```
normalPlot()
positionOnScreen()
showAndPause()
waitForInput()
keySelection()
```

4.3 Displaying Statistical Datasets

The first step in the data analysis should always be a visual inspection of the raw-data. Between 30 and 50 % of our cortex are involved in the processing of visual information, and as a result our brain is very good at recognizing patterns in visually represented data. The trick is choosing the most informative type of display for your data.

The easiest way to find and implement one of the many image types that *matplotlib* offers is to browse their gallery (http://matplotlib.org/gallery.html), and copy the corresponding *Python* code into your program.

For statistical data analysis, the *Python* package *seaborn* (http://www.stanford.edu/~mwaskom/software/seaborn/) builds on *matplotlib*, and aims to provide a concise, high-level interface for drawing statistical graphics that are both informative and attractive. Also *pandas* builds on *matplotlib* and offers many convenient ways to visualize DataFrames.

Other interesting plotting packages are:

- *plot.ly* is a package that is available for *Python*, *Matlab*, and *R*, and makes beautiful graphs (https://plot.ly).
- *bokeh* is a *Python* interactive visualization library that targets modern web browsers for presentation. *bokeh* can help anyone who would like to quickly and easily create interactive plots, dashboards, and data applications (http://bokeh.pydata.org/).
- *ggplot* for *Python*. It emulates the *R*-package *ggplot*, which is loved by many *R*-users (http://ggplot.yhathq.com/).

4.3.1 Univariate Data

The following examples all have the same format. Only the “Plot-command” line changes.

```
# Import standard packages
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import scipy.stats as stats
import seaborn as sns
```

```
# Generate the data
x = np.random.randn(500)

# Plot-command start ---------------------
plt.plot(x, '.')
# Plot-command end -----------------------

# Show plot
plt.show()
```

a) Scatter Plots

This is the simplest way to represent univariate data: just plot each individual data point. The corresponding plot-command is either

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

or, equivalently,

```
plt.scatter(np.arange(len(x), x))
```

Note: In cases where we only have few discrete values on the *x*-axis (e.g., *Group1, Group2, Group3*), it may be helpful to spread overlapping data points slightly (also referred to as *"adding jitter")* to show each data point. An example can be found at http://stanford.edu/~mwaskom/software/seaborn/generated/seaborn.stripplot.html)

