Chapter 3: Control Structures

1. Higher order organization of Python instructions

In the previous chapters, we have introduced the different types of variables known by Python, as well as the operators that manipulate these variables. The programs we have studied so far have all been sequential, with each line corresponding to one instruction: this is definitely not optimal. For example, we have introduced in the previous chapter the concept of lists and arrays, to avoid having to use many scalar variables to store data (remember that if we were to store the whole human genome, we would need either 30,000 scalar variables, one for each gene, or a single array, whose items are the individual genes); if we wanted to perform the same operation on each of these genes, we would still have to write one line for each gene. In addition, the programs we have written so far would attempt to perform all their instructions, once given the input. Again, this is not always desired: we may want to perform some instructions only if a certain condition is satisfied.
Again, Python has thought about these issues, and offers solutions in the form of control structures: the if structure that allows to control if a block of instruction need to be executed, and the for structure (and equivalent), that repeats a set of instructions for a preset number of times. In this chapter, we will look in details on the syntax and usage of these two structures.

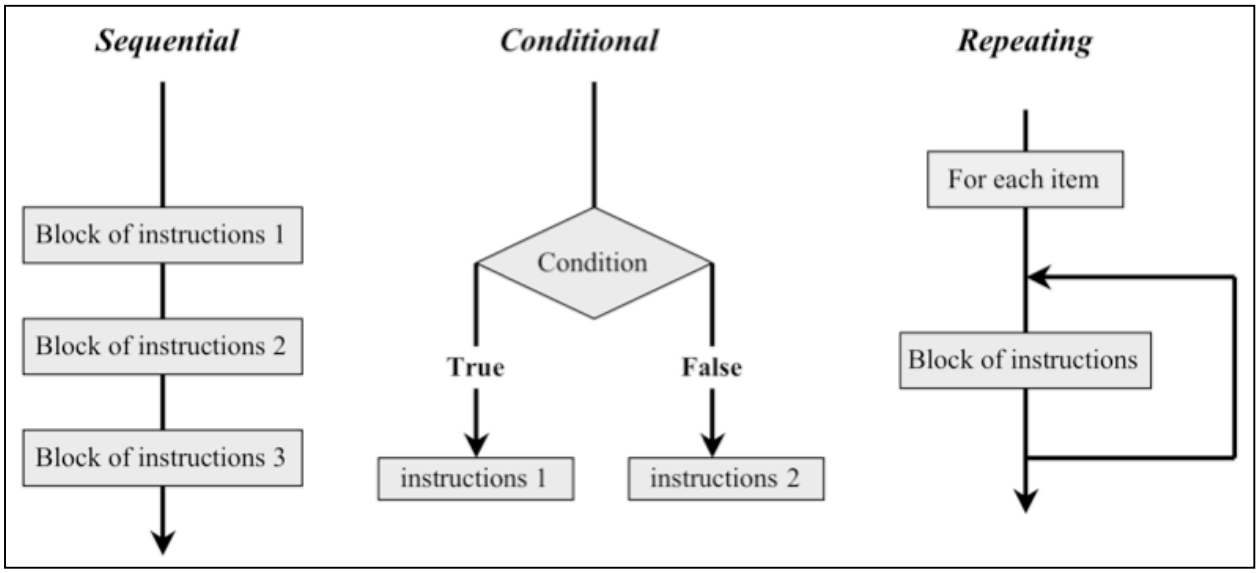

***Figure 3.1**: The three main types of flow in a computer program: **sequential**, in which instructions are executed successively, **conditional**, in which the blocks "instructions 1" and "instructions 2" are executed if the Condition is True or False, respectively, and **repeating**, in which instructions are repeated over a whole list.*

2. Logical operators

Most of the control structure we will see in this chapter test if a condition is true or false. For programmers, "truth" is easier to define in terms of what is not truth! In Python, there is a short, specific list of false values:

- An empty string, " ", is false
- The number zero and the string "0" are both false.
- An empty list, (), is false.
- The singleton None (i.e. no value) is false.

Everything else is true.

2.1 Comparing numbers and strings

We can test whether a number is bigger, smaller, or the same as another. Similarly, we can test if a string comes before or after another string, based on the alphabetical order. All the results of these tests are TRUE or FALSE. Table 3.1 lists the common comparison operators available in Python.

Notice that the numeric operators look a little different from what we have learned in Math: this is because Python does not use the fancy fonts available in text editors, so symbols like ≠, ≤, ≥ do not exist. Notice also that the numeric comparison for equality uses two = symbols (==): this is because the single = is reserved for assignment.

Table 3.1 : Python comparison operators

comparison	Corresponding question
a == b	Is a equal to b ?
a != b	Is a not equal to b ?
a > b	Is a greater than b ?
a >= b	Is a greater than or equal to b ?
a < b	Is a less than b ?
a <= b	Is a less than or equal to b ?
a in b	Is the value a in the list (or tuple) b?
a not in b	Is the value a not in the list (or tuple) b?

These comparisons apply both to numeric value and to strings. Note that you can compare numbers to strings, but the result can be arbitrary: I would strongly advise to make sure that the types of the variables that are compared are the same!

2.2 Combining logical operators

We can join together several tests into one, by the use of the logical operator **and** and **or**

a and **b**	True if both **a** and **b** are true.
a or **b**	True if either **a**, or **b**, or both are true.
not **a**	True if **a** is false.

3. Conditional structures

3.1 If

The most fundamental control structure is the if structure. It is used to protect a block of code that only needs to be executed if a prior condition is met (i.e. is TRUE). The generate format of an if statement is:

```
>>> if condition:
        code block
```

Note about the format:

- the : indicates the end of the condition, and flags the beginning of the end structure
- Notice that the code block is indented with respect to the rest of the code: this is required; it allows you to clearly identify which part of the code is conditional to the current condition.

