Loops and lists

2

This chapter explains how repetitive tasks in a program can be automated by loops. We also introduce list objects for storing and processing collections of data with a specific order. Loops and lists, together with functions and `if` tests from Chapter 3, lay the fundamental programming foundation for the rest of the book. The programs associated with the chapter are found in the folder `src/looplist`[1].

2.1 While loops

Our task now is to print out a conversion table with Celsius degrees in the first column of the table and the corresponding Fahrenheit degrees in the second column. Such a table may look like this:

```
-20  -4.0
-15   5.0
-10  14.0
 -5  23.0
  0  32.0
  5  41.0
 10  50.0
 15  59.0
 20  68.0
 25  77.0
 30  86.0
 35  95.0
 40 104.0
```

2.1.1 A naive solution

The formula for converting C degrees Celsius to F degrees Fahrenheit is $F = 9C/5 + 32$. Since we know how to evaluate the formula for one value of C, we can just repeat these statements as many times as required

[1] `http://tinyurl.com/pwyasaa/looplist`

H.P. Langtangen, *A Primer on Scientific Programming with Python*,
Texts in Computational Science and Engineering 6,
DOI 10.1007/978-3-642-54959-5_2, © Springer-Verlag Berlin Heidelberg 2014

for the table above. Using three statements per line in the program, for compact layout of the code, we can write the whole program as

```
C = -20;  F = 9.0/5*C + 32;  print C, F
C = -15;  F = 9.0/5*C + 32;  print C, F
C = -10;  F = 9.0/5*C + 32;  print C, F
C =  -5;  F = 9.0/5*C + 32;  print C, F
C =   0;  F = 9.0/5*C + 32;  print C, F
C =   5;  F = 9.0/5*C + 32;  print C, F
C =  10;  F = 9.0/5*C + 32;  print C, F
C =  15;  F = 9.0/5*C + 32;  print C, F
C =  20;  F = 9.0/5*C + 32;  print C, F
C =  25;  F = 9.0/5*C + 32;  print C, F
C =  30;  F = 9.0/5*C + 32;  print C, F
C =  35;  F = 9.0/5*C + 32;  print C, F
C =  40;  F = 9.0/5*C + 32;  print C, F
```

Running this program (which is stored in the file `c2f_table_repeat.py`), demonstrates that the output becomes

```
-20 -4.0
-15 5.0
-10 14.0
-5 23.0
0 32.0
5 41.0
10 50.0
15 59.0
20 68.0
25 77.0
30 86.0
35 95.0
40 104.0
```

This output suffers from somewhat ugly formatting, but that problem can quickly be fixed by replacing `print C, F` by a `print` statement based on printf formatting. We will return to this detail later.

The main problem with the program above is that lots of statements are identical and repeated. First of all it is boring to write this sort of repeated statements, especially if we want many more C and F values in the table. Second, the idea of the computer is to automate repetition. Therefore, all computer languages have constructs to efficiently express repetition. These constructs are called *loops* and come in two variants in Python: `while` loops and `for` loops. Most programs in this book employ loops, so this concept is extremely important to learn.

2.1.2 While loops

The `while` loop is used to repeat a set of statements as long as a condition is true. We shall introduce this kind of loop through an example. The task is to generate the rows of the table of C and F values. The C value starts at -20 and is incremented by 5 as long as $C \leq 40$. For each C value we compute the corresponding F value and write out the two temperatures. In addition, we also add a line of dashes above and below the table.

The list of tasks to be done can be summarized as follows:

- Print line with dashes
- $C = -20$
- While $C \leq 40$:
  - $F = \frac{9}{5}C + 32$
  - Print C and F
  - Increment C by 5
- Print line with dashes

This is the *algorithm* of our programming task. The way from a detailed algorithm to a fully functioning Python code can often be made very short, which is definitely true in the present case:

```
print '------------------'     # table heading
C = -20                       # start value for C
dC = 5                        # increment of C in loop
while C <= 40:                # loop heading with condition
    F = (9.0/5)*C + 32        # 1st statement inside loop
    print C, F                # 2nd statement inside loop
    C = C + dC                # 3rd statement inside loop
print '------------------'     # end of table line (after loop)
```

A very important feature of Python is now encountered: the block of statements to be executed in each pass of the `while` loop must be indented. In the example above the block consists of three lines, and all these lines must have exactly the same indentation. Our choice of indentation in this book is four spaces. The first statement whose indentation coincides with that of the `while` line marks the end of the loop and is executed after the loop has terminated. In this example this is the final `print` statement. You are encouraged to type in the code above in a file, indent the last line four spaces, and observe what happens (you will experience that lines in the table are separated by a line of dashes: `----`).

Many novice Python programmers forget the colon at the end of the `while` line - this colon is essential and marks the beginning of the indented block of statements inside the loop. Later, we will see that there are many other similar program constructions in Python where there is a heading ending with a colon, followed by an indented block of statements.

Programmers need to fully understand what is going on in a program and be able to simulate the program by hand. Let us do this with the program segment above. First, we define the start value for the sequence of Celsius temperatures: `C = -20`. We also define the increment `dC` that will be added to `C` inside the loop. Then we enter the loop condition `C <= 40`. The first time `C` is `-20`, which implies that `C <= 40` (equivalent to $C \leq 40$ in mathematical notation) is true. Since the loop condition is true, we enter the loop and execute all the indented statements. That is, we compute `F` corresponding to the current `C` value, print the temperatures, and increment `C` by `dC`. For simplicity, we have used a plain `print C, F`

without any formatting so the columns will not be aligned, but this can easily be fixed later.

Thereafter, we enter the second pass in the loop. First we check the condition: `C` is `-15` and `C <= 40` is still true. We execute the statements in the indented loop block, `C` becomes `-10`, this is still less than or equal to `40`, so we enter the loop block again. This procedure is repeated until `C` is updated from `40` to `45` in the final statement in the loop block. When we then test the condition, `C <= 40`, this condition is no longer true, and the loop is terminated. We proceed with the next statement that has the same indentation as the `while` statement, which is the final `print` statement in this example.

Newcomers to programming are sometimes confused by statements like

```
C = C + dC
```

This line looks erroneous from a mathematical viewpoint, but the statement is perfectly valid computer code, because we first evaluate the expression on the right-hand side of the equality sign and then let the variable on the left-hand side refer to the result of this evaluation. In our case, `C` and `dC` are two different `int` objects. The operation `C+dC` results in a new `int` object, which in the assignment `C = C+dC` is bound to the name `C`. Before this assignment, `C` was already bound to an `int` object, and this object is automatically destroyed when `C` is bound to a new object and there are no other names (variables) referring to this previous object (if you did not get this last point, just relax and continue reading!).

Since incrementing the value of a variable is frequently done in computer programs, there is a special short-hand notation for this and related operations:

```
C += dC   # equivalent to C = C + dC
C -= dC   # equivalent to C = C - dC
C *= dC   # equivalent to C = C*dC
C /= dC   # equivalent to C = C/dC
```

2.1.3 Boolean expressions

In our first example on a `while` loop, we worked with a condition `C <= 40`, which evaluates to either true or false, written as `True` or `False` in Python. Other comparisons are also useful:

```
C == 40    # C equals 40
C != 40    # C does not equal 40
C >= 40    # C is greater than or equal to 40
C >  40    # C is greater than 40
C <  40    # C is less than 40
```

Not only comparisons between numbers can be used as conditions in `while` loops: any expression that has a boolean (`True` or `False`) value can be used. Such expressions are known as *logical* or *boolean* expressions.

The keyword `not` can be inserted in front of the boolean expression to change the value from `True` to `False` or from `False` to `True`. To evaluate `not C == 40`, we first evaluate `C == 40`, for `C = 1` this is `False`, and then `not` turns the value into `True`. On the opposite, if `C == 40` is `True`, `not C == 40` becomes `False`. Mathematically it is easier to read `C != 40` than `not C == 40`, but these two boolean expressions are equivalent.

Boolean expressions can be combined with `and` and `or` to form new compound boolean expressions, as in

```
while x > 0 and y <= 1:
    print x, y
```

If `cond1` and `cond2` are two boolean expressions with values `True` or `False`, the compound boolean expression `cond1 and cond2` is `True` if both `cond1` and `cond2` are `True`. On the other hand, `cond1 or cond2` is `True` if at least one of the conditions, `cond1` or `cond2`, is `True`

Remark

In Python, `cond1 and cond2` or `cond1 or cond2` returns one of the operands and not just `True` or `False` values as in most other computer languages. The operands `cond1` or `cond2` can be expressions or objects. In case of expressions, these are first evaluated to an object before the compound boolean expression is evaluated. For example, `(5+1) or -1` evaluates to `6` (the second operand is not evaluated when the first one is `True`), and `(5+1) and -1` evaluates to `-1`.

