CSC110 Supplemental Reading: Week04

Contents

1 Repetition Algorithm

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 tell a 2 year old to count from 1 to 10, he says " 1, 2, 3, 4, ... 10", a repetitive process. When we want to access our account at an ATM, we enter a password and **re-enter** it when we get it wrong.

Let's think about the 2 year old and the process he goes through:

Step 1 -- His mind starts at number 1 # because that's where you told him to start
Step 2 -- Is the number <= 10? If yes continue, 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 from 1 to 10. The 2 year old knows it; we need to tell the computer how to do it. Lets rewrite the algorithm in pseudocode :

```
number = 1 # initialize -- where to start
while number <= 10 # test -- repeat 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 you see commented: **initialize**, **test** and **update**. Test is the easiest one to think about first. What it is you want to be ***true in order to repeat***? (Think back to the examples above). The test line identifies the variable (or variables) that control the repetitions of the loop. This variable is known as the **control variable**.

This control variable must be **initialized**, or have a starting value before it is tested. That is why the initialization must come before the test.

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

If a loop never fails (meaning the update is not working properly or there's a problem with the test or initialization) 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? He'd just repeat "1...1...1..." for ever!

1.1 Accumulator variables

Think about a checkbook example. 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.

If I have the following checkbook entries:	**Here is the accumulated value:**
deposit $40	$ 40
check for $19.95	$ 20.05 (subtracted the 19.95)
check for $6.05	$ 14.00
deposit $100	$114.00 (added the deposit)

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 as the control variable iterates through the values 1 to 10. Here is some pseudocode:

```
sum = 0 # accumulator variable being initialized
count = 1 # loop control variable being initialized
while count <= 10
sum = sum + count # running total syntax to add in the new value (accumulate)
count = count + 1 # update for the loop
repeat
```

Notice there are 2 variables now, each with a different job. The count variable is the one controlling the loop (notice that it is the variable used in the test). The sum variable is known as the **accumulator**.

1.2 Design Patterns

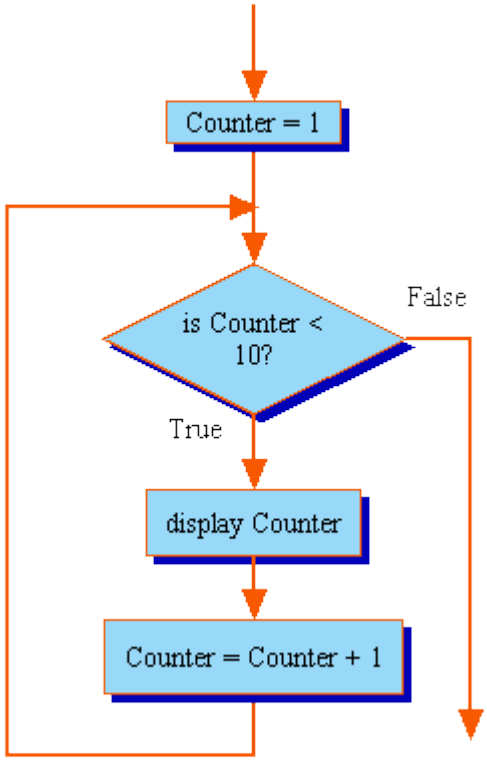
Think of how you might change this to add the numbers from 200 to 500? You would change the initial value of the control variable, and the test value. What if you want to sum from 3 to 330 counting by 3s? You'd change the initial and test values for the control variable, and you'd add 3 to the control 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 set the initial value in the accumulator to 1 (why?). 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.

2 While loop

2.1 A Simple while Loop

What is the syntax to write code that repeats? (sometimes we say "iterates.") The table below has the Python code to display the numbers from 1 to 9 as well as a flowchart. It uses one variable, counter, in order to control the loop. This introduce a new Python keyword: **while**.

