Ch.2: Loops and lists

Joakim Sundnes[1,2]

[1]Simula Research Laboratory
[2]University of Oslo, Dept. of Informatics

Sep 2, 2019

In this chapter we will see the first examples of how we can create truly useful Python programs. The tools introduced in the previous chapter are essential building blocks in most mathematically oriented computer programs, but the programs only performed a few calculations, which we could easily do using 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 fundamental building blocks in computer programs, and they look very similar across a wide range of programming langugages. We will primarily use loops for calculations, but as you gain more experience you will be able to automate other repetitive tasks. The following key concepts will be introduced in this chapter:

- Using loops for repeating similar operations:
  - The `while` loop
  - The `for` loop
- Boolean expressions (True/False)
- Lists for storing sequences of data.

1 Loops for automating repetitive tasks

To start with a motivating example, think of the simple mathematical formula converting Celcius to Fahrenheit degrees

$$F = (9/5) \cdot C + 32.$$

We can easily program this equation as a single-line Python program, as we did with the formulas in the previous chapter, but say we want to make a table of Celsius and Fahrenheit degrees:

```
-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
```

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

```
C = -20
F = 9.0/5*C + 32
print(C, F)
```

and we could simply repeat these statements to write the complete program:

```
C = -20;  F = 9.0/5*C + 32;  print(C, F)
C = -15;  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)
```

This is obviously not a very good solution, as it is very boring to write, and it is easy to introduce errors 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 strenghts of computers, that they are extremely good at performing a large number of simple and repetitive tasks. For this purpose we use *loops*.

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

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

Here, `condition` is a Python expression that evaluates to 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 like this, all the lines we want to be repeated inside the loop must be indented, and 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 the while-loop to produce the Celsius-Fahrenheit table above. More precisely, the task we want to solve is the following: Given a range of Celsius degrees from -20 to 40, in steps of 5, calculate the corresponding degrees Fahrenheit 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, i.e., 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: (i) The loop should start with -20 Celsius degrees, so our initial value should be `C = -20`, (ii) the statements to be repeated are the calculation of the Fahrenheit degrees and the printing of `F` and `C`, (iii) we want the loop to stop when `C` reaches 40 degrees, so our `condition` becomes something like `C <= 40`, and (iv) we want to print the values for steps of 5 degrees Celsius, so we need to increase `C` with 5 for every pass of the loop. Inserting these details into the general while-loop framework above yields the following code:

```python
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              # last statement inside loop
```

The flow of this program is as follows:

- First `C` is -20, $-20 \leq 40$ is true, therefore we execute the loop statements
- Compute `F`, print, and update `C` to -15
- Since we have reached the last line inside the loop, we jump back up to the `while` line, evaluate $C \leq 40$ again, which is still true, and hence we enter a new round in the loop
- We continue this way until `C` is updated to 45
- Now the loop condition $45 \leq 40$ is false, and the program jumps to the first line after the loop - the loop is over

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 `C = C + dC`. This error will lead to an eternal loop, which repeats 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.

2 Boolean expressions

An expression with value 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. Examples of (mathematical) boolean expressions are $C = 40$, $C \neq 40$, $C \geq 40$, $C > 40$, $C < 40$. In Python code, these are written as

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

Notice the use of the double `==` when checking for equality. As we saw in the last chapter the single equality sign has a slightly different meaning in Python (and many other programming langugages) than we are used to from mathematics, since it is used for assigning 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 by statements like `print(C<40)` or in an interactive Python shell:

```
>>> 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 is probably well-known 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 evalating such compound expressions are as you would expect: `C1 and C2` is `True` if both `C1` and `C2` are `True`, while `C1 or C2` is `True` if at least one of `C1` or `C2` is `True`. One can also negate a boolean expression using the word `not`, which simply gives that `not C` is `True` if `C` is `False`, and vice versa. To get a feel for compound boolean expressions, go through the following examples by hand and predict the outcome, and then run the code to get 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 the next chapter. Their evaluation and use should be fairly familiar from mathematics, but it is always a good idea to explore fundamental concepts like this by typing in a few examples in an interactive Python shell.

3 Using lists to store sequences of objects

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 Celsius degrees $-20, -15, -10, -5, 0, \ldots, 40$ created in the example above. In some cases, like the one above, we are simply interested in writing out all the values to the screen, and in this case it works fine to use a single variable that we update and print for each pass of the loop. However, often we want to store such a sequence of variables, for instance to process it further elsewhere in our program. We could of course use a separate variable for each celsius value:

```
C1 = -20
C2 = -15
C3 = -10
...
C13 = 40
```

However, this is another example of programming becoming extremely repetive 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:

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

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

We will not cover all aspects of lists and list operations in this course, but some of the more basic ones will be used frequently. 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, i.e. the fourth element (having value 0) in the list since the numbering starts at 0. List indices start at 0 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 builtin Python function `len`, which returns the number of element in the list. This function works on lists and any other object that has a natural length (for instance strings), and is very useful.

Other built-in list operations allow us to append an element to a list and to add two lists together:

```
C = [-10, -5, 0, 5, 10, 15, 20, 25, 30]
C.append(35)   # add new element 35 at the end
print(C)
C = C + [40, 45]      # extend C at the end
print(C)
print(len(C))               # length of list
```

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

4 Iterating over a list with a for-loop

Having introduced lists, we are ready to look at the second type of loop that will be used in this course; 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:

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

The key line here 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 some 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 way 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 conditional 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 pre-defined 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 pre-defined. The for-loop is the best choice in most cases where we know in advance how many times we want to pass through a list. For cases where this number is not known, the while-loop is usually the best choice.

To make a concrete for-loop example, we return to the task considered above of writing out the temperature conversion table. To write a for-loop for a given task, the two key questions to answer (similar to the four questions for the while loop above) are: (i) How should the list be initiated, and (ii) what operations should be performed on all elements in the list? For the present case, the natural answers are (i) the list should be a range of Celsius values from -20 to 40, in steps of 5, and (ii) the operations to be repeated are the computation of `F` and the printing of the two values, essentially the same as in the while loop. The full program using a for-loop becomes

```
degrees = [-20, -15, -10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40]
for C in degrees:
    F = 9/5.*C + 32
    print(C, F)
```

As with the while-loop, the statements inside the loop must be indented. Simply by counting the lines of code in the two programs we get an indication 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 for checking 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 may notice that the definition of the list `degrees` in the code above is not very scalable to long lists, and quickly becomes repetitive and boring. And as stated above, when programming become repetitive and boring there usually exists a better solution. So also in this case, and we will very rarely fill values into a list explicitly like we have done here. A better alternative is to use a for-loop to fill the list, and Python also offers a convenient solution known as a `list comprehension`, which we will get to later. When running the code, one may also observe that the two columns of degrees values are not nicely aligned, since `print` always uses the minimum amount of space to output the numbers. This problem is easily fixed by using the f-string formatting we introduced in Chapter 1. The resulting code may look like this:

```
degrees = [-20, -15, -10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40]
for C in degrees:
    F = 9/5.*C + 32
    print(f'{C:5d}{F:5.2f}')
```

Output is now nicely aligned:

```
-20  -4.0
-15   5.0
-10  14.0
 -5  23.0
  0  32.0
  ......
 35  95.0
 40 104.0
```

A for-loop can always be translated to a while loop. As described above, the while loop is more flexible than a for-loop. In fact, a for-loop can always be transformed into a while-loop, but not all while loops can be expressed as for loops. For-loops always traverse through lists, do some processing of each element, and stop when they reach 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 on this 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
```

Filling a list with values using a for-loop. One motivation for introducing lists was to conveniently store a sequence of numbers as a single variable, for instance if we want to process it 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 only the Celsius values were stored as a list. If we want to store the numbers for later processing, it would make sense to have the Fahrenheit degrees in a list as well. This can easily be achieved with a for-loop, and the resulting code illustrates a very common way to fill lists with values in Python:

```
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)   # add new element to Fdegrees
print(Fdegrees)
```

`print(Fdegrees)` results in

```
[-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]
```

The parts worth noticing in this code are `Fdegrees = []`, 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 is a convenient and very commonly used way of filling a Python list with values.

