CONTROL FLOW

As we have already discussed, the programs we have been writing are sets of instructions that are read and processed by the central processing unit (CPU) of the computer in the order in which they are written. For this reason, the type of programming that we have been doing is called ***sequential programming***, i.e., commands are evaluated and executed in sequence. Sometimes, however, you could imagine that it might be useful to either execute more than command at the same time (***parallel programming***), or execute commands in a sequence other than the one in which they are written. Because a single CPU can only execute one statement at a time, your typical desktop computer is only capable of sequential processing (you need more than one CPU to do parallel processing). However, most computer languages, Python included provide ways around the limitations of sequential processing that allow us to control the order (or flow) in which commands are evaluated and executed.

There are three types of control flow statements, two of which we have already seen. These are Jump or **Goto** Statements, **Loops** Statements, and **Conditional** (or Branching) Statements.

Jump and Goto are outdated terms from the early days of programming. They basically refer to commands that tells the program to jump to another line in the code (e.g., in BASIC, GOTO 102 means go to line 102 of the program and execute the command written on that line). Most modern programming languages lack these sorts of statements, and the programming task they accomplished is now done through the use of ***functions***; i.e., when we invoke a function, we are interrupting the sequential flow of the program and telling it to "GOTO" the function definition and execute that code block before returning to the main flow.

As we have already seen, with `for`-statements, ***loops*** provide another means of controlling the flow of the program. They tell the program to repeat a block of code some number of times before returning to the usual sequential flow of the program. *for*-statements are only one of several types of loops; we will introduce another one below.

When we first started to discuss computing, we learned that a computer's processor can do three types of operations: it can add 0, it can add 1, and it can compare zeros to ones and see if they are the same or different. It is this third type of operation that differentiates a computer from a simple adding machine: a computer both adds and uses comparisons to make decisions. Thus, making simple decisions is an important part of computer programming. **Conditional** or **branching** statements allow us change the flow of the program based upon such decisions, i.e., they basically create fork in the path of our sequential program, and allow the programmer to redirect the flow of the program depending on whether or not certain ***conditions*** are met.

RUNNING AROUND IN CIRCLES: MORE LOOPS

We have already learned that repetition is one of the key concepts in computing. For example, we could use repetition to compute the world's population in ten years by repeatedly computing values for successive years:

```
for year in range(10):
    population = population * (1 + growthRate)
```

That is, repeatedly add the increase in population, based upon a rate of growth, ten times.

Do This: Write a module that implements the above code block as a program and name it worldPop.py. Modify the program so that it requests the user to input the current population, growth rate, and number of years to project ahead and computes the resulting total population size. Run your program on several different values (Google: "world population growth" to latest numbers). Can you estimate when the world will have a population size of 9 billion?

Do This: By this time you have been introduced to the rules of naming in Python. You may have noticed that we have made extensive use of *mixed case* in naming some entities. For example, growthRate. There are several naming conventions used by programmers and that has led to an interesting culture in of itself. Look up the phrase *CamelCase Controversy* in your favorite search engine to learn about naming conventions. For an interesting article on this, see The Semicolon Wars

Another way of doing the same thing is to use the a different kind of loop, a **`while`** loop as follows:

```
year = 0
while year < 10:
       population = population * (1 + growthRate)
       year = year + 1
```

This loop illustrates a new type of statement, the `while`-statement. Its general form is as follows:

```
while <some condition is true>:
       <do something>
```

That is, you can specify any condition in `<some condition is true>`. The condition is tested and if it results in a `True` value, the step(s) specified in the body of the loop (`<do something>`) is/are performed. Then the condition is tested again, and as long it remains True, the loop will continue to repeat. In the example above, we use the following condition:

```
year < 10
```

If the above example condition is true, it implies that `year` is less than 10. If it is false, it implies that `year` is greater than or equal to 10 and evaluating the condition results in a `False` value, and the loops stops.