```
counter = 1
while counter < 10:
  print (counter)
  counter += 1
```



Sequence of Execution
counter initialized to 1

Test: counter < 10 (true)
display counter
add 1 to counter: counter is now 2

Test: counter < 10 (true)
display counter
add 1 to counter: counter is now 3

.
.

Test: counter < 10 (true)
display counter
add 1 to counter: counter is now 9

Test: Counter < 10 (true)
display Counter
add 1 to counter: counter is now 10

Test: counter < 10 (**false**)
go to next statement in code

So what's going on here?

- At first the counter variable is initialized.
- Then the test of the loop executes. If the test is true, the body of the loop executes (the indented code), which includes updating the counter variable.
- When the bottom of the loop is reached, execution goes back up to the test.
- This process keeps repeating (test/body/test/body...) until the test is false.
- Once the test is false, the flow of control skips over the body of the loop and continues on with the rest of the program.

What is the value of counter after the loop finishes? It would be 10, not 9. The variable has to contain the value that made the test false.

Notice that the flowchart of the loop looks very similar to the flowchart of an if statement. The if statement causes a decision, but both paths flow forward. In a loop there is also a decision and two

paths, but one of the paths flows *backward*. Because the loop structure also makes a decision, it uses boolean expressions. You don't have to learn anything new: everything we've learned previously about logical expressions is still true for loops.

Notice also that the syntax of the **while loop** is very similar to the syntax of an if statement. After the keyword while, you have the test expression. The body of the while loop (like the action of the if statement) is indented. You can put any legal Python statement within the body of the loop (just like with an if statement).

Hopefully you can see that the meaning of the keyword while is the same as in the English language. The word while means "as long as this is true". Once the expression is false, the loop is done, and the program executes the code following the loop.

There are several possible things you could change in this program code:

- switch the position of the `print` and `counter += 1` statements... what would this do?
- change the test to `counter <= 10` what would this do?

2.2 Tracing the Operation of a Loop ("Walk-Throughs")

You will recall that a variable is simply a named memory location -- a place to store information. It is a fundamental principle that only one piece of information may be stored in a variable at a time. If a new piece of information is stored in the variable, its previous value is lost. The process of tracing the operation of a loop is simply the process of illustrating how the contents of memory (the values of the variables) change as the loop is executed. If we take the trouble of numbering the lines in a program, we can also show the sequence in which lines are executed. Here is an example that displays the squares of the numbers from 1 to 3:

Line	Program
1	`num = 1`
2	`while num <= 3:`
3	`    square = num ** 2`
4	`    print (num, "squared = ", square)`
5	`    num += 1`
6	`print ('Have a nice day')`

Line #	num	square	num <= 3	display
1	1			
2			True	
3		1		
4				1 squared = 1
5	2			
jump to 2			true	
3		4		
4				2 squared = 4
5	3			
jump to 2			True	
3		9		
4				3 square = 9
5	4			
jump to 2			False	
jump to 6				Have a nice day

Here are some things to notice in this example:

- The program listing and line numbers are shown for reference. The 5 right-hand columns are a trace of the program execution.
    - The trace has one row for each line of code that is executed.
    - The first column in each row is the line number being executed.
    - The next 2 columns show the value of the variables in the program after the line is executed.
    - The 4th column shows the value of the test expression and
    - the last column shows what is displayed.
- Look at the sequence of numbers in the first column and you will see that the lines in the program are not executed in sequence. The order of execution is altered by the control structure -- in this case, by the loop. Do not expect the line numbers to be listed in sequence -- an important part of performing the trace is determining and illustrating the order in which lines are executed. For reference, a line number is underlined each time the order of execution departs from sequential.
- The table has a column for each variable, test expression, and the output of the program. An entry is made in the column only when the value has changed. For example, when line 5 is executed for the first time, the result is to change the value of num to 2. The value of a variable at any point in time may be determined by examining the most recent entry in its column. So, at the time that line 6 executes, the value of num is 4. This reflects the fact that a variable may have only one value at a time. Each new value replaces the previous value.

2.3 Accumulator Variables

Accumulator variables are used to accumulate values, whether it is to sum up values, or to keep track of a product, or some other accumulation. Here is an example that sums up the even numbers from 2 to 20. If you add up those numbers by hand, you'd get 110. See if you get the same answer if you trace through this code:

```
counter = 2 # this is the control variable for the loop
sum = 0 # this is the accumulator variable. It will be holding the sum
while counter <= 20:
    sum += counter
    counter += 2; # notice here in the update that we're adding 2!
print ("The sum from 2 to 20 = ", sum)
```

How would you change this loop to add up the multiples of 5 from 5 to 100?

Here is a different example that calculates a product. For a change, we are counting down from 5 to 1. We could just as easily count up from 1 to 5. There is no difference, as long as the counter variable contains each of the values between 1 and 5 at some point.

