Publication Pending
Author: Dr. Ami Gates
Date: 8/11/2016

Chapter 4: Loops and Decisions

Loops structures, such as the `for` loop, the `while` loop, and many others, can be found in most high level programming languages. Loops (or repeating code statements) allows for a group of code to complete the same type of task (possibly with different inputs or values each time) until a given goal or job is done. Repeating is as common in programming as it is in everyday life. As an odd, but true example, I drink my morning coffee in a loop. Once I am sitting in front of my coffee, I reach over, pick it up by the handle, sip some, make an "mmmmm" sound, and put it back on the table. I repeat these steps until the coffee is done. Humans and programs require the option to repeat or loop.

Decision structures, such as `if/elif/else`, allow code to make decisions and then enact given behaviors based on those decisions. In programming, decisions must be **deterministic**. In other words, it must be logically possible for the computer to determine which decision to make. It is not possible (yet) for computers to decide for themselves completely. Even though there are methods that might simulate decision making, they are still deterministic.

The following sections will review and offer examples for loop and decision structures, as well as **flowcharts**.

4.1: For/in Loops

It is often the case, when writing a program, that sections of code need to be repeated, such as collecting the first and last names of many people, or calculating the factorial of a number. One of the most commonly used looping structures is the for/in loop. The `for/in` loop is very versatile in Python and can loop through lists, names, variables, ranges, and other structures.

The `for/in` loop for lists has the following basic syntax:

```
for <variable name> in <list>:
     <statements>
```

Example 4.1.1: The for/in loop and a list.

The following small program illustrates the syntax and a utilization of the for/in loop structure.

```
# ForInLoopChapter4.py
# by Ami Gates
# This code offers an example of the for/in loop structure

def main():
    for k in [1, 2, 3, 4]:
        print(k)


main ()
```

The output for this program is the following:

```
1
2
3
4
```

There are a few things to note here. First, the function used in this program is called `main()`. The `main()` function in any program is generally considered the main focus or starting point for the overall program. Other functions can be called from `main()`, and `main()` can control the overall order of execution (the flow) of the program.

The above program would have worked correctly no matter what name was used for the function. In other words, the function `main()` could have been `SomeFunctionName()` instead. However, for reasons not yet discussed, it is best to use `main()` as the primary function of a program.

Next, notice the `for/in` loop. The variable named *k* is used to hold each value in the given **list** of values. In other words, k will equal 1 first and all following lines of code will execute with k having the value of 1. Next, k will equal 2, and all following lines of code will execute with k having the value of 2. The same is true for 3, and finally 4. Because the only statement in the body of the for/in loop is a print statement, each value of k is printed as output to the console.

Remember that calling the main function by using the statement `main`(), which is at the end of this program example, is required for the function main to execute. As an experiment, remove the last line of code (the call the `main ( )`) and see what happens. The result will be that main ( ) is never called and there is no output. However, there will also be no error. Not calling main is not a syntax error, it is a programming choice. One could argue that not calling the main function is a logical error as the program cannot perform its goal without the call.

Example 4.1.2: The for/in loop using range.

The `for/in` loop for range has the following basic syntax:

```
for <variable name> in range(<integer value>):
     <statements>
```

The range function can create a sequence of values. The syntax for the range function is:

```
range(start, stop, [step])
```

The `start` is the first value in the sequence. The `stop` will not be in the sequence and will be the value after the last value in the sequence. For example, `range(1,5)` is `1, 2, 3, 4`. The `step` is optional and is the integer amount that each value in the sequence is separated by. For example, `range(1,10,2)` is `1, 3, 5, 7, 9`.

As a note, the `start, stop`, and `step` are called **arguments** or **parameters** to the range function. The only parameter that is required in the range function is the value of start. All parameter values must be integers.

The for/in loop can loop through any range. The following small program offers three illustrations of the for/in loop with range.

```
# ForInLoopRangeChapter4.py
# by Ami Gates
# This code offers examples of for/in and range()

def main():
    # for/in loop basic range(start)
    print("Range(start) example")
    for j in range(5):
          print(j)

    # for/in loop basic range(start,stop)
    print("Range(start,stop) example")
    for val in range(5,10):
          print(val)

    # for/in loop basic range9start, stop, step)
    print("Range(start,stop,step) example")
    for item in range(2,11,2):
          print(item)

main()
```

The output for this program is:

```
Range(start) example
0
1
2
3
4
Range(start,stop) example
5
6
7
8
9
Range(start,stop,step) example
2
4
6
8
10
```

Ranges and lists can be equivalent. For example, if `range(6)`is used, it is equivalent to the list `[0, 1, 2, 3, 4, 5]`. All ranges start at "0", unless a start parameter and a stop parameter are both used in the range function. For example, `range(6)` is `0, 1, 2, 3, 4, 5`, but `range(2,6)` is `2, 3, 4, 5.`

Within each of the three for/in examples above, there is a variable that holds the current value in the loop. For example, in the code:

```
print("Range(start) example")
for j in range(5):
     print(j)
```

The variable `j` holds the value 0, then 1, then 2, and so on until the end of the range, which is 4.

In this code:

```
print("Range(start,stop,step) example")
for item in range(2,11,2):
     print(item)
```

The variable `item` holds the value 2, then 4, then 6, etc. until the end of the range (which is 10).

