mathrm{nm}$')

# Finally, add some colorful rectangles representing a rainbow in the
# visible region of the spectrum.
# Dictionary mapping of wavelength regions (nm) to approximate RGB values
rainbow_rgb = { (400, 440): '#8b00ff', (440, 460): '#4b0082',
                (460, 500): '#0000ff', (500, 570): '#00ff00',
                (570, 590): '#ffff00', (590, 620): '#ff7f00',
                (620, 750): '#ff0000'}
for wv_range, rgb in rainbow_rgb.items():
    ax.axvspan(*wv_range, color=rgb, ec='none', alpha=1)
plt.show()
```

7.1.5 ◊ Circles, ellipses, rectangles and other patches

Almost everything that gets rendered in a Matplotlib figure is a subclass of the abstract base class, `Artist`. This includes lines (through `Line2D`) and text (through `Text`).[11] An important collection of rendered objects is further derived from the `Artist` subclass `Patch`: a two-dimensional shape. The wedges of a pie chart (Section 7.1.2) and the arrows of an annotation (Section 7.1.4) are examples we have met before.

[11] In fact, there are two kinds of `Artist`: *primitives* and *containers*. Primitives are the graphical objects (such as `Line2D` themselves) and containers are the elements of a figure onto which they are rendered (for example `Axes`).

To add a shape to an `Axes` object, create a patch using one of the classes described in full in the Matplotlib documentation[12] and call `ax.add_artist(patch)`. To set the color, line widths, transparency, etc. of the patch, pass one or more of the keywords listed in Table 7.11 when creating the patch.

Below we describe this usage for a few `Patch` objects.

Circles and Ellipses

A `Circle` centered at `xy = (x,y)` (in data coordinates) and with radius `r` is created with:

```
from matplotlib.patches import Circle
circle = Circle(xy, r, **kwargs)
```

It is added to the Axes with `ax.add_artist`:

```
ax.add_artist(circle)
```

The supported keyword arguments indicated by `**kwargs` are the usual patch styling ones, summarized in Table 7.11.

`Ellipse` patches are similar but take arguments `width` and `height` (the total length of the horizontal and vertical axes of the ellipse before rotation) and `angle` (the angle of counterclockwise rotation of the ellipse *in degrees*).

```
from matplotlib.patches import Ellipse
ellipse = Ellipse(xy, width, height, angle, **kwargs)
```

Example E7.16 The following code reads in the heights and masses of 260 women and 247 men from the data set published by Heinz *et al.*[13] and available for download at www.amstat.org/publications/jse/datasets/body.dat.txt. It plots the (height, mass) pairs for each individual on a scatterplot and, for each sex, draws a 3σ covariance ellipse around the mean point. The dimensions of this ellipse are given by the (scaled) eigenvalues of the covariance matrix and it is rotated such that its semi-major axis lies along the largest eigenvector.

Listing 7.16 An analysis of the height-mass relationship in 507 healthy individuals

```
# eg7-body-mass-height.py
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.patches import Ellipse

FEMALE, MALE = 0, 1
dt = np.dtype([('mass', 'f8'), ('height', 'f8'), ('gender', 'i2')])
data = np.loadtxt('body.dat.txt', usecols=(22,23,24), dtype=dt)

fig, ax = plt.subplots()
```

[12] http://matplotlib.org/api/artist_api.html.

[13] G. Heinz *et al.*, *Journal of Statistical Education* **11** (2), 2003. This article is available at www.amstat.org/publications/jse/v11n2/datasets.heinz.html.

```
def get_cov_ellipse(cov, center, nstd, **kwargs):
    """
    Return a matplotlib Ellipse patch representing the covariance matrix
    cov centered at center and scaled by the factor nstd.

    """

    # Find and sort eigenvalues and eigenvectors into descending order
    eigvals, eigvecs = np.linalg.eigh(cov)
    order = eigvals.argsort()[::-1]
    eigvals, eigvecs = eigvals[order], eigvecs[:, order]

    # The counterclockwise angle to rotate our ellipse by
    vx, vy = eigvecs[:,0][0], eigvecs[:,0][1]
❶   theta = np.arctan2(vy, vx)

    # Width and height of ellipse to draw
    width, height = 2 * nstd * np.sqrt(eigvals)
    return Ellipse(xy=center, width=width, height=height,
                   angle=np.degrees(theta), **kwargs)

labels, colors =['Female', 'Male'], ['magenta', 'blue']
for gender in (FEMALE, MALE):
    sdata = data[data['gender']==gender]
    height_mean = np.mean(sdata['height'])
    mass_mean = np.mean(sdata['mass'])
    cov = np.cov(sdata['mass'], sdata['height'])
    ax.scatter(sdata['height'], sdata['mass'], color=colors[gender],
               label=labels[gender])
    e = get_cov_ellipse(cov, (height_mean, mass_mean), 3,
                        fc=colors[gender], alpha=0.4)
    ax.add_artist(e)

ax.set_xlim(140, 210)
ax.set_ylim(30, 120)
ax.set_xlabel('Height /cm')
ax.set_ylabel('Mass /kg')
ax.legend(loc='upper left', scatterpoints=1)
plt.show()
```

❶ The function `np.arctan2` returns the "two-argument arctangent": `np.arctan2(y, x)` is the angle in radians between the positive x-axis and the point (x, y).

Figure 7.17 shows the resulting plot.

Rectangles

`Rectangle` patches are created in a similar way to `Ellipses`:

```
from matplotlib.patches import Rectangle
rectangle = Rectangle(xy, width, height, angle, **kwargs)
```

Here, however, the tuple `xy=(x,y)` gives the coordinates of the *lower-left* corner of the rectangle. A square is simply a rectangle with the same `width` and `height`, of course.

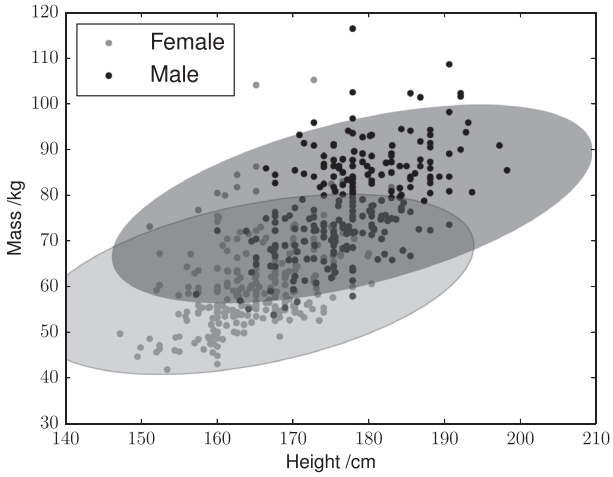

Figure 7.17 Scatterplots for each gender of mass and height for a total of 507 students, with their covariance ellipses annotated.

Polygons

A `Polygon` patch is created by passing an array of shape `(N,2)`, in which each row represents the (x, y) coordinates of a vertex. If the additional argument, `closed` is `True` (the default), the polygon will be closed so that the start and end points are the same. This is illustrated in the following example.

Example E7.17 This code produces an image (Figure 7.18) of some colorful shapes.

Listing 7.17 Some colorful shapes

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.patches import Polygon, Circle, Rectangle

red, blue, yellow, green = '#ff0000', '#0000ff', '#ffff00', '#00ff00'
square = Rectangle((0.7, 0.1), 0.25, 0.25, facecolor=red)
circle = Circle((0.8, 0.8), 0.15, facecolor=blue)
triangle = Polygon(((0.05,0.1), (0.396,0.1), (0.223, 0.38)), fc=yellow)
rhombus = Polygon(((0.5,0.2), (0.7,0.525), (0.5,0.85), (0.3,0.525)),  fc=green)

fig = plt.figure()
ax = fig.add_subplot(111, axisbg='k', aspect='equal')
for shape in (square, circle, triangle, rhombus):
    ax.add_artist(shape)
ax.xaxis.set_visible(False)
ax.yaxis.set_visible(False)

plt.show()
```

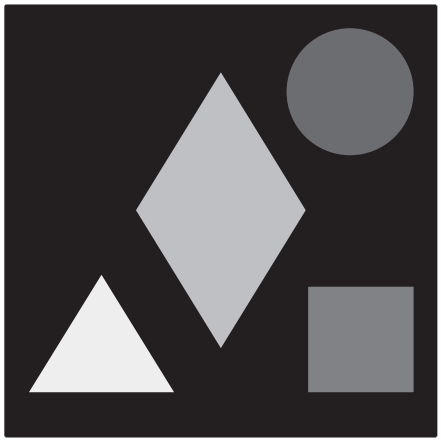

Figure 7.18 Some colorful shapes using Matplotlib patches.

Questions

Q7.1.1 Compare plots of $y = x^3$ for $-10 \leq x \leq 10$ using a logarithmic scale on the x-axis, y-axis and both axes. What is the difference between using `ax.set_xscale('log')` and `ax.set_xscale('symlog')`?

Q7.1.2 Adapt Example E7.7 to produce a *horizontal* bar chart, with the bars in order of decreasing letter frequency (i.e., with the most common letter, E, at the bottom).

Problems

P7.1.1 *The Economist*'s Big Mac Index is a lighthearted measure of purchasing power parity between two currencies. Its premise is that the difference between the price of a McDonald's Big Mac hamburger in one currency (converted into US dollars (USD) at the prevailing exchange rate) and its price in the United States is a measure of the extent to which that currency is over- or under-valued (relative to the dollar).

The files at scipython.com/ex/aga provide the historical Big Mac prices and exchange rates for four currencies. For each currency, calculate the percentage over- or under-valuation of each currency as

$$\frac{(\text{local price converted to USD} - \text{US price})}{(\text{US price})} \times 100$$

and plot it as a function of time.

P7.1.2 Plot, as a histogram, the data in the table below concerning the number of cases of West Nile virus disease in the United States between 1999 and 2008. The two types of disease, neuroinvasive and non-neuroinvasive, should be plotted as separate bars on the same chart for each year.

Year	Neuroinvasive cases	Non-neuroinvasive cases
1999	59	3
2000	19	2
2001	64	2
2002	2,946	1,210
2003	2,866	6,996
2004	1,148	1,391
2005	1,309	1,691
2006	1,495	2,774
2007	1,227	117
2008	689	667

P7.1.3 A bubble chart is a type of scatterplot that can depict three dimensions of data through the position (x- and y-coordinates) and size of the marker. The `plt.scatter` method can produce bubble charts by passing the marker size to its `s` attribute (in $(\text{points})^2$ such that the area of the marker is proportional to the magnitude of the third dimension).

The files `gdp.tsv`, `bmi_men.tsv` and `population_total.tsv`, available at scipython.com/ex/agc, contain the following data from 2007 for each country: the GDP per person per capita in international dollars fixed at 2005 prices, the body mass index (BMI) of men (in kg/m^2) and the total population. Generate a bubble chart of BMI against GDP, in which the population is depicted by the size of the bubble markers. Beware: some data are missing for some countries.

Bonus exercise: color the bubbles by continent using the list provided in the file `continents.tsv`.

P7.1.4 The US National Oceanic and Atmospheric Administration (NOAA) makes a data set of atmospheric carbon dioxide concentrations since 1958 freely available to the public at ftp://aftp.cmdl.noaa.gov/products/trends/co2/co2_mm_mlo.txt. Using this data, plot the "interpolated" and "trend" CO_2 concentration against time on the same graph.

P7.1.5 Write a program to plot the Planck function, $B(\lambda)$, for the spectral radiance of a Black body at temperature T as a function of wavelength, λ for the Sun ($T = 5778$ K):

$$B(\lambda) = \frac{2hc^2}{\lambda^5} \frac{1}{\exp\left(hc/\lambda k_B T\right) - 1}$$

Use a NumPy array to store values of $B(\lambda)$ from 100 to 5,000 nm, but set the wavelength range to *decrease* from 4,000 nm to 0.