Here are some more examples from an interactive session where we just evaluate the boolean expressions themselves without using them in loop conditions:

```
>>> x = 0;  y = 1.2
>>> x >= 0 and y < 1
False
>>> x >= 0 or y < 1
True
>>> x > 0 or y > 1
True
>>> x > 0 or not y > 1
False
>>> -1 < x <= 0   #  -1 < x and x <= 0
True
>>> not (x > 0 or y > 0)
False
```

In the last sample expression, `not` applies to the value of the boolean expression inside the parentheses: `x>0` is `False`, `y>0` is `True`, so the combined expression with `or` is `True`, and `not` turns this value to `False`.

The common boolean values in Python are `True`, `False`, `0` (false), and any integer different from zero (true). To see such values in action, we recommend doing Exercises 2.21 and 2.17.

Boolean evaluation of an object

All objects in Python can in fact be evaluated in a boolean context, and all are `True` except `False`, zero numbers, and empty strings, lists, and dictionaries:

```
>>> s = 'some string'
>>> bool(s)
True
>>> s = ''  # empty string
>>> bool(s)
False
>>> L = [1, 4, 6]
>>> bool(L)
True
>>> L = []
>>> bool(L)
False
>>> a = 88.0
>>> bool(a)
True
>>> a = 0.0
>>> bool(a)
False
```

Essentially, `if a` tests if `a` is a non-empty object or if it is non-zero value. Such constructions are frequent in Python code.

Erroneous thinking about boolean expressions is one of the most common sources of errors in computer programs, so you should be careful every time you encounter a boolean expression and check that it is correctly stated.

2.1.4 Loop implementation of a sum

Summations frequently appear in mathematics. For instance, the sine function can be calculated as a polynomial:

$$\sin(x) \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots, \tag{2.1}$$

where $3! = 3 \cdot 2 \cdot 1$, $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$, etc., are factorial expressions. Computing $k! = k(k-1)(k-2)\cdots 2 \cdot 1$ is done by `math.factorial(k)`.

An infinite number of terms are needed on the right-hand side of (2.1) for the equality sign to hold. With a finite number of terms, we obtain

an approximation to $\sin(x)$, which is well suited for being calculated in a program since only powers and the basic four arithmetic operations are involved. Say we want to compute the right-hand side of (2.1) for powers up to $N = 25$. Writing out and implementing each one of these terms is a tedious job that can easily be automated by a loop.

Computation of the sum in (2.1) by a `while` loop in Python, makes use of (i) a counter `k` that runs through odd numbers from 1 up to some given maximum power `N`, and (ii) a summation variable, say `s`, which accumulates the terms, one at a time. The purpose of each pass of the loop is to compute a new term and add it to `s`. Since the sign of each term alternates, we introduce a variable `sign` that changes between -1 and 1 in each pass of the loop.

The previous paragraph can be precisely expressed by this piece of Python code:

```
x = 1.2  # assign some value
N = 25   # maximum power in sum
k = 1
s = x
sign = 1.0
import math

while k < N:
    sign = - sign
    k = k + 2
    term = sign*x**k/math.factorial(k)
    s = s + term

print 'sin(%g) = %g (approximation with %d terms)' % (x, s, N)
```

The best way to understand such a program is to simulate it by hand. That is, we go through the statements, one by one, and write down on a piece of paper what the state of each variable is.

When the loop is first entered, `k < N` implies `1 < 25`, which is `True` so we enter the loop block. There, we compute `sign = -1.0`, `k = 3`, `term = -1.0*x**3/(3*2*1))` (note that `sign` is float so we always have `float` divided by `int`), and `s = x - x**3/6`, which equals the first two terms in the sum. Then we test the loop condition: `3 < 25` is `True` so we enter the loop block again. This time we obtain `term = 1.0*x**5/math.factorial(5)`, which correctly implements the third term in the sum. At some point, `k` is updated to from 23 to 25 inside the loop and the loop condition then becomes `25 < 25`, which is `False`, implying that the program jumps over the loop block and continues with the `print` statement (which has the same indentation as the `while` statement).

2.2 Lists

Up to now a variable has typically contained a single number. Sometimes numbers are naturally grouped together. For example, all Celsius degrees in the first column of our table from Section 2.1.2 could be conveniently stored together as a group. A Python *list* can be used to represent such a group of numbers in a program. With a variable that refers to the list, we can work with the whole group at once, but we can also access individual elements of the group. Figure 2.1 illustrates the difference between an `int` object and a list object. In general, a list may contain a sequence of arbitrary objects in a given order. Python has great functionality for examining and manipulating such sequences of objects, which will be demonstrated below.

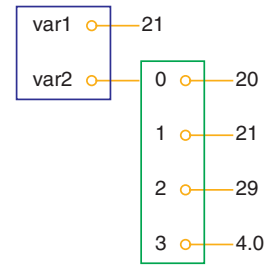

Fig. 2.1 Illustration of two variables: `var1` refers to an `int` object with value 21, created by the statement `var1 = 21`, and `var2` refers to a `list` object with value `[20, 21, 29, 4.0]`, i.e., three `int` objects and one `float` object, created by the statement `var2 = [20, 21, 29, 4.0]`.

2.2.1 Basic list operations

To create a list with the numbers from the first column in our table, we just put all the numbers inside square brackets and separate the numbers by commas:

```
C = [-20, -15, -10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40]
```

The variable `C` now refers to a `list` object holding 13 list *elements*. All list elements are in this case `int` objects.

Every element in a list is associated with an *index*, which reflects the position of the element in the list. The first element has index 0, the second index 1, and so on. Associated with the `C` list above we have 13 indices, starting with 0 and ending with 12. To access the element with index 3, i.e., the fourth element in the list, we can write `C[3]`. As we see from the list, `C[3]` refers to an `int` object with the value -5.

Elements in lists can be deleted, and new elements can be inserted anywhere. The functionality for doing this is built into the list object

and accessed by a dot notation. Two examples are `C.append(v)`, which appends a new element `v` to the end of the list, and `C.insert(i,v)`, which inserts a new element `v` in position number `i` in the list. The number of elements in a list is given by `len(C)`. Let us exemplify some list operations in an interactive session to see the effect of the operations:

```
>>> C = [-10, -5, 0, 5, 10, 15, 20, 25, 30]     # create list
>>> C.append(35)              # add new element 35 at the end
>>> C                         # view list C
[-10, -5, 0, 5, 10, 15, 20, 25, 30, 35]
```

Two lists can be added:

```
>>> C = C + [40, 45]          # extend C at the end
>>> C
[-10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40, 45]
```

What adding two lists means is up to the list object to define, and not surprisingly, addition of two lists is defined as appending the second list to the first. The result of `C + [40,45]` is a new list object, which we then assign to `C` such that this name refers to this new list. In fact, every object in Python and everything you can do with it is defined by programs made by humans. With the techniques of class programming (see Chapter 7) you can create your own objects and define (if desired) what it means to add such objects. All this gives enormous power in the hands of programmers. As one example, you can define your own list object if you are not satisfied with the functionality of Python's own lists.

New elements can be inserted anywhere in the list (and not only at the end as we did with `C.append`):

```
>>> C.insert(0, -15)          # insert new element -15 as index 0
>>> C
[-15, -10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40, 45]
```

With `del C[i]` we can remove an element with index `i` from the list `C`. Observe that this changes the list, so `C[i]` refers to another (the next) element after the removal:

```
>>> del C[2]                  # delete 3rd element
>>> C
[-15, -10, 0, 5, 10, 15, 20, 25, 30, 35, 40, 45]
>>> del C[2]                  # delete what is now 3rd element
>>> C
[-15, -10, 5, 10, 15, 20, 25, 30, 35, 40, 45]
>>> len(C)                    # length of list
11
```

The command `C.index(10)` returns the index corresponding to the first element with value `10` (this is the 4th element in our sample list, with index 3):

```
>>> C.index(10)              # find index for an element (10)
3
```

To just test if an object with the value `10` is an element in the list, one can write the boolean expression `10 in C`:

```
>>> 10 in C                  # is 10 an element in C?
True
```

Python allows negative indices, which leads to indexing from the right. As demonstrated below, `C[-1]` gives the last element of the list `C`. `C[-2]` is the element before `C[-1]`, and so forth.

```
>>> C[-1]                    # view the last list element
45
>>> C[-2]                    # view the next last list element
40
```

Building long lists by writing down all the elements separated by commas is a tedious process that can easily be automated by a loop, using ideas from Section 2.1.4. Say we want to build a list of degrees from -50 to 200 in steps of 2.5 degrees. We then start with an empty list and use a `while` loop to append one element at a time:

```
C = []
C_value = -50
C_max = 200
while C_value <= C_max:
     C.append(C_value)
     C_value += 2.5
```

In the next sections, we shall see how we can express these six lines of code with just one single statement.

