1.5 Arithmetic Expressions in Python

In the two examples in the previous section, we used arithmetic expressions on the right-hand side of the assignment statement (equal sign). Python has its set of rules about how these expressions are to be evaluated, so that there is no ambiguity. So, far, we've simply illustrated that multiplication and division have higher precedence than addition and subtraction, as is frequently taught in grade school mathematics. Furthermore, parentheses have the highest precedence and can be used to "force" the order in which operations are evaluated as is illustrated in the this line of code that was previously shown:

```
total_price = item_price*(1+tax_rate/100)
```

In this expression, we first evaluate the contents of the parentheses before multiplying. In evaluating the contents of the parentheses, we first perform the division, since that has higher precedence than addition. Thus, for example, if tax_rate is 7, then we first take 7/100 to get .07 and then add that to 1 to get 1.07. Then, the current value of item_price is multiplied by 1.07, which is then assigned to total_price.

Python also provides three more operators for us to use:

1. **, for exponentiation.
2. //, for integer division
3. %, for modulus

The following sections describe exactly how each of these operators work, and the order of operations for each of them.

Exponentiation (**)

Many students first learn to associate the caret symbol (^) with exponentiation because that key is typically linked to exponentiation on most calculators that students use in grade school. However, in most programming languages, the caret symbol is either not defined or means something very different than exponentiation.

Some programming languages don't define exponentiation through an operator at all, but python does. The operator is defined to work for real numbers. Here are several examples of its use:

```
>>> 2 ** 3
8
>>> 3 ** 5
```

```
243
>>> 4 ** 10
1048576
>>> 25 ** .5
5.0
>>> 2 ** -3
0.125
>>> 9999 ** 0
1
>>> -5 ** -3
-0.008
>>> -5 ** 3
-125
>>> 1.6743 ** 2.3233
3.311554089370817
```

If both operands to an exponentiation operation are integers and the answer is an integer, then it will be expressed as one. If both operands are integers but the answer isn't, then the answer will be expressed as a real number with decimals. As the examples above illustrate, if an exponent b is negative, then a^b is defined as $1/a^{-b}$. (Technically, this rule is true for all exponents, not just negative ones.)

Integer division (/ /)

Python provides a second division operator, //, which performs integer division. In particular, the answer of an integer division operation is always an integer. In particular, a//b is defined as the largest number of whole times b divides into a. Here are a few examples of evaluating expressions with integer division:

```
>>> 25//4
6
>>> 13//6
2
>>> 100//4
25
>>> 99//100
0
>>> -17//18
-1
>>> -1//10000000
-1
```

```
>>> -99999999//99999999
-1
>>> -25//4
-7
>>> -13//6
-3
>>> -12//6
-2
>>> -11//6
-2
>>> 0//5
0
```

It's important to note that python deals with this operation differently than many other programming languages and different than most people's intuitive notion of integer division. In particular, when most people see -13//6, they tend to think that this is pretty close to -2, so that the answer should be -2. But, if we look at the technical definition of integer division in python, we see that -2 is greater than -13/6, which is roughly -2.166667, and the largest integer less than or equal to this value is really -3.

Incidentally, the definition of integer division in Python does not require the two numbers that are being divided to be integers. Thus, integer division operations are permissible on numbers that aren't integers. Consider the following examples:

```
>>> 6.4 // 3.1
2.0
>>> 3.4 // 3.5
0.0
>>> 9.9 // .03
330.0
>>> -45.3 // -11.2
4.0
>>> 45.3 // -11.2
-5.0
```

This feature of Python is used relatively rarely, so no further details will be given at this point.

Modulus Operator (%)

For those who have never programmed, it's likely that the modulus operator, denoted by the percent sign(%), is not familiar. In mathematics, modulus is typically defined for integers only. However in python, the modulus operator is defined for both integers and real numbers. If both operands are integers, then the answer will be an integer, otherwise, it will be a real number.