You have now seen how to write programs that have commands that can be repeated a fixed number of times or for a certain duration:

```
# do something N times
for step in range(N):
    do something...


# do something for some duration
while <waiting for condition>:
       do something...

# do something for some duration
value = <initialState>
while value < finalState:
       do something...
```

We can also write loops that repeat forever:

```
# do something forever
while True:
       do something...
```

All of the above are useful in different situations. Sometimes it becomes a matter of personal preference; i.e., if `while`-loops make more sense to you, there is no reason not to use them instead of a `for`-loop!

Do this: Write a for-loop that prints out the numbers from 0 to 10 to the IDLE window. Now write a while-loop that does the same. Where is the indexing variable in the `for`-statement? Does the `while`-statement have an equivalent variable (i.e., one that keeps track of how many times the loop is being repeated)? In these two types of loops, what is the difference between how we define the starting condition, stopping condition, and increment of the value of the indexing variable?

EVALUATING CONDITIONS

As you can see from the above examples, learning about writing conditions is essential to writing more sophisticated programs. Any decision making in your programs depends on forming appropriate conditional expressions.

The first thing to realize is that all conditions result in either of two values: `True` or `False`. As we learned in Chapter 2, variables that have these values have a **Boolean** data type. Most programming languages allow you to substitute the integer values of 1 and 0 for the Booleans `True` and `False`, respectively. In Python, the values `False`, `None`, 0, 0.0, and empty strings and lists are interpreted as false. All other values are true.

Boolean values can be used just like numbers in your programs. However, remember that the ***type*** of a value determines that kinds of operations that can be performed on the value. Because Booleans are logical data types (`True`, `False`), they can be manipulated with two types of operators: ***logical*** and ***relational***. For example, simple conditions can be written using comparisons (relational) operations: < (less than), <= (less than or equal to), > (greater than), >= (greater than or equal to), == (equal to), and != (not equal to). These operations can be used to compare all kinds of values. Here are some examples:

```
>>> 42 > 23
True
>>> 42 < 23
False
>>> 42 == 23
False
>>> 42 != 23
True
>>> (42 + 23) < 100
True
>>> a, b, c = 10, 20, 10
>>> a == b
False
>>> a == c
True
>>> a == a
True
>>> True == 1
True
>>> False == 1
False
```

The last two examples above also show how the values `True` and `False` are related to 1 and 0. `True` is the same as 1 and 0 is the same as `False`. You can form many useful conditions using the comparison operations and all conditions result in either a `True` or `False` when evaluated.

True is the same as 1 and 0 is the same as False. You can also compare other types of values, like strings:

```
>>> 'Hello' == 'Good Bye'
False
>>> 'Hello' != 'Good Bye'
True
>>> 'Elmore' < 'Elvis'
True
>>> 'New York' < 'Paris'
True
>>> 'A' < 'B'
True
>>> 'a' < 'A'
```

Unicode

Text characters have an computer coding or representation that enforces lexicographic ordering. This internal encoding is very important in the design of computers and this is what enables all computers and devices like iPhones etc. to exchange information consistently. All language characters in the world have been assigned a standard computer encoding. This is called *Unicode.*

```
False
```

Study the above examples carefully, there are two important things to notice First, strings are compared in alphabetical order (i.e., lexicographically). Thus `'Elmore'` is less than `'Elvis'`, since `'m'` is less than `'v'`. Second, is that uppercase letters are less than their equivalent lowercase letters. This is by design (see box).

Besides relational operators, you can build more complex conditional expressions using the ***logical** operations* (also called *Boolean operations*): `and`, `or`, `not`. Here are some examples:

```
>>> (5 < 7) and (8 > 3)
True
>>> not ((5 < 7) and (8 > 3))
False
>>> (6 > 7) or (3 > 4)
False
>>> (6 > 7) or (3 > 2)
True
```

We can define the meaning of these logical operators as follows:

- `<expression-1>` **`and`** `<expression-2>`: Such an expression will result in a value `True` ***only if both*** `<expression-1>` and `<expression-2>` are `True`. In all other cases (i.e., if either one or both of <expression-1> and <expression-2> are `False`) it results in `False`.

