(Python) Chapter 3: Repetition

3.1 while loop

Motivation

Using our current set of tools, repeating a simple statement many times is tedious. The only item we can currently repeat easily is printing the exact same message multiple times. For example,

```
print("I love programming in Python!\n"*10)
```

will produce the output:

```
I love programming in Python!
I love programming in Python!
I love programming in Python!
I love programming in Python!
I love programming in Python!
I love programming in Python!
I love programming in Python!
I love programming in Python!
I love programming in Python!
I love programming in Python!
```

Imagine that we wanted to number this list so that we printed:

```
1. I love programming in Python!
2. I love programming in Python!
3. I love programming in Python!
4. I love programming in Python!
5. I love programming in Python!
6. I love programming in Python!
7. I love programming in Python!
8. I love programming in Python!
9. I love programming in Python!
10. I love programming in Python!
```

Now, the times operator (*) is no longer capable of allowing us to produce this output. Luckily, Python provides us with multiple general tools for repetition where we'll simply specify which statements we want repeated and a way to determine how many times to repeat those statements.

Definition

The while loop is the most simple of the loops in Python. The syntax for the loop is as follows:

```
while <Boolean expression>:
    stmt1
    stmt2
    ...
    stmtn
stmtA
```

The manner in which this gets executed is as follows:

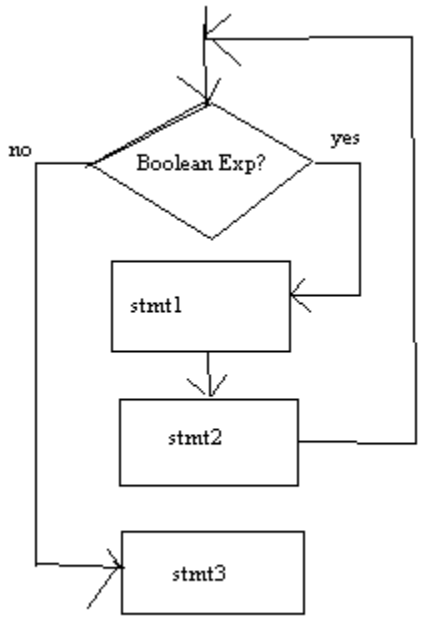
1) Evaluate the Boolean expression.
2) If it's true
   a) Go ahead and execute `stmt1` through `stmtn`, in order.
   b) Go back to step 1.
3) If the Boolean expression is false, skip over the loop and continue to `stmtA`.

The key to the repetition is that if the Boolean expression is true, after we complete the statement, we check the Boolean expression again instead of continuing. (In an if statement we would have continued on.)

Flow Chart Representation

Here is a flow chart representing of the following code segment:

```
while <Boolean expression>:
    stmt1
    stmt2
stmt3
```



I love C Programming 10 Times Over

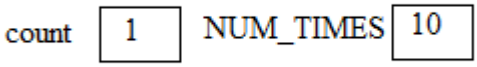
Now, with the while loop, we can more succinctly write code that prints out "I love programming in Python!" ten times in a numbered list.

Our first realization is that we must utilize the loop construct so that it can "count." Namely, on any given running of the loop, we must have some way of "remembering" how many times we've already printed out message. In order to do this we'll use an integer that will act as a counting variable. At any point in time, it'll simply represent how many times we've printed our message.

Let's take a look at a program that accomplishes our task:

```
def main():

    count = 1
    NUM_TIMES = 10

    while count <= NUM_TIMES:
        print(count,". I love programming in Python!", sep ="")
        count = count + 1

main()
```

At the very beginning of the program, our variables look like this:



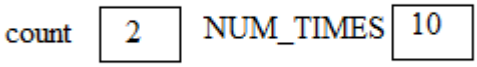
Thus, the while condition is true and the loop is entered. When we print, the value of count is 1, so we get the first line of output:

```
1. I love programming in Python!
```

Next, we execute the line

```
count = count + 1
```

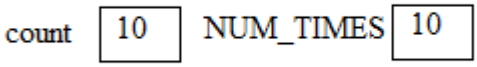
The left-hand side equals 2, thus, count will get set to two:



Since this a while loop and no more statements are indented, we go back to the top to check to see if the Boolean condition is true. Since count is less than or equal to NUM_TIMES, we enter the loop a second time, this time printing:

```
2. I love programming in Python!
```

Subsequently, we increment count to 3. Now, let's pick up execution when we get to the top of the loop and count is equal to 10:



Now, we enter the loop since the Boolean condition is still true. Then we print:

```
10. I love programming in Python!
```

Next, we increment count to 11. When we go back to the top of the loop, the Boolean condition is no longer true! Thus, we exit the loop after having printed out the desired output.

