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.

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!

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.

Study Guide: while Loops

This study guide provides a quick-reference summary of what you learned in this segment and serves as a guide for the upcoming practice quiz.

In the **while** Loops segment, you learned about the logical structure and syntax of **while** loops. You also learned about the importance of initializing variables and how to resolve infinite **while** loops with the **break** keyword.

while Loops

A **while** loop executes the body of the loop while a specified condition remains True. They are commonly used when there's an unknown number of operations to be performed, and a condition needs to be checked at each iteration.

Syntax:

```
while specified condition is True:
    body of loop
```

Example **while** loop:

```
multiplier = 1
result = multiplier*5
while result <= 50:
  print(result)
  multiplier += 1
  result = multiplier*5
print("Done")

# This while loop prints the multiples of 5 between 1 and 50. The
# "multiplier" variable is initialized with the starting value of 1.
# The "result" variable is initialized with the value of the
# "multiplier" variable times 5.

# The while loop specifies that the loop must iterate while it is True
# that the "result" is less than or equal to 50. Within the while loop,
# the code tells the Python interpreter to print the value of the
# "result" variable. Then, the "multiplier" is incremented by 1 and the
# "result" is assigned the new value of the "multiplier" times 5.

# The end of the while loop is indicated by the indentation of the next
# line of code moving one tab to the left. At this point, the Python
# interpreter automatically loops back to the beginning of the while
# loop to check the condition again with the new value of the "result"
# variable. When the while loop condition becomes False (meaning
# "result" is no longer less than or equal to 50), the interpreter exits
# the loop and reads the next line of code outside of the loop. In this
# case, that next line tells the interpreter to print the string "Done".

# Click the "Run" button to check the result of this while loop.
```

Common Errors in while Loops

If you get an error message on a loop or it appears to hang, your debugging checklist should include the following checks:

- **Failure to initialize variables.** Make sure all the variables used in the loop’s condition are initialized before the loop.
- **Unintended infinite loops.** Make sure that the body of the loop modifies the variables used in the condition, so that the loop will eventually end for all possible values of the variables. You can often prevent an infinite loop by using the **break** keyword or by adding end criteria to the condition part of the **while** loop.

while Loop Terms

- **while loop** - Tells the computer to execute a set of instructions while a specified condition is True. In other words, **while** loops keep executing the same group of instructions until the condition becomes False.
- **infinite loop** - Missing a method for exiting the loop, causing the loop to run forever.
- **break** - A keyword that can be used to end a loop at a specific point.

Math Concepts on the Practice Quiz

The coding problems on the upcoming practice quiz will involve a few math concepts. Don’t worry if you are rusty on math. You will have plenty of support with these concepts on the quiz. The following is a quick overview of the math terms you will encounter on the quiz:

- **prime numbers** - Integers that have only two factors, which are the number itself multiplied by 1. The lowest prime number is 2.
- **prime factors** - Prime numbers that are factors of an integer. For example, the prime numbers 2 and 5 are the prime factors of the number 10 (2x5=10). The prime factors of an integer will not produce a remainder when used to divide that integer.
- **divisor** - A number (denominator) that is used to divide another number (numerator). For example, if the number 10 is divided by 5, the number 5 is the divisor.
- **sum of all divisors of a number** - The result of adding all of the divisors of a number together.
- **multiplication table** - An integer multiplied by a series of numbers and their results formatted as a table or a list. For example:

4x1=4	4x2=8	4x3=12	4x4=16	4x5=20

Coding skills

Skill Group 1

- Initialize a variable
- Use a **while** loop that runs while a specific condition is true
- Ensure the **while** loop will not be an infinite loop
- Increment a value within a **while** loop

```
# This function counts the number of integer factors for a
# "given_number" variable, passed through the function’s parameters.
# The "count" return value includes the "given_number" itself as a
# factor (n*1).
def count_factors(given_number):

  # To include the "given_number" variable as a "factor", initialize
  # the "factor" variable with the value 1 (if the "factor" variable
  # were to start at 2, the "given_number" itself would be excluded).
  factor = 1
  count = 1

  # This "if" block will run if the "given_number" equals 0.
  if given_number == 0:
    # If True, the return value will be 0 factors.
    return 0

  # The while loop will run while the "factor" is still less than
  # the "given_number" variable.
  while factor < given_number:
    # This "if" block checks if the "given_number" can be divided by
    # the "factor" variable without leaving a remainder. The modulo
    # operator % is used to test for a remainder.
    if given_number % factor == 0:
      # If True, then the "factor" variable is added to the count of
      # the "given_number"’s integer factors.
      count += 1
    # When exiting the if block, increment the "factor" variable by 1
    # to divide the "given_number" variable by a new "factor" value
    # inside the while loop.
    factor += 1

  # When the interpreter exits either the while loop or the top if
  # block, it will return the value of the "count" variable.
  return count

print(count_factors(0)) # Count value will be 0
print(count_factors(3)) # Should count 2 factors (1x3)
print(count_factors(10)) # Should count 4 factors (1x10, 2x5)
print(count_factors(24)) # Should count 8 factors (1x24, 2x12, 3x8,
```