Intuitively, the modulus operator calculates the remainder in a division, while integer division calculates the quotient. Another way of thinking about modulus is that is simply calculates the leftover when dividing two numbers. This intuitive definition is correct when dealing with positive numbers in Python. Negative numbers are another story, however.

The formal definition of a % b is as follows:

a % b evaluates to a - (a // b)*b.

a // b represents the whole number of times b divides into a. Thus, we are looking for the total number of times b goes into a, and subtracting out of a, that many multiples of b, leaving the "leftover."

Here are some conventional examples of mod, using only non-negative numbers:

```
>>> 17 % 3
2
>>> 37 % 4
1
>>> 17 % 9
8
>>> 0 % 17
0
>>> 98 % 7
0
>>> 199 % 200
199
```

We see in these examples, if the first value is ever less than the second value, then the answer of the operation is simply the first value, since the second value divides into it 0 times. In the rest of the examples, we see that if we simply subtract out the appropriate number of multiples of the second number, we arrive at the answer.

With negative integers however, one must simply plug into the formal defintion instead of attempting to use intuition. Consider the following examples:

```
>>> 25 % -6
-5
>>> -25 % -6
-1
>>> -25 % 6
5
>>> -48 % 8
0
>>> -48 % -6
0
>>> -47 % 6
1
>>> -47 % 8
1
>>> -47 % -8
-7
```

The key issue that explains the first two results is that there is a different answer for the two integer divisions 25//-6 and -25//-6. The first evaluates to -5 while the second evaluates to 4. Thus, for the first, we calculate -6 * -5 = 30, and need to subtract 5 to obtain 25. For the second, we calculate -6 * 4 = -24, and need to subtract 1 to obtain -25.

See if you can properly apply the definition given to explain each of the other answers shown above.

In Python, the modulus operator is also defined for real numbers, using the same definition as the one shown. Here are a few examples of its use:

```
>>> 6.4 % 3.1
0.20000000000000018
>>> 3.4 % 3.5
3.4
>>> 9.9 % .03
7.216449660063518e-16
>>> -45.3 % -11.2
-0.5
>>> 45.3 % -11.2
-10.7
```

If we take a look at the first and third examples, we see the answer that python gives is slightly different than we expect. We expect the first to be 0.2 exactly and the third to be 0. Unfortunately, many real numbers are not stored perfectly in the computer. (This is true in all computer languages.) Thus, in calculations involving real numbers, occasionally there are some slight round-off errors. In this case, the digits 1 and 8 way at the right end of the number represent a round-off error. In the third case, the last part of it, e-16, simply means to multiply the previously shown number by 10^{-16}, so the entire part printed represents the round-off error, which is still tiny, since it's less than 10^{-15}. As long as we don't need extreme precision in our calculations, we can live with the slight errors produced by real number calculations on a typical computer. The more complicated calculations are, the greater the possible error could cascade to be. But, for our purposes, we'll simply assume that our real number answers are "close enough" for our purposes. (There are a variety of advanced techniques that help programmers contend with round off error of real number calculations.)

1.6 Reading User Input in Python

input statement

Python makes reading input from the user very easy. In particular, Python makes sure that there is always a prompt (a print) for the user to enter some information. Consider the following example entered

```
>>> name = input("What is your name?\n")
What is your name?
Simone
>>> print("Please to meet you ", name, ".", sep="")
Please to meet you Simone.
```

In this way, instead of the print always printing out the same name, it will print out whatever name the user entered. The key is that the input statement read in whatever the user entered, and then the assignment statement, with the equal sign, assigned this value to the variable name. Then, we were free to use name as we pleased, knowing that it stored the value the user entered.

Our previous programs that calculated the price of an item with tax and the area of a square, were limited because they always calculated the same price and area. Our program would be much more powerful if we allowed the user to enter the appropriate values so that our program could calculate the information THEY are interested in (as opposed to the same value every time.)

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.