Alternatively, the following variation would work as well:

```
def main():

    count = 0
    NUM_TIMES = 10

    while count < NUM_TIMES:
        count = count + 1
        print(count,". I love programming in Python!", sep ="")

main()
```

It's important to realize that problems can be solved in different ways, even one as simple as this one. In our original solution, we used a 1-based counting system, starting our counting variable at 1. In the second example, we find if we change the order of statements in the loop, we must adjust the initial value of our count AND change our Boolean condition.

All of this is evidence that when dealing with loops, attention to detail and consistency are very important. It's fine to start count at either 0 or 1, but the rest of the code must be consistent with this starting value. (Note: in certain situations we'll find the 0-based counting system to be more natural and in others we'll prefer the 1-based system.)

In general, after examining this program in detail it should be fairly clear that we can execute any set of statements a set number of times using the same general construct:

```
count = 1

while count <= NUM_TIMES:
    # Insert code to be repeated here.
    count = count + 1
```

where NUM_TIMES is set to however many times the code block is to be repeated.

Using Loops to Print Out Different Patterns of Numbers

In the previous program, we used a loop variable to count from 1 to 10. Now, consider trying to count all the positive even numbers up to 100. One way to solve this problem would be to use a variable to count from 1 through 50, but then to multiply the variable by 2 when printing out the numbers, so instead of printing 1, 2, 3, ..., 50, the values 2, 4, 6, ..., 100 would get printed:

```
count = 1
while count <= 50:
    print(2*count)
    count = count + 1
```

Another technique would be to have count equal to the values that are being printed and then just print count. This means count must start at 2, and that we must add 2 to count each time around:

```
count = 2
while count <= 100:
    print(count)
    count = count + 2
```

Now, consider writing a code segment to print all the positive odd integer less than 100. We can employ techniques virtually identical to the ones above. In both cases, we must print a number 1 lower than what we were printing. Here are both solutions:

```
count = 1
while count <= 50:
    print(2*count-1)
    count = count + 1
```

```
count = 1
while count <= 100:
    print(count)
    count = count + 2
```

Finally consider writing a code segment to print the following sequence of numbers, one per line:

2, 4, 8, 16, …, 1024

In all of the previous examples, we would add a constant to the previous term to get the next term. In this exam, we need to multiply the previous term by 2 to get the next. This leads to the following solution:

```
count = 2
while count <= 1024:
    print(count)
    count = 2*count
```

Example: Add Up the Numbers from 1 to 100

Now let's look at a slightly more difficult example that utilizes the counting variable inside of the loop.

Consider adding the numbers 1, 2, 3, 4, …, 100. If you were to do this task by hand, you'd start a counter at 0, add in 1 to it to obtain 1, then add in 2 to it to obtain 3, then add in 3 to it to obtain 6, then add in 4 to it to obtain 10, etc.

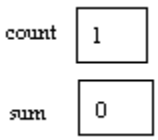
Let's use the while loop to automate this task.

We will need a counting variable similar to count in the previous example. But, we will also need a variable that keeps track of our running sum. Let's call this variable sum. Any variable that keeps a running tally going is known as an accumulator variable. The idea behind accumulator variables is critical. It is used in many practical programs.