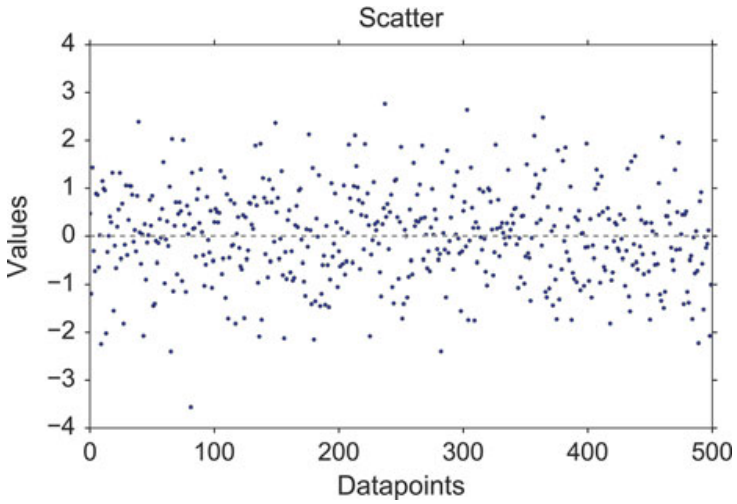

Fig. 4.1 Scatter plot

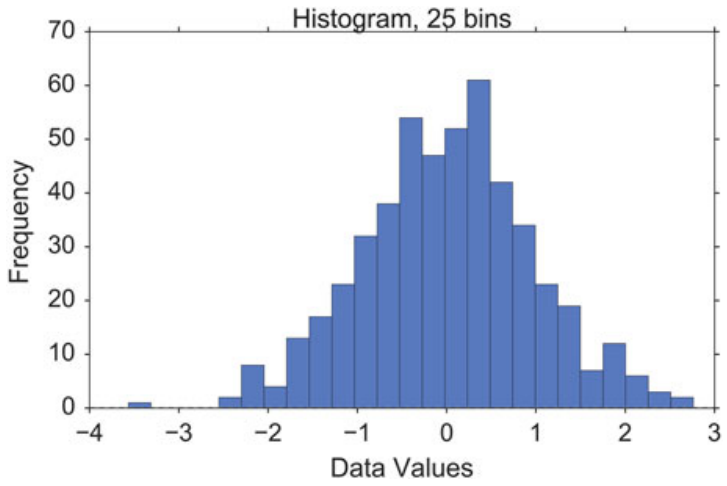

Fig. 4.2 Histogram

b) Histograms

Histograms provide a first good overview of the distribution of your data. If you divide by the overall number of data points, you get a *relative frequency histogram*; and if you just connect the top center points of each bin, you obtain a *relative frequency polygon*.

```
plt.hist(x, bins=25)
```

c) Kernel-Density-Estimation (KDE) Plots

Histograms have the disadvantage that they are discontinuous, and that their shape critically depends on the chosen bin-width. In order to obtain smooth *probability densities*, i.e., curves describing the likelihood of finding an event in any given interval, the technique of *Kernel Density Estimation* (KDE) can be used. Thereby a normal distribution is typically used for the kernel. The width of this kernel function determines the amount of smoothing. To see how this works, we compare the construction of histogram and kernel density estimators, using the following six data points:

`x = [-2.1, -1.3, -0.4, 1.9, 5.1, 6.2].`

For the histogram, first the horizontal axis is divided into subintervals or bins which cover the range of the data. In Fig. 4.3, left, we have six bins each of width 2. Whenever a data point falls inside this interval, we place a box of height 1/12. If more than one data point falls inside the same bin, we stack the boxes on top of each other.

For the kernel density estimate, we place a normal kernel with variance 2.25 (indicated by the red dashed lines in Fig. 4.3, right) on each of the data points x_i. The kernels are summed to make the kernel density estimate (solid blue curve). The smoothness of the kernel density estimate is evident. Compared to the discreteness