Skill Group 2

- Initialize variables to assign data types before they are used in a **while** loop
- Use the **break** keyword as an exit point for a **while** loop

```
# This function outputs an addition table. It is written to end after
# printing 5 lines of the addition table, but it will break out of the
# loop if the "my_sum" variable exceeds 20.

# The function accepts a "given_number" variable through its
# parameters.
def addition_table(given_number):

    # The "iterated_number" and "my_sum" variables are initialized with
    # the value of 1. Although the "my_sum" variable does not need any
    # specific initial value, it still must be assigned a data type
    # before being used in the while loop. By initializing "my_sum"
    # with any integer, the data type will be set to int.
    iterated_number = 1
    my_sum = 1

    # The while loop will run while it is True that the
    # "iterated_number" is less than or equal to 5.
    while iterated_number <= 5:

        # The "my_sum" variable is assigned the value of the
        # "given_number" plus the "iterated_number" variables.
        my_sum = given_number + iterated_number

        # Test to see if the "my_sum" variable is greater than 20.
        if my_sum > 20:
            # If True, then use the break keyword to exit the loop.
            break
        # If False, the Python interpreter will move to the next line
        # in the while loop after the if-statement has ended.

        # The print function will output the "given_number" plus
        # the "iterated_number" equals "my_sum".
        print(str(given_number), "+", str(iterated_number), "=", str(my_sum))

        # Increment the "iterated_number" before the while loop starts
        # over again to print a new "my_sum" value.
        iterated_number += 1
```

Python practice information

For additional Python practice, the following links will take you to several popular online interpreters and codepads:

- Welcome to Python
- Online Python Interpreter
- Create a new Repl
- Online Python-3 Compiler (Interpreter)
- Compile Python 3 Online
- Your Python Trinket

For Loops Recap

For loops allow you to iterate over a sequence of values. Let's take the example from the beginning of the video:

```
for x in range(5):

  print(x)
```

Similar to *if* statements and *while* loops, *for* loops begin with the keyword ***for*** with a colon at the end of the line. Just like in function definitions, *while* loops and *if* statements, the body of the *for* loop begins on the next line and is indented to the right. But what about the stuff in between the *for* keyword and the colon? In our example, we're using the *range()* function to create a sequence of numbers that our *for* loop can iterate over. In this case, our variable **x** points to the current element in the sequence as the *for* loop iterates over the sequence of numbers. Keep in mind that in Python and many programming languages, a range of numbers will start at 0, and the list of numbers generated will be one less than the provided value. So *range(5)* will generate a sequence of numbers from 0 to 4, for a total of 5 numbers.

Bringing this all together, the range(5) function will create a sequence of numbers from 0 to 4. Our *for* loop will iterate over this sequence of numbers, one at a time, making the numbers accessible via the variable **x** and the code within our loop body will execute for each iteration through the sequence. So for the first loop, **x** will contain 0, the next loop, 1, and so on until it reaches 4. Once the end of the sequence comes up, the loop will exit and the code will continue.

The power of *for* loops comes from the fact that it can iterate over a sequence of any kind of data, not just a range of numbers. You can use *for* loops to iterate over a list of strings, such as usernames or lines in a file.

Not sure whether to use a *for* loop or a *while* loop? Remember that a *while* loop is great for performing an action over and over until a condition has changed. A *for* loop works well when you want to iterate over a sequence of elements.

A Closer Look at the Range() Function

The **in** keyword, when used with the **range()** function, generates a sequence of integer numbers, which can be used with a **for** loop to control the start point, the end point, and the incremental values of the loop.

Syntax:

```
for n in range(x, y, z):
    print(n)
```

The **range()** function uses a set of indices that point to integer values, which start at the number 0. The numeric values 0, 1, 2, 3, 4 correlate to ordinal index positions 1st, 2nd, 3rd, 4th, 5th. So, when a range call to the 5th index position is made using **range(5)** the index is pointing to the numeric value of 4.

