

Files are used for permanent storage of information on a computer. From previous computer experience you are hopefully used to save information to files and open the files at a later time for inspection again. The present chapter tells you how Python programs can access information in files (Chapter 6.1) and also create new files (Chapter 6.5). The chapter builds on programming concepts introduced in the first four chapters of this book.

Since files often contain structured information that one wants to map to objects in a running program, there is a need for flexible objects where various kinds of other objects can be stored. Dictionaries are very handy for this purpose and are described in Chapter 6.2.

Information in files often appear as pure text, so to interpret and extract data from files it is sometimes necessary to carry out sophisticated operations on the text. Python strings have many methods for performing such operations, and the most important functionality is described in Chapter 6.3.

The World Wide Web is full of information and scientific data that may be useful to access from a program. Chapter 6.4 tells you how to read web pages from a program and interpret the contents using string operations.

The folder `src/files` contains all the program example files referred to in the present chapter.

## 6.1 Reading Data from File

Suppose we have recorded some measurements in a file `data1.txt`, located in the `src/files` folder. The goal of our first example of reading files is to read the measurement values in `data1.txt`, find the average value, and print it out in the terminal window.

Before trying to let a program read a file, we must know the *file format*, i.e., what the contents of the file looks like, because the structure of the text in the file greatly influences the set of statements needed to read the file. We therefore start with viewing the contents of the file `data1.txt`. To this end, load the file into a text editor or viewer<sup>1</sup>. What we see is a column with numbers:

```
21.8
18.1
19
23
26
17.8
```

Our task is to read this column of numbers into a list in the program and compute the average of the list items.

### 6.1.1 Reading a File Line by Line

To read a file, we first need to *open* the file. This action creates a file object, here stored in the variable `infile`:

```
infile = open('data1.txt', 'r')
```

The second argument to the `open` function, the string `'r'`, tells that we want to open the file for reading. We shall later see that a file can be opened for writing instead, by providing `'w'` as the second argument. After the file is read, one should close the file object with `infile.close()`.

*For Loop over Lines.* We can read the contents of the file in various ways. The basic recipe for reading the file line by line applies a `for` loop like this:

```
for line in infile:
    # do something with line
```

The `line` variable is a string holding the current line in the file. The `for` loop over lines in a file has the same syntax as when we go through a list. Just think of the file object `infile` as a collection of elements, here lines in a file, and the `for` loop visits these elements in sequence such that the `line` variable refers to one line at a time. If something seemingly goes wrong in such a loop over lines in a file, it is useful to do a `print line` inside the loop.

Instead of reading one line at a time, we can load all lines into a list of strings (`lines`) by

---

<sup>1</sup> You can use `emacs`, `vim`, `more`, or `less` on Unix and Mac. On Windows, `WordPad` is appropriate, or the `type` command in a DOS window. Word processors such as OpenOffice or Microsoft Word can also be used.

```
lines = infile.readlines()
```

This statement is equivalent to

```
lines = []
for line in infile:
    lines.append(line)
```

or the list comprehension:

```
lines = [line for line in infile]
```

In the present example, we load the file into the list `lines`. The next task is to compute the average of the numbers in the file. Trying a straightforward sum of all numbers on all lines,

```
mean = 0
for number in lines:
    mean = mean + number
mean = mean/len(lines)
```

gives an error message:

```
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

The reason is that `lines` holds each line (`number`) as a string, not a float or int that we can add to other numbers. A fix is to convert each line to a float:

```
mean = 0
for line in lines:
    number = float(line)
    mean = mean + number
mean = mean/len(lines)
```

This code snippet works fine. The complete code can be found in the file `files/mean1.py`.

Summing up a list of numbers is often done in numerical programs, so Python has a special function `sum` for performing this task. However, `sum` must in the present case operate on a list of floats, not strings. We can use a list comprehension to turn all elements in `lines` into corresponding float objects:

```
mean = sum([float(line) for line in lines])/len(lines)
```

An alternative implementation is to load the lines into a list of float objects directly. Using this strategy, the complete program (found in file `mean2.py`) takes the form

```
infile = open('data1.txt', 'r')
numbers = [float(line) for line in infile.readlines()]
infile.close()
mean = sum(numbers)/len(numbers)
print mean
```

A newcomer to programming might find it confusing to see that one problem is solved by many alternative sets of statements, but this is the very nature of programming. A clever programmer will judge several alternative solutions to a programming task and choose one that is either particularly compact, easy to understand, and/or easy to extend later. We therefore present more examples on how to read the `data1.txt` file and compute with the data.

*While Loop over Lines.* The call `infile.readline()` returns a string containing the text at the current line. A new `infile.readline()` will read the next line. When `infile.readline()` returns an empty string, the end of the file is reached and we must stop further reading. The following `while` loop reads the file line by line using `infile.readline()`:

```
while True:
    line = infile.readline()
    if not line:
        break
    # process line
```

This is perhaps a somewhat strange loop, but it is a well-established way of reading a file in Python (especially in older codes). The shown `while` loop runs forever since the condition is always `True`. However, inside the loop we test if `line` is `False`, and it is `False` when we reach the end of the file, because `line` then becomes an empty string, which in Python evaluates to `False`. When `line` is `False`, the `break` statement breaks the loop and makes the program flow jump to the first statement after the `while` block.

Computing the average of the numbers in the `data1.txt` file can now be done in yet another way:

```
infile = open('data1.txt', 'r')
mean = 0
n = 0
while True:
    line = infile.readline()
    if not line:
        break
    mean += float(line)
    n += 1
mean = mean/float(n)
```

*Reading a File into a String.* The call `infile.read()` reads the whole file and returns the text as a string object. The following interactive session illustrates the use and result of `infile.read()`:

```
>>> infile = open('data1.txt', 'r')
>>> filestr = infile.read()
>>> filestr
'21.8\n18.1\n19\n23\n26\n17.8\n'
>>> print filestr
21.8
```

```
18.1
19
23
26
17.8
```

Note the difference between just writing `filestr` and writing `print filestr`. The former dumps the string with newlines as “backslash n” characters, while the latter is a “pretty print” where the string is written out without quotes and with the newline characters as visible line shifts<sup>2</sup>.

Having the numbers inside a string instead of inside a file does not look like a major step forward. However, string objects have many useful functions for extracting information. A very useful feature is *split*: `filestr.split()` will split the string into words (separated by blanks or any other sequence of characters you have defined). The “words” in this file are the numbers:

```
>>> words = filestr.split()
>>> words
['21.8', '18.1', '19', '23', '26', '17.8']
>>> numbers = [float(w) for w in words]
>>> mean = sum(numbers)/len(numbers)
>>> print mean
20.95
```

A more compact program looks as follows (`mean3.py`):

```
infile = open('data1.txt', 'r')
numbers = [float(w) for w in infile.read().split()]
mean = sum(numbers)/len(numbers)
```

The next section tells you more about splitting strings.

### 6.1.2 Reading a Mixture of Text and Numbers

The `data1.txt` file has a very simple structure since it contains numbers only. Many data files contain a mix of text and numbers. The file `rainfall.dat` provides an example<sup>3</sup>:

```
Average rainfall (in mm) in Rome: 1188 months between 1782 and 1970
Jan 81.2
Feb 63.2
Mar 70.3
Apr 55.7
May 53.0
Jun 36.4
Jul 17.5
Aug 27.5
Sep 60.9
Oct 117.7
Nov 111.0
Dec 97.9
Year 792.9
```

<sup>2</sup> The difference between these two outputs is explained in Chapter 7.3.9.

<sup>3</sup> <http://www.worldclimate.com/cgi-bin/data.pl?ref=N41E012+2100+1623501G1>

How can we read the rainfall data in this file and make a plot of the values?

The most straightforward solution is to read the file line by line, and for each line split the line into words, pick out the last (second) word on the line, convert this word to `float`, and store the `float` objects in a list. Having the rainfall values in a list of real numbers, we can make a plot of the values versus the month number. The complete code, wrapped in a function, may look like this (file `rainfall.py`):

```
def extract_data(filename):
    infile = open(filename, 'r')
    infile.readline() # skip the first line
    numbers = []
    for line in infile:
        words = line.split()
        number = float(words[1])
        numbers.append(number)
    infile.close()
    return numbers

values = extract_data('rainfall.dat')
from scitools.std import plot
month_indices = range(1, 13)
plot(month_indices, values[:-1], 'o2')
```

Note that the first line in the file is just a comment line and of no interest to us. We therefore read this line by `infile.readline()`. The `for` loop over the lines in the file will then start from the next (second) line.

Also note that `numbers` contain data for the 12 months plus the average annual rainfall. We want to plot the average rainfall for the months only, i.e., `values[0:12]` or simply `values[:-1]` (everything except the last entry). Along the “x” axis we put the index of a month, starting with 1. A call to `range(1,13)` generates these indices.

We can condense the `for` loop over lines in the file, if desired, by using a list comprehension:

```
def extract_data(filename):
    infile = open(filename, 'r')
    infile.readline() # skip the first line
    numbers = [float(line.split()[1]) for line in infile]
    infile.close()
    return numbers
```

### 6.1.3 What Is a File, Really?

This section is not mandatory for understanding the rest of the book. However, we think the information here is fundamental for understanding what files are about.

A file is simply a sequence of characters. In addition to the sequence of characters, a file has some data associated with it, typically the

name of the file, its location on the disk, and the file size. These data are stored somewhere by the operating system. Without this extra information beyond the pure file contents as a sequence of characters, the operating system cannot find a file with a given name on the disk.

Each character in the file is represented as a *byte*, consisting of eight *bits*. Each bit is either 0 or 1. The zeros and ones in a byte can be combined in  $2^8 = 256$  ways. This means that there are 256 different types of characters. Some of these characters can be recognized from the keyboard, but there are also characters that do not have a familiar symbol. The name of such characters looks cryptic when printed.

*Pure Text Files.* To see that a file is really just a sequence of characters, invoke an editor for plain text, e.g., the editor you use to write Python programs. Write the four characters ABCD into the editor, do not press the Return key, and save the text to a file `test1.txt`. Use your favorite tool for file and folder overview and move to the folder containing the `test1.txt` file. This tool may be Windows Explorer, My Computer, or a DOS window on Windows; a terminal window, Konqueror, or Nautilus on Linux; or a terminal window or Finder on Mac. If you choose a terminal window, use the `cd` (change directory) command to move to the proper folder and write `dir` (Windows) or `ls -l` (Linux/Mac) to list the files and their sizes. In a graphical program like Windows Explorer, Konqueror, Nautilus, or Finder, select a view that shows the *size* of each file<sup>4</sup>. You will see that the `test1.txt` file has a size of 4 bytes<sup>5</sup>. The 4 bytes are exactly the 4 characters ABCD in the file. Physically, the file is just a sequence of 4 bytes on your harddisk.

Go back to the editor again and add a newline by pressing the Return key. Save this new version of the file as `test2.txt`. When you now check the size of the file it has grown to five bytes. The reason is that we added a newline character (symbolically known as “backslash n”).

Instead of examining files via editors and folder viewers we may use Python interactively:

```
>>> file1 = open('test1.txt', 'r').read() # read file into string
>>> file1
'ABCD'
>>> len(file1)          # length of string in bytes/characters
4
>>> file2 = open('test2.txt', 'r').read()
>>> file2
'ABCD\n'
>>> len(file2)
5
```

Python has in fact a function that returns the size of a file directly:

<sup>4</sup> Choose “view as details” in Windows Explorer, “View as List” in Nautilus, the list view icon in Finder, or you just point at a file icon in Konqueror and watch the pop-up text.

<sup>5</sup> If you use `ls -l`, the size measured in bytes is found in column 5, right before the date.

```
>>> import os
>>> size = os.path.getsize('test1.txt')
>>> size
4
```

*Word Processor Files.* Most computer users write text in a word processing program, such as Microsoft Word or OpenOffice. Let us investigate what happens with our four characters ABCD in such a program. Start the word processor, open a new document, and type in the four characters ABCD only. Save the document as a .doc file (Microsoft Word) or an .odt file (OpenOffice). Load this file into an editor for pure text and look at the contents. You will see that there are numerous strange characters that you did not write (!). This additional “text” contains information on what type of document this is, the font you used, etc. The OpenOffice version of this file has 5725 bytes! However, if you save the file as a pure text file, with extension .txt, the size is not more than four bytes, and the text file contains just the corresponding characters ABCD.

Instead of loading the OpenOffice file into an editor we can again read the file contents into a string in Python and examine this string:

```
>>> infile = open('test3.odt', 'r') # open OpenOffice file
>>> s = infile.read()
>>> len(s) # file size
5725
>>> s
'PK\x03\x04\x14\x00\x00\x00\x00\x00r\x80E6^\xc62\x0c\...
\x00\x00mimetypeapplication/vnd.oasis.opendocument.textPK\x00...
\x00\x00content.xml\xa5VMS\xdb0\x10\xbd\x7fWx|\xe8\xcd\x11...'
```

Each backslash followed by x and a number is a code for a special character not found on the keyboard (recall that there are 256 characters and only a subset is associated with keyboard symbols). Although we show just a small portion of all the characters in this file in the above output<sup>6</sup>, we can guarantee that you cannot find the pure sequence of characters ABCD. However, the computer program that generated the file, OpenOffice in this example, can easily interpret the meaning of all the characters in the file and translate the information into nice, readable text on the screen.

*Image Files.* A digital image – captured by a digital camera or a mobile phone – is a file. And since it is a file, the image is just a sequence of characters. Loading some JPEG file into a pure text editor, you can see all the strange characters in there. On the first line you will (normally) find some recognizable text in between the strange characters. This text reflects the type of camera used to capture the image and the date and time when the picture was taken. The next lines contain

<sup>6</sup> Otherwise, the output would have occupied several pages in this book with about five thousand backslash-x-number symbols...



more information about the image. Thereafter, the file contains a set of numbers representing the image. The basic representation of an image is a set of  $m \times n$  pixels, where each pixel has a color represented as a combination of 256 values of red, green, and blue. A 6 megapixel camera will then need to store  $256 \times 3$  bytes for each of the 6,000,000 pixels, which results in 4,608,000,000 bytes (or 4.6 gigabytes, written 4.6 Gb). The JPEG file contains only a couple of megabytes. The reason is that JPEG is a *compressed* file format, produced by applying a smart technique that can throw away pixel information in the original picture such that the human eye hardly can detect the inferior quality.

A video is just a sequence of images, and therefore a video is also a stream of bytes. If the change from one video frame (image) to the next is small, one can use smart methods to compress the image information in time. Such compression is particularly important for videos since the file sizes soon get too large for being transferred over the Internet. A small video file occasionally has bad visual quality, caused by too much compression.

*Music Files.* An MP3 file is much like a JPEG file: First, there is some information about the music (artist, title, album, etc.), and then comes the music itself as a stream of bytes. A typical MP3 file has a size of something like five million bytes<sup>7</sup>, i.e., five megabytes (5 Mb). On a 2 Gb MP3 player you can then store roughly  $2,000,000,000 / 5,000,000 = 400$  MP3 files. MP3 is, like JPEG, a compressed format. The complete data of a song on a CD (the WAV file) contains about ten times as many bytes. As for pictures, the idea is that one can throw away a lot of bytes in an intelligent way, such that the human ear hardly detects the difference between a compressed and uncompressed version of the music file.

*PDF Files.* Looking at a PDF file in a pure text editor shows that the file contains some readable text mixed with some unreadable characters. It is not possible for a human to look at the stream of bytes and deduce the text in the document<sup>8</sup>. A PDF file reader can easily interpret the contents of the file and display the text in a human-readable form on the screen.

*Remarks.* We have repeated many times that a file is just a stream of bytes. A human can interpret (read) the stream of bytes if it makes sense in a human language – or a computer language (provided the human is a programmer). When the series of bytes does not make

<sup>7</sup> The exact size depends on the complexity of the music, the length of the track, and the MP3 resolution.

<sup>8</sup> From the assumption that there are always some strange people doing strange things, there might be somebody out there who – with a lot of training – can interpret the pure PDF code with the eyes.

sense to any human, a computer program must be used to interpret the sequence of characters.

Think of a report. When you write the report as pure text in a text editor, the resulting file contains just the characters you typed in from the keyboard. On the other hand, if you applied a word processor like Microsoft Word or OpenOffice, the report file contains a large number of extra bytes describing properties of the formatting of the text. This stream of extra bytes does not make sense to a human, and a computer program is required to interpret the file content and display it in a form that a human can understand. Behind the sequence of bytes in the file there are strict rules telling what the series of bytes means. These rules reflect the *file format*. When the rules or file format is publicly documented, a programmer can use this documentation to make her own program for interpreting the file contents<sup>9</sup>. It happens, though, that secret file formats are used, which require certain programs from certain companies to interpret the files.

## 6.2 Dictionaries

So far in the book we have stored information in various types of objects, such as numbers, strings, list, and arrays. A *dictionary* is a very flexible object for storing various kind of information, and in particular when reading files. It is therefore time to introduce the dictionary type.

A list is a collection of objects indexed by an integer going from 0 to the number of elements minus one. Instead of looking up an element through an integer index, it can be more handy to use a text. Roughly speaking, a list where the index can be a text is called a dictionary in Python. Other computer languages use other names for the same thing: HashMap, hash, associative array, or map.

### 6.2.1 Making Dictionaries

Suppose we need to store the temperatures from three cities: Oslo, London, and Paris. For this purpose we can use a list,