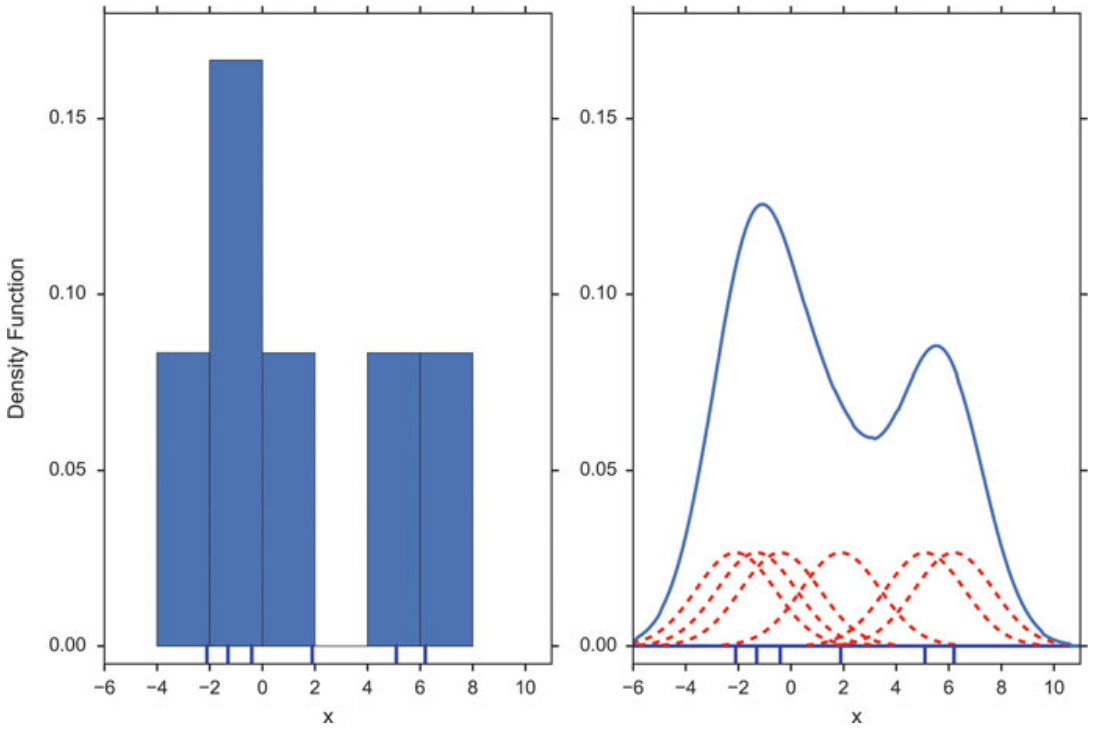

Fig. 4.3 Comparison of the histogram (*left*) and kernel density estimate (*right*) constructed using the same data. The six individual kernels are the *red dashed curves*, the kernel density estimate the *solid blue curve*. The data points are the rug plot on the horizontal axis

of the histogram, the kernel density estimates converge faster to the true underlying density for continuous random variables.

```
sns.kdeplot(x)
```

The bandwidth of the kernel is the parameter which determines how much we smooth out the contribution from each event. To illustrate its effect, we take a simulated random sample from the standard normal distribution, plotted as the blue spikes in the *rug plot* on the horizontal axis in Fig. 4.4, left. (A *rug plot* is a plot where every data entry is visualized by a small vertical tick.) The right plot shows the true density in blue. (A normal density with mean 0 and variance 1.) In comparison, the gray curve is undersmoothed since it contains too many spurious data artifacts arising from using a bandwidth $h = 0.1$ which is too small. The green dashed curve is oversmoothed since using the bandwidth $h = 1$ obscures much of the underlying structure. The red curve with a bandwidth of $h = 0.42$ is considered to be optimally smoothed since its density estimate is close to the true density.

It can be shown that under certain conditions the optimal choice for h is

$$h = \left(\frac{4\hat{\sigma}^5}{3n}\right)^{\frac{1}{5}} \approx 1.06\hat{\sigma}n^{-1/5}, \tag{4.1}$$

where $\hat{\sigma}$ is the standard deviation of the samples ("Silverman's rule of thumb").

