

This chapter introduces some fundamental topics in programming: list objects, `while` and `for` loops, `if-else` branches, and user-defined functions. Everything covered here will be essential for programming in general - and of course in the rest of the book. The programs associated with the chapter are found in the folder `src/basic`.

2.1 Loops and Lists for Tabular Data

The goal of our next programming example 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:

-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

```

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 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:

```

C = -20
while C ≤ 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 similar¹:

```

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 $C \leq 40$. The first time C is -20 , which implies that $C \leq 40$ (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-

¹ For this table we also add (of teaching purposes) a line above and below the table.

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 object².

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 example regarding 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
```

² 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`, say this is `True`, and then `not` turns the value into `False`. On the opposite, if `C == 40` is `False`, `not C == 40` becomes `True`. 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`³

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 common⁴ 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 2.54 and 2.47.

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 6.27 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 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 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. Python has great functionality for examining and manipulating such sequences of objects, which will be demonstrated below.

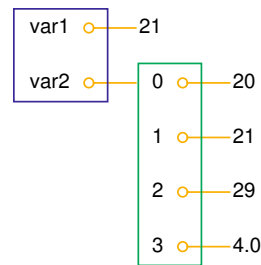


Fig. 2.1 Illustration of two variables: `var1` refers to an `int` object with value 21, created by the statement `var1 = 21`, and `var2` refers to a list object with value `[20, 21, 29, 4.0]`, i.e., three `int` objects and one `float` object, created by the statement `var2 = [20, 21, 29, 4.0]`.

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 define⁵, 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()`):

```
>>> 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):

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.

```
>>> 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 “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.

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

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.

2.1.5 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 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 2.55 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 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:

```
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
```

2.1.6 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 `c2f_table_lists.py`.

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
```

Storing the Table Columns as Lists. A slight change of the previous program could be to store both the Celsius and Fahrenheit degrees in lists. For the Fahrenheit numbers we may start with an empty list `Fdegrees` and use `append` to add list elements inside the loop:

```
Cdegrees = [-20, -15, -10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40]
Fdegrees = [] # start with empty list
for C in Cdegrees:
    F = (9.0/5)*C + 32
    Fdegrees.append(F)
```

If we now print `Fdegrees` we get

```
[-4.0, 5.0, 14.0, 23.0, 32.0, 41.0, 50.0, 59.0,
 68.0, 77.0, 86.0, 95.0, 104.0]
```

Loops with List Indices. Instead of having a `for` loop over the list elements we may use a `for` loop over the list indices. The indices are integers going from 0 up to the length of the list minus one. Python has a `range` function returning such a list of integers:

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

All legal indices of a list `a` are obtained by calling `range(len(a))`.

The previous `for` loop can alternatively make use of the `range` function and loops over list indices:

```
Cdegrees = range(-20, 45, 5) # generate C values
Fdegrees = [0.0]*len(Cdegrees) # list of 0.0 values
for i in range(len(Cdegrees)):
    Fdegrees[i] = (9.0/5)*Cdegrees[i] + 32
```

Observe that we need to initialize `Fdegrees` to be a list of length `len(Cdegrees)` (setting each element to zero for convenience). If we fail to let `Fdegrees` have the same length as `Cdegrees`, and set `Fdegrees`

= [] instead, `Fdegrees[i]` will lead to an error messages saying that the index `i` is out of range. (Even index 0, referring to the first element, is out of range if `Fdegrees` is an empty list.)

Loops over Real Numbers. So far, the data in `Cdegrees` have been integers. To make real numbers in the `Cdegrees` list, we cannot simply call the `range` function since it only generates integers. A loop is necessary for generating real numbers:

```
C_step = 0.5
C_start = -5
n = 16
Cdegrees = [0.0]*n; Fdegrees = [0.0]*n
for i in range(n):
    Cdegrees[i] = C_start + i*C_step
    Fdegrees[i] = (9.0/5)*Cdegrees[i] + 32
```

A while loop with growing lists can also be used if we specify a stop value for `C`:

```
C_start = -5; C_step = 0.5; C_stop = 20
C = C_start
Cdegrees = []; F_degrees = []
while C <= C_stop:
    Cdegrees.append(C)
    F = (9.0/5)*C + 32
    Fdegrees.append(F)
    C = C + C_step
```

About Changing a List. 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.

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.

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.

2.1.7 Nested Lists

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 numbers⁶:

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

table = [Cdegrees, Fdegrees]
```

With the subscript `table[0]` we can access the first element (the `Cdegrees` list), and with `table[0][2]` we reach the third element in the list that constitutes the first element in `table` (this is the same as `Cdegrees[2]`).

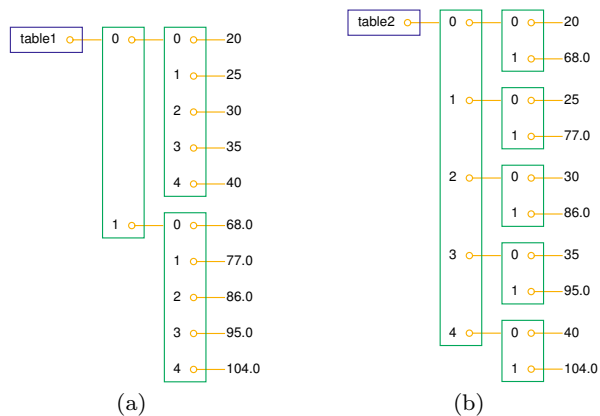


Fig. 2.2 Two ways of creating a table as a nested list: (a) table of columns `C` and `F` (`C` and `F` are lists); (b) table of rows (`[C, F]` lists of two floats).

⁶ Any value in `[41, 45]` can be used as second argument (stop value) to `range` and will ensure that 40 is included in the range of generate numbers.

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

2.1.8 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)
```

2.1.9 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:

```
>>> 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]]
```

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
[100, 4, 3]          # l1 is modified
>>> l2
[1, 4]                # l2 is not modified
```

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 (cf. Chapter 2.1.7) 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, ..., 10 in `table`.

2.1.10 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 2.58 to get a better understanding of nested for loops.

2.1.11 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 Chapter 6.2): tuples can be used as keys in dictionaries while lists can not.

2.2 Functions

In a computer language like Python, the term *function* means more than just a mathematical function. A function is a collection of statements that you can execute wherever and whenever you want in the program. You may send variables to the function to influence what is getting computed by statements in the function, and the function may return new objects. In particular, functions help to avoid duplicating code snippets by putting all similar snippets in a common place. This strategy saves typing and makes it easier to change the program later. Functions are also often used to just split a long program into smaller, more manageable pieces, so the program and your own thinking about it become clearer. Python comes with lots of functions (`math.sqrt`, `range`, and `len` are examples we have met so far). This section outlines how you can define your own functions.

2.2.1 Functions of One Variable

Let us start with making a Python function that evaluates a mathematical function, more precisely the function $F(C)$ defined in (1.2): $F(C) = \frac{9}{5}C + 32$. The corresponding Python function must take C as argument and return the value $F(C)$. The code for this looks like

```
def F(C):
    return (9.0/5)*C + 32
```

All Python functions begin with `def`, followed by the function name, and then inside parentheses a comma-separated list of *function arguments*. Here we have only one argument `C`. This argument acts as a standard variable inside the function. The statements to be performed inside the function must be indented. At the end of a function it is common to *return* a value, that is, send a value “out of the function”. This value is normally associated with the name of the function, as in the present case where the returned value is $F(C)$.

The `def` line with the function name and arguments is often referred to as the *function header*, while the indented statements constitute the *function body*.

To use a function, we must *call*⁷ it. Because the function returns a value, we need to store this value in a variable or make use of it in other ways. Here are some calls to `F`:

```
a = 10
F1 = F(a)
temp = F(15.5)
print F(a+1)
sum_temp = F(10) + F(20)
```

The returned object from `F(C)` is in our case a `float` object. The call `F(C)` can therefore be placed anywhere in a code where a `float` object would be valid. The `print` statement above is one example. As another example, say we have a list `Cdegrees` of Celsius degrees and we want to compute a list of the corresponding Fahrenheit degrees using the `F` function above in a list comprehension:

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

As an example of a slight variation of our `F(C)` function, we may return a formatted string instead of a real number:

```
>>> def F2(C):
...     F_value = (9.0/5)*C + 32
...     return '%.1f degrees Celsius corresponds to '\
...             '%.1f degrees Fahrenheit' % (C, F_value)
...
>>> s1 = F2(21)
>>> s1
'21.0 degrees Celsius corresponds to 69.8 degrees Fahrenheit'
```

The assignment to `F_value` demonstrates that we can create variables inside a function as needed.

⁷ Sometimes the word *invoke* is used as an alternative to *call*.

2.2.2 Local and Global Variables

Let us reconsider the `F2(C)` function from the previous section. The variable `F_value` is a *local* variable in the function, and a local variable does not exist outside the function. We can easily demonstrate this fact by continuing the previous interactive session:

```
>>> c1 = 37.5
>>> s2 = F2(c1)
>>> F_value
...
NameError: name 'F_value' is not defined
```

The surrounding program outside the function is not aware of `F_value`. Also the argument to the function, `C`, is a local variable that we cannot access outside the function:

```
>>> C
...
NameError: name 'C' is not defined
```

On the contrary, the variables defined outside of the function, like `s1`, `s2`, and `c1` in the above session, are *global* variables. These can be accessed everywhere in a program.

Local variables are created inside a function and destroyed when we leave the function. To learn more about this fact, we may study the following session where we write out `F_value`, `C`, and some global variable `r` inside the function:

```
>>> def F3(C):
...     F_value = (9.0/5)*C + 32
...     print 'Inside F3: C=%s F_value=%s r=%s' % (C, F_value, r)
...     return '%.1f degrees Celsius corresponds to '\
...           '%.1f degrees Fahrenheit' % (C, F_value)
...
>>> C = 60      # make a global variable C
>>> r = 21      # another global variable
>>> s3 = F3(r)
Inside F3: C=21 F_value=69.8 r=21
>>> s3
'21.0 degrees Celsius corresponds to 69.8 degrees Fahrenheit'
>>> C
60
```

This example illustrates that there are two `C` variables, one global, defined in the main program with the value 60 (an `int` object), and one local, living when the program flow is inside the `F3` function. The value of this `C` is given in the call to the `F3` function (also an `int` object in this case). Inside the `F3` function the local `C` “hides” the global `C` variable in the sense that when we refer to `C` we access the local variable⁸.

