Chapter 3
Loops and Lists

In this chapter, programming starts to become useful. The concepts introduced in the previous chapter are essential building blocks in all computer programs, but our example programs only performed a few calculations, which we could easily do with a regular calculator. In this chapter, we will introduce the concept of *loops*, which can be used to automate repetitive and tedious operations. Loops are used in most computer programs, and they look very similar across a wide range of programming languages. We will primarily use loops for calculations, but as you gain more experience, you will be able to automate other repetitive tasks. Two types of loops will be introduced in this chapter: the `while` loop and the `for` loop. Both will be used extensively in all subsequent chapters. In addition to the loop concept, we will introduce Boolean expressions, which are expressions with a true/false value, and a new variable type called a list, which is used to store sequences of data.

3.1 Loops for Automating Repetitive Tasks

To start with a motivating example, consider again the simple interest calculation formula;

$$A = P \cdot (1 + (r/100))^n.$$

In Chapter 2 we implemented this formula as a single-line Python program, but what if we want to generate a table showing how the invested amount grows with the years? For instance, we could write n and A in two columns like this

```
0    100
1    105
2    110
3    ...
...  ...
```


J. Sundnes, *Introduction to Scientific Programming with Python*, Simula SpringerBriefs on Computing 6,
https://doi.org/10.1007/978-3-030-50356-7_3

How can we make a program that writes such a table? We know from the previous chapter how to generate one line in the table:

```
P = 100
r = 5.0
n = 7
A = P * (1+r/100)**n
print(n,A)
```

We could then simply repeat these statements to write the complete program:

```
P =100; r = 5.0;
n=0;  A = P * (1+r/100)**n;  print(n,A)
n=1;  A = P * (1+r/100)**n;  print(n,A)
...
n=9;  A = P * (1+r/100)**n;  print(n,A)
n=10;  A = P * (1+r/100)**n;  print(n,A)
```

This is obviously not a very good solution, since it is very boring to write and errors are easily introduced in the code. As a general rule, when programming becomes repetitive and boring, there is usually a better way of solving the problem at hand. In this case, we will utilize one of the main strengths of computers: their strong ability to perform large numbers of simple and repetitive tasks. For this purpose, we use *loops*.

The most general loop in Python is called a while loop. A while loop will repeatedly execute a set of statements as long as a given condition is satisfied. The syntax of the while loop looks like the following:

```
while condition:
    <statement 1>
    <statement 2>
    ...
<first statement after loop>
```

The `condition` here is a Python expression that is evaluated as either true or false, which, in computer science terms, is called a Boolean expression. Notice also the indentation of all the statements that belong inside the loop. Indentation is the way Python groups code together in blocks. In a loop such as this one, all the lines we want to be repeated inside the loop must be indented, with exactly the same indentation. The loop ends when an unindented statement is encountered.

To make things a bit more concrete, let us use write a while loop to produce the investment growth table above. More precisely, the task we want to solve is the following: Given a range of years n from zero to 10, in steps of one year, calculate the corresponding amount and print both values to the screen. To write the correct while loop for solving a given task, we need to answer four key questions: (i) Where/how does the loop start, that is, what are the initial values of the variables; (ii) which statements should be repeated inside the loop; (iii) when does the loop stop, that is, what condition should become false to make the loop stop; and (iv) how should variables be updated for each

pass of the loop? Looking at the task definition above, we should be able to answer all of these questions: (i) The loop should start at zero years, so our initial condition should be `n = 0`; (ii) the statements to be repeated are the evaluation of the formula and the printing of `n` and `A`; (iii) we want the loop to stop when `n` reaches 10 years, so our `condition` becomes something like `n <= 10`; and (iv) we want to print the values for steps of one year, so we need to increase `n` by one for every pass of the loop. Inserting these details into the general while loop framework above yields the following code:

```
P = 100
r = 5.0
n = 0
while n <= 10:              # loop heading with condition
    A = P * (1+r/100)**n    # 1st statement inside loop
    print(n, A)             # 2nd statement inside loop
    n = n + 1               # last statement inside loop
```