Index Number	**1st index**	**2nd index**	**3rd index**	**4th index**	**5th index**
Value	0	1	2	3	4

The **range()** function can take up to three parameters: **range(start, stop, step)**

Start The first item in the **range()** function parameters is the starting position of the range. The default is the first index position, which points to the numeric value 0. This value is included in the range.

Stop The second item in the **range()** function parameters is the ending position of the range. There is no default index position, so this index number must be given to the **range()** parameters. For example, the line **for n in range(4)** will loop 4 times with the **n** variable starting at 0 and looping 4 index positions: 0, 1, 2, 3. As you can see, **range(4)** (meaning index position 4) ends at the numeric value 3. In Python, this structure may be phrased as "the end-of-range value is *excluded* from the range." In order to include the value 4 in **range(4)**, the syntax can be written as **range(4+1)** or **range(5)**. Both of these ranges will produce the numeric values 0, 1, 2, 3, 4.

Step The third item in the **range()** function parameters is the incremental step value. The default increment is +1. The default value can be overridden with any valid increment. However, note that the loop will still end at the end-of-range index position, regardless of the incremental value. For example, if you have a loop with the range: **for n in range(1, 5, 6)**, the range will only produce the numeric value 1. This is because the incremental value of 6 exceeded the ending point of the range.

Practice Exercise

You can use the code block below to test the values of **n** with various **range()** parameters. A few suggestions to test are:

range(stop)

- range(3)
- range(3+1)

range(start, stop)

- range(2, 6)
- range(5,10+1)

range(start, stop, step)

- range(4, 15+1, 2)
- range(2*2, 25, 3+2)
- range(10, 0, -2)

```
for n in range(1, 5, 6):
    print(n)
```

Examples of the range() function in code:

Example 1

```
# This loop iterates on the value of the "n" variable in a range
# of 0 to 10 (the value of the end-of-range index 11 is excluded).
# The incremental value for the loop is 2. The print() function will
# output the resulting value of "n" as the loop counts from 0 to 10
# (end-of-range index 11) in incremental steps of 2. This is one
# method that can be used in Python to print a list of even numbers.


for n in range(0,11,2):
    print(n)


# The loop should print 0, 2, 4, 6, 8, 10
```

Example 2

```
# This loop iterates on the value of the "number" variable in a range
# of 2 to 7+1 (the value of the end-of-range index 7 is excluded, so
# +1 has been added to the parameter to include the numeric value 7 in
# the range). The incremental value for the loop is the default of +1.
# The print() function will output the resulting value of "number"
# multiplied by 3.


for number in range(2,7+1):
    print(number*3)


# The loop should print 6, 9, 12, 15, 18, 21
```

Example 3

```
# This loop iterates on the value of the "x" variable in a range
# of 2 to -1 (the end-of-range index -2 is excluded). The third
# parameter is also a negative number, making it a decremental value
# of -1. The print() function will output the resulting value of
# "x" as it starts at 2 and counts down to -1 (index -2).


for x in range(2, -2, -1):
    print(x)


# The loop should print 2, 1, 0, -1
```

Key takeaways

The roles of the **range(start, stop, step)** function parameters are:

- **Start** - Beginning of range
  - value included in range
  - default = 0
- **Stop** - End of range

    - value excluded from range (to include, use stop+1)
    - no default
    - must provide the ending index number
- **Step** - Incremental value
    - default = 1

Resources for more information

- Python range() function - This site provides some helpful visualizations for the range index positions. It also offers multiple **for** x **in range()** examples and practice exercises.

For additional Python practice, the following links will take you to several popular online interpreters and codepads:

- Welcome to Python
- Online Python Interpreter
- Create a new Repl
- Online Python-3 Compiler (Interpreter)
- Compile Python 3 Online
- Your Python Trinket

Study Guide: for Loops

This study guide provides a summary of what you learned in this segment and serves as a guide for the upcoming practice quiz.