There is a compact syntax for creating variables that refer to the various list elements. Simply list a sequence of variables on the left-hand side of an assignment to a list:

```
>>> somelist = ['book.tex', 'book.log', 'book.pdf']
>>> texfile, logfile, pdf = somelist
>>> texfile
'book.tex'
>>> logfile
'book.log'
>>> pdf
'book.pdf'
```

The number of variables on the left-hand side must match the number of elements in the list, otherwise an error occurs.

A final comment regards the syntax: some list operations are reached by a dot notation, as in `C.append(e)`, while other operations requires the list object as an argument to a function, as in `len(C)`. Although `C.append` for a programmer behaves as a function, it is a function that is reached through a list object, and it is common to say that `append` is a *method* in the list object, not a function. There are no strict rules in

Python whether functionality regarding an object is reached through a method or a function.

2.2.2 For loops

The nature of for loops. When data are collected in a list, we often want to perform the same operations on each element in the list. We then need to walk through all list elements. Computer languages have a special construct for doing this conveniently, and this construct is in Python and many other languages called a `for` loop. Let us use a `for` loop to print out all list elements:

```
degrees = [0, 10, 20, 40, 100]
for C in degrees:
    print 'list element:', C
print 'The degrees list has', len(degrees), 'elements'
```

The `for C in degrees` construct creates a loop over all elements in the list `degrees`. In each pass of the loop, the variable `C` refers to an element in the list, starting with `degrees[0]`, proceeding with `degrees[1]`, and so on, before ending with the last element `degrees[n-1]` (if `n` denotes the number of elements in the list, `len(degrees)`).

The `for` loop specification ends with a colon, and after the colon comes a block of statements that does something useful with the current element. Each statement in the block must be indented, as we explained for `while` loops. In the example above, the block belonging to the `for` loop contains only one statement. The final `print` statement has the same indentation (none in this example) as the `for` statement and is executed as soon as the loop is terminated.

As already mentioned, understanding all details of a program by following the program flow by hand is often a very good idea. Here, we first define a list `degrees` containing 5 elements. Then we enter the `for` loop. In the first pass of the loop, `C` refers to the first element in the list `degrees`, i.e., the `int` object holding the value `0`. Inside the loop we then print out the text `'list element:'` and the value of `C`, which is `0`. There are no more statements in the loop block, so we proceed with the next pass of the loop. `C` then refers to the `int` object `10`, the output now prints `10` after the leading text, we proceed with `C` as the integers `20` and `40`, and finally `C` is `100`. After having printed the list element with value `100`, we move on to the statement after the indented loop block, which prints out the number of list elements. The total output becomes

```
list element: 0
list element: 10
list element: 20
list element: 40
list element: 100
The degrees list has 5 elements
```

Correct indentation of statements is crucial in Python, and we therefore strongly recommend you to work through Exercise 2.22 to learn more about this topic.

Making the table. Our knowledge of lists and `for` loops over elements in lists puts us in a good position to write a program where we collect all the Celsius degrees to appear in the table in a list `Cdegrees`, and then use a `for` loop to compute and write out the corresponding Fahrenheit degrees. The complete program may look like this:

```
Cdegrees = [-20, -15, -10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40]
for C in Cdegrees:
    F = (9.0/5)*C + 32
    print C, F
```

The `print C, F` statement just prints the value of `C` and `F` with a default format, where each number is separated by one space character (blank). This does not look like a nice table (the output is identical to the one shown in Section 2.1.1. Nice formatting is obtained by forcing `C` and `F` to be written in fields of fixed width and with a fixed number of decimals. An appropriate printf format is `%5d` (or `%5.0f`) for `C` and `%5.1f` for `F`. We may also add a headline to the table. The complete program becomes:

```
Cdegrees = [-20, -15, -10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40]
print '    C    F'
for C in Cdegrees:
    F = (9.0/5)*C + 32
    print '%5d %5.1f' % (C, F)
```

This code is found in the file `c2f_table_list.py` and its output becomes

```
    C    F
  -20  -4.0
  -15   5.0
  -10  14.0
   -5  23.0
    0  32.0
    5  41.0
   10  50.0
   15  59.0
   20  68.0
   25  77.0
   30  86.0
   35  95.0
   40 104.0
```

2.3 Alternative implementations with lists and loops

We have already solved the problem of printing out a nice-looking conversion table for Celsius and Fahrenheit degrees. Nevertheless, there are usually many alternative ways to write a program that solves a specific problem. The next paragraphs explore some other possible Python constructs and programs to store numbers in lists and print out tables. The various code snippets are collected in the program file `session.py`.

2.3.1 While loop implementation of a for loop

Any `for` loop can be implemented as a `while` loop. The general code

```
for element in somelist:
    <process element>
```

can be transformed to this `while` loop:

```
index = 0
while index < len(somelist):
    element = somelist[index]
    <process element>
    index += 1
```

In particular, the example involving the printout of a table of Celsius and Fahrenheit degrees can be implemented as follows in terms of a `while` loop:

```
Cdegrees = [-20, -15, -10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40]
index = 0
print '    C    F'
while index < len(Cdegrees):
    C = Cdegrees[index]
    F = (9.0/5)*C + 32
    print '%5d %5.1f' % (C, F)
    index += 1
```

2.3.2 The range construction

It is tedious to write the many elements in the `Cdegrees` in the previous programs. We should use a loop to automate the construction of the `Cdegrees` list. The `range` construction is particularly useful in this regard:

- `range(n)` generates integers `0, 1, 2, ..., n-1`.
- `range(start, stop, step)` generates a sequence if integers `start`, `start+step`, `start+2*step`, and so on up to, *but not including*, `stop`. For example, `range(2, 8, 3)` returns 2 and 5 (and not 8), while `range(1, 11, 2)` returns 1, 3, 5, 7, 9.
- `range(start, stop)` is the same as `range(start, stop, 1)`.

A `for` loop over integers are written as

```
for i in range(start, stop, step):
    ...
```

We can use this construction to create a `Cdegrees` list of the values $-20, -15, \ldots, 40$:

```
Cdegrees = []
for C in range(-20, 45, 5):
    Cdegrees.append(C)

# or just
Cdegrees = range(-20, 45, 5)
```

Note that the upper limit must be greater than `40` to ensure that `40` is included in the range of integers.

Suppose we want to create `Cdegrees` as $-10, -7.5, -5, \ldots, 40$. This time we cannot use `range` directly, because `range` can only create integers and we have decimal degrees such as -7.5 and 1.5. In this more general case, we introduce an integer counter i and generate the C values by the formula $C = -10 + i \cdot 2.5$ for $i = 0, 1, \ldots, 20$. The following Python code implements this task:

```
Cdegrees = []
for i in range(0, 21):
     C = -10 + i*2.5
     Cdegrees.append(C)
```

2.3.3 For loops with list indices

Instead of iterating over a list directly with the construction

```
for element in somelist:
    ...
```

we can equivalently iterate of the list indices and index the list inside the loop:

```
for i in range(len(somelist)):
    element = somelist[i]
    ...
```

Since `len(somelist)` returns the length of `somelist` and the largest legal index is `len(somelist)-1`, because indices always start at 0, `range(len(somelist))` will generate all the correct indices: `0`, `1`, ..., `len(somelist)-1`.

Programmers coming from other languages, such as Fortran, C, C++, Java, and C#, are very much used to `for` loops with integer counters and usually tend to use `for i in range(len(somelist))` and work with `somelist[i]` inside the loop. This might be necessary or convenient, but if possible, Python programmers are encouraged to use `for element in somelist`, which is more elegant to read.

Iterating over loop indices is useful when we need to process two lists simultaneously. As an example, we first create two `Cdegrees` and `Fdegrees` lists, and then we make a list to write out a table with `Cdegrees` and `Fdegrees` as the two columns of the table. Iterating over a loop index is convenient in the final list:

```
Cdegrees = []
n = 21
C_min = -10
C_max = 40
dC = (C_max - C_min)/float(n-1)  # increment in C
```

```
for i in range(0, n):
     C = -10 + i*dC
     Cdegrees.append(C)

Fdegrees = []
for C in Cdegrees:
    F = (9.0/5)*C + 32
    Fdegrees.append(F)

for i in range(len(Cdegrees)):
    C = Cdegrees[i]
    F = Fdegrees[i]
    print '%5.1f %5.1f' % (C, F)
```

Instead of appending new elements to the lists, we can start with lists of the right size, containing zeros, and then index the lists to fill in the right values. Creating a list of length `n` consisting of zeros (for instance) is done by

```
somelist = [0]*n
```

With this construction, the program above can use `for` loops over indices everywhere:

```
n = 21
C_min = -10
C_max = 40
dC = (C_max - C_min)/float(n-1)  # increment in C

Cdegrees = [0]*n
for i in range(len(Cdegrees)):
     Cdegrees[i] = -10 + i*dC

Fdegrees = [0]*n
for i in range(len(Cdegrees)):
    Fdegrees[i] = (9.0/5)*Cdegrees[i] + 32

for i in range(len(Cdegrees)):
    print '%5.1f %5.1f' % (Cdegrees[i], Fdegrees[i])
```

Note that we need the construction `[0]*n` to create a list of the right length, otherwise the index `[i]` will be illegal.