The more general rule, when you have several variables with the same name, is that Python first tries to look up the variable name

⁸ The global `C` can technically be accessed as `globals()['C']`, but one should avoid working with local and global variables with the same names at the same time!

among the local variables, then there is a search among global variables, and finally among built-in Python functions. Here is a complete sample program with several versions of a variable `sum` which aims to illustrate this rule:

```
print sum # sum is a built-in Python function
sum = 500 # rebind the name sum to an int
print sum # sum is a global variable

def myfunc(n):
    sum = n + 1
    print sum # sum is a local variable
    return sum

sum = myfunc(2) + 1 # new value in global variable sum
print sum
```

In the first line, there are no local variables, so Python searches for a global value with name `sum`, but cannot find any, so the search proceeds with the built-in functions, and among them Python finds a function with name `sum`. The printout of `sum` becomes something like `<built-in function sum>`.

The second line rebinds the global name `sum` to an `int` object. When trying to access `sum` in the next `print` statement, Python searches among the global variables (no local variables so far) and finds one. The printout becomes 500. The call `myfunc(2)` invokes a function where `sum` is a local variable. Doing a `print sum` in this function makes Python first search among the local variables, and since `sum` is found there, the printout becomes 3 (and not 500, the value of the global variable `sum`). The value of the local variable `sum` is returned, added to 1, to form an `int` object with value 4. This `int` object is then bound to the global variable `sum`. The final `print sum` leads to a search among global variables, and we find one with value 4.

The values of global variables can be accessed inside functions, but the values cannot be changed unless the variable is declared as `global`:

```
a = 20; b = -2.5 # global variables

def f1(x):
    a = 21 # this is a new local variable
    return a*x + b # 21*x - 2.5

print a # yields 20

def f2(x):
    global a
    a = 21 # the global a is changed
    return a*x + b # 21*x - 2.5

f1(3); print a # 20 is printed
f2(3); print a # 21 is printed
```

Note that in the `f1` function, `a = 21` creates a local variable `a`. As a programmer you may think you change the global `a`, but it does not

happen! Normally, this feature is advantageous because changing global variables often leads to errors in programs.

2.2.3 Multiple Arguments

The previous $F(C)$ and $F2(C)$ functions are functions of one variable, C , or as we phrase it in computer science: the functions take one argument (C). Functions can have as many arguments as desired; just separate the argument names by commas.

Consider the function $y(t)$ in (1.1). Here is a possible Python function taking two arguments:

```
def yfunc(t, v0):  
    g = 9.81  
    return v0*t - 0.5*g*t**2
```

Note that g is a local variable with a fixed value, while t and $v0$ are arguments and therefore also local variables. Examples on valid calls are

```
y = yfunc(0.1, 6)  
y = yfunc(0.1, v0=6)  
y = yfunc(t=0.1, v0=6)  
y = yfunc(v0=6, t=0.1)
```

The possibility to write `argument=value` in the call makes it easier to read and understand the call statement. With the `argument=value` syntax for all arguments, the sequence of the arguments does not matter in the call, which here means that we may put $v0$ before t . When omitting the `argument=` part, the sequence of arguments in the call must perfectly match the sequence of arguments in the function definition. The `argument=value` arguments must appear after all the arguments where only `value` is provided (e.g., `yfunc(t=0.1, 6)` is illegal).

Whether we write `yfunc(0.1, 6)` or `yfunc(v0=6, t=0.1)`, the arguments are initialized as local variables in the function in the same way as when we assign values to variables:

```
t = 0.1  
v0 = 6
```

These statements are not visible in the code, but a call to a function automatically initializes the arguments in this way.

Some may argue that `yfunc` should be a function of t only, because we mathematically think of y as a function of t and write $y(t)$. This is easy to reflect in Python:

```
def yfunc(t):  
    g = 9.81  
    return v0*t - 0.5*g*t**2
```

The main difference is that `v0` now must be a *global* variable, which needs to be initialized before we call `yfunc`. The next session demonstrates what happens if we fail to initialize such a global variable:

```
>>> def yfunc(t):
...     g = 9.81
...     return v0*t - 0.5*g*t**2
...
>>> yfunc(0.6)
...
NameError: global name 'v0' is not defined
```

The remedy is to define `v0` as a global variable prior to calling `yfunc`:

```
>>> v0 = 5
>>> yfunc(0.6)
1.2342
```

So far our Python functions have typically computed some mathematical function, but the usefulness of Python functions goes far beyond mathematical functions. Any set of statements that we want to repeatedly execute under slightly different circumstances is a candidate for a Python function. Say we want to make a list of numbers starting from some value and stopping at another value, with increments of a given size. With corresponding variables `start=2`, `stop=8`, and `inc=2`, we should produce the numbers 2, 4, 6, and 8. Our tables in this chapter typically needs such functionality for creating a list of C values or a list of t values. Let us therefore write a function doing the task⁹, together with a couple of statements that demonstrate how we call the function:

```
def makelist(start, stop, inc):
    value = start
    result = []
    while value <= stop:
        result.append(value)
        value = value + inc
    return result

mylist = makelist(0, 100, 0.2)
print mylist # will print 0, 0.2, 0.4, 0.6, ... 99.8, 100
```

The `makelist` function has three arguments: `start`, `stop`, and `inc`, which become local variables in the function. Also `value` and `result` are local variables. In the surrounding program we define only one variable, `mylist`, and this is then a global variable.

⁹ You might think that `range(start, stop, inc)` makes the `makelist` function redundant, but `range` can only generate integers, while `makelist` can generate real numbers too – and more, see Exercise 2.40.

2.2.4 Multiple Return Values

Python functions may return more than one value. Suppose we are interested in evaluating both $y(t)$ defined in (1.1) and its derivative

$$\frac{dy}{dt} = v_0 - gt.$$

In the current application, $y'(t)$ has the physical interpretation as the velocity of the ball. To return y and y' we simply separate their corresponding variables by a comma in the `return` statement:

```
def yfunc(t, v0):
    g = 9.81
    y = v0*t - 0.5*g*t**2
    dydt = v0 - g*t
    return y, dydt
```

When we call this latter `yfunc` function, we need two values on the left-hand side of the assignment operator because the function returns two values:

```
position, velocity = yfunc(0.6, 3)
```

Here is an application of the `yfunc` function for producing a nicely formatted table of positions and velocities of a ball thrown up in the air:

```
t_values = [0.05*i for i in range(10)]
for t in t_values:
    pos, vel = yfunc(t, v0=5)
    print 't=%-10g position=%-10g velocity=%-10g' % (t, pos, vel)
```

The format `%-10g` prints a real number as compactly as possible (decimal or scientific notation) in a field of width 10 characters. The minus (“-”) sign after the percentage sign implies that the number is *left-adjusted* in this field, a feature that is important for creating nice-looking columns in the output:

t=0	position=0	velocity=5
t=0.05	position=0.237737	velocity=4.5095
t=0.1	position=0.45095	velocity=4.019
t=0.15	position=0.639638	velocity=3.5285
t=0.2	position=0.8038	velocity=3.038
t=0.25	position=0.943437	velocity=2.5475
t=0.3	position=1.05855	velocity=2.057
t=0.35	position=1.14914	velocity=1.5665
t=0.4	position=1.2152	velocity=1.076
t=0.45	position=1.25674	velocity=0.5855

When a function returns multiple values, separated by a comma in the `return` statement, a tuple (Chapter 2.1.11) is actually returned. We can demonstrate that fact by the following session:

```
>>> def f(x):
...     return x, x**2, x**4
... 
```

```
>>> s = f(2)
>>> s
(2, 4, 16)
>>> type(s)
<type 'tuple'>
>>> x, x2, x4 = f(2)
```

Note that storing multiple return values into separate variables, as we do in the last line, is actually the same functionality as we use for storing list elements in separate variables, see on page 58.

Our next example concerns a function aimed at calculating the sum

$$L(x; n) = \sum_{i=1}^n \frac{1}{i} \left(\frac{x}{1+x} \right)^i. \quad (2.1)$$

It can be shown that $L(x; n)$ is an approximation to $\ln(1+x)$ for a finite n and $x \geq 1$. The approximation becomes exact in the limit:

$$\ln(1+x) = \lim_{n \rightarrow \infty} L(x; n).$$

To compute a sum in a Python program, we use a loop and add terms to a “summing variable” inside the loop. This variable must be initialized to zero outside the loop. For example, we can sketch the implementation of $\sum_{i=1}^n c(i)$, where $c(i)$ is some formula depending on i , as

```
s = 0
for i in range(1, n+1):
    s += c(i)
```

For the specific sum (2.1) we just replace $c(i)$ by the right term $(1/i)(x/(1+x))^i$ inside the `for` loop¹⁰:

```
s = 0
for i in range(1, n+1):
    s += (1.0/i)*(x/(1.0+x))**i
```

It is natural to embed the computation of the sum in a function which takes x and n as arguments and returns the sum:

```
def L(x, n):
    s = 0
    for i in range(1, n+1):
        s += (1.0/i)*(x/(1.0+x))**i
    return s
```

Instead of just returning the value of the sum, we could return additional information on the error involved in the approximation of $\ln(1+x)$ by $L(x; n)$. The first neglected term in the sum provides

¹⁰ Observe the 1.0 numbers: These avoid integer division (`i` is `int` and `x` may be `int`).

an indication of the error¹¹. We could also return the exact error. The new version of the $L(x, n)$ function then looks as this:

```
def L(x, n):
    s = 0
    for i in range(1, n+1):
        s += (1.0/i)*(x/(1.0+x))**i
    value_of_sum = s
    first_neglected_term = (1.0/(n+1))*(x/(1.0+x))**(n+1)
    from math import log
    exact_error = log(1+x) - value_of_sum
    return value_of_sum, first_neglected_term, exact_error

# typical call:
value, approximate_error, exact_error = L2(x, 100)
```

The next section demonstrates the usage of the L function to judge the quality of the approximation $L(x; n)$ to $\ln(1 + x)$.

2.2.5 Functions with No Return Values

Sometimes a function just performs a set of statements, and it is not natural to return any values to the calling code. In such situations one can simply skip the `return` statement. Some programming languages use the terms *procedure* or *subroutine* for functions that do not return anything.