The flow of this program is as follows:

1. First, `n` is 0, $0 \leq 10$ is true; therefore we enter the loop and execute the loop statements:
   - Compute `A`
   - Print `n` and `A`
   - Update `n` to 1
2. When we have reached the last line inside the loop, we return to the `while` line and evaluate $n \leq 10$ again. This condition is still true, and the loop statements are therefore executed again. A new `A` is computed and printed, and `n` is updated to the value of two.
3. We continue this way until `n` is updated from 10 to 11; now, when we return to evaluate $11 \leq 10$, the condition is false. The program then jumps straight to the first line after the loop, and the loop is finished.

Useful tip: A very common mistake in while loops is to forget to update the variables inside the loop, in this case forgetting the line `n = n + 1`. This error will lead to an infinite loop, which will keep printing the same line forever. If you run the program from the terminal window it can be stopped with `Ctrl+C`, so you can correct the mistake and re-run the program.

3.2 Boolean Expressions

An expression with a value of true or false is called a Boolean expression. Boolean expressions are essential in while loops and other important programming constructs, and they exist in most modern programming languages. We have seen a few examples already, including comparisons such as `a == 5` in

Chapter 2 and the condition `n <= 10` in the while loop above. Other examples of (mathematical) Boolean expressions are $t = 140$, $t \neq 140$, $t \geq 40$, $t > 40$, $t < 40$. In Python code, these are written as

```
t == 40  # note the double ==, t = 40 is an assignment!
t != 40
t >= 40
t >  40
t <  40
```

Notice the use of the double `==` when checking for equality. As we mentioned in Chapter 2 the single equality sign has a different meaning in Python (and many other programming languages) than we are used to from mathematics, since it is used to assign a value to a variable. Checking two variables for equality is a different operation, and to distinguish it from assignment, we use `==`. We can output the value of Boolean expressions with statements such as `print(C<40)` or in an interactive Python shell, as follows:

```
>>> C = 41
>>> C != 40
True
>>> C < 40
False
>>> C == 41
True
```

Most of the Boolean expressions we will use in this course are of the simple kind above, consisting of a single comparison that should be familiar from mathematics. However, we can combine multiple conditions using `and/or` to construct while loops such as these:

```
while condition1 and condition2:
    ...

while condition1 or condition2:
    ...
```

The rules for evaluating such compound expressions are as expected: `C1 and C2` is `True` if both `C1` and `C2` are `True`, while `C1 or C2` is `True` if at least one of the two conditions `C1` and `C2` is `True`. One can also negate a Boolean expression using the term `not`, which simply yields that `not C` is `True` if `C` is `False`, and vice versa. To gain a feel for compound Boolean expressions, you can go through the following examples by hand and predict the outcome, and then try to run the code to obtain the result:

```
x = 0;  y = 1.2
print(x >= 0 and y < 1)
print(x >= 0 or y < 1)
print(x > 0 or y > 1)
print(x > 0 or not y > 1)
print(-1 < x <= 0)   # same as -1 < x and x <= 0
print(not (x > 0 or y > 0))
```

Boolean expressions are important for controlling the flow of programs, both in while loops and in other constructs that we will introduce in Chapter 4. Their evaluation and use should be fairly familiar from mathematics, but it is always a good idea to explore fundamental concepts such as this by typing in a few examples in an interactive Python shell.

3.3 Using Lists to Store Sequences of Data

So far, we have used one variable to refer to one number (or string). Sometimes we naturally have a collection of numbers, such as the n-values (years) $0, 1, 2, \ldots, 10$ created in the example above. In some cases, such as the one above, we are simply interested in writing all the values to the screen, in which case using a single variable that is updated and printed for each pass of the loop works fine. However, sometimes we want to store a sequence of such variables, for instance, to process them further elsewhere in the program. We could, of course, use a separate variable for each value of `n`, as follows:

```
n0 = 0
n1 = 1
n2 = 2
...
n10 = 10
```