2.3.4 Changing list elements

We have two seemingly alternative ways to traverse a list, either a loop over elements or over indices. Suppose we want to change the `Cdegrees` list by adding 5 to all elements. We could try

```
for c in Cdegrees:
    c += 5
```

but this loop leaves `Cdegrees` unchanged, while

```
for i in range(len(Cdegrees)):
    Cdegrees[i] += 5
```

works as intended. What is wrong with the first loop? The problem is that `c` is an ordinary variable, which refers to a list element in the loop, but when we execute `c += 5`, we let `c` refer to a new `float` object (`c+5`). This object is never inserted in the list. The first two passes of the loop are equivalent to

```
c = Cdegrees[0]   # automatically done in the for statement
c += 5
c = Cdegrees[1]   # automatically done in the for statement
c += 5
```

The variable `c` can only be used to read list elements and never to change them. Only an assignment of the form

```
Cdegrees[i] = ...
```

can change a list element.

There is a way of traversing a list where we get both the index and an element in each pass of the loop:

```
for i, c in enumerate(Cdegrees):
    Cdegrees[i] = c + 5
```

This loop also adds 5 to all elements in the list.

2.3.5 List comprehension

Because running through a list and for each element creating a new element in another list is a frequently encountered task, Python has a special compact syntax for doing this, called *list comprehension*. The general syntax reads

```
newlist = [E(e) for e in list]
```

where `E(e)` represents an expression involving element `e`. Here are three examples:

```
Cdegrees = [-5 + i*0.5 for i in range(n)]
Fdegrees = [(9.0/5)*C + 32 for C in Cdegrees]
C_plus_5 = [C+5 for C in Cdegrees]
```

List comprehensions are recognized as a `for` loop inside square brackets and will be frequently exemplified throughout the book.

2.3.6 Traversing multiple lists simultaneously

We may use the `Cdegrees` and `Fdegrees` lists to make a table. To this end, we need to traverse both arrays. The `for element in list` construction

is not suitable in this case, since it extracts elements from one list only. A solution is to use a `for` loop over the integer indices so that we can index both lists:

```
for i in range(len(Cdegrees)):
    print '%5d %5.1f' % (Cdegrees[i], Fdegrees[i])
```

It happens quite frequently that two or more lists need to be traversed simultaneously. As an alternative to the loop over indices, Python offers a special nice syntax that can be sketched as

```
for e1, e2, e3, ... in zip(list1, list2, list3, ...):
    # work with element e1 from list1, element e2 from list2,
    # element e3 from list3, etc.
```

The `zip` function turns n lists (`list1, list2, list3, ...`) into one list of n-tuples, where each n-tuple `(e1,e2,e3,...)` has its first element (`e1`) from the first list (`list1`), the second element (`e2`) from the second list (`list2`), and so forth. The loop stops when the end of the shortest list is reached. In our specific case of iterating over the two lists `Cdegrees` and `Fdegrees`, we can use the `zip` function:

```
for C, F in zip(Cdegrees, Fdegrees):
    print '%5d %5.1f' % (C, F)
```

It is considered more *Pythonic* to iterate over list elements, here `C` and `F`, rather than over list indices as in the `for i in range(len(Cdegrees))` construction.

2.4 Nested lists

Nested lists are list objects where the elements in the lists can be lists themselves. A couple of examples will motivate for nested lists and illustrate the basic operations on such lists.

2.4.1 A table as a list of rows or columns

Our table data have so far used one separate list for each column. If there were n columns, we would need n list objects to represent the data in the table. However, we think of a table as *one* entity, not a collection of n columns. It would therefore be natural to use one argument for the whole table. This is easy to achieve using a *nested list*, where each entry in the list is a list itself. A table object, for instance, is a list of lists, either a list of the row elements of the table or a list of the column elements of the table. Here is an example where the table is a list of two columns, and each column is a list of numbers:

```
Cdegrees = range(-20, 41, 5)   # -20, -15, ..., 35, 40
Fdegrees = [(9.0/5)*C + 32 for C in Cdegrees]

table = [Cdegrees, Fdegrees]
```

(Note that any value in $[41, 45]$ can be used as second argument (stop value) to `range` and will ensure that 40 is included in the range of generate numbers.)

With the subscript `table[0]` we can access the first element in `table`, which is nothing but the `Cdegrees` list, and with `table[0][2]` we reach the third element in the first element, i.e., `Cdegrees[2]`.

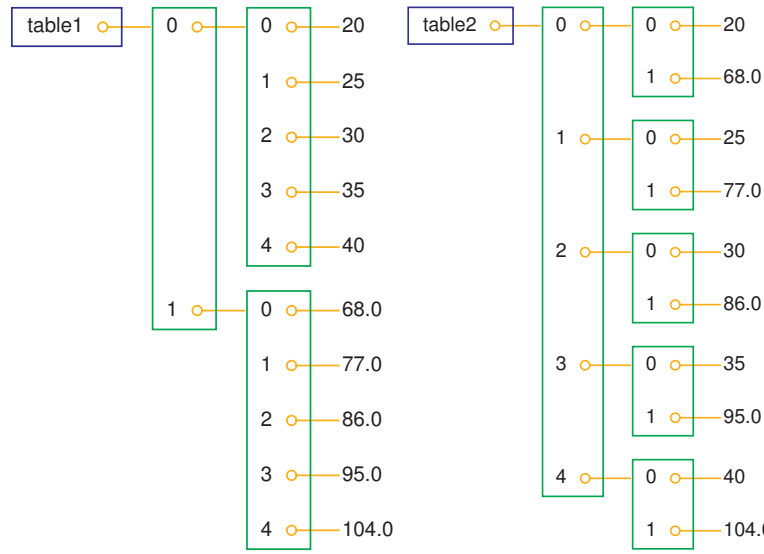

Fig. 2.2 Two ways of creating a table as a nested list. Left: table of columns `C` and `F`, where `C` and `F` are lists. Right: table of rows, where each row `[C, F]` is a list of two floats.

However, tabular data with rows and columns usually have the convention that the underlying data is a nested list where the first index counts the rows and the second index counts the columns. To have `table` on this form, we must construct `table` as a list of `[C, F]` pairs. The first index will then run over rows `[C, F]`. Here is how we may construct the nested list:

```
table = []
for C, F in zip(Cdegrees, Fdegrees):
    table.append([C, F])
```

We may shorten this code segment by introducing a list comprehension:

```
table = [[C, F] for C, F in zip(Cdegrees, Fdegrees)]
```

This construction loops through pairs `C` and `F`, and for each pass in the loop we create a list element `[C, F]`.

The subscript `table[1]` refers to the second element in `table`, which is a `[C, F]` pair, while `table[1][0]` is the `C` value and `table[1][1]` is the `F` value. Figure 2.2 illustrates both a list of columns and a list of pairs. Using this figure, you can realize that the first index looks up an element in the outer list, and that this element can be indexed with the second index.

2.4.2 Printing objects

Modules for pretty print of objects. We may write `print table` to immediately view the nested list `table` from the previous section. In fact, any Python object `obj` can be printed to the screen by the command `print obj`. The output is usually one line, and this line may become very long if the list has many elements. For example, a long list like our `table` variable, demands a quite long line when printed.

```
[[-20, -4.0], [-15, 5.0], [-10, 14.0], ............., [40, 104.0]]
```

Splitting the output over several shorter lines makes the layout nicer and more readable. The `pprint` module offers a *pretty print* functionality for this purpose. The usage of `pprint` looks like

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

and the corresponding output becomes

```
[[-20, -4.0],
 [-15, 5.0],
 [-10, 14.0],
 [-5, 23.0],
 [0, 32.0],
 [5, 41.0],
 [10, 50.0],
 [15, 59.0],
 [20, 68.0],
 [25, 77.0],
 [30, 86.0],
 [35, 95.0],
 [40, 104.0]]
```

With this book comes a slightly modified `pprint` module having the name `scitools.pprint2`. This module allows full format control of the printing of the `float` objects in lists by specifying `scitools.pprint2.float_format` as a printf format string. The following example demonstrates how the output format of real numbers can be changed:

```
>>> import pprint, scitools.pprint2
>>> somelist = [15.8, [0.2, 1.7]]
>>> pprint.pprint(somelist)
[15.800000000000001, [0.20000000000000001, 1.7]]
>>> scitools.pprint2.pprint(somelist)
[15.8, [0.2, 1.7]]
>>> # default output is '%g', change this to
>>> scitools.pprint2.float_format = '%.2e'
```

```
>>> scitools.pprint2.pprint(somelist)
[1.58e+01, [2.00e-01, 1.70e+00]]
```

As can be seen from this session, the `pprint` module writes floating-point numbers with a lot of digits, in fact so many that we explicitly see the round-off errors. Many find this type of output is annoying and that the default output from the `scitools.pprint2` module is more like one would desire and expect.

