Chapter 4

Loop structures

1. Introduction

Suppose we wish to print a message 5 times. This is done using the following algorithm.

algo
```
1 print(“I print a message”)
2 print(“I print a message”)
3 print(“I print a message”)
4 print(“I print a message”)
5 print(“I print a message”)
```

We can see that it is sufficient to print the statement 5 times. It is thus possible to create a program that outputs the message as many times as desired just by copying and pasting the print statement (though this is not recommended).

Now, what if we want the program to print the message as many times as the user defines? In this situation, how many times should we use the print statement? To deal with this issue, loops are used. A loop is a way for executing a statement or block as many times as necessary without having to write it down each time. Each time that the block is executed is called an *iteration*. Loops are an excellent way to creating powerful programs, but they introduce a serious problem regarding execution termination. A loop has the ability to be infinite, that is, it never ends. More about that issue will be discussed in this chapter.

Programming languages offer a variety of loops. Some are straightforward, whereas others show a looping behavior indirectly. We distinguish two kinds of loops: condition-based loops (`while`) and index-based loops (`for`).

2. Simple `while` loops

A while loop executes a code as long a given condition is satisfied. In the algorithmic language its form is:

algo
```
1 while expression do block
```

In Python, we use the following syntax:

py
```
1 while expression:
2 |  block
```

In both languages, the execution of the `while` is made as follows:

1 – Evaluate `expression`.

2 – If the value of `expression` is `true` (or any equivalent value in Python), then the `block` is executed; otherwise, the execution resumes with the statement following the `while`.

3 – After the execution of `block`, the `expression` is evaluated again, and steps 2 and 3 are repeated until the value of `expression` becomes false.

Let us consider an example using the `while` loop. The factorial is defined by $n! = \prod_{i=1}^{i=n} i = n \times n-1 \times ...2 \times 1$, with $0! = 1$. The following code gives the algorithm to compute the factorial of a number:

algo

```
1  algorithm factorial
2  var i,n,res:integer
3  begin
4      n=integer(input("Give the value of n="))
5      res=1
6      i=2
7      while i<=n do
8          begin
9          res=res*i
10         i=i+1
11         end
12     print(n,"!=",res)
13 end
```

This is written as follows in Python:

py

```
1 if __name__=="__main__":
2   n:int=int(input("Give the value of n="))
3   i:int=2
4   res:int=1
5   while i<=n:
6     res*=i
7     i+=1
8   print(n,"!=",res)
```

To understand how `while` works, we will go through it step by step while keeping an eye (watching) on some expressions and/or variables (we use the Python code here). Suppose that user enters the value 5 for n. The following table gives the values after the execution of each line.

Line	n	i	res	i<=n
3	5	2	-	True
4	5	2	1	True
5	5	2	1	True
5	5	2	1	True
6	5	2	2	True
7	5	3	2	True
5	5	3	2	True
6	5	3	6	True
7	5	4	6	True
5	5	4	6	True
6	5	4	24	True
7	5	5	24	True
5	5	5	24	True
6	5	5	120	True
7	5	6	120	True

5	5	6	120	False

We can see that the code terminates. There were 4 iterations. Notice that the loop terminates because it was possible to reach a state in which the condition `i<=n` is no longer true. In other words, the negation of the loop condition ($\neg i \leq n \Leftrightarrow i > n$) becomes true. We call this last condition the *stop condition*.

What if the `while` stop condition never becomes true? This means that the loop will run forever. We call that an *infinite loop*. In most cases, this is a fatal error, and it should be avoided by fine-tuning the stop condition and/or the block statements. For instance, in the previous code, if we omit the statement `i+=1`, the loop will run forever (or until the user stops the execution). However, in some special cases (for example, in a server), infinite loops are intentionally used.

Remark: In this code, the variable `i` somehow counts the number of iterations. We usually refer to such variables as counters.

2.1. Controlling the loop

Beside the loop condition and the loop statements, it is possible to control the execution of iterations. In Python, two special statements are used:

- The `break` statement: this statement ends the loop and transfers control to the statement that follows it. This statement is particularly helpful when the stop condition is too complex to compute or some of the variable values are unknown.
- The `continue` statement: assume that a statement has to be executed in all iterations except one. The `continue` statement allows to *skip* the statement of the loop and go back to the loop condition evaluation.

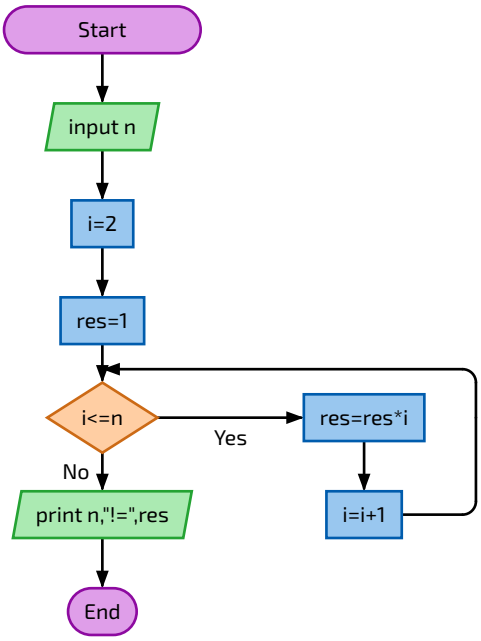
Although the following program is not recommended, we use it as an illustration of the `break` statement:

py

```
if __name__=="__main__":
    n:int=int(input("Give the value of n="))
    i:int=2
    res:int=1
    while True:
        res*=i
        if i<6:i+=1
        else:break
    print(n,"!=",res)
```

2.2. Loops and flowcharts

There is no specific box to represent loops. Instead, they are represented in the flowchart as a cycle: a path that starts from one box and returns to the same box after multiple steps. The first box is usually a test box. To illustrate this idea, we give a flowchart of the factorial program.