However, this is another example of programming that becomes extremely repetitive and boring, and there is obviously a better solution. In Python, the most flexible way to store such a sequence of variables is to use a list:

```
n = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Notice the square brackets and the commas separating the values, which is how we tell Python that `n` is a list variable. Now we have a single variable that can hold all the values we want. Python lists are not reserved just for numbers and can hold any kind of object, and even different kinds of objects. They also have a great deal of convenient built-in functionality, which makes them very flexible and useful and extremely popular in Python programs.

We will not cover all the aspects of lists and list operations in this book, but we will use some of the more basic ones. We have already seen how to initialize a list using square brackets and comma-separated values, such as

```
L1 = [-91, 'a string', 7.2, 0]
```

To retrieve individual elements from the list, we can use an index, for instance `L1[3]` will pick out the element with index 3, that is, the fourth element (having a value of zero) in the list, since the numbering starts at zero. List indices start at zero and run to the $n-1$, where n is the number of elements in the list:

```
mylist = [4, 6, -3.5]
print(mylist[0])
print(mylist[1])
print(mylist[2])
len(mylist)  # length of list
```

The last line uses the built-in Python function `len`, which returns the number of elements in the list. This function works on lists and any other object that has a natural length (e.g., strings), and is very useful.

Other built-in list operations allow us, for instance, to append an element to a list, add two lists together, check if a list contains a given element, and delete an element from a list:

```
n = [0, 1, 2, 3, 4, 5, 6, 7, 8]
n.append(9)   # add new element 9 at the end
print(n)
n = n  + [10, 11]     # extend n at the end
print(n)
print(9 in n)       #is the value 9 found in n? True/False
del n[0]            #remove the first item from the list
```

These list operations, in particular those to initialize, append to, and index a list, are extremely common in Python programs, and will be used throughout this book. It is a good idea to spend some time making sure you fully understand how they work.

It is also worth noting one important difference between lists and the simpler variable types we introduced in Chapter 2. For instance, two statements, such as `a = 2; b = a` would create two integer variables, both having value 2, but they are not the same variable. The second statement `b=a` will create a copy of `a` and assign it to `b`, and if we later change `b`, `a` will not be affected. With lists, the situation is different, as illustrated by the following example:

```
>>> l[0] = 2
>>> a = [1,2,3,4]
>>> b = a
>>> b[-1] = 6
>>> a
[1, 2, 3, 6]
```

Here, both `a` and `b` are lists, and when `b` changes `a` also changes. This happens because assigning a list to a new variable does not copy the original list, but instead creates a *reference* to the same list. So `a` and `b` are, in this case, just two variables pointing to the exact same list. If we actually want to create a copy of the original list, we need to state this explicitly with `b = a.copy()`.

3.4 Iterating Over a List with a for Loop

Having introduced lists, we are ready to look at the second type of loop we will use in this book: the for loop. The for loop is less general than the while loop, but it is also a bit simpler to use. The for loop simply iterates over elements in a list, and performs operations on each one:

```
for element in list:
    <statement 1>
    <statement 2>
    ...
<first statement after loop>
```

The key line is the first one, which will simply run through the list, element by element. For each pass of the loop, the single element is stored in the variable `element`, and the block of code inside the for loop typically involves calculations using this `element` variable. When the code lines in this block are completed, the loop moves on to the next element in the list, and continues in this manner until there are no more elements in the list. It is easy to see why this loop is simpler than the while loop, since no condition is needed to stop the loop and there is no need to update a variable inside the loop. The for loop will simply iterate over all the elements in a predefined list, and stop when there are no more elements. On the other hand, the for loop is slightly less flexible, since the list needs to predefined. The for loop is the best choice in most cases in which we know in advance how many times we want to perform a set of operations. In cases in which this number is not known, the while loop is usually the best choice.

For a concrete for loop example, we return to the investment growth example introduced above. To write a for loop for a given task, two key questions must be answered: (i) What should the list contain, and (ii) what operations should be performed on the elements in the list? In the present case, the natural answers are (i) the list should be a range of n-values from zero to 10, in steps of 1, and (ii) the operations to be repeated are the computation of `A` and the printing of the two values, essentially the same as in the while loop. The full program using a for loop thus becomes