In the **for** Loops segment, you learned about the logical structure and syntax of **for** loops. You took a closer look at the **range()** function. You learned about nested **for** loops and complex nested **for** loops with **if** statements. You also learned how to fix common errors in **for** loops.

for Loops vs. while Loops

for loops and **while** loops share several characteristics. Both loops can be used with a variety of data types, both can be nested, and both can be used with the keywords **break** and **continue**. However, there are important differences between the two types of loops:

- **while** loops are used when a segment of code needs to execute repeatedly **while** a condition is true
- **for** loops iterate over a sequence of elements, executing the body of the loop **for** each element in the sequence

Syntax

The syntax of a **for** loop with the **in** keyword:

```
    body of loop
for variable in sequence:
```

Common for Loop Structures

for Loop with range()

The **in** keyword with the **range()** function generates a sequence of integer numbers, which can be used with a **for** loop to configure the iterations of the code. The range of numbers [0, 1, 2] correlates to ordinal index positions

(1st, 2nd, 3rd), rather than the cardinal (quantity) values of the numbers 0, 1, and 2. For example, range(5) means the five index positions in the range [0, 1, 2, 3, 4].

The **range()** function can take up to three parameters. The roles of the three possible **range(x,y,z)** parameters are:

- **x = Start** - Starting index position of the range
    - Default index position is 0.
    - The starting index position is included in the range.
    - Example syntax: range(**2**, y, z) or range(**x+3**, y, z)
- **y = Stop** - Ending index position of range
    - No default index position. Must include the ending index position in the range() parameters.
        - Example syntax: range(**y**)
    - The value of the ending index position is excluded from the range.
    - To include the ending index number, use the expression: **y+1** (index + 1)
        - Example syntax: range(x, **y+1**, z)
        - Alternatively, if **y** = 10, you can write: range(x, **11**, z)
- **z = Step** - Incremental value
    - Default increment is +1.
    - The default value can be overridden with any valid increment.
    - The incremental value will end the for loop before it reaches the end of range index position (end of range index minus 1).

Example of a **for** loop with the **in** keyword and the **range()** function:

```
# This loop iterates on the value of the "number" variable in a range
# of 1 to 6+1 (the upper range limit of 6 is excluded, so +1 has
# been added to it to include 6 in the range). The incremental value
# for the loop is 2 (number+2). The print() function will output the
# resulting value of "number" multiplied by 3.


for number in range(1,6+1,2):
    print(number*3)


# The loop should print 3, 9, 15
```

Common pitfalls when using the range() function:

- Forgetting that **the upper limit of a range() isn't included** in the range.
- **Iterating over non-sequences.** For example, strings are iterable letter by letter, but not word by word.

```
# This loop iterates on the value of the "number" variable in a range
# of 2 to 7 (the upper range limit of 8 is excluded). The print()
# function will output the resulting value of "number" squared.


for number in range(2,8):
    print(number**2)


# The loop should print 4, 9, 16, 25, 36, 49
```

Nested for Loops

The syntax of nested **for** loops:

```
for x in sequence:
    # start of the outer loop body
    for y in sequence:
        # start of the inner loop body

        # end of of the inner loop body
    # continue body of the outer loop
    # end of the outer loop body
```

Example of nested **for** loops:

```
# This code demonstrates the outer and inner loop iterations of a pair
# of nested for loops. Click "Run" to see the results. The outer loop
# will run twice for the range pointer positions [0, 1] in range(2).
# The inner loop will run 4 times for the range pointer positions
# [0, 1, 2, 3] in range(3+1) or range(4) each time the outer loop runs.
# So, the inner loop will execute 8 times in total.


for x in range(2):
    print("This is the outer loop iteration number " + str(x))
    for y in range(3+1):
        print("Inner loop iteration number " + str(y))
    print("Exit inner loop")
```

for Loop with nested if Statement

The syntax of a **for** Loop with nested **if** Statement:

```
for x in sequence:
    # start of body of for loop
    if condition is true:
        # start of body of if-statement

        # end of body of if-statement
    # continue body of for loop
    # end of body of for loop
```

Example of a **for** Loop with Nested **if** Statement:

```
# This for loop iterates through the numbers 0 to 6. The if statement
# uses the modulo operator to test if the "x" variable is divisible by
# 2. If True, the if statement will print the value of "x" and exit
# back into the for loop for the next iteration of "x". Since no
# incremental value is specified in the range() parameters, the default
# increment is +1.


for x in range(7):
    if x % 2 == 0:
        print(x)
```

```
# The loop should print 0, 2, 4, 6
```

Python practice information

For additional Python practice, the following links will take you to several popular online interpreters and codepads:

- Welcome to Python
- Online Python Interpreter
- Create a new Repl
- Online Python-3 Compiler (Interpreter)
- Compile Python 3 Online
- Your Python Trinket

Additional Recursion Sources

Additional Recursion Sources

In the past videos, we visited the basic concepts of recursive functions.