2.3 random class

Computer programs, particularly games, are much more fun if there's some randomness involved. Unfortunately, we have no reliable way of producing truly random numbers. However, most programming languages, including Python, include a pseudorandom number generator. These generators use a set of steps to generate numbers that appear random. Thus, if you or I knew the exact set of steps the generator was using, we could reproduce every numer the generator created. However, to the casual observer, the numbers produced would appear random. Random numbers in programs allow us to play games with some uncertainty (think dice) and allow us to simulate real life events that have some uncertainty (think stock market). Python makes using random numbers fairly easy. In order to do so, we must do the following import:

```
import random
```

At the beginning of our program (preferably in main), we must seed our random number generator as follows:

```
random.seed()
```

For now, the details of this function call are not important. Simply include this as one of the first lines of your main function in any program that uses random numbers.

From this point on, if you want a random number selected between two integers a and b, inclusive, simply make the following function call:

```
random.randint(a, b)
```

Since this function call returns a value, like a majority of the math functions, we must call it as part of a greater line of code, typically storing its return value in a variable.

The following program will use the random number generator to generate one random number in between 1 and 100, and allow two players to guess the number. The winner will be the player who comes closest to the number without guessing too high. If both players guess too high, or if both players guess the same number, the outcome will be a tie. In all other cases there will be a unique winner.

Though this example generates only one random number, a program is allowed to generate many random numbers, if necessary. In these cases, we still only seed the random number generator only once.

```
import random

def main():

    random.seed()

    secretNum = random.randint(1,100)

    # Get the user input.
    guess1 = int(input("Player 1, enter your guess(1-100).\n"))
    guess2 = int(input("Player 2, enter your guess(1-100).\n"))

    # Calculate how close both players are.
    diff1 = secretNum - guess1
    diff2 = secretNum - guess2

    print("The correct number was",secretNum)

    # Check all of the cases!
    if diff1 < 0 and diff2 < 0:
        print("Both players bust. The game is a tie.")
    elif diff2 < 0:
        print("Player 1 wins since Player 2 busted.")
    elif diff1 < 0:
        print("Player 2 wins since Player 1 busted.")
    elif diff1 < diff2:
        print("Player 1 is closer to the correct number and wins!")
    elif diff1 == diff2:
```

```
        print("Both players guessed the same number and tie.")
    else:
        print("Player 2 is closer to the correct number and wins!")

main()
```

In this program, we generate a single random number in between 1 and 100, inclusive and store it in secretNum. After that, we use the if statement to separate out several different cases to determine which of the two players has won. As this example illustrates, a problem that seems so simple to us intuitively can have a rather complex, detailed solution. Though our brain can carry out this "Price is Right" logic effortlessly, we see that when formalized, there are several conditions that need to be checked.

The following section will contain an example that utilizes multiple random numbers in the same program.

2.4 Writing Our Own Functions

Motivation for User-Defined Functions

Up until now, we've only called functions that Python has provided for us. Some examples of the functions we've used are: print, input, int, float, sqrt, round and randint. Python and nearly all programming languages allow users to define their own functions.
Simply put, a function is a mini-program that completes a specified task. For example, the sqrt function takes its input value and returns its square root. The randint function takes in a low bound and a high bound and selects a random integer within the specified range. The ceil function finds the smallest integer greater than or equal to its input value and returns it. We can define functions to do any sort of task that we would like or to make any sort of calculation we want. Once we define a function, we can use it over and over again.

(Python) Chapter 2: If Statement, Random Class, Introduction to Defining Functions

2.1 Conditional Execution

Basic Idea

One limitation to programs created only using the statements presented in chapter 1 is that the same exact statements in a program will run every time the program is interpreted. The problem with this is that in real life, when we carry out directions, we don't always execute the same steps. Consider the situation of determining whether or not you will go out with a friend. If your homework is done, you would like to go out with your friend. But, if your homework isn't done, you won't go out with your friend. Similarly, in programming, it makes sense to allow conditional execution. Namely, if some condition is true, then execute some set of statements.

