Use of Boolean Variables

In addition to checking simple relationships between arithmetic expressions for truth, we can use variables that simply equal True or False to control our if statements. Consider a situation where you students are eligible for a scholarship if they either have a GPA of at least 3.5 or have an SAT scoore of at least 1800 (out of 2400). One way to solve the problem would be as follows:

```
def main():

    eligible = False
    gpa = float(input("What is your GPA?\n"))
    sat = int(input("What is your SAT score?\n"))

    if gpa >= 3.5:
        eligible = True
    if sat >= 1800:
        eligible = True

    if eligible:
        print("You are eligible for the scholarship.")
    else:
        print("Sorry, you aren't eligible for the scholarship.")

main()
```

Our strategy here is to initialize our Boolean variable eligible to False, indicating that the user isn't currently eligible for the scholarship. In python, we indicate the literal value of false with the word false with the first letter capitalized. The value ISN'T a string, so no quotes appear around it. Similarly, the literal value true is indicated with the first letter of the word being capital.

Once we read in the user input, there are two ways in which the user can become eligible for the scholarship. We check both using separate if statements, and in both cases, change our variable eligible to True. If neither of these if statements trigger, then the user is not eligible for the scholarship. Note that the two if statements may be written as an a single if-elif statement and still work properly. (As previously mentioned, almost always, there is more than one method to solve a problem correctly.)

Complex Boolean Expressions

Some may look at the previous example and complain that having to use two separate if statements seems inefficient. A natural followup question would be: can we check multiple conditions in a single Boolean expression? Python (and most other programming languages) allows this. In particular, we can evaluate the result of checking two Boolean conditions in two ways:

(1) the "and" of two Boolean expressions
(2) the "or" of two Boolean expressions

The following two tables (often called truth tables) illustrate the meaning of "and" and "or", which correspond naturally to their English meanings:

Operand #1	Operand #2	Result (or)
`False`	`False`	`False`
`False`	`True`	`True`
`True`	`False`	`True`
`True`	`True`	`True`

Namely, the Boolean expression resulting by composing the "or" of two given Boolean expressions is True as long as at least one of the two given Boolean expressions is true. Note that it's perfectly permissible for both to be true. (In our scholarship example, we are still eligible if we have a GPA of 3.7 and an SAT score of 1900.)

Here is the truth table for the Boolean operator "and":

Operand #1	Operand #2	Result (and)
`False`	`False`	`False`
`False`	`True`	`False`
`True`	`False`	`False`
`True`	`True`	`True`

Using the Boolean operator "or", we can shorten our program as follows:

```
def main():

    gpa = float(input("What is your GPA?\n"))
    sat = int(input("What is your SAT score?\n"))

    if gpa >= 3.5 or sat >= 1800:
        print("You are eligible for the scholarship.")
    else:
        print("Sorry, you aren't eligible for the scholarship.")

main()
```

Example Using and

One of the classic problems used to illustrate complicated Boolean logic is the leap year problem. Everyone knows that all leap years are divisible by 4. However, some may not know that not all years divisible by 4 are leap years. In particular, of the years divisible by 4, those divisible by 100, but not 400 are NOT leap years. Thus, the years 1700, 1800 and 1900 were NOT leap years, but 2000, since it's divisible by both 100 and 400, was.

In the following program we ask the user to enter a year and we print out whether or not it is/was a leap year.

```
def main():

    year = int(input("What year would you like to check?\n"))

    leapYear = True

    if year%4 != 0:
        leapYear = False
    elif year%100 == 0 and year%400 != 0:
        leapYear = False

    if leapYear:
        print(year,"is a leap year.")
    else:
        print(year,"is not a leap year.")

main()
```

We make use of the and operator by checking for the exceptions to the divisible by 4 rule. The general strategy taken by this program is to assume that a given year is a leap year and change our answer to false if the year doesn't pass one of the tests. Both of the conditions listed must be true in order for the year in question to be marked as a non-leap year.

A great deal of logic in programming involves checking whether or not multiple conditions are True or False, using various combinations of or's and and's. Further examples utilizing these operators will be shown throughout this textbook.

Short-Circuiting

A short circuit in electronics is generally not a good thing. Luckily, in programming, the term refers to something different without catastrophic consequences. Some complex Boolean expressions can be evaluated without looking at both operands. Consider the following code segment:

```
dx = 0
dy = 5

if dx != 0 and dy/dx > 0:
    print("The slope is positive.")
```

In this situation, we see that dx is zero, so the first part of our Boolean expression is false. But we also know that in order for the and of two expressions to be True, both have to be true. Thus, without ever checking the second Boolean expression, we can ascertain that the entire expression will be False. The Python interpreter is smart enough to do this logic! Thus, it never bothers to evaluate dy/dx > 0. This is short-circuiting in programming. Any time the compiler can determine a definitive value to a Boolean expression, it never checks the rest of it.

This is a good thing because if we had tried to divide 5 by 0, then we would get a division by zero error. In fact, the programmers purposefully count on short-circuiting while writing their code to avoid many errors like this one. (If the interpreter didn't use short-circuiting, the if statement would have to be split into two that checked the two conditions separately, checking the second only when the first was true.

Another situation where short-circuiting applies with a complex statement is as follows:

```
dx = 3
dy = 5

if dx > 0 or dy > 0:
    print("Not in the third quadrant.")
```

Here, after checking that the first Boolean expression is True, we can ascertain that the whole expression is True and have no need to check the value of dy. (Note that in this case, nothing bad would have happened had we checked the expression.)

Special Cases

Python goes to great lengths to help programmers avoid errors using the if statement as compared to the if statement in other languages. The following works code segment works exactly as the programmer intends:

```
age = int(input("How old are you?\n"))

if 15 < age < 25:
    print("You may drive, but not rent a car.")
```

The programmer's intent here is for the if statement to trigger as true if the variable age is in between 16 and 24, inclusive. In other languages, this expression does not work as intended.

A common mistake many programmers make is to mistakenly use one equal sign instead of two. Luckily, this results in a syntax error in Python and the programmer is alerted before her program can run. Consider the following code segment:

```
age = int(input("How old are you?\n"))

if age = 16:
    print("Woohoo, time to drive!!!")
```

When attempting to interpret this code segment, Python's interpreter responds with:

```
SyntaxError: invalid syntax
```

and highlights the offending single equal sign in red. At this point, hopefully the programmer realizes her error.