This chapter shows 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.

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

Since we know how to evaluate the formula (1.2) for one value of $C$, we can just repeat these statements as many times as required 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
Loops and Lists

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 ≤ 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 hyphens above and below the table. We postpone to nicely format the C and F columns of numbers and perform for simplicity a plain print C, F statement inside the loop.

Using a mathematical type of notation, we could write the while loop as follows:
2.1 While Loops

\[ C = -20 \]

while \( C \leq 40 \) repeat the following:

\[ F = \frac{9}{5}C + 32 \]

print \( C \), \( F \)

set \( C \) to \( C + 5 \)

The three lines after the “while” line are to be repeated as long as the condition \( C \leq 40 \) is true. This algorithm will then produce a table of \( C \) and corresponding \( F \) values.

A complete Python program, implementing the repetition algorithm above, looks quite similar:

```python
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 state-
ments. That is, we compute $F$ corresponding to the current $C$ value, print the temperatures, and increment $C$ by $dC$.

Thereafter, we enter the second pass in the loop. First we check the condition: $C$ is $-15$ and $C \leq 40$ 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, $C \leq 40$, 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 a 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.

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 $C \leq 40$, 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
```

If you did not get the last point here, just relax and continue reading.
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.

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 to do Exercises 2.29 and 2.25.

---

3 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.

4 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. See Exercise 6.24 for more details.
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:

\[
\sin(x) \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots,
\]

(2.1)

where \(3! = 3 \cdot 2 \cdot 1\), \(5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1\), etc., are factorials. The expression \(k! = k(k-1)(k-2)\cdots2\cdot1\) can be computed 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 \(\text{sign}\) 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:

```python
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 \( \text{sign} = -1.0 \), \( k = 3 \), \( \text{term} = -1.0 \times x^{**3}/(3 \times 2 \times 1) \) (note that \( \text{sign} \) 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 \( \text{term} = 1.0 \times x^{**5}/\text{math.factorial}(5) \), 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 \texttt{print} statement (which has the same indentation as the \texttt{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 Chapter 2.1.2 could be conveniently stored together as a group. A Python \texttt{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 \texttt{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](image_url) Illustration of two variables: \texttt{var1} refers to an \texttt{int} object with value 21, created by the statement \texttt{var1 = 21}, and \texttt{var2} refers to a \texttt{list} object with value [20, 21, 29, 4.0], i.e., three \texttt{int} objects and one \texttt{float} object, created by the statement \texttt{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:

```python
>>> 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:

```python
>>> 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\(^5\), but 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.

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

\(^5\) Every object in Python and everything you can do with them is defined by programs made by humans. With the techniques of 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 easily define your own list objects if you are not satisfied with Python’s own lists.
2.2 Lists

```python
>>> 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:

```python
>>> 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):

```python
>>> 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`:

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

Python allows negative indices, which “count 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.

```python
>>> 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 Chapter 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:

```python
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:

```python
>>> 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 require 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:

```python
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 which 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 2.30 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:

```python
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 on page 52). 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:

```python
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
```
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

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

can be transformed to this `while` loop:

```python
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:

```python
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:

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

A for loop over integers are written as

\begin{verbatim}
for i in range(start, stop, step):
  ...
\end{verbatim}

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

\begin{verbatim}
Cdegrees = []
for C in range(-20, 45, 5):
  Cdegrees.append(C)
\end{verbatim}

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, -15, \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 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:

\begin{verbatim}
Cdegrees = []
for i in range(0, 21):
  C = -10 + i*2.5
  Cdegrees.append(C)
\end{verbatim}

### 2.3.3 For Loops with List Indices

Instead of iterating over a list directly with the construction

\begin{verbatim}
for element in somelist:
  ...
\end{verbatim}

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

\begin{verbatim}
for i in range(len(somelist)):
  element = somelist[i]
  ...
\end{verbatim}

Since \texttt{len(somelist)} returns the length of \texttt{somelist} and the largest legal index is \texttt{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:

```python
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

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

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

```python
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
```
2.3 Alternative Implementations with Lists and Loops

```python
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

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

but this loop leaves `Cdegrees` unchanged, while