P7.1.6 Reproduce Figure 7.19 using `Circle` patches.

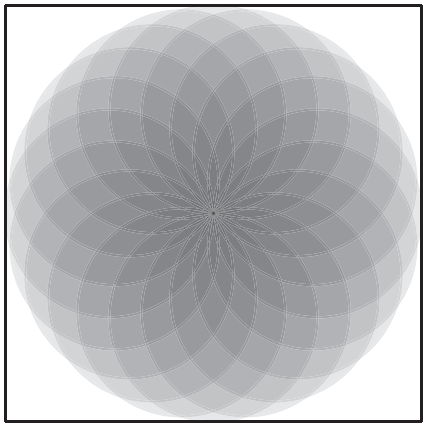

Figure 7.19 An image produced using Matplotlib `Circle` patches.

7.2 Contour plots, heatmaps and 3D plots

Until now, we have looked only at plotting one-dimensional data (that is, functions of one coordinate only). Matplotlib also supports several ways to plot data that is a function of two dimensions.

7.2.1 Contour plots

The `pyplot` method `contour` makes a contour plot of a provided two-dimensional array. In its simplest invocation, `contour(Z)`, no further arguments are required: the (x, y) values are indexes into the two-dimensional array `Z` and contour intervals are selected automatically. To explicitly include (x, y) coordinates, pass them as `contour(X, Y, Z)`. The arrays `X` and `Y` must have the same shape as `Z` (for example, as produced by `np.meshgrid`: see Section 6.1.6) or be one-dimensional such that `X` has the same length as the number of *columns* in `Z`, and `Y` has the same length as the number of *rows* in `Z`.

The contour levels can be controlled by a further argument: either a scalar, `N`, giving the total number of contour levels, or a sequence, `V`, explicitly listing the values of `Z` at which to draw contours.

The contours are colored according to Matplotlib's default *colormap*. In this process, the data are normalized linearly onto the interval `[0,1]`, which is then mapped onto a list of colors that are used to style the contours at the corresponding values. The module `matplotlib.cm` provides several colormap schemes:[14] some of the more practical ones are `cm.hot`, `cm.bone`, `cm.winter`, `cm.jet`, `cm.Greys` and `cm.hsv`. If you want to use a colormap with its colors reversed, tack a `_r` on the end of its name (e.g,, `cm.hot_r`).

[14] See the page http://wiki.scipy.org/Cookbook/Matplotlib/Show_colormaps for a complete list.

As an alternative, `contour` supports the `colors` argument which takes either a single Matplotlib color specifier or a sequence of such specifiers. For single-color contour plots, contours corresponding to negative values are plotted in dashed lines. The widths of the contour lines can be styled individually or all together with the argument `linewidths`.

Example E7.18 The following code produces a plot of the electrostatic potential of an electric dipole $\mathbf{p} = (qd, 0, 0)$ in the (x, y) plane for $q = 1.602 \times 10^{-19}$ C, $d = 1$ pm using the point dipole approximation (see Figure 7.20).

Listing 7.18 The electrostatic potential of a point dipole

```
# eg7-elec-dipole-pot.py
import numpy as np
import matplotlib.pyplot as plt

# Dipole charge (C), Permittivity of free space (F.m-1)
q, eps0 = 1.602e-19, 8.854e-12
# Dipole +q, -q distance (m) and a convenient combination of parameters
d = 1.e-12
k = 1/4/np.pi/eps0 * q * d

# Cartesian axis system with origin at the dipole (m)
X = np.linspace(-5e-11, 5e-11, 1000)
Y = X.copy()
X, Y = np.meshgrid(X, Y)

# Dipole electrostatic potential (V), using point dipole approximation
Phi = k * X / np.hypot(X, Y)**3

fig = plt.figure()
ax = fig.add_subplot(111)
# Draw contours at values of Phi given by levels
levels = np.array([10**pw for pw in np.linspace(0,5,20)])
levels = list(-levels) + list(levels)
# Monochrome plot of potential
ax.contour(X, Y, Phi, levels=levels, colors='k', linewidths=2)
plt.show()
```

To add labels to the contours, store the `ContourSet` object returned by the call to `ax.contour` and pass it to `ax.clabel` (perhaps with some additional parameters dictating the font properties). A further method, `ax.contourf`, which takes the same arguments as `contour`, draws *filled* contours. `contour` and `ax.contourf` can be used together, as in the following example.

Example E7.19 This program produces a filled contour plot of a function, labels the contours and provides some custom styling for their colors (see Figure 7.21).

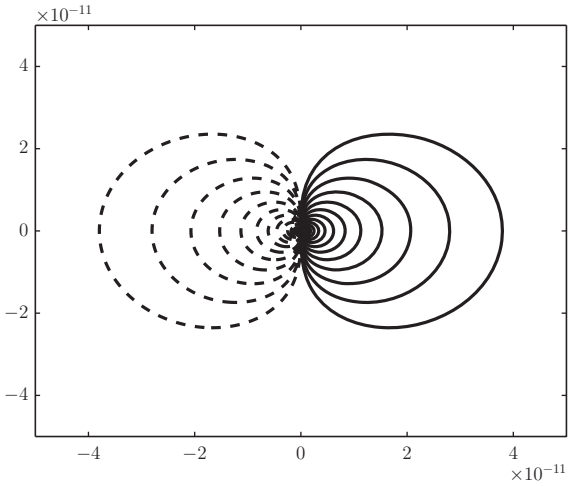

Figure 7.20 A contour plot of the electrostatic potential of a point dipole.

