

## **(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 considered 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:

`item_price` 10.99

After the second line, the picture in memory is:

`item_price` 10.99    `is_taxed` "no"

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:

item\_price 10.99    is\_taxed "yes"

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:

item\_price 10.99    is\_taxed "yes"    tax\_rate 6.5

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:

item\_price 10.70435    is\_taxed "yes"    tax\_rate 6.5

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
```

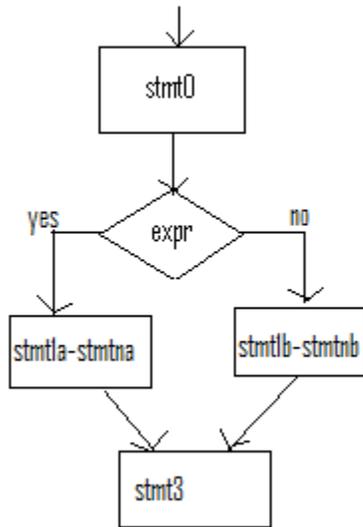
where var is the variable to round. Here is an application of this syntax to displaying the item price rounded:

```
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 through stmtmb and then skip to stmtA. Finally, if the second Boolean expression is also false, we go to the else clause and execute statements 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 expression 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 score 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 number 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.

## *Function Example*

Consider a game where two people both roll a pair of regular dice and the first person wins if their roll sums to a higher number than the other person. (Thus, if both people roll the same value, then the second person wins.) In this program, a task we need to repeat is rolling a pair of dice. First the code will be shown in its entirety and then an explanation will be given for how it works:

```
import random

def main():

    random.seed()

    # Roll both pairs of dice.
    score1 = rollPairDice()
    score2 = rollPairDice()

    print("Player 1 rolled a",score1,"and player 2 rolled a",score2)

    # Print out the winner.
    if score1 > score2:
        print("Player 1, you win!")
    else:
        print("Player 2, you win!")

def rollPairDice():
    return random.randint(1,6) + random.randint(1,6)

main()
```

We first define each of our functions. When we define our functions, they don't actually get interpreted. Rather, we are defining them so that when we call them later, the interpreter knows what the function calls mean. Thus, before we ever call main, both the function main and rollPairDice are defined.

As we interpret main, we run across the call to rollPairDice. We then start executing the steps of this function. This function has a single statement, which, itself, contains a couple function calls. The first randint call will generate a random integer in between 1 and 6 and return it. The second randint call will do the same thing. Note that this second call may very well return a different number than the first call. The function adds these two values and then returns this sum. The return statement is a special statement designated for functions. The return statement indicates that the function is ending and that it is returning a value to the function that called it.

In this case, the rollPairDice function returns a value to main. This value is then stored in the variable score1. We repeat this process, calling the rollPairDice function a second time. Once it finishes, the value it returns gets stored in score2. Note that any time a function returns a value, the function that calls it should store that value in a variable or use it in some way.

Once these lines are done, the program proceeds as we'd expect, comparing the two variables and printing out the appropriate message.

### *Function Example with Different-Sided Dice*

Though most games with dice have their users roll a pair of 6-sided dice, not all games are this way. We can make our function more general by allowing the user to determine the number of sides on each of the two dice. In order for this to occur, our function needs some information. It needs to know how many sides are on each die. This information is given to a function in its parameter list. A parameter list is a list of information the function needs to do its task. Here is our modified function definition:

```
def rollPairDice(numSides):  
    return random.randint(1,numSides) + random.randint(1,numSides)
```

Now, in order to call this function, we must put a number in the parentheses in between `rollPairDice`. Now, let's look at the modified program in full that accomplishes the same exact task as the previous program:

```
import random  
  
def main():  
  
    random.seed()  
  
    # Roll both pairs of dice.  
    score1 = rollPairDice(6)  
    score2 = rollPairDice(6)  
  
    print("Player 1 rolled a",score1,"and player 2 rolled a",score2)  
  
    # Print out the winner.  
    if score1 > score2:  
        print("Player 1, you win!")  
    else:  
        print("Player 2, you win!")  
  
def rollPairDice(numSides):  
    return random.randint(1,numSides) + random.randint(1,numSides)  
  
main()
```

## *Extending Our Program*

Now that we have a more general function, we can use it to write a program that allows users to play this game with a pair of Dice with any number of sides:

```
import random

def main():

    random.seed()

    # Get the number of sides on each die.
    sides = int(input("How many sides are on each die?\n"))

    # Roll both pairs of dice.
    score1 = rollPairDice(sides)
    score2 = rollPairDice(sides)

    print("Player 1 rolled a",score1,"and player 2 rolled a",score2)

    # Print out the winner.
    if score1 > score2:
        print("Player 1, you win!")
    else:
        print("Player 2, you win!")

# Returns the sum of rolling two dice labeled 1 to numSides.
def rollPairDice(numSides):
    return random.randint(1,numSides) + random.randint(1,numSides)

main()
```