Exercise 4.1.1: For/in loops and calculations

Complete the following steps:
1. Create a new .py file called Chapter4Loops.py and save it.
2. Add comments to the top that include the file name, your name, and what the program does.
3. Write a small program that asks a user for an integer Fahrenheit temperature number between 30 and 80.
4. Use a for/in loop to output *every other number* between the user input value and the user input value plus 10.
5. Use another for/in loop to output all equivalent Celsius temperatures starting with the user input and including the next two following higher temperatures (in increments of one). For example, if the user enters 60, the Celsius outputs will be for 60, 61, and 62, and so would be 15.56, 16.11, 16.67.
6. Place all code inside one main function. Remember to call main at the end. Use print statements as needed to clarify output.

Solution and Review for Exercise 4.1.1:

One possible solution to this exercise is the following program.

```
# Chapter4Loops.py
# by Ami Gates
# This program illustrates for/in loops

def main():
    TempF=eval(input("Enter temp in F between 30 and 80 with no 
decimals: "))

    print("Every other number from ",TempF, " to ", TempF+10)
    for i in range(TempF,TempF+10,2):
        print(i)

    print("Celsius temperatures for ", TempF, " and ", TempF+1, 
" and ", TempF+2)
    for t in range(TempF, TempF+3):
        TempC= (t -32)* (5/9)
        print(round(TempC,2))



main()
```

An input and output for this example program is:

```
Enter temperature in F between 30 and 80 with no decimals:
60
Every other number from  60  to  70
60
62
64
66
68
Celsius temperatures for  60  and  61  and  62
15.56
16.11
16.67
```

In this exercise, consider the code:

```
for i in range(TempF,TempF+10,2):
        print(i)
```

Here, the start of the range is the temperature the user enters. The stopping point (not included in the sequence) is the temperature plus 10, and the step is 2. This will create an output of every other value between the temperature entered and temperature+10. The print statement prints each of the values.

Next, consider the code:

```
for t in range(TempF, TempF+3):
        TempC= (t -32)* (5/9)
        print(round(TempC,2))
```

Here, the range starts at the temperature the user enters and ends at the user temperature plus three. Why plus three? Because the goal is to print the Celsius equivalents for the temperatures entered and the next two. Recall that the stop in the range is not included. Change this value to 2 and see that it does not work as desired.

Notice that inside of for/in loops there can be calculations, expressions, and other statements. Inside of loops can be other loops, decision structures, functions, and so on.

Example 4.1.3: Stepping through code

When first learning about loops, it can be very helpful to write out the values for all involved variables as the loop runs-- from beginning to end. This is called **stepping through code**. Many debuggers also offer an advanced version of this concept, but at the start of coding, it is more valuable to do this by hand.

Consider this code:

```
# SumProdLoop.py
# Ami Gates
def main():
    counter=0
    k = 1
    for i in [1,2,3,4]:
        counter=counter+i
        k = k * i
    print("The counter is ", counter)
    print("The k value is ", k)
main()
```

The output for this program is:

```
The counter is  10
The k value is  24
```

Step through this loop, keeping track of the values of all variables:

```
Before entering the loop:
      counter = 0
      k = 1

Enter the loop
Step 1
      i = 1
      counter = counter + i  ➔    counter = 0 + 1  ➔ counter = 1
      k = k * i ➔ k = 1 * 1 ➔ k = 1

Step 2
      i = 2
      counter = counter + i  ➔    counter = 1 + 2  ➔ counter = 3
      k = k * i ➔ k = 1 * 2 ➔ k = 2

Step 3
      i = 3
```

```
        counter = counter + i  ➔    counter = 3 + 3  ➔ counter = 6
        k = k * i ➔ k = 2 * 3 ➔ k = 6

Step 4
        i = 4
        counter = counter + i  ➔    counter = 6 + 4  ➔ counter = 10
        k = k * i ➔ k = 6 * 4 ➔ k = 24
```

At the end of the program, the value of `counter` is 10 and the value of `k` is 24. This ability to update and manipulate variables within loops is a critical part of programming logic.

Thus far, examples have shown that the for/in loop can iterate through values in a range or values in a list. In addition, loops, such as the for/in loop, can iterate through strings. A **string** is any collection of characters, special characters (like $ or #), or numbers. Strings are always contained in either single or double quotes. The following are all examples of strings.

```
name="Bob Smith"

type(name)
Out[66]: str

sentence1="You go first!"

type(sentence1)
Out[68]: str

phrase="The shop collected $45,000.43 in 2016!!"

type(phrase)
Out[70]: str
```

In each case above, the string is contained in quotes. The `type()` function returns the type (string, int, float, etc.) of data the variable represents. Note that when a number is contained in quotes, it is considered a string (just like any other word, sentence, or phrase), and so does not have any true numerical value. If a number is saved or collected as a string, it cannot be manipulated like a number. In other words, it cannot be part of a mathematical calculation. It is possible to convert a string number into a true number using the function called `int()`; or to create a decimal number, the function called `float`(). These concepts will be discussed in more detail in the chapter on data types. The following code also illustrates this idea.

```
x = "45.67"
```

```
type(x)
Out[72]: str

z = x+3
Traceback (most recent call last):

  File "<ipython-input-73-a486c061ffc6>", line 1, in
<module>
    z = x+3

TypeError: Can't convert 'int' object to str implicitly


x = float(x)

z = x+3

z
Out[76]: 48.67
```

In the above example code, the variable x is set equal to 48.67 as a string. The type function confirms that the type of x is str (string). Next, if x is added to a number, an error occurs because x is not a number itself – it is a string. Next, if x is converted to a float (a decimal number), it can then be added to the number 3 successfully.