Listing 7.19 An example of filled and styled contours

```
# eg7-2dgau.py
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cm as cm

X = np.linspace(0,1,100)
Y = X.copy()
X, Y = np.meshgrid(X, Y)
alpha = np.radians(25)
cX, cY = 0.5, 0.5
sigX, sigY = 0.2, 0.3
rX = np.cos(alpha) * (X-cX) - np.sin(alpha) * (Y-cY) + cX
rY = np.sin(alpha) * (X-cX) + np.cos(alpha) * (Y-cY) + cY

Z = (rX-cX)*np.exp(-((rX-cX)/sigX)**2) * np.exp(- ((rY-cY)/sigY)**2)
fig = plt.figure()
ax = fig.add_subplot(111)

# Reversed Greys colormap for filled contours
cpf = ax.contourf(X,Y,Z, 20, cmap=cm.Greys_r)
# Set the colors of the contours and labels so they're white where the
# contour fill is dark (Z < 0) and black where it's light (Z >= 0)
colors = ['w' if level<0 else 'k' for level in cpf.levels]
cp = ax.contour(X, Y, Z, 20, colors=colors)
ax.clabel(cp, fontsize=12, colors=colors)
plt.show()
```

7.2.2 Heatmaps

Another way to depict two-dimensional data is as a *heatmap*: an image in which the color of each pixel is determined by the corresponding value in the array of data. The use

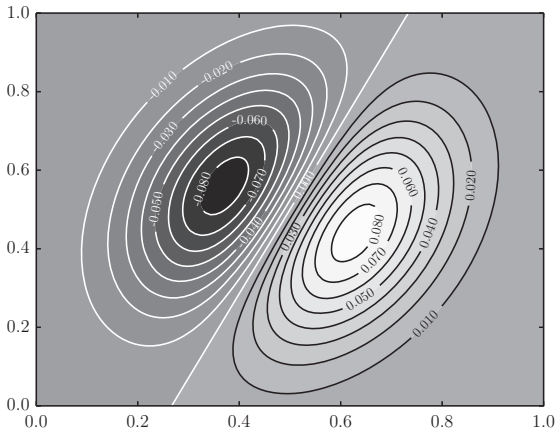

Figure 7.21 A two-dimensional plot with labeled contours.

of Matplotlib's functions, `ax.imshow`, `ax.pcolor` and `ax.pcolormesh` is described in this section.

`ax.imshow`

The `Axes` method `ax.imshow` displays an image on the axes. In its basic usage, it takes a two-dimensional array and maps its values to the pixels on an image according to some interpolation scheme and normalization. If the array data are taken from an image read in with the Matplotlib method `image.imread`, this is usually all that is required:

```
In [x]: import matplotlib.pyplot as plt
In [x]: import matplotlib.image as mpimg
In [x]: im = mpimg.imread('image.jpg')
In [x]: plt.imshow(im)
In [x]: plt.show()
```

(In this case, `im` is a three-dimensional array of shape `(n, m, 3)` in which the "depth" coordinate corresponds to the red, green and blue components of each pixel in the `n`-by-`m` image.)

`imshow` is frequently used to visualize matrices or other two-dimensional arrays of data. The default interpolation produces somewhat blurry-looking images for small arrays; for example, to visualize a 10×10 matrix as a 100×100 pixel image, a lot of intermediate points need to be approximated. To display the image with no interpolation, set `interpolation='none'` or `interpolation='nearest'`, as shown in the following example. Note that `imshow` takes a `cmap` argument that assigns a colormap in the same way as it does for `ax.contourf`.

Example E7.20 The following code compares two interpolation schemes: `'bilinear'`, which is the default on many new installations of Matplotlib and for a small array is blurry and `'nearest'`, which should look "blocky" (i.e., more faithful to the data): see Figure 7.22.

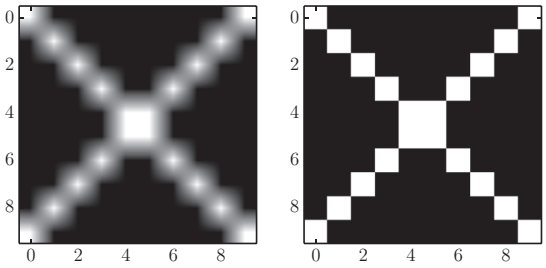

Figure 7.22 A small matrix visualized using `ax.imshow` with two different interpolation schemes.

Listing 7.20 A comparison of interpolation schemes for a small array visualized with `imshow()`

```
# eg7-matrix-show.py
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cm as cm

# Make an array with ones in the shape of an 'X'
a = np.eye(10,10)
a += a[::-1,:]

fig = plt.figure()
ax1 = fig.add_subplot(121)
# Bilinear interpolation - this will look blurry
ax1.imshow(a, cmap=cm.Greys_r)

ax2 = fig.add_subplot(122)
# 'nearest' interpolation - faithful but blocky
ax2.imshow(a, interpolation='nearest', cmap=cm.Greys_r)

plt.show()
```

Example E7.21 The *Barnsley Fern* is a fractal that resembles the Black Spleenwort species of fern. It is constructed by plotting a sequence of points in the (x, y) plane, starting at $(0, 0)$, generated by the following affine transformations f_1, f_2, f_3, and f_4 where each transformation is applied to the previous point and chosen at random with probabilities $p_1 = 0.01$, $p_2 = 0.85$, $p_3 = 0.07$ and $p_4 = 0.07$.

$$f_1(x,y) = \begin{pmatrix} 0 & 0 \\ 0 & 0.16 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

$$f_2(x,y) = \begin{pmatrix} 0.85 & 0.04 \\ -0.04 & 0.85 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} 0 \\ 1.6 \end{pmatrix}$$

$$f_3(x,y) = \begin{pmatrix} 0.2 & -0.26 \\ 0.23 & 0.22 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} 0 \\ 1.6 \end{pmatrix}$$

$$f_4(x,y) = \begin{pmatrix} -0.15 & 0.28 \\ 0.26 & 0.24 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} 0 \\ 0.44 \end{pmatrix}$$

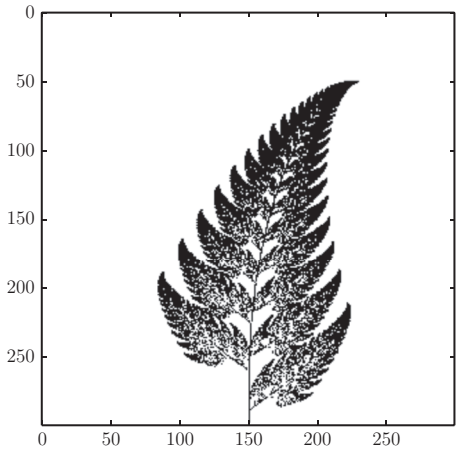

Figure 7.23 The Barnsley fern fractal.