In this example, we can see the mechanics of functions working. After the second line of code in main, the picture of memory in main is as follows, assuming that the user enters 20 for the number of sides on a die:

sides 

20
----

Now, consider what occurs when we make the following function call:

```
score1 = rollPairDice(sides)
```

Before the rollPairDice function runs, the first task is assigning each parameter to the appropriate value. Each function gets its own memory for each time it runs. Thus, our picture for the function is as follows:

numSides 20

Essentially, what has happened is that the value of sides in main (20), has been copied into the box for numSides in the function rollPairDice. From here, we run the function, which itself calls the randint function, which will return a random integer in between 1 and 20, inclusive. This function gets called a second time and the second random integer returned is added to the first, and then this sum is returned to main and stored in score1.

This whole process is repeated again. For our purposes, we call the rollPairDice function a second time with the same parameter (sides), but there's no reason why we couldn't call it with a different parameter.

### *Example of a void Function*

Though many functions return values, it's not required for a function to return a value. Some functions may simply carry out a specified task and have no need to return any value to the function that called it. Consider the following function that prints out a given string a certain number of times:

```
def printString(myStr, n):  
    print(myStr*n, sep="", end="")
```

We can use this function to print out a simple design:

```
def main():  
  
    printString(' ', 2)  
    printString('*', 1)  
    printString('\n', 1)  
    printString(' ', 1)  
    printString('*', 3)  
    printString('\n', 1)  
    printString('*', 5)  
    printString('\n', 1)  
    printString(' ', 1)  
    printString('*', 3)  
    printString('\n', 1)  
    printString(' ', 2)  
    printString('*', 1)  
    printString('\n', 1)
```

```
main()
```

Note: It's clear that for this particular example, the use of the function has NOT eased our task. Rather, this example is being used to illustrate how a void function (a function that doesn't return anything) works. In particular, it's just as easy to directly call the print function in each of these instances instead of going through our defined printString function. One thing to note is that the printString function is quite flexible. We can give it any string we want, and any number of repetitions. On the first function call, our picture of memory in the printString function is as follows:



This will then print out two space characters. In the following function call, the picture of memory in the printString function is as follows:



In this manner, we can use the printString function in various ways to get our desired output result:

```
*
***
*****
***
*
```

## 2.5 Blackjack Example

### *Function Reusability*

One advantage to writing functions is that once a function is written, if it's written in a general manner, like the rollPairDice function, it can be used multiple times. There are two clear benefits to using functions in this context. One benefit is that programs with functions that are reused tend to be shorter to write. Secondly, and more importantly, if we write a function and find all the errors in it, then we can call that function many times and be fairly confident that if our program has errors, the errors aren't in the function. Without the function, different errors could creep up in each "copy" of the code that essentially does the same thing.

### *Blackjack Program*

In the following example, we will simulate a simple game of blackjack with some modifications, based on the amount of Python we currently know. In blackjack, each player is initially dealt two cards. The value of each card ranges from 2 to 11. Number cards are worth their value, face

cards are worth 10 points and aces can be worth either 1 or 11 points. In this version, we'll make all aces worth 11 points to simplify the simulation. In regular blackjack, a player may deal as many extra cards as she wants. The goal of the game is to get as close to 21 points as possible, without going over. The winner is whoever gets the most points without exceeding 21. In a casino, there can be multiple winners – anyone who beats the dealer wins in a casino. But, in our version of the game, there will be two players. If both exceed 21 or both get the same score less than or equal to 21, we will declare no winner. Otherwise, whoever has the greatest score less than or equal to 21 will be the winner. Since we don't yet know how to repeat a set of statements many times, we will allow each player to only receive one extra card, if they chose.

We will employ four functions in our implementation:

- 1) `getCards(n)` – returns the sum of `n` cards, where `n` is either 1 or 2.
- 2) `oneCard()` – returns the score of one randomly selected card
- 3) `wantCard(name, curscore)` – returns `True` if `name`, who currently has `curscore` points wants an extra card, `False` otherwise.
- 4) `printOutcome(name1, score1, name2, score2)` – prints the outcome of a game between `name1` and `name2` where they've scored `score1` and `score2` points, respectively.