```
years = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
r = 5.0
P = 100.0
for n in years:
    A = P * (1+r/100)**n
    print(n, A)
```

As with the while loop, the statements inside the loop must be indented. Simply by counting the lines of code in the two programs shows that the for loop is somewhat simpler and quicker to write than the while loop. Most people will argue that the overall structure of the program is also simpler and less error-prone, with no need to check a criterion to stop the loop or to

update any variables inside it. The for loop will simply iterate over a given list, perform the operations we want on each element, and then stop when it reaches the end of the list. Tasks of this kind are very common, and for loops are extensively used in Python programs.

The observant reader might notice that the way we defined the list `years` in the code above is not very scalable to long lists, and quickly becomes repetitive and boring. As stated above, when programming become repetitive and boring, a better solution usually exists. Such is the case here, and very rarely do values in a list need to be filled explicitly, as done here. Better alternatives include a built-in Python function called `range`, often in combination with a for loop or a so-called *list comprehension*. We will return to these tools later in the chapter. When running the code, one can also observe that the two columns of degrees values are not perfectly aligned, since `print` always uses the minimum amount of space to output the numbers. If we want the output in two nicely aligned columns, this is easily achieved by using the f-string formatting we introduced in the previous chapter. The resulting code can look like this:

```
years = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
for n in years:
    r = 5.0
    P = 100.0
    A = P * (1+r/100)**n
    print(f'{n:5d}{A:8.2f}')
```

Output is now nicely aligned:

```
  0  100.00
  1  105.00
  2  ...
```

A for loop can always be translated to a while loop. As described above, a while loop is more flexible than a for loop. A for loop can always be transformed into a while loop, but not all while loops can be expressed as for loops. A for loop always traverses traverses a list, carries out some processing on each element, and stops when it reaches the last one. This behavior is easy to mimic in a while loop using list indexing and the `len` function, which were both introduced above. A for loop of the form

```
for element in somelist:
    # process element
```

translates to the following while loop:

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

Using the function `range` to loop over indices. Sometimes we do not have a list, but want to repeat the same operation a given number of times. If we know the number of repetitions this task is an obvious candidate for a for loop, but for loops in Python always iterate over an existing list (or a list-like object). The solution is to use a built-in Python function named `range`, which returns a list of integers[1]:

```
P = 100
r = 5.0
N = 10
for n in range(N+1):
    A = P * (1+r/100)**n
    print(n,A)
```

Here we used `range` with a single argument $N+1$ which will generate a list of integers from zero to N (not including $N+1$). We can also use `range` with two or three arguments. The most general case `range(start, stop, inc)` generates a list of integers `start`, `start+inc`, `start+2*inc`, and so on up to, *but not including*, `stop`. When used with just a single argument, as above, this argument is treated as the `stop` value, and `range(stop)` is short for `range(0, stop, 1)`. With two arguments, the interpretation is `range(start,stop)`, short for `range(start,stop,1)`. This behavior, where a single function can be used with different numbers of arguments, is common both in Python and many other programming languages, and makes the use of such functions very flexible and efficient. If we want the most common behavior, we need only provide a single argument and the others are automatically set to default values; however, if we want something different, we can do so easily by including more arguments. We will use the `range` function in combination with for loops extensively through this book, and it is a good idea to spend some time becoming familiar with it. A good way to gain a feel for how the `range`-function works is to test statements such as `print(list(range(start,stop,inc)))` in an interactive Python shell, for different argument values.

Filling a list with values using a for loop. One motivation for introducing lists is to conveniently store a sequence of numbers as a single variable, for instance, for processing later in the program. However, in the code above, we did not really utilize this, since all we did was print the numbers to the screen, and the only list we created was a simple sequence from zero to 10. It could be more useful to store the amounts in a list, which can be easily be achieved with a for loop. The following code illustrates a very common way to fill lists with values in Python:

[1] In Python 3, `range` does not technically produce a list, but a list-like object called an iterator. In terms of use in a for loop, which is the most common use of `range`, there is no practical difference between a list and an iterator. However, if we try, for instance, `print(range(3))` the output does not look like a list. To obtain output that looks like a list, which can be useful for debugging, the iterator must be converted to an actual list: `print(list(range(3)))`.