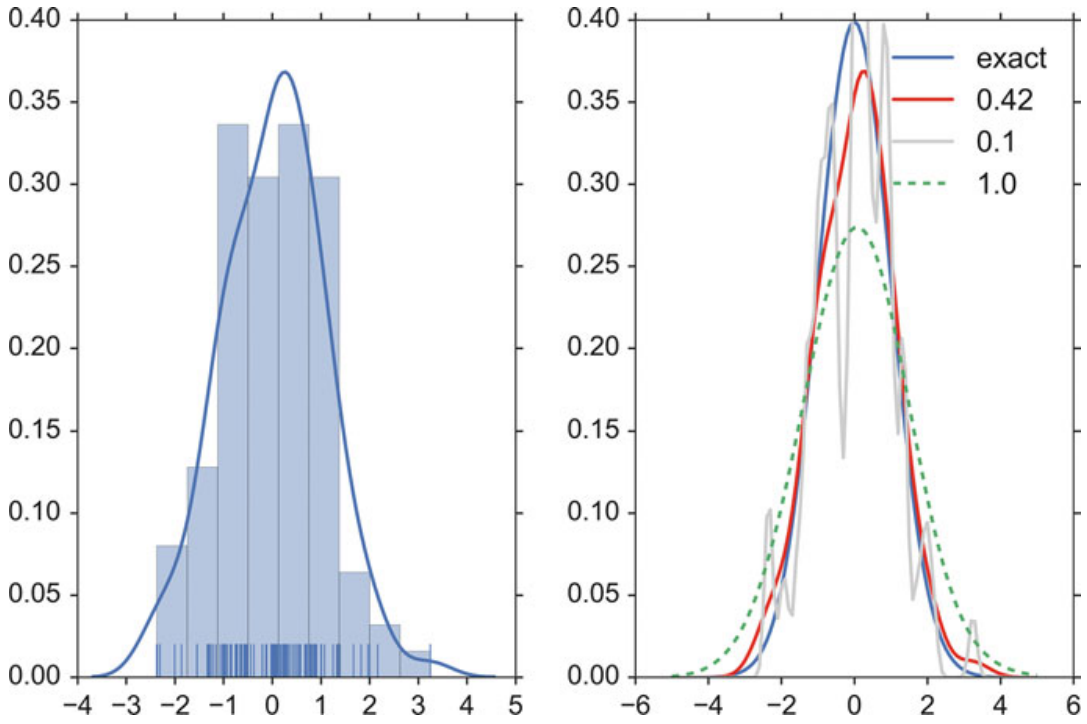

Fig. 4.4 *Left*: Rug plot, histogram, and Kernel density estimate (KDE) of a random sample of 100 points from a standard normal distribution. *Right*: True density distribution (*blue*), and KDE with different bandwidths. *Gray dashed*: KDE with $h = 0.1$; *red*: KDE with $h = 0.42$; *green dashed*: KDE with $h = 1.0$

d) Cumulative Frequencies

A *cumulative frequency* curve indicates the number (or percent) of data with less than a given value. This curve is very useful for statistical analysis, for example when we want to know the data range containing 95 % of all the values. Cumulative frequencies are also useful for comparing the distribution of values in two or more different groups of individuals.

When you use percentage points, the cumulative frequency presentation has the additional advantage that it is bounded: $0 \leq cumfreq(x) \leq 1$

```
plt.plot(stats.cumfreq(x,numbins)[0])
```

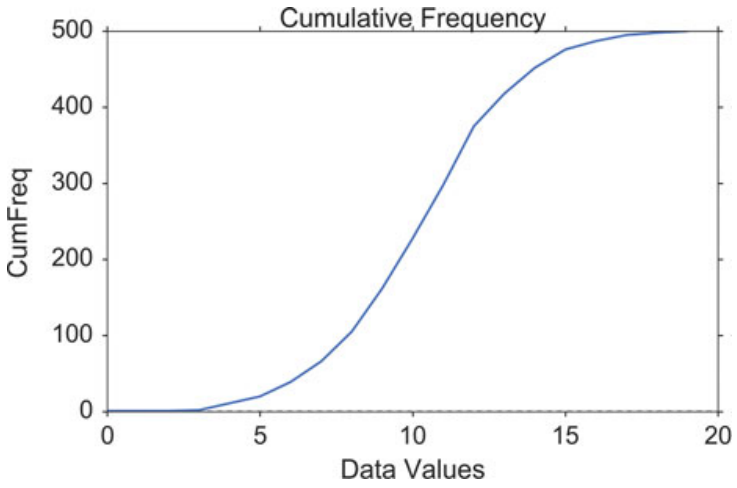

Fig. 4.5 Cumulative frequency function for a normal distribution

e) Error-Bars

Error-bars are a common way to show mean value and variability when comparing measurement values. Note that it always has to be stated explicitly if the error-bars correspond to the *standard deviation* or to the *standard error* of the data. Using *standard errors* has a nice feature: When error bars for the standard errors for two groups overlap, one can be sure the difference between the two means is not statistically significant ($p > 0.05$). However, the opposite is not always true!

```
index = np.arange(5)
y = index**2
errorBar = index/2  # just for demonstration
plt.errorbar(index,y, yerr=errorBar, fmt='o',
             capsize=5, capthick=3)
```