```
temps = [13, 15.4, 17.5]
```

but then we need to remember the sequence of cities, e.g., that index 0 corresponds to Oslo, index 1 to London, and index 2 to Paris. That is, the London temperature is obtained as `temps[1]`. A dictionary with the city name as index is more convenient, because this allows us to

---

<sup>9</sup> Interpreting such files is much more complicated than our examples on reading human-readable files in this book.

write `temps['London']` to look up the temperature in London. Such a dictionary is created by one of the following two statements

```
temps = {'Oslo': 13, 'London': 15.4, 'Paris': 17.5}
# or
temps = dict(Oslo=13, London=15.4, Paris=17.5)
```

Additional text-value pairs can be added when desired. We can, for instance, write

```
temps['Madrid'] = 26.0
```

The `temps` dictionary has now four text-value pairs, and a `print temps` yields

```
{'Oslo': 13, 'London': 15.4, 'Paris': 17.5, 'Madrid': 26.0}
```

### 6.2.2 Dictionary Operations

The string “indices” in a dictionary are called *keys*. To loop over the keys in a dictionary `d`, one writes `for key in d:` and works with `key` and the corresponding value `d[key]` inside the loop. We may apply this technique to write out the temperatures in the `temps` dictionary from the previous paragraph:

```
>>> for city in temps:
...     print 'The temperature in %s is %g' % (city, temps[city])
...
The temperature in Paris is 17.5
The temperature in Oslo is 13
The temperature in London is 15.4
The temperature in Madrid is 26
```

We can check if a key is present in a dictionary by the syntax `if key in d:`

```
>>> if 'Berlin' in temps:
...     print 'Berlin:', temps['Berlin']
... else:
...     print 'No temperature data for Berlin'
...
No temperature data for Berlin
```

Writing `key in d` yields a standard boolean expression, e.g.,

```
>>> 'Oslo' in temps
True
```

The keys and values can be extracted as lists from a dictionary:

```
>>> temps.keys()
['Paris', 'Oslo', 'London', 'Madrid']
>>> temps.values()
[17.5, 13, 15.4, 26.0]
```

An important feature of the `keys` method in dictionaries is that the order of the returned list of keys is unpredictable. If you need to traverse the keys in a certain order, you will need to sort the keys. A loop over the keys in the `temps` dictionary in alphabetic order is written as

```
>>> for city in sorted(temps):
...     print city
...
London
Madrid
Oslo
Paris
```

A key-value pair can be removed by `del d[key]`:

```
>>> del temps['Oslo']
>>> temps
{'Paris': 17.5, 'London': 15.4, 'Madrid': 26.0}
>>> len(temps) # no of key-value pairs in dictionary
3
```

Sometimes we need to take a copy of a dictionary:

```
>>> temps_copy = temps.copy()
>>> del temps_copy['Paris'] # this does not affect temps
>>> temps_copy
{'London': 15.4, 'Madrid': 26.0}
>>> temps
{'Paris': 17.5, 'London': 15.4, 'Madrid': 26.0}
```

Note that if two variables refer to the same dictionary and we change the contents of the dictionary through either of the variables, the change will be seen in both variables:

```
>>> t1 = temps
>>> t1['Stockholm'] = 10.0 # change t1
>>> temps # temps is also changed
{'Stockholm': 10.0, 'Paris': 17.5, 'London': 15.4, 'Madrid': 26.0}
```

To avoid that `temps` is affected by adding a new key-value pair to `t1`, `t1` must be a copy of `temps`.

### 6.2.3 Example: Polynomials as Dictionaries

The keys in a dictionary are not restricted to be strings. In fact, any Python object whose contents cannot be changed can be used as key<sup>10</sup>. For example, we may use integers as keys in a dictionary. This is a handy way of representing polynomials, as will be explained next.

Consider the polynomial

<sup>10</sup> Such objects are known as *immutable* data types and consist of `int`, `float`, `complex`, `str`, and `tuple`. Lists and dictionaries can change their contents and are called *mutable* objects. These cannot be used as keys in dictionaries. If you desire a list as key, use a tuple instead.

$$p(x) = -1 + x^2 + 3x^7.$$

The data associated with this polynomial can be viewed as a set of power-coefficient pairs, in this case the coefficient  $-1$  belongs to power 0, the coefficient 1 belongs to power 2, and the coefficient 3 belongs to power 7. A dictionary can be used to map a power to a coefficient:

```
p = {0: -1, 2: 1, 7: 3}
```

A list can, of course, also be used, but in this case we must fill in all the zero coefficients too, since the index must match the power:

```
p = [-1, 0, 1, 0, 0, 0, 0, 3]
```

The advantage with a dictionary is that we need to store only the non-zero coefficients. For the polynomial  $1 + x^{100}$  the dictionary holds two elements while the list holds 101 elements (see Exercise 6.16).

The following function can be used to evaluate a polynomial represented as a dictionary:

```
def poly1(data, x):
    sum = 0.0
    for power in data:
        sum += data[power]*x**power
    return sum
```

The `data` argument must be a dictionary where `data[power]` holds the coefficient associated with the term `x**power`. A more compact implementation can make use of Python's `sum` function to sum the elements of a list:

```
def poly1(data, x):
    return sum([data[p]*x**p for p in data])
```

That is, we first make a list of the terms in the polynomial using a list comprehension, and then we feed this list to the `sum` function. Note that the name `sum` is different in the two implementations: In the first, `sum` is a `float` object, and in the second, `sum` is a function. When we set `sum=0.0` in the first implementation, we bind the name `sum` to a new `float` object, and the built-in Python function associated with the name `sum` is then no longer accessible inside the `poly1` function<sup>11</sup>. Outside the function, nevertheless, `sum` will be the summation function (unless we have bound the global name `sum` to another object somewhere else in the main program – see Chapter 2.4.2 for a discussion of this issue).

With a list instead of dictionary for representing the polynomial, a slightly different evaluation function is needed:

<sup>11</sup> This is not strictly correct, because `sum` is a local variable while the summation function is associated with a global name `sum`, which can be reached through `globals()['sum']`.

```
def poly2(data, x):
    sum = 0
    for power in range(len(data)):
        sum += data[power]*x**power
    return sum
```

If there are many zeros in the `data` list, `poly2` must perform all the multiplications with the zeros, while `poly1` computes with the non-zero coefficients only and is hence more efficient.

Another major advantage of using a dictionary to represent a polynomial rather than a list is that negative powers are easily allowed, e.g.,

```
p = {-3: 0.5, 4: 2}
```

can represent  $\frac{1}{2}x^{-3} + 2x^4$ . With a list representation, negative powers require much more book-keeping. We may, for example, set

```
p = [0.5, 0, 0, 0, 0, 0, 0, 2]
```

and remember that `p[i]` is the coefficient associated with the power `i-3`. In particular, the `poly2` function will no longer work for such lists, while the `poly1` function works also for dictionaries with negative keys (powers).

You are now encouraged to solve Exercise 6.17 on page 327 to become more familiar with the concept of dictionaries.

### 6.2.4 Example: File Data in Dictionaries

*Problem.* The file `files/densities.dat` contains a table of densities of various substances measured in  $\text{g/cm}^3$ :

air	0.0012
gasoline	0.67
ice	0.9
pure water	1.0
seawater	1.025
human body	1.03
limestone	2.6
granite	2.7
iron	7.8
silver	10.5
mercury	13.6
gold	18.9
platinum	21.4
Earth mean	5.52
Earth core	13
Moon	3.3
Sun mean	1.4
Sun core	160
proton	2.8E+14

In a program we want to access these density data. A dictionary with the name of the substance as key and the corresponding density as value seems well suited for storing the data.

*Solution.* We can read the `densities.dat` file line by line, split each line into words, use a float conversion of the last word as density value, and the remaining one or two words as key in the dictionary.

```
def read_densities(filename):
    infile = open(filename, 'r')
    densities = {}
    for line in infile:
        words = line.split()
        density = float(words[-1])

        if len(words[:-1]) == 2:
            substance = words[0] + ' ' + words[1]
        else:
            substance = words[0]

        densities[substance] = density
    infile.close()
    return densities

densities = read_densities('densities.dat')
```

This code is found in the file `density.py`. With string operations from Chapter 6.3.1 we can avoid the special treatment of one or two words in the name of the substance and achieve simpler and more general code, see Exercise 6.10.

### 6.2.5 Example: File Data in Nested Dictionaries

*Problem.* We are given a data file with measurements of some properties with given names (here A, B, C ...). Each property is measured a given number of times. The data are organized as a table where the rows contain the measurements and the columns represent the measured properties:

	A	B	C	D
1	11.7	0.035	2017	99.1
2	9.2	0.037	2019	101.2
3	12.2	no	no	105.2
4	10.1	0.031	no	102.1
5	9.1	0.033	2009	103.3
6	8.7	0.036	2015	101.9

The word “no” stands for no data, i.e., we lack a measurement. We want to read this table into a dictionary `data` so that we can look up measurement no. `i` of (say) property `C` as `data['C'][i]`. For each property `p`, we want to compute the mean of all measurements and store this as `data[p]['mean']`.

*Algorithm.* The algorithm for creating the `data` dictionary goes as follows:

```

examine the first line: split it into words and
                        initialize a dictionary with the property names
                        as keys and empty dictionaries ({}) as values
for each of the remaining lines in the file:
    split the line into words
    for each word after the first:
        if the word is not "no":
            transform the word to a real number and store
            the number in the relevant dictionary

```

*Implementation.* The solution requires familiarity with dictionaries and list slices (also called sublists, see Chapter 2.1.9). A new aspect needed in the solution is *nested dictionaries*, that is, dictionaries of dictionaries. The latter topic is first explained, via an example:

```
>>> d = {'key1': {'key1': 2, 'key2': 3}, 'key2': 7}
```

Observe here that the value of `d['key1']` is a dictionary which we can index with its keys `key1` and `key2`:

```

>>> d['key1']           # this is a dictionary
{'key2': 3, 'key1': 2}
>>> type(d['key1'])     # proof
<type 'dict'>
>>> d['key1']['key1']   # index a nested dictionary
2
>>> d['key1']['key2']
3

```

In other words, repeated indexing works for nested dictionaries as for nested lists. The repeated indexing does not apply to `d['key2']` since that value is just an integer:

```

>>> d['key2']['key1']
...
TypeError: unsubscriptable object
>>> type(d['key2'])
<type 'int'>

```

When we have understood the concept of nested dictionaries, we are in a position to present a complete code that solves our problem of loading the tabular data in the file `table.dat` into a nested dictionary `data` and computing mean values. First, we list the program, stored in the file `table2dict.py`, and display the program's output. Thereafter, we dissect the code in detail.

```

infile = open('table.dat', 'r')
lines = infile.readlines()
infile.close()
data = {} # data[property][measurement_no] = propertyvalue
first_line = lines[0]
properties = first_line.split()
for p in properties:

```



```

    data[p] = {}

for line in lines[1:]:
    words = line.split()
    i = int(words[0])      # measurement number
    values = words[1:]     # values of properties
    for p, v in zip(properties, values):
        if v != 'no':
            data[p][i] = float(v)

# compute mean values:
for p in data:
    values = data[p].values()
    data[p]['mean'] = sum(values)/len(values)

for p in sorted(data):
    print 'Mean value of property %s = %g' % (p, data[p]['mean'])

```

The corresponding output from this program becomes

```

Mean value of property A = 9.875
Mean value of property B = 0.0279167
Mean value of property C = 1678.42
Mean value of property D = 98.0417

```

To view the nested data dictionary, we may insert

```
import scitools.pprint2; scitools.pprint2.pprint(temps)
```

which produces something like

```

{'A': {1: 11.7, 2: 9.2, 3: 12.2, 4: 10.1, 5: 9.1, 6: 8.7,
      'mean': 10.1667},
 'B': {1: 0.035, 2: 0.037, 4: 0.031, 5: 0.033, 6: 0.036,
      'mean': 0.0344},
 'C': {1: 2017, 2: 2019, 5: 2009, 6: 2015, 'mean': 2015},
 'D': {1: 99.1,
      2: 101.2,
      3: 105.2,
      4: 102.1,
      5: 103.3,
      6: 101.9,
      'mean': 102.133}}

```

*Dissection.* To understand a computer program, you need to understand what the result of every statement is. Let us work through the code, almost line by line, and see what it does.

First, we load all the lines of the file into a list of strings called `lines`. The `first_line` variable refers to the string

```
'      A      B      C      D'
```

We split this line into a list of words, called `properties`, which then contains

```
['A', 'B', 'C', 'D']
```

With each of these property names we associate a dictionary with the measurement number as key and the property value as value, but first we must create these “inner” dictionaries as empty before we can add the measurements:

```
for p in properties:
    data[p] = {}
```

The first pass in the `for` loop picks out the string

```
'1      11.7    0.035    2017    99.1'
```

as the `line` variable. We split this line into words, the first word (`words[0]`) is the measurement number, while the rest `words[1:]` is a list of property values, here named `values`. To pair up the right properties and values, we loop over the `properties` and `values` lists simultaneously:

```
for p, v in zip(properties, values):
    if v != 'no':
        data[p][i] = float(v)
```

Recall that some values may be missing and we drop to record that value<sup>12</sup>. Because the `values` list contains strings (words) read from the file, we need to explicitly transform each string to a `float` number before we can compute with the values.

After the `for line in lines[1:]` loop, we have a dictionary `data` of dictionaries where all the property values are stored for each measurement number and property name. Figure 6.1 shows a graphical representation of the `data` dictionary.

It remains to compute the average values. For each property name `p`, i.e., key in the `data` dictionary, we can extract the recorded values as the list `data[p].values()` and simply send this list to Python's `sum` function and divide by the number of measured values for this property, i.e., the length of the list:

```
for p in data:
    values = data[p].values()
    data[p]['mean'] = sum(values)/len(values)
```

Alternatively, we can write an explicit loop to compute the average:

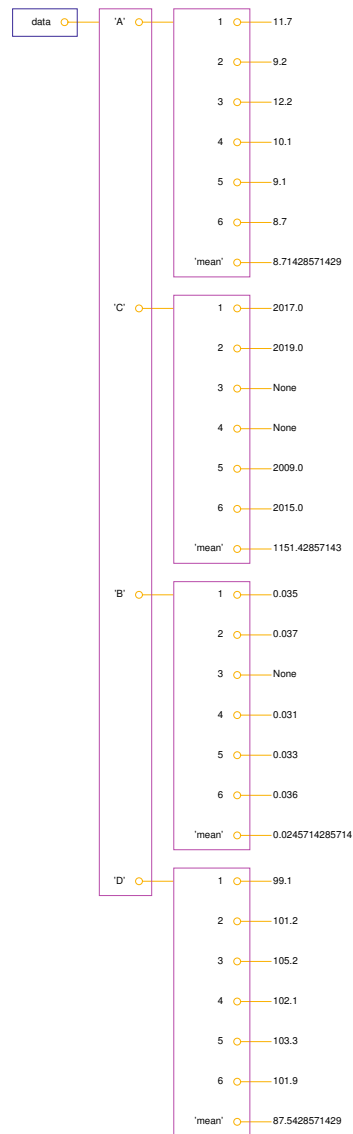
```
for p in data:
    sum_values = 0
    for value in data[p]:
        sum_values += value
    data[p]['mean'] = sum_values/len(data[p])
```

When we want to look up a measurement no. `n` of property `B`, we must recall that this particular measurement may be missing so we must do a test if `n` is key in the dictionary `data[p]`:

```
if n in data['B']:
    value = data['B'][n]

# alternative:
value = data['B'][n] if n in data['B'] else None
```

<sup>12</sup> We could, alternatively, set the value to `None`.



**Fig. 6.1** Illustration of the nested dictionary created in the `table2dict.py` program.

### 6.2.6 Example: Comparing Stock Prices

*Problem.* We want to compare the evolution of the stock prices of three giant companies in the computer industry: Microsoft, Sun Microsystems, and Google. Relevant data files for stock prices can be downloaded from *finance.yahoo.com*. Fill in the company's name and click on "GET QUOTES" in the top bar of this page, then choose "Historical Prices". On the resulting web page we can specify start and end dates for the historical prices of the stock. We let this be January 1, 1988, for Microsoft and Sun, and January 1, 2005, for Google. The end dates were set to June 1, 2008, in this example. Ticking off "Monthly"

values and clicking “Get Prices” result in a table of stock prices. We can download the data in a tabular format by clicking “Download To Spreadsheet” below the table. Here is an example of such a file:

```
Date,Open,High,Low,Close,Volume,Adj Close
2008-06-02,12.91,13.06,10.76,10.88,16945700,10.88
2008-05-01,15.50,16.37,12.37,12.95,26140700,12.95
2008-04-01,15.78,16.23,14.62,15.66,10330100,15.66
2008-03-03,16.35,17.38,15.41,15.53,12238800,15.53
2008-02-01,17.47,18.03,16.06,16.40,12147900,16.40
2008-01-02,17.98,18.14,14.20,17.50,15156100,17.50
2007-12-03,20.61,21.55,17.96,18.13,9869900,18.13
2007-11-01,5.65,21.60,5.10,20.78,17081500,20.78
```

The file format is simple: columns are separated by comma, the first line contains column headings, and the data lines have the date in the first column and various measures of stock prices in the next columns. Reading about the meaning of the various data on the Yahoo! web pages reveals that our interest concerns the final column (these prices are adjusted for splits and dividends). Three relevant data files can be found in `src/files` with the names `company_monthly.csv`, where `company` is Microsoft, Sun, or Google.

The task is to plot the evolution of stock prices of the three companies. It is natural to scale the prices to start at a unit value in January 1988 and let the Google price start at the maximum of the Sun and Microsoft stock values in January 2005.

*Solution.* There are two major parts of this problem: (i) reading the file and (ii) plotting the data. The reading part is quite straightforward, while the plotting part needs some special considerations since the “x” values in the plot are dates and not real numbers. In the forthcoming text we solve the individual subproblems one by one, showing the relevant Python snippets. The complete program is found in the file `stockprices.py`.

We start with the reading part. Since the reading will be repeated for three files, we make a function with the filename as argument. The result of reading a file should be two lists (or arrays) with the dates and the stock prices, respectively. We therefore return these two lists from the function. The algorithm for reading the data goes as follows:

```
open the file
create two empty lists, dates and prices, for collecting the data
read the first line (of no interest)
for each line in the rest of the file:
    split the line wrt. colon
    append the first word on the line to the dates list
    append the last word on the line to the prices list
close the file
```

There are a couple of additional points to consider. First, the words on a line are strings, and at least the prices (last word) should be

converted to a float. The first word, the date, has the form year-month-day (e.g., 2008-02-04). Since we asked for monthly data only, the day part is of no interest. Skipping the day part can be done by extracting a substring of the date string: `date[:-3]`, which means everything in the string except the last three characters (see Chapter 6.3.1 for more on substrings). The remaining date specification is now of the form year-month (e.g., 2008-02), represented as a string. Turning this into a number for plotting is not so easy, so we keep this string as it is in the list of dates.

The second point of consideration in the algorithm above is the sequence of data in the two lists: the files have the most recent date at the top and the oldest at the bottom, while it is natural to plot the evolution of stock prices against *increasing* time. Therefore, we must reverse the two lists of data before we return them to the calling code.

The algorithm above, together with the two additional comments, can now be translated into Python code:

```
def read_file(filename):
    infile = open(filename, 'r')
    infile.readline() # read column headings
    dates = []; prices = []
    for line in infile:
        columns = line.split(',')
        date = columns[0]
        date = date[:-3] # skip day of month
        price = columns[-1]
        dates.append(date)
        prices.append(float(price))
    infile.close()
    dates.reverse()
    prices.reverse()
    return dates, prices
```

The reading of a file is done by a call to this function, e.g.,

```
dates_Google, prices_Google = read_file('stockprices_Google.csv')
```

Instead of working with separate variables for the file data, we may collect the data in dictionaries, with the company name as key. One possibility is to use two dictionaries:

```
dates = {}; prices = {}
d, p = read_file('stockprices_Sun.csv')
dates['Sun'] = d; prices['Sun'] = p
d, p = read_file('stockprices_Microsoft.csv')
dates['MS'] = d; prices['MS'] = p
d, p = read_file('stockprices_Google.csv')
dates['Google'] = d; prices['Google'] = p
```

We can also collect the dates and prices dictionaries in a dictionary data:

```
data = {'prices': prices, 'dates': dates}
```

Note that `data` is a nested dictionary, so that to extract, e.g., the prices of the Microsoft stock, one writes `data['prices']['MS']`.

The next task is to normalize the stock prices so that we can easily compare them. The idea is to let Sun and Microsoft start out with a unit price and let Google start out with the best of the Sun and Microsoft prices. Normalizing the Sun and Microsoft prices is done by dividing by the first prices:

```
norm_price = prices['Sun'][0]
prices['Sun'] = [p/norm_price for p in prices['Sun']]
```

with a similar code for the Microsoft prices. Normalizing the Google prices is more involved as we need to extract the prices of Sun and Microsoft stocks from January 2005. Since the `dates` and `prices` lists correspond to each other, element by element, we can get the index corresponding to the date '2005-01' in the list of dates and use this index to extract the corresponding price. The normalization can then be coded as

```
jan05_MS = prices['MS'][dates['MS'].index('2005-01')]
jan05_Sun = prices['Sun'][dates['Sun'].index('2005-01')]
norm_price = prices['Google'][0]/max(jan05_MS, jan05_Sun)
prices['Google'] = [p/norm_price for p in prices['Google']]
```

The purpose of the final plot is to show how the prices evolve in time. The problem is that our time data consists of strings of the form year-month. We need to convert this string information to some “x” coordinate information in the plot. The simplest strategy is to just plot the prices against the list index, i.e., the “x” coordinates correspond to counting months. Suitable lists of monthly based indices for Sun and Microsoft are straightforward to create with the `range` function:

```
x = {}
x['Sun'] = range(len(prices['Sun']))
x['MS'] = range(len(prices['MS']))
```

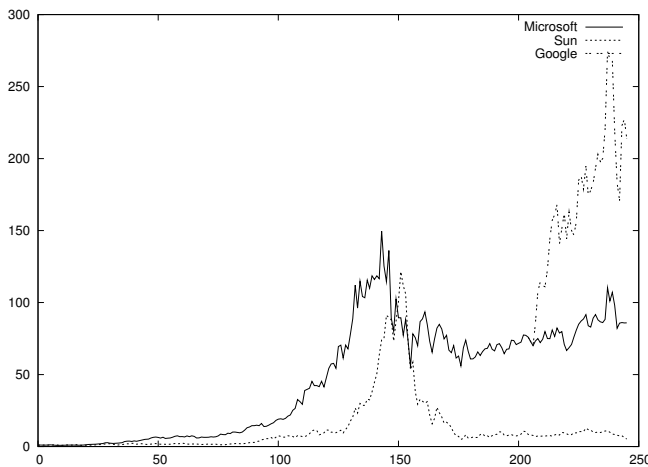
The “x” coordinates for the Google prices are somewhat more complicated, because the indices must start at the index corresponding to January 2005 in the Sun and Microsoft data. However, we extracted that index in the normalization of the Google prices, so we have already done most of the work:

```
jan05 = dates['Sun'].index('2005-01')
x['Google'] = range(jan05, jan05 + len(prices['Google']), 1)
```

The final step is to plot the three set of data:

```
from scitools.std import plot
plot(x['MS'], prices['MS'], 'r-',
     x['Sun'], prices['Sun'], 'b-',
     x['Google'], prices['Google'], 'y-',
     legend=('Microsoft', 'Sun', 'Google'))
```

Figure 6.2 displays the resulting plot. As seen from the plot, the best investment would be to start with Microsoft stocks in 1988 and switch all the money to Google stocks in 2005. You can easily modify the program to explore what would happen if you started out with Sun stocks and switched to Google in 2005.



**Fig. 6.2** The evolution of stock prices for three companies in the period January 1998 to June 2008.

*Generalization.* We can quite easily generalize the program to handle data from an arbitrary collection of companies, at least if we restrict the time period to be the same for all stocks. Exercise 6.18 asks you to do this. As you will realize, the use of dictionaries instead of separate variables in our program constitutes one important reason why the program becomes easy to extend. Avoiding different time periods for different price data also makes the generalized program simpler than the one we developed above.

## 6.3 Strings

Many programs need to manipulate text. For example, when we read the contents of a file into a string or list of strings (lines), we may want to change parts of the text in the string(s) – and maybe write out the modified text to a new file. So far in this chapter we have converted parts of the text to numbers and computed with the numbers. Now it is time to learn how to manipulate the text strings themselves.

### 6.3.1 Common Operations on Strings

Python has a rich set of operations on string objects. Some of the most common operations are listed below.

*Substring Specification.* The expression `s[i:j]` extracts the substring starting with character number `i` and ending with character number `j-1` (similarly to lists, 0 is the index of the first character):

```
>>> s = 'Berlin: 18.4 C at 4 pm'
>>> s[8:]      # from index 8 to the end of the string
'18.4 C at 4 pm'
>>> s[8:12]    # index 8, 9, 10 and 11 (not 12!)
'18.4'
```

A negative upper index counts, as usual, from the right such that `s[-1]` is the last element, `s[-2]` is the next last element, and so on.

```
>>> s[8:-1]
'18.4 C at 4 p'
>>> s[8:-8]
'18.4 C'
```

*Searching for Substrings.* The call `s.find(s1)` returns the index where the substring `s1` first appears in `s`. If the substring is not found, -1 is returned.

```
>>> s.find('Berlin') # where does 'Berlin' start?
0
>>> s.find('pm')
20
>>> s.find('Oslo')   # not found
-1
```

Sometimes the aim is to just check if a string is contained in another string, and then we can use the syntax:

```
>>> 'Berlin' in s:
True
>>> 'Oslo' in s:
False
```

Here is a typical use of the latter construction in an `if` test:

```
>>> if 'C' in s:
...     print 'C found'
... else:
...     print 'no C'
...
C found
```

Two other convenient methods for checking if a string starts with or ends with a specified string are `startswith` and `endswith`:



```
>>> s.startswith('Berlin')
True
>>> s.endswith('am')
False
```

*Substitution.* The call `s.replace(s1, s2)` replaces substring `s1` by `s2` everywhere in `s`:

```
>>> s.replace(' ', '_')
'Berlin:_18.4_C__at_4_pm'
>>> s.replace('Berlin', 'Bonn')
'Bonn: 18.4 C at 4 pm'
```

A variant of the last example, where several string operations are put together, consists of replacing the text before the first colon<sup>13</sup>:

```
>>> s.replace(s[:s.find(':')], 'Bonn')
'Bonn: 18.4 C at 4 pm'
```

*String Splitting.* The call `s.split()` splits the string `s` into words separated by whitespace (space, tabulator, or newline):

```
>>> s.split()
['Berlin:', '18.4', 'C', 'at', '4', 'pm']
```

Splitting a string `s` into words separated by a text `t` can be done by `s.split(t)`. For example, we may split with respect to colon:

```
>>> s.split(':')
['Berlin', ' 18.4 C at 4 pm']
```

We know that `s` contains a city name, a colon, a temperature, and then `C`:

```
>>> s = 'Berlin: 18.4 C at 4 pm'
```

With `s.splitlines()`, a multi-line string is split into lines (very useful when a file has been read into a string and we want a list of lines):

```
>>> t = '1st line\n2nd line\n3rd line'
>>> print t
1st line
2nd line
3rd line
>>> t.splitlines()
['1st line', '2nd line', '3rd line']
```

---

<sup>13</sup> Take a “break” and convince yourself that you understand how we specify the substring to be replaced.

*Upper and Lower Case.* `s.lower()` transforms all characters to their lower case equivalents, and `s.upper()` performs a similar transformation to upper case letters:

```
>>> s.lower()
'berlin: 18.4 c at 4 pm'
>>> s.upper()
'BERLIN: 18.4 C AT 4 PM'
```

*Strings Are Constant.* A string cannot be changed, i.e., any change always results in a new string. Replacement of a character is not possible:

```
>>> s[18] = 5
...
TypeError: 'str' object does not support item assignment
```

If we want to replace `s[18]`, a new string must be constructed, for example by keeping the substrings on either side of `s[18]` and inserting a `'5'` in between:

```
>>> s[:18] + '5' + s[19:]
'Berlin: 18.4 C at 5 pm'
```

*Strings with Digits Only.* One can easily test whether a string contains digits only or not:

```
>>> '214'.isdigit()
True
>>> ' 214 '.isdigit()
False
>>> '2.14'.isdigit()
False
```

*Whitespace.* We can also check if a string contains spaces only by calling the `isspace` method. More precisely, `isspace` tests for *whitespace*, which means the space character, newline, or the TAB character:

```
>>> ' '.isspace() # blanks
True
>>> '\n'.isspace() # newline
True
>>> '\t'.isspace() # TAB
True
>>> ''.isspace() # empty string
False
```

The `isspace` is handy for testing for blank lines in files. An alternative is to strip first and then test for an empty string:

```
>>> line = '\n'
>>> empty.strip() == ''
True
```

Stripping off leading and/or trailing spaces in a string is sometimes useful:

```
>>> s = '   text with leading/trailing space   \n'
>>> s.strip()
'text with leading/trailing space'
>>> s.lstrip() # left strip
'text with leading/trailing space   \n'
>>> s.rstrip() # right strip
'   text with leading/trailing space'
```

*Joining Strings.* The opposite of the `split` method is `join`, which joins elements in a list of strings with a specified delimiter in between. That is, the following two types of statements are inverse operations:

```
t = delimiter.join(words)
words = t.split(delimiter)
```

An example on using `join` may be

```
>>> strings = ['Newton', 'Secant', 'Bisection']
>>> t = ', '.join(strings)
>>> t
'Newton, Secant, Bisection'
```

As an illustration of the usefulness of `split` and `join`, we want to remove the first two words on a line. This task can be done by first splitting the line into words and then joining the words of interest:

```
>>> line = 'This is a line of words separated by space'
>>> words = line.split()
>>> line2 = ' '.join(words[2:])
>>> line2
'a line of words separated by space'
```

There are many more methods in string objects. All methods are described in the Python Library Reference, see “string methods” in the index.

### 6.3.2 Example: Reading Pairs of Numbers

*Problem.* Suppose we have a file consisting of pairs of real numbers, i.e., text of the form  $(a, b)$ , where  $a$  and  $b$  are real numbers. This notation for a pair of numbers is often used for points in the plane, vectors in the plane, and complex numbers. A sample file may look as follows:

```
(1.3,0)    (-1,2)    (3,-1.5)
(0,1)      (1,0)    (1,1)
(0,-0.01)  (10.5,-1) (2.5,-2.5)
```

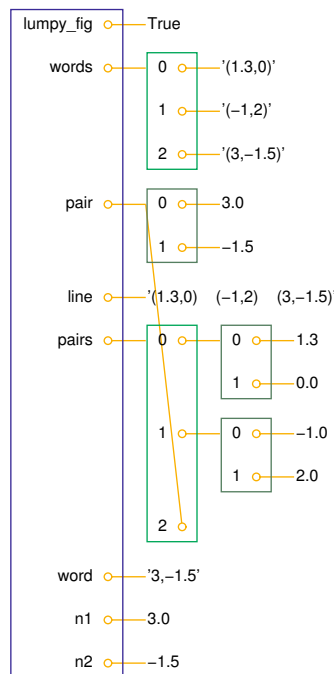
The file can be found as `read_pairs1.dat`. Our task is to read this text into a nested list `pairs` such that `pairs[i]` holds the pair with index  $i$ , and this pair is a tuple of two `float` objects. We assume that there are no blanks inside the parentheses of a pair of numbers (we rely on a `split` operation which would otherwise not work).

*Solution.* To solve this programming problem, we can read in the file line by line; for each line: split the line into words (i.e., split with respect to whitespace); for each word: strip off the parentheses, split with respect to comma, and convert the resulting two words to floats. Our brief algorithm can be almost directly translated to Python code:

```
lines = open('read_pairs1.dat', 'r').readlines()

pairs = [] # list of (n1, n2) pairs of numbers
for line in lines:
    words = line.split()
    for word in words:
        word = word[1:-1] # strip off parenthesis
        n1, n2 = word.split(',')
        n1 = float(n1); n2 = float(n2)
        pair = (n1, n2)
        pairs.append(pair) # add 2-tuple to last row
```

This code is available in the file `read_pairs1.py`. Figure 6.3 shows a snapshot of the state of the variables in the program after having treated the first line. You should explain each line in the program to yourself, and compare your understanding with the figure.



**Fig. 6.3** Illustration of the variables in the `read_pairs.py` program after the first pass in the loop over words in the first line of the data file.

The output from the program becomes

```
[(1.3, 0.0),
 (-1.0, 2.0),
 (3.0, -1.5),
 (0.0, 1.0),
```

```
(1.0, 0.0),
(1.0, 1.0),
(0.0, -0.01),
(10.5, -1.0),
(2.5, -2.5)]
```

We remark that our solution to this programming problem relies heavily on the fact that spaces inside the parentheses are not allowed. If spaces were allowed, the simple split to obtain the pairs on a line as words would not work. What can we then do?

We can first strip off all blanks on a line, and then observe that the pairs are separated by the text ')(' . The first and last pair on a line will have an extra parenthesis that we need to remove. The rest of code is similar to the previous code and can be found in `read_pairs2.py`:

```
infile = open('read_pairs2.dat', 'r')
lines = infile.readlines()

pairs = [] # list of (n1, n2) pairs of numbers
for line in lines:
    line = line.strip() # remove whitespace such as newline
    line = line.replace(' ', '') # remove all blanks
    words = line.split(')(')
    # strip off leading/trailing parenthesis in first/last word:
    words[0] = words[0][1:] # (-1,3 -> -1,3
    words[-1] = words[-1][:-1] # 8.5,9) -> 8.5,9
    for word in words:
        n1, n2 = word.split(',')
        n1 = float(n1); n2 = float(n2)
        pair = (n1, n2)
        pairs.append(pair)

infile.close()
```

The program can be tested on the file `read_pairs2.dat`:

```
(1.3 , 0)      (-1 , 2 )      (3, -1.5)
(0 , 1)      ( 1, 0)      ( 1 , 1 )
(0,-0.01)    (10.5,-1)    (2.5, -2.5)
```

A third approach is to notice that if the pairs were separated by commas,

```
(1, 3.0),      (-1, 2),      (3, -1.5)
(0, 1),      (1, 0),      (1, 1)
```

the file text is very close to the Python syntax of a list of 2-tuples. By adding enclosing brackets, plus a comma at the end of each line,

```
[(1, 3.0),      (-1, 2),      (3, -1.5),
(0, 1),      (1, 0),      (1, 1),]
```

we have a string to which we can apply `eval` to get the `pairs` list directly. Here is the code doing this (program `read_pairs3.py`):

```
infile = open('read_pairs3.dat', 'r')
listtext = '['
for line in infile:
    # add line, without newline (line[:-1]), with a trailing comma:
    listtext += line[:-1] + ', '
infile.close()
listtext = listtext + ']'
pairs = eval(listtext)
```

In general, it is a good idea to construct file formats that are as close as possible to valid Python syntax such that one can take advantage of the `eval` or `exec` functions to turn text into “live objects”.

### 6.3.3 Example: Reading Coordinates

*Problem.* Suppose we have a file with coordinates  $(x, y, z)$  in three-dimensional space. The file format looks as follows:

```
x=-1.345      y= 0.1112      z= 9.1928
x=-1.231      y=-0.1251     z= 1001.2
x= 0.100      y= 1.4344E+6   z=-1.0100
x= 0.200      y= 0.0012     z=-1.3423E+4
x= 1.5E+5     y=-0.7666     z= 1027
```

The goal is to read this file and create a list with  $(x, y, z)$  3-tuples, and thereafter convert the nested list to a two-dimensional array with which we can compute.

Note that there is sometimes a space between the `=` signs and the following number and sometimes not. Splitting with respect to space and extracting every second word is therefore not an option. We shall present three solutions.

*Solution 1: Substring Extraction.* The file format looks very regular with the `x=`, `y=`, and `z=` texts starting in the same columns at every line. By counting characters, we realize that the `x=` text starts in column 2, the `y=` text starts in column 16, while the `z=` text starts in column 31. Introducing

```
x_start = 2
y_start = 16
z_start = 31
```

the three numbers in a line string are obtained as the substrings

```
x = line[x_start+2:y_start]
y = line[y_start+2:z_start]
z = line[z_start+2:]
```

The following code, found in file `file2coor_v1.py`, creates the `coor` array with shape  $(n, 3)$ , where  $n$  is the number of  $(x, y, z)$  coordinates.

```
infile = open('xyz.dat', 'r')
coor = [] # list of (x,y,z) tuples
for line in infile:
    x_start = 2
    y_start = 16
    z_start = 31
    x = line[x_start+2:y_start]
    y = line[y_start+2:z_start]
    z = line[z_start+2:]
    print 'debug: x="%s", y="%s", z="%s"' % (x,y,z)
    coor.append((float(x), float(y), float(z)))
infile.close()
```

```
from numpy import *
coor = array(coor)
print coor.shape, coor
```

The `print` statement inside the loop is always wise to include when doing string manipulations, simply because counting indices for substring limits quickly leads to errors. Running the program, the output from the loop looks like this

```
debug: x="-1.345    ", y=" 0.1112    ", z=" 9.1928
"
```

for the first line in the file. The double quotes show the exact extent of the extracted coordinates. Note that the last quote appears on the next line. This is because `line` has a newline at the end (this newline must be there to define the end of the line), and the substring `line[z_start:]` contains the newline at the end of `line`. Writing `line[z_start:-1]` would leave the newline out of the `z` coordinate. However, this has no effect in practice since we transform the substrings to `float`, and an extra newline or other blanks make no harm.

The `coor` object at the end of the program has the value

```
[[-1.34500000e+00  1.11200000e-01  9.19280000e+00]
 [-1.23100000e+00 -1.25100000e-01  1.00120000e+03]
 [ 1.00000000e-01  1.43440000e+06 -1.01000000e+00]
 [ 2.00000000e-01  1.20000000e-03 -1.34230000e+04]
 [ 1.50000000e+05 -7.66600000e-01  1.02700000e+03]]
```

*Solution 2: String Search.* One problem with the solution approach above is that the program will not work if the file format is subject to a change in the column positions of `x=`, `y=`, or `z=`. Instead of hardcoding numbers for the column positions, we can use the `find` method in string objects to locate these column positions:

```
x_start = line.find('x=')
y_start = line.find('y=')
z_start = line.find('z=')
```

The rest of the code is similar to the complete program listed above, and the complete code is stored in the file `file2coor_v2.py`.

*Solution 3: String Split.* String splitting is a powerful tool, also in the present case. Let us split with respect to the equal sign. The first line in the file then gives us the words

```
['x', '-1.345', 'y', ' 0.1112', 'z', ' 9.1928']
```

We throw away the first word, and strip off the last character in the next word. The final word can be used as is. The complete program is found in the file `file2coor_v3.py` and looks like

```
infile = open('xyz.dat', 'r')
coor = [] # list of (x,y,z) tuples
for line in infile:
    words = line.split('=')
```

```
x = float(words[1][:-1])
y = float(words[2][:-1])
z = float(words[3])
coor.append((x, y, z))
infile.close()

from numpy import *
coor = array(coor)
print coor.shape, coor
```

More sophisticated examples of string operations appear in Chapter 6.4.4.

## 6.4 Reading Data from Web Pages

Python has a module `urllib` which makes it possible to read data from a web page as easily as we can read data from an ordinary file<sup>14</sup>. Before we do this, a few concepts from the Internet world must be touched.

### 6.4.1 About Web Pages

Web pages are viewed with a web browser. There are many competing browsers, although most people have only heard of Internet Explorer from Microsoft. Mac users may prefer Safari, while Firefox and Opera are browsers that run on different types of computers, including Windows, Linux, and Mac.

Any web page you visit is associated with an address, usually something like

```
http://www.some.where.net/some/file.html
```

This type of web address is called a URL (which stands for Uniform Resource Locator<sup>15</sup>). The graphics you see in a web browser, i.e., the web page you see with your eyes, is produced by a series of commands that specifies the text on the page, the images, buttons to be pressed, etc. Roughly speaking, these commands are like statements in computer programs. The commands are stored in a text file and follow rules in a language, exactly as you are used to when writing statements in a programming language.

A common language for defining web pages is HTML. A web page is then simply a text file with text containing HTML commands. Instead

<sup>14</sup> In principle this is true, but in practice the text in web pages tend to be much more complicated than the text in the files we have treated so far.

<sup>15</sup> Another term is URI (Uniform Resource Identifier), which is replacing URL in technical documentation. We stick to URL, however, in this book because Python's tools for accessing resources on the Internet have `url` as part of module and function names.



of a physical file, the web page can also be the output text from a program. In that case the URL is the name of the program file. The web browser interprets the text and the commands, and displays the information visually. Let us demonstrate this for a very simple web page shown in Figure 6.4. This page was produced by the following



Fig. 6.4 Example of what a very simple HTML file looks like in a web browser.

text with embedded HTML commands:

```
<html>
<body bgcolor="orange">
<h1>A Very Simple HTML Page</h1> <!-- headline -->
Web pages are written in a language called
<a href="http://www.w3.org/MarkUp/Guide/">HTML</a>.
Ordinary text is written as ordinary text, but when we
need links, headlines, lists,
<ul>
<li><em>emphasized words</em>, or
<li><b>boldface text</b>,
</ul>
we need to embed the text inside HTML tags. We can also
insert GIF or PNG images, taken from other Internet sites,
if desired.
<hr> <!-- horizontal line -->

</body>
</html>
```

A typical HTML command consists of an opening and a closing *tag*. For example, emphasized text is specified by enclosing the text inside *em* (emphasize) tags:

```
<em>emphasized words</em>
```

The opening tag is enclosed in less than and greater than signs, while the closing tag has an additional forward slash before the tag name.

In the HTML file we see an opening and closing `html` tag around the whole text in the file. Similarly, there is a pair of `body` tags, where the first one also has a parameter `bgcolor` which can be used to specify a background color in the web page. Section headlines are specified by enclosing the headline text inside `h1` tags. Subsection headlines apply `h2` tags, which results in a smaller font compared with `h1` tags. Comments appear inside `<!--` and `-->`. Links to other web pages are written inside

a tags, with an argument `href` for the link's web address. Lists apply the `ul` (unordered list) tag, while each item is written with just an opening tag `li` (list item), but no closing tag is necessary. Images are also specified with just an opening tag having name `img`, and the image file is given as a file name or URL of a file, enclosed in double quotes, as the `src` parameter.

The ultra-quick HTML course in the previous paragraphs gives a glimpse of how web pages can be constructed. One can either write the HTML file by hand in a pure text editor, or one can use programs such as Dream Weaver to help design the page graphically in a user-friendly environment, and then the program can automatically generate the right HTML syntax in files.

### 6.4.2 How to Access Web Pages in Programs

Why is it useful to know some HTML and how web pages are constructed? The reason is that the web is full of information that we can get access to through programs and use in new contexts. What we can get access to is not the visual web page you see, but the underlying HTML file. The information you see on the screen appear in text form in the HTML file, and by extracting text, we can get hold of the text's information in a program.

Given the URL as a string stored in a variable, there are two ways of accessing the HTML text in a Python program:

1. Download the HTML file and store it as a local file with a given name, say `webpage.html`:

```
import urllib
url = 'http://www.simula.no/research/scientific/cbc'
urllib.urlretrieve(url, filename='webpage.html')
```

2. Open the HTML file as a file-like object:

```
infile = urllib.urlopen(url)
```

This `f` has methods such as `read`, `readline`, and `readlines`.

### 6.4.3 Example: Reading Pure Text Files

HTML files are often like long and complicated programs. The information you are interested in might be buried in lots of HTML tags and ugly syntax. Extracting data from HTML files in Python programs is therefore not always an easy task. How to approach this problem is exemplified in Chapter 6.4.4. Nevertheless, the world of numerical computing can occasionally be simpler: data that we are interested in

computing with often appear in plain text files which can be accessed by URLs and viewed in web browsers. One example is temperature data from cities around the world. The temperatures in Oslo from Jan 1, 1995, until today are found in the URL

```
ftp://ftp.engr.udayton.edu/jkisssock/gsod/N00SLO.txt
```

The data reside in an ordinary text file, because the URL ends with a text file name N00SLO.txt. This text file can be downloaded via the URL:

```
import urllib
url = 'ftp://ftp.engr.udayton.edu/jkisssock/gsod/N00SLO.txt'
urllib.urlretrieve(url, filename='Oslo.txt')
```

By looking at the file Oslo.txt in a text editor, or simply by watching the web page in a web browser, we see that the file contains four columns, with the number of the month (1-12), the date, the year, and the temperature that day (in Fahrenheit). Here is an extract of the file:

1	1	1995	24.0
1	2	1995	22.4
1	3	1995	9.3
1	4	1995	6.1
1	5	1995	26.2
1	6	1995	24.8
1	7	1995	27.9
1	8	1995	33.0
1	9	1995	31.4
1	10	1995	29.4
1	11	1995	23.0

How can we load these file data into a Python program? First, we must decide on the data structure for storing the data. A nested dictionary could be handy for this purpose: `temp[year][month][date]`. That is, given the year, the number of the month, and the date as keys, the value of `temp` yields the corresponding temperature. The process of loading the file into such a dictionary is then a matter of reading the file line by line, and for each line, split the line into words, use the three first words as keys and the last as value.

A first attempt to load the file data into the `temps` dictionary could be like

```
infile = open('Oslo.txt', 'r')
temps = {}
for line in infile:
    month, date, year, temperature = line.split()
    temps[year][month][date] = temperature
```

However, running these lines gives a `KeyError: '1995'`. One can normally just assign new keys to a dictionary, but this is a nested dictionary, and each level must be initialized, not just the first level (which is initialized in the line before the `for` loop). Recall that for a period of 15 years, `temps` will consist of 15 dictionaries, and each of these contains 12 dictionaries for the months. This fact complicates the code:

We must test if a year or month key is present, and if not, an empty dictionary must be assigned. We must also transform the strings returned by `line.split()` into `int` and `float` objects (month, date, and year are integers while the temperature is a real number). The complete program is listed below and available in the file `webtemps.py`.