Let us exemplify a function without return values by making a table of the accuracy of the $L(x; n)$ approximation to $\ln(1 + x)$ from the previous section:

```
def table(x):
    print '\nx=%g, ln(1+x)=%g' % (x, log(1+x))
    for n in [1, 2, 10, 100, 500]:
        value, next, error = L(x, n)
        print 'n=%-4d %-10g (next term: %8.2e '\
              'error: %8.2e)' % (n, value, next, error)
```

This function just performs a set of statements that we may want to run several times. Calling

```
table(10)
table(1000)
```

gives the output :

```
x=10, ln(1+x)=2.3979
n=1    0.909091    (next term: 4.13e-01    error: 1.49e+00)
n=2    1.32231    (next term: 2.50e-01    error: 1.08e+00)
n=10   2.17907    (next term: 3.19e-02    error: 2.19e-01)
n=100  2.39789    (next term: 6.53e-07    error: 6.59e-06)
n=500  2.3979     (next term: 3.65e-24    error: 6.22e-15)
```

¹¹ The size of the terms decreases with increasing n , and the first neglected term is then bigger than all the remaining terms, but not necessarily bigger than their sum. The first neglected term is therefore only an indication of the size of the total error we make.

```

x=1000, ln(1+x)=6.90875
n=1    0.999001    (next term: 4.99e-01    error: 5.91e+00)
n=2    1.498      (next term: 3.32e-01    error: 5.41e+00)
n=10   2.919      (next term: 8.99e-02    error: 3.99e+00)
n=100  5.08989    (next term: 8.95e-03    error: 1.82e+00)
n=500  6.34928    (next term: 1.21e-03    error: 5.59e-01)

```

From this output we see that the sum converges much more slowly when x is large than when x is small. We also see that the error is an order of magnitude or more larger than the first neglected term in the sum. The functions `L` and `table` are found in the file `lnsum.py`.

When there is no explicit `return` statement in a function, Python actually inserts an invisible `return None` statement. `None` is a special object in Python that represents something we might think of as “empty data” or “nothing”. Other computer languages, such as C, C++, and Java, use the word “void” for a similar thing. Normally, one will call the `table` function without assigning the return value to any variable, but if we assign the return value to a variable, `result = table(500)`, `result` will refer to a `None` object.

The `None` value is often used for variables that should exist in a program, but where it is natural to think of the value as conceptually undefined. The standard way to test if an object `obj` is set to `None` or not reads

```

if obj is None:
    ...
if obj is not None:
    ...

```

One can also use `obj == None`. The `is` operator tests if two names refer to the same object, while `==` tests if the contents of two objects are the same:

```

>>> a = 1
>>> b = a
>>> a is b    # a and b refer to the same object
True
>>> c = 1.0
>>> a is c
False
>>> a == c    # a and c are mathematically equal
True

```

2.2.6 Keyword Arguments

Some function arguments can be given a default value so that we may leave out these arguments in the call, if desired. A typical function may look as

```

>>> def somefunc(arg1, arg2, kwarg1=True, kwarg2=0):
>>>     print arg1, arg2, kwarg1, kwarg2

```

The first two arguments, `arg1` and `arg2`, are *ordinary* or *positional* arguments, while the latter two are *keyword arguments* or *named arguments*. Each keyword argument has a name (in this example `kwarg1` and `kwarg2`) and an associated default value. The keyword arguments must always be listed after the positional arguments in the function definition.

When calling `somefunc`, we may leave out some or all of the keyword arguments. Keyword arguments that do not appear in the call get their values from the specified default values. We can demonstrate the effect through some calls:

```
>>> somefunc('Hello', [1,2])
Hello [1, 2] True 0
>>> somefunc('Hello', [1,2], kwarg1='Hi')
Hello [1, 2] Hi 0
>>> somefunc('Hello', [1,2], kwarg2='Hi')
Hello [1, 2] True Hi
>>> somefunc('Hello', [1,2], kwarg2='Hi', kwarg1=6)
Hello [1, 2] 6 Hi
```

The sequence of the keyword arguments does not matter in the call. We may also mix the positional and keyword arguments if we explicitly write `name=value` for all arguments in the call:

```
>>> somefunc(kwarg2='Hello', arg1='Hi', kwarg1=6, arg2=[1,2],)
Hi [1, 2] 6 Hello
```

Example: Function with Default Parameters. Consider a function of t which also contains some parameters, here A , a , and ω :

$$f(t; A, a, \omega) = Ae^{-at} \sin(\omega t). \quad (2.2)$$

We can implement f as a Python function where the independent variable t is an ordinary positional argument, and the parameters A , a , and ω are keyword arguments with suitable default values:

```
from math import pi, exp, sin

def f(t, A=1, a=1, omega=2*pi):
    return A*exp(-a*t)*sin(omega*t)
```

Calling `f` with just the `t` argument specified is possible:

```
v1 = f(0.2)
```

In this case we evaluate the expression $e^{-0.2} \sin(2\pi \cdot 0.2)$. Other possible calls include

```
v2 = f(0.2, omega=1)
v3 = f(1, A=5, omega=pi, a=pi**2)
v4 = f(A=5, a=2, t=0.01, omega=0.1)
v5 = f(0.2, 0.5, 1, 1)
```

You should write down the mathematical expressions that arise from these four calls. Also observe in the third line above that a positional argument, t in that case, can appear in between the keyword arguments if we write the positional argument on the keyword argument form `name=value`. In the last line we demonstrate that keyword arguments can be used as positional argument, i.e., the name part can be skipped, but then the sequence of the keyword arguments in the call must match the sequence in the function definition exactly.

Example: Computing a Sum with Default Tolerance. Consider the $L(x;n)$ sum and the Python implementation $L(x, n)$ from Chapter 2.2.4. Instead of specifying the number of terms in the sum, n , it is better to specify a tolerance ϵ of the accuracy. We can use the first neglected term as an estimate of the accuracy. This means that we sum up terms as long as the absolute value of the next term is greater than ϵ . It is natural to provide a default value for ϵ :

```
def L2(x, epsilon=1.0E-6):
    x = float(x)
    i = 1
    term = (1.0/i)*(x/(1+x))**i
    s = term
    while abs(term) > epsilon:    # abs(x) is |x|
        i += 1
        term = (1.0/i)*(x/(1+x))**i
        s += term
    return s, i
```

Here is an example involving this function to make a table of the approximation error as ϵ decreases:

```
from math import log
x = 10
for k in range(4, 14, 2):
    epsilon = 10**(-k)
    approx, n = L2(x, epsilon=epsilon)
    exact = log(1+x)
    exact_error = exact - approx
    print 'epsilon: %5.0e, exact error: %8.2e, n=%d' % \
        (epsilon, exact_error, n)
```

The output becomes

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

We see that the `epsilon` estimate is almost 10 times smaller than the exact error, regardless of the size of `epsilon`. Since `epsilon` follows the exact error quite well over many orders of magnitude, we may view `epsilon` as a useful indication of the size of the error.

2.2.7 Doc Strings

There is a convention in Python to insert a documentation string right after the `def` line of the function definition. The documentation string, known as a *doc string*, should contain a short description of the purpose of the function and explain what the different arguments and return values are. Interactive sessions from a Python shell are also common to illustrate how the code is used. Doc strings are usually enclosed in triple double quotes `"""`, which allow the string to span several lines.

Here are two examples on short and long doc strings:

```
def C2F(C):
    """Convert Celsius degrees (C) to Fahrenheit."""
    return (9.0/5)*C + 32

def line(x0, y0, x1, y1):
    """
    Compute the coefficients a and b in the mathematical
    expression for a straight line  $y = a*x + b$  that goes
    through two points (x0, y0) and (x1, y1).

    x0, y0: a point on the line (floats).
    x1, y1: another point on the line (floats).
    return: coefficients a, b (floats) for the line (y=a*x+b).
    """
    a = (y1 - y0)/float(x1 - x0)
    b = y0 - a*x0
    return a, b
```

Note that the doc string must appear before any statement in the function body.

There are several Python tools that can automatically extract doc strings from the source code and produce various types of documentation, see [5, App. B.2]. The doc string can be accessed in a code as `funcname.__doc__`, where `funcname` is the name of the function, e.g.,

```
print line.__doc__
```

which prints out the documentation of the `line` function above:

```
Compute the coefficients a and b in the mathematical
expression for a straight line  $y = a*x + b$  that goes
through two points (x0, y0) and (x1, y1).

x0, y0: a point on the line (float objects).
x1, y1: another point on the line (float objects).
return: coefficients a, b for the line (y=a*x+b).
```

Doc strings often contain interactive sessions, copied from a Python shell, to illustrate how the function is used. We can add such a session to the doc string in the `line` function:

```
def line(x0, y0, x1, y1):
    """
    Compute the coefficients a and b in the mathematical
    expression for a straight line  $y = a*x + b$  that goes
    through two points (x0,y0) and (x1,y1).
```

```

x0, y0: a point on the line (float).
x1, y1: another point on the line (float).
return: coefficients a, b (floats) for the line (y=a*x+b).

```

```

Example:
>>> a, b = line(1, -1, 4, 3)
>>> a
1.3333333333333333
>>> b
-2.3333333333333333
"""
a = (y1 - y0)/float(x1 - x0)
b = y0 - a*x0
return a, b

```

A particularly nice feature is that all such interactive sessions in doc strings can be automatically run, and new results are compared to the results found in the doc strings. This makes it possible to use interactive sessions in doc strings both for exemplifying how the code is used and for testing that the code works.

2.2.8 Function Input and Output

It is a convention in Python that function arguments represent the input data to the function, while the returned objects represent the output data. We can sketch a general Python function as

```

def somefunc(i1, i2, i3, io4, io5, i6=value1, io7=value2):
    # modify io4, io5, io6; compute o1, o2, o3
    return o1, o2, o3, io4, io5, io7

```

Here `i1`, `i2`, `i3` are positional arguments representing input data; `io4` and `io5` are positional arguments representing input *and* output data; `i6` and `io7` are keyword arguments representing input and input/output data, respectively; and `o1`, `o2`, and `o3` are computed objects in the function, representing output data together with `io4`, `io5`, and `io7`. All examples later in the book will make use of this convention.