The `pprint` and `scitools.pprint2` modules also have a function `pformat`, which works as the `pprint` function, but it returns a pretty formatted string rather than printing the string:

```
s = pprint.pformat(somelist)
print s
```

This last `print` statement prints the same as `pprint.pprint(somelist)`.

Manual printing. Many will argue that tabular data such as those stored in the nested `table` list are not printed in a particularly pretty way by the `pprint` module. One would rather expect pretty output to be a table with two nicely aligned columns. To produce such output we need to code the formatting manually. This is quite easy: we loop over each row, extract the two elements `C` and `F` in each row, and print these in fixed-width fields using the printf syntax. The code goes as follows:

```
for C, F in table:
    print '%5d %5.1f' % (C, F)
```

2.4.3 Extracting sublists

Python has a nice syntax for extracting parts of a list structure. Such parts are known as *sublists* or *slices*:

`A[i:]` is the sublist starting with index `i` in `A` and continuing to the end of `A`:

```
>>> A = [2, 3.5, 8, 10]
>>> A[2:]
[8, 10]
```

`A[i:j]` is the sublist starting with index `i` in `A` and continuing up to and including index `j-1`. Make sure you remember that the element corresponding to index `j` is not included in the sublist:

```
>>> A[1:3]
[3.5, 8]
```

`A[:i]` is the sublist starting with index 0 in `A` and continuing up to and including the element with index `i-1`:

```
>>> A[:3]
[2, 3.5, 8]
```

`A[1:-1]` extracts all elements except the first and the last (recall that index `-1` refers to the last element), and `A[:]` is the whole list:

```
>>> A[1:-1]
[3.5, 8]
>>> A[:]
[2, 3.5, 8, 10]
```

In nested lists we may use slices in the first index, e.g.,

```
>>> table[4:]
[[0, 32.0], [5, 41.0], [10, 50.0], [15, 59.0], [20, 68.0],
 [25, 77.0], [30, 86.0], [35, 95.0], [40, 104.0]]
```

We can also slice the second index, or both indices:

```
>>> table[4:7][0:2]
[[0, 32.0], [5, 41.0]]
```

Observe that `table[4:7]` makes a list `[[0, 32.0], [5, 41.0], [10, 50.0]]` with three elements. The slice `[0:2]` acts on this sublist and picks out its first two elements, with indices 0 and 1.

Sublists are always copies of the original list, so if you modify the sublist the original list remains unaltered and vice versa:

```
>>> l1 = [1, 4, 3]
>>> l2 = l1[:-1]
>>> l2
[1, 4]
>>> l1[0] = 100
>>> l1              # l1 is modified
[100, 4, 3]
>>> l2              # l2 is not modified
[1, 4]
```

The fact that slicing makes a copy can also be illustrated by the following code:

```
>>> B = A[:]
>>> C = A
>>> B == A
True
>>> B is A
False
>>> C is A
True
```

The `B == A` boolean expression is `True` if all elements in `B` are equal to the corresponding elements in `A`. The test `B is A` is `True` if `A` and `B` are names for the same list. Setting `C = A` makes `C` refer to the same list object as `A`, while `B = A[:]` makes `B` refer to a copy of the list referred to by `A`.

Example. We end this information on sublists by writing out the part of the `table` list of `[C, F]` rows (see Section 2.4) where the Celsius degrees are between 10 and 35 (not including 35):

```
>>> for C, F in table[Cdegrees.index(10):Cdegrees.index(35)]:
...     print '%5.0f %5.1f' % (C, F)
...
   10  50.0
   15  59.0
   20  68.0
   25  77.0
   30  86.0
```

You should always stop reading and convince yourself that you understand why a code segment produces the printed output. In this latter example, `Cdegrees.index(10)` returns the index corresponding to the value `10` in the `Cdegrees` list. Looking at the `Cdegrees` elements, one realizes (do it!) that the `for` loop is equivalent to

```
for C, F in table[6:11]:
```

This loop runs over the indices $6, 7, \ldots, 10$ in `table`.

2.4.4 Traversing nested lists

We have seen that traversing the nested list `table` could be done by a loop of the form

```
for C, F in table:
    # process C and F
```

This is natural code when we know that `table` is a list of `[C, F]` lists. Now we shall address more general nested lists where we do not necessarily know how many elements there are in each list element of the list.

Suppose we use a nested list `scores` to record the scores of players in a game: `scores[i]` holds a list of the historical scores obtained by player number `i`. Different players have played the game a different number of times, so the length of `scores[i]` depends on `i`. Some code may help to make this clearer:

```
scores = []
# score of player no. 0:
scores.append([12, 16, 11, 12])
# score of player no. 1:
scores.append([9])
# score of player no. 2:
scores.append([6, 9, 11, 14, 17, 15, 14, 20])
```

The list `scores` has three elements, each element corresponding to a player. The element no. `g` in the list `scores[p]` corresponds to the score obtained in game number `g` played by player number `p`. The length of

the lists `scores[p]` varies and equals 4, 1, and 8 for `p` equal to 0, 1, and 2, respectively.

In the general case we may have n players, and some may have played the game a large number of times, making `scores` potentially a big nested list. How can we traverse the `scores` list and write it out in a table format with nicely formatted columns? Each row in the table corresponds to a player, while columns correspond to scores. For example, the data initialized above can be written out as

```
  12  16  11  12
   9
   6   9  11  14  17  15  14  20
```

In a program, we must use two *nested loops*, one for the elements in `scores` and one for the elements in the sublists of `scores`. The example below will make this clear.

There are two basic ways of traversing a nested list: either we use integer indices for each index, or we use variables for the list elements. Let us first exemplify the index-based version:

```
for p in range(len(scores)):
    for g in range(len(scores[p])):
        score = scores[p][g]
        print '%4d' % score,
    print
```

With the trailing comma after the print string, we avoid a newline so that the column values in the table (i.e., scores for one player) appear at the same line. The single `print` command after the loop over `c` adds a newline after each table row. The reader is encouraged to go through the loops by hand and simulate what happens in each statement (use the simple `scores` list initialized above).

The alternative version where we use variables for iterating over the elements in the `scores` list and its sublists looks like this:

```
for player in scores:
    for game in player:
        print '%4d' % game,
    print
```

Again, the reader should step through the code by hand and realize what the values of `player` and `game` are in each pass of the loops.

In the very general case we can have a nested list with many indices: `somelist[i1][i2][i3]...`. To visit each of the elements in the list, we use as many nested `for` loops as there are indices. With four indices, iterating over integer indices look as

```
for i1 in range(len(somelist)):
    for i2 in range(len(somelist[i1])):
        for i3 in range(len(somelist[i1][i2])):
            for i4 in range(len(somelist[i1][i2][i3])):
                value = somelist[i1][i2][i3][i4]
                # work with value
```

The corresponding version iterating over sublists becomes

```
for sublist1 in somelist:
    for sublist2 in sublist1:
        for sublist3 in sublist2:
            for sublist4 in sublist3:
                value = sublist4
                # work with value
```

We recommend to do Exercise 3.29 to get a better understanding of nested `for` loops.

2.5 Tuples

Tuples are very similar to lists, but tuples cannot be changed. That is, a tuple can be viewed as a *constant list*. While lists employ square brackets, tuples are written with standard parentheses:

```
>>> t = (2, 4, 6, 'temp.pdf')    # define a tuple with name t
```

One can also drop the parentheses in many occasions:

```
>>> t = 2, 4, 6, 'temp.pdf'
>>> for element in 'myfile.txt', 'yourfile.txt', 'herfile.txt':
...     print element,
...
myfile.txt yourfile.txt herfile.txt
```

The `for` loop here is over a tuple, because a comma separated sequence of objects, even without enclosing parentheses, becomes a tuple. Note the trailing comma in the `print` statement. This comma suppresses the final newline that the `print` command automatically adds to the output string. This is the way to make several `print` statements build up one line of output.