Basic if Statement

In Python, the syntax of the most basic if statement is as follows:

```
if <Boolean Expression>:
    stmt1
    stmt2
    ...
    stmtn
stmtA
```

A Boolean expression is one that always evaluates to true or false. Details about how to create a Boolean expression will be covered shortly. If this expression evaluates to true, then the statements stmt1 through stmtn are executed in order, followed by stmtA. However, if this expression evaluates to false, then all of these statements are skipped and stmtA is then executed. Note: It's not required for there to be a statement such as stmtA after the completion of the if statement.

The interpreter determines which statements are inside of the if clause based on indentation. For a statement to be consider inside of the if, it must be indented to the right from the if statement itself. All subsequent statements inside of the if must be indented to the same level.

Sales Tax Example Revisited

When buying most items, sales tax is added to the price. However, for some items, such as basic food, no sales tax is added. In this example we'll ask the user to enter the item price. Then we'll ask them if sales tax is to be assessed. If it is, then we'll ask the for percentage of sales tax and calculate the final price.

```
# Arup Guha
# 6/26/2012
# Sales Tax Program Revisited - conditionally charges sales tax.

def main():

    # Get the user input.
    item_price = float(input("Please enter the price of your item.\n"))
    is_taxed = input("Is your item taxed(yes,no)?\n")

    # If the item is taxed, ask the sales tax percentage and add tax.
    if is_taxed == "yes":
        tax_rate = float(input("What is the sales tax percentage?\n"))
        item_price = item_price + item_price*tax_rate/100

    # Calculate the total price and round.
    print("Your total cost is $",item_price,".",sep="")

# Start the program.
main()
```

This program shows our first example of a Boolean expression. The Boolean expression in this program is:

```
is_taxed == "yes"
```

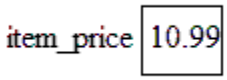
This is how we check to see if the variable is_taxed is equal to the string "yes". If it is, then this Boolean expression evaluates to true. Otherwise, it evaluates to false.

Thus, if the user enters "yes", then they will be prompted to enter the percentage of sales tax. Then the variable item_price will be reassigned to include sales tax. If the user enters anything but "yes", then these two statements are skipped. Afterwards, the value of the variable item_price is printed.

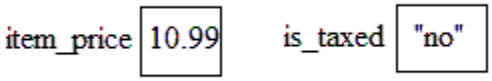
Let's take a look of running this program two separate times:

```
>>>
Please enter the price of your item.
10.99
Is your item taxed(yes,no)?
no
Your total cost is $10.99.
```

After the first line, the picture in memory is as follows:



After the second line, the picture in memory is:



At this point, we approach the if statement. We compare the value of the variable is_taxed to the string literal "yes", and see that they are not equal. Note that when we type in strings we don't type in the double quotes, but when we denote string literals (string values instead of string variables) inside of our programs, we denote them with either matching double quotes or matching single quotes, as was previously discussed in the section about the print statement.

Since this if statement evaluates to false, the following statements that are indented get skipped. The next statement that runs is:

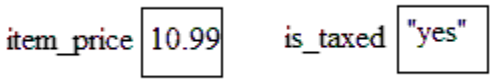
```
print("Your total cost is $",item_price,".",sep="")
```

Since the value of the variable item_price is 10.99 at this point in time, this is what gets printed for the total cost.

Now, consider the following execution of the program:

```
>>>
Please enter the price of your item.
10.99
Is your item taxed(yes,no)?
yes
What is the sales tax percentage?
6.5
Your total cost is $11.70435.
```

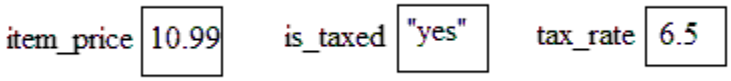
The picture for this execution after the first two lines of code is:



At this point, when we evaluate the Boolean expression in the if statement, we find that it's true since the variable is_taxed stores the string "yes". Then we go ahead and execute the following statement:

```
tax_rate = float(input("What is the sales tax percentage?\n"))
```