While strings, string manipulation and types will be covered more thoroughly in a later chapter, it is valuable to illustrate a for/in loop with a list of strings.

Example 4.1.4: The for/in loop with strings

The following program illustrates the use of the for/in loop with a list of strings.

```
# Chapter4LoopsStrings.py
# by Ami Gates
# This program illustrates the for/in loop w/strings

## main function
def main():
    Namer()
```

```
## Definition of the Namer function
def Namer():
    x = ""
    for w in ["Dr. ", "Henry ", "Hughs"]:
        print(w)
        x = x + w

    print(x)


## Call to main()
main()
```

The output for this code is:

```
Dr.
Henry
Hughs
Dr. Henry Hughs
```

Key elements in this program:

1) The definition of two functions, `main()` and `Namer()`.
2) The use of string concatenation, the + symbol.
3) Traversing through a list of strings using `for/in`.

There are several details to discuss about this example program. Notice first, the extra use of **comments**. Recall that the # symbol will turn any line into a comment. The # symbol tells the interpreter to skip that line during the execution of the program. The use of comments can be very personalized. For example, I used ## for comments within the body of the program. This is not required and is a style for ease of reading. Notice also that the comments explain and describe parts of the program, making it easier to read and to understand. Commenting is invaluable in coding.

This program defines two functions, `main( )` and `Namer()`. The Namer function is defined outside of the main function and is called from within the main function. Functions will be discussed in detail in a later chapter. However, a program can have as many defined functions as desired. Functions can be defined inside or outside of other functions.

Inside of the definition of the `Namer()` function, the first statement is an assignment of the variable *x*. Here, `x = ""`, which is the same as saying that *x* takes on the value of an empty or blank string. Next, the for/in loop begins. There are two statements inside the for/in loop. The first statement prints the value of the variable `w` which is assigned each value in the list of strings as the loop progresses. Remember, a string is any collection of characters or symbols that are inside of a set of quotes. Therefore, in this case, the list contains three strings, "Dr. ", "Henry ", "Hughs". The space after Dr. is part of the string, as is the space after Henry.

As the for/in loop executes, the first value assigned to `w` is `"Dr.  "`. Inside of the loop, the print statement will print the value of `w` and so will print `Dr.  `.

The next statement in the loop is the following:

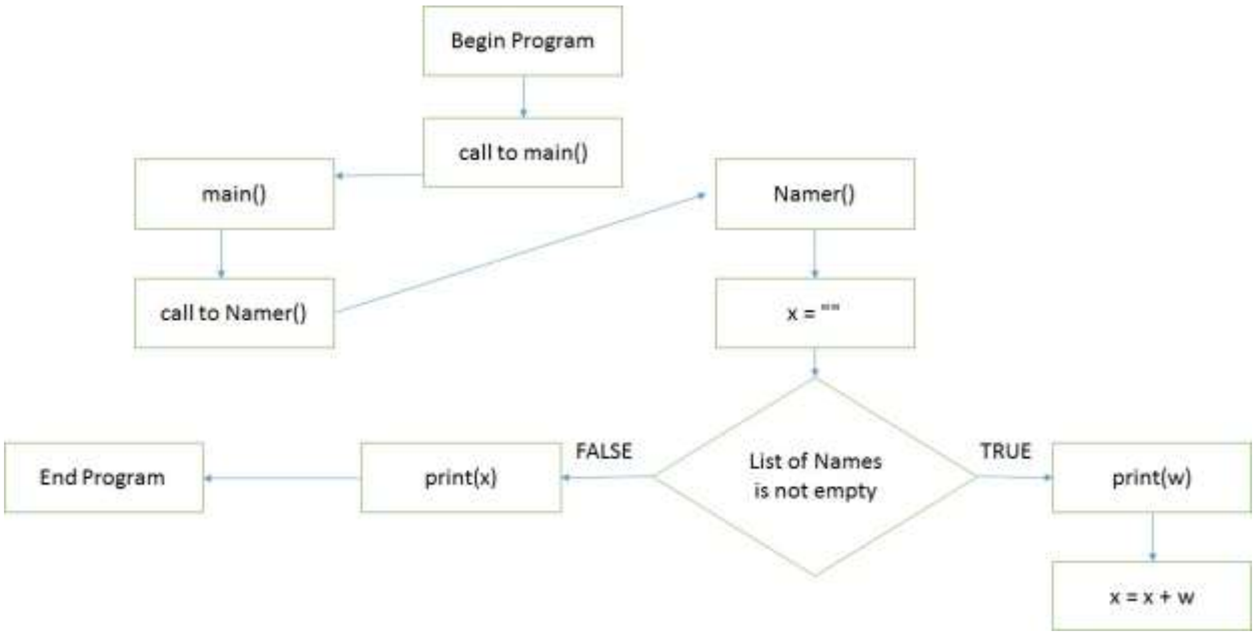
```
x = x + w
```

Both `x` and `w` are strings (words). So, how can they add together? While this topic will be covered in great detail in the chapter on strings, this is an introduction to the idea of **string concatenation**, which is a fancy word for combining or placing two strings together to make one longer string. Therefore, the string value in the variable *x* will be the combination of blank (because *x* is currently an empty string at this point in the loop) and the string in *w*. The following illustrates the step through of the for/in loop:

```
x = ""

Loop1:
     w = "Dr. "
     x = x + w = "Dr. "

Loop2:
     w = "Henry "
     x = x + w = "Dr. Henry "

Loop3:
     w = "Hughs"
     x = x + w = "Dr. Henry Hughs"
```

Therefore, the print statement outside of the loop will print `Dr. Henry Hughs`. To visualize the flow of this example program, a flowchart can be created.