```
P = 100
r = 5.0
N = 10
amounts = []                # start with empty list
for n in range(N+1):
    A = P*(1+r/100)**n
    amounts.append(A)       # add new element to amounts list
print(amounts)
```

The parts worth noting in this code are `amounts = []`, which simply creates a list with no elements, and the use of the `append` function inside the for loop to add elements to the list. This simple way of creating a list and filling it with values is very common in Python programs.

Mathematical sums are implemented as for loops. A very common example of a repetitive task in mathematics is the computation of a sum, for instance,

$$S = \sum_{i=1}^{N} i^2.$$

For large values of N such sums are tedious to calculate by hand, but they are very easy to program using `range` and a for loop:

```
N = 14
S = 0
for i in range(1, N+1):
    S += i**2
```

Notice the structure of this code, which is quite similar to the way we filled a list with values in the previous example. First, we initialize the summation variable (`S`) to zero, and then the terms of the sum are added one by one for each iteration of the for loop. The example shown here illustrates the standard recipe for implementing mathematical sums, which are common in scientific programming and appear frequently in this book. It is worthwhile spending some time to fully understand and remember how such sums is implemented.

How can we change the elements in a list? In some cases we want to change elements in a list. Consider first a simple example where we have a list of numbers, and want to add the value of two to all the numbers. Following the ideas introduced above, a natural approach is to use a for loop to traverse the list, as follows:

```
v = [-1, 1, 10]
for e in v:
    e = e + 2
print(v)
```

```
[-1, 1, 10]   # unaltered!!
```

As demonstrated by this small program, the result is not what we want. We added the value of two to every element, but after the loop finished, our list `v` was unchanged. The reason for this behavior is that although the list is traversed as desired when we create the for loop using `for e in v:`, the variable `e` is an ordinary (`int`) variable, and it is in fact a *copy* of each element in the list, and not the actual element. Therefore, when we change `e`, we change only the copy and not the actual list element. The copy is overwritten in the next pass of the loop anyway, so, in this case, all the numbers that are incremented by two are simply lost. The solution is to access the actual elements by indexing into the list:

```
v = [-1, 1, 10]
for i in range(len(v)):
    v[i] = v[i] + 2
print(v)
```

```
[1, 3, 12]
```

Notice in particular the use of `range(len(v))`, which is a common construction in Python programs. It creates a set of integers running from zero to `len(v)-1` that can be iterated over with the for loop and used to loop through all the elements in the list `v`.

List comprehensions for compact creation of lists. Above, we introduced one common way of constructing lists, which is to start with an empty list and use a for loop to fill it with values. We can extend this example to fill several lists in one loop, for instance, if we want to examine the effect of low and high interest rates on our bank deposit. We start with two empty lists and fill both with values in the same loop:

```
P = 100
r_low = 2.5
r_high = 5.0
N = 10
A_high = []
A_low = []
for n in range(N+1):
    A_low.append(P*(1+r_low/100)**n)
    A_high.append(P*(1+r_high/100)**n)
```

This approach to using a for loop to fill a list with values is so common in Python that a compact construct has been introduced, called a *list comprehension*. The code in the previous example can be replaced by the following:

```
P = 100
r_low = 2.5
r_high = 5.0
N = 10
A_low = [P*(1+r_low/100)**n for n in range(N+1)]
A_high = [P*(1+r_high/100)**n for n in range(N+1)]
```

The resulting lists `A_low` and `A_high` are exactly the same as those from the for loop, but the code is obviously much more compact. To an experienced Python programmer, the use of list comprehensions also makes the code more readable, since it becomes obvious that the code creates a list, and the contents of the list are usually easy to understand from the code inside the brackets. The general form of a list comprehension looks like

