Loops & Iterations

Lecture #5 Notes – Python - Name _____ Class _____

5.1 A common pattern in assignment statements is an assignment statement that updates a variable – where the new value of the variable depends on the old.

```
x = x+1
```

This means "**get the current value of `x`, add 1, and then update `x` with the new value**."

Before you can update a variable, you have to **initialize** it, usually with a simple assignment:

```
>>> x = 0
>>> x = x+1
```

Updating a variable by adding 1 is called an **increment**; subtracting 1 is called a **decrement**.

5.2 More formally, here is the flow of execution for a `while` statement:

Evaluate the condition, yielding **`True`** or **`False`**.

If the condition is false, **exit the `while` statement and continue execution at the next statement.**

If the condition is true, **execute the body and then go back to step 1**.

```
n = 5
while n>0:
   print (n)
   n=n-1
print ('Blastoff')
print (n)
```

How many times will the above program loop run? _____

This type of flow is called a **loop** because the third step loops back around to the top. We call each time we execute the body of the loop an **iteration**. For the above loop, we would say, "It had five iterations", which means that the body of the loop was executed five times.

5.4 Infinite loops and break

PROGRAM

```
n = 10
while True:
   print (n),
   n = n - 1
print ('Done!')
```

This loop above is obviously an **infinite loop** because the logical expression on the `while` statement is simply the logical constant **`True.`**

While this is a dysfunctional infinite loop, we can still use this pattern to build useful loops as long as we carefully add code to the body of the loop to explicitly exit the loop using `break` when we have reached the exit condition.

For example, suppose you want to take input from the user until they type `done`. You could write:

```
PROGRAM
while True:
    line = input('type a word> ')
    if line == 'done':
        break
    print (line)
print ('Done!')
```

The loop condition is `True`, which is always true, so the loop runs repeatedly until it hits the break statement.

This way of writing `while` loops is common because you can check the condition **anywhere in the loop** (not just at the top) and you can express the stop condition affirmatively ("**stop when this happens**") rather than negatively ("**keep going until that happens**.").

5.5 Finishing iterations with continue

Sometimes you are in an iteration of a loop and want to finish the current iteration and immediately jump to the next iteration. In that case you can use the continue statement to skip to the next iteration without finishing the body of the loop for the current iteration.

Here is an example of a loop that copies its input until the user types "done", but treats lines that start with the hash character as lines not to be printed (kind of like Python comments).

PROGRAM – run this several times until you understand it

```
while True:
    line = input('> ')
    if line[0] == '-' :
        continue
    if line == 'done':
        break
    print (line)
print ('Done!')
```

Here is a sample run of this new program with continue added.

```
> hello there
hello there
> # don't print this
> print this!
print this!
> done
Done!
```

All the lines are printed except the one that starts with the hash sign because when the continue is executed, it ends the current iteration and **jumps back to the while statement** to start the next iteration, thus **skipping the print statement**.

5.6 Definite loops using for

Sometimes we want to loop through a **set of things** such as a **list of words**, the **lines in a file**, or a **list of numbers**. When we have a list of things to loop through, we can construct a ***definite*** loop using a for statement. We call the while statement an ***indefinite*** loop because it simply loops until some condition becomes **False**, whereas the for loop is looping through a known set of items so it runs through as many iterations **as there are items in the set**.

The syntax of a for loop is similar to the while loop in that there is a for statement and a loop body:

PROGRAM

```
friends = ['Joseph', 'Glenn', 'Sally']
for friend in friends:
   print ('Happy New Year:', friend)
print ('Done!')
```

Translating this `for` loop to English is not as direct as the `while`, but if you think of friends as a **set**, it goes like this: "Run the statements in the body of the for loop once for each friend *in* the set named friends."

Looking at the `for` loop, **for** and **in** are **reserved Python keywords**, and `friend` and `friends` are **variables**.

In particular, `friend` is the **iteration variable** for the for loop. The variable **`friend`** changes for each iteration of the loop and controls when the `for` loop completes. The **iteration variable** steps successively through the three strings stored in the `friends` variable.

5.7 Loop patterns

Often we use a `for` or `while` loop to go through a list of items or the contents of a file and we are looking for something such as the largest or smallest value of the data we scan through.

These loops are generally constructed by:

- **Initializing** one or more variables before the loop starts
- **Performing some computation on each item in the loop body**, possibly changing the variables in the body of the loop
- Looking at the **resulting variables** when the loop completes

We will use a list of numbers to demonstrate the concepts and construction of these loop patterns.

Counting loop – PROGRAM

```
count = 0
for itervar in [3, 41, 12, 9, 74, 15]:
   count = count + 1
print ('Count: ', count)
```

Total loop – PROGRAM

```
total = 0
for itervar in [3, 41, 12, 9, 74, 15]:
   total = total + itervar
print ('Total: ', total)
```

Neither the counting loop nor the summing loop are particularly useful in practice because there are built-in functions `len()` and `sum()` that compute the number of items in a list and the total of the items in the list respectively.

PROGRAM –

```
k = [3, 41, 12, 9, 74, 15]
print ('Total: ', sum(k))
```

PROGRAM -

```
k = [3, 41, 12, 9, 74, 15]
print ('Total: ', len(k))
```

5.7.2 Maximum and minimum loops – READ THIS section and write the following programs

PROGRAM –

```
largest = None
print ('Before:', largest)
for itervar in [3, 41, 12, 9, 74, 15]:
   if largest is None or itervar > largest :
      largest = itervar
   print ('Loop:', itervar, largest)
print ('Largest:', largest)
```

PROGRAM –

```
smallest = None
print ('Before:', smallest)
for itervar in [3, 41, 12, 9, 74, 15]:
   if smallest is None or itervar < smallest:
      smallest = itervar
   print ('Loop:', itervar, smallest)
print ('Smallest:', smallest)
```

shorter function –

PROGRAM

```
k = [3, 41, 12, 9, 74, 15]
print (min(k))
```

5.8 Debugging – READ THIS SECTION FOR EXCELLENT ADVICE

Work on Ex 5.10 in online textbook (#1 and #2)

Score : ____ / 10 Answers

____ / 10 Participation / Attitude