print("x=" + str(x))

while x < 5:

  print("Not there yet, x=" + str(x))

  x = x + 1

x = 0

How many times will "Not there yet" be printed?



**Correct Response**

You got it! The variable x starts at 0 and gets incremented once per iteration, so there are 5 iterations for which x is smaller than 5.

Anatomy of a While Loop

A *while* loop will continuously execute code depending on the value of a condition. It begins with the keyword *while,* followed by a comparison to be evaluated, then a colon. On the next line is the code block to be executed, indented to the right. Similar to an *if* statement, the code in the body will only be executed if the comparison is evaluated to be true. What sets a *while* loop apart, however, is that this code block will keep executing as long as the evaluation statement is true. Once the statement is no longer true, the loop exits and the next line of code will be executed.

Can you work out what this function does? Try passing different parameters to the attempts function to see what it does.

def attempts(n):

    x = 1

    while x <= n:

        print("Attempt " + str(x))

        x += 1

    print("Done")

attempts(5)

Attempt 1

Attempt 2

Attempt 3

Attempt 4

Attempt 5

Done

In this code, there's an initialization problem that's causing our function to behave incorrectly. Can you find the problem and fix it?

  while (start\_number > 0):

    print(start\_number)

    start\_number -= 1

  print("Zero!")

def count\_down(start\_number):

count\_down(3)

Here is your output:

3

2

1

Zero!

You nailed it! By initializing the current variable you got

the function to behave correctly.

Common Pitfalls With Variable Initialization

You'll want to watch out for a common mistake: forgetting to initialize variables. If you try to use a variable without first initializing it, you'll run into a **NameError**. This is the Python interpreter catching the mistake and telling you that you’re using an undefined variable. The fix is pretty simple: initialize the variable by assigning the variable a value before you use it.

Another common mistake to watch out for that can be a little trickier to spot is forgetting to initialize variables with the correct value. If you use a variable earlier in your code and then reuse it later in a loop without first setting the value to something you want, your code may wind up doing something you didn't expect. Don't forget to initialize your variables before using them!

The following code causes an infinite loop. Can you figure out what’s missing and how to fix it?

def print\_range(start, end):

    n = start

    while n <= end:

        print(n)

        n+=1

print\_range(1, 5)  # Should print 1 2 3 4 5 (each number on its own line)

Here is your output:

1

2

3

4

5

Great work! You've managed to fix the error in the code that

was causing an infinite loop!

Infinite loops and Code Blocks

Another easy mistake that can happen when using loops is introducing an infinite loop. An infinite loop means the code block in the loop will continue to execute and never stop. This can happen when the condition being evaluated in a *while* loop doesn't change. Pay close attention to your variables and what possible values they can take. Think about unexpected values, like zero.

In the Coursera code blocks, you may see an error message that reads "Evaluation took more than 5 seconds to complete." This means that the code encountered an infinite loop, and it timed out after 5 seconds. You should take a closer look at the code and variables to spot where the infinite loop is.