Example 4.1.5: Flowchart for example 4.1.4



Once the `print(x)` statement in the function `Namer()` has executed, the program returns to `main()`. However, there are no further statements in `main()` and so the program ends. As an exercise, add a print statement to the main function under the call to the Namer function and evaluate the result.

For/in loops are often considered to be predetermined loops. The number of times a for/in loop will run is determined by the size of the range or the length of the list. In some cases, it is necessary for a loop to repeat for a variable number of times, based on a condition. The `while` loop offers this option.

4.2: While Loops

A **while loop** will repeat itself until a **logical condition** is met. One issue with while loops, as opposed to for/in loops is that if the while loop is not properly controlled, it can be **infinite** and will eventually use excess resources and fail. The syntax for the while loop is the following:

```
while <condition is true>:
     <statements>
```

Before discussing logical and logical conditions, review the next few examples for while loops.

Example 4.2.1: A Simple while loop example

```
# WhileLoop1.py
# by Ami Gates
# Code for a simple while loop

def main():
    i = 10
    while i < 20:
        print(i)
        i = i + 2
main()
```

The output for this small program is the following:

```
10
12
14
16
18
```

There are a few things to notice in this example. The first is that the variable `i` was initialized to 10. Next, the while loop condition is, `i < 20`. Notice also that inside the while loop, the value of `i` is increased by 2 each time the loop runs. There is also a print statement inside the loop that prints all the values that `i` takes on throughout the loop. Specifically, because `i` starts at 10, the first print statement will print the value 10, as shown in the output.

Next, the value of `i` is incremented by 2 so that its new value is 12. The condition of the while loop is then checked to determine whether or not to repeat the loop or exit the loop. The condition of the while loop is , `i < 20`. At this point, `i` is 12, which is less than 20. Thus, the while loop condition is **True** and the loop continues to run. The loop with stop running when the condition becomes **False**. The following step through illustrates the loops and the outputs at each iteration of the loop:

```
i = 10
Loop 1: i < 20? True, enter loop
print (i)


i = i + 2 = 10 + 2 = 12
Loop 2: i < 20? True. Enter loop.
```

```
print (i)

i = i + 2 = 12 + 2 = 14
Loop 3: i < 20? True. Enter loop.
print (i)

i = i + 2 = 14 + 2 = 16
Loop 4: i < 20? True. Enter loop.
print (i)

i = i + 2 = 16 + 2 = 18
Loop 5: i < 20? True. Enter loop.
print (i)

i = i + 2 = 18 + 2 = 20
Loop 6: i < 20? False. Do not enter loop. Loop ends.
```

This example illustrates that the while loop checks its condition each time the loop is entered. The loop repeats until the condition is False. If the condition is never False, the loop with be infinite and will eventually crash. Loop conditions can also be determined by user input, as well as by other variable values.

Example 4.4.2: A while loop based on user input

In this next example, a while loop condition will be determined by user input. In the following program, the user is asked to input a number between 1 and 10. The program gets the input from the user with the `input` function and evaluates the input using `eval`. Recall that the eval function is needed when input is numerical.

Next, the program initializes the variable called `guess` to the number 1. The `while loop` checks the condition of equality between `num`, the user input number, and `guess`, the variable holding the current programmed random guess for the what the user's number is. The loop will stop when condition `guess!=num` is False. Note that the `!=` symbol stands for *not equal to*.

```
# WhileLoop2.py
# by Ami Gates
# Example program to illustrate a while loop conditioned on
input

import random
def main():
    num=eval(input("Choose any number between 1 and 10: "))
    counter=0
    guess=1
    while guess != num:
        guess=random.randint(1,10)
        print("Is your number a ", guess)
        counter=counter+1

    print("I guessed your number in ", counter, "tries.")

main()
```

One possible output of this program might be the following:

```
Choose any number between 1 and 10: 3
Is your number a  5
Is your number a  6
Is your number a  4
Is your number a  5
Is your number a  3
I guessed your number in  5 tries.
```

This program contains a few new elements. First, this program **imports** a module called `random`. The syntax for importing a Python module is:

```
import <Module Name>
```

A **module** is a collection of code that offers certain functionality. Modules will be discussed in detail in future chapters. By importing the `random` module using `import,` the program can gain access to the methods and functions contained in that module. There are hundreds of modules and packages available for Python, and they are generally selected based on the need for additional functionality.

