Additional Repetition Notes

**Repetitive Logic**

Let's think for a moment where we find repetition in our daily lives.

* If we try to call someone and it is busy, we repeat the process until we get frustrated or we get through (notice the logic operator OR in there!).
* When we balance our check book, we repeatedly add or subtract transactions until we have processed them all.
* When we want to access our account at an ATM, we enter a password and re-enter it when we get it wrong.
* When we tell a 2 year old to count from 1 to 10, she says "1, 2, 3, 4, ... 10", a repetitive process.

Let's think about the 2 year old and the process her brain goes through.

* Step 1 -- Her mind starts at number 1 (because that's where you told her to start)
* Step 2 -- Is the number <= 10? If yes continue to Step 3, otherwise STOP (jump to Step 6)
* Step 3 -- say the number
* Step 4 -- go to the next number (we as adults know this is + 1)
* Step 5 -- go back to Step 2
* Step 6 -- DONE!! (can I have a cookie now?)

This is the algorithm to count to 10. The 2 year old knows it; we need to tell the computer how to do it. Let's rewrite the algorithm in **pseudocode** (not actual Python statements -- just 'English-like' statements that represent the logic of what we want to accomplish):

number = 1 # **INITIALIZE** -- where to start  
while number <= 10 # **LOOP** **TEST** -- continue or not

print number  
number = number + 1 # **UPDATE**  
go back to the top of the loop and repeat

execute this line when the loop finishes (and then continue with the rest of the algorithm)

When the repetitive process (or loop) is done, execution jumps to the line after the loop.

All loops have the 3 pieces, or **elements** you see commented: **INITIALIZE**, **LOOP TEST** and **UPDATE**. The loop test is the easiest one to think about first. What needs to be true in order to repeat? (Think back to the examples above).

The variable that has its value tested controls the repetitions of the loop. This variable is known as the **loop control variable**. The loop control variable must be initialized -- be given a starting value -- before it is tested. That's why the initialization must come before the test. In some situations, there may be more than one loop control variable.

Finally, there must be an update -- something that changes the value of the loop control variable. We want the loop test to be false at some point, so the variable's value must change; at some point that value will cause the loop test to fail. The update needs to be inside the body of the loop!

If the loop test never fails (meaning the update is not working properly) this is known as an infinite loop. It is an error in your algorithm. Think again about the 2 year old counting to 10. What if the child doesn't go to the next number? She'd just repeat "1...1...1..." forever!

The Five Elements of a Loop

There are five key elements that should be considered during the design of any loop, and we have just finished discussing three of them. Here is a complete definition of all five:

* **Initialization** - Set up the proper conditions for the start of the loop.
  + Identify the **loop control variable**. This is the variable that will have its value changed with each iteration (repetition) of the loop. It must be assigned an initial value before the loop begins. Make any other initializations needed for the loop to operate properly.
* **Loop Test** - Specify the conditions that must be satisfied for the loop body to be executed.
* **Main Work** - Statements to be executed during each iteration of the loop. (Saying the number in the example of the child counting.)
* **UPDATE** or "Change" or "Making Progress Toward Termination" step.
  + The value of the **loop control variable** should be **changed** inside the body of the loop.
  + The pattern of changes must be such that the loop eventually terminates! In other words, the loop must continually make progress toward termination. In the example above, the number is repeatedly increased by one, making it get closer to 10 with each iteration of the loop.
* **Finalization** - Add any statements needed after the end of the loop to properly wrap things up (e.g., asking for a cookie)-- an optional step.

The **order** in which these elements appear in a loop **may vary**. Initialization will always be first, and finalization (if used) will always be last (as their names imply), but the other elements may appear in any order -- depending on the iteration structure used and the design of the algorithm being implemented.

Accumulator variables

Think about the checkbook example from above. As you process each individual entry, you keep track of a running total -- how much you have in the account so far. Once you've processed all the data you then have the final account balance. Same with programs.

|  |  |
| --- | --- |
| Checkbook example:  If I have the following entries:  deposit $40 check for $19.95 check for $6.05 deposit $80 | Here is the Accumulated Value:  $ 40.00 $ 20.05 $ 14.00 $ 94.00 |

If I want a program to add the numbers 1 to 10, the program has to **accumulate** the sum and keep track of how much it has added up so far. Here is an example:

sum = 0 # accumulator variable being initialized  
count = 1 # loop control variable being initialized  
while count <= 10

sum = sum + count # add the new value to the running total (**accumulate**)  
count = count + 1 # update for the loop

Notice there are two variables now, each with a different job. The variable count is the one controlling the loop (notice that it is the variable checked in the loop test). The variable sum is known as the **accumulator**.

Design Patterns

Think of how you might change this to add the numbers from 200 to 500? You would change the initial value of the count variable, and the test for the upper limit. What if you want to calculate the sum of all numbers from 3 to 330, counting by 3s? You'd change the initial and test values for the count variable, and you'd add 3 to the count variable in the update. Notice that you only have to change some constants. What if I said multiply instead of sum up? You'd only change an arithmetic operator (and the accumulator's initialization). The **design** of the loop is the same. This is important in programming. You may be asked to write many different loops; but the patterns will all be similar. The loops above are known as **definite loops**. You can tell from the algorithm how often they will repeat. We will also learn about **indefinite loops** -- those that you can't determine how often they repeat just by viewing the code.