```
counter = 5
product = 1 # this is the accumulator variable. Why do you think it is
initialized to 1 instead of 0?
while counter > 1:
    product *= counter
    counter -= 1
print ("The answer is ", product)
```

This code would display the value 120. Did you get that when you traced it?

3 for loop

The *for* statement repeats a block of code based on working through a sequence of values, in order. Here is the syntax for the *for* statement.

for *var* in *sequence* :

    block

It's easiest to look at the semantics of this statement if we look at a concrete example:

```
for j in [ 1 , 2 , 3 ] :
    print (j)
```

Let's look at what we have here. Between the keywords *for* and *in*, we have **j**. This is the variable. Between the keyword *in* and the colon, we have a *list*. The list is a sequence of values enclosed within square brackets and separated by commas. (We'll look at lists in greater detail later in the quarter. For the moment, it's enough to know that this is a list of three values: 1 and 2 and 3, in that order.)

Just for the record, this is typically not quite so spacey.

```
for j in [1, 2, 3]:
    print (j)
```

So, what happens when this runs? It prints out:

1
2
3

Let's take a look at how this happens.

When this *for* loop is run, the variable j initially gets the first value in the sequence, 1. Then the block is performed, so the value 1 is printed out. At the end of the block, flow-of-control loops back to the beginning of the *for* loop. The *for* loop keeps track of how much of the sequence has been processed. So, when it loops back up to the top, for the second iteration, the variable j gets the second value in the sequence, 2. Then the block prints out this value, 2. Similarly, at the end of the block, the flow-of-control loops back up to the top of the for loop. This time j gets 3, which the block then prints out. Notice … after printing out 3, the flow-of-control goes back to the top of the loop again. However, there is nothing left in the sequence, no new value for j. So, the *for* loop is done. The next statement, the one following the *for* loop is performed.

The list can contain anything, not just int values as we saw in the preceding example. In fact, the list can be stored in a variable. So, what will this loop do?

```
stuff = ['pi', 'is', 3.1416, 'more or less']
for k in stuff:
    print (k)
```

Remember the range() function we look at in the Python library? It returns a list. We can use that to specify a sequence for a for loop.

Can you figure out what this will do?

```
for j in range(5, 30, 5):
      print (j)
```

How about, if instead of printing these values, we wanted to sum them up. Then we would need an accumulator variable:

```
sum = 0
for j in range(5, 30, 5):
sum = sum + j
print (sum)
```

How about strings? They are also sequences, sequences of characters. We can use a for loop to visit every character in a string:

```
for ch in "alphabet":
      print (ch)
```

Can you figure out what this would display?

How about if you wanted to print out a string in reverse order, or only print every other character of a string? Then you would need to work with the index numbers of the characters and the slice syntax. Feel free to try these and post your answers on the forum.

4 Indefinite Loops and Input Validation

We've already seen some definite loops. Loops where you can tell (roughly) how many times they will execute. There are other types of loop applications that we'll cover here, and they are described as **indefinite loops**.

4.1 Data Validation Loop Algorithms

Think about an ATM for example. After you enter your card you must enter a password. What happens if you get it wrong? The ATM asks you to try again. If you get it wrong a second time? The ATM will ask again. Assuming there is no limit, this process will repeat while the user input is invalid. No counting or accumulation going on here, just repetition until the user gets it right. The ATM has no idea how many tries it will take you to get the input correct. That's what makes it an indefinite loop.

An algorithm for that might be:

> input user guess # user's attempt at his password
> while guess != stored password
>
> display an error message
> input user guess
>
> repeat

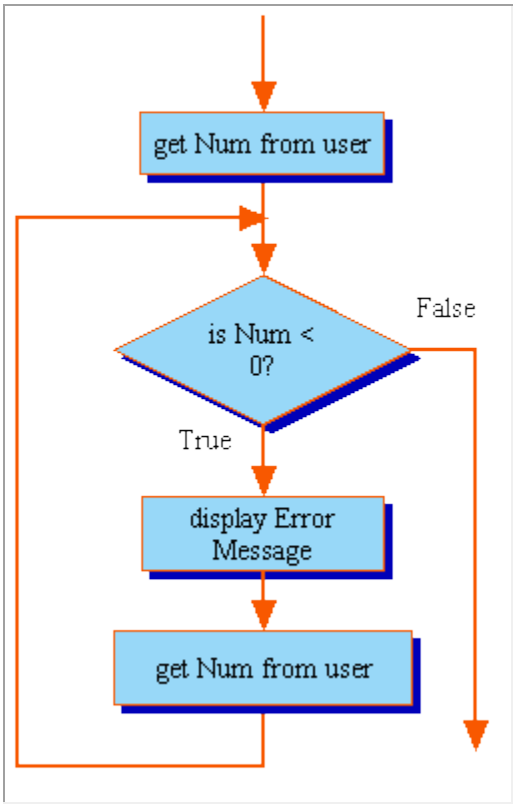
Can you find the initialization, test and update for this loop? The control variable is **guess**. The initialization occurs at the first input statement (above the loop); the update occurs at the other

input statement (within the loop). This type of loop is known as a **data validation loop**. While the data *does not equal* the stored password, the process repeats.

4.2 Data validation loop

This next program asks the user for a positive number, then calculates and displays the square root. The program has to ensure that the user input is positive (valid) before moving on to the calculation/display.