One such method contained in the `random` module is called, `randint(a, b).` The `randint(a, b)` function allows the user to generate random integers between any given values of `a` and `b`. In the program above, consider the statement:

```
guess = random.randint(1,10)
```

This statement creates a variable called `guess`. Then, `randint(1,10)` will generate a random integer between 1 and 10 and assign it to `guess`. As noted, Python 3 has hundreds of ready-made modules, each with methods and functions that can be utilized. There is no magic trick for knowing all possible modules, and the best approach to learning about a new module is to perform an Internet or textbook search on a goal (such as creating a random number between 1 and 10 that is an integer) and then viewing the results.

Program Example Review by Line of Code

```
1. import random
2. def main():

3.      num=eval(input("Choose any number between 1 and 10: "))
4.      counter=0
5.      guess=1
6.      while guess != num:
7.            guess=random.randint(1,10)
8.            print("Is your number a ", guess)
9.            counter=counter+1

10.     print("I guessed your number in ", counter, "tries.")

11. main()
```

Code line 1:

Imports the module called random so that the method random.randint(a,b) can be used.

Code line 2:

Start the definition of the main function. All lines of code indented under main belong to main. Recall that indentation denotes scope.

Code line 3:

The variable called num is assigned the value input by the user. The eval is required as the number is numerical and not a string.

Code lines 4 and 5:

The variable called counter is initialized to the value 0 and will keep track of the number of cycles in the while loop.

The variable called guess is initialize to the value of 1, and will keep track of the latest random number generated by line 7: `guess=random.randint(1,10)`

Code line 6:

The while loop is defined with condition: `guess != num` and it will repeat as long as the condition is True.

Code line 7:

The variable guess is assigned to the result of calling the `random.randint(1,10)` which will generate a random integer between 1 and 10.

Code lines 8 and 9:

Line 8 prints out the current guess and line 9 updates the counter. Line 9 is the end of the loop. Once line 9 executes, the program returns to the condition of the while loop. If the condition is True, the loop repeats. If the condition is not True, the program exits the loop and progresses to line 10.

Code line 10:

Line 10 is outside of the while loop and will execute only when the loop ends (when `guess!= num` is False). The pint statement will print the number of tries it took to guess the number. This value is stored in the `counter` variable.

Code line 11:

This is actually the first line of code that is executed in the program, after the import of random. This line of code calls the main function and allows all the statements inside of main to run. Without line 11, this code would not do anything. Try it and see.

One of the key differences between the `for/in` loop and the `while` loop that the ability to run indefinitely. The for/in loop cannot run forever, and the list or range of values that it will run for can be predetermined. The while loop can run forever. This is often referred to as an **infinite loop**, which is considered a computer error, unless it is purposeful and the loop contains other statements or conditions that can end it (a topic for later chapters). When creating a while loop, pay special consideration to the condition being tested, *whether* it is true or false, *when* it is true or false, and that it will eventually be false and end.

4.3: Logical Boolean Operators

Many loop structures in programming depend on **logical conditions**. In the while loop, for example, the logical condition must be evaluated as “true” before the loop can be entered. Similarly, once the logical condition of the while loop becomes “false”, the while loop cannot be entered, and so will end.

Consider the logical condition, `guess!=num`. Here, the logical operator is the `!=` symbol, which stands for “not equal to”. If the value in the variable `guess` and the value in the variable `num` are not equal to each other, then the logical condition, `guess!=num`, is “true”. However, if the value in the variable `guess` and the value in the variable `num` are equal to each other, then the logical condition, `guess!=num`, is “false”.

In Python (and most other programming languages), there are a set of **Boolean logical operators**. Table 4.1 illustrates these operators in example format.

Operator	Meaning
==	Is equal to
>=	Is greater or equal to
<=	Is less than or equal to
>	Is strictly greater than
<	Is strictly less than
!=	Is not equal to

Table 4.1: Logical Boolean Operators

Logical operators can be investigated using the IPython console. The following example illustrates a few logical conditions and their **truth values** (True or False).
Example 4.3.1: Logical condition examples using the console.

```
a, b, c = 1, 2, 2

a == b
Out[107]: False

b == c
Out[108]: True

a > b
Out[109]: False

a <= c
Out[110]: True

a > c and b > c
Out[111]: False
```

In this example illustration, a multiple assignment statement is first entered. The statement,

```
a, b, c = 1, 2, 2
```

assigns the value of 1 to a, the value of 2 to b, and the value of 2 to c. Next, when `a==b` is entered, the resulting output is false (because 1 does not equal 2). Similarly, when `a <= c` is entered, the result output is true. Logical conditions are often used to control loops in programs, as well as other decision structures.

In the example above, the logical expression, `a > c and b > c` is evaluated and the result output is false. Note the use of the `and` in the logical expression. The logical operators, *`and`*, *`or`*, and *`not`* are also known as **Boolean operators**.

Table 4.2 is a **truth table** and illustrates a few basic **Boolean logic** rules. While there are many options for creating Boolean statements, the evaluation of each will be either true or false, depending on the truth of the starting variables. Note that R and S represent any two statements or variables. The first row of the table assumes that both R and S are True, and then displays the truth evaluation for compound logical statements that follow. Each row represents the truth evaluation for each statement, given the initial truth values of R and S.

R	**S**	**R and S**	**R or S**	**(R and S) or R**	**not S and R**
True	True	True	True	True	False
True	False	False	True	True	True
False	True	False	True	False	False
False	False	False	False	False	False

Table 4.2: Logical Truth Table

Both logical operators and Boolean operators can be utilized in the conditions of loops and other decision structures.

