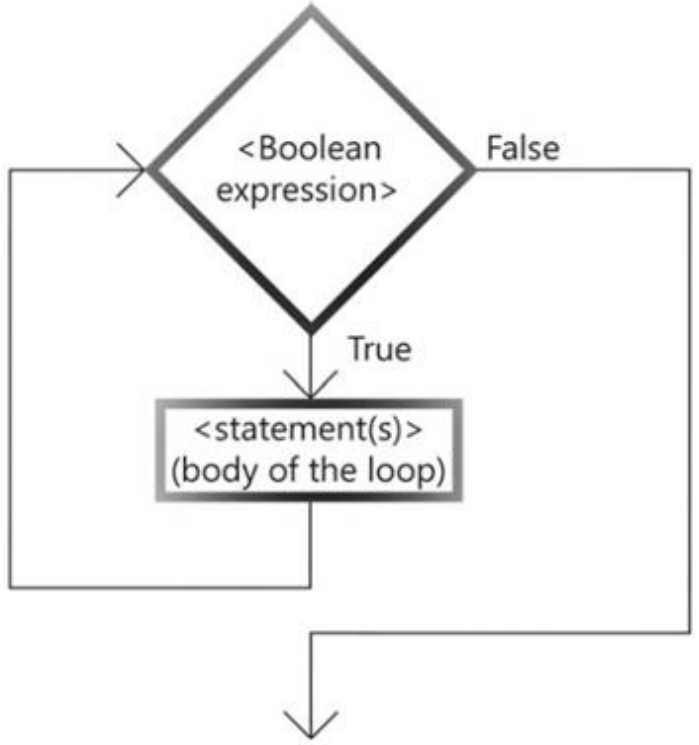
DSC 510

WEEK 5 LOOPS

Loops

A loop is a block of code that is repeated until a certain condition is met. Within Python there are various types of loops including the for loop (which we previously introduced) and while loops.



Real world Example

To understand how this works, let's start with a silly real-life example.

```
  I am hungry
  while hungry
       take a bite of food
       chew
       swallow
       if thirsty:
            take a drink
  go back to the point of checking if I am still hungry
eat dessert!
```

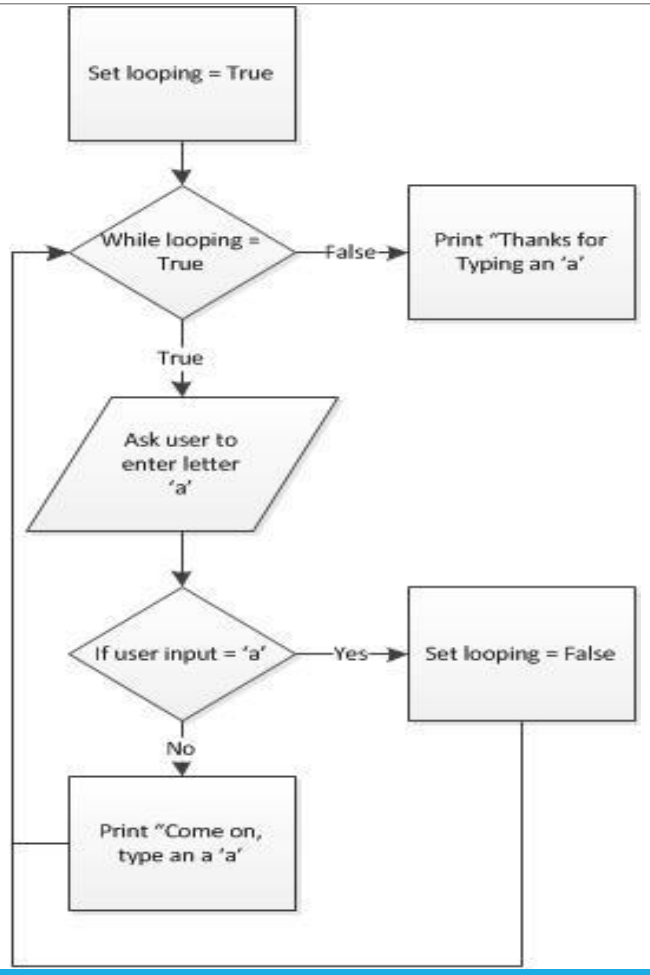
In this example, we repeatedly take a bite of food, chew, swallow, and if we are thirsty, take a drink. This process keeps going as long as we are still hungry. When we eventually reach the point where we are no longer hungry, the loop finishes and we eat dessert.

While Statement

In Python (as with many languages), a loop is implemented with a while statement. This is the generic form of the while statement:

```
while <Boolean expression>: # as long as the expression evaluates to True
 <indented block of statements>
```

While Example Flowchart



While Example

```
looping = True
while looping == True:
    answer = input("Please type the letter 'a': ")
    if answer == 'a':
        looping = False   # we're done, exit the loop
    else:
        print "Come on, type an 'a'! ”
print("Thanks for typing an 'a'")
```

If the example above we set a Boolean variable looping to True so that the while loop runs for the first time the program runs. Within the loop we ask the user to enter a letter. We then evaluate the letter to see if it matches the letter ”a”. If so we set looping to false, otherwise we ask the user to again and looping remains set to true. Regardless of our evaluation of the user input the loop will return to be evaluated again as demonstrated in the flowchart.