```
import math
num = float(input("Enter a positive number: "))
while num < 0:
  num = float(input ("Error! Please enter a positive number: "))
# at this point we know the value in valid, so continue
print ("The square root is", math.sqrt(num))
```



This application of the while loop is known as an **indefinite loop**. We can't tell by looking at the loop how often it will execute. It all depends on the user. In fact, the body of the loop may never execute! Think about it for a minute, the user may enter a valid number on the very first try. The test of the loop would be false and so we'd skip the body of the loop and just display the square root.

4.3 Input Loop Algorithms with Exit Sentinels

How many of you have been asked to enter a code over the phone (like a pin) followed by the '#' symbol (the pound symbol)? This symbol is an example of an **exit sentinel** or **exit code**. An exit sentinel is used to signal that the user is done entering input (where entering the input was a repetitive process). It is used when the programmer has no idea ahead of time how much data the user has, so the user must signal when they are done.

Here is another example. Assume a program's job is to add up the total cost of grocery items input by the user. But how many items are there? The programmer has no way of knowing when she is writing the program. One user may have 2 items while the next user has 100. So the algorithm has to be flexible enough to stop whenever the user wants to.

Take a look at the pseudocode below.

```
totalCost = 0
groceryItemCost = input the cost of a grocery item, or enter -1 as the exit code
while groceryItemCost != -1

totalCost += groceryItemCost // totalCost is an accumulator variable
groceryItemCost = input the cost of another grocery item, or enter -1 as the exit code

repeat
```

What variable is controlling the loop operation? **groceryItemCost** Where are the test, initialize and update now? Remember, initialize happens before the loop and update happens in the loop body.

In this example, the exit sentinel is the value -1. Note that the exit code must be a value that would not be a valid piece of data. You wouldn't want an exit code of 1.5 because a grocery item might cost 1.50. You could also have an exit code of -99 (or any other negative value). Make sure that the exit code does not get processed as a piece of the data set.

This sentinel-controlled loop is another example of **indefinite loops**. From looking at the code, you cannot tell how often it will repeat. It is all up to the user. These loops may execute 0 or more times. Think about it, the user could enter a value that would cause the test to be false the very first time, so the body of the loop would never execute.

4.3.1 Exit Sentinel Loop

Here is the implementation of the algorithm above.