Example 4.3.2: Loops and Boolean conditions

Consider the following program.

```
      # BooleanWhile.py
      # by Ami Gates
      # Illustrates a while loop with a boolean condition
      # The main function will call the And_fun function and the
      # Or_fun function.

      def main():
```

```
    And_fun()
    Or_fun()

# The And_fun function creates three variables and assigns
# values to them. It then illustrates a while loop that
# contains a boolean "and" condition.
def And_fun():

    a = 1
    b = 5
    c = 4

    while a < b and a < c:
        print("In the And_fun we have: ", a, b, c)
        a = a + 1

# The Or_fun function creates three variables and assigns
# values to them. It then illustrates a while loop that
# contains a boolean "or" condition.
def Or_fun():
    a = 1
    b = 5
    c = 4

    while a < b or a < c:
        print("In the Or_fun we have: ", a, b, c)
        a = a + 1

main()
```

The output for this program is:

```
In the And_fun we have:  1 5 4
In the And_fun we have:  2 5 4
In the And_fun we have:  3 5 4
In the Or_fun we have:  1 5 4
In the Or_fun we have:  2 5 4
In the Or_fun we have:  3 5 4
In the Or_fun we have:  4 5 4
```

There are several items of note in this example program.

First, there are three functions. The first is `main(),` the second is `And_fun()`, and the third is `Or_fun()`. The main function calls the other two functions and the program calls the main function at the end.

Next, the `And_fun()` function contains a while loop whose condition is based on the truth value of the condition: `a < b and a < c`. If either a < b is False, or a < c is False, or both are False, then the condition, `a < b and a < c,` is False and the loop is not entered.

When the `And_fun()` starts, it initializes *a* to 1, *b* to 5, and *c* to 4. Therefore, when `a < b and a < c` is first evaluated, it is True and the while loop can be entered. Within the while loop, the print statement outputs the values of *a*, *b*, and *c*.

In addition, inside the while loop is the following statement: `a = a + 1`. This statement increments the value of *a* by 1. As such, after the loop runs one time, the value of *a* becomes 2. What would happen if this statement were removed? The answer is that the loop would never end and the program would crash and note an error.

The while loop inside the `And_fun( )` function will repeat three times in this example, until *a* becomes 4. Once *a* takes on the value of 4, the condition, `a < b and a < c` is no longer true and the loop will end.

Similarly, the `Or_fun( )` initializes *a*, *b*, and *c*. However, the condition for the while loop contains a Boolean *or* (rather than *and)*. This alters the truth of the condition because in the case of Boolean *or*, either *a < b*, or, *a <c*, or both can be true for the entire condition to be true. The output shows that the while loop runs four times and prints the values of *a*, *b*, and *c* each time.

Finally, notice the use of comments throughout the program. As you write larger and larger programs, including comments will become necessary and invaluable.

4.4: The if/else/elif Decision Control Structure

Thus far, the for/in and the while loops have been described. The for/in loop repeats or iterates over a list of numbers or strings, or over a defined range. The number of loops or iterations of a for/in loop is predefined. The while loop repetition is based on the truth of a condition, or a special exit statement called a **break** (which will be discussed later in this section). The while loop can be infinite and therefore must be controlled. The number of repetitions in a while loop is not predetermined and may depend on other inputs that affect the truth of the while loop condition.

In many cases, decisions must be made within a program. Decisions can be based on logical conditions that are predefined or that depend on results generated from other portions of the program. For example, a function can return a value that is then used to make a decision, or a user might offer a certain input via the console that affects a logical truth in a decision structure.

The if/else/elif statements enable such decision making and flow-control within a program. The syntax for the if/else/elif decision structure is the following. The `elif` and the `else` are both optional.

```
if <logical condition>:
    <statements>

elif <logical condition>:
    <statements>

else:
    <statements>
```

The logical conditions may be simple, such as `x >10`. In this case, if the value in x is larger than 10, the condition will be true and the `if` statements will execute. Logical conditions may also be complex and dependent on other variables, such as `fname[0]=="A" or fname[0]=="a"`. In this case, if `fname` contains a string that is a person's first name (such as Alice), then `fname[0]` will be the first letter in that string (so A for Alice). This logical statement checks to see if this first letter is either A or a. If it is, the condition is true and the `if` statement(s) are executed. Strings will be covered in more detail on the chapter on data types and structures.

Example 4.4.1: An if/elif/else example

Consider the following program example. It is best to type in, save, run, and make updates to this program to learn what it does and what more it can do.

```
# If_Else_Example.py
# Ami Gates
# This program illustrations decisions and flow

def main():
    print("Welcome to the Grades Program")
    NumGrades=eval(input("How many grades will you enter?: "))

    print("Please enter your ",NumGrades," class grades.")
    gcount = 1
    gradeSum=0
    while gcount <= NumGrades:
        Grade=eval(input("Enter the next grade: "))
        gradeSum = gradeSum + Grade
        gcount = gcount + 1

    Average=round(gradeSum/NumGrades,2)

    print("\nThe average of your grades is ", Average)

    if Average > 89.4:
        print("Letter grade is an A")
    elif Average > 79.4 and Average < 89.5:
        print("Letter grade is a B")
    elif Average > 69.4 and Average < 79.5:
        print("Letter grade is a C")
    elif Average > 59.4 and Average < 69.5:
        print("Letter grade is a D")
    else:
        print("Letter grade is an F")

#-----------Program starts here ---call to main()-------
main()
```