Let's take a look at a program that solves this problem and trace through the first few steps carefully:

```
def main():

    MAX_TERM = 100

    sum = 0
    count = 1

    while count <= MAX_TERM:
        sum = sum + count;
        count = count + 1;

    print("The total is ",sum,".", sep="")

main()
```

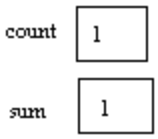
At the very beginning of the program, before the loop starts, memory looks like the following:



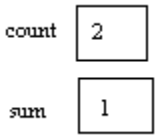
Initially, the Boolean expression for the while loop is true, since count, which is 1, is less than or equal to MAX_TERM. Now we encounter the critical line of code:

```
sum = sum + count;
```

Currently, sum is 0 and count is 1, so adding we get 1.This value is then stored back into the variable sum:



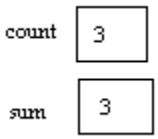
Next, we increase count by 1, so now our picture looks like:



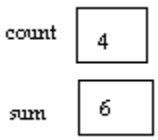
We then check the Boolean expression again and see that it is true. At this point in time, we've added in 1 to our total and are ready to add in 2. Now, we hit the line

```
sum = sum + count;
```

This time sum is 1 and count is 2, which add up to 3. This value gets stored in sum. Then, we follow this line by adding one to count, which changes to 3:



Now, we've added up 1 and 2 and are waiting to add in the next number, 3. The Boolean expression is still true, so we then add sum and count to obtain 6 and then change sum to 6. This is followed by adding one to count, making it 4:

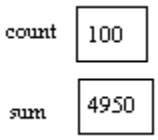


At this point, hopefully the pattern can be ascertained. At the end of each loop interation, sum represents the sum of all the numbers from 1 to count-1, and count is the next value to add into the sum. The key idea behind an accumulator variable is that you must initialize it to 0, and then each time you want to add something into it, you use a line with the following format:

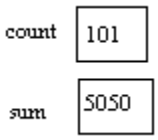
```
<accum var> = <accum var> + <expr to add in>;
```

Whichever variable is the accumulator variable is the one that is set, using an assignment statement. It's set to the old value of the variable plus whatever value needs to be added in.

In our example, if right before the very last loop iteration, the state of the variables is as follows:



Now, we go and check the Boolean expression and see that count is still less than or equal to MAX_TIMES (both are 100), so we enter the loop one last time. We add sum and count to get 5050 and then update count to be 101:



Now, when we check the Boolean expression, it's false. We exit the loop and print sum.
To better understand this example, add the line:

```
print("count =",count,"sum =", sum)
```

as the last line of the body of the loop and adjust MAX_TERM as necessary:

```
while count <= MAX_TERM:
    sum = sum + count;
    count = count + 1;
    print("count =",count,"sum =", sum)
```

When programs aren't working properly, inserting simple print statements like this one can help uncover the problem. This process of finding mistakes is called debugging. Though inserting print statements isn't the ideal way to debug, it's the most simple way and recommended for beginners. More will be mentioned about debugging later. In this particular example, the print is to aid understanding, so that we can see what each variable is equal to at the end of each loop iteration.

Now, consider editing this program so it calculated the sum of the numbers 1, 3, 5, …, 99. We just change MAX_TERM to 99 and change our loop as follows:

```
while count <= MAX_TERM:
    sum = sum + count
    count = count + 2
```

The key difference here is that count no longer represents how many times the loop has run. Rather, count simply represents the next number to add.

Yet another approach edits the loop as follows:

```
while 2*count-1 <= MAX_TERM:
    sum = sum + (2*count – 1)
    count = count + 1
```

In this approach, count keeps track of how many times the loop has run, but when we need to access the current term, we use an expression in terms of count (2*count – 1) that equals it.

As previously mentioned, programming problems can be solved in many different ways. It's important to get comfortable manipulating counting variables in loops as shown in these examples so that a programmer can solve different types of problems.

Sentinel Controlled Loop

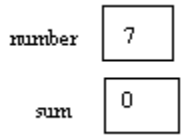
Each of the loops we've seen so far is a counting loop. Namely, a variable is used to keep track of how many times the loop has run, and the loop is terminated once it has run a particular number of times.

Now consider a slightly different problem: adding up a list on numbers until 0 is entered. In this problem, 0 is considered the sentinel value, since it represents the trigger for ending the loop. (The loop exits not because it's run a particular number of times, but because some value, known as the sentinel value, has been triggered.)