```python
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

```python
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

```python
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:

```python
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
New list = [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 Nested 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\(^6\):

\[
C\text{degrees} = \text{range}(-20, 41, 5) \quad # \text{ -20, -15, \ldots, 35, 40} \\
F\text{degrees} = [(9.0/5)*C + 32 \text{ for } C \text{ in } C\text{degrees}] \\
\text{table} = [C\text{degrees}, F\text{degrees}]
\]

With the subscript \( \text{table}[0] \) we can access the first element (the \( C\text{degrees} \) list), and with \( \text{table}[0][2] \) we reach the third element in the list that constitutes the first element in \( \text{table} \) (this is the same as \( C\text{degrees}[2] \)).

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 \( \text{table} \) on this form, we must construct \( \text{table} \) 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:

\[\text{table1} = \text{table2} = \begin{array}{c}
0 & 0 & 20 \\
1 & 1 & 25 \\
2 & 2 & 30 \\
3 & 3 & 35 \\
4 & 4 & 40 \\
\end{array}
\]

\[\begin{array}{c}
0 & 0 & 68.0 \\
1 & 1 & 77.0 \\
2 & 2 & 86.0 \\
3 & 3 & 95.0 \\
4 & 4 & 104.0 \\
\end{array}
\]

(a) \hspace{2cm} (b)

**Fig. 2.2** Two ways of creating a table as a nested list: (a) table of columns \( C \) and \( F \) (\( C \) and \( F \) are lists); (b) table of rows (\( [C, F] \) lists of two floats).

\(^6\) Any value in \([41, 45]\) can be used as second argument (stop value) to \texttt{range} and will ensure that 40 is included in the range of generate numbers.
We may shorten this code segment by introducing a list comprehension:

```python
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 the “main list”, while the second index looks up the “sublist”.

### 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

```python
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:

```python
>>> 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:

```python
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:

```python
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[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:

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

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

```python
>>> 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:

```python
>>> 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.,

```python
>>> 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:

```python
>>> table[4:6][0:2]
[[0, 32.0], [5, 41.0]]
```

Observe that table[4:6] 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:

```python
>>> 11 = [1, 4, 3]
>>> 12 = 11[:]
>>> 12
[1, 4]
>>> 11[0] = 100 # 11 is modified
>>> 11
[100, 4, 3]
>>> 12 # 12 is not modified
[1, 4]
```

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

We have seen that traversing the nested list \texttt{table} could be done by a loop of the form

\begin{verbatim}
for C, F in table:
    # process C and F
\end{verbatim}

This is natural code when we know that \texttt{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 \texttt{scores} to record the scores of players in a game: \texttt{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 \( \text{scores}[i] \) depends on \( i \). Some code may help to make this clearer:

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

The list \( \text{scores} \) has three elements, each element corresponding to a player. The element no. \( g \) in the list \( \text{scores}[p] \) corresponds to the score obtained in game number \( g \) played by player number \( p \). The length of the lists \( \text{scores}[p] \) 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 \( \text{scores} \) potentially a big nested list. How can we traverse the \( \text{scores} \) 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 \( \text{scores} \) and one for the elements in the sublists of \( \text{scores} \). 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:

```python
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 \( g \) 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 \( \text{scores} \) list initialized above).

The alternative version where we use variables for iterating over the elements in the \( \text{scores} \) list and its sublists looks like this:

```python
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 \( \text{player} \) and \( \text{game} \) 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

```python
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

```python
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 2.31 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:

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

One can also drop the parentheses in many occasions:

```python
>>> 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:

```python
>>> t = t + (-1.0, -2.0) # add two tuples
>>> t
(2, 4, 6, 'temp.pdf', -1.0, -2.0)
>>> t[1] # indexing
4
```
Any list operation that changes the list will not work for tuples:

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

```python
>>> t.append(0)
...  
AttributeError: 'tuple' object has no attribute 'append'
```

```python
>>> 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 Chapter 6.2): 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`:

```python
>>> 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.
A list element can be any Python object, including numbers, strings, functions, and other lists, for instance. Table 2.1 shows some important list operations (only a subset of these are explained in the present chapter).