The most difficult part of seeing this example in its entirety is conceiving how it was designed. The first three functions return values while the last one returns nothing. Each of the four functions takes in different pieces of information. It takes time to get comfortable designing functions in programs and putting everything together. At this point, don't be concerned if you don't feel that you could design a program this complex. Designing programs that use functions effectively takes time and students gradually get better at the process. The goal of this program is to provide a more complicated example that clearly exemplifies some of the benefits of functions that aren't clear from smaller examples. (As an exercise, try rewriting this program all in main, and the benefit of functions will be crystal clear!) Download the example and run it a few times. Though an explanation will be given here, try to figure out how each function communicates with one another. There are two lines of communication: input a function receives to do its job and a return value. Functions receive input in the form of parameters. This is the name of the items inside the parentheses in a function call. When the function is done, it potentially returns a value to the function that called it. The receiving function is then responsible for using this value as it sees fit.

For example, the `getCards` function takes in an integer. This integer tells it how many cards to give the player. The function's job however, is to simply return the total score of these cards. The specific information about the cards themselves is actually lost. (If you run the program, you realize that you never find out which specific cards you hold.)

The `oneCard` function has a more simple task: It returns the value of a single random card. As you might imagine, once this function is written, we can easily write the `getCards` function by calling the `oneCard` function once or twice, as is necessary. The nature of the `getCards` function

is general enough that we can call it for the initial deal OR to obtain the extra card. Let's look at both of these functions:

```
# Pre-condition: n must be either 1 or 2.
def getCards(n):

    # We must have at least one card.
    total = oneCard()

    # Add a second card, if necessary.
    if n == 2:
        total = total + oneCard()

    return total

# Returns a random blackjack card value, assuming
# all aces are worth 11 points.
def oneCard():

    card = random.randint(2, 14)

    # Numeric cards
    if card <= 10:
        return card

    # Face cards
    elif card <= 13:
        return 10

    # Aces!
    else:
        return 11
```

Notice that in the `oneCard` function, we simulate a random card by a random number in a 13 value range, to correspond to the 13 kinds of cards. Then, we assign each number to a specific kind of card. It's easiest to assign 2 – 10 to the same values, 11 – 13 to the face cards, and 14 to an Ace. It would have been equally valid to generate a random number in between 1 and 13, inclusive and assign Aces to 1. In fact, any system where each kind has a 1/13 chance of being chosen would be valid. Once we make our selection, we calculate the appropriate number of points and return it!

In the `getCards` function, we call the `oneCard` function once, and then use an `if` statement to check to see if we should call it again.

Let's take a look at the program in its entirety:

```

# Arup Guha
# 8/13/2012
# Simplified Blackjack Example with Functions

import random

def main():

    name1 = input("Player #1, what is your name?\n")
    name2 = input("Player #2, what is your name?\n")

    score1 = getCards(2)
    score2 = getCards(2)

    if wantCard(name1, score1):
        score1 = score1 + getCards(1)

    if wantCard(name2, score2):
        score2 = score2 + getCards(1)

    printOutcome(name1, score1, name2, score2)

# Pre-condition: n must be either 1 or 2.
def getCards(n):

    # We must have at least one card.
    total = oneCard()

    # Add a second card, if necessary.
    if n == 2:
        total = total + oneCard()

    return total

# Returns a random blackjack card value, assuming
# all aces are worth 11 points.
def oneCard():

    card = random.randint(2, 14)

    # Numeric cards
    if card <= 10:
        return card

    # Face cards
    elif card <= 13:
        return 10

    # Aces!
    else:
        return 11

```

```

# Returns true iff name wants an extra card.
def wantCard(name, curscore):

    # Get the user's answer.
    print(name,", your score is currently ",curscore, sep="")
    answer = input("Would you like another card(yes/no)?\n")

    # Return accordingly.
    if answer == "yes":
        return True
    else:
        return False

# Prints the outcome of a game between name1 and name2
# where they've scored score1 and score2, respectively.
def printOutcome(name1, score1, name2, score2):

    # Print final scores.
    print(name1,", your score is ",score1,".", sep="")
    print(name2,", your score is ",score2,".", sep="")

    # Lots of cases here. Go through each.
    if score1 > 21 and score2 > 21:
        print("Sorry, no one wins, both players busted.")
    elif score2 > 21:
        print(name1,"wins because", name2, "busted.")
    elif score1 > 21:
        print(name2,"wins because", name1,"busted.")
    elif score1 > score2:
        print(name1,"wins with a higher score than",name2)
    elif score2 > score1:
        print(name2,"wins with a higher score than",name1)
    else:
        print("Both",name1,"and",name2,"tie.")

# Call main!
main()

```

## 2.6 Practice Programs

1) Write a program that does temperature conversion from either Fahrenheit to Celsius, or the other way around. Your program should first prompt the user for which type of conversion they want to do. Then your program should prompt the user for the temperature they want to convert. Finally, your program should output the proper converted temperature. The formula for conversion from Celsius to Fahrenheit is:

$$F = 1.8 * C + 32$$

2) Debbie likes numbers that have the same tens digit and units digit. For example, Debbie likes 133 and 812355, but she does not like 137 or 4. Write a program that asks the user for a number and then prints out whether or not Debbie likes the number.