Much functionality for lists is also available for tuples, for example:

```
>>> t = t + (-1.0, -2.0)           # add two tuples
>>> t
(2, 4, 6, 'temp.pdf', -1.0, -2.0)
>>> t[1]                           # indexing
4
>>> t[2:]                          # subtuple/slice
(6, 'temp.pdf', -1.0, -2.0)
>>> 6 in t                         # membership
True
```

Any list operation that changes the list will not work for tuples:

```
>>> t[1] = -1
...
TypeError: object does not support item assignment

>>> t.append(0)
...
```

```
AttributeError: 'tuple' object has no attribute 'append'

>>> del t[1]
...
TypeError: object doesn't support item deletion
```

Some list methods, like `index`, are not available for tuples. So why do we need tuples when lists can do more than tuples?

- Tuples protect against accidental changes of their contents.
- Code based on tuples is faster than code based on lists.
- Tuples are frequently used in Python software that you certainly will make use of, so you need to know this data type.

There is also a fourth argument, which is important for a data type called dictionaries (introduced in Section 6.1): tuples can be used as keys in dictionaries while lists can not.

2.6 Summary

2.6.1 Chapter topics

While loops. Loops are used to repeat a collection of program statements several times. The statements that belong to the loop must be consistently indented in Python. A `while` loop runs as long as a condition evaluates to `True`:

```
>>> t = 0; dt = 0.5; T = 2
>>> while t <= T:
...       print t
...       t += dt
...
0
0.5
1.0
1.5
2.0
>>> print 'Final t:', t, '; t <= T is', t <= T
Final t: 2.5 ; t <= T is False
```

Lists. A list is used to collect a number of values or variables in an ordered sequence.

```
>>> mylist = [t, dt, T, 'mynumbers.dat', 100]
```

A list element can be any Python object, including numbers, strings, functions, and other lists, for instance.

The table below shows some important list operations (only a subset of these are explained in the present chapter).

Construction	Meaning
`a = []`	initialize an empty list
`a = [1, 4.4, 'run.py']`	initialize a list
`a.append(elem)`	add `elem` object to the end
`a + [1,3]`	add two lists
`a.insert(i, e)`	insert element `e` before index `i`
`a[3]`	index a list element
`a[-1]`	get last list element
`a[1:3]`	slice: copy data to sublist (here: index 1, 2)
`del a[3]`	delete an element (index `3`)
`a.remove(e)`	remove an element with value `e`
`a.index('run.py')`	find index corresponding to an element's value
`'run.py' in a`	test if a value is contained in the list
`a.count(v)`	count how many elements that have the value `v`
`len(a)`	number of elements in list `a`
`min(a)`	the smallest element in `a`
`max(a)`	the largest element in `a`
`sum(a)`	add all elements in `a`
`sorted(a)`	return sorted version of list `a`
`reversed(a)`	return reversed sorted version of list `a`
`b[3][0][2]`	nested list indexing
`isinstance(a, list)`	is `True` if `a` is a list
`type(a) is list`	is `True` if `a` is a list

Nested lists. If the list elements are also lists, we have a nested list. The following session summarizes indexing and loop traversal of nested lists:

```
>>> nl = [[0, 0, 1], [-1, -1, 2], [-10, 10, 5]]
>>> nl[0]
[0, 0, 1]
>>> nl[-1]
[-10, 10, 5]
>>> nl[0][2]
1
>>> nl[-1][0]
-10
>>> for p in nl:
...     print p
...
[0, 0, 1]
[-1, -1, 2]
[-10, 10, 5]
>>> for a, b, c in nl:
...     print '%3d %3d %3d' % (a, b, c)
...
  0   0   1
 -1  -1   2
-10  10   5
```

Tuples. A tuple can be viewed as a constant list: no changes in the contents of the tuple is allowed. Tuples employ standard parentheses or no parentheses, and elements are separated with comma as in lists:

```
>>> mytuple = (t, dt, T, 'mynumbers.dat', 100)
>>> mytuple =  t, dt, T, 'mynumbers.dat', 100
```

Many list operations are also valid for tuples, but those that changes the list content cannot be used with tuples (examples are `append`, `del`, `remove`, `index`, and `sort`).

An object `a` containing an ordered collection of other objects such that `a[i]` refers to object with index `i` in the collection, is known as a *sequence* in Python. Lists, tuples, strings, and arrays are examples on sequences. You choose a sequence type when there is a natural ordering of elements. For a collection of unordered objects a *dictionary* (see Section 6.1) is often more convenient.

For loops. A `for` loop is used to run through the elements of a list or a tuple:

```
>>> for elem in [10, 20, 25, 27, 28.5]:
...     print elem,
...
10 20 25 27 28.5
```

The trailing comma after the `print` statement prevents the newline character, which otherwise `print` would automatically add.

The `range` function is frequently used in `for` loops over a sequence of integers. Recall that `range(start, stop, inc)` does not include the upper limit `stop` among the list item.

```
>>> for elem in range(1, 5, 2):
...     print elem,
...
1 3
>>> range(1, 5, 2)
[1, 3]
```

Implementation of a sum $\sum_{j=M}^{N} q(j)$, where $q(j)$ is some mathematical expression involving the integer counter j, is normally implemented using a `for` loop. Choosing, e.g., $q(j) = 1/j^2$, the sum is calculated by

```
s = 0  # accumulation variable
for j in range(M, N+1, 1):
    s += 1./j**2
```

Pretty print. To print a list `a`, `print a` can be used, but the `pprint` and `scitools.pprint2` modules and their `pprint` function give a nicer layout of the output for long and nested lists. The `scitools.pprint2` module has the possibility to control the formatting of floating-point numbers.

Terminology. The important computer science terms in this chapter are

- list
- tuple
- nested list (and nested tuple)
- sublist (subtuple) or slice `a[i:j]`
- `while` loop
- `for` loop
- list comprehension
- boolean expression

2.6.2 Example: Analyzing list data

Problem. The file src/misc/Oxford_sun_hours.txt[2] contains data of the number of sun hours in Oxford, UK, for every month since January 1929. The data are already on a suitable nested list format:

```
[
[43.8, 60.5, 190.2, ...],
[49.9, 54.3, 109.7, ...],
[63.7, 72.0, 142.3, ...],
...
]
```

The list in every line holds the number of sun hours for each of the year's 12 months. That is, the first index in the nested list corresponds to year and the second index corresponds to the month number. More precisely, the double index `[i][j]` corresponds to year 1929 + `i` and month 1 + `j` (January being month number 1).

The task is to define this nested list in a program and do the following data analysis.

- Compute the average number of sun hours for each month during the total data period (1929-2009).
- Which month has the best weather according to the means found in the preceding task?
- For each decade, 1930-1939, 1940-1949, ..., 2000-2009, compute the average number of sun hours per day in January and December. For example, use December 1949, January 1950, ..., December 1958, and January 1959 as data for the decade 1950-1959. Are there any noticeable differences between the decades?

Solution. Initializing the data is easy: just copy the data from the `Oxford_sun_hours.txt` file into the program file and set a variable name on the left hand side (the long and wide code is only indicated here):

```
data = [
[43.8, 60.5, 190.2, ...],
[49.9, 54.3, 109.7, ...],
[63.7, 72.0, 142.3, ...],
...
]
```

For task 1, we need to establish a list `monthly_mean` with the results from the computation, i.e., `monthly_mean[2]` holds the average number of sun hours for March in the period 1929-2009. The average is computed in the standard way: for each month, we run through all the years, sum up the values, and finally divide by the number of years $(2009 - 1929 + 1)$.

When looping over years and months it is convenient to have loop variables running over the true years (1929 to 2009) and the standard

[2] `http://tinyurl.com/pwyasaa/misc/Oxford_sun_hours.txt`

month number (1 to 12). These variables must be correctly translated to indices in the `data` list such that all indices start at 0. The following code produces the answers to task 1:

```
monthly_mean = [0]*12      # list with 12 elements
for month in range(1, 13):
    m = month - 1   # corresponding list index (starts at 0)
    s = 0           # sum
    n = 2009 - 1929 + 1  # no of years
    for year in range(1929, 2010):
        y = year - 1929  # corresponding list index (starts at 0)
        s += data[y][m]
    monthly_mean[m] = s/n
```

An alternative solution would be to introduce separate variables for the monthly averages, say `Jan_mean`, `Feb_mean`, etc. The reader should as an exercise write the code associated with such a solution and realize that using the `monthly_mean` list is more elegant and yields much simpler and shorter code. Separate variables might be an okay solution for 2-3 variables, but as many as 12.