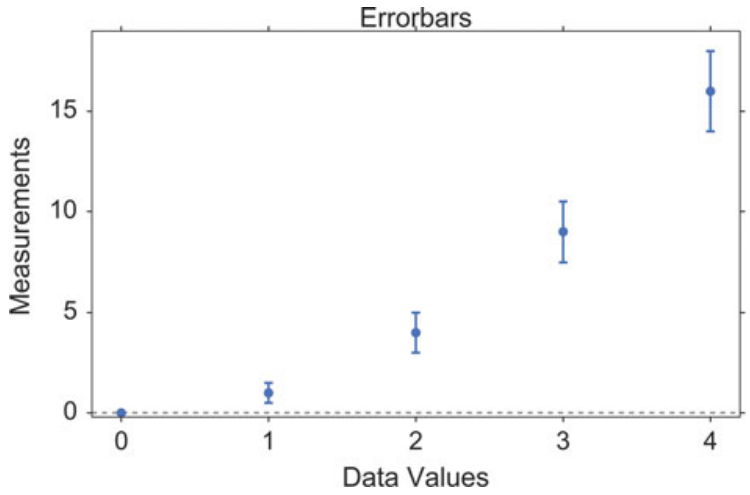

Fig. 4.6 Errorbars

f) Box Plots

Boxplots are frequently used in scientific publications to indicate values in two or more groups. The bottom and top of the box indicate the first quartile and third quartile, respectively, and the line inside the box shows the median. Care has to be taken with the whiskers, as different conventions exist for them. The most common form is that the lower whisker indicates the lowest value still within 1.5 * *inter-quartile-range* (IQR) of the lower quartile, and the upper whisker the highest value still within 1.5 * IQR of the upper quartile. Outliers (outside the whiskers) are plotted separately. Another convention is to have the whiskers indicate the full data range.

There are a number of tests to check for outliers. The method suggested by Tukey, for example, is to check for data which lie more than 1.5 * IQR above or below the first/third quartile (see Sect. 6.1.2).

```
plt.boxplot(x, sym='*')
```

Boxplots can be combined with KDE-plots to produce the so-called *violin plots*, where the vertical axis is the same as for the box-plot, but in addition a KDE-plot is shown symmetrically along the horizontal direction (Fig. 4.8).

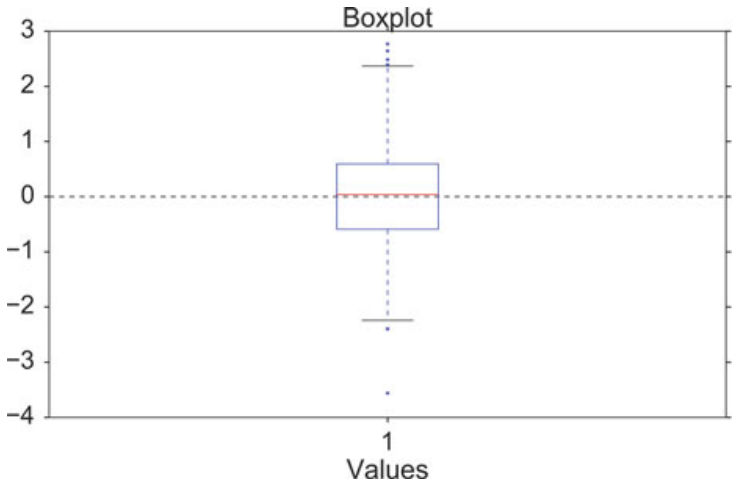

Fig. 4.7 Box plot

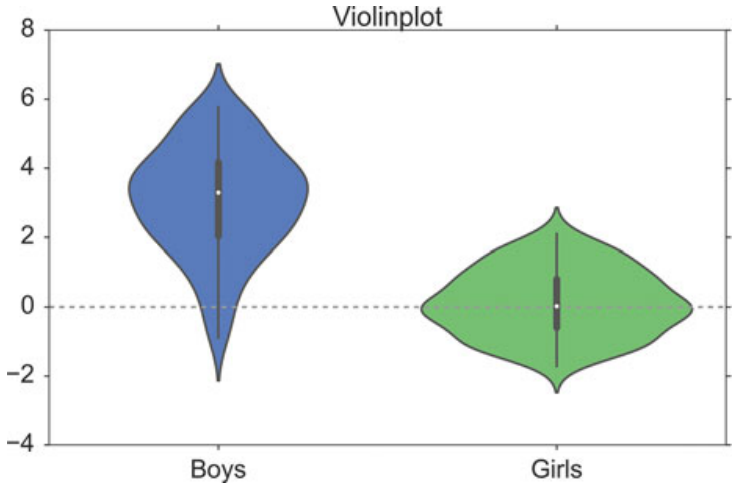

Fig. 4.8 Violinplot, produced with *seaborn*

```
# Generate the data
nd = stats.norm
data = nd.rvs(size=(100))

nd2 = stats.norm(loc = 3, scale = 1.5)
data2 = nd2.rvs(size=(100))

# Use pandas and the seaborn package
# for the violin plot
df = pd.DataFrame({'Girls':data, 'Boys':data2})
sns.violinplot(df)
```

g) Grouped Bar Charts

For some applications the plotting abilities of *pandas* can facilitate the generation of useful graphs, e.g., for grouped barplots (Figs. 4.9, 4.10, 4.11, and 4.12):

```
df = pd.DataFrame(np.random.rand(10, 4),
        columns=['a', 'b', 'c', 'd'])
df.plot(kind='bar', grid=False)
```