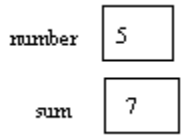
Just like the previous problem, we'll require the use of an accumulator variable, sum. We will also need to use a second variable to read in each number. Let's call this number. Now, we must read in each value into the variable number until this variable equals 0 (which means our loop runs so long as number doesn't equal 0.) Let's take a look at the program:

```
def main():

    sum = 0
    number = int(input("Enter your list, ending with 0.\n"))

    while number != 0:
        sum = sum + number
        number = int(input(""))

    print("The sum of your numbers is ",sum,".", sep="")

main()
```

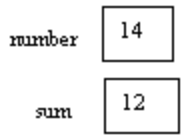
Let's trace through the example where the user enters 7, 5, 14, and 0. After the very first input, the state of the variables is as follows:



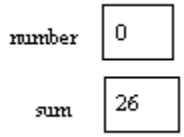
Since number isn't 0, the loop is entered. Adding the current values of sum and number yields 7, and we set sum to 7. Then, we read in the next number, 5:



Once again, number isn't 0, so we enter the loop again. Adding sum and number yields 12, and we set sum to 12. Then we read in 14 to number:



Since number isn't 0, the loop is entered again and sum and number are added to obtain 26, which is stored in sum and the next value read into number is 0:



Now, when we check the Boolean condition, it is evaluated as false and the loop is exited. Then, the value of sum, 26, is printed.

Guessing Game Example

A common game many young kids play is guessing a number from 1 to 100. After each guess, the person who chose the secret number tells the guesser whether their guess was too low or too high. The game continues until the guesser gets the number. The natural goal of the game is to minimize the number of guesses to get the number.

In this example, the "computer" will play the role of the person generating the secret number and user will be the guesser. We'll simply output the number of guesses it took to get the number.

Just like the previous example, since we don't know how many times the loop will run, we will check for a trigger condition to exit the loop. In particular, so long as the user's guess doesn't equal the secret number, our loop must continue.

A variable will be needed to store both the secret number and the user's guess. The rand function discussed in the previous chapter will be used to help generate the random number.

The basic strategy is as follows:

1) Generate a secret number
2) Set the number of guesses the user has made
3) Loop until the user's guess equals the secret number
    a) Prompt the user for a guess
    b) Read in a guess
    c) Output an appropriate message
    d) Update the guess counter
4) Output the number of moves the user took to guess the secret number.

Here's the program:

```
import random

def main():

    MAX_VAL = 100

    num_guesses = 0
    guess = -1

    secret = 1 + random,randint(1, MAX_VAL)

    while guess != secret:

        guess = int(input("Enter your guess.(1 to "+ str(MAX_VAL)+")\n"))

        if guess == secret:
            print("You win!")
        elif guess < secret:
            print("Your guess is too low. Try again!")
        else:
            print("Your guess is too high. Try again!")

        num_guesses = num_guesses + 1;

    print("You won with",num_guesses,"guesses.")

main()
```

This is the first example with an if statement inside of a loop. Since an if statement is a regular statement, it's perfectly fine to place in a loop. In this particular example, we are guaranteed to initially enter the loop because guess is set to -1, which is guaranteed not to equal the secret number, which is guaranteed to be in between 1 and 100.

When we read in the first guess, we need to decide which message to output the user. There are three possibilities, so we use a nested if to output the appropriate message. Finally, we use num_guesses as a counting variable through the loop. When the user enters the correct guess, "You win!" will be printed, the total number of guesses will be updated and when the Boolean expression is checked again, the loop will exit and the correct number of total guesses will be printed.

Example: Guessing Game Rewritten using a flag-controlled loop

Yet another technique often used to control a loop is a flag controlled loop. Simply put, a flag is a variable that keeps track of whether or not to enter a loop. A common method for using a flag is to set it to 0 while the task is not done, to indicate that the loop should continue and then set it to 1 when the task is done. This general technique can be used in many settings.

Here is the guessing game rewritten to use a flag to control the loop. A couple other changes have been made to streamline this version:

```
import random

def main():

    MAX_VAL = 100

    num_guesses = 0
    guess = -1
    done = False

    secret = 1 + random.randint(1, MAX_VAL)

    while not done:

        guess = int(input("Enter your guess.(1 to "+ str(MAX_VAL)+")\n"))

        if guess == secret:
            done = True
        elif guess < secret:
            print("Your guess is too low. Try again!")
        else:
            print("Your guess is too high. Try again!")

        num_guesses = num_guesses + 1;

    print("You won with",num_guesses,"guesses.")

main()
```

The key here is when we realize that the correct guess was made, we must set our flag done to True, to indicate that the loop is now done. This technique of using a Boolean variable to control a loop running is very common.