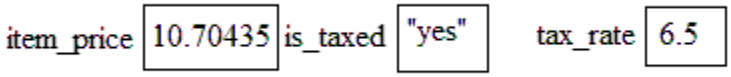
After this statement is executed, our picture of memory is as follows:



Then we execute the following statement in the if:

```
item_price = item_price + item_price*tax_rate/100
```

item_price currently evaluates to 10.99 while item_price*tax_rate/100 is equals to .71435. Adding these, we evaluate the right-hand side of the assignment statement to equal 11.70435, thus our picture in memory AFTER this statement is:



One of the basic building blocks of a Boolean expression is a relational operator. Here are the six relational operators and their meanings:

Relational Operator	Meaning
==	Equal to
!=	Not equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to

Thus, in this Boolean expression we are checking to see **IF** the variable is_taxed has the value "yes". Notice that checking for equality uses two equal signs instead of one. This is because one

equal sign already has a well-defined meaning: the assignment operator. Assigning a variable changes its value while checking for equality between two expressions doesn't change the value of any of the variables involved.

Formatting Decimal Output to a Specific Number of Places

In our previous examples, when we printed out real numbers, they printed out to many decimal places. Python uses a method similar to the language C to format real numbers to a fixed number of decimals. The syntax is strange and uses that percent sign (%), which we use for mod, in a different way. The expression that evaluates to a variable rounded to two decimal places is:

```
"%.2f"%var
```

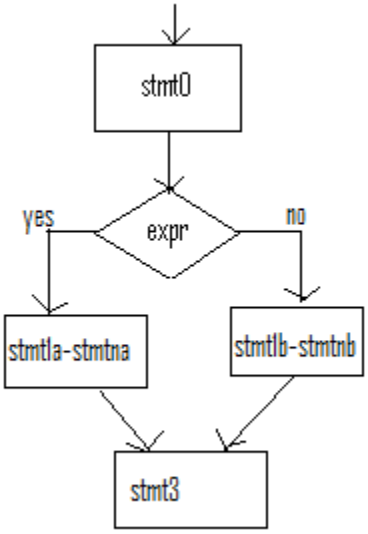
where var is the variable to round. Here is an application of this syntax to displaying the item price rounded:

```
print("Your total cost is $", "%.2f"%item_price, ".",sep="")
```

If you want a different number of decimal places displayed, just change the 2 in the format string. (Note: The f in that set of double quotes indicates float and the .2 indicates to print out two digits after the decimal.)

2.2 if Statement with else Clause

In the previous example, if the item was taxed, we wanted to carry out an action, but if it wasn't we simply wanted to skip that action. In many cases however, if some condition is true, we want to execute one set of statements, but if it's false, we want to execute a separate set of statements.



The basic syntax for this type of situation is as follows:

```
if <Boolean Expression>:
    stmt1a
    stmt2a
    ...
    stmtna
else:
    stmt1b
    stmt2b
    ...
    stmtmb
stmtA
```

The basic flow of control here is that we first evaluate the Boolean expression. If it's true, we complete statements stmt1a through stmtna and then continue to stmtA. Alternatively, if the Boolean expression is false, skip stmt1a through stmtna, but do execute statements stmt1b through stmtmb, and then continue to stmtA.

Let's take a look at a couple examples that utilize this component of the if statement.

Work Example

Consider a job with flexible hours where you must spend a certain number of hours a week. During the week if you've exceeded that number, let's say you have to take the excess hours as vacation in future weeks. Alternatively, if you haven't exceeded that number, you'll have to work the remainder of the hours. In this program, we will ask the user to enter the number of hours they are supposed to work a week and how many she's worked thus far. Then, our program will print the appropriate output, asking the user to either work more hours, or take vacation.

```
# Arup Guha
# 7/2/2012
# Example of a Basic if-else statment - determines if you need to
# work more or if you need to take vacation time.

def main():

    work_week = int(input("How many hours are you supposed to work?\n"))
    this_week = int(input("How many hours have you worked this week?\n"))

    # You've worked enough!
    if this_week > work_week:
        print("You must take",this_week-work_week,"hours of vacation.")

    # Need to put in some more hours!!!
    else:
        print("You must still work",work_week-this_week,"hours this week.")

main()
```

Now, in the case that the Boolean expression is true, we print out the vacation hours. Alternatively, we print out the hours left to work. Incidentally, what happens if you've worked the exact correct number of hours?