Table 2.1 Summary of important functionality for list objects.

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

**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:

```python
>>> nl = [[0, 0, 1], [-1, -1, 2], [-10, 10, 5]]
>>> nl[0]
[0, 0, 1]
>>> nl[-1]
[-10, 10, 5]
>>> nl[0][2]
1
>>> 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. In Table 2.1, all operations can be applied to a tuple \( a \), except those involving \texttt{append}, \texttt{del}, \texttt{remove}, \texttt{index}, and \texttt{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 \textit{sequence} in Python. Lists, tuples, strings, and arrays (Chapter 5) are examples on sequences. You choose a sequence type when there is a natural ordering of elements. For a collection of unordered objects a \textit{dictionary} (introduced in Chapter 6.2) is often more convenient.

For Loops. A \texttt{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 \texttt{print} statement prevents the newline character which \texttt{print} otherwise adds to the character.

The \texttt{range} function is frequently used in \texttt{for} loops over a sequence of integers. Recall that \texttt{range(start, stop, inc)} does not include the “end value” \texttt{stop} in the list.

```
>>> 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 \texttt{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 \( a \), \texttt{print} \( a \) can be used, but the \texttt{pprint} and \texttt{scitools.pprint2} modules and their \texttt{pprint} function give a nicer layout of the output for long and nested lists. The \texttt{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,
- while loop,
- for loop,
- list comprehension,
- boolean expression.

2.6.2 Summarizing Example: Analyzing List Data

Problem. The file src/misc/Oxford_sun_hours.txt 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:\footnote{Actually, the data are taken from a web page as explained in Chapter 6.4.3 and easily written out in the list format shown here.}

\[
\begin{bmatrix}
[43.8, 60.5, 190.2, \ldots], \\
[49.9, 54.3, 109.7, \ldots], \\
[63.7, 72.6, 142.3, \ldots], \\
\end{bmatrix}
\]

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.

1. Compute the average number of sun hours for each month during the total data period (1929–2009).
2. Which month has the best weather according to the means found in the preceding task?
3. For each decade, 1930-1939, 1940-1949, \ldots, 2000-2009, compute the average number of sun hours per day in January and December. For example, use December 1949, January 1950, \ldots, 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):
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 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:

```python
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
    n = 2010 - 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`:

```python
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 \texttt{max\_value} of a list like \texttt{monthly\_mean} is simply obtained by \texttt{max(monthly\_mean)}. The corresponding index, needed to find the right name of the corresponding month, is found from \texttt{monthly\_mean.index(max\_value)}. The code for task 2 is then

\[
\begin{align*}
\texttt{max\_value} &= \texttt{max(monthly\_mean)} \\
\texttt{month} &= \texttt{month\_names[monthly\_mean.index(max\_value)]} \\
\texttt{print } &\texttt{’%s has best weather with %.1f sun hours on average’ % } \\
&\texttt{(month, max\_value)}
\end{align*}
\]

(Instead of using the Python’s \texttt{max} and \texttt{index} functionality, we could code everything ourselves to get some training, see Exercise 3.18 for ideas.)

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:

\[
\begin{align*}
\texttt{decade\_mean} &= [] \\
\texttt{for \ decade\_start in range(1930, 2010, 10):} \\
&\quad \texttt{Jan\_index = 0; Dec\_index = 11 \ # \ indices} \\
&\quad \texttt{s = 0} \\
&\quad \texttt{for \ year \ in \ range(decade\_start, decade\_start+10):} \\
&\quad &\quad \texttt{y = year - 1929 \ # \ list \ index} \\
&\quad &\quad \texttt{print \ data[y-1][Dec\_index] + data[y][Jan\_index]} \\
&\quad &\quad \texttt{s += data[y-1][Dec\_index] + data[y][Jan\_index]} \\
&\quad &\quad \texttt{decade\_mean.append(s/(20.*30))} \\
&\quad \texttt{for \ i \ in \ range(len(decade\_mean)):} \\
&\quad &\quad \texttt{print ‘Decade %d-%d: %.1f’ % } \\
&\quad &\quad &\texttt{(1930+i*10, 1939+i*10, decade\_mean[i])}
\end{align*}
\]

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 \texttt{sun\_data.py}.
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: docs.python.org. Here you can find a Python tutorial, the very useful Python Library Reference, an index of all modules that come with the basic Python distribution, and a Language Reference, to mention some key documents. You should in particular discover the index of the Python Library Reference. When you wonder what functions you can find in a module, say the math module, you should go to this index, find the “math” keyword, and press the link. This brings you right to the official documentation of the math module. 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.