2.2.9 Functions as Arguments to Functions

Programs doing calculus frequently need to have functions as arguments in other functions. For example, for a mathematical function $f(x)$ we can have Python functions for

1. numerical root finding: solve $f(x) = 0$ approximately (Chapters 3.6.2 and 5.1.9)
2. numerical differentiation: compute $f'(x)$ approximately (Appendix A and Chapters 7.3.2 and 9.2)
3. numerical integration: compute $\int_a^b f(x)dx$ approximately (Appendix A and Chapters 7.3.3 and 9.3)

4. numerical solution of differential equations: $\frac{dx}{dt} = f(x)$ (Appendix B and Chapters 7.4 and 9.4)

In such Python functions we need to have the $f(x)$ function as an argument `f`. This is straightforward in Python and hardly needs any explanation, but in most other languages special constructions must be used for transferring a function to another function as argument.

As an example, consider a function for computing the second-order derivative of a function $f(x)$ numerically:

$$f''(x) \approx \frac{f(x-h) - 2f(x) + f(x+h)}{h^2}, \quad (2.3)$$

where h is a small number. The approximation (2.3) becomes exact in the limit $h \rightarrow 0$. A Python function for computing (2.3) can be implemented as follows:

```
def diff2(f, x, h=1E-6):
    r = (f(x-h) - 2*f(x) + f(x+h))/float(h*h)
    return r
```

The `f` argument is like any other argument, i.e., a name for an object, here a function object that we can call as we normally call function objects. An application of `diff2` can read

```
def g(t):
    return t**(-6)

t = 1.2
d2g = diff2(g, t)
print "g' (%f)=%f" % (t, d2g)
```

The Behaviour of the Numerical Derivative as $h \rightarrow 0$. From mathematics we know that the approximation formula (2.3) becomes more accurate as h decreases. Let us try to demonstrate this expected feature by making a table of the second-order derivative of $g(t) = t^{-6}$ at $t = 1$ as $h \rightarrow 0$:

```
for k in range(1,15):
    h = 10**(-k)
    d2g = diff2(g, 1, h)
    print 'h=%0e: %.5f' % (h, d2g)
```

The output becomes

```
h=1e-01: 44.61504
h=1e-02: 42.02521
h=1e-03: 42.00025
h=1e-04: 42.00000
h=1e-05: 41.99999
h=1e-06: 42.00074
h=1e-07: 41.94423
h=1e-08: 47.73959
h=1e-09: -666.13381
h=1e-10: 0.00000
h=1e-11: 0.00000
h=1e-12: -666133814.77509
```

```
h=1e-13: 66613381477.50939
h=1e-14: 0.00000
```

With $g(t) = t^{-6}$, the exact answer is $g''(1) = 42$, but for $h < 10^{-8}$ the computations give totally wrong answers! The problem is that for small h on a computer, round-off errors in the formula (2.3) blow up and destroy the accuracy. The mathematical result that (2.3) becomes an increasingly better approximation as h gets smaller and smaller does not hold on a computer! Or more precisely, the result holds until h in the present case reaches 10^{-4} .

The reason for the inaccuracy is that the numerator in (2.3) for $g(t) = t^{-6}$ and $t = 1$ contains subtraction of quantities that are almost equal. The result is a very small and inaccurate number. The inaccuracy is magnified by h^{-2} , a number that becomes very large for small h . Switching from the standard floating-point numbers (`float`) to numbers with arbitrary high precision resolves the problem. Python has a module `decimal` that can be used for this purpose. The file `highprecision.py` solves the current problem using arithmetics based on the `decimal` module. With 25 digits in `x` and `h` inside the `diff2` function, we get accurate results for $h \leq 10^{-13}$. However, for most practical applications of (2.3), a moderately small h , say $10^{-3} \leq h \leq 10^{-4}$, gives sufficient accuracy and then round-off errors from `float` calculations do not pose problems. Real-world science or engineering applications usually have many parameters with uncertainty, making the end result also uncertain, and formulas like (2.3) can then be computed with moderate accuracy without affecting the overall computational error.

2.2.10 The Main Program

In programs containing functions we often refer to a part of the program that is called the *main program*. This is the collection of all the statements outside the functions, plus the definition of all functions. Let us look at a complete program:

```
from math import *           # in main

def f(x):                    # in main
    e = exp(-0.1*x)
    s = sin(6*pi*x)
    return e*s

x = 2                        # in main
y = f(x)                    # in main
print 'f(%g)=%g' % (x, y)   # in main
```

The main program here consists of the lines with a comment `in main`. The execution always starts with the first line in the main program. When a function is encountered, its statements are just used to define the function – nothing gets computed inside the function before

we explicitly call the function, either from the main program or from another function. All variables initialized in the main program become global variables (see Chapter 2.2.2).

The program flow in the program above goes as follows:

1. Import functions from the `math` module,
2. define a function `f(x)`,
3. define `x`,
4. call `f` and execute the function body,
5. define `y` as the value returned from `f`,
6. print the string.

In point 4, we jump to the `f` function and execute the statement inside that function for the first time. Then we jump back to the main program and assign the `float` object returned from `f` to the `y` variable.

More information on program flow and the jump between the main program and functions is covered in Chapter 2.4.2 and Appendix D.1.

2.2.11 Lambda Functions

There is a quick one-line construction of functions that is sometimes convenient:

```
f = lambda x: x**2 + 4
```

This so-called *lambda function* is equivalent to writing

```
def f(x):  
    return x**2 + 4
```

In general,

```
def g(arg1, arg2, arg3, ...):  
    return expression
```

can be written as

```
g = lambda arg1, arg2, arg3, ...: expression
```

Lambda functions are usually used to quickly define a function as argument to another function. Consider, as an example, the `diff2` function from Chapter 2.2.9. In the example from that chapter we want to differentiate $g(t) = t^{-6}$ twice and first make a Python function `g(t)` and then send this `g` to `diff2` as argument. We can skip the step with defining the `g(t)` function and instead insert a lambda function as the `f` argument in the call to `diff2`:

```
d2 = diff2(lambda t: t**(-6), 1, h=1E-4)
```

Because lambda functions can be defined “on the fly” and thereby save typing of a separate function with `def` and an intended block, lambda functions are popular among many programmers.

Lambda functions may also take keyword arguments. For example,

```
d2 = diff2(lambda t, A=1, a=0.5: -a*2*t*A*exp(-a*t**2), 1.2)
```

2.3 If Tests

The flow of computer programs often needs to branch. That is, if a condition is met, we do one thing, and if not, we do another thing. A simple example is a function defined as

$$f(x) = \begin{cases} \sin x, & 0 \leq x \leq \pi \\ 0, & \text{otherwise} \end{cases} \quad (2.4)$$

In a Python implementation of this function we need to test on the value of x , which can be done as displayed below:

```
def f(x):
    if 0 <= x <= pi:
        value = sin(x)
    else:
        value = 0
    return value
```

The general structure of an if-else test is

```
if condition:
    <block of statements, executed if condition is True>
else:
    <block of statements, executed if condition is False>
```

When condition evaluates to true, the program flow branches into the first block of statements. If condition is False, the program flow jumps to the second block of statements, after the `else:` line. As with `while` and `for` loops, the block of statements are indented. Here is another example:

```
if C < -273.15:
    print '%g degrees Celsius is non-physical.' % C
    print 'The Fahrenheit temperature will not be computed.'
else:
    F = 9.0/5*C + 32
    print F
print 'end of program'
```

The two `print` statements in the `if` block are executed if and only if `C < -273.15` evaluates to True. Otherwise, we jump over the first two `print` statements and carry out the computation and printing of `F`. The printout of `end of program` will be performed regardless of the outcome of the `if` test since this statement is not indented and hence neither a part of the `if` block nor the `else` block.

The `else` part of an `if` test can be skipped, if desired:

```
if condition:
    <block of statements>
<next statement>
```

For example,

```
if C < -273.15:
    print '%s degrees Celsius is non-physical!' % C
F = 9.0/5*C + 32
```

In this case the computation of `F` will always be carried out, since the statement is not indented and hence not a part of the `if` block.

With the keyword `elif`, short for *else if*, we can have several mutually exclusive `if` tests, which allows for multiple branching of the program flow:

```
if condition1:
    <block of statements>
elif condition2:
    <block of statements>
elif condition3:
    <block of statements>
else:
    <block of statements>
<next statement>
```

The last `else` part can be skipped if it is not needed. To illustrate multiple branching we will implement a “hat” function, which is widely used in advanced computer simulations in science and industry. One example of a “hat” function is

$$N(x) = \begin{cases} 0, & x < 0 \\ x, & 0 \leq x < 1 \\ 2 - x, & 1 \leq x < 2 \\ 0, & x \geq 2 \end{cases} \quad (2.5)$$

The solid line in Figure 4.11 on page 203 illustrates the shape of this function. The Python implementation associated with (2.5) needs multiple `if` branches:

```
def N(x):
    if x < 0:
        return 0.0
    elif 0 <= x < 1:
        return x
    elif 1 <= x < 2:
        return 2 - x
    elif x >= 2:
        return 0.0
```

This code corresponds directly to the mathematical specification, which is a sound strategy that usually leads to fewer errors in programs. We could mention that there is another way of constructing this `if` test that results in shorter code:

```
def N(x):  
    if 0 <= x < 1:  
        return x  
    elif 1 <= x < 2:  
        return 2 - x  
    else:  
        return 0
```

As a part of learning to program, understanding this latter sample code is important, but we recommend the former solution because of its direct similarity with the mathematical definition of the function.

A popular programming rule is to avoid multiple `return` statements in a function – there should only be one `return` at the end. We can do that in the `N` function by introducing a local variable, assigning values to this variable in the blocks and returning the variable at the end. However, we do not think an extra variable and an extra line make a great improvement in such a short function. Nevertheless, in long and complicated functions the rule can be helpful.

A variable is often assigned a value that depends on a boolean expression. This can be coded using a common `if-else` test:

```
if condition:  
    a = value1  
else:  
    a = value2
```

Because this construction is often needed, Python provides a one-line syntax for the four lines above:

```
a = (value1 if condition else value2)
```

The parentheses are not required, but recommended style. One example is

```
def f(x):  
    return (sin(x) if 0 <= x <= 2*pi else 0)
```

Since the inline `if` test is an expression with a value, it can be used in lambda functions:

```
f = lambda x: sin(x) if 0 <= x <= 2*pi else 0
```

The traditional `if-else` construction with indented blocks cannot be used inside lambda functions because it is not just an expression (lambda functions cannot have statements inside them, only a single expression).

