Chapter 3

Control Structures

Control structures determine the order in which the statements of a program are executed. They are the most fundamental building blocks of programs. It can even be shown that in a programming language with conditional statements, loops, and assignment statements (and nothing else) one can write any program that can be written in any language. Control structures are all that you really need. In this chapter we will look at control structures that are common to most programming languages in wide use, including **if**-statements, **while**-loops and **for**-loops.

3.1 If-statements

An **if**-statement is used to create conditional code: code that is executed if some condition is true, and not executed if it isn't. There are several versions of the **if**-statement; all of these have the meanings you would expect from similar usage in English.

```
if <condition>:
        statement_block
```

If the condition is True the statement_block is executed; if the condition is False, the statement_block is ignored. The statement_block is any Python code indented inside the **if**-statement.

```
if <condition>:
        statement_block_1
else:
        statement_block_2
```

If the condition is True then statement_block_1 is executed; if the condition is False then statement_block_2 is executed. The statement_block_1 is any code indented between the **if** and the **else**. The statement_block_2 is any code indented under the **else**.

```
if <condition_1>:
        statement_block_1
elif <condition_2>:
        statement_block_2
elif <condition_3>:
        statement_block_3
        ⋮
else:
        final_statement_block
```

The conditions are evaluated one at a time, starting from the top. If any of them evaluate to True the corresponding statement_block (the group of statements indented under the condition) is executed and then we leave the entire statement. If none of the conditions are True then the final_statement_block

is executed. Note that **elif** is just an abbreviation for **else if**. In all of these statements the conditions are anything that evaluates to True or False. We call such expressions *Boolean* expressions; the next section has more details about them. The statement blocks contain any valid Python statements.

This is much simpler than it might appear. Consider, for example, the following program. This asks the user to enter two numbers, and says which of the numbers is the larger one and which is the smaller.

```
# This asks for two numbers and
# prints them out in order.

def main():
    x = eval( input( "Enter a number: ") )
    y = eval( input( "Enter another number: ") )
    if x == y:
        print("Those numbers are the same.")
    elif x < y:
        print("%d < %d" % (x, y))
    else:
        print("%d < %d" % (y, x))

main()
```

Program 3.1.1: Example of an **if**-statement

When you are writing a program using conditional statements, keep in mind that you don't know which blocks will be executed unless you know what the user input will be. You can get into trouble if you create variables inside some branches of an **if**-statement and not in others. Consider the following code:

```
x = eval( input("Enter a number: ") )
if x < 10:
    size = "small"
print(size)
```

If the user enters a number that is less than 10 this prints the word "small". However, if the user enters a number that is 10 or larger the variable size is never created and so you will get an error message for the unknown varible size when it gets to the **print**-statement. One way to correct this is to define variable size outside the **if**-statement, so it will exist in any case:

```
size = "large"
x = eval( input("Enter a number: ") )
if x < 10:
```

```
        size = "small"
print(size)
```

Alternatively, you can put an **else** on the **if**-statement to define size in case the condition is False:

```
x = eval( input("Enter a number: ") )
if x < 10:
    size = "small"
else:
    size="large"
print(size)
```

When you have a chain of **if**-statements, as in **if-elif-elif** ..., remember that you only get to the lower conditions if all of the upper conditions fail. You can make use of this to simplify the conditions. For example, suppose we want a program that takes a number and says how many digits it has. Program 3.1.2 does this for numbers up to 4 digits long.

```
# This asks for a number and reports how
# many digits it contains.

def main():
    x = eval( input("Enter a number between 0 and 9999: ") )
    if x < 10:
        digits = 1
    elif x < 100:
        digits = 2
    elif x < 1000:
        digits = 3
    else:
        digits = 4
    print("%d has %d digits." % (x, digits))

main()
```

Program 3.1.2: Compound **if**-statement

Note that by the time we get to the condition

```
elif x < 100:
```

we already know that the condition $x < 10$ has failed, so if x is less than 100 it must have exactly 2 digits. If would be correct, but unnecessarily complicated to write this condition as

```
elif x >= 10 and x < 100:
```

By carefully analyzing a situation, it is often possible to simplify conditions in your code and thus make your programs easier to read and easier to write correctly.

We finish this section with one more example, this time computing leap years. We will make use of this code for several programs in subsequent chapter

Example The Julian Calendar, introduced by Julius Caesar in 46 B.C., had leap years occurring every 4 years. This equates the *tropical year*, the span from one point in the cycle of the seasons to the same point the following year, to 365.25 days. This is slightly too long, and over the centuries the Julian calendar grew out of synchronization with the natural seasons. In 1582 Pope Gregory XIII created the Gregorian Calendar, which differs from the Julian Calendar only in the way it computes leap-years. The Gregorian calendar was gradually adopted throughout Europe and most of Asia. Here is its leap-year algorithm:

> A year is a leap year if it is divisible by 4, unless it is also divisible by 100, in which case it is not a leap year unless it is also divisible by 400.

According to this algorithm, the year 1900 was not a leap year because it is divisible by 100 and not by 400, but the year 2000 was a leap year. This puts the length of the tropical year at 365.2425, which is very close to current observations. Our task is to write a program that asks the user to specify a year; the program will say whether or not this year is a leap year.

The way we stated the leap-year algorithm is not very helpful; the "unless" expressions don't translate easily into code. For our program we will reformulate the algorithm, taking the most specific conditions first. This is often a useful strategy. We express the most specific conditions first because we know what the answer is if those conditions are met.

For our program the conditions are:

- the year is divisible by 4
- the year is divisible by 100
- the year is divisible by 400

The most specific of these is being divisible by 400, so we start there: if the year is divisible by 400 it must be a leap year. We can use our remainder operator % to check for divisibility: a `year` is divisible by 400 if `year % 400 == 0`. Rather than sprinkling print statements throughout the code, we introduce a boolean varible `isLeap` that becomes `True` or `False` in the various conditions of our **if**-statement. At the end we use this variable to print out whether the year is a leap year.

The remaining cases of the analysis are similar. Program 3.1.3 shows the resulting code for this.

```
# This reads a year from the user and
# says whether this year is a leap-year.

def main():
    year = eval( input("Enter a year: ") )
    if year % 400 == 0:
        isLeap = True
    elif year % 100 == 0:
        isLeap = False
    elif year % 4 == 0:
        isLeap = True
    else:
        isLeap = False

    if isLeap:
        print("%d is a leap-year." % year)
    else:
        print("%d is not a leap-year." % year)

main()
```

Program 3.1.3: Determining leap years