```
import urllib
url = 'ftp://ftp.engr.udayton.edu/jkissock/gsod/NOOSLO.txt'
urllib.urlretrieve(url, filename='Oslo.txt')

infile = open('Oslo.txt', 'r')
temps = {}
for line in infile:
    month, date, year, temperature = line.split()
    month = int(month)
    date = int(date)
    year = int(year)
    temperature = float(temperature)
    if not year in temps:
        temps[year] = {}
    if not month in temps[year]:
        temps[year][month] = {}
    temps[year][month][date] = temperature
infile.close()

# pick a day to verify that temps is correct:
year = 2003; month = 3; date = 31
T = temps[year][month][date]
print '%d.%d.%d: %.1f' % (year, month, date, T)
```

Running this code results in

```
2003.3.31: 38.5
```

We can view the `Oslo.txt` file and realize that the printed temperature is correct.

#### 6.4.4 Example: Extracting Data from an HTML Page

The weather forecast on Yahoo! contains a lot of graphics and advertisements along with weather data. Suppose you want to quickly see today's weather and temperature in a city. Can we make a program that finds this information in the web page? The answer is yes if we can download the page with `urllib.urlretrieve` (or open the web page as a file with `urllib.urlopen`) and if we know some string operations to help us search for some specific text.

The Yahoo! page for the weather in Oslo is (at the time of this writing) found at

```
http://weather.yahoo.com/forecast/NOXX0029\_c.html
```

All the text and images on this weather page are defined in the file associated with this address. First we download the file,

```
import urllib
w = 'http://weather.yahoo.com/forecast/NOXX0029_c.html'
urllib.urlretrieve(url=w, filename='weather.html')
```

Since the weather and temperature data are known to reside somewhere inside this file, we take a look at the downloaded file, named `weather.html`, in a pure text editor. What we see, is quite complicated text like

```
<!-- BEGIN TOP SEARCH -->
<div id="ynneck">
  <div id="searchbartop" class="searchbar">
    <form action="http://news.search.yahoo.com/news/search" method="get">
      <strong>Search:</strong>
      <input type="text" name="p" size="30">
      <!-- Include search dropdown box -->

      <select name=c>
        <option value="">All News & Blogs</option>
        <option value=yahoo_news>Yahoo! News Only</option>
        <option value=news_photos>News Photos</option>
        <option value=av>Video/Audio</option>
```

The file is written in the HTML format, using a lot of different tag names. Most of the text is uninteresting to us, but we see in the web browser that the data we are looking for appear under a text “Current conditions ...”. Searching for “Current conditions” in the `weather.html` file brings us down to the middle of the file. Here, some interesting text is embedded in some surrounding, uninteresting text:

```
<div class="forecast-module">
  <em>Current conditions as of 2:19 pm CET</em>
  <h3>Mostly Cloudy</h3>

  <dl>
    <dt>Feels Like:</dt>
    <dd>2&deg;</dd>
    <dt>Barometer:</dt>
    <dd>--</dd>
    <dt>Humidity:</dt>
    <dd>53%</dd>
    <dt>Visibility:</dt>
    <dd>
      9.99 km</dd>
    <dt>Dewpoint:</dt>
    <dd>-3&deg;</dd>
    <dt>Wind:</dt>
    <dd>N 24 kph</dd>
    <dt>Sunrise:</dt>
    <dd>6:19 am</dd>
    <dt>Sunset:</dt>
    <dd>
      6:29 pm</dd>
    </dd>
  </dl>

  <div id="forecast-temperature">
    <h3>6&deg;</h3>
    <p>High: 4&deg; Low: -2&deg;</p>
```

We are interested in two pieces of text:

1. After the line containing the text “Current conditions”, we have a line stating today’s weather:

```
<em>Current conditions as of 2:19 pm CET</em>
<h3>Mostly Cloudy</h3>
```

2. After the line containing “forecast-temperature”, we have a line with today’s temperature:

```
<div id="forecast-temperature">
<h3>6&deg;</h3>
```

The text `&deg;` is a special HTML command for the degrees symbol. We are not interested in this symbol, but the number prior to it.

We can extract the weather description and the temperature by running through all lines in the file, check if a line contains either “Current conditions” or “forecast-temperature”, and in both cases extract information from the next line. Since we need to go through all lines and look at the line after the current one, it is a good strategy to load all lines into a list and traverse the list with a `for` loop over list indices, because when we are at a line with index `i`, we can easily look at the next one with index `i+1`:

```
lines = infile.readlines()
for i in range(len(lines)):
    line = lines[i] # short form
    if 'Current conditions' in line:
        weather = lines[i+1][4:-6]
    if 'forecast-temperature' in line:
        temperature = float(lines[i+1][4:].split('&')[0])
        break # everything is found, jump out of loop
```

The lines computing `weather` and `temperature` probably need some comments. Looking at a line containing the text which `weather` should be set equal to,

```
<h3>Mostly Cloudy</h3>
```

we guess that in the general case we are interested in the text between `<h3>` and `</h3>`. The text in the middle can be extracted as a substring. Since we do not know the length of the weather description, we count from the right when creating the substring. The substring starts from index 4 and should go to, but not include, index -6 (you might think it would be index -5, but there is an invisible newline character at the end of the `line` string which we must also count). A small test in an interactive session can be used to control that we are doing the right thing:

```
>>> s = "<h3>Mostly Cloudy</h3>\n"
>>> s[4:-6]
'Mostly Cloudy'
```

The extraction of the temperature is more involved. The actual line looks like

```
<h3>6&deg;</h3>
```

We want to extract the number that comes after `<h3>` and before the ampersand sign. One method is to strip off the leading `<h3>` text by extracting the substring `lines[i+1][4:]`. Then we can split this substring with respect to the ampersand character. The first “word” from this split is the temperature, but the type is a string, so we need to convert it to `float` to be ready for calculations with the temperature number. All these steps can be done separately to better explain the individual tasks:

```
next_line = lines[i+1]
substring = next_line[4:]
words = substring.split('&')
temperature = words[0]
temperature = float(temperature)
```

The experienced programmer often prefers to condense the code and combine the five statements into one:

```
temperature = float(lines[i+1][4:].split('&')[0])
```

Note that when we have found the temperature, there is no need to examine more lines in the file. We therefore execute a `break` statement, which forces the program control to jump out of the loop.

Finally, we wrap the code extracting the weather and temperature conditions in a function:

```
def get_data(url):
    urllib.urlretrieve(url=url, filename='weather.html')
    infile = open('weather.html')
    lines = infile.readlines()
    ...
    infile.close()
    return weather, temperature
```

We might visit the Yahoo! weather pages and pick out a collection of cities and corresponding URLs, to be stored in a dictionary:

```
cities = {
    'Sandefjord':
        'http://weather.yahoo.com/forecast/NOXX0032_c.html',
    'Oslo':
        'http://weather.yahoo.com/forecast/NOXX0029_c.html',
    'Gothenburg':
        'http://weather.yahoo.com/forecast/SWXX0007_c.html',
    'Copenhagen':
        'http://weather.yahoo.com/forecast/DAXX0009_c.html',
}
```

A simple loop over this dictionary and a call to `get_data` gives a quick look at the weather conditions and temperatures in several cities:

```
for city in cities:
    weather, temperature = get_data(cities[city])
    print city, weather, temperature
```

The complete code is found in the file `weather.py`. A sample run on a winter day gave this result:

```
Oslo Fair 3.0
Copenhagen Partly Cloudy 5.0
Sandefjord Fair 4.0
Gothenburg Fair 0.0
```

The techniques described above are useful when you need to extract data from the web and process the data, either for presentation in a compact format as above or for further calculations. Turning web page information into compact text can also be useful for constructing SMS messages to be sent to mobile phones.

*Remark.* Interpretation of text in files is based on string operations in this book. A much more powerful and often more convenient approach is to apply so-called *regular expressions*. An introduction to regular expressions of relevance to scientific computations is given in the book [5]. We strongly recommend to learn about regular expressions if you end up interpreting a lot of HTML text in web pages. An even more powerful technique to extract information from web pages is to use an HTML parser, which comes with standard Python. This technique requires more programming compared to using string operations or regular expressions, but can handle cases that are impossible or difficult to address with the two other techniques. Googling for “HTML parsing Python” gives a lot of links to what HTML parsing is about and how it is done.

## 6.5 Writing Data to File

Writing data to file is easy. There is basically one function to pay attention to: `outfile.write(s)`, which writes a string `s` to a file handled by the file object `outfile`. Unlike `print`, `outfile.write(s)` does not append a newline character to the written string. It will therefore often be necessary to add a newline character,

```
outfile.write(s + '\n')
```

if the string `s` is meant to appear on a single line in the file and `s` does not already contain a trailing newline character. File writing is then a matter of constructing strings containing the text we want to have in the file and for each such string call `outfile.write`.

An alternative syntax to `outfile.write(s)` is `print ■ f, s`. This `print` statement adds a newline character, as usual.

Writing to a file demands the file object `f` to be opened for writing:



```
# write to new file, or overwrite file:
outfile = open(filename, 'w')

# append to the end of an existing file:
outfile = open(filename, 'a')
```

### 6.5.1 Example: Writing a Table to File

*Problem.* As a worked example of file writing, we shall write out a nested list with tabular data to file. A sample list may take look as

```
[ [ 0.75,      0.29619813, -0.29619813, -0.75      ],
  [ 0.29619813, 0.11697778, -0.11697778, -0.29619813],
  [-0.29619813, -0.11697778, 0.11697778, 0.29619813],
  [-0.75,      -0.29619813, 0.29619813, 0.75      ]]
```

*Solution.* We iterate through the rows (first index) in the list, and for each row, we iterate through the column values (second index) and write each value to the file. At the end of each row, we must insert a newline character in the file to get a linebreak. The code resides in the file `write1.py`:

```
data = [ [ 0.75,      0.29619813, -0.29619813, -0.75      ],
          [ 0.29619813, 0.11697778, -0.11697778, -0.29619813],
          [-0.29619813, -0.11697778, 0.11697778, 0.29619813],
          [-0.75,      -0.29619813, 0.29619813, 0.75      ] ]

outfile = open('tmp_table.dat', 'w')
for row in data:
    for column in row:
        outfile.write('%14.8f' % column)
    outfile.write('\n')
outfile.close()
```

The resulting data file becomes

```
0.75000000    0.29619813   -0.29619813   -0.75000000
0.29619813    0.11697778   -0.11697778   -0.29619813
-0.29619813   -0.11697778    0.11697778    0.29619813
-0.75000000   -0.29619813    0.29619813    0.75000000
```

An extension of this program consists in adding column and row headings:

	column 1	column 2	column 3	column 4
row 1	0.75000000	0.29619813	-0.29619813	-0.75000000
row 2	0.29619813	0.11697778	-0.11697778	-0.29619813
row 3	-0.29619813	-0.11697778	0.11697778	0.29619813
row 4	-0.75000000	-0.29619813	0.29619813	0.75000000

To obtain this end result, we need to add some statements to the program `write1.py`. For the column headings we need to know the number of columns, i.e., the length of the rows, and loop from 1 to this length:

```
ncolumns = len(data[0])
outfile.write('')
for i in range(1, ncolumns+1):
    outfile.write('%10s' % ('column %2d' % i))
outfile.write('\n')
```

Note the use of a nested `printf` construction: The text we want to insert is itself a `printf` string. We could also have written the text as `'column' + str(i)`, but then the length of the resulting string would depend on the number of digits in `i`. It is recommended to always use `printf` constructions for a tabular output format, because this gives automatic padding of blanks so that the width of the output strings remain the same. As always, the tuning of the widths is done in a trial-and-error process.

To add the row headings, we need a counter over the row numbers:

```
row_counter = 1
for row in data:
    outfile.write('row %2d' % row_counter)
    for column in row:
        outfile.write('%14.8f' % column)
    outfile.write('\n')
    row_counter += 1
```

The complete code is found in the file `write2.py`. We could, alternatively, iterate over the indices in the list:

```
for i in range(len(data)):
    outfile.write('row %2d' % (i+1))
    for j in range(len(data[i])):
        outfile.write('%14.8f' % data[i][j])
    outfile.write('\n')
```

### 6.5.2 Standard Input and Output as File Objects

Reading user input from the keyboard applies the function `raw_input` as explained in Chapter 3.1. The keyboard is a medium that the computer in fact treats as a file, referred to as *standard input*.

The `print` command prints text in the terminal window. This medium is also viewed as a file from the computer's point of view and called *standard output*. All general-purpose programming languages allow reading from standard input and writing to standard output. This reading and writing can be done with two types of tools, either file-like objects or special tools like `raw_input` (see Chapter 3.1.1) and `print` in Python. We will here describe the file-like objects: `sys.stdin` for standard input and `sys.stdout` for standard output. These objects behave as file objects, except that they do not need to be opened or closed. The statement

```
s = raw_input('Give s:')
```

is equivalent to

```
print 'Give s: ',  
s = sys.stdin.readline()
```

Recall that the trailing comma in the `print` statement avoids the new-line that `print` by default adds to the output string. Similarly,

```
s = eval(raw_input('Give s:'))
```

is equivalent to

```
print 'Give s: ',  
s = eval(sys.stdin.readline())
```

For output to the terminal window, the statement

```
print s
```

is equivalent to

```
sys.stdout.write(s + '\n')
```

Why it is handy to have access to standard input and output as file objects can be illustrated by an example. Suppose you have a function that reads data from a file object `infile` and writes data to a file object `outfile`. A sample function may take the form

```
def x2f(infile, outfile, f):  
    for line in infile:  
        x = float(line)  
        y = f(x)  
        outfile.write('%g\n' % y)
```

This function works with all types of files, including web pages as `infile` (see Chapter 6.4). With `sys.stdin` as `infile` and/or `sys.stdout` as `outfile`, the `x2f` function also works with standard input and/or standard output. Without `sys.stdin` and `sys.stdout`, we would need different code, employing `raw_input` and `print`, to deal with standard input and output. Now we can write a single function that deals with all file media in a unified way.

There is also something called *standard error*. Usually this is the terminal window, just as standard output, but programs can distinguish between writing ordinary output to standard output and error messages to standard error, and these output media can be redirected to, e.g., files such that one can separate error messages from ordinary output. In Python, standard error is the file-like object `sys.stderr`. A typical application of `sys.stderr` is to report errors:

```
if x < 0:  
    sys.stderr.write('Illegal value of x'); sys.exit(1)
```

This message to `sys.stderr` is an alternative to `print` or raising an exception.

*Redirecting Standard Input, Output, and Error.* Standard output from a program `prog` can be redirected to a file `out` using the greater than sign<sup>16</sup>:

---

```
Unix/DOS> prog > output
```

---

That is, the program writes to `output` instead of the terminal window. Standard error can be redirected by

---

```
Unix/DOS> prog &> output
```

---

When the program reads from standard input, we can equally well redirect standard input to a file, say with name `raw_input`, such that the program reads from this file rather than from the keyboard:

---