A word of caution is probably necessary here. Reference manuals, such as the Python Library Reference, 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 Library Reference is the pydoc program. In a terminal window you write

```
Unix/DOS> pydoc math
```

In Python there are two 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 Python Library Reference.

---

8 Any command you can run in the terminal window can also be run inside IPython if you start the command with an exclamation mark.
There are also numerous books about Python. Beazley [1] is an excellent reference that improves and extends the information in the Python Library Reference. The “Learning Python” book [8] has been very popular for many years as an introduction to the language. There is a special web page http://wiki.python.org/moin/PythonBooks listing most Python books on the market. A comprehensive book on the use of Python for doing scientific research is [5].

Quick references, which list “all” Python functionality in compact tabular form, are very handy. We recommend in particular the one by Richard Gruet: http://rgruet.free.fr/PQR26/PQR2.6.html.

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 program that prints out a table with Fahrenheit degrees 0, 10, 20, . . . , 100 in the first column and the corresponding Celsius degrees in the second column. Hint: Modify the c2f_table_while.py program from Chapter 2.1.2. Name of program file: f2c_table_while.py.

⋄

Exercise 2.2. Write an approximate Fahrenheit–Celsius conversion table.

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

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

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

⋄

Exercise 2.3. 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 $n-1$.) Name of program file: odd.py.

⋄

Exercise 2.4. Store odd numbers in a list.

Modify the program from Exercise 2.3 to store the generated odd numbers in a list. Start with an empty list and use a while loop where you in each pass of the loop append a new element to the list. Finally, print the list elements to the screen. Name of program file: odd_list1.py.

⋄
Exercise 2.5. Generate odd numbers by a list comprehension.
Solve Exercise 2.4 using a list comprehension (with for and range). Name of program file: odd_list2.py.

Exercise 2.6. Make a table of function values.
Write a program that prints a table with \( t \) values in the first column and the corresponding \( y(t) = v_0 t - 0.5gt^2 \) values in the second column. Use \( n \) uniformly spaced \( t \) values throughout the interval \([0, 2v_0/g]\). Set \( v_0 = 1 \), \( g = 9.81 \), and \( n = 11 \). Name of program file: ball_table1.py.

Exercise 2.7. Store numbers in lists.
Modify the program from Exercise 2.6 so that the \( t \) and \( y \) values are stored in two lists \( t \) and \( y \). Thereafter, traverse the lists with a for loop and write out a nicely formatted table of \( t \) and \( y \) values (using either a zip or range construction). Name of program file: ball_table2.py.

Exercise 2.8. 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 whole new list. Name of program file: primes.py.

Exercise 2.9. Simulate operations on lists by hand.
You are given the following program:

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

Explain what is printed by each print statement.

Exercise 2.10. Generate equally spaced coordinates.
We want to generate \( x \) coordinates between 1 and 2 with spacing 0.01. The coordinates are given by the formula \( x_i = 1 + ih \), where \( h = 0.01 \) and \( i \) runs over integers \( 0, 1, \ldots, 100 \). Compute the \( x_i \) values and store them in a list. Use a for loop, and append each new \( x_i \) value to a list, which is empty initially. Name of program file: coor1.py.

Exercise 2.11. Use a list comprehension to solve Exer. 2.10.
The problem is the same as in Exercise 2.10, but now we want the \( x_i \) values to be stored in a list using a list comprehension construct (see Chapter 2.3.5). Name of program file: coor2.py.