2.4 Summary

2.4.1 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. Table 2.1 shows some important list operations (only a subset of these are explained in the present chapter).

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 `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 (Chapter 4) 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* (introduced in Chapter 6.2) is often more convenient.

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

Table 2.1 Summary of important functionality for list objects.

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

```
>>> 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 `print` otherwise adds to the character.

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

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

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 better control the output of floating-point numbers.

If Tests. The `if-elif-else` tests are used to “branch” the flow of statements. That is, different sets of statements are executed depending on whether a set of conditions is true or not.

```
def f(x):
    if x < 0:
        value = -1
    elif x >= 0 and x <= 1:
        value = x
    else:
        value = 1
    return value
```

User-Defined Functions. Functions are useful (i) when a set of commands are to be executed several times, or (ii) to partition the program into smaller pieces to gain better overview. Function arguments are local variables inside the function whose values are set when calling the function. Remember that when you write the function, the values of the arguments are not known. Here is an example of a function for polynomials of 2nd degree:

```
# function definition:
def quadratic_polynomial(x, a, b, c)
    value = a*x*x + b*x + c
    derivative = 2*a*x + b
    return value, derivative

# function call:
x = 1
p, dp = quadratic_polynomial(x, 2, 0.5, 1)
p, dp = quadratic_polynomial(x=x, a=-4, b=0.5, c=0)
```

The sequence of the arguments is important, unless all arguments are given as name=value.

Functions may have no arguments and/or no return value(s):

```
def print_date():
    """Print the current date in the format 'Jan 07, 2007'."""
    import time
    print time.strftime("%b %d, %Y")

# call:
print_date()
```

A common error is to forget the parentheses: `print_date` is the function object itself, while `print_date()` is a call to the function.

Keyword Arguments. Function arguments with default values are called keyword arguments, and they help to document the meaning of arguments in function calls. They also make it possible to specify just a subset of the arguments in function calls.

```
from math import exp, sin, pi

def f(x, A=1, a=1, w=pi):
    return A*exp(-a*x)*sin(w*x)

f1 = f(0)
x2 = 0.1
f2 = f(x2, w=2*pi)
```

```
f3 = f(x2, w=4*pi, A=10, a=0.1)
f4 = f(w=4*pi, A=10, a=0.1, x=x2)
```

The sequence of the keyword arguments can be arbitrary, and the keyword arguments that are not listed in the call get their default values according to the function definition. The “non-keyword arguments” are called positional arguments, which is `x` in this example. Positional arguments must be listed before the keyword arguments. However, also a positional argument can appear as `name=value` in the call (see the last line above), and this syntax allows any positional argument to be listed anywhere in the call.

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,
- function,
- method,
- return statement,
- positional arguments,
- keyword arguments,
- local and global variables,
- doc strings,
- if tests (branching),
- the `None` object.

2.4.2 Summarizing Example: Tabulate a Function

Problem. Make a program for evaluating the formula (1.1) for time points equally spaced by Δt as long as $y \geq 0$. These time points and their associated y values are to be stored in a nested list, to be printed out on the screen. Also search through the list to find the maximum $y(t)$ value.

Solution. We first present the program solving the problem stated above, and then we explain in detail how the program works. The code is found in the file `ball_table.py`.

```

g = 9.81; v0 = 5
dt = 0.25

def y(t):
    return v0*t - 0.5*g*t**2

def table():
    data = [] # store [t, y] pairs in a nested list
    t = 0
    while y(t) >= 0:
        data.append([t, y(t)])
        t += dt
    return data

data = table()

for t, y in data:
    print '%5.1f %5.1f' % (t, y)

# extract all y values from data:
y = [y for t, y in data]
print y
# find maximum y value:
ymax = 0
for yi in y:
    if yi > ymax:
        ymax = yi
print 'max y(t) =', ymax

data = table() # this does not work now - why?

```

Recall that a program is executed from top to bottom, line by line, but the program flow may “jump around” because of functions and loops. Understanding the program flow in detail is a necessary and important ability if (when!) you need to find errors in a program that produces wrong results.

The present program starts with executing the first two lines, which bring the variables `g`, `v0`, and `dt` into play. Thereafter, two functions `y` and `table` are defined, but nothing inside these functions is computed. The computations in the function bodies are performed when we call the functions.

The first function call, `data = table()`, causes the program flow to jump into the `table` function and execute the statements in this function. When entering the `while` loop, the boolean expression¹² `y(t) >= 0` is evaluated. This expression requires the evaluation of `y(t)`, which causes the program flow to jump to the `y` function. Inside this function, the argument `t` is a local variable that has the value 0, because the local variable `t` in the `table` function has the value 0 in the first call `y(t)`. The expression in the `return` statement in the `y` function is then evaluated to 0 and returned.

¹² Some experienced programmers may criticize the `table` function for having an unnecessary extra call to `y(t)` in each pass in the loop. Exercise 2.53 asks you to rewrite the function such that there is only one call to `y(t)` in the loop. However, in this summary section we have chosen to write code that is as easy to understand as possible, instead of writing as computationally efficient code as possible. We believe this strategy is beneficial for newcomers to programming.

We are now back to the boolean condition in the `while` loop. Since `y(t)` was evaluated to 0, the condition reads `0 >= 0`, which is `True`. Therefore we are to execute the block of statements inside the loop. This block of statements is executed repeatedly until the loop condition is `False`. In each pass of the loop, the condition `y(t) >= 0` must be evaluated, and this task causes a call to the `y` function and an associated jump of the program flow. For some `t` value, `y(t)` will be negative, i.e., the loop condition is `False` and the program flow jumps to the first statement after the loop. This statement is `return data` in the present case.

The value of the call `table()` is now computed to be the object with name `data` inside the `table` function. We assign this object to a new global variable `data` in the statement `data = table()`. That is, there are two `data` variables in this program, one local in the `table` function and one global. The local `data` variable is demolished (along with the other local variable, `t`) when the program flow leaves the `table` function. Although the local `data` variable in the `table` function dies, the object that it refers to survives, because we “send out” this object from the function and bind it to a new, global name `data` in the main program. As long as we have some name that refers to an object, the object is alive and can be used. More aspects of this subject are discussed below.

The program flow proceeds to the `for` loop, where we run through all the pairs of `t`, `y` values in the nested list `data`. Recall that all elements of the `data` list are lists with two numbers, `t` and `y`.

The next statement applies a list comprehension to make a new list, `y`, holding all the `y` values that are in `data`. Again, we pass through all pairs `y`, `t` (i.e., elements) in `data`, but we place only the `y` part in the new list.

The next step is to find the maximum `y` value. We first set the maximum value `ymax` to the smallest relevant value, here it is 0 since $y \geq 0$. The `for` loop runs through all elements, and if an element is larger than `ymax`, `ymax` refers to this new element. When all elements in `y` are examined, `ymax` holds the largest value, and we can print it out. Since finding the largest (or smallest) element in a list is a frequently encountered task, Python has a special function `max` (or `min`) such that we could have written `ymax = max(y)` and hence avoided the `for` loop with the `if` test.

The final statement, `data = table()`, causes the program to abort with an error message

```
File "ball_table.py", line 10, in table
    while y(t) >= 0:
TypeError: 'list' object is not callable
```

What has happened? The `table` function worked fine the first time we called it! As the error message tells us, the problem lies in the `while` loop line. You need some programming experience to understand what such an error message means and what the problem can be.

The global name `y` is originally used for a function in the program. However, after the call to `table`, we use the name `y` for several types of objects. First, `y` is used in the `for` loop and will there hold `float` objects. In the list comprehension, `y` is used in an identical way inside a `for` loop, and then `y` is the name for the resulting list! At the second call to `table`, the global name `y` refers to a list. The first time we use the global `y` inside the `table` function is in the loop condition `y(t) >= 0`. Here we try to call a function `y`, but `y` is a list, and the syntax of a call `y(t)` is illegal if `y` is a list. That is why the error message states that 'list' object is not callable. Figure 2.3 illustrates the state of variables in the program after the first `table()` call and at the end.

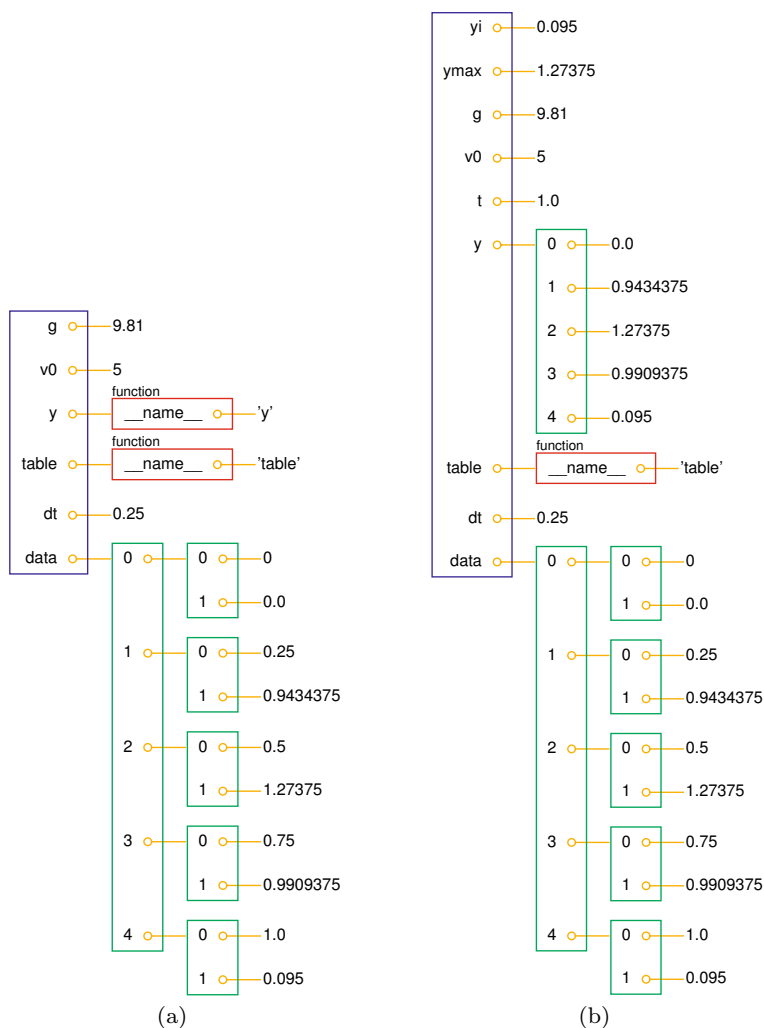


Fig. 2.3 State of variables in `ball_table.py` (a) right after `table()` is called; (b) at the end. Note that `y` refers to a function in (a) and a list in (b).

How can we recover from this error? The simplest remedy is to give the `y` function another name, e.g., `yfunc`. Doing so, the program works. We still use the variable `y` for `float` and `list` objects, but this is okay as long as no errors arise.

In programming languages such as Fortran, C, C++, Java, and C#, any variable `y` is declared with a fixed type, and the problem that `y` is a function and later a number or list cannot occur. This principle removes some errors, but the flexibility of using the symbol `y` for different object types, whose conceptual name is conveniently taken as “`y`”, makes the program easier to read and understand. (There is a Python package “Traits” that offers type checking of variables – in a more flexible way than what is possible with compile-time type checking as in Fortran, C, C++, Java, and C#.)

To better understand the flow of statements in a program, it can be handy to use a debugger. Appendix D.1 explains in detail how we can investigate the program flow in the present example with the aid of Python’s built-in debugger.

2.4.3 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. 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 grab 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

Terminal

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

In Python there are two possibilities, either¹³

```
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 webpages, but `pydoc` has often less information compared to the Python Library Reference.