The condition is one of the logical expressions we have seen in the previous section. The code block is a set of instructions grouped together. The code block is only executed if the condition is TRUE.

if statements are very useful to check inputs from users, to check if a variable contains 0 before it is used in a division,….

As example, let us write a small program that asks the user to enter a number, reads it in, and checks if it is divisible by n, where n is also read in:

```
number=int(raw_input("Enter your number  -->  "))
n=int(raw_input("Test divisibility by -->  "))
i=number%n
if i != 0:
        print "The number ",number," is not divisible by ",n,"\n"
```

3.2 Else

When making a choice, sometimes you have two different things you want to do, depending upon the outcome of the conditional. This is done using an if …else structure that has the following format:

```
if  condition:
        block code 1
else:
        block code 2
```

Block code 1 is executed if the condition is true, and block code 2 is executed otherwise. Here is an example of a program asking for a password, and comparing it with a pre-stored string:

```
hidden=”Mypasscode”
password=raw_input(“Enter your password : “)
if password == hidden:
        print “You entered the right password\n”
else:
        print “Wrong password !!\n”;
```

Python also provides a control structure when there are more than two choices: the elif structure is a combination of else and if. It is written as:

```
if  CONDITION1:
        block code 1
elif  CONDITION2:
        block code 2
else :
        block code 3
```

Note that any numbers of elif can follow an if.

4. Loops

One of the most obvious things to do with an array is to apply a code block to every item in the array: loops allow you to do that.
Every loop has three main parts:

- An entry condition that starts the loop
- The code block that serves as the “body” of the loop
- An exit condition

Obviously, all three are important. Without the entry condition, the loop won't be executed; a loop without body won't do any thing; and finally, without a proper exit condition, the program will never exit the loop (this leads to what is referred to an infinite loop, and often results from a bug in the exit loop).
There are two types of loops: ***determinate*** and ***indeterminate***. Determinate loops carry their end condition with them from the beginning, and repeat its code block an exact number of times. Indeterminate loops rely upon code within the body of the loop to alter the exit condition so the loop can exit. We will see one determinate loop structure, ***for***, and one indeterminate loop structure, ***while***.

4.1 For loop

The most basic type of determinate loop is the **for** loop. Its basic structure is:

```
for variable in listA:
        code block
```

Note the syntax similar to the syntax of an if statement: the : at the end of the condition, and the indentation of the code block.

A for loop is simple: it loops over all possible values of variable as found in the list listA, executing each time the code block.

For example, the Python program:

```
>>> names=["John","Jane","Smith"]
>>> j=0
>>> for name in names:
        j+=1
        print "The name number ",j," in the list is ",name
```

Will print out:

```
The name number 1 in the list is 'John'
The name number 2 in the list is 'Jane'
The name number 3 in the list is 'Smith'
```

Note that if the list is empty the loop is not executed at all.

The **for** loop is very useful for iterating over the elements of an array. It can also be used to loop over a set of integer values: remember that you can create a list of integers using the function

range. Here is an example program that computes the sum of the squares of the numbers from 0 to N, where N in read as input:

```
>>> N=int(raw_input("Enter the last integer considered --> "))
>>> Sum=0
>>> for i in range(0,N+1,1):
        Sum+=i**2
>>> print "The sum of the squares between 0 and ",N," is ",Sum
```

4.2 While loop

Sometimes, we face a situation where neither Python nor we know in advance how many times a loop will need to execute. This is the case for example when reading a file: we do not know in advance how many lines it has. Python has a structure for that: the while loop:

```
while TEST:
        code block;
```

The while structure executes the code block as long as the TEST expression evaluates as TRUE. For example, here is a program that prints the number between 0 and N, where N is input:

```
>>> N=int(raw_input("Enter N --> "))
>>> print "Counting numbers from 0 to ",N,"\n"
 i=0
while  i < N+1:
        print i,"\n"
        i+=1
```

Note that it is important to make sure that the code block includes a modification of the test: if we had forgotten the line i+=1 in the example above, the while loop would have become an infinite loop.

Note that any for loop can be written as a while loop. In practice however, it is better to use a for loop, as Python executes them faster

4.3 Break points in loops

Python provides two functions that can be used to control loops from inside its code block: **break** allows you to exit the loop, while **continue** skips the following step in the loop.

Here is an example of a program that counts and prints number from 1 to N (given as input), skipping numbers that are multiples of 5 and stopping the count if the number reached squared is equal to N:

```
>>> N = int(raw_input("Enter N --> "))        # Input N
>>> for i in range(0,N+1,1):                  # Start loop from 0 to N (N included)
        if i**2 == N:                         # test if i**2 is equal to N…
            break                             # if it is stop counting
        else:
            if  i%5==0:                       # Test if  i is a multiple of 5
                continue                      # if it is, move to next value
            print i,"\n"
```

Exercises:

1. Write a program that reads in an integer value *n* and outputs *n!* (Reminder: *n!=1x2x3x4….xn*).

2. Write a program that reads in a word, and writes it out with the letters at even positions in uppercase, and the letters at odd positions in lower case.

3. In cryptography, the Ceasar cipher is a type of substitution cipher in which each letter in the plaintext is replaced by a letter some fixed number of positions down the alphabet. For example, with a shift of 3, A would be replaced by D, B would become E, …, X would become A, Y would become B, and Z would become C.

    a. Write a program that reads in a sentence, and substitutes it using the Ceasar cipher with a shift of 3.
    b. Write a program that reads in sentence that has been encrypted with the Ceasar cipher with a shift of 3, and decrypts it.
    c. Repeat a and b above for a Ceasar cipher with a shift N, where N is given as input, N between 0 and 10