A possible input and output for the above program:

```
Welcome to the Grades Program

How many grades will you enter?: 5
Please enter your  5  class grades.

Enter the next grade: 89.6

Enter the next grade: 75.2

Enter the next grade: 97.1

Enter the next grade: 88.4

Enter the next grade: 91.0

The average of your grades is  88.26
Letter grade is a B
```

For each example, be sure to type in the code, run it, alter it, and test it. See what the code does and see what it does not do. Notice also that so far, none of the examples have had any error checking to mitigate user based errors – such as, if a user enters "Green" instead of a numerical grade. Users can create errors be misusing a program. Program must contain methods for helping the user to use the program correctly. We will discuss error checking in future chapters. For now, think about what can be added to this program to assist the user and to avoid user-caused failure.

4.5: Nesting Loops and Decisions

When loops occur inside of other loops, this is referred to as **nested loops**. Nested loops can create complicated logic, and flowcharting is recommended to clarify coding goals and outcomes. Any type of loop or decision structure can be placed inside of each other. While there is no limit to the depth of the nesting (the number of nested loops or decisions), the greater the nesting, the greater the chance for error (both logical and syntactical).

The syntax for three nested while loops is:

```
while <logical condition>:
    <statements>
    while  <logical condition>:
        <statements>
        while  <logical condition>:
            <statements>
```

The syntax for nesting an `if` statement inside of a `for/in` statement inside of a `while loop` is:

```
while <logical condition>:
   <statements>
   for variable in list:
       <statements>
       if  <logical condition>:
           <statements>
```

These examples are only two of many possible combinations.

Example 4.5.1: The double while loop

```
# Double_While.py
# by Ami Gates
# This program illustrates nested while loops

def main():

    num_students=eval(input("Please enter the number of students
you will have grades for: "))
    num_grades=eval(input("Please enter the number of grades per
student: "))
    print("\n")
    gcounter=0
    scounter=1

    while scounter <= num_students:
        print("Enter the grades for Student #", scounter)
        grades_total=0
        while gcounter < num_grades:
            getgrade=eval(input("Enter grade: "))
            grades_total = grades_total + getgrade
            gcounter=gcounter+1

        print("The average grade for Student ", scounter, "is ",
grades_total/gcounter, "\n")
        num_students = num_students-1
        scounter=scounter+1
        gcounter=0

main()
```

One possible input/output for the above program:

```
Please enter the number of students you will have
grades for: 3

Please enter the number of grades per student: 2


Enter the grades for Student # 1
Enter grade: 90

Enter grade: 80
The average grade for Student  1 is  85.0

Enter the grades for Student # 2

Enter grade: 70

Enter grade: 60
The average grade for Student  2 is  65.0
```

In this example, two nested while loops are used because there will be multiple students to loop through and each of those students will have multiple grades to loop through. So, for each student (the outer loop), all grades must be collected and averaged (the inner loop).

Notice that once the inner loop has completed one cycle, the average for that student is printed. Then, some of the variables are **reinitialized,** to start again for the next student. For example, the variable called, `gcounter`, keeps track of the number (count) of grades that have been entered for the current student. As each grade is entered, the value of gcounter is incremented (1 is added to it). This allows for the use of gcounter for calculating the grade average. However, once a particular student grade-set is collected, the gcounter must be reset (reinitialized) to its initial value of 0.

As an exercise, type in this example program and run it for a few different inputs. Are the outputs what you expect? Next, create a flow chart that describes this program. Pay special attention to the creation and initialization of variables, as well as the need for the nested while loops. What would happen if the second while loop were not nested inside the first while loop? Try it and see.

Example 4.5.2: Nesting for/in and if/else

This example will illustrate the nesting of a for/in structure and an if/else structure. For this example, note that the formula for home mortgage monthly payments will also be used. The formula is included here for reference.

monthlypayment = mortcost*((mrate*((1+mrate)**months))/(((1 + mrate)**months) - 1))

From the formula, the "mrate" is the yearly rate divided by 12, the "**" means exponent, and the "months" is the number of months of payments. The "mortcost" is the amount that is being borrowed, and so is the price of the home minus the down payment. The following program illustrates for/in and if/else nesting.

```
# NestingLoopsDec.py
# Ami Gates
# This program will illustrate the nesting of for/in and if/else

#--------definition of main() function ----------
def main():
    print("This program will display the monthly mortgage payments for four
different rates, ")
    print("and for three duration options.\n")
    Mortgage()

#-------definition of Mortgage() function----------------
def Mortgage():
    houseprice=eval(input("Please enter the house price: "))
    downpayment=eval(input("Please enter the downpayment: "))
    maxpay=eval(input("Please enter the maximum amount you wish to pay each
month: "))
    mortcost = houseprice - downpayment

    for yearrate in [.03, .035, .045, .05]:
        for years in [15, 20, 30]:
            mrate = yearrate/12
            months = years*12

            monthlypayment = mortcost*((mrate*((1+mrate)**months))/(((1 +
mrate)**months) - 1))
            yearpercent=yearrate*100

            if round(monthlypayment,2) <= maxpay:
                print("The monthly payment for an interest rate of ",
round(yearpercent,2), "%")
                print("and for ", years, "years")
                print("will be :", round(monthlypayment,2), "\n")
            else:
                print("The rate of ", round(yearpercent,2), "% combined with
",years, "years, does not match")
                print("your budget constraints.\n\n")


#------Start of Program – call to main() --------
main()
```