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 webpage <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/PQR25/PQR2.5.html>.

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

2.5 Exercises

Exercise 2.1. *Make a Fahrenheit–Celsius conversion table.*

Modify the `c2f_table_while.py` program so that it prints out a table with Fahrenheit degrees 0, 10, 20, ..., 100 in the first column and the corresponding Celsius degrees in the second column. Name of program file: `c2f_table_while.py`. ◇

Exercise 2.2. *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`. ◇

¹³ Any command you can run in the terminal window can also be run inside IPython if you start the command with an exclamation mark.

Exercise 2.3. *Store odd numbers in a list.*

Modify the program from Exercise 2.2 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 2.4. *Generate odd numbers by the range function.*

Solve Exercise 2.3 by calling the `range` function to generate a list of odd numbers. Name of program file: `odd_list2.py`. ◇

Exercise 2.5. *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
d = [e+1 for e in a]
print d
d.append(b[0] + 1)
d.append(b[-1] + 1)
print d
```

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

Exercise 2.6. *Make a table of values from formula (1.1).*

Write a program that prints a table of t and $y(t)$ values from the formula (1.1) to the screen. Use 11 uniformly spaced t values throughout the interval $[0, 2v_0/g]$, and fix the value of v_0 . Name of program file: `ball_table1.py`. ◇

Exercise 2.7. *Store values from formula (1.1) in lists.*

In a program, make a list `t` with 6 t values $0.1, 0.2, \dots, 0.6$. Compute a corresponding list `y` of $y(t)$ values using formula (1.1). Write out a nicely formatted table of t and y values. Name of program file: `ball_table2.py`. ◇

Exercise 2.8. *Work with a list.*

Set a variable `primes` to a list containing the numbers 1, 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. *Generate equally spaced coordinates.*

We want to generate x coordinates between 1 and 2 with spacing 0.01. The i -th coordinate, x_i , is then $1 + ih$ where $h = 0.01$ and i runs over integers $0, 1, \dots, 100$. Compute the x_i values and store them in a list. Hint: 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 2.10. *Use a list comprehension to solve Exer. 2.9.*

The problem is the same as in Exercise 2.9, but now we want the x_i values to be stored in a list using a list comprehension construct (see Chapter 2.1.6). Name of program file: `coor2.py`. ◇

Exercise 2.11. *Store data from Exer. 2.7 in a nested list.*

After having computed the two lists of t and y values in the program from Exercise 2.7, store the two lists in a new list `t1`. Write out a table of t and y values by traversing the data in the `t1` list. Thereafter, make a list `t2` which holds each row in the table of t and y values (`t1` is a list of table columns while `t2` is a list of table rows, as explained in Chapter 2.1.7). Write out the table by traversing the `t2` list. Name of program file: `ball_table3.py`. ◇

Exercise 2.12. *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 a program.) Write a correct program. Name of program file: `compute_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. *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
```

- 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.
- 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 2.14. *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 is used instead of a `while` loop. Name of program file: `compute_sum_for.py`. ◇

Exercise 2.15. *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 `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`. Name of program file: `index_nested_list.py`. ◇

Exercise 2.16. *Construct a double for loop over a nested list.*

Consider the list from Exercise 2.15. We can visit all elements of `q` using this nested `for` loop:

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

What type of objects are `i` and `j`? Name of program file: `nested_list_iter.py`. ◇

Exercise 2.17. *Compute the area of an arbitrary triangle.*

An arbitrary triangle can be described by the coordinates of its three vertices: (x_1, y_1) , (x_2, y_2) , (x_3, y_3) . The area of the triangle is given by the formula

$$A = \frac{1}{2} [x_2 y_3 - x_3 y_2 - x_1 y_3 + x_3 y_1 + x_1 y_2 - x_2 y_1] . \quad (2.6)$$

Write a function `area(vertices)` that returns the area of a triangle whose vertices are specified by the argument `vertices`, which is a nested list of the vertex coordinates. For example, `vertices` can be `[[0,0], [1,0], [0,2]]` if the three corners of the triangle have coordinates $(0,0)$, $(1,0)$, and $(0,2)$. Test the `area` function on a triangle with known area. Name of program file: `area_triangle.py`. ◇

Exercise 2.18. *Compute the length of a path.*

Some object is moving along a path in the plane. At n points of time we have recorded the corresponding (x, y) positions of the object:

$(x_0, y_0), (x_1, y_1), \dots, (x_{n-1}, y_{n-1})$. The total length L of the path from (x_0, y_0) to (x_{n-1}, y_{n-1}) is the sum of all the individual line segments $((x_{i-1}, y_{i-1})$ to (x_i, y_i) , $i = 1, \dots, n-1$):

$$L = \sum_{i=1}^{n-1} \sqrt{(x_i - x_{i-1})^2 + (y_i - y_{i-1})^2}. \quad (2.7)$$

Make a function `pathlength(x, y)` for computing L according to the formula. The arguments `x` and `y` hold all the x_0, \dots, x_{n-1} and y_0, \dots, y_{n-1} coordinates, respectively. Test the function on a triangular path with the four points $(1, 1)$, $(2, 1)$, $(1, 2)$, and $(1, 1)$. Name of program file: `pathlength.py`. \diamond

Exercise 2.19. *Approximate π .*

The value of π equals the circumference of a circle with radius $1/2$. Suppose we approximate the circumference by a polygon through $N+1$ points on the circle. The length of this polygon can be found using the `pathlength` function from Exercise 2.18. Compute $N+1$ points (x_i, y_i) along a circle with radius $1/2$ according to the formulas

$$x_i = \frac{1}{2} \cos(2\pi i/N), \quad y_i = \frac{1}{2} \sin(2\pi i/N), \quad i = 0, \dots, N.$$

Call the `pathlength` function and write out the error in the approximation of π for $N = 2^k$, $k = 2, 3, \dots, 10$. Name of program file: `pi_approx.py`. \diamond

Exercise 2.20. *Write a Fahrenheit-Celsius conversion table.*

Given a temperature F in Fahrenheit degrees, the corresponding degrees in Celsius are found by solving (1.2) (on page 18) with respect to C , yielding formula (2.9). Many people use an approximate formula for quickly calculating the Celsius degrees: subtract 30 from the Fahrenheit degrees and divide by two, i.e.,

$$C = (F - 30)/2 \quad (2.8)$$

We want to produce a table that compares the exact formula (2.9) and the rough approximation (2.8) for Fahrenheit degrees between 0 and 100 (in steps of, e.g., 10). Write a program for storing the F values, the exact C values, and the approximate C values in a nested list. Print out a nicely formatted table by traversing the nested list with a `for` loop. Name of program file: `f2c_shortcut_table.py`. \diamond

Exercise 2.21. *Convert nested list comprehensions to nested standard loops.*

Rewrite the generation of the nested list `q`,

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

by using standard `for` loops instead of list comprehensions. Name of program file: `listcomp2for.py`. ◇

Exercise 2.22. *Write a Fahrenheit–Celsius conversion function.*

The formula for converting Fahrenheit degrees to Celsius reads

$$C = \frac{5}{9}(F - 32). \quad (2.9)$$

Write a function `C(F)` that implements this formula. To verify the implementation of `C(F)`, you can convert a Celsius temperature to Fahrenheit and then back to Celsius again using the `F(C)` function from Chapter 2.2.1 and the `C(F)` function implementing (2.9). That is, you can check that a temperature `c` equals `C(F(c))` (be careful with comparing real numbers with `==`, see Exercise 2.51). Name of program file: `c2f2c.py`. ◇

Exercise 2.23. *Write some simple functions.*

Write three functions:

1. `hw1`, which takes no arguments and returns the string `'Hello, World!'`
2. `hw2`, which takes no arguments and returns nothing, but the string `'Hello, World!'` is printed in the terminal window
3. `hw3`, which takes two string arguments and prints these two arguments separated by a comma

Use the following main program to test the three functions:

```
print hw1()
hw2()
hw3('Hello ', 'World!')
```

Name of program: `hw_func.py`. ◇

Exercise 2.24. *Write the program in Exer. 2.12 as a function.*

Define a Python function `s(M)` that computes the sum s as defined in Exercise 2.12. Name of program: `compute_sum_func.py`. ◇

Exercise 2.25. *Implement a Gaussian function.*

Make a Python function `gauss(x, m=0, s=1)` for computing the Gaussian function (1.6) on page 45. Call `gauss` and print out the result for x equal to $-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5$, using default values for m and s . Name of program file: `Gaussian_function2.py`. ◇

Exercise 2.26. *Find the max and min values of a function.*

Write a function `maxmin(f, a, b, n=1000)` that returns the maximum and minimum values of a mathematical function $f(x)$ (evaluated at n points) in the interval between a and b . The following test program

```
from math import cos, pi
print maxmin(cos, -pi/2, 2*pi)
```

should write out (1.0, -1.0).