Perhaps we want a nice-looking printout of the results. This can elegantly be created by first defining a tuple (or list) of the names of the months and then running through this list in parallel with `monthly_mean`:

```
month_names = 'Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun',\
              'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'
for name, value in zip(month_names, monthly_mean):
    print '%s: %.1f' % (name, value)
```

The printout becomes

```
Jan: 55.9
Feb: 71.8
Mar: 115.1
Apr: 151.3
May: 188.7
Jun: 196.1
Jul: 191.4
Aug: 182.1
Sep: 136.7
Oct: 103.4
Nov: 66.6
Dec: 51.7
```

Task 2 can be solved by pure inspection of the above printout, which reveals that June is the winner. However, since we are learning programming, we should be able to replace our eyes with some computer code to automate the task. The maximum value `max_value` of a list like `monthly_mean` is simply obtained by `max(monthly_mean)`. The corresponding index, needed to find the right name of the corresponding month, is found from `monthly_mean.index(max_value)`. The code for task 2 is then

```
max_value = max(monthly_mean)
month = month_names[monthly_mean.index(max_value)]
print '%s has best weather with %.1f sun hours on average' % \
      (month, max_value)
```

Task 3 requires us to first develop an algorithm for how to compute the decade averages. The algorithm, expressed with words, goes as follows. We loop over the decades, and for each decade, we loop over its years, and for each year, we add the December data of the previous year and the January data of the current year to an accumulation variable. Dividing this accumulation variable by $10 \cdot 2 \cdot 30$ gives the average number of sun hours per day in the winter time for the particular decade. The code segment below expresses this algorithm in the Python language:

```
decade_mean = []
for decade_start in range(1930, 2010, 10):
    Jan_index = 0; Dec_index = 11  # indices
    s = 0
    for year in range(decade_start, decade_start+10):
        y = year - 1929  # list index
        print data[y-1][Dec_index] + data[y][Jan_index]
        s += data[y-1][Dec_index] + data[y][Jan_index]
    decade_mean.append(s/(20.*30))
for i in range(len(decade_mean)):
    print 'Decade %d-%d: %.1f' % \
          (1930+i*10, 1939+i*10, decade_mean[i])
```

The output becomes

```
Decade 1930-1939: 1.7
Decade 1940-1949: 1.8
Decade 1950-1959: 1.8
Decade 1960-1969: 1.8
Decade 1970-1979: 1.6
Decade 1980-1989: 2.0
Decade 1990-1999: 1.8
Decade 2000-2009: 2.1
```

The complete code is found in the file `sun_data.py`.

Remark. The file `Oxford_sun_hours.txt` is based on data from the UK Met Office[3]. A Python program for downloading the data, interpreting the content, and creating a file like `Oxford_sun_hours.txt` is explained in detail in Section 6.3.3.

2.6.3 How to find more Python information

This book contains only fragments of the Python language. When doing your own projects or exercises you will certainly feel the need for looking up more detailed information on modules, objects, etc. Fortunately, there is a lot of excellent documentation on the Python programming language.

The primary reference is the official Python documentation website[4]: `docs.python.org`. Here you can find a Python tutorial, the very useful *Library Reference* [2], and a *Language Reference*, to mention some key documents. When you wonder what functions you can find in a module, say the `math` module, you can go to the Library Reference search for

[3] `http://www.metoffice.gov.uk/climate/uk/stationdata/`
[4] `http://docs.python.org/index.html`

math, which quickly leads you to the official documentation of the `math` module. Alternatively, you can go to the index of this document and pick the `math (module)` item directly. Similarly, if you want to look up more details of the printf formatting syntax, go to the index and follow the *printf-style formatting* index.

Warning

A word of caution is probably necessary here. Reference manuals are very technical and written primarily for experts, so it can be quite difficult for a newbie to understand the information. An important ability is to browse such manuals and dig out the key information you are looking for, without being annoyed by all the text you do not understand. As with programming, reading manuals efficiently requires a lot of training.

A tool somewhat similar to the Python Standard Library documentation is the `pydoc` program. In a terminal window you write

Terminal

```
Terminal> pydoc math
```

In IPython there are two corresponding possibilities, either

```
In [1]: !pydoc math
```

or

```
In [2]: import math
In [3]: help(math)
```

The documentation of the complete `math` module is shown as plain text. If a specific function is wanted, we can ask for that directly, e.g., `pydoc math.tan`. Since `pydoc` is very fast, many prefer `pydoc` over web pages, but `pydoc` has often less information compared to the web documentation of modules.

There are also a large number of books about Python. Beazley [1] is an excellent reference that improves and extends the information in the web documentation. The *Learning Python* book [17] has been very popular for many years as an introduction to the language. There is a special web page[5] listing most Python books on the market. Very few books target scientific computing with Python, but [3] gives an introduction to Python for mathematical applications and is more compact and advanced than the present book. It also serves as an excellent reference for the capabilities of Python in a scientific context. A comprehensive book on the use of Python for assisting and automating scientific work is [13].

[5] `http://wiki.python.org/moin/PythonBooks`

Quick references, which list almost to all Python functionality in compact tabular form, are very handy. We recommend in particular the one by Richard Gruet[6] [5].

The website http://www.python.org/doc/ contains a list of useful Python introductions and reference manuals.

2.7 Exercises

Exercise 2.1: Make a Fahrenheit-Celsius conversion table

Write a Python program that prints out a table with Fahrenheit degrees $0, 10, 20, \ldots, 100$ in the first column and the corresponding Celsius degrees in the second column.

Hint. Modify the `c2f_table_while.py` program from Section 2.1.2. Filename: `f2c_table_while.py`.

Exercise 2.2: Generate an approximate Fahrenheit-Celsius conversion table

Many people use an approximate formula for quickly converting Fahrenheit (F) to Celsius (C) degrees:

$$C \approx \hat{C} = (F - 30)/2 \tag{2.2}$$

Modify the program from Exercise 2.1 so that it prints three columns: F, C, and the approximate value $\hat{C}$. Filename: `f2c_approx_table.py`.

Exercise 2.3: Work with a list

Set a variable `primes` to a list containing the numbers 2, 3, 5, 7, 11, and 13. Write out each list element in a `for` loop. Assign 17 to a variable `p` and add `p` to the end of the list. Print out the entire new list. Filename: `primes.py`.

Exercise 2.4: Generate odd numbers

Write a program that generates all odd numbers from 1 to `n`. Set `n` in the beginning of the program and use a `while` loop to compute the numbers. (Make sure that if `n` is an even number, the largest generated odd number is `n-1`.) Filename: `odd.py`.

[6] `http://rgruet.free.fr/PQR27/PQR2.7.html`

Exercise 2.5: Sum of first n integers

Write a program that computes the sum of the integers from 1 up to and including n. Compare the result with the famous formula $n(n+1)/2$.
Filename: `sum_int.py`.

Exercise 2.6: Generate equally spaced coordinates

We want to generate $n+1$ equally spaced x coordinates in $[a, b]$. Store the coordinates in a list.

a) Start with an empty list, use a `for` loop and append each coordinate to the list.

Hint. With n intervals, corresponding to $n+1$ points, in $[a, b]$, each interval has length $h = (b-a)/n$. The coordinates can then be generated by the formula $x_i = a + ih$, $i = 0, \ldots, n+1$.

b) Use a list comprehension (see Section 2.3.5).
Filename: `coor.py`.

Exercise 2.7: Make a table of values from a formula

The purpose of this exercise is to write code that prints a nicely formatted table of t and $y(t)$ values, where

$$y(t) = v_0 t - \frac{1}{2} g t^2 \,.$$

Use $n+1$ uniformly spaced t values throughout the interval $[0, 2v_0/g]$.

a) Use a `for` loop to produce the table.

b) Add code with a `while` loop to produce the table.

Hint. Because of potential round-off errors, you may need to adjust the upper limit of the `while` loop to ensure that the last point ($t = 2v_0/g$, $y = 0$) is included.
Filename: `ball_table1.py`.

Exercise 2.8: Store values from a formula in lists

This exercise aims to produce the same table of numbers as in Exercise 2.7, but with different code. First, store the t and y values in two lists `t` and `y`. Thereafter, write out a nicely formatted table by traversing the two lists with a `for` loop.

Hint. In the `for` loop, use either `zip` to traverse the two lists in parallel, or use an index and the `range` construction.
Filename: `ball_table2.py`.

Exercise 2.9: Simulate operations on lists by hand

You are given the following program:

```
a = [1, 3, 5, 7, 11]
b = [13, 17]
c = a + b
print c
b[0] = -1
d = [e+1 for e in a]
print d
d.append(b[0] + 1)
d.append(b[-1] + 1)
print d[-2:]
for e1 in a:
    for e2 in b:
        print e1 + e2
```

Go through each statement and explain what is printed by the program.

Exercise 2.10: Compute a mathematical sum

The following code is supposed to compute the sum $s = \sum_{k=1}^{M} \frac{1}{k}$:

```
s = 0;  k = 1;  M = 100
while k < M:
    s += 1/k
print s
```

This program does not work correctly. What are the three errors? (If you try to run the program, nothing will happen on the screen. Type `Ctrl+c`, i.e., hold down the Control (`Ctrl`) key and then type the `c` key, to stop the program.) Write a correct program.