Quadratic Equation Example

A common formula taught in Algebra I is the quadratic formula. However, sometimes this formula leads to "impossible" roots, which we later learn are "complex." In this program, given the coefficients of a quadratic equation from the user, if the roots are real, we will print them out. If they are not, we'll print out an error message.

The quadratic formula is as follows: $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$. This equation has to real roots so long as what is under the square root sign is non-negative. This leads to the following program:

```
# Arup Guha
# 7/2/2012
# Quadratic Equation Solver

def main():

    # Get user input.
    a = float(input("Please enter a from your quadratic equation.\n"))
    b = float(input("Please enter b from your quadratic equation.\n"))
    c = float(input("Please enter c from your quadratic equation.\n"))

    # Calculate the discriminant.
    disc = b**2 - 4*a*c

    # Deal with real roots.
    if disc >= 0:

        x1 = (-b + disc**.5)/(2*a)
        x2 = (-b - disc**.5)/(2*a)

        print("Your roots are ",x1," and ",x2,".", sep="")

    # Error message for complex roots.
    else:

        print("Sorry, your roots are complex.")

main()
```

elif clause

In the work week example, if the user worked the exact correct number of hours, our program would print the following message:

```
You must still work 0 hours this week.
```

While this is technically accurate, the tone of this message is a bit misleading. It would be nice if we had a third "option" to print out in this special equal case.

Luckily, python gives us the facility to check for 3 or even more different options and choose at most one of them. This is through the elif branch of the if statement. elif is short for "else if." The general syntax of an if statement with one of these branches is as follows:

```
if <Boolean Expression 1>:
    stmt1a
    ...
    stmtna
elif <Boolean Expression 2>:
    stmt1b
    ...
    stmtmb
else:
    stmt1c
    ...
    stmtpc
stmtA
```

This works as follows: We first check the first Boolean expression. If it's true, we do stmt1a through stmtna, and then skip to stmtA. Alternatively, if this is false, we then check the second Boolean expression. If this one's true, then we execute stmt1b thorugh stmtmb and then skip to stmtA. Finally, if the second Boolean expression is also false, we go to the else clause and execute statemnts stmt1c through stmtpc and then move onto stmtA.

Thus, we can edit the if statement in our work program as follows:

```
    # You've worked enough!
    if this_week > work_week:
        print("You must take",this_week-work_week,"hours of vacation.")

    # Correct hours worked
    elif this_week == work_week:
        print("Perfect, your done for work for the week!")

    # Need to put in some more hours!!!
    else:
        print("You must still work",work_week-this_week,"hours this week.")
```

Grade Example

A very common example given to illustrate an if statement with several clauses is a program that prints out the grade a student should get based on the percentage they earned in a class. In this example, we'll use the typical A (90-100), B (80-89), C(70-79), D(60-69) and F (0-59) breakdown.

```
def main():

    perc = int(input("What is your percentage in class?\n"))

    if perc >= 90:
        print("You got an A!")
    elif perc >= 80:
        print("You got a B!")
    elif perc >= 70:
        print("You got a C.")
    elif perc >= 60:
        print("You got a D.")
    else:
        print("Sorry, you got a F.")

main()
```

Notice that we only need to check one condition for each letter grade since the order in which they are checked. All grades 90 or higher are "caught" by the first clause, so if the second clause (elif perc >= 80) is ever evaluated, then we know that perc must be less than 90. Thus, if this boolean expession is true, it follows that perc is in greater than or equal to 80 AND less than 90. Continuing this logic, each of the first four clauses properly maps to their corresponding letter ranges. The only way the else clause executes is if perc is less than 60.