```
Unix/DOS> prog < input
```

---

Combinations are also possible:

---

```
Unix/DOS> prog < input > output
```

---

*Note.* The redirection of standard output, input, and error does not work for programs run inside IPython, only when run directly in the operating system in a terminal window.

Inside a Python program we can also let standard input, output, and error work with ordinary files instead. Here is the technique:

```
sys_stdout_orig = sys.stdout
sys.stdout = open('output', 'w')
sys_stdin_orig = sys.stdin
sys.stdin = open('input', 'r')
```

Now, any `print` statement will write to the `output` file, and any `raw_input` call will read from the `input` file. (Without storing the original `sys.stdout` and `sys.stdin` objects in new variables, these objects would get lost in the redefinition above and we would never be able to reach the common standard input and output in the program.)

### 6.5.3 Reading and Writing Spreadsheet Files

From school you are probably used to spreadsheet programs such as Microsoft Excel or OpenOffice. This type of program is used to represent a table of numbers and text. Each table entry is known as a

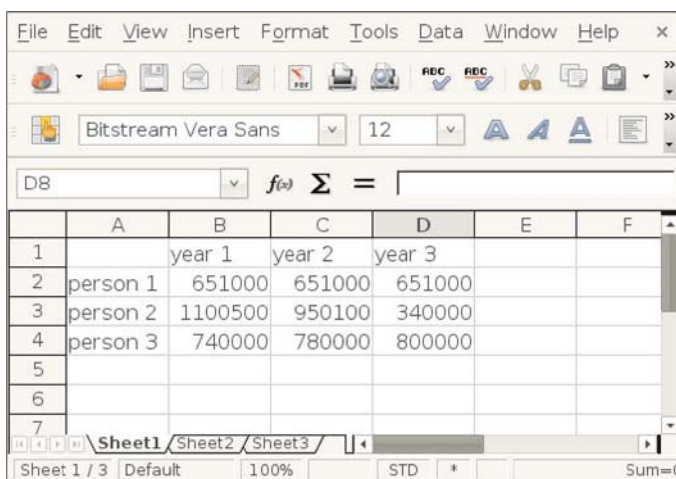
<sup>16</sup> `prog` can be any program, including a Python program run as, e.g., `python myprog.py`.

*cell*, and one can easily perform calculations with cells that contain numbers. The application of spreadsheet programs for mathematical computations and graphics is steadily growing.

Also Python may be used to do spreadsheet-type calculations on tabular data. The advantage of using Python is that you can easily extend the calculations far beyond what a spreadsheet program can do. However, even if you can view Python as a substitute for a spreadsheet program, it may be beneficial to combine the two. Suppose you have some data in a spreadsheet. How can you read these data into a Python program, perform calculations on the data, and thereafter read the data back to the spreadsheet program? This is exactly what we will explain below through an example. With this example, you should understand how easy it is to combine Excel or OpenOffice with your own Python programs.

The table of data in a spreadsheet can be saved in so-called CSV files, where CSV stands for *comma separated values*. The CSV file format is very simple: each row in the spreadsheet table is a line in the file, and each cell in the row is separated by a comma or some other specified separation character. CSV files can easily be read into Python programs, and the table of cell data can be stored in a nested list (table, cf. Chapter 2.1.7), which can be processed as we desire. The modified table of cell data can be written back to a CSV file and read into the spreadsheet program for further processing.

Figure 6.5 shows a simple spreadsheet in the OpenOffice program. The table contains  $4 \times 4$  cells, where the first row contains column headings and the first column contains row headings. The remaining  $3 \times 3$  subtable contains numbers that we may compute with. Let us



The screenshot shows the OpenOffice spreadsheet interface. The menu bar includes File, Edit, View, Insert, Format, Tools, Data, Window, and Help. The toolbar contains various icons for file operations and editing. The font settings are set to Bitstream Vera Sans, size 12. The active cell is D8, and the formula bar shows a sum function. The spreadsheet data is as follows:

	A	B	C	D	E	F
1		year 1	year 2	year 3		
2	person 1	651000	651000	651000		
3	person 2	1100500	950100	340000		
4	person 3	740000	780000	800000		
5						
6						
7						

The status bar at the bottom indicates 'Sheet 1 / 3', 'Default', '100%', 'STD', and 'Sum=0'.

Fig. 6.5 A simple spreadsheet in OpenOffice.

save this spreadsheet to a file in the CSV format. The complete file will typically look as follows:

```
, "year 1", "year 2", "year 3"
"person 1", 651000, 651000, 651000
"person 2", 1100500, 950100, 340000
"person 3", 740000, 780000, 800000
```

*Reading CSV Files.* Our goal is to write a Python code for loading the spreadsheet data into a table. The table is technically a nested list, where each list element is a row of the table, and each row is a list of the table's column values. CSV files can be read, row by row, using the `csv` module from Python's standard library. The recipe goes like this, if the data reside in the CSV file `budget.csv`:

```
infile = open('budget.csv', 'r')
import csv
table = []
for row in csv.reader(infile):
    table.append(row)
infile.close()
```

The `row` variable is a list of column values that are read from the file by the `csv` module. The three lines computing `table` can be condensed to one using a list comprehension:

```
table = [row for row in csv.reader(infile)]
```

We can easily print `table`,

```
import pprint
pprint.pprint(table)
```

to see what the spreadsheet looks like when it is represented as a nested list in a Python program:

```
[['', 'year 1', 'year 2', 'year 3'],
 ['person 1', '651000', '651000', '651000'],
 ['person 2', '1100500', '950100', '340000'],
 ['person 3', '740000', '780000', '800000']]
```

Observe now that all entries are surrounded by quotes, which means that all entries are string (`str`) objects. This is a general rule: the `csv` module reads all cells into string objects. To compute with the numbers, we need to transform the string objects to `float` objects. The transformation should not be applied to the first row and first column, since the cells here hold text. The transformation from strings to numbers therefore applies to the indices `r` and `c` in `table` (`table[r][c]`), such that the row counter `r` goes from 1 to `len(table)-1`, and the column counter `c` goes from 1 to `len(table[0])-1` (`len(table[0])` is the length of the first row, assuming the lengths of all rows are equal to the length of the first row). The relevant Python code for this transformation task becomes

```
for r in range(1,len(table)):
    for c in range(1, len(table[0])):
        table[r][c] = float(table[r][c])
```

A `pprint.pprint(table)` statement after this transformation yields

```
[['', 'year 1', 'year 2', 'year 3'],
 ['person 1', 651000.0, 651000.0, 651000.0],
 ['person 2', 1100500.0, 950100.0, 340000.0],
 ['person 3', 740000.0, 780000.0, 800000.0]]
```

The numbers now have a decimal and no quotes, indicating that the numbers are float objects and hence ready for mathematical calculations.

*Processing Data.* Let us perform a very simple calculation with `table`, namely adding a final row with the sum of the numbers in the columns:

```
row = [0.0]*len(table[0])
row[0] = 'sum'
for c in range(1, len(row)):
    s = 0
    for r in range(1, len(table)):
        s += table[r][c]
    row[c] = s
```

As seen, we first create a list `row` consisting of zeros. Then we insert a text in the first column, before we invoke a loop over the numbers in the table and compute the sum of each column. The `table` list now represents a spreadsheet with four columns and five rows:

```
[['', 'year 1', 'year 2', 'year 3'],
 ['person 1', 651000.0, 651000.0, 651000.0],
 ['person 2', 1100500.0, 950100.0, 340000.0],
 ['person 3', 740000.0, 780000.0, 800000.0],
 ['sum', 2491500.0, 2381100.0, 1791000.0]]
```

*Writing CSV Files.* Our final task is to write the modified `table` list back to a CSV file so that the data can be loaded in a spreadsheet program. The write task is done by the code segment

```
outfile = open('budget2.csv', 'w')
writer = csv.writer(outfile)
for row in table:
    writer.writerow(row)
outfile.close()
```

The `budget2.csv` looks like this:

```
,year 1,year 2,year 3
person 1,651000.0,651000.0,651000.0
person 2,1100500.0,950100.0,340000.0
person 3,740000.0,780000.0,800000.0
sum,2491500.0,2381100.0,1791000.0
```

The final step is to read `budget2.csv` into a spreadsheet. The result is displayed in Figure 6.6 (in OpenOffice one must specify in the “open” dialog that the spreadsheet data are separated by commas, i.e., that the file is in CSV format).

	A	B	C	D	E	F
1		year 1	year 2	year 3		
2	person 1	651000	651000	651000		
3	person 2	1100500	950100	340000		
4	person 3	740000	780000	800000		
5	sum	2491500	2381100	1791000		
6						
7						

**Fig. 6.6** A spreadsheet processed in a Python program and loaded back into OpenOffice.

The complete program reading the `budget.csv` file, processing its data, and writing the `budget2.csv` file can be found in `rw_csv.py`. With this example at hand, you should be in a good position to combine spreadsheet programs with your own Python programs.

*Remark.* You may wonder why we used the `csv` module to read and write CSV files when such files have comma separated values which we can extract by splitting lines with respect to the comma (in Chapter 6.2.6 we used this technique to read a CSV file):

```
infile = open('budget.csv', 'r')
for line in infile:
    row = line.split(',')

```

This works well for the present `budget.csv` file, but the technique breaks down when a text in a cell contains a comma, for instance "Aug 8, 2007". The `line.split(',')` will split this cell text, while the `csv.reader` functionality is smart enough to avoid splitting text cells with a comma.

*Representing Number Cells with Numerical Python Arrays.* Instead of putting the whole spreadsheet into a single nested list, we can make a Python data structure more tailored to the data at hand. What we have are two headers (for rows and columns, respectively) and a subtable of numbers. The headers can be represented as lists of strings, while the subtable could be a two-dimensional Numerical Python array. The latter makes it easier to implement various mathematical operations on the numbers. A dictionary can hold all the three items: two header lists and one array. The relevant code for reading, processing, and writing the data is shown below and can be found in the file `rw_csv_numpy.py`:



```

infile = open('budget.csv', 'r')
import csv
table = [row for row in csv.reader(infile)]
infile.close()

# convert subtable of numbers (string to float):
subtable = [[float(c) for c in row[1:]] for row in table[1:]]

data = {'column headings': table[0][1:],
        'row headings': [row[0] for row in table[1:]],
        'array': array(subtable)}

# add a new row with sums:
data['row headings'].append('sum')
a = data['array'] # short form
data['column sum'] = [sum(a[:,c]) for c in range(a.shape[1])]

outfile = open('budget2.csv', 'w')
writer = csv.writer(outfile)
# turn data dictionary into a nested list first (for easy writing):
table = a.tolist() # transform array to nested list
table.append(data['column sum'])
table.insert(0, data['column headings'])
# extend table with row headings (a new column):
table = [table[r].insert(0, data['row headings'][r]) \
          for r in range(len(table))]
for row in table:
    writer.writerow(row)
outfile.close()

```

The code makes heavy use of list comprehensions, and the transformation between a nested list, for file reading and writing, and the data dictionary, for representing the data in the Python program, is non-trivial. If you manage to understand every line in this program, you have digested a lot of topics in Python programming!

## 6.6 Summary

### 6.6.1 Chapter Topics

*File Operations.* This chapter has been concerned with file reading and file writing. First a file must be opened, either for reading, writing, or appending:

```

infile = open(filename, 'r') # read
outfile = open(filename, 'w') # write
outfile = open(filename, 'a') # append

```

There are four basic reading commands:

```

line    = infile.readline() # read the next line
filestr = infile.read()     # read rest of file into string
lines   = infile.readlines() # read rest of file into list
for line in infile:         # read rest of file line by line

```

File writing is usually about repeatedly using the command

```
outfile.write(s)
```

where `s` is a string. Contrary to `print s`, no newline is added to `s` in `outfile.write(s)`.

When the reading and writing is finished,

```
somefile.close()
```

should be called, where `somefile` is the file object.

*Downloading Internet Files.* Internet files can be downloaded if we know their URL:

```
import urllib
url = 'http://www.some.where.net/path/thing.html'
urllib.urlretrieve(url, filename='thing.html')
```

The downloaded information is put in the local file `thing.html` in the current working folder. Alternatively, we can open the URL as a file object:

```
webpage = urllib.urlopen(url)
```

HTML files are often messy to interpret by string operations.

**Table 6.1** Summary of important functionality for dictionary objects.

<code>a = {}</code>	initialize an empty dictionary
<code>a = {'point': [0,0.1], 'value': 7}</code>	initialize a dictionary
<code>a = dict(point=[2,7], value=3)</code>	initialize a dictionary w/string keys
<code>a['hide'] = True</code>	add new key-value pair to a dictionary
<code>a['point']</code>	get value corresponding to key <code>point</code>
<code>'value' in a</code>	True if <code>value</code> is a key in <code>a</code>
<code>del a['point']</code>	delete a key-value pair from <code>a</code>
<code>a.keys()</code>	list of keys
<code>a.values()</code>	list of values
<code>len(a)</code>	number of key-value pairs in <code>a</code>
<code>for key in a:</code>	loop over keys in unknown order
<code>for key in sorted(a):</code>	loop over keys in alphabetic order
<code>isinstance(a, dict)</code>	is True if <code>a</code> is a dictionary

*Dictionaries.* Array or list-like objects with text or other (fixed-valued) Python objects as indices are called dictionaries. They are very useful for storing general collections of objects in a single data structure. Table 6.1 displays some of the most important dictionary operations.

*Strings.* Some of the most useful functionalities in a string object `s` are listed below.

- Split the string into substrings separated by `delimiter`:

```
words = s.split(delimiter)
```

- Join elements in a list of strings:

```
string = delimiter.join(words[i:j])
```

- Extract substring:

```
substring = s[2:n-4]
```

- Substitute a substring by new a string:

```
modified_string = s.replace(sub, new)
```

- Search for the start (first index) of some text:

```
index = s.find(text)
if index == -1:
    print 'Could not find "%s" in "%s" (text, s)
else:
    substring = s[index:] # strip off chars before text
```

- Check if a string contains whitespace only:

```
if s.isspace():
    ...
```

### 6.6.2 Summarizing Example: A File Database

*Problem.* We have a file containing information about the courses that students have taken. The file format consists of blocks with student data, where each block starts with the student's name (Name:), followed by the courses that the student has taken. Each course line starts with the name of the course, then comes the semester when the exam was taken, then the size of the course in terms of credit points, and finally the grade is listed (letters A to F). Here is an example of a file with three student entries:

Name: John Doe	
Astronomy	2003 fall 10 A
Introductory Physics	2003 fall 10 C
Calculus I	2003 fall 10 A
Calculus II	2004 spring 10 B
Linear Algebra	2004 spring 10 C
Quantum Mechanics I	2004 fall 10 A
Quantum Mechanics II	2005 spring 10 A
Numerical Linear Algebra	2004 fall 5 E
Numerical Methods	2004 spring 20 C
Name: Jan Modaal	
Calculus I	2005 fall 10 A
Calculus II	2006 spring 10 A
Introductory C++ Programming	2005 fall 15 D
Introductory Python Programming	2006 spring 5 A
Astronomy	2005 fall 10 A

Basic Philosophy	2005 fall 10 F
Name: Kari Nordmann	
Introductory Python Programming	2006 spring 5 A
Astronomy	2005 fall 10 D

Our problem consists of reading this file into a dictionary `data` with the student name as key and a list of courses as value. Each element in the list of courses is a dictionary holding the course name, the semester, the credit points, and the grade. A value in the `data` dictionary may look as