This algorithm is implemented in the program below and the result is depicted in Figure 7.23.

Listing 7.21 Barnsley's fern

```
# eg7-fern.py
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cm as cm

f1 = lambda x,y: (0., 0.16*y)
f2 = lambda x,y: (0.85*x + 0.04*y, -0.04*x + 0.85*y + 1.6)
f3 = lambda x,y: (0.2*x - 0.26*y, 0.23*x + 0.22*y + 1.6)
f4 = lambda x,y: (-0.15*x + 0.28*y, 0.26*x + 0.24*y + 0.44)
fs = [f1, f2, f3, f4]

npts = 50000
# Canvas size (pixels)
width, height = 300, 300
aimg = np.zeros((width, height))

x, y = 0, 0
for i in range(npts):
    # Pick a random transformation and apply it
    f = np.random.choice(fs, p=[0.01, 0.85, 0.07, 0.07])
    x, y = f(x,y)
    # Map (x,y) to pixel coordinates.
    # NB we "know" that -2.2 < x < 2.7 and 0 <= y < 10

    ix, iy = width / 2 + x * width / 10, y * height / 12
    # Set this point of the array to 1 to mark a point in the fern
    aimg[iy, ix] = 1

plt.imshow(aimg[::-1,:], cmap=cm.Greens)
plt.show()
```

`ax.pcolor` and `ax.pcolormesh`

There are a couple of other similar Matplotlib methods that you will come across: `ax.pcolor` and `ax.pcolormesh`. These are very similar. The precise differences are beyond the scope of this book, but `pcolormesh` is very much faster than `pcolor` and is the recommended alternative to `imshow` for this reason. The most noticeable difference is that `imshow` follows the convention used in the image-processing community that places the origin in the *top-left* corner; the `pcolor` methods associate the origin with the bottom-left corner.

Color bars

It is often useful to have a legend indicating how the colors of the plot relate to the values of the array used to derive it. This is added with the `fig.colorbar` method. In its most simple usage, simply call `fig.colorbar(`*`mappable`*`)` where *`mappable`* is the `Image`, `ContourSet` or other suitable object to which the colorbar applies and a new `Axes` object holding the colorbar will be created (and room made in the figure to accommodate it). This object can be further customized and labeled, as shown in the following examples.

Example E7.22 The following code reads in a data file of maximum daily temperatures in Boston for 2012 and plots them on a heatmap, with a labeled colorbar legend (see Figure 7.24). The data file may be downloaded from scipython.com/eg/aah.

Listing 7.22 Heatmap of Boston's temperatures in 2012

```
# eg7-heatmap.py

import numpy as np
import matplotlib.pyplot as plt

# Read in the relevant data from our input file
dt = np.dtype([('month', np.int), ('day', np.int), ('T', np.float)])
data = np.genfromtxt('boston2012.dat', dtype=dt, usecols=(1,2,3),
                     delimiter=(4,2,2,6))

# In our heatmap, nan will mean "no such date", e.g., 31 June
heatmap = np.empty((12, 31))
heatmap[:] = np.nan

for month, day, T in data:
    # NumPy arrays are zero-indexed; days and months are not!
    heatmap[month-1, day-1] = T

# Plot the heatmap, customize and label the ticks
fig = plt.figure()
ax = fig.add_subplot(111)
im = ax.imshow(heatmap, interpolation='nearest')
ax.set_yticks(range(12))
ax.set_yticklabels(['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun',
                    'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'])
days = np.array(range(0, 31, 2))
ax.set_xticks(days)
```

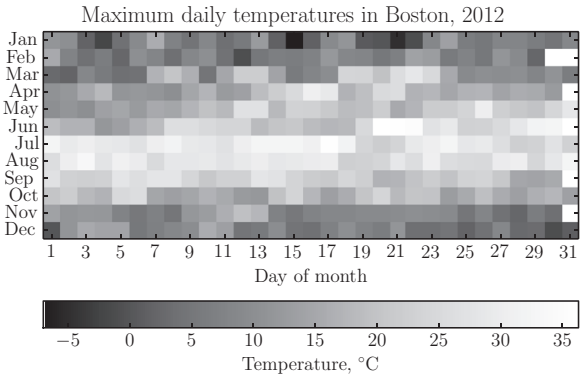

Figure 7.24 A heatmap of maximum daily temperatures in Boston during 2012.

```
ax.set_xticklabels(['{:d}'.format(day+1) for day in days])
ax.set_xlabel('Day of month')
ax.set_title('Maximum daily temperatures in Boston, 2012')

# Add a colorbar along the bottom and label it
cbar = fig.colorbar(ax=ax, mappable=im, orientation='horizontal')
cbar.set_label('Temperature, $^\circ\mathrm{C}$')

plt.show()
```

❶ The "mappable" object passed to `fig.colorbar` is the `AxesImage` object returned by `ax.imshow`.