3) Write a program that prompts the user for 2 pieces of information: (1) age, (2) amt. of cash they have. Based upon these inputs, your program should produce one of the four outputs below for the given situations:

Situation	Output
$A < 21, M < 100$	"You have some time before you need more money."
$A < 21, M \geq 100$	"You have got it made!"
$A \geq 21, M < 100$	"You need to get a job!"
$A \geq 21, M \geq 100$	"You are right on track."

4) Scholarships are given based on students' SAT score and GPA. In particular, based on these two items, a composite score is calculated with the following formula:

$$(\text{SAT score})/1000 + \text{GPA}$$

(Note: The formula refers to real number division.) The amount of scholarship money awarded is based on this composite score as follows:

Composite Score (s)	Scholarship Amount
$4 < s < 5$	\$1000
$5 \leq s < 6$	\$2500
$s \geq 6$	\$5000

Write a program that asks the user for their SAT score and GPA and prints out the amount of scholarship money they receive.

5) Write a guessing game for two players where both players guess a number in between 1 and a 100. Your program should generate the "secret number" randomly. The winner is determined as follows:

If both players guess the same number, the first player wins.

If one player's guess is closer than the other player, then that player wins.

If both players' guesses are off by the same value, then the player that guessed the lower number wins.

For example, if the secret number is 71 and the two players guesses are 64 and 77, then the player that guesses 77 wins, since  $77 - 71 = 6$  and  $71 - 64 = 7$ . Alternatively, if the secret number is 45 and the two players guesses are 36 and 54, then the player that guesses 36 wins, since  $45 - 36 = 9$ ,  $54 - 45 = 9$ , and 36 is less than 54.

6) Sales clerks at "Computers R Us" get paid a bonus for their sales. In particular, they get \$10 for each of the first 10 computers they sell in a month, \$20 for the next 10 computers they sell in a month, and \$40 for each computer thereafter. For example, if a clerk sells 15 computers, they make \$100 for the first 10 and \$100 for the next 5 for a total of \$200. Alternatively, if a clerk sells 22 computers, they make \$380 in bonuses. Write a program that asks the user how many computers they sold and prints out their bonus for the month.

7) Write a program that calculates the number of pitchers of lemonade that can be made with certain raw materials. Ask the user to enter the number of teaspoons of sugar and number of lemons necessary to make a pitcher of lemonade. Follow this by asking the user the total number of teaspoons of sugar and number of lemons that are available to make lemonade. Use this information to calculate the number of full pitchers of lemonade that can be made and also print out the leftover ingredients (number of teaspoons of sugar and lemons) after those pitchers have been made.

8) Two competing companies offer different bulk buying plans for buying boxes of cereal. The first 100 boxes are at one price point, the next 1000 are at a second price point and all subsequent boxes are at a third price point. Ask the user to enter the three price points for two companies as well as the number of boxes they desire to buy. Your program should print out which company they should go with and how much their purchase will cost. If both companies will provide the same price, you may choose either. Consider the price points of two companies shown below:

Company	Price for first 100	Price for next 1000	Price for rest
1	\$2.99	\$1.99	\$1.50
2	\$2.50	\$2.25	\$1.75

If we buy 2000 boxes of cereal from company 1, we spend  $100 * \$2.99 + 1000 * \$1.99 + 900 * \$1.50 = \$3639$ . If we buy the same boxes from company 2, we spend  $100 * \$2.50 + 1000 * \$2.25 + 900 * \$1.75 = \$4075$ . Thus, in this case, we should go with company 1.

9) On planet C, leap years occur on every year divisible by 7, except for years that are divisible by either 35 or 77. (Thus, the first ten leap years after 0 on planet C are: 7, 14, 21, 28, 42, 49, 56, 63, 84 and 91.) Write a program that prompts the user to enter a year (positive integer) and prints out whether or not this year is a leap year on planet C.

10) Write a program that determines whether or not Anna and Bob were at the restaurant at the same time. Ask the user when Anna and Bob arrived at and left the restaurant, respectively. (The user should answer in number of minutes after midnight, and assume that both arrived and left on the same day.) Determine whether or not Anna and Bob were at the restaurant at the same time, and if so, for how many minutes they overlapped. For example, if Anna arrived at 720 minutes after midnight and left 800 minutes after midnight and Bob arrived at 600 minutes after midnight and left 740 minutes after midnight, they were at the restaurant together for 20 minutes. (Note: If Bob leaves at the same time Anna arrives, then they were not at the restaurant at the same time.)