Hint. There are two basic ways to find errors in a program:

1. read the program carefully and think about the consequences of each statement,
2. print out intermediate results and compare with hand calculations.

First, try method 1 and find as many errors as you can. Thereafter, try method 2 for $M = 3$ and compare the evolution of `s` with your own hand calculations.
Filename: `sum_while.py`.

Exercise 2.11: Replace a while loop by a for loop

Rewrite the corrected version of the program in Exercise 2.10 using a `for` loop over `k` values instead of a `while` loop. Filename: `sum_for.py`.

Exercise 2.12: Simulate a program by hand

Consider the following program for computing with interest rates:

```
initial_amount = 100
p = 5.5  # interest rate
amount = initial_amount
years = 0
while amount <= 1.5*initial_amount:
    amount = amount + p/100*amount
    years = years + 1
print years
```

a) Use a pocket calculator or an interactive Python shell and work through the program calculations by hand. Write down the value of `amount` and `years` in each pass of the loop.

b) Set `p = 5` instead. Why will the loop now run forever? (Apply Ctrl+c, see Exercise 2.10, to stop a program with a loop that runs forever.) Make the program robust against such errors.

c) Make use of the operator `+=` wherever possible in the program.

d) Explain with words what type of mathematical problem that is solved by this program.
Filename: `interest_rate_loop.py`.

Exercise 2.13: Explore Python documentation

Suppose you want to compute with the inverse sine function: $\sin^{-1} x$. How do you do that in a Python program?

Hint. The `math` module has an inverse sine function. Find the correct name of the function by looking up the module content in the online Python Standard Library[7] document or use `pydoc`, see Section 2.6.3.
Filename: `inverse_sine.py`.

Exercise 2.14: Index a nested lists

We define the following nested list:

```
q = [['a', 'b', 'c'], ['d', 'e', 'f'], ['g', 'h']]
```

a) Index this list to extract 1) the letter `a`; 2) the list `['d', 'e', 'f']`; 3) the last element `h`; 4) the `d` element. Explain why `q[-1][-2]` has the value `g`.

b) We can visit all elements of `q` using this nested `for` loop:

[7] `http://docs.python.org/2/library/`

```
for i in q:
    for j in range(len(i)):
        print i[j]
```

What type of objects are `i` and `j`?
Filename: `index_nested_list.py`.

Exercise 2.15: Store data in lists

Modify the program from Exercise 2.2 so that all the F, C, and $\hat{C}$ values are stored in separate lists `F`, `C`, and `C_approx`, respectively. Then make a nested list `conversion` so that `conversion[i]` holds a row in the table: `[F[i], C[i], C_approx[i]]`. Finally, let the program traverse the `conversion` list and write out the same table as in Exercise 2.2.
Filename: `f2c_approx_lists.py`.

Exercise 2.16: Store data in a nested list

a) Compute two lists of t and y values as explained in Exercise 2.8. Store the two lists in a new nested list `ty1` such that `ty1[0]` and `ty1[1]` correspond to the two lists. Write out a table with t and y values in two columns by looping over the data in the `ty1` list. Each number should be written with two decimals.

b) Make a list `ty2` which holds each row in the table of t and y values (`ty1` is a list of table columns while `ty2` is a list of table rows, as explained in Section 2.4). Loop over the `ty2` list and write out the t and y values with two decimals each.
Filename: `ball_table3.py`.

Exercise 2.17: Values of boolean expressions

Explain the outcome of each of the following boolean expressions:

```
C = 41
C == 40
C != 40 and C < 41
C != 40 or  C < 41
not C == 40
not C > 40
C <= 41
not False
True and False
False or True
False or False or False
True and True and False
False == 0
True == 0
True == 1
```

Note

It makes sense to compare `True` and `False` to the integers 0 and 1, but not other integers (e.g., `True == 12` is `False` although the *integer* 12 evaluates to `True` in a boolean context, as in `bool(12)` or `if 12`).

Exercise 2.18: Explore round-off errors from a large number of inverse operations

Maybe you have tried to hit the square root key on a calculator multiple times and then squared the number again an equal number of times. These set of inverse mathematical operations should of course bring you back to the starting value for the computations, but this does not always happen. To avoid tedious pressing of calculator keys, we can let a computer automate the process. Here is an appropriate program:

```
from math import sqrt
for n in range(1, 60):
    r = 2.0
    for i in range(n):
        r = sqrt(r)
    for i in range(n):
        r = r**2
    print '%d times sqrt and **2: %.16f' % (n, r)
```

Explain with words what the program does. Then run the program. Round-off errors are here completely destroying the calculations when `n` is large enough! Investigate the case when we come back to 1 instead of 2 by fixing an `n` value where this happens and printing out `r` in both `for` loops over `i`. Can you now explain why we come back to 1 and not 2? Filename: `repeated_sqrt`.

Exercise 2.19: Explore what zero can be on a computer

Type in the following code and run it:

```
eps = 1.0
while 1.0 != 1.0 + eps:
    print '...............', eps
    eps = eps/2.0
print 'final eps:', eps
```

Explain with words what the code is doing, line by line. Then examine the output. How can it be that the "equation" $1 \neq 1 + \text{eps}$ is not true? Or in other words, that a number of approximately size 10^{-16} (the final `eps` value when the loop terminates) gives the same result as if `eps` were zero? Filename: `machine_zero.py`.

Remarks. The nonzero `eps` value computed above is called *machine epsilon* or *machine zero* and is an important parameter to know, especially when certain mathematical techniques are applied to control round-off errors.

Exercise 2.20: Compare two real numbers with a tolerance

Run the following program:

```
a = 1/947.0*947
b = 1
if a != b:
    print 'Wrong result!'
```

The lesson learned from this program is that one should never compare two floating-point objects directly using `a == b` or `a != b`, because round-off errors quickly make two identical mathematical values different on a computer. A better result is to test if `abs(a - b) < tol`, where `tol` is a very small number. Modify the test according to this idea. Filename: `compare_floats.py`.

Exercise 2.21: Interpret a code

The function `time` in the module `time` returns the number of seconds since a particular date (called the Epoch, which is January 1, 1970, on many types of computers). Python programs can therefore use `time.time()` to mimic a stop watch. Another function, `time.sleep(n)` causes the program to pause for `n` seconds and is handy for inserting a pause. Use this information to explain what the following code does:

```
import time
t0 = time.time()
while time.time() - t0 < 10:
    print '....I like while loops!'
    time.sleep(2)
print 'Oh, no - the loop is over.'
```

How many times is the `print` statement inside the loop executed? Now, copy the code segment and change the `<` sign in the loop condition to a `>` sign. Explain what happens now. Filename: `time_while.py`.

Exercise 2.22: Explore problems with inaccurate indentation

Type in the following program in a file and check carefully that you have exactly the same spaces:

```
C = -60; dC = 2
while C <= 60:
    F = (9.0/5)*C + 32
      print C, F
C = C + dC
```

Run the program. What is the first problem? Correct that error. What is the next problem? What is the cause of that problem? (See Exercise 2.10 for how to stop a hanging program.) Filename: `indentation.*`.

Remarks. The lesson learned from this exercise is that one has to be very careful with indentation in Python programs! Other computer languages usually enclose blocks belonging to loops in curly braces, parentheses, or begin-end marks. Python's convention with using solely indentation contributes to visually attractive, easy-to-read code, at the cost of requiring a pedantic attitude to blanks from the programmer.

Exercise 2.23: Explore punctuation in Python programs

Some of the following assignments work and some do not. Explain in each case why the assignment works/fails and, if it works, what kind of object `x` refers to and what the value is if we do a `print x`.

```
x = 1
x = 1.
x = 1;
x = 1!
x = 1?
x = 1:
x = 1,
```

Hint. Explore the statements in an interactive Python shell.
Filename: `punctuation.*`.

Exercise 2.24: Investigate a for loop over a changing list

Study the following interactive session and explain in detail what happens in each pass of the loop, and use this explanation to understand the output.

```
>>> numbers = range(10)
>>> print numbers
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> for n in numbers:
...     i = len(numbers)/2
...     del numbers[i]
...     print 'n=%d, del %d' % (n,i), numbers
...
n=0, del 5 [0, 1, 2, 3, 4, 6, 7, 8, 9]
n=1, del 4 [0, 1, 2, 3, 6, 7, 8, 9]
n=2, del 4 [0, 1, 2, 3, 7, 8, 9]
n=3, del 3 [0, 1, 2, 7, 8, 9]
n=8, del 3 [0, 1, 2, 8, 9]
```

Warning

The message in this exercise is to *never modify a list that we are looping over*. Modification is indeed technically possible, as shown above, but you really need to know what you are doing. Otherwise you will experience very strange program behavior.