```
newlist = [expression for element in somelist]
```

where `expression` typically involves `element`. The list comprehension works exactly like a for loop; it runs through all the elements in `somelist`, stores a copy of each element in the variable `element`, evaluates `expression`, and appends the result to the list `newlist`. The resulting list `newlist` will have the same length as `somelist`, and its elements are given by `expression`. List comprehensions are important to know about, since you will see them frequently when reading Python code written by others. They are convenient to use for the programming tasks covered in this book, but not strictly necessary, since the same thing can always be accomplished with a regular for loop.

Traversing multiple lists simultaneously with `zip`. Sometimes we want to loop over two lists at the same time. For instance, consider printing out the contents of the `A_low` and `A_high` lists of the example above. We can accomplish this using `range` and list indexing, as in

```
for i in range(len(A_low)):
    print(A_low[i], A_high[i])
```

However, a built-in Python function named `zip` provides an alternative solution, which many consider more elegant and "Pythonic":

```
for low, high in zip(A_low, A_high):
    print(low, high)
```

The output is exactly the same, but the use of `zip` makes the for loop more similar to the way we traverse a single list. We run through both lists, extract the elements from each one into the variables `low` and `high`, and use these variables inside the loop, as we are used to. We can also use `zip` with three lists:

```
>>> l1 = [3, 6, 1];  l2 = [1.5, 1, 0];  l3 = [9.1, 3, 2]
>>> for e1, e2, e3 in zip(l1, l2, l3):
...     print(e1, e2, e3)
...
3 1.5 9.1
6 1 3
1 0 2
```

Lists traversed with `zip` typically have the same length, but the function also works for lists of different lengths. In this case, the for loop will simply stop when it reaches the end of the shortest list, and the remaining elements of the longer lists are not visited.

3.5 Nested Lists and List Slicing

As described above, lists in Python are quite general and can store *any* object, including another list. The resulting list of lists is often referred to as a *nested list*. Instead of storing the amounts resulting from the low and high interest rates above as two separate lists, we could put them together in a new list:

```
A_low = [P*(1+2.5/100)**n for n in range(11)]
A_high = [P*(1+5.0/100)**n for n in range(11)]

amounts = [A_low, A_high]  # list of two lists

print(amounts[0])    # the A_low list
print(amounts[1])    # the A_high list
print(amounts[1][2])  # the 3rd element in A_high
```

The indexing of nested lists illustrated here is quite logical, but can take some time getting used to. The important thing is that, if `amounts` is a list containing lists, then, for instance, `amounts[0]` is also a list and can be indexed in the way we are used to. Indexing into this list is done in the usual way, such that, for instance, `amounts[0][0]` is the first element of the first list contained in `amounts`. Playing a bit with indexing nested lists in the interactive Python shell is a useful exercise to understand how they are used.

Iterating over nested lists also works as expected. Consider, for instance, the following code

```
for sublist1 in somelist:
    for sublist2 in sublist1:
            for value in sublist2:
                # work with value
```

Here, `somelist` is a three-dimensional nested list, that is, its elements are lists, which, in turn, contain lists. The resulting nested for loop looks a bit complicated, but it follows exactly the same logic as the simpler for loops used above. When the outer loop starts, the first element from `somelist` is copied into the variable `sublist1`, and then we then enter the code block inside the loop, which is a new for loop that will start traversing `sublist1`, that is, first copying the first element into the variable `sublist2`. Then the process is repeated, with the innermost loop traversing all the elements of `sublist2`, copying each element into the variable `value`, and doing some calculations with this variable. When it reaches the end of `sublist2`, the innermost for loop is over, we "move outward" one level in terms of the loops, to the loop `for sublist2 in sublist`, which moves to the next element and starts a new run through the innermost loop.

Similar iterations over nested loops can be obtained by looping over the list indices, as follows:

```
for i1 in range(len(somelist)):
    for i2 in range(len(somelist[i1])):
```

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

Although their logic is the same as regular (one-dimensional) for loops, nested loops look more complicated and it can take some time to fully understand how they work. As noted above, a good way to obtain such understanding is to create some examples of small nested lists in a Python shell or a small Python program, and examine the results of indexing and looping over the lists. The following code is one such example. Try to step through this program by hand and predict the output before running the code and checking the result:

```
L = [[9, 7], [-1, 5, 6]]
for row in L:
    for column in row:
        print(column)
```

List slicing is used to extract parts of a list. We have seen how we can index a list to extract a single element, but sometimes it is useful to capture parts of a list, for instance, all the elements from an index n to an index m. Python offers *list slicing* for such tasks. For a list `A`, we have seen that a single element is extracted with `A[n]`, where `n` is an integer, but we can also use the more general syntax `A[start:stop:step]` to extract a *slice* of `A`. The arguments resemble those of the `range` function, and such a list slicing will extract all elements starting from index `start` up to but not including `stop`, with a step `step`. As for the `range` function, we can omit some of the arguments and rely on default values. The following examples illustrate the use of slicing:

```
>>> a = [2, 3.5, 8, 10]
>>> a[2:]   # from index 2 to end of list
[8, 10]

>>> a[1:3]  # from index 1 up to, but not incl., index 3
[3.5, 8]

>>> a[:3]   # from start up to, but not incl., index 3
[2, 3.5, 8]

>>> a[1:-1] # from index 1 to next last element
[3.5, 8]

>>> a[:]    # the whole list
[2, 3.5, 8, 10]
```

Note that these sublists (slices) are *copies* of the original list. A statement such as, for instance, `b = a[:]` will make a copy of the entire list `a`, and any subsequent changes to `b` will not change `a`. As for the nested lists considered above, a good way to become familiar with list slicing is to create a small

list in the interactive Python shell and explore the effect of various slicing operations. It is, of course ,possible to combine list slicing with nested lists, and the results can be confusing even to experienced Python programmers. Fortunately, we will consider only fairly simple cases of list slicing in this book, and we will work mostly with lists of one or two dimensions (i.e., non-nested lists or the simplest lists-of-lists).

3.6 Tuples

Lists are a flexible and user-friendly way to store sequences of numbers, and are used in nearly all Python programs. However, a few other data types are also made to store sequences of data. One of the most important ones is called a *tuple*, and it is essentially a constant list that cannot be changed. A tuple is defined in almost the same way as a list, but with normal parentheses instead of the square brackets. Alternatively, we can skip the parentheses and just use a comma-separated sequence of values to define a tuple. The following are two examples that are entirely equivalent and define the same tuple:

```
>>> t = (2, 4, 6, 'temp.pdf')    # define a tuple
>>> t =  2, 4, 6, 'temp.pdf'     # can skip parentheses
```

Tuples also provide much of the same functionality as lists, including indexing and and slicing:

```
>>> 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
```

However, tuples are *immutable*, which means that they cannot be changed. Therefore, some operations we are used to from lists will not work. Continuing the interactive session from above, the following are some examples of illegal tuple operations:

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

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

>>> del t[1]
```

```
...
TypeError: 'tuple' object doesn't support item deletion
```

The observant reader might wonder why the line `t = t + (-1.0, -2.0)` in the example above works, since `t` is supposed to be immutable and therefore impossible to change. The answer is related to the way assignment statements work in programming. As briefly explained in Chapter 2, assignment works by first evaluating the expression on the right hand side, which in this example means to add two tuples together. The result is a new tuple, and neither `t` nor `(-1.0, 2.0)` are changed in the process. Then, the new tuple is *assigned* to the variable `t`, meaning that the original tuple is replaced by the new and longer tuple. The tuple itself is never changed, but the contents of the variable `t` is replaced with a new one.

A natural question to ask, then, is why do we need tuples at all, when lists can do the same job and are much more flexible? The main reason for this is that, in many cases, it is convenient to work on item that is constant, since it is protected against accidental changes and can be used as a key in so-called *dictionaries*, an important Python datastructure that will be introduced in Chapter 7. Throughout this book, we will not do much explicit programming with tuples, but we will run into them as part of the modules we import and use, so it is important to know what they are.