Using the function `range` to loop over indices. Sometimes we don't have a list, but want to repeat an operation N times. Since we know the number of repetitions this 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 builtin Python function `range`, which returns a list of integers[1]:

```
C = 0
for i in range(N):
    F = (9.0/5)*C + 32
    C += 10
    print(C, F)
```

Here we used `range` with a single argument (N, the number of repetitions), but `range` can be used with one, two, or three arguments. The most general

[1]In Python 3, `range` does not technically produce a list, but a list-like object called an iterator. For 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 get output that looks like a list, which may be useful for debugging, the iterator must be converted to an actual list: `print(list(range(3)))`.

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, like above, this argument is treated as the `stop` value, and `range(stop)` becomes 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 only need to provide a single argument, and the others are automatically set to default values, but if we want something different this is easily obtained by including more argument. We will use the `range`-function in combination with for-loops extensively through this course, and it is a good idea to spend some time getting familiar with it. Testing statements like `print(list(range(start,stop, inc)))` in an interactive Python shell, for different argument values, is a good way to get a feel for how the `range`-function works.

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. First we initialize the summation variable (`S`) to zero, and then the terms 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 appear frequently in this course and later. It is worthwhile spending some time to fully understand and remember how they are 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 2 to all the numbers. Following the ideas introduced above, a natural approach is to use a for-loop to traverse the list:

```
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 2 to every element, but after the loop finished our list `v` remained unchanged. The reason for this behavior is that when we created the for-loop using `for e in v:`, the list is traversed as we want, but the variable `e` is an ordinary (`int`) variable, and is in fact a *copy* of each element in the list, and not the actual element. So when we change `e`, we only change the copy and not the actual list element. The copy is over-written in the next pass of the loop anyway, so in this case all the numbers that we increment with 2 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 very common construction to see in Python programs. It creates a set of integers running from 0 to `len(v)-1`, which 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 to construct lists, which was to start with an empty list and use a for-loop to fill the list with values. In this example we do the same, but start with two empty lists and fill both in the for-loop:

```
n = 16
Cdegrees = [];  Fdegrees = []  # empty lists

for i in range(n):
    Cdegrees.append(-5 + i*0.5)
    Fdegrees.append((9.0/5)*Cdegrees[i] + 32)
```

This way to construct the `Cdegrees` list is obviously more convenient than the method used above, in particular for longer lists. In fact, the use of a for-loop to fill values in a list 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:

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

The resulting lists `Cdegrees` and `Fdegrees` are exactly the same as we got 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 is 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 given by `expression`. List comprehensions are important to know about, since you will see them frequently when reading Python code written by others. For the programming tasks IN1900 they are convenient to use, but not strictly necessary, since you can always accomplish the same thing 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 `Cdegrees` and `Fdegrees` lists above. We can do this using `range` and list indexing, as in

```
for i in range(len(Cdegrees)):
    print(Cdegrees[i], Fdegrees[i])
```

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

```
for C, F in zip(Cdegrees, Fdegrees):
    print(C, F)
```

The output is exactly the same, but the use of `zip` makes the for-loop more similar to how we traverse a single list. We run through both lists, extract the elements from each into the variables `C` and `F`, and use these variables inside the loop like 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 we traverse with `zip` typically have the same length, but the function actually works also for lists of different length. 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.

4.1 Nested lists: list of lists

As described above, lists in Python are quite general and can store *any* object, incuding another list. The resulting list of lists is usually referred to as a *nested list*. For the Celsius and Fahrenheit degrees above, instead of storing the table as two separate lists (one for each column), we could stick the two lists together in a new list:

```
Cdegrees = list(range(-20, 41, 5)) #range returns an iterator, convert to a list
Fdegrees = [(9.0/5)*C + 32 for C in Cdegrees]

table1 = [Cdegrees, Fdegrees]  # list of two lists

print(table1[0])    # the Cdegrees list
print(table1[1])    # the Fdegrees list
print(table1[1][2])  # the 3rd element in Fdegrees
```

The indexing of nested lists illustrated here is quite logical, but may take some time getting used to. The important thing to consider is that if `table1` is a list containing lists, then for instance `table1[0]` is also a list and can be indexed as we are used to. Indexing into this list is done in the usual way, so for instance `table1[0][0]` is the first element of the first list contained in `table`. 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 we would expect, 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, i.e. 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 we used above. When the "outer" loop starts, the first element from `somelist` is copied into the variable `sublist1`, and then we enter the code block inside the loop, which is a new for-loop that will start traversing `sublist1`, i.e. first coying the first element into the variable `sublist2`. Then the process is repeated, the innermost loop traverses all the elements of `sublist2`, copies each element into the variable `value`, and does some calculations with this variable. When it reaches the end of `sublist2`, the innermost for-loop is over, we move "out" one level in the loops, the loop `for sublist2 in sublist` moves to the next element and starts a new iteration through the innermost loop.

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

```
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 the logic is the same as regular (one-dimensional) for loops, nested loops look more complicated and it may 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, 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 for extracting parts of a list. We have seen how we can index a list to extract a single element, but sometimes it is useful to grab parts of a list, for instance all elements from an index n to index m. Python offers so-called *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`, and with 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. In contrast to regular indexing, where we could actually access and change a single elements, the slices are copies and changing them will not affect the original lists. As for the nested lists considered above, a good way to get 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 may be confusing even to experienced Python programmers. Fortunately, we will only consider fairly simple cases of list slicing in this course, and we will mostly work with lists of one or two dimensions (i.e. non-nested lists or the simplest lists-of-lists).

4.2 Tuples

Lists are a flexible and user friendly way to store sequences of numbers, and are used in nearly all Python programs. However, there are also a few other data types that are made for storing sequences of data. One of the most important ones is called a *tuple*, and is essentially a constant list that cannot be changed:

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

Tuples can do much of what lists can do:

```
>>> 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
```

A natural question to ask is why we need tuples when lists can do the same job and are much more flexible. There are several reasons for using tuples:

- Tuples are constant and thus protected against accidental changes
- Tuples are faster than lists
- Tuples are widely used in Python software
  (so you need to know about them!)
- Tuples (but not lists) can be used as keys is dictionaries
  (more about dictionaries later)

We will not program much with tuples as part of this course, but we will run into them as part of modules we import and use, so it is important to know what they are.

5 Summary of loops, lists and tuples

Key topics covered in this chapter:

- While loops
- Boolean expressions
- For loops
- Lists
- Nested lists

- Tuples

While loops and for loops are used to repeat similar operations many times. Their general syntax looks as follows:

```
while condition:
    <block of statements>

for element in somelist:
    <block of statements>
```

The while-loop is the most general, but the for-loop is a bit simpler to use. The rule of thumb is that whenever the number of iterations in the loop is known in advance, it is best to use a for-loop. When this number is not known, for instance if we want to keep updating a sum until some convergence criterion is met, we need to use a while-loop.

Lists and tuples ar used to store sequences of values. They are essentially the same, with the important exception that tuples are immutable, i.e. they cannot be changed after they are created. We will mostly use lists in this course, but it is important to know about both.

```
mylist  = ['a string', 2.5, 6, 'another string']
mytuple = ('a string', 2.5, 6, 'another string')
mylist[1]  = -10
mylist.append('a third string')
mytuple[1] = -10  # illegal: cannot change a tuple
```

Important list operations worth remembering. The table below summarizes some important list operations. It is a good idea to ensure that you understand and remember all of these, for instance by testing the statements in an interactive Python shell.

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