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

```python
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 a program.) Write a correct program. Name of program file: `sum_while.py`.

There are two basic ways to find errors in a program: (i) read the program carefully and think about the consequences of each statement, and (ii) print out intermediate results and compare with hand calculations. First, try method (i) and find as many errors as you can. Then, try method (ii) for $M = 3$ and compare the evolution of $s$ with your own hand calculations.

Exercise 2.13. Use a for loop in Exer. 2.12.

Rewrite the corrected version of the program in Exercise 2.12 using a `for` loop over $k$ values instead of a `while` loop. Name of program file: `sum_for.py`.


The program in Exercise 2.13 can be greatly condensed by applying the `sum` function to a list of all the elements $1/k$ in the sum $\sum_{k=1}^{M} \frac{1}{k}$:

```python
print sum([1.0/k for k in range(1, M+1, 1)])
```

The list comprehension here first builds a list of all elements in the sum, and this may consume a lot of memory in the computer. Python offers an alternative syntax

```python
print sum(1.0/k for k in xrange(1, M+1, 1))
```

where we get rid of the list produced by a list comprehension. We also get rid of the list returned by `range`, because `xrange` generates a sequence of the same integers as `range`, but the integers are not stored in a list (they are generated as they are needed, and for very large lists, `xrange` is therefore more efficient than `range`).

The purpose of this exercise is to compare the efficiency of the two calls to `sum` as listed above. Use the `time` module from Appendix G.6.1 to measure the CPU time spent by each construction. Write out $M$ and the CPU time. Set $M = 10^6$ or high enough to cause an execution time of a few seconds. Name of program file: `sum_compact.py`.  

Exercise 2.15. *Compute a polynomial via a product.*

Given \( n + 1 \) roots \( r_0, r_1, \ldots, r_n \) of a polynomial \( p(x) \) of degree \( n + 1 \), \( p(x) \) can be computed by

\[
p(x) = \prod_{i=0}^{n} (x - r_i) = (x - r_0)(x - r_1) \cdots (x - r_{n-1})(x - r_n).
\]

(2.3)

Store the roots \( r_0, \ldots, r_n \) in a list and make a loop that computes the product in (2.3). Test the program on a polynomial with roots \(-1, 1, \) and \(2\). Name of program file: `polynomial_prod.py`

Exercise 2.16. *Simulate a program by hand.*

Consider the following program for computing with interest rates:

```python
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) Explain with words what type of mathematical problem that is solved by this program. Compare this computerized solution with the technique your high school math teacher would prefer.

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

(c) Change the value of `p` to 5. Why will the loop now run forever? (See Exercise 2.12 for how to stop the program if you try to run it.) Make the program more robust against such errors.

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

Insert the text for the answers to (a) and (b) in a multi-line string in the program file. Name of program file: `interest_rate_loop.py`.

Exercise 2.17. *Explore the Python Library Reference.*

Suppose you want to compute the inverse sine function: \( \sin^{-1} x \). The `math` module has a function for computing \( \sin^{-1} x \), but what is the right name of this function? Read Chapter 2.6.3 and use the `math` entry in the index of the Python Library Reference to find out how to compute \( \sin^{-1} x \). Make a program where you compute \( \sin^{-1} x \) for \( n \) \( x \) values uniformly distributed between 0 and 1, and write out the results in a nicely formatted table. For each \( x \) value, check that the sine of \( \sin^{-1} x \) equals \( x \). Name of program file: `inverse_sine.py`.

Exercise 2.18. *Implement the sum function.*

The standard Python function called `sum` takes a list as argument and computes the sum of the elements in the list:
Implement your own version of \texttt{sum}. Name of program: \texttt{sum.py}.  

**Exercise 2.19. Index a nested lists.**

We define the following nested list:

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

Index this list to extract 1) the letter \texttt{a}; 2) the list ['d', 'e', 'f']; 3) the last element \texttt{h}; 4) the \texttt{d} element. Explain why \texttt{q[-1][-2]} has the value \texttt{g}. Name of program file: \texttt{index_nested_list.py}.

**Exercise 2.20. Construct a double for loop over a nested list.**

Consider the list from Exercise 2.19. We can visit all elements of \texttt{q} using this nested \texttt{for} loop:

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

What type of objects are \texttt{i} and \texttt{j}? Name of program file: \texttt{nested_list_iter.py}.