While output can vary as it is based on input collected from the user, the following is a possible input/output example when running the program:

```
     This program will display the monthly mortgage payments for
four different rates,
     and for three duration options.


     Please enter the house price: 400000

     Please enter the downpayment: 40000

     Please enter the maximum amount you wish to pay each month:
2000
     The rate of  3.0 % combined with  15 years, does not match
     your budget constraints.


     The monthly payment for an interest rate of  3.0 %
     and for  20 years
     will be : 1996.55

     The monthly payment for an interest rate of  3.0 %
     and for  30 years
     will be : 1517.77

     The rate of  3.5 % combined with  15 years, does not match
     your budget constraints.


     The rate of  3.5 % combined with  20 years, does not match
     your budget constraints.


     The monthly payment for an interest rate of  3.5 %
     and for  30 years
     will be : 1616.56

     The rate of  4.5 % combined with  15 years, does not match
     your budget constraints.


     The rate of  4.5 % combined with  20 years, does not match
     your budget constraints.
```

```
The monthly payment for an interest rate of  4.5 %
and for  30 years
will be : 1824.07

The rate of  5.0 % combined with  15 years, does not match
your budget constraints.


The rate of  5.0 % combined with  20 years, does not match
your budget constraints.


The monthly payment for an interest rate of  5.0 %
and for  30 years
will be : 1932.56
```

This program contains several details. First, this program contains two functions. The first is the `main()` function and the second is the `Mortgage()` function. The *main()* function is very small and simply prints out a brief explanation. It then calls the *Mortgage( )* function. The *Mortgage( )* function first collects information from the user, such as the cost of the home, the down-payment (which must be subtracted from the cost as it will not be part of the mortgage), and the maximum monthly payment desired.

Next, the *Mortgage( )* function contains two, nested for/in loops, with an if/else nested inside of the second for/in loop. In this case, the double for/in loops are needed because the program will calculate the monthly mortgage payment for all yearly rates, `[.03, .035, .045, .05]`, and for 15, 20, and 30 year durations. The nested for/in loop allows the program to start with the .03 (3%) yearly rate, and calculate all monthly mortgage costs for 15, 20, and 30 years (and to print the results). Once the inner for/in loop has completed its calculations and output for .03 (3%), it returns to the outer loop, progresses to the next element in the `yearrate` list, and repeats the inner loop again. In other words, for all the rates in the outer loop, including .03, .035, .045, and .05, the inner loop will run. This can be observed in the output of the program. As an exercise, try to rewrite this program without nested loops. What happens?

Finally, the if/else statements are nested inside of the inner for/in loop. The if/else statements allow the inner loop to determine if the monthly payment for each combination of duration and rate meets or exceeds the users maximum requirements. If the monthly payment is less than or equal to the users maximum payment, the information is printed as output by the inner loop. If the payment exceeds the user’s maximum payment (in the *else* statement), the user is notified of this via output as well.

There is no limit to the number of nested loops or decision structures a program can contain. However, nesting adds complexity, and exceeding three nested loops may result in less easily predictable outcomes. As an exercise, type in and run this program. Make changes and see what effects those changes have.

4.6: Break, Continue, and Pass

The break statement offers an option for breaking out of a loop and proceeding from the next statement that occurs after the loop body. The break statement generally appears after a condition, if true, will execute the break.

The syntax for break is:

```
break
```

The following is a small collection of code using the break statement.

```
def main():

   userinput="True"

   while userinput[0] == "T" or userinput[0]=="t":
       print("Do these statements...")
       userinput=input("Enter True or False: ")

       if userinput[0] == "F" or userinput[0]=="f":
           break

main()
```

The continue statement stops the current loop at the point of the continue statement, and returns to the loop starting statement. Once a continue statement is executed, no further statements, in the loop, following the continue statement, are executed. However, unlike the break statement, the continue statement does not break out of the loop. Instead, the continue statement continues the loop at its next iteration. The continue statement is often coupled with a decision control statement, such as an if statement.

The syntax for continue is:

```
continue
```

The following is a small collection of code using the continue statement.

```
x=0
while x <10:
    x=x+1
    if x==5 or x==6:
        continue
    print(x)
```

The output for this will print 1, 2, 3, 4, 7, 8, 9, 10. The continue statement causes it not to print the 5 or the 6.

The pass statement is used as a place holder for an otherwise empty location. The pass statement is used when a statement is required due to syntax rules, but logically no statement is needed or wanted.

The syntax for pass is:

```
pass
```

The following example illustrates the use of the pass statement.

```
grade=eval(input("Enter grade: "))
if grade < 59.4:
    pass
else:
    print(grade)
```

While these flow control statements can be very useful, they also add complexity. This is especially true if they are used within nested loop or decision structures. Creating visual flowcharts can be very effective during the predevelopment and planning stages and program development.

Summary

Chapter 4 covered concepts of loops, including the for/in loop and the while loop. It also discussed the if/elif/else decision structure. Examples illustrated that loops and decisions can work together, either nested or sequential, to affect the flow or control of a program. Python also offers other options for repetition, flow, and control, including the break statement, the continue statement, and the pass statement.