While Loop Example

```
#Add up numbers from 1 to a target number

target = input('Enter a target number: ')
target = int(target)
total = 0
nextNumberToAddIn = 1
while nextNumberToAddIn <= target:
 # add in the next value
        total = total + nextNumberToAddIn #add in the next number
        print('Added in:', nextNumberToAddIn, 'Total so far is:', total)
 nextNumberToAddIn = nextNumberToAddIn + 1
print('The sum of the numbers from 1 to', target, 'is:', total)
```

Input Error Checking with While Loops

- While loops are useful for validating conditions when running a program.

Sentinel Values

Sentinel values are often used in programming in order to control loops. It is common to have a while loop which will run until the user wishes to stop. In these situations we use a Sentinel value to allow the user to stop the running of the program.

Sentinel Example:

```
total = 0.0
count = 0
x = float(input("Enter a number (negative to quit) >> "))
while x >= 0:
    total = total + x
    count = count + 1
    x = float(input("Enter a number (negative to quit) >> "))
print("\nThe average of the numbers is", total / count)
```

Definite Loops

- A definite loop is one in which the number of times the loop will iterate is determine prior to the loop being executed.

In the below example the loop will run `n` times:

Indefinite Loops

- An Indefinite loop is one in which the number of times the loop will run is not predetermined. An example of this might be a loop which performs error checking. The loop will run as many times as necessary until a correct condition is entered.

In the following example the loop will run as long as the input is not Y:

While vs For Loops

```
i = 0
while i <= 10:
    print(i)
    i = i + 1
```

This code will have the same output as if we had written a for loop like this:

```
for i in range(11):
     print(i)
```

Notice that the while version requires us to take care of initializing `i` before the loop and incrementing `i` at the bottom of the loop body. In the for loop, the loop variable is handled automatically. In our while loop example, if we forgot to increment `i` we would end up with a loop which is indefinite and runs forever. It's important to have a condition which stops the loop. This is not a common occurrence in a "for" loop since the condition which stops the loop is often built into the statement itself.

For Statement

In Python the For Statement is often used to iterate over data as seen previously in our discussion of Lists. Below is the generic form of a For statement.

```
for iterating_var in sequence:
 <indented block of statements(s)>
```

For Statement Example

```
>>> #Measure some strings:
>>> words = ['Omaha', 'Chicago', 'New York', 'Singapore']
>>> for w in words:
 print(w, len(w))
Omaha 5
Chicago 7
New York 8
Singapore 9
>>>
```

In this example we use a for loop to iterate through a List of cities. We then print the length of the each string (city name) by using the `len` function.

For Statement with Else Clause

It is also possible to use an Else statement from within a For statement to have a default condition. Below is an example using a list. Notice that when the below program is run, the Else runs once the conditions for the for are no longer met.

```
>>> list = ["geeks", "for", "python"]
>>> for index in range(len(list)):
 print (list[index])
else:
 print("Inside Else Block")

Geeks
For
Python
Inside Else Block
>>>
```

Loops with Control Statements

- Loops can also use control statements in order to control the execution of the loop's normal flow. These include the continue, break, and pass statements:
  - The break statement, like in C, breaks out of the innermost enclosing for or while loop.
  - The pass statement does nothing. It can be used when a statement is required syntactically but the program requires no action.
  - The continue statement, also borrowed from C, continues with the next iteration of the loop.

Break Example

The break statement is extremely useful and will solve the issue we saw previously where the else within a “for” loop executes no matter what. Loop statements may have an else clause; it is executed when the loop terminates through exhaustion of the list (with for) or when the condition becomes false (with while), but not when the loop is terminated by a break statement. This is exemplified by the following loop, which searches for prime numbers:

```
>>> for n in range(2, 10):
 for x in range(2, n):
        if n % x == 0:
                print (n, 'equals', x, '*', n/x )
        break
        else:
        print (n, 'is a prime number’)
```

Continue Example

```
>>> for num in range(2, 10):
 if num % 2 == 0:
        print ("Found an even number", num)
        continue
 print ("Found a number", num)

Found an even number 2
Found a number 3
Found an even number 4
Found a number 5
Found an even number 6
Found a number 7
Found an even number 8
Found a number 9
>>>
```

Items()

- PEP 20 states “There should be one– and preferably only one –obvious way to do it.” The preferred way to iterate over the key-value pairs of a dictionary is to declare two variables in a for loop, and then call dictionary.items(), where dictionary is the name of your variable representing a dictionary. For each loop iteration, Python will automatically assign the first variable as the key and the second variable as the value for that key.
- Best Practice is to use items() to iterate across dictionaries. The updated code below demonstrates the Pythonic style for iterating through a dictionary. When you define two variables in a for loop in conjunction with a call to items() on a dictionary, Python automatically assigns the first variable as the name of a key in that dictionary, and the second variable as the corresponding value for that key. See example below:

```
d = {"first_name": "Alfred", "last_name":"Hitchcock"}

for key,val in d.items():
    print("{} = {}".format(key, val))
```