```
total = 0 # this is our accumulator variable
price = float(input ("Please enter the cost for the first item, or -1 to finish: "))
while price != -1:
if price <= 0: # any other negative value is invalid input
    print ("Error! The only negative value allowed is -1 to stop. Please try again")
else:
    total += price
price = float(input ("Please enter the cost for the first item, or -1 to finish: "))
print ("Your total grocery bill = " , total)
```

Is this loop a definite or indefinite loop? Can you tell just by looking how many times it will repeat or is it based on the user? If you said indefinite loop, you'd be right!

4.4 Flag variables

We've seen control and accumulator variables already. There is one more use of a variable that we need to revisit, the **flag variable**. If you recall, a flag variable is usually a boolean variable that is used to remember if something has occurred within the code. It makes code simpler to test one variable instead of testing a complex condition.

5 Common Loop Errors

Several problems commonly occur when loops are used. Here is a short list, along with some tips to avoid these errors:

- A missing update step is a common cause of **infinite loops**. These are loops that never stop, because the test is always true. To prevent this, think about the update step when you are designing a loop. Make sure that every path through the loop includes an update step that changes the value of the control variable.
- A loop test with an **incorrect 'sense'** (< rather than > or vice versa) can also cause an infinite loop. When writing the loop test, ask yourself "under what conditions should the body of the loop be executed?"
- A loop may contain an **off-by-one error** due to an incorrect relational operator (<= rather than < or similar). When writing the loop test, ask yourself "should the body of the loop be executed when the loop control variable is equal to the limit?"
- The use of a **floating-point number** as a loop control variable can cause errors in several ways. A better choice is to use an integer value as a loop counter if at all possible. If a floating-point number must be used as the loop control variable, never test its value using == or !=; better to use <= or >=.

6 Stepwise Refinement

As you've been learn more features of Python, your programs are becoming more complex. So the question is, when someone (a client, a boss, a teacher) asks you to write a program how should you get started? As you've seen before, it must start with a plan.

6.1 Planning your Design

If you recall, it is important to get your design ideas down on paper before you code. Just like you want a contractor to have plans to build your house, you don't want to start writing code without a design. The tools we've seen for helping to organize your design ideas are flowcharts, pseudocode, and hierarchy charts.

Now that our programs are getting bigger, there are more details to plan, sometimes too much to think about at once. That's why programmers adopt the **stepwise refinement** or **decomposition** approach. Stepwise refinement attempts to solve a complex problem by breaking down larger tasks into smaller tasks. These subtasks may be broken down into even smaller (or finer) tasks, until the size of the task is manageable and you can easily "see" the solution.

6.2 Stepwise Refinement with a Research Paper