The `maxmin` function can compute a set of `n` coordinates between `a` and `b` stored in a list `x`, then compute `f` at the points in `x` and store the values in another list `y`. The Python functions `max(y)` and `min(y)` return the maximum and minimum values in the list `y`, respectively. Name of program file: `func_maxmin.py`. ◇

Exercise 2.27. *Explore the Python Library Reference.*

Suppose you want to make a program for printing out $\sin^{-1} x$ for n x values between 0 and 1. The `math` module has a function for computing $\sin^{-1} x$, but what is the right name of this function? Read Chapter 2.4.3 and use the `math` entry in the index of the Python Library Reference to find out how to compute $\sin^{-1} x$. Name of program file: `inverse_sine.py`. ◇

Exercise 2.28. *Make a function of the formula in Exer. 1.12.*

Implement the formula (1.8) from Exercise 1.12 in a Python function with three arguments: `egg(M, To=20, Ty=70)`. The parameters ρ , K , c , and T_w can be set as local (constant) variables inside the function. Let t be returned from the function. Compute t for a soft and hard boiled egg, of a small ($M = 47$ g) and large ($M = 67$ g) size, taken from the fridge ($T_o = 4$ C) and from a hot room ($T = 25$ C). Name of program file: `egg_func.py`. ◇

Exercise 2.29. *Write a function for numerical differentiation.*

The formula

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h} \quad (2.10)$$

can be used to find an approximate derivative of a mathematical function $f(x)$ if h is small. Write a function `diff(f, x, h=1E-6)` that returns the approximation (2.10) of the derivative of a mathematical function represented by a Python function `f(x)`.

Apply (2.10) to differentiate $f(x) = e^x$ at $x = 0$, $f(x) = e^{-2x^2}$ at $x = 0$, $\cos x$ at $x = 2\pi$, and $f(x) = \ln x$ at $x = 1$. Use $h = 0.01$. In each case, write out the error, i.e., the difference between the exact derivative and the result of (2.10). Name of program file: `diff_f.py`. ◇

Exercise 2.30. *Write a function for numerical integration.*

An approximation to the integral of a function $f(x)$ over an interval $[a, b]$ can be found by first approximating $f(x)$ by the straight line that goes through the end points $(a, f(a))$ and $(b, f(b))$, and then finding the area under the straight line (which is the area of a trapezoid). The resulting formula becomes

$$\int_a^b f(x) dx \approx \frac{b-a}{2} (f(a) + f(b)). \quad (2.11)$$

Write a function `integrate(f, a, b)` that returns this approximation to the integral. The argument `f` is a Python implementation `f(x)` of the mathematical function $f(x)$.

Compute the error, i.e., the difference between the approximation (2.11) and the exact result, for Using (2.11), compute the following integrals $\int_0^{\ln 3} e^x dx$, $\int_0^\pi \cos x dx$, $\int_0^\pi \sin x dx$, and $\int_0^{\pi/2} \sin x dx$. In each case, write out the error, i.e., the difference between the exact integral and the approximation (2.11). Make rough sketches of (2.11) for each integral in order to understand how the method behaves in the different cases. Name of program file: `int_f.py`. \diamond

Exercise 2.31. *Improve the formula in Exer. 2.30.*

We can easily improve the formula 2.11 from Exercise 2.30 by approximating the function $f(x)$ by a straight line from $(a, f(a))$ to the midpoint $(c, f(c))$ between a and b , and then from the midpoint to $(b, f(b))$. The midpoint c equals $\frac{1}{2}(a + b)$. The area under the two straight lines equals the area of two trapezoids. Derive a formula for this area and implement the formula in a function `integrate2(f, a, b)`. Run the examples from Exercise 2.30 and see how much better the new formula is. Name of program file: `int2_f.py`. \diamond

Exercise 2.32. *Compute a polynomial via a product.*

Given n roots r_0, r_1, \dots, r_n of a polynomial of degree n , the polynomial $p(x)$ can be computed by

$$p(x) = \prod_{i=0}^n (x - r_i) = (x - r_0)(x - r_1) \cdots (x - r_{n-1})(x - r_n). \quad (2.12)$$

Store the roots r_0, \dots, r_n in a list and make a loop that computes the product in (2.12). Test the program on a polynomial with roots -1 , 1 , and 2 . Name of program file: `polyprod.py`. \diamond

Exercise 2.33. *Implement the factorial function.*

The factorial of n , written as $n!$, is defined as

$$n! = n(n-1)(n-2) \cdots 2 \cdot 1, \quad (2.13)$$

with the special cases

$$1! = 1, \quad 0! = 1. \quad (2.14)$$

For example, $4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24$, and $2! = 2 \cdot 1 = 2$. Write a function `fact(n)` that returns $n!$. Return 1 immediately if x is 1 or 0, otherwise use a loop to compute $n!$. Name of program file: `fact.py`.

Remark. You can import a ready-made factorial function by

```
from scitools.std import factorial
```


This `factorial` function offers many different implementations, with different computational efficiency, for computing $x!$ (see the source code of the function for details). \diamond

Exercise 2.34. *Compute velocity and acceleration from position data; one dimension.*

Let $x(t)$ be the position of an object moving along the x axis. The velocity $v(t)$ and acceleration $a(t)$ can be approximately computed by the formulas

$$v(t) \approx \frac{x(t + \Delta t) - x(t - \Delta t)}{2\Delta t}, \quad a(t) \approx \frac{x(t + \Delta t) - 2x(t) + x(t - \Delta t)}{\Delta t^2}, \quad (2.15)$$

where Δt is a small time interval. As $\Delta \rightarrow 0$, the above formulas approach the first and second derivative of $x(t)$, which coincides with the well-known definitions of velocity and acceleration.

Write a function `kinematics(x, t, dt=1E-6)` for computing x , v , and a at time t , using the above formulas for v and a with Δt corresponding to `dt`. Let the function return x , v , and a . Test the function with the position function $x(t) = e^{-(t-4)^2}$ and the time point $t = 5$ (use $\Delta t = 10^{-5}$). Name of program: `kinematics1.py`. \diamond

Exercise 2.35. *Compute velocity and acceleration from position data; two dimensions.*

An object moves a long a path in the xy plane such that at time t the object is located at the point $(x(t), y(t))$. The velocity vector in the plane, at time t , can be approximated as

$$v(t) \approx \left(\frac{x(t + \Delta t) - x(t - \Delta t)}{2\Delta t}, \frac{y(t + \Delta t) - y(t - \Delta t)}{2\Delta t} \right). \quad (2.16)$$

The acceleration vector in the plane, at time t , can be approximated as

$$a(t) \approx \left(\frac{x(t + \Delta t) - 2x(t) + x(t - \Delta t)}{\Delta t^2}, \frac{y(t + \Delta t) - 2y(t) + y(t - \Delta t)}{\Delta t^2} \right). \quad (2.17)$$

Here, Δt is a small time interval.

Make a function `kinematics(x, y, t, dt=1E-6)` for computing the velocity and acceleration of the object according to the formulas above (`t` corresponds to t , and `dt` corresponds to Δt). The function should return three 2-tuples holding the position, the velocity, and the acceleration, all at time t . Test the function for the motion along a circle with radius R and absolute velocity $R\omega$: $x(t) = R \cos \omega t$ and $y(t) = R \sin \omega t$. Compute the velocity and acceleration for $t = 0$ and $t = \pi$ using $R = 1$ and $\omega = 2\pi$. Name of program: `kinematics2.py`. \diamond

Exercise 2.36. *Express a step function as a Python function.*

The following “step” function is known as the Heaviside function and is widely used in mathematics:

$$H(x) = \begin{cases} 0, & x < 0 \\ 1, & x \geq 0 \end{cases} \quad (2.18)$$

Write a Python function `H(x)` that evaluates the formula for $H(x)$. Test your implementation for $x = -\frac{1}{2}, 0, 10$. Name of program file: `Heaviside.py`. \diamond

Exercise 2.37. *Rewrite a mathematical function.*

We consider the $L(x; n)$ sum as defined in Chapter 2.2.4 and the corresponding function `L2(x, epsilon)` function from Chapter 2.2.6. The sum $L(x; n)$ can be written as

$$L(x; n) = \sum_{i=1}^n c_i, \quad c_i = \frac{1}{i} \left(\frac{x}{1+x} \right)^i.$$

Derive a relation between c_i and c_{i-1} ,

$$c_i = a c_{i-1},$$

where a is an expression involving i and x . This relation between c_i and c_{i-1} means that we can start with `term` as c_1 , and then in each pass of the loop implementing the sum $\sum_i c_i$ we can compute the next term c_i in the sum as

```
term = a*term
```

Rewrite the `L2` function to make use of this alternative computation. Compare the new version with the original one to verify the implementation. Name of program file: `L2_recursive.py`. \diamond

Exercise 2.38. *Make a table for approximations of $\cos x$.*

The function $\cos(x)$ can be approximated by the sum

$$C(x; n) = \sum_{j=0}^n c_j, \quad (2.19)$$

where

$$c_j = -c_{j-1} \frac{x^2}{2j(2j-1)}, \quad j = 1, 2, \dots, n,$$

and $c_0 = 0$. Make a Python function for computing $C(x; n)$. (Hint: Represent c_j by a variable `term`, make updates `term = -term*...` inside a `for` loop, and accumulate the `term` variable in a variable for the sum.)

Also make a function for writing out a table of the errors in the approximation $C(x; n)$ of $\cos(x)$ for some x and n values given as arguments to the function. Let the x values run downward in the rows

and the n values to the right in the columns. For example, a table for $x = 4\pi, 6\pi, 8\pi, 10\pi$ and $n = 5, 25, 50, 100, 200$ can look like

x	5	25	50	100	200
12.5664	1.61e+04	1.87e-11	1.74e-12	1.74e-12	1.74e-12
18.8496	1.22e+06	2.28e-02	7.12e-11	7.12e-11	7.12e-11
25.1327	2.41e+07	6.58e+04	-4.87e-07	-4.87e-07	-4.87e-07
31.4159	2.36e+08	6.52e+09	1.65e-04	1.65e-04	1.65e-04

Observe how the error increases with x and decreases with n . Name of program file: `cossum.py`. ◇

Exercise 2.39. *Implement Exer. 1.13 with a loop.*

Make a function for evaluating $S(t; n)$ as defined in Exercise 1.13 on page 46, using a loop to sum up the n terms. The function should take t , T , and n as arguments and return $S(t; n)$ and the error in the approximation of $f(t)$. Make a main program that sets $T = 2$ and writes out a table of the approximation errors for $n = 1, 5, 20, 50, 100, 200, 500, 1000$ and $t = 1.01, 1.1, 1.8$. Use a row for each n value and a column for each t value. Name of program file: `compute_sum_S.py`. ◇

Exercise 2.40. *Determine the type of objects.*

Consider the following calls to the `makelist` function from page 76:

```
11 = makelist(0, 100, 1)
12 = makelist(0, 100, 1.0)
13 = makelist(-1, 1, 0.1)
14 = makelist(10, 20, 20)
15 = makelist([1,2], [3,4], [5])
16 = makelist((1,-1,1), ('myfile.dat', 'yourfile.dat'))
17 = makelist('myfile.dat', 'yourfile.dat', 'herfile.dat')
```

Determine in each case what type of objects that become elements in the returned list and what the contents of `value` is after one pass in the loop.