**Exercise 2.21. Store data in lists in Exercise 2.2.**

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

**Exercise 2.22. Store data from Exer. 2.7 in a nested list.**

After having computed the two lists of \texttt{t} and \texttt{y} values in the program from Exercise 2.7, store the two lists in a new list \texttt{ty1}. Write out a table of \texttt{t} and \texttt{y} values by traversing the data in the \texttt{ty1} list. Thereafter, make a list \texttt{ty2} which holds each row in the table of \texttt{t} and \texttt{y} values (\texttt{ty1} is a list of table columns while \texttt{ty2} is a list of table rows, as explained in Chapter 2.4). Write out the table by traversing the \texttt{ty2} list. Name of program file: \texttt{ball_table3.py}.

**Exercise 2.23. Convert nested list comprehensions to nested standard loops.**

Rewrite the generation of the nested list \texttt{q},

```python
q = [r**2 for r in [10**i for i in range(5)]]
```

by using standard \texttt{for} loops instead of list comprehensions. Name of program file: \texttt{listcomp2for.py}.
Exercise 2.24. Demonstrate list functionality.
Create an interactive session where you demonstrate the effect of each of the operations in Table 2.1 on page 75. Use IPython and log the results (see Exercise 1.12). Name of program file: list_demo.py.

Exercise 2.25. 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 2.26. 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 the n value and printing out r in both for loops over i. Can you now explain why we come back to 1 and not 2? Name of program file: repeated_sqrt.py.

Exercise 2.27. Explore what zero can be on a computer.
Type in the following code and run it:
2.7 Exercises

```python
es = 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\(^9\) were zero? Name of program file: `machine_zero.py`.

If somebody shows you this interactive session

```python
>>> 0.5 + 1.45E-22
0.5
```

and claims that Python cannot add numbers correctly, what is your answer? ◊

**Exercise 2.28.** *Compare two real numbers on a computer.*

Consider the following simple program inspired by Chapter 1.4.3:

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

Try to run this example!

One should never compare two floating-point objects directly using `==` or `!=`, because round-off errors quickly make two identical mathematical values different on a computer. A better result is to test if $|a - b|$ is sufficiently small, i.e., if $a$ and $b$ are “close enough” to be considered equal. Modify the test according to this idea.

Thereafter, read the documentation of the function `float_eq` from SciTools: `scitools.numpyutils.float_eq` (see page 80 for how to bring up the documentation of a module or a function in a module). Use this function to check whether two real numbers are equal within a tolerance. Name of program file: `compare_float.py`. ◊

**Exercise 2.29.** *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 “sleep” $n$ seconds and is handy to insert a pause. Use this information to explain what the following code does:

---

\(^9\) This nonzero eps value 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.
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 < with > in the loop condition. Explain what happens now. Name of program: time_while.py.

Exercise 2.30. Explore problems with inaccurate indentation.

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

```python
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 2.12 for how to stop a hanging program.)

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 2.31. Simulate nested loops by hand.

Go through the code below by hand, statement by statement, and calculate the numbers that will be printed.

```python
n = 3
for i in range(-1, n):
    if i != 0:
        print i
for i in range(1, 13, 2*n):
    for j in range(n):
        print i, j
for i in range(1, n+1):
    for j in range(i):
        if j:
            print i, j
for i in range(1, 13, 2*n):
    for j in range(0, i, 2):
        for k in range(2, j, 1):
            b = i > j > k
            if b:
                print i, j, k
```

You may use a debugger, see Appendix F.1, to step through the code to see what happens.
Exercise 2.32. 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`.

\[
\begin{align*}
\text{x = 1} \\
\text{x = 1.} \\
\text{x = 1;} \\
\text{x = 1!} \\
\text{x = 1?} \\
\text{x = 1:} \\
\text{x = 1,}
\end{align*}
\]

Hint: Explore the statements in an interactive Python shell.

Exercise 2.33. 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.

```python
>>> 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]
```

The message in this exercise is to *never modify a list that is used in a for loop.* Modification is indeed technically possible, as we show above, but you really need to know what you are doing – to avoid getting frustrated by strange program behavior.