Think for a minute how you go about writing a paper. You don't just sit down and start writing, correct? (Hopefully...) What you do is start with an outline. Something like this:

I.
II.
III.
IV.

These are the major topics or sections of your paper. The next step is to go into each section and start to work out the details. You typically focus on one area of the paper at a time:

I.

a.
b.

II.

a.
b.

1.

2.

c.

III.

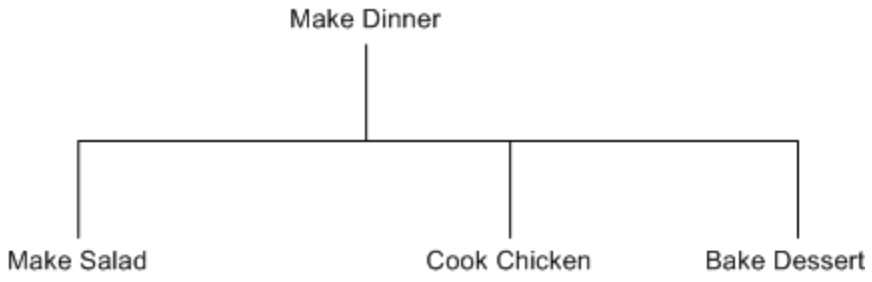
a.
b.

IV.

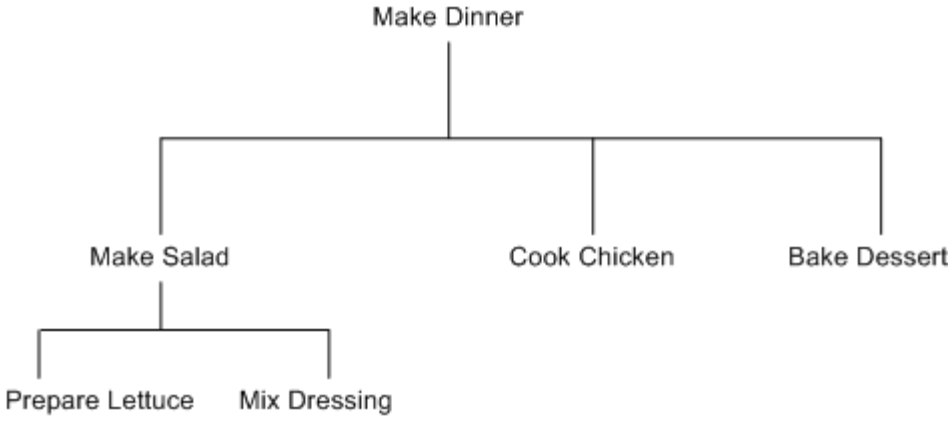
Now, some sections (like section IV) may not need any extra details. You can envision what will go there already. Some sections (like section II)may need a lot more detail (subsection IIb needed to be broken down into even finer details). Once you have this outline done to the level of detail necessary for you to write, then you can start writing the actual paper.

6.3 Another Stepwise Refinement Example

Let's say the task is to cook dinner. You first refinement may look like this:



Now, focus on just one step: making salad. What does that entail? Your continued refinement may look like this:



Realize that everyone's refinement may be different. For some, "Mix Dressing" means opening a bottle, no refinement required. Others, who may make the dressing from scratch, would refine this step further.

How about "Cook Chicken?" What steps would go there? See how the process evolves?

6.4 Stepwise Refinement for an Algorithm

You should adopt the same approach when working on the design of a program. Break the large task (the entire program) into smaller pieces that you can focus on one at a time. Start with the basic flow of any computer program:

INPUT
PROCESS
OUTPUT

Then, think for a moment just on the input process. What are the inputs? What are the data types? Do you need data validation? You may then update the design like so

INPUT

get user's name (string)
get user's grade and validate (integer from 0 to 100)

PROCESS
OUTPUT

This was the first step in refinement. Now, getting data is pretty straightforward, no need to refine that further. But what about validating? At this point in your experience, that may require a little more thought. We've learned to use a loop for data validation, so let's refine this step further

INPUT

get user's name via a prompt (string)
get user's grade via a prompt (integer from 0 to 100) Remember parseFloat!
while (invalid grade)

ask again

PROCESS
OUTPUT

Notice the use of pseudocode here. I'm not focused on syntax, just the logic. There's still more refining to do: putting the actual test in the while loop.

You can see how you would continue this procedure as you move through the PROCESS and OUTPUT sections. Expect the PROCESS section to be the most challenging. That's the place you really want to think about using refinement. Start by just listing the big tasks within the PROCESS section, then focus on solving each task.

6.5 Ideas to Keep in Mind

- When you are working on one section, don't be thinking about another section. Keep your attention focused.
- You don't have to solve everything at once. That's the idea of refinement. For example, try a first pass at refining a section: if a step feels pretty big, just write the step down as one line (such as "DO THIS TASK"), don't worry about the details of that step yet. Then go back to that step and work out its details.
- Don't be impatient with yourself. It takes time to think about how to solve even a small problem. As you gain experience you will get better.