Example 4.1 : Fibonacci series

The Fibonacci series x_n is defined by $u_0 = u_1 = 1$ and the recurrence relation $u_{n+1} = u_n + u_{n-1}$ (for $n > 0$). Write an algorithm and a Python program to compute the term u_n (n is given by the user).

Example 4.2 : Let's play a game

In a game, a player must guess a number chosen randomly by the computer (between 1 and 200). When the player makes a guess, the computer tells him if it is greater or less than the chosen number. The game ends when the player gives the right answer.

Give the algorithm, the Python program and the flowchart of the game.

To generate a random number, first place the line `import random` at the beginning of the file. Then, use the function `random.randint(1,200)`.

3. Index-based loops

`while` loops are useful when the number of iterations is not known in advance. In contrast, when the iteration number is known, it is preferable to use another kind of loop called index-bases loops, also known as `for` loops. The first advantage of `for` loops is that the programmer would not forget to increment the loop counter. In addition, counters in `for` loops can be incremented by 1 (this is the default case) or by any value (positive or negative), giving the loop a more concise form.

In the algorithmic language, `for` loops has the following form:

algo
```
1 for i=a to b [step n] do block
```

In this code, the part `step n` is *optional*. If omitted, the default value of the step is 1. The code executes as follows:

1 – The expression `a` is evaluated, then its value is assigned to `i`

2 – If the step is positive, the test `i<=b` is evaluated; otherwise, the test `i>=b` is evaluated

3 – If the test value is false, then stop the execution of the loop

4 – Else, execute `block` then assign `i+n` to `i`. The execution goes back to step 2.

The factorial algorithm cas be rewritten as:

algo
```
algorithm factorial
var i,n,res:integer
begin
    n=integer(input(“Give the value of n=”))
    res=1
    for i=2 to n do
        res=res*i
    print(n,"!=",res)
end
```

Remark: The loop counter is automatically initialized and incremented (or decremented at the end of each iteration). Although manually changing the value of the counter within the loop is not forbidden, this is not a recommended practice. If doing so is necessary, the `while` loop will be more suitable.

Python `for` loops are by far more powerful than those in the algorithmic language. Some aspects of `for` loops are beyond the scope of this chapter, but keep in mind that `for` loop is a means to iterate over iterables. An iterable is any data structure that can be browsed sequentially. In this chapter, we will just use the case of iterating over a list of integers through the function `range` which takes one, two or three arguments:

- If one argument is used, then `range(n)` represents integers from 0 to $n-1$ (if $n \leq 0$, then the list is empty). The step value is 1.
- If two arguments are used, then `range(n,p)` represents integers from n to $p-1$. The step value is 1.
- If three arguments are used, then `range(n,p,s)` represents integers from n to $p-1$ with step s. If the step is negative, then the function represents integers from n down to $p+1$ with step s.

The factorial program is rewritten as:

py
```
if __name__=="__main__":
    n:int=int(input("Give the value of n="))
    res:int=1
    i:int
    for i in range(2,n+1):
        res*=i
    print(n,"!=",res)
```

Python `for` loops can involve many variables using the unpack syntax, but we won't use that for the moment. Python also has other means to loop over iterables, notably through *comprehension* or *filtering*. Again, these topics will not be covered in this chapter.

3.1. Nested loops

The inner block of a loop statement may also include another loop. Hence, we talk about *nested loops*. In this case, the inner loop is executed in each iteration of the outer loop. If the outer loop executes for n iterations and the inner iteration executes for p iterations each time, the inner loop's inner block will be executed $n \times p$ times, this is known as the *complexity* of the loop. Obviously, we can nest any number of loops within one another but the complexity will increase as many levels are introduced.

In nested loops, if counters are used, then each loop will generally have its own counter.

We will illustrate nested loops through a simple example. Let's consider that we want to compute the factorial of numbers from 1 to 30. In the algorithmic language, this is done by this code:

algo

```
algorithm factorials
var i,n,res:integer
begin
    for n=1 to 30
    begin
        res=1
        for i=2 to n do
            res=res*i
        print(n,"!=",res)
    end
end
```

The Python code for this problem is:

py

```
if __name__=="__main__":
  n:int
  for n in range(1,31):
    res:int=1
    i:int
    for i in range(2,n+1):
      res*=i
    print(n,"!=",res)
```

Example 4.3 : A chessboard

Write a Python program that draws a chessboard on the screen. Black squares will be noted by `[#]` and the white square by `[ ]`.

4. A word about efficiency

As previously stated, simple or nested loops introduces complexity to an algorithm. The complexity of a code is roughly defined as the number of operations executed by the algorithm. In general, we're interested in the relationship between an algorithm's number of inputs and the number of operations performed. Obviously, no one likes it when a computer hangs and takes a long time to run programs, but could we avoid it? The answer is somewhat difficult because it depends on the problem being solved.

We say that an algorithm A_1 is less complex than an algorithm A_2 for solving the same problem if the first executes fewer operations. We may even go a step further by looking for the least complex algorithm for solving the problem. It is always crucial to look for optimal algorithms, even if it is a difficult task. In some cases, we're not even sure if optimal algorithms exist. In this case, we can accept *good* algorithms: suboptimal algorithms that are easier to build while still providing accepted runtime performance.