To note that the order here is important, consider what would happen with the following if statement:

```
    if perc >= 70:
        print("You got an C.")
    elif perc >= 90:
        print("You got a A!")
    elif perc >= 80:
        print("You got a B!")
    elif perc >= 60:
        print("You got a D.")
    else:
        print("Sorry, you got a F.")
```

What would this code segment print out if perc equals 95 right before it? Or 83? Will this code segment ever print out "A" or "B"?

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.

2.3 random class

Computer programs, particularly games, are much more fun if there's some randomness involved. Unfortunately, we have no reliable way of producing truly random numbers. However, most programming languages, including Python, include a pseudorandom number generator. These generators use a set of steps to generate numbers that appear random. Thus, if you or I knew the exact set of steps the generator was using, we could reproduce every numer the generator created. However, to the casual observer, the numbers produced would appear random. Random numbers in programs allow us to play games with some uncertainty (think dice) and allow us to simulate real life events that have some uncertainty (think stock market). Python makes using random numbers fairly easy. In order to do so, we must do the following import:

```
import random
```

At the beginning of our program (preferably in main), we must seed our random number generator as follows:

```
random.seed()
```

For now, the details of this function call are not important. Simply include this as one of the first lines of your main function in any program that uses random numbers.

From this point on, if you want a random number selected between two integers a and b, inclusive, simply make the following function call:

```
random.randint(a, b)
```

Since this function call returns a value, like a majority of the math functions, we must call it as part of a greater line of code, typically storing its return value in a variable.

The following program will use the random number generator to generate one random number in between 1 and 100, and allow two players to guess the number. The winner will be the player who comes closest to the number without guessing too high. If both players guess too high, or if both players guess the same number, the outcome will be a tie. In all other cases there will be a unique winner.

Though this example generates only one random number, a program is allowed to generate many random numbers, if necessary. In these cases, we still only seed the random number generator only once.

```
import random

def main():

    random.seed()

    secretNum = random.randint(1,100)

    # Get the user input.
    guess1 = int(input("Player 1, enter your guess(1-100).\n"))
    guess2 = int(input("Player 2, enter your guess(1-100).\n"))

    # Calculate how close both players are.
    diff1 = secretNum - guess1
    diff2 = secretNum - guess2

    print("The correct number was",secretNum)

    # Check all of the cases!
    if diff1 < 0 and diff2 < 0:
        print("Both players bust. The game is a tie.")
    elif diff2 < 0:
        print("Player 1 wins since Player 2 busted.")
    elif diff1 < 0:
        print("Player 2 wins since Player 1 busted.")
    elif diff1 < diff2:
        print("Player 1 is closer to the correct number and wins!")
    elif diff1 == diff2:
```

```
        print("Both players guessed the same number and tie.")
    else:
        print("Player 2 is closer to the correct number and wins!")

main()
```

In this program, we generate a single random number in between 1 and 100, inclusive and store it in secretNum. After that, we use the if statement to separate out several different cases to determine which of the two players has won. As this example illustrates, a problem that seems so simple to us intuitively can have a rather complex, detailed solution. Though our brain can carry out this "Price is Right" logic effortlessly, we see that when formalized, there are several conditions that need to be checked.

The following section will contain an example that utilizes multiple random numbers in the same program.

2.4 Writing Our Own Functions

Motivation for User-Defined Functions

Up until now, we've only called functions that Python has provided for us. Some examples of the functions we've used are: print, input, int, float, sqrt, round and randint. Python and nearly all programming languages allow users to define their own functions.
Simply put, a function is a mini-program that completes a specified task. For example, the sqrt function takes its input value and returns its square root. The randint function takes in a low bound and a high bound and selects a random integer within the specified range. The ceil function finds the smallest integer greater than or equal to its input value and returns it. We can define functions to do any sort of task that we would like or to make any sort of calculation we want. Once we define a function, we can use it over and over again.