- Write your ideas down. Write your variable names down. Write your formulas down. Write **everything** down! If you are not writing them down and instead are trying to remember it all, you will not be able to focus on the task at hand.
- You may go through different designs the way you go through drafts of a paper. Don't think you have to get it right on the first try. You may end up throwing away some idea because it didn't work. That's ok. Professionals do that too. It's expected.
- Remember, this is pseudocode or flowcharts, not Python syntax. Don't get hung up on syntax at this point. That will just distract you from creating the logic to solve the problem.
- Review your logic as you go. It doesn't make sense to refine something that doesn't work to begin with.
- Once you have your algorithm done, you can rewrite that as Python comments. Use these comments at the beginning of the blocks where you write your code. This way your design is with your code that you (and others) can refer to.

6.6 Functions and Decomposition

Where do functions fit in all this? Sometimes you'll get a design (from a teacher or co-worker) asking for a specific function in your program. Other times, you'll notice that a piece of logic is being repeated in several places; and you know that writing functions allows us to reuse that logic whenever needed. At other times, you'll see that a task has been broken down into several subtasks. To organize this well you may want to put each of those subtasks into a function.

I'll still be giving you specific instructions on functions I want included in an assignment, but feel free to add others as you see fit.

7 Nested Structures

7.1 General Rule about Programming Structures

When drawing flowcharts, we use a rectangular box to represent a simple **sequence** of statements (sequential flow). You can see them in the flowchart example below. The rectangle represents any simple group of statements without a decision or loop. The important thing about the rectangle is that there is one path flowing in and one path flowing out. That is, there is one entry point and one exit point.

A decision structure (if statement) has one entry point and one exit point. A loop also has one entry point and one exit point. Therefore, **in any flowchart, you can replace any rectangle with a complete loop or decision structure**. This allows us to create much more complex structures to solve more complex problems. The different combinations are:

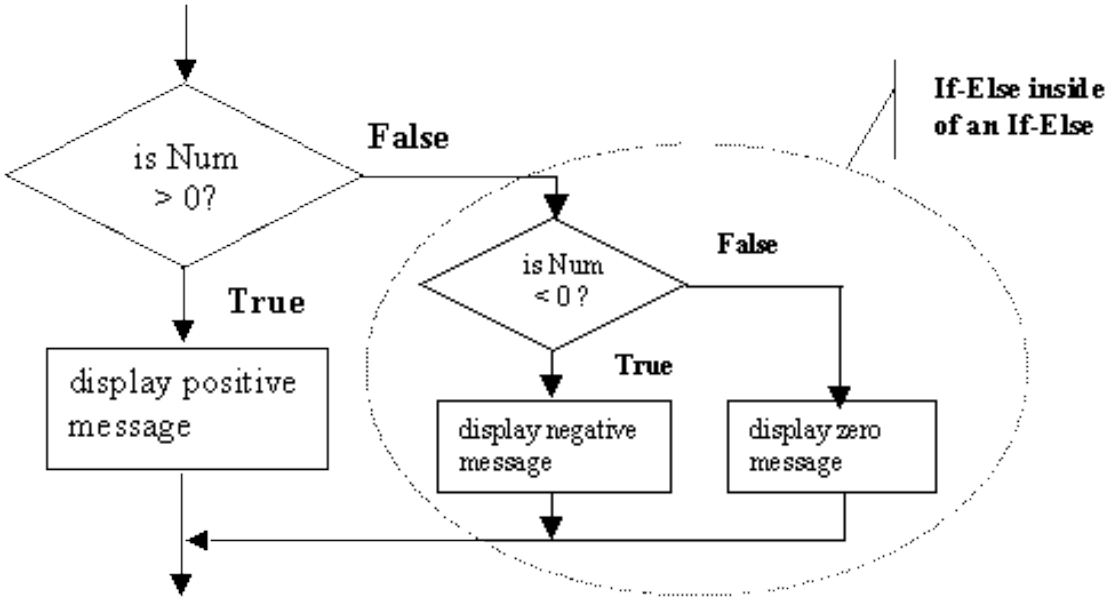
1. if structure within an if structure
2. Loop within a loop
3. if structure within a loop
4. Loop within an if structure

We'll discuss 1 and 2 below. You can see examples of all 4 in the posted examples for last week and this week.

(continues…)

7.2 The Nested if-else if Statement

We have already seen a nested if structure; you accomplish this by **nesting** one if-else structure inside of another. Look at the flowchart below to see three paths. One path will be followed if the value entered is positive (> 0), one will be followed if the value is negative (< 0), and the third path will be followed if neither of the first two paths is followed (num == 0). The code for this flowchart is shown below. You see that the red if - else statement is completely nested within the else clause of the blue if - else statement. The curly braces help to illustrate that.