```
'Kari Nordmann': [{'credit': 5,
                    'grade': 'A',
                    'semester': '2006 spring',
                    'title': 'Introductory Python Programming'},
                  {'credit': 10,
                    'grade': 'D',
                    'semester': '2005 fall',
                    'title': 'Astronomy'}],
```

Having the `data` dictionary, the next task is to print out the average grade of each student.

*Solution.* We divide the problem into two major tasks: loading the file data into the `data` dictionary, and computing the average grades. These two tasks are naturally placed in two functions.

We need to have a strategy for reading the file and interpreting the contents. It will be natural to read the file line by line, and for each line check if this is a line containing a new student's name, a course information line, or a blank line. In the latter case we jump to the next pass in the loop. When a new student name is encountered, we initialize a new entry in the `data` dictionary to an empty list. In the case of a line about a course, we must interpret the contents on that line, which we postpone a bit.

We can now sketch the algorithm described above in terms of some unfinished Python code, just to get the overview:

```
def load(studentfile):
    infile = open(studentfile, 'r')
    data = {}
    for line in infile:
        i = line.find('Name:')
        if i != -1:
            # line contains 'Name:', extract the name
            ...
        elif line.isspace():      # blank line?
            continue             # go to next loop iteration
        else:
            # this must be a course line
            # interpret the line
            ...
    infile.close()
    return data
```

If we find 'Name:' as a substring in `line`, we must extract the name. This can be done by the substring `line[i+5:]`. Alternatively, we can split the line with respect to colon and strip off the first word:

```
words = line.split(':')
name = ' '.join(words[1:])
```

We have chosen the former strategy of extracting the name as a substring in the final program.

Each course line is naturally split into words for extracting information:

```
words = line.split()
```

The name of the course consists of a number of words, but we do not know how many. Nevertheless, we know that the final words contain the semester, the credit points, and the grade. We can hence count from the right and extract information, and when we are finished with the semester information, the rest of the `words` list holds the words in the name of the course. The code goes as follows:

```
grade = words[-1]
credit = int(words[-2])
semester = ' '.join(words[-4:-2])
course_name = ' '.join(words[:-4])
data[name].append({'title': course_name,
                  'semester': semester,
                  'credit': credit,
                  'grade': grade})
```

This code is a good example of the usefulness of split and join operations when extracting information from a text.

Now to the second task of computing the average grade. Since the grades are letters we cannot compute with them. A natural way to proceed is to convert the letters to numbers, compute the average number, and then convert that number back to a letter. Conversion between letters and numbers is easily represented by a dictionary:

```
grade2number = {'A': 5, 'B': 4, 'C': 3, 'D': 2, 'E': 1, 'F': 0}
```

To convert from numbers to grades, we construct the “inverse” dictionary:

```
number2grade = {}
for grade in grade2number:
    number2grade[grade2number[grade]] = grade
```

In the computation of the average grade we should use a weighted sum such that larger courses count more than smaller courses. The weighted mean value of a set of numbers  $r_i$  with weights  $w_i$ ,  $i = 0, \dots, n - 1$ , is given by

$$\frac{\sum_{i=0}^{n-1} w_i r_i}{\sum_{i=0}^{n-1} w_i}.$$

This weighted mean value must then be rounded to the nearest integer, which can be used as key in `number2grade` to find the corresponding grade expressed as a letter. The weight  $w_i$  is naturally taken as the number of credit points in the course with grade  $r_i$ . The whole process is performed by the following function:

```
def average_grade(data, name):
    sum = 0; weights = 0
    for course in data[name]:
        weight = course['credit']
        grade = course['grade']
        sum += grade2number[grade]*weight
        weights += weight
    avg = sum/float(weights)
    return number2grade[round(avg)]
```

The complete code is found in the file `students.py`. Running this program gives the following output of the average grades:

```
John Doe: B
Kari Nordmann: C
Jan Modaal: C
```

One feature of the `students.py` code is that the output of the names are sorted after the last name. How can we accomplish that? A straight `for name in data` loop will visit the keys in an unknown (random) order. To visit the keys in alphabetic order, we must use

```
for name in sorted(data):
```

This default sort will sort with respect to the first character in the name strings. We want a sort according to the last part of the name. A tailored sort function can then be written (see Exercise 2.44 for an introduction to tailored sort functions). In this function we extract the last word in the names and compare them:

```
def sort_names(name1, name2):
    last_name1 = name1.split()[-1]
    last_name2 = name2.split()[-1]
    if last_name1 < last_name2:
        return -1
    elif last_name1 > last_name2:
        return 1
    else:
        return 0
```

We can now pass on `sort_names` to the `sorted` function to get a sequence that is sorted with respect to the last word in the students' names:

```
for name in sorted(data, sort_names):
    print '%s: %s' % (name, average_grade(data, name))
```

## 6.7 Exercises

### Exercise 6.1. *Read a two-column data file.*

The file `src/files/xy.dat` contains two columns of numbers, corresponding to  $x$  and  $y$  coordinates on a curve. The start of the file looks as this:

```
-1.0000    -0.0000
-0.9933    -0.0087
-0.9867    -0.0179
-0.9800    -0.0274
-0.9733    -0.0374
```

Make a program that reads the first column into a list `x` and the second column into a list `y`. Then convert the lists to arrays, and plot the curve. Print out the maximum and minimum  $y$  coordinates. (Hint: Read the file line by line, split each line into words, convert to `float`, and append to `x` and `y`.) Name of program file: `read_2columns.py` ◇

### Exercise 6.2. *Read a data file.*

The files `density_of_water.dat` and `density_of_air.dat` files in the folder `src/files` contain data about the density of water and air (resp.) for different temperatures. The data files have some comment lines starting with `#` and some lines are blank. The rest of the lines contain density data: the temperature in the first column and the corresponding density in the second column. The goal of this exercise is to read the density data and plot them. Let the program take the name of the data file as command-line argument, load the density data into NumPy arrays, and plot the data using circles for the data points. Demonstrate that the program can read both files. Name of program file: `read_density_data.py` ◇

### Exercise 6.3. *Simplify the implementation of Exer. 6.1.*

Files with data in a tabular fashion are very common and so is the operation of the reading the data into arrays. Therefore, the `scitools.filetable` module offers easy-to-use functions for loading data files with columns of numbers into NumPy arrays. First read about `scitools.filetable` using `pydoc` in a terminal window (cf. page 98). Then solve Exercise 6.1 using appropriate functions from the `scitools.filetable` module. Name of program file: `read_2columns_filetable.py`. ◇

### Exercise 6.4. *Fit a polynomial to data.*

The purpose of this exercise is to find a simple mathematical formula for the how the density of water or air depends on the temperature. First, load the density data from file as explained in Exercises 6.2 or 6.3. Then we want to experiment with NumPy utilities that can find a polynomial that approximate the density curve.

NumPy has a function `polyfit(x, y, deg)` for finding a “best fit” of a polynomial of degree `deg` to a set of data points given by the array

arguments  $x$  and  $y$ . The `polyfit` function returns a list of the coefficients in the fitted polynomial, where the first element is the coefficient for the term with the highest degree, and the last element corresponds to the constant term. For example, given points in  $x$  and  $y$ , `polyfit(x, y, 1)` returns the coefficients  $a$ ,  $b$  in a polynomial  $a*x + b$  that fits the data in the best way<sup>17</sup>.

NumPy also has a utility `poly1d` which can take the tuple or list of coefficients calculated by, e.g., `polyfit` and return the polynomial as a Python function that can be evaluated. The following code snippet demonstrates the use of `polyfit` and `poly1d`:

```
coeff = polyfit(x, y, deg)
p = poly1d(coeff)
print p # prints the polynomial expression
y_fitted = p(x)
plot(x, y, 'r-', x, y_fitted, 'b-',
      legend=('data', 'fitted polynomial of degree %d' % deg))
```

For the density–temperature relationship we want to plot the data from file and two polynomial approximations, corresponding to a 1st and 2nd degree polynomial. From a visual inspection of the plot, suggest simple mathematical formulas that relate the density of air to temperature and the density of water to temperature. Make three separate plots of the Name of program file: `fit_density_data.py` ◇

**Exercise 6.5.** *Read acceleration data and find velocities.*

A file `src/files/acc.dat` contains measurements  $a_0, a_1, \dots, a_{n-1}$  of the acceleration of an object moving along a straight line. The measurement  $a_k$  is taken at time point  $t_k = k\Delta t$ , where  $\Delta t$  is the time spacing between the measurements. The purpose of the exercise is to load the acceleration data into a program and compute the velocity  $v(t)$  of the object at some time  $t$ .

In general, the acceleration  $a(t)$  is related to the velocity  $v(t)$  through  $v'(t) = a(t)$ . This means that

$$v(t) = v(0) + \int_0^t a(\tau) d\tau. \quad (6.1)$$

If  $a(t)$  is only known at some discrete, equally spaced points in time,  $a_0, \dots, a_{n-1}$  (which is the case in this exercise), we must compute the integral (6.1) in numerically, for example by the Trapezoidal rule:

$$v(t_k) \approx \Delta t \left( \frac{1}{2}a_0 + \frac{1}{2}a_k + \sum_{i=1}^{k-1} a_i \right), \quad 1 \leq k \leq n-1. \quad (6.2)$$

<sup>17</sup> More precisely, a line  $y = ax + b$  is a “best fit” to the data points  $(x_i, y_i)$ ,  $i = 0, \dots, n-1$  if  $a$  and  $b$  are chosen to make the sum of squared errors  $R = \sum_{j=0}^{n-1} (y_j - (ax_j + b))^2$  as small as possible. This approach is known as *least squares approximation* to data and proves to be extremely useful throughout science and technology.



We assume  $v(0) = 0$  so that also  $v_0 = 0$ .

Read the values  $a_0, \dots, a_{n-1}$  from file into an array, plot the acceleration versus time, and use (6.2) to compute one  $v(t_k)$  value, where  $\Delta t$  and  $k \geq 1$  are specified on the command line. Name of program file: `acc2vel_v1.py`.  $\diamond$

**Exercise 6.6.** *Read acceleration data and plot velocities.*

The task in this exercise is the same as in Exercise 6.5, except that we now want to compute  $v(t_k)$  for all time points  $t_k = k\Delta t$  and plot the velocity versus time. Repeated use of (6.2) for all  $k$  values is very inefficient. A more efficient formula arises if we add the area of a new trapezoid to the previous integral:

$$v(t_k) = v(t_{k-1}) + \int_{t_{k-1}}^{t_k} a(\tau) d\tau \approx v(t_{k-1}) + \Delta t \frac{1}{2} (a_{k-1} + a_k), \quad (6.3)$$

for  $k = 1, 2, \dots, n-1$ , while  $v_0 = 0$ . Use this formula to fill an array `v` with velocity values. Now only  $\Delta t$  is given on the command line, and the  $a_0, \dots, a_{n-1}$  values must be read from file as in Exercise 6.5. Name of program file: `acc2vel.py`.  $\diamond$

**Exercise 6.7.** *Find velocity from GPS coordinates.*

Imagine that a GPS device measures your position at every  $s$  seconds. The positions are stored as  $(x, y)$  coordinates in a file `src/files/pos.dat` with the an  $x$  and  $y$  number on each line, except for the first line which contains the value of  $s$ .

First, load  $s$  into a `float` variable and the  $x$  and  $y$  numbers into two arrays and draw a straight line between the points (i.e., plot the  $y$  coordinates versus the  $x$  coordinates).

The next task is to compute and plot the velocity of the movements. If  $x(t)$  and  $y(t)$  are the coordinates of the positions as a function of time, we have that the velocity in  $x$  direction is  $v_x(t) = dx/dt$ , and the velocity in  $y$  direction is  $v_y = dy/dt$ . Since  $x$  and  $y$  are only known for some discrete times,  $t_k = ks$ ,  $k = 0, \dots, n-1$ , we must use numerical differentiation. A simple (forward) formula is

$$v_x(t_k) \approx \frac{x(t_{k+1}) - x(t_k)}{s}, \quad v_y(t_k) \approx \frac{y(t_{k+1}) - y(t_k)}{s}, \quad k = 0, \dots, n-2.$$

Compute arrays `vx` and `vy` with velocities based on the formulas above for  $v_x(t_k)$  and  $v_y(t_k)$ ,  $k = 0, \dots, n-2$ . Plot `vx` versus time and `vy` versus time. Name of program file: `position2velocity.py`.  $\diamond$

**Exercise 6.8.** *Make a dictionary from a table.*

The file `src/files/constants.txt` contains a table of the values and the dimensions of some fundamental constants from physics. We want to load this table into a dictionary `constants`, where the keys are

the names of the constants. For example, `constants['gravitational constant']` holds the value of the gravitational constant ( $6.67259 \cdot 10^{-11}$ ) in Newton's law of gravitation. You may either initialize the dictionary by a program that reads and interprets the text in the file, or you may manually cut and paste text in the file into a program where you define the dictionary. Name of program file: `fundamental_constants.py`. ◇

**Exercise 6.9.** *Explore syntax differences: lists vs. dictionaries.*

Consider this code:

```
t1 = {}
t1[0] = -5
t1[1] = 10.5
```

Explain why the lines above work fine while the ones below do not:

```
t2 = []
t2[0] = -5
t2[1] = 10.5
```

What must be done in the last code snippet to make it work properly? Name of program file: `list_vs_dict.py`. ◇

**Exercise 6.10.** *Improve the program from Ch. 6.2.4.*

Consider the program `density.py` from Chapter 6.2.4. One problem we face when implementing this program is that the name of the substance can contain one or two words, and maybe more words in a more comprehensive table. The purpose of this exercise is to use string operations to shorten the code and make it more general. Implement the following two methods in separate functions in the same program, and control that they give the same result.

1. Let `substance` consist of all the words but the last, using the `join` method in string objects to combine the words.
2. Observe that all the densities start in the same column file and use substrings to divide `line` into two parts. (Hint: Remember to strip the first part such that, e.g., the density of ice is obtained as `densities['ice']` and not `densities['ice ']`.)

Name of program file: `density_improved.py`. ◇

**Exercise 6.11.** *Interpret output from a program.*

The program `src/basic/lnsum.py` produces, among other things, this output:

```
epsilon: 1e-04, exact error: 8.18e-04, n=55
epsilon: 1e-06, exact error: 9.02e-06, n=97
epsilon: 1e-08, exact error: 8.70e-08, n=142
epsilon: 1e-10, exact error: 9.20e-10, n=187
epsilon: 1e-12, exact error: 9.31e-12, n=233
```

Redirect the output to a file. Write a Python program that reads the file and extracts the numbers corresponding to `epsilon`, `exact error`,

and `n`. Store the numbers in three arrays and plot `epsilon` and the exact error versus `n`. Use a logarithmic scale on the  $y$  axis, which is enabled by the `log='y'` keyword argument to the `plot` function. Name of program file: `read_error.py`. ◇

**Exercise 6.12.** *Make a dictionary.*

Based on the stars data in Exercise 2.44, make a dictionary where the keys contain the names of the stars and the values correspond to the luminosity. Name of program file: `stars_data_dict1.py`. ◇

**Exercise 6.13.** *Make a nested dictionary.*

Store the data about stars from Exercise 2.44 in a nested dictionary such that we can look up the distance, the apparent brightness, and the luminosity of a star with name `N` by `stars[N]['distance']`, `stars[N]['apparent brightness']`, and `stars[N]['luminosity']`. Name of program file: `stars_data_dict2.py`. ◇

**Exercise 6.14.** *Make a nested dictionary from a file.*

The file `src/files/human_evolution.txt` holds information about various human species and their height, weight, and brain volume. Make a program that reads this file and stores the tabular data in a nested dictionary `humans`. The keys in `humans` correspond to the specie name (e.g., “homo erectus”), and the values are dictionaries with keys for “height”, “weight”, “brain volume”, and “when” (the latter for when the specie lived). For example, `humans['homo neanderthalensis']['mass']` should equal '55-70'. Let the program write out the `humans` dictionary in a nice tabular form similar to that in the file. Name of program file: `humans.py`. ◇

**Exercise 6.15.** *Compute the area of a triangle.*

The purpose of this exercise is to write an `area` function as in Exercise 2.17, but now we assume that the vertices of the triangle is stored in a dictionary and not a list. The keys in the dictionary correspond to the vertex number (1, 2, or 3) while the values are 2-tuples with the  $x$  and  $y$  coordinates of the vertex. For example, in a triangle with vertices (0, 0), (1, 0), and (0, 2) the `vertices` argument becomes