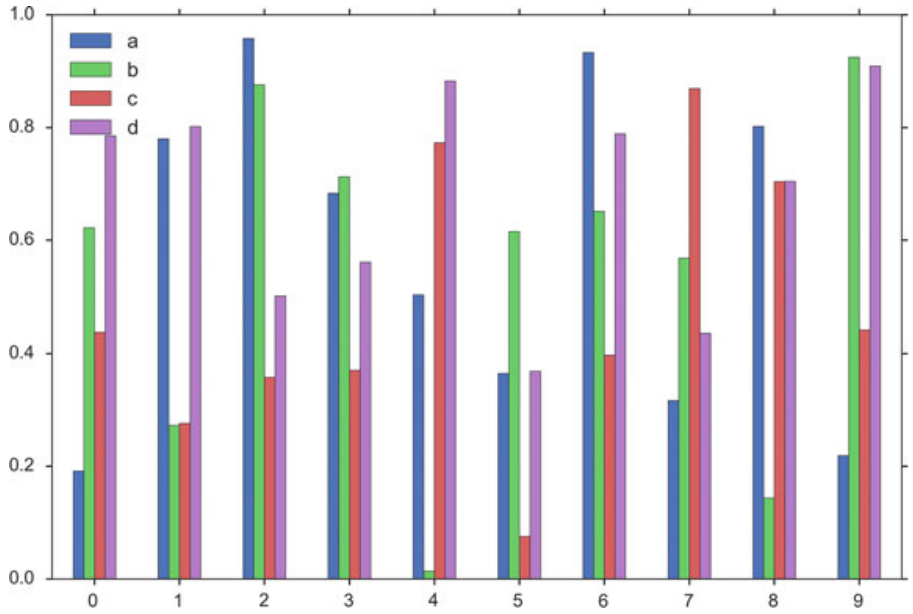

Fig. 4.9 Grouped barplot, produced with *pandas*

h) Pie Charts

Pie charts can be generated with a number of different options, e.g.

```
import seaborn as sns
import matplotlib.pyplot as plt

txtLabels = 'Cats', 'Dogs', 'Frogs', 'Others'
fractions = [45, 30, 15, 10]
offsets =(0, 0.05, 0, 0)

plt.pie(fractions, explode=offsets, labels=txtLabels,
         autopct='%1.1f%%', shadow=True, startangle=90,
         colors=sns.color_palette('muted') )
plt.axis('equal')
```