```
num = float(input("Enter a number: "))
if num > 0:
    print ( "The number is greater than zero" )
else:
    if num < 0:
        print ( "The number is less than zero"  )
    else:
        print ( "The number is 0" )
```

Nested loops

Nested loops are often used to solve problems that require two or more variables to be varied independently. Here are two simple rules to follow when nesting loops:

1. The **inner loop** must be contained completely within the body of the **outer loop**. This is just an extension of the general rule for nesting any structure -- the inner structure must be completely contained within the outer structure.
2. The loop control variables must be different. (However, the starting value or the test value of the inner loop's control variable may be based on the outer loop's control variable)

Trace nested loops in the same way you would any other -- one line at a time. One thing to realize is that the inner loop begins its process over again from scratch, with each iteration of the outer

loop. This also means that the outer loop cannot continue to its next iteration until the inner loop has gone through a complete cycle.

Here is one example of nested loops and its output. Trace through the example so it makes sense to you.

Code

```
for i in [1, 2, 3]:
    print ( "i = ", i )
    # here's the inner loop
    for j in [1, 2, 3]:
        print  ( j, end ='' ) # write all on one line
    print ( ) # move to a new line
```

Display

```
i = 1
1 2 3
i = 2
1 2 3
i = 3
1 2 3
```

Here is another example, where the starting value of the inner loop control variable is based on the outer loops value. Again, trace through it so that it makes sense. Notice how the inner loop's ending place gets bigger each time, because the inner loop's test is based on **i**, and **i** gets bigger each time.

Code

```
for i in [1, 2, 3]:
    print ( "i = ", i  )
    # here's the inner loop make sure you understand the call to range()
    for j in range(1, i * 2 + 1):
        print ( j, end='' ) # write all on one line
    print ( ) # move to a new line
```

Display

```
i = 1
1 2
i = 2
1 2 3 4
i = 3
1 2 3 4 5 6
```

7.3 Designing Nested Loops - an example

How can we create a simple 10 x 10 multiplication table? Like this diagram:

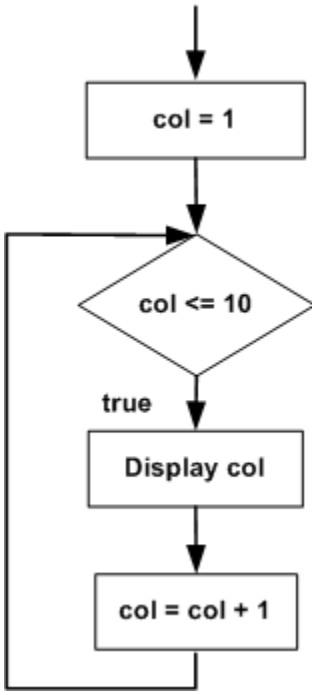
1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30
4	8	12	16	20	24	28	32	36	40
5	10	15	20	25	30	35	40	45	50
6	12	18	24	30	36	42	48	54	60
7	14	21	28	35	42	49	56	63	70
8	16	24	32	40	48	56	64	72	80
9	18	27	36	45	54	63	72	81	90
10	20	30	40	50	60	70	80	90	100

Imagine how we would use a loop to make one row; then we will use a 2nd loop (an outer loop) to repeatedly create all the rows.

Step 1 - create one row

Use a loop to create one row. If you look at the first row, that's just counting to 10 (we'll adjust for the other numbers later). The code and the flowchart are shown below:

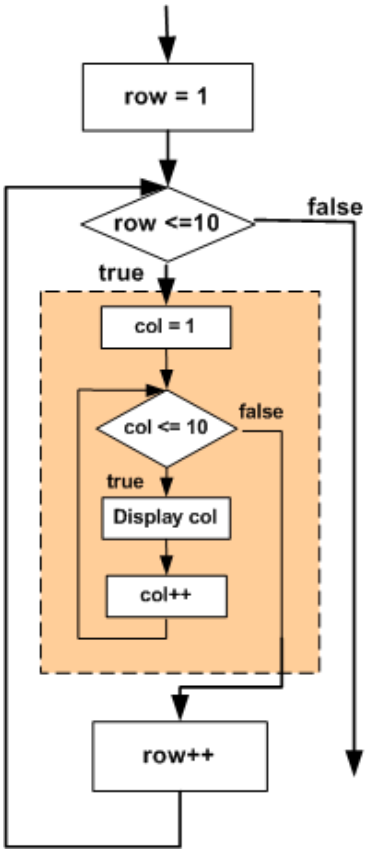
```
for col in range(1, 11): # determines how many in each row
    print ( col, end='' )
print ( )  # now advance to the next line
```



Step 2 - create multiple rows

Imagine that the loop above is now a unit of code. It creates one row. In order to create many rows, you can use another loop and put this unit of code inside of it. The **inner loop** creates 1 row, the **outer loop** determines how many rows. In the example above there are 10 rows, so again we'll count from 1 to 10.

```
for row in range(1, 11): # this determines the number of rows
    for col in range(1, 11): # determines how many per row
        print ( col, end = '' )
    print( ) # now advance to the next line
```



Step 3 - Make it Calculate the Entries

At this point, you have 10 rows each displaying 1 to 10. In a multiplication table, each entry is based upon its location in the table. That is, if an entry is in row i and in column j, its value is i * j. We make one modification to the nested loops to accomplish this:

```
for row in range(1, 11):
    for col in range(1, 11):
        print (row*col, end='') # multiply row and column together
    print ( ) # now advance to the next line
```