Example E7.23 The two-dimensional diffusion equation is

$$\frac{\partial U}{\partial t} = D\left(\frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2}\right)$$

where D is the diffusion coefficient. A simple numerical solution on the domain of the unit square $0 \le x < 1, 0 \le y < 1$ approximates $U(x, y; t)$ by the discrete function $u_{i,j}^{(n)}$ where $x = i\Delta x$, $y = j\Delta y$ and $t = n\Delta t$. Applying finite difference approximations yields

$$\frac{u_{i,j}^{(n+1)} - u_{i,j}^{(n)}}{\Delta t} = D\left[\frac{u_{i+1,j}^{(n)} - 2u_{i,j}^{(n)} + u_{i-1,j}^{(n)}}{(\Delta x)^2} + \frac{u_{i,j+1}^{(n)} - 2u_{i,j}^{(n)} + u_{i,j-1}^{(n)}}{(\Delta y)^2}\right],$$

and hence the state of the system at time step $n + 1$, $u_{i,j}^{(n+1)}$ may be calculated from its state at time step n, $u_{i,j}^{(n)}$ through the equation

$$u_{i,j}^{(n+1)} = u_{i,j}^{(n)} + D\Delta t\left[\frac{u_{i+1,j}^{(n)} - 2u_{i,j}^{(n)} + u_{i-1,j}^{(n)}}{(\Delta x)^2} + \frac{u_{i,j+1}^{(n)} - 2u_{i,j}^{(n)} + u_{i,j-1}^{(n)}}{(\Delta y)^2}\right].$$

Consider the diffusion equation applied to a metal plate initially at temperature T_{cold} apart from a disc of a specified size, which is at temperature T_{hot}. We suppose that the edges of the plate are held fixed at T_{cool}. The following code applies the above formula to follow the evolution of the temperature of the plate. It can be shown that the maximum time step, Δt, that we can allow without the process becoming unstable is

$$\Delta t = \frac{1}{2D}\frac{(\Delta x \Delta y)^2}{(\Delta x)^2 + (\Delta y)^2}.$$

In the code below, each call to `do_timestep` updates the numpy array `u` from the results of the previous time step, `u0`. The simplest approach to applying the partial difference equation is to use a Python loop:

```
for i in range(1, nx-1):
    for j in range(1, ny-1):
        uxx = (u0[i+1,j] - 2*u0[i,j] + u0[i-1,j]) / dx2
        uyy = (u0[i,j+1] - 2*u0[i,j] + u0[i,j-1]) / dy2
        u[i,j] = u0[i,j] + dt * D * (uxx + uyy)
```

However, this runs extremely slowly and using vectorization will farm out these explicit loops to the much faster precompiled C code underlying NumPy's array implementation.

The state of the system is plotted as an image at four different stages of its evolution (see Figure 7.25).

Listing 7.23 The two-dimensional diffusion equation applied to the temperature of a steel plate

```
# eg7-diffusion2d.py
import numpy as np
import matplotlib.pyplot as plt

# plate size, mm
w = h = 10.
# intervals in x-, y- directions, mm
dx = dy = 0.1
# Thermal diffusivity of steel, mm2.s-1
D = 4.

Tcool, Thot = 300, 700

nx, ny = int(w/dx), int(h/dy)

dx2, dy2 = dx*dx, dy*dy
dt = dx2 * dx2 / (2 * D * (dx2 + dy2))

u0 = Tcool * np.ones((nx, ny))
u = np.empty((nx, ny))

# Initial conditions - ring of inner radius r, width dr centered at (cx,cy) (mm)
r, cx, cy = 2, 5, 5
r2 = r**2
for i in range(nx):
    for j in range(ny):
        p2 = (i*dx-cx)**2 + (j*dy-cy)**2
        if p2 < r2:
            u0[i,j] = Thot
```

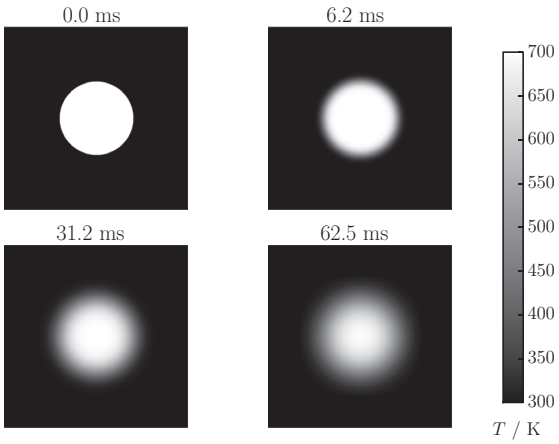

Figure 7.25 A representation of the temperature of a circular disc at four times after its instantaneous heating.

```
def do_timestep(u0, u):
    # Propagate with forward-difference in time, central-difference in space
    u[1:-1, 1:-1] = u0[1:-1, 1:-1] + D * dt * (
          (u0[2:, 1:-1] - 2*u0[1:-1, 1:-1] + u0[:-2, 1:-1])/dx2
          + (u0[1:-1, 2:] - 2*u0[1:-1, 1:-1] + u0[1:-1, :-2])/dy2 )

    u0 = u.copy()
    return u0, u

# Number of timesteps
nsteps = 101
# Output 4 figures at these timesteps
mfig = [0, 10, 50, 100]
fignum = 0
fig = plt.figure()
for m in range(nsteps):
    u0, u = do_timestep(u0, u)
    if m in mfig:
        fignum += 1
        print(m, fignum)
        ax = fig.add_subplot(220 + fignum)
        im = ax.imshow(u.copy(), cmap=plt.get_cmap('hot'), vmin=Tcool,vmax=Thot)
        ax.set_axis_off()
        ax.set_title('{:.1f} ms'.format(m*dt*1000))
fig.subplots_adjust(right=0.85)
cbar_ax = fig.add_axes([0.9, 0.15, 0.03, 0.7])
cbar_ax.set_xlabel('$T$ / K', labelpad=20)
fig.colorbar(im, cax=cbar_ax)
plt.show()
```

❶ To set a common `colorbar` for the four plots we define its own `Axes`, `cbar_ax` and make room for it with `fig.subplots_adjust`. The plots all use the same color range, defined by `vmin` and `vmax`, so it doesn't matter which one we pass in the first argument to `fig.colorbar`.