```
{1: (0,0), 2: (1,0), 3: (0,2)}
```

Name of program file: `area_triangle_dict.py`. ◇

**Exercise 6.16.** *Compare data structures for polynomials.*

Write a code snippet that uses both a list and a dictionary to represent the polynomial  $-\frac{1}{2} + 2x^{100}$ . Print the list and the dictionary, and use them to evaluate the polynomial for  $x = 1.05$  (you can apply the `poly1` and `poly2` functions from Chapter 6.2.3). Name of program file: `poly_repr.py`. ◇

**Exercise 6.17.** *Compute the derivative of a polynomial.*

A polynomial can be represented by a dictionary as explained in Chapter 6.2.3. Write a function `diff` for differentiating such a polynomial. The `diff` function takes the polynomial as a dictionary argument and returns the dictionary representation of the derivative. Recall the formula for differentiation of polynomials:

$$\frac{d}{dx} \sum_{j=0}^n c_j x^j = \sum_{j=1}^n j c_j x^{j-1}. \quad (6.4)$$

This means that the coefficient of the  $x^{j-1}$  term in the derivative equals  $j$  times the coefficient of  $x^j$  term of the original polynomial. With `p` as the polynomial dictionary and `dp` as the dictionary representing the derivative, we then have `dp[j-1] = k*p[j]` for  $j$  running over all keys in `p`, except when  $j$  equals 0.

Here is an example of the use of the function `diff`:

```
>>> p = {0: -3, 3: 2, 5: -1}      # -3 + 2*x**3 - x**5
>>> diff(p)                       # should be 6*x**2 - 5*x**4
{2: 6, 4: -5}
```

Name of program file: `poly_diff.py`. ◇

**Exercise 6.18.** *Generalize the program from Ch. 6.2.6.*

The program from Chapter 6.2.6 is specialized for three particular companies. Suppose you download  $n$  files from *finance.yahoo.com*, all with monthly stock price data for the *same* period of time. Also suppose you name these files `company.csv`, where `company` reflects the name of the company. Modify the program from Chapter 6.2.6 such that it reads a set of filenames from the command line and creates a plot that compares the evolution of the corresponding stock prices. Normalize all prices such that they initially start at a unit value. Name of program file: `stockprices3.py`. ◇

**Exercise 6.19.** *Write function data to file.*

We want to dump  $x$  and  $f(x)$  values to a file, where the  $x$  values appear in the first column and the  $f(x)$  values appear in the second. Choose  $n$  equally spaced  $x$  values in the interval  $[a, b]$ . Provide  $f$ ,  $a$ ,  $b$ ,  $n$ , and the filename as input data on the command line. Use the `StringFunction` tool (see Chapters 3.1.4 and 4.4.3) to turn the textual expression for  $f$  into a Python function. (Note that the program from Exercise 6.1 can be used to read the file generated in the present exercise into arrays again for visualization of the curve  $y = f(x)$ .) Name of program files `write_cml_function.py`. ◇

**Exercise 6.20.** *Specify functions on the command line.*

Explain what the following two code snippets do and give an example of how they can be used. Snippet 1:

```
import sys
from scitools.StringFunction import StringFunction
parameters = {}
for prm in sys.argv[4:]:
    key, value = prm.split('=')
    parameters[key] = eval(value)
f = StringFunction(sys.argv[1], independent_variables=sys.argv[2],
                  **parameters)
var = float(sys.argv[3])
print f(var)
```

Snippet 2:

```
import sys
from scitools.StringFunction import StringFunction
f = eval('StringFunction(sys.argv[1], ' + \
        'independent_variables=sys.argv[2], %s)' % \
        ('', '.join(sys.argv[4:]))')
var = float(sys.argv[3])
print f(var)
```

Hint: Read about the `StringFunction` tool in Chapter 3.1.4 and about a variable number of keyword arguments in Appendix E.5. Name of program file: `cml_functions.py`. ◇

**Exercise 6.21.** *Interpret function specifications.*

To specify arbitrary functions  $f(x_1, x_2, \dots; p_1, p_2, \dots)$  with independent variables  $x_1, x_2, \dots$  and a set of parameters  $p_1, p_2, \dots$ , we allow the following syntax on the command line or in a file:

`<expression>` is function of `<list1>` with parameter `<list2>`

where `<expression>` denotes the function formula, `<list1>` is a comma-separated list of the independent variables, and `<list2>` is a comma-separated list of name=value parameters. The part with parameters `<list2>` is omitted if there are no parameters. The names of the independent variables and the parameters can be chosen freely as long as the names can be used as Python variables. Here are some examples of this syntax can be used to specify:

```
sin(x) is a function of x
sin(a*y) is a function of y with parameter a=2
sin(a*x-phi) is a function of x with parameter a=3, phi=-pi
exp(-a*x)*cos(w*t) is a function of t with parameter a=1,w=pi,x=2
```

Create a Python function that takes such function specifications as input and returns an appropriate `StringFunction` object. This object must be created from the function expression and the list of independent variables and parameters. For example, the last function specification above leads to the following `StringFunction` creation:

```
f = StringFunction('exp(-a*x)*sin(k*x-w*t)',
                  independent_variables=['t'],
                  a=1, w=pi, x=2)
```

Hint: Use string operations to extract the various parts of the string. For example, the expression can be split out by calling `split('is a`

function'). Typically, you need to extract <expression>, <list1>, and <list2>, and create a string like

```
StringFunction(<expression>, independent_variables=[<list1>],
               <list2>)
```

and sending it to `eval` to create the object. Name of program file: `text2func.py`. ◇

**Exercise 6.22.** *Compare average temperatures in two cities.*

Chapter 6.4 exemplifies how we can extract temperature data from the web. Similar data for the city of Stockholm is available at

```
ftp://ftp.engr.udayton.edu/jkisssock/gsod/SNSTKHLM.txt
```

If we inspect the `*.txt` files containing temperature data from Oslo and Stockholm, we observe that even though most of the temperatures seem reasonable, the value `-99` keeps appearing as a temperature. This value indicates a missing observation, and the value must not enter the computations of the average temperatures.

Make a Python program that computes the average temperature in Celsius degrees since 1995 in Oslo and Stockholm. Hint: You do not need to store the data in a dictionary as in Chapter 6.4 – adding up the numbers in the 4th column is sufficient. Name of program file: `compare_mean_temp.py`. ◇

**Exercise 6.23.** *Compare average temperatures in many cities.*

You should do Exercise 6.22 first. The URL

```
http://www.engr.udayton.edu/weather/citylistWorld.htm
```

contains links to temperature data for many cities around the world. The task of this exercise is to make a list of the cities and their average temperatures since 1995 and until the present date. Your program will download the files containing the temperature data from all the cities listed on the web page. This may be a rather long process, so make sure you have quite some time available. Fortunately, you only need to download all the files once. Use the test

```
if os.path.isfile(filename):
```

to check if a particular file with name `filename` is already present in the current folder, so you can avoid a new download.

First, you need to interpret the HTML text in the `citylistWorld.htm` file whose complete URL is given above. Download the file, inspect it, and realize that there are two types of lines of interest, one with the city name, typically on the form,

```
mso-list:l6 level1 lfo3;tab-stops:list .5in''<b>Algiers ( </b><b>...
```

and one with the URL of the temperature data,

```
href="ftp://ftp.engr.udayton.edu/jkisssock/gsod/ALALGIER.txt">ALA...
```

The first one can be detected by a test `line.find(" .5in'>")`, if `line` is a string containing a line from the file. Extracting the city name can be performed by, for example, a `line.split(">")` and picking the right component, and stripping off leading and trailing characters. The URL for the temperature data appears some lines below the city name. The line can be found by using `line.find("href")`. One can extract the URL by splitting with respect to `'>'` and stripping off the initial `href=` text.

Store the city names and the associated URLs in a dictionary. Thereafter, go through all the cities and compute their average temperature values. You can change the values of the dictionary to store both the temperature value and the URL as a 2-tuple.

Finally, write out the cities and their average temperatures in sorted sequence, starting with the hottest city and ending with the coolest. To sort the dictionary, first transform it to a list of 3-tuples (cityname, URL, temperature), and then write a tailored sort function for this type of list elements (see Exercise 2.44 for details about a similar tailored sort function). Make sure that the values `-99` for missing data do not enter any computations.

Organize the program as a module (Chapter 3.5), i.e., a set of functions for carrying out the main steps and a test block where the functions are called. Some of the functionality in this module can be reused in Exercises 6.24 and 6.25. Name of program file: `sort_mean_temp_cities.py`. ◇

**Exercise 6.24.** *Plot the temperature in a city, 1995-today.*

The purpose of this exercise is to read the name of a city from the command line, and thereafter present a plot of the temperature in that city from 1995 to today. You must first carry out Exercise 6.23 so that you have a module with a function that returns a dictionary with city names as keys and the corresponding URL for the temperature data files as values. The URL must be opened, and the temperature data must be read into an array. We plot this array against its indices, not against year, month, and day (that will be too complicated). Note that the temperature data may contain values `-99`, indicating missing recordings, and these values will lead to wrong, sudden jumps in the plots. If you insert the value `NaN` (a NumPy type for representing “Not a Number”) instead of the numerical values `-99` in arrays, some plotting programs will plot the array correctly, i.e., as several curve segments where the missing data are left out. This is true if you use Easyviz with Gnuplot as plotting program.

Construct the program as a module, where there are two functions that can be imported in other programs:

```
def get_city_URLs():
    """Return dictionary d[cityname] = URL."""

def get_temperatures(URL):
    """Return array with temperature values read from URL."""
```

Name of program file: `plot_temp.py`. ◇

**Exercise 6.25.** *Plot temperatures in several cities.*

This exercise is a continuation of Exercise 6.24. Make a program that starts with printing out the names of all cities for which we have temperature data from 1995 to today. Then ask the user for the names of some cities (separated by blanks). Thereafter load the temperature data for these cities (use the `get_city_URLs` and `get_temperatures` functions from Exercise 6.24) and visualize them in the same plot. Name of program file: `plot_multiple_temps.py`. ◇

**Exercise 6.26.** *Try Word or OpenOffice to write a program.*

The purpose of this exercise is to tell you how hard it may be to write Python programs in the standard programs that most people use for writing text.

Type the following one-line program in either Microsoft Word or OpenOffice:

```
print "Hello, World!"
```

Both Word and OpenOffice are so “smart” that they automatically edit “print” to “Print” since a sentence should always start with a capital. This is just an example that word processors are made for writing documents, not computer programs.

Save the program as a `.doc` (Word) or `.odt` (OpenOffice) file. Now try to run this file as a Python program. You will get a message

```
SyntaxError: Non-ASCII character
```

Explain why you get this error.

Then save the program as a `.txt` file. Run this file as a Python program. It may work well if you wrote the program text in Microsoft Word, but with OpenOffice there may still be strange characters in the file. Use a text editor to view the exact contents of the file. Name of program file: `office.py`. ◇

**Exercise 6.27.** *Evaluate objects in a boolean context.*

Writing `if a:` or `while a:` in a program, where `a` is some object, requires evaluation of `a` in a boolean context. To see the value of an object `a` in a boolean context, one can call `bool(a)`. Try the following program to learn what values of what objects that are `True` or `False` in a boolean context:



```

objects = [
    '',          # empty string
    'string',    # non-empty string
    [],          # empty list
    [0],         # list with one element
    (),          # empty tuple
    (0,),        # tuple with one element
    {},          # empty dict
    {0:0},       # dict with one element
    0,           # int zero
    0.0,         # float zero
    0j,          # complex zero
    10,          # int 10
    10.,         # float 10
    10j,         # imaginary 10
    zeros(0),    # empty array
    zeros(1),    # array with one element (zero)
    zeros(1)+10, # array with one element (10)
    zeros(2),    # array with two elements (watch out!)
]
for element in objects:
    object = eval(element)
    print 'object = %s; if object: is %s' % \
        (element, bool(object))

```

Write down a rule for the family of Python objects that evaluate to False in a boolean context. ◇

**Exercise 6.28.** *Generate an HTML report.*

Extend the program made in Exercise 5.22 with a report containing all the plots. The report can be written in HTML and displayed by a web browser. The plots must then be generated in PNG format. The source of the HTML file will typically look as follows:

```

<html>
<body>
<p>
<p>
<p>
<p>
...
<p>
</html>
</body>

```

Let the program write out the HTML text. You can let the function making the plots return the name of the plotfile, such that this string can be inserted in the HTML file. Name of program file: `growth_logistic4.py`. ◇

**Exercise 6.29.** *Fit a polynomial to experimental data.*

Suppose we have measured the oscillation period  $T$  of a simple pendulum with a mass  $m$  at the end of a massless rod of length  $L$ . We have varied  $L$  and recorded the corresponding  $T$  value. The measurements are found in a file `src/files/pendulum.dat`, containing two columns. The first column contains  $L$  values and the second column has the corresponding  $T$  values.

Load the  $L$  and  $T$  values into two arrays. Plot  $L$  versus  $T$  using circles for the data points. We shall assume that  $L$  as a function of

$T$  is a polynomial. Use the NumPy utilities `polyfit` and `poly1d`, as explained in Exercise 6.4, and experiment with fitting polynomials of degree 1, 2, and 3. Visualize the polynomial curves together with the experimental data. Which polynomial fits the measured data best? Name of program file: `fit_pendulum_data.py`. ◇

**Exercise 6.30.** *Interpret an HTML file with rainfall data.*

The file `src/files/rainfall.url` contains the URL to several web pages with average rainfall data from major cities in the world. The goal of this exercise is to download all these web pages, and for each web page load the rainfall data into an array where the first 12 elements corresponds to the rainfall in each month of the year, and the 13th element contains the total rainfall in a year.

Make a function `getdata(url)` which downloads a web page with address `url` and returns the name of the weather station (usually a city) and an array with 13 elements containing the average rainfall data found in the web page. Make another function `plotdata(data, location)` which plots the array returned from `getdata` with the location of the weather station as plot title. Let `plotdata` make a hard-copy with `location` as a part of the filename (some `location` names in the web pages contain a slash or other characters that are not appropriate in filenames – remove these). Call `getdata` and `plotdata` for each for each of the URLs in the `rainfall.url` file.

Hint: The rainfall data in the web pages appear in an HTML table. The relevant line in the file starts like

```
<tr><td> mm <td align=right>193.0 <td align=right>143.6
```

Assuming that the line is available as the string `line` in the program, a test if `line.startswith('<tr><td> mm')` makes you pick out the right line. You can then strip off the `mm` column by `line[12:]`. Then you can replace `<td align=right>` by an empty string. The line ends with `<br>`, which must be removed. The result is a line with numbers, the monthly and annual rainfall, separated by blanks. A split operation and conversion to `float` creates a list of the data.

Regarding, the location of the weather station, this is found in a line which starts out as

```
<p>Weather station <strong>OSLO/BLINDERN</strong>
```

You can, for example, use `line.find` method to locate the tags `<strong>` and `/strong>` and thereby extract the location of the weather station.

Name of program file: `download_rainfall_data.py`. ◇

**Exercise 6.31.** *Generate an HTML report with figures.*

The goal of this exercise is to let a program write a report in HTML format. The report starts with the Python code for the  $f(x, t)$  function from Exercise 4.17 on page 229. Program code can be placed inside `<pre>` and `</pre>` tags. The report should continue with three plots of

the function in Exercise 4.11 for three different  $t$  values (find suitable  $t$  values that illustrate the displacement of the wave packet). At the end, there is an animated GIF file with the movie from Exercise 4.17. Insert appropriate headlines (`<h1>` tags) in the report. Name of program file: `wavepacket_report.py`.  $\diamond$