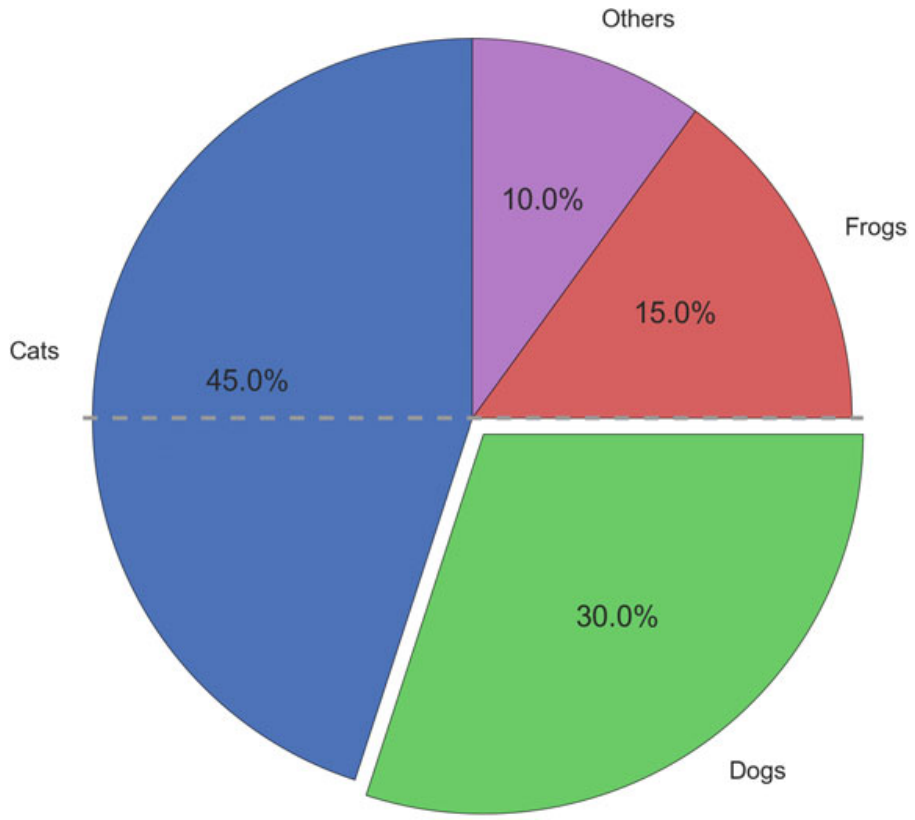

Fig. 4.10 "Sometimes it is raining cats and dogs"

i) Programs: Data Display

Code: "ISP_showPlots.py"[3] shows how the plots in this section have been generated.

4.3.2 *Bivariate and Multivariate Plots*

a) Bivariate Scatter Plots

Simple scatter plots are trivial. But *pandas* also makes fancy scatter plots easy:

```
df2 = pd.DataFrame(np.random.rand(50, 4),
                   columns=['a', 'b', 'c', 'd'])
df2.plot(kind='scatter', x='a', y='b', s=df['c']*300);
```

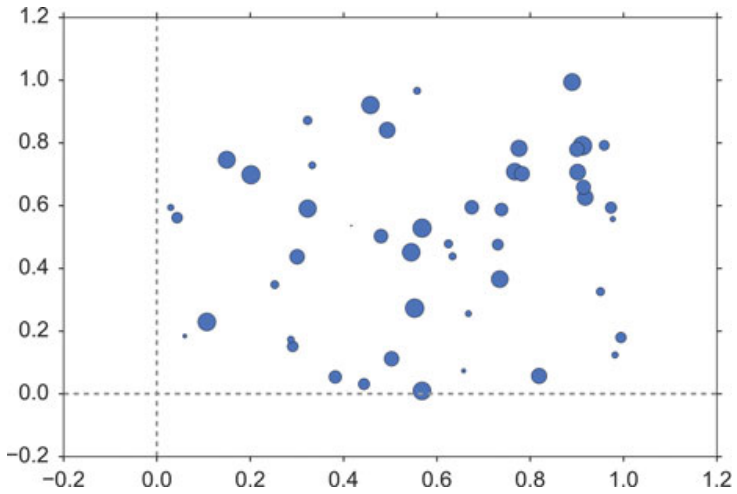

Fig. 4.11 Scatterplot, with scaled datapoints

[3] https://github.com/thomas-haslwanter/statsintro_python/tree/master/ISP/Code_Quantlets/04_DataDisplay/showPlots.

b) 3D Plots

3D plots in *matplotlib* are a bit awkward, because separate modules have to be imported, and axes for 3D plots have to be explicitly declared. However, once the axis is correctly defined, the rest is straightforward. Here are two examples:

```
# imports specific to the plots in this example
import numpy as np
from matplotlib import cm
from mpl_toolkits.mplot3d.axes3d import get_test_data

# Twice as wide as it is tall.
fig = plt.figure(figsize=plt.figaspect(0.5))

#---- First subplot
# Note that the declaration "projection='3d'"
# is required for 3d plots!
ax = fig.add_subplot(1, 2, 1, projection='3d')

# Generate the grid
X = np.arange(-5, 5, 0.1)
Y = np.arange(-5, 5, 0.1)
X, Y = np.meshgrid(X, Y)

# Generate the surface data
R = np.sqrt(X**2 + Y**2)
Z = np.sin(R)

# Plot the surface
surf = ax.plot_surface(X, Y, Z, rstride=1, cstride=1,
        cmap=cm.GnBu, linewidth=0, antialiased=False)
ax.set_zlim3d(-1.01, 1.01)

fig.colorbar(surf, shrink=0.5, aspect=10)

#---- Second subplot
ax = fig.add_subplot(1, 2, 2, projection='3d')
X, Y, Z = get_test_data(0.05)
ax.plot_wireframe(X, Y, Z, rstride=10, cstride=10)

outfile = '3dGraph.png'
plt.savefig(outfile, dpi=200)
print('Image saved to {0}'.format(outfile))
plt.show()
```