Hint: Simulate the program by hand and check out in an interactive session what type of objects that result from the arithmetics. It is only necessary to simulate one pass of the loop to answer the questions. Some of the calls will lead to infinite loops if you really execute the `makelist` calls on a computer.

This exercise demonstrates that we can write a function and have in mind certain types of arguments, here typically `int` and `float` objects. However, the function can be used with other (originally unintended) arguments, such as lists and strings in the present case, leading to strange and irrelevant behavior (the problem here lies in the boolean expression `value <= stop` which is meaningless for some of the arguments). ◇

Exercise 2.41. *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:

```
>>> sum([1,3,5,-5])
4
```

Implement your own version of `sum`. Name of program: `sum.py`. ◇

Exercise 2.42. *Find the max/min elements in a list.*

Given a list `a`, the `max` function in Python's standard library computes the largest element in `a`: `max(a)`. Similarly, `min(a)` returns the smallest element in `a`. The purpose of this exercise is to write your own `max` and `min` function. Use the following technique: Initialize a variable `max_elem` by the first element in the list, then visit all the remaining elements (`a[1:]`), compare each element to `max_elem`, and if greater, make `max_elem` refer to that element. Use a similar technique to compute the minimum element. Collect the two pieces of code in functions. Name of program file: `maxmin_list.py`. ◇

Exercise 2.43. *Demonstrate list functionality.*

Create an interactive session where you demonstrate the effect of each of the operations in Table 2.1 on page 92. Use IPython and log the results (see Exercise 1.11). Name of program file: `list_demo.py`. ◇

Exercise 2.44. *Write a sort function for a list of 4-tuples.*

Below is a list of the nearest stars and some of their properties. The list elements are 4-tuples containing the name of the star, the distance from the sun in light years, the apparent brightness, and the luminosity. The apparent brightness is how bright the stars look in our sky compared to the brightness of Sirius A. The luminosity, or the true brightness, is how bright the stars would look if all were at the same distance compared to the Sun. The list data are found in the file `stars.list`, which looks as follows:

```
data = [
('Alpha Centauri A', 4.3, 0.26, 1.56),
('Alpha Centauri B', 4.3, 0.077, 0.45),
('Alpha Centauri C', 4.2, 0.00001, 0.00006),
('Barnard's Star', 6.0, 0.00004, 0.0005),
('Wolf 359', 7.7, 0.000001, 0.00002),
('BD +36 degrees 2147', 8.2, 0.0003, 0.006),
('Luyten 726-8 A', 8.4, 0.000003, 0.00006),
('Luyten 726-8 B', 8.4, 0.000002, 0.00004),
('Sirius A', 8.6, 1.00, 23.6),
('Sirius B', 8.6, 0.001, 0.003),
('Ross 154', 9.4, 0.00002, 0.0005),
]
```

The purpose of this exercise is to sort this list with respect to distance, apparent brightness, and luminosity.

To sort a list `data`, one can call `sorted(data)`, which returns the sorted list (cf. Table 2.1). However, in the present case each element is a 4-tuple, and the default sorting of such 4-tuples result in a list with the stars appearing in alphabetic order. We need to sort with respect to the 2nd, 3rd, or 4th element of each 4-tuple. If a tailored

sort mechanism is necessary, we can provide our own sort function as a second argument to `sorted`, as in `sorted(data, mysort)`. Such a tailored sort function `mysort` must take two arguments, say `a` and `b`, and returns `-1` if `a` should become before `b` in the sorted sequence, `1` if `b` should become before `a`, and `0` if they are equal. In the present case, `a` and `b` are 4-tuples, so we need to make the comparison between the right elements in `a` and `b`. For example, to sort with respect to luminosity we write

```
def mysort(a, b):
    if a[3] < b[3]:
        return -1
    elif a[3] > b[3]:
        return 1
    else:
        return 0
```

Write the complete program which initializes the data and writes out three sorted tables: star name versus distance, star name versus apparent brightness, and star name versus luminosity. Name of program file: `sorted_stars_data.py`. ◇

Exercise 2.45. *Find prime numbers.*

The *Sieve of Eratosthenes* is an algorithm for finding all prime numbers less than or equal to a number N . Read about this algorithm on Wikipedia and implement it in a Python program. Name of program file: `find_primes.py`. ◇

Exercise 2.46. *Condense the program in Exer. 2.14.*

The program in Exercise 2.14 can be greatly condensed by applying the `sum` function to a list of all the elements $1/k$ in the sum $\sum_{k=1}^M \frac{1}{k}$:

```
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

```
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). 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 E.6.1 to measure the CPU time spent by each construction. Write out M and the CPU time for $M = 10^4, 10^6, 10^8$. (Your computer may become very busy and “hang” in the last case because the list comprehension and `range` calls demand $2M$ numbers to be stored, which

may exceed the computer's memory capacity.) Name of program file: `compute_sum_compact.py`. ◇

Exercise 2.47. *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 2.48. *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(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 2.49. *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`¹⁴ were zero? Name of program file: `machine_zero.py`.

If somebody shows you this interactive session

```

>>> 0.5 + 1.45E-22
0.5

```

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

Exercise 2.50. *Resolve a problem with a function.*

Consider the following interactive session:

```

>>> def f(x):
...     if 0 <= x <= 2:
...         return x**2
...     elif 2 < x <= 4:
...         return 4
...     elif x < 0:
...         return 0
...
>>> f(2)
4
>>> f(5)
>>> f(10)

```

Why do we not get any output when calling `f(5)` and `f(10)`? (Hint: Save the `f` value in a variable `r` and write `print r`.) ◇

Exercise 2.51. *Compare two real numbers on a computer.*

Consider the following simple program inspired by Chapter 1.4.3:

```

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.

¹⁴ 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.

Thereafter, read the documentation of the function `float_eq` from SciTools: `scitools.numpyutils.float_eq` (see page 98 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 2.52. *Use None in keyword arguments.*

Consider the functions `L(x, n)` and `L2(x, epsilon)` from Chapter 2.2.6, whose program code is found in the file `lsum.py`. Let us make a more flexible function `L3` where we can either specify a tolerance `epsilon` or a number of terms `n` in the sum, and we can choose whether we want the sum to be returned or the sum and the number of terms. The latter set of return values is only meaningful with `epsilon` and not `n` is specified. The starting point for all this flexibility is to have some keyword arguments initialized to an “undefined” value that can be recognized:

```
def L3(x, n=None, epsilon=None, return_n=False):
```

You can test if `n` is given using the phrase¹⁵

```
if n is not None:
```

A similar construction can be used for `epsilon`. Print error messages for incompatible settings when `n` and `epsilon` are `None` (none given) or not `None` (both given). Name of program file: `L3_flexible.py`. ◇

Exercise 2.53. *Improve the program from Ch. 2.4.2.*

The `table` function in the program from Chapter 2.4.2 evaluates `y(t)` twice for the same value of the argument `t`. This waste of work has no practical consequences in this little program because the `y` function is so fast to calculate. However, mathematical computations soon lead to programs that takes minutes, hours, days, and even weeks to run. In those cases one should avoid repeating calculations. It is in general considered a good habit to make programs efficient, at least to remove obvious redundant calculations.

Write a new `table` function that has only one `y(t)` in the `while` loop. (Hint: Store the `y(t)` value in a variable.)

How can you make the `y` function more efficient by reducing the number of arithmetic operations? (Hint: Factorize `t` and precompute `0.5*g` in a global variable.) Name of program file: `ball_table_efficient.py`. ◇

Exercise 2.54. *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,

¹⁵ One can also apply `if n != None`, but the `is` operator is most common (it tests if `n` and `None` are identical objects, not just objects with equal contents).

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:

```
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. Name of program: `time_while.py`.
◇

Exercise 2.55. *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 2.12 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 and `if`-tests 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 2.56. *Find an error in a program.*

Consider the following program for computing

$$f(x) = e^{rx} \sin(mx) + e^{sx} \sin(nx),$$

```
def f(x, m, n, r, s):
    return expsin(x, r, m) + expsin(x, s, n)

x = 2.5
print f(x, 0.1, 0.2, 1, 1)

from math import exp, sin

def expsin(x, p, q):
    return exp(p*x)*sin(q*x)
```

Running this code results in

```
NameError: global name 'expsin' is not defined
```

What is the problem? Simulate the program flow by hand or use the debugger to step from line to line. Correct the program. ◇

Exercise 2.57. *Find programming errors.*

What is wrong in the following code segments? Try first to find the errors in each case by visual inspection of the code. Thereafter, type in the code snippet and test it out in an interactive Python shell.

```
def f(x)
    return 1+x**2;
```

Case 1:

```
def f(x):
    term1 = 1
    term2 = x**2
    return term1 + term2
```

Case 2:

```
def f(x, a, b):
    return a + b*x

print f(1), f(2), f(3)
```

Case 3:

```
def f(x, w):
    from math import sin
    return sin(w*x)

f = 'f(x, w)'
w = 10
x = 0.1
print f(x, w)
```

Case 4:

```
from math import *

def log(message):
    print message

print 'The logarithm of 1 is', log(1)
```

Case 5:

```
import time

def print_CPU_time():
    print 'CPU time so far in the program:', time.clock()

print_CPU_time;
```

Case 6:

◇

Exercise 2.58. *Simulate nested loops by hand.*

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

```
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 D.1, to step through the code and to see what happens.

◇

Exercise 2.59. *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.

◇

Exercise 2.60. *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]
```

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 do – to avoid getting frustrated by strange program behavior. ◇