- `<expression-1>` **`or`** `<expression-2>`: Such an expression will result in a value `True` ***if either or both*** `<expression-1>` and `<expression-2>` are `True`. In all other cases (i.e., if either both <expression-1> and <expression-2> are `False`) it results in `False`.

- **`not`** `<expression>`: Such an expression will result in a value `True` if `<expression>` if `False` or `True` if `<expression>` is `False`. In other words, it flips or complements the value of `<expression>`.

Logical operators can be combined with relational expressions to form arbitrarily complex conditional expressions. These operators were invented by the logician, George Boole in the mid-19th century. Boolean algebrea (and hence the Boolean data type), named after Boole, defines some simple, but important logical laws that govern the behavior of logical operators. The following are some useful ones:

- `(A or True)` is always `True`.
- `(not (not A))` is just `A`
- `(A or (B and C))` is the same as `((A or B) and (A or C))`
- `(A and (B or C))` is the same as `((A and B) or (A and C))`
- `(not (A or B))` is the same as `((not A) and (not B))`

- `(not (A and B))` is the same as `((not A) or (not B))`

These identities, or properties, can help you simplify expressions and increase the readability of your code.

Conditional expressions can be used to write several useful conditions to control the execution of some program statements. We have already seen conditional repetitions:

```
while <some condition is True>:
      do something...
```

And we can understand why the following is a way of saying "do something forever":

```
while True:
      do something...
```

Since the condition is always `True`, the statements will be repeated forever. Similarly in the loop below:

```
year = 0
while year < 10:
      population = population * (1 + growthRate)
      year = year + 1
```

As soon as `year` equals 10, the value of the condition `year < 10` will become `False` and the repetition will stop. Controlling repetitions based upon conditions is a powerful idea in computing.

EXERCISES

1. Complete the following truth tables (10 pts):

A	B	not A	A and B	A or B	A == B
True	True				
True	False				
False	True				
False	**False**				

A	B	A != B	A >= B	A < B	A == B
1	1				
1	0				
0	1				
0	0				
'Amy'	'Amelia'				
'L'	'l'				

2. If you have not yet, create a program that executes the world population example using the while-loop version and allows the user to specify values for all values (as suggested in the text). Considering what you have learned, try the following and explain the resulting behavior: (20 pts)
    a. Use the values 900000000 and 1.42 as input values. Enter them in any order (i.e., don't be concerned about to which variable (pop size or growth rate) you are assigning the value). What happens?
    b. For any values to be input, enter a string when prompted for a number. What happens?
    c. Modify your world population program so that it repeats as long as the population size is below 9 billion. Have your program output the year by which the world's population would exceed that threshold. For extra credit, modify your program so that the user can specify the population limit.

3. Other than Pylab and the graphics library there are many other libraries that we can import into our Python programs to increase functionality. One such library is the **`time`** library. Import it as you would any other library or module: (20 pts)

```
>>> from time import *
```

The time library allows us to access the current date and time stored on the computer. It does so through a function that called `localtime` that works as follows:

```
>>> localtime()
(2008, 3, 27, 12, 33, 16, 3, 87, 1)
```

`localtime` returns all of the following in order:
    1. year
    2. month
    3. day
    4. hour
    5. minute
    6. seconds
    7. weekday
    8. day of the year
    9. whether it is daylight savings time or not

In the example above, it is March 27, 2008 at 12:33 pm and 16 seconds. It is also the 3rd day of the week, 87th day of the year, and we are using daylight savings time. You can assign each the values to named variables with the following syntax:

```
year, month, day,…, dayOfWeek = localtime()
```

Then, for the example above the variable year will have the value 2008, the month will have 3, etc.

Do the following:

a. Write a program that prints out the current date and time in the format illustrated above (March 27, 2008 at 12:33 pm and 16 seconds)
b. Write a program that starts at a value at zero and increments it by one for 2 minutes. What is the result of the calculation? Move to a different computer and run your program again. Did you come up with the same result? Try your program on at least 5 different computers. Are the results the same? Why or why not?