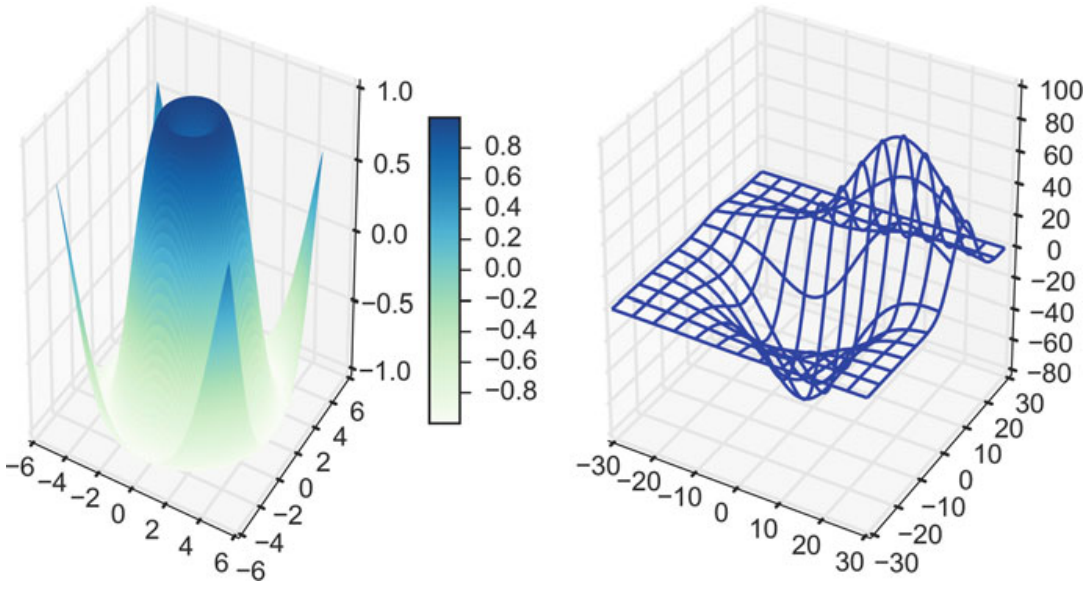

Fig. 4.12 Two types of 3D graphs. *(Left)* surface plot. *(Right)* wireframe plot

4.4 Exercises

4.1 Data Display

1. Read in the data from 'Data\amstat\babyboom.dat.txt'.
2. Inspect them visually, and give a numerical description of the data.
3. Are the data normally distributed?

Part II
Distributions and Hypothesis Tests

This part of the book moves the focus from *Python* to statistics.

The first chapter serves to define the statistical basics, like the concepts of *populations* and *samples*, and of *probability distributions*. It also includes a short overview of *study design*. The design of statistical studies is seriously underestimated by most beginning researchers: faulty study design produces garbage data, and the best analysis cannot remedy those problems ("Garbage in–garbage out"). However, if the study design is good, but the analysis faulty, the situation can be fixed with a new analysis, which typically takes much less time than an entirely new study.

The next chapter shows how to characterize the position and the variability of a distribution, and then uses the normal distribution to describe the most important *Python* methods common to all distribution functions. After that, the most important discrete and continuous distributions are presented.

The third chapter in this part first describes a typical workflow in the analysis of statistical data. Then the concept of *hypothesis tests* is explained, as well as the different types of errors, and common concepts like *sensitivity* and *specificity.*

The remaining chapters explain the most important hypothesis tests, for continuous variables and for categorical variables. A separate chapter is dedicated to survival analysis (which also encompasses the statistical characterization of material failures and machine breakdowns), as this question requires a somewhat different approach than the other hypothesis tests presented here. Each of these chapters also includes working *Python* sample code (including the required data) for each of the tests presented. This should make it easy to implement the tests for different data sets.