7.2.3 3D plots

Matplotlib is primarily a 2D plotting library, but it does support 3D plotting functionality that is good enough for many purposes. The easiest way to set up a 3D plot is to import `Axes3D` from the `mpl_toolkits.mplot3d` module and to set the subplot's `projection` argument to `'3d'`:

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
```

The corresponding `Axes` object can then depict data in three dimensions as a line plot, scatterplot, wireframe plot or surface plot.[15]

`ax.plot_wireframe` and `ax.plot_surface`

The simplest kind of surface plot is a wireframe plot that draws lines in 3D perspective joining the provided two-dimensional array of points, `Z`, on a grid of data values provided as two-dimensional arrays `X` and `Y` (as for `imshow` and `contour`). By default, wires are drawn for every point in the array: if this is too many, set the arguments `rstride` and `cstride` to specify the array row step size and column step size.

The `ax.plot_surface` method is similar but produces a surface plot of filled patches. The patch colors can be set to a single color with the `color` argument or styled to a specifed color map with the `cmap` argument. `rstride` and `cstride` default to 10 for the `ax.plot_surface` method. Both methods are illustrated in the following example.

Example E7.24 Some of the different options for producing surface plots are illustrated by the code below, which produces Figure 7.26.

Listing 7.24 Four 3D plots of a simple two-dimensional Gaussian function

```
# eg7-3d-surface-plots.py
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.cm as cm

L, n = 2, 400
x = np.linspace(-L, L, n)
y = x.copy()
X, Y = np.meshgrid(x, y)
Z = np.exp(-(X**2 + Y**2))

fig, ax = plt.subplots(nrows=2, ncols=2, subplot_kw={'projection': '3d'})
ax[0,0].plot_wireframe(X, Y, Z, rstride=40, cstride=40)
ax[0,1].plot_surface(X, Y, Z, rstride=40, cstride=40, color='m')
```

---

[15] It is even possible to produce three-dimensional contour plots and bar charts, though these are of doubtful use in practice.

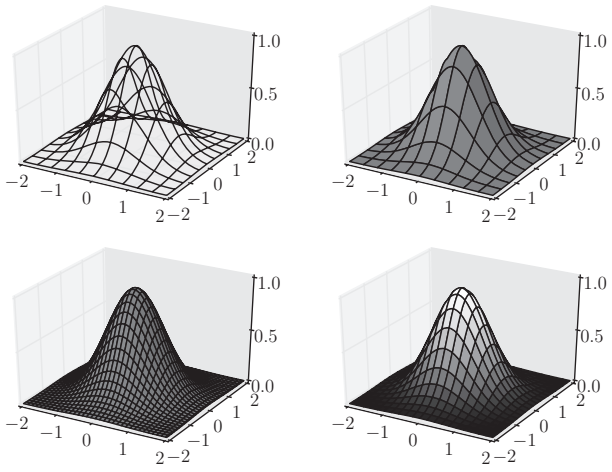

Figure 7.26 Four different 3D surface plots of the same function.

```
ax[1,0].plot_surface(X, Y, Z, rstride=12, cstride=12, color='m')
ax[1,1].plot_surface(X, Y, Z, rstride=20, cstride=20, cmap=cm.hot)
for axes in ax.flatten():
    axes.set_xticks([-2, -1, 0, 1, 2])
    axes.set_yticks([-2, -1, 0, 1, 2])
    axes.set_zticks([0, 0.5, 1])
fig.tight_layout()
plt.show()
```

In an interactive plot, the viewing direction can be changed by clicking and dragging on the plot. To fix a particular viewing direction for a static plot image, pass the required elevation and azimuthal angles (in degrees, in that order) to `ax.view_init`, as in the following example.

Example E7.25 The parametric description of a torus with radius c and tube radius a is

$$x = (c + a\cos\theta)\cos\phi$$
$$y = (c + a\cos\theta)\sin\phi$$
$$z = a\sin\theta$$

for θ and ϕ each between 0 and 2π. The code below outputs two views of a torus rendered as a surface plot (Figure 7.27).

Listing 7.25 A 3D surface plot of a torus

```
# eg7-torus-surface.py
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
```

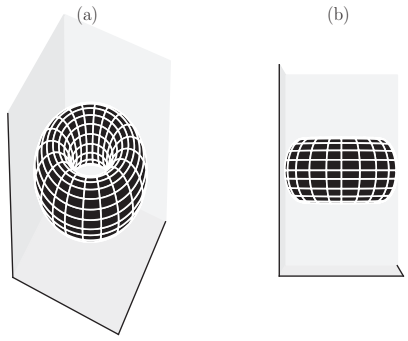

Figure 7.27 Two views of the same torus: (a) $\theta = 36°, \phi = 26°$, (b) $\theta = 0°, \phi = 0°$.

```
n = 100

theta = np.linspace(0, 2.*np.pi, n)
phi = np.linspace(0, 2.*np.pi, n)
❶ theta, phi = np.meshgrid(theta, phi)
c, a = 2, 1
x = (c + a*np.cos(theta)) * np.cos(phi)
y = (c + a*np.cos(theta)) * np.sin(phi)
z = a * np.sin(theta)

fig = plt.figure()
ax1 = fig.add_subplot(121, projection='3d')
ax1.set_zlim(-3,3)
❷ ax1.plot_surface(x, y, z, rstride=5, cstride=5, color='k', edgecolors='w')
❸ ax1.view_init(36, 26)
ax2 = fig.add_subplot(122, projection='3d')
ax2.set_zlim(-3,3)
ax2.plot_surface(x, y, z, rstride=5, cstride=5, color='k', edgecolors='w')
ax2.view_init(0, 0)
ax2.set_xticks([])
plt.show()
```

❶ We need θ and ϕ to range over the interval $(0, 2\pi)$ independently, so we use a `meshgrid`.

❷ Note that we can use keywords such as `edgecolors` to style the polygon patches created by `ax.plot_surface`.

❸ Elevation angle above the *xy*-plane of 36°, azimuthal angle in the *xy*-plane of 26°.

Line plots and scatterplots

Line plots and scatterplots work in 3D in a way similar to how they work in 2D: the basic method call is `ax.plot(x, y, z)` and `ax.scatter(x, y, z)`, where `x`, `y` and `z` are equal-length, one-dimensional arrays. Only limited annotation of such plots is possible without using advanced methods, however.

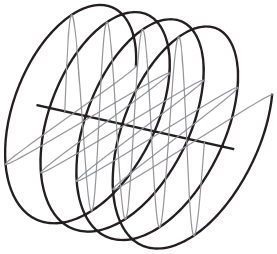

Figure 7.28 A depiction of circularly polarized light as a helix on a three-dimensional plot.

Example E7.26 Below is a simple example of a three-dimensional plot of a helix, which could represent circularly polarized light, for example. See Figure 7.28.

Listing 7.26 A depiction of a helix on a three-dimensional plot