Idea of a Simulation

Now that we can repeat statements many times and generate random numbers, we have the power to run basic simulations. One practical use of computers is that they can be used to simulate items from real life. A nice property of a simulation is that it can be done really quickly and often times is very cheap to do. Actually rolling dice a million times would take a great deal of time, and actually running a military training exercise may be costly in terms of equipment, for example. As you learn more programming tools, you'll be able to simulate more complicated behavior to find out answers to more questions. In this section we'll run a simple simulation from the game Monopoly.

There are 40 squares on a Monopoly board and whenever a player rolls doubles, they get to go again. If they roll doubles three times in a row, they go to jail. There are a few more ways that affect movement, but these are more complicated to simulate. Let's say we wanted to know how many turns it would take on average to get around the board. We have the tools to simulate this. One simplification we'll make is that we'll stop a turn at three doubles in a row and we won't put the player in jail. We are making this simplification so that our code will be manageable. We will still get a reasonable result without having to add complicated code. Whenever writing simulations, we must make decisions about how accurately we want to carry out our simulations. If an extremely accurate answer is necessary, we must spend more time to make sure that the details of our simulation match reality. But, if we are running our simulation just to get a ballpark figure, as is the case in this simulation, certain simplifications are warranted. (It's relatively rare for us to roll three doubles in a row, so if our model is inaccurate in this one instance, it won't affect our overall results much. Technically, we'll roll three doubles in a row once every 216 rolls or so, which is less than 1% of the time.)

For our simulation, we want to run several trials, so we'll create an outside loop that loops through each trial. In a single trial we will accumulate rolls of the dice until the sum gets to 40. As we are running the trial, we will count each turn. After the first turn that exceeds a sum of 40 for all the dice rolls in that trial, we stop the trial and count the number of turns it took us to "get around the board." We add this to a variable and will use this sum to calculate our end average. Here is the program in full:

```
# Arup Guha
# 7/6/2012
# Simulates how many turns it takes to go around a Monopoly board
# (without special squares...)
import random

def main():

    # Initialize necessary variables.
    i = 0
    NUMTRIALS = 100000
    sumturns = 0

    # Run each trial
    while i < NUMTRIALS:

        # Set up one trip around the board.
        spot = 0
        turns = 0

        # Stop when we get around the board.
        while spot < 40:

            # Start this turn.
            turns = turns + 1

            # Make sure we don't go more than three times.
            doublecnt = 0
            while doublecnt < 3:

                # Do this roll and update.
                roll1 = random.randint(1,6)
                roll2 = random.randint(1,6)
                roll = roll1 + roll2
                spot = spot + roll

                # Get out if we didn't get doubles.
                if roll1 != roll2:
                    break

                doublecnt = doublecnt + 1

        # Update number of total turns.
        sumturns = sumturns + turns
        i = i+1

    # Print out our final average.
    average = sumturns/NUMTRIALS
    print("Average is",average)

main()
```

Note: When we run this code, it takes several seconds to complete, since so much work is being done in the simulation. Also, it's important to note that python, due to its interpreted nature, runs slower than other languages. A comparable simulation in C takes less than a second.

Though this program is more complex that ones we've previously seen, if you take it apart line by line, there is nothing complicated in its logic. The key is to break down each subtask into logical steps and then convert those steps into code. It this program, we have three levels of loops. The outer most level takes care of multiple trials while the middle loop takes care of a single trial and the inner most loop takes care of a single turn, which can have multiple dice rolls, due to doubles. In regular Monopoly, when someone rolls doubles three times in a row, they go to square number 10, or jail. From there, they lose a turn and can get start rolling again, if they pay $50. If we ignore the monetary consequence, we can simulate this relatively easily, by checking to see if doublecnt is 3, resetting spot to 10 in that case, and adding an extra one to turns.

A great way to improve your programming skills is to think about some event from real life and see if you can write a simulation for it. In this program, we found out that on average, it takes about 5.3 to 5.4 turns to circle the Monopoly board!