```
# eg7-circular-polarization.py
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

n = 1000
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# Plot a helix along the x-axis
theta_max = 8 * np.pi
theta = np.linspace(0, theta_max, n)
x = theta
z =  np.sin(theta)
y =  np.cos(theta)
ax.plot(x, y, z, 'b', lw=2)

# An line through the center of the helix
ax.plot((-theta_max*0.2, theta_max * 1.2), (0,0), (0,0), color='k', lw=2)
# sin/cos components of the helix (e.g., electric and magnetic field
# components of a circularly polarized electromagnetic wave
ax.plot(x, y, 0, color='r', lw=1, alpha=0.5)
ax.plot(x, [0]*n, z, color='m', lw=1, alpha=0.5)

# Remove axis planes, ticks and labels
ax.set_axis_off()
plt.show()
```

7.2.4 Exercises

Questions

Q7.2.1 Generate an image plot of the sinc function in the Cartesian plane, $\text{sinc}(r) = \sin r/r$ where $r = \sqrt{x^2 + y^2}$.

Q7.2.2 The data provided in the comma-separated file `birthday-data.csv`, available at scipython.com/ex/agd gives the number of births recorded by the US Centers for Disease Control and Prevention's National Center for Health Statistics for each day of the year as a total from years 1969–1988. The columns are month number (1=January, 12=December), day number and number of live births.

Use NumPy to estimate, for each day of the year, the probability of a particular individual's birthday being on that day. Plot the probabilities as a heatmap like that of Example E7.22 and investigate any features of interest.

Hint: the data need "cleaning" to a small extent – inspect the data file first to establish the presence of any incorrect entries.

Problems

P7.2.1 The so-called *'chaos game'* is an algorithm for generating a fractal. First define the n vertices of a regular polygon and an initial point, (x_0, y_0) selected at random within the polygon. Then generate a sequence of points, starting with (x_0, y_0), where each point is a fraction r of the distance between the previous one and a polygon vertex chosen at random. For example, the algorithm applied with parameters $n = 3, r = 0.5$ generates a Sierpinski triangle.

Write a program to draw fractals using the chaos game algorithm.

P7.2.2 Extend the code in Example E7.16 to include contours of body mass index, defined by BMI $=$ (mass/kg)/(height/m)^2. Plot these contours to delimit the supposed categories of "under-weight" (< 18.5), "over-weight" (> 25) and "obese" (> 30). Manually place the contour labels so that they are out of the way of the scatterplotted data points and format them to one decimal place.

P7.2.3 The two-dimensional *advection equation* may be written

$$\frac{\partial U}{\partial t} = -v_x \frac{\partial U}{\partial x} - v_y \frac{\partial U}{\partial y},$$

where $\mathbf{v} = (v_x, v_y)$ is the vector velocity field (giving the velocity components v_x and v_y, which may vary as a function of position, (x, y)). In a similar way to the approach taken in Example E7.23, this equation may be discretized and solved numerically. With forward-differences in time and central-differences in space, we have

$$u_{i,j}^{(n+1)} = u_{i,j}^{(n)} - \Delta t \left[v_{x;i,j} \frac{u_{i+1,j}^{(n)} - u_{i-1,j}^{(n)}}{2\Delta x} + v_{y;i,j} \frac{u_{i,j+1}^{(n)} - u_{i,j-1}^{(n)}}{2\Delta y} \right].$$

Implement this approximate numerical solution on the domain $0 \leq x < 10, 0 \leq y < 10$ discretized with $\Delta x = \Delta y = 0.1$ with the initial condition

$$u_0(x, y) = \exp\left(-\frac{(x - c_x)^2 + (y - c_y)^2}{\alpha^2} \right),$$

where $(c_x, c_y) = (5, 5)$ and $\alpha = 2$. Take the velocity field to be a circulation at constant speed 0.1 about an origin at $(7, 5)$.

P7.2.4 The *Julia set* associated with the complex function $f(z) = z^2 + c$ may be depicted using the following algorithm.

For each point, z_0, in the complex plane such that $-1.5 \leq \mathrm{Re}[z_0] \leq 1.5$ and $-1.5 \leq \mathrm{Im}[z_0] \leq 1.5$, iterate according to $z_{n+1} = z_n^2 + c$. Color the pixel in an image corresponding to this region of the complex plane according to the number of iterations required for $|z|$ to exceed some critical value, $|z|_{\max}$ (or black if this does not happen before a certain maxmimum number of iterations $n_{\max}$).

Write a program to plot the Julia set for $c = -0.1 + 0.65j$, using $|z|_{\max} = 10$ and $n_{\max} = 500$.

P7.2.5 The mean altitudes of the 10 km × 10 km *hectad* squares used by the UK's Ordnance Survey in mapping Great Britain are given in the NumPy array file `gb-alt.npy`, available at scipython.com/ex/agb. NaN values in this array denote the sea.

Plot a map of the island using this data with `ax.imshow` and plot further maps assuming a mean sea-level rise of (a) 10 m, (b) 50 m, (c) 200 m. In each case, deduce the percentage of land area remaining, relative to its present value.