A recursive function must include a recursive case and base case. The recursive case calls the function again, with a different value. The base case returns a value without calling the same function.

A recursive function will usually have this structure:

```
def recursive_function(parameters):
    if base_case_condition(parameters):
        return base_case_value
    recursive_function(modified_parameters)
```

For more information on recursion, check out these resources:

- Wikipedia Recursion page
- See what happens when you Search Google for Recursion

Study Guide: Module 3 Graded Quiz

It is time to prepare for the Module 3 Graded Quiz. Please review the following items from this module before beginning the Module 3 Graded Quiz. If you would like to refresh your memory on these materials, please also revisit the **while** Loop Study Guide and the **for** Loop Study Guide located before the Practice Quizzes in Module 3. You will not be tested on the Recursion lesson content, which is optional in this module.

Knowledge

Terms

- **variables** - Know how to properly initialize or increment a variable. You will also need to recognize a coding error due to the failure to properly initialize or increment a variable.
- **infinite loops** - Know how to recognize infinite loops and use common solutions to prevent them. For example, check loop conditions, ranges, iterators, control statements, etc. to ensure that at least one of these controls are in place to prevent an infinite loop.

- **iterators** - Know the various options available for iterating a variable (e.g., using assignment operators, using the third **range()** function parameter). You will also need to analyze where the iteration should occur. A misplaced iterator could produce the wrong output or create an infinite loop.
- **control statements** - Know how and when to use the **break** and **continue** control statements to prevent infinite loops.

Common Functions

- **range() Function Parameters** - Know the roles of the three possible **range(x, y, z)** function parameters:
  - **x** Start of Range (included)
  - **y** End of Range (excluded index)
    - To include the end of range index, use the expression **y+1**
    - The end of range must be included in the **range()** parameters.
  - **z** Incremental value
  - **Example 1:** range(4, 12**+1**, **2**)
    - This example creates a range that starts at 4 and ends at 12 (without the **+1**, the range would end at 11).
    - The third parameter increments the range iteration by 2, as opposed to the default increment of 1. The **range(4, 12+1, 2)** expression would produce the values: 4, 6, 8, 10, 12
  - **Example 2:** range(10, 2**-1**, **-2**)
    - This example creates a range that starts at 10 and ends at 2**-1**, with a decremental value of **-2**. When counting down, to include the value of the end of the range index, use **-1** (end of range minus 1). This range produces the sequence: 10, 8, 6, 4, 2

- **print() Function Default Behavior** - Know the default behavior of the **print()** function is to insert a new line character after the print statement runs.
  - To override the insertion of the new line character and replace it with a space, add **end=" "** as the last item in the **print()** parameters. This makes it possible to add the next print output to the same line, separated by a space. You might use this technique when a print() function is part of a **for** or **while** loop. Example syntax: **print(x+1, end=" ")**

Coding Skills

Skill 1: Using **for** loops with the **range()** function

- Use a **for** loop with the **range()** function with the end-of-range value included in the range.

```
# This function will accept an integer variable "end" and count by 10
# from 0 to the "end" value.
def count_by_10(end):
    # Initializeq the "count" variable as a string.
    count = ""

    # The range function parameters instruct Python to start the count
    # at 0 and stop at the variable given as the upper end of the range.
    # Since the value of the high end of a range is excluded by default,
    # you can make Python include the "end" value by adding +1 to it.
    # The third parameter tells Python to increment the count by 10.
    for number in range(0,end+1,10):

        # Although the variable "count" will hold a count of integers,
        # this example will be converted to a string using "str(number)"
        # in order to display the incremental count from 0 to the "end"
        # value on the same line with a space " " separating each
        # number.
        count += str(number) + " "

    # The .strip() method will trim the final space " " from the end of
    # the string "count"
    return count.strip()


# Call the function with 1 integer parameter.
print(count_by_10(100))
# Should print 0 10 20 30 40 50 60 70 80 90 100
```

- Use a set of nested **for** loops with the **range()** function to create a matrix of numbers.
- Include the upper range value in the **range()** function using end+1.

```
# This function uses a set of nested for loops with the range() function
# to create a matrix of numbers. The upper range value in the range()
# function should be included in the matrix. The matrix should consist
# of a set of numbers that fill both rows and columns.
def matrix(initial_number, end_of_first_row):


    # It is an optional code style to assign the long variable names in the
    # function parameters to shorter variable names.
    n1 = initial_number
    n2 = end_of_first_row+1  # include the upper range value with +1

    # The first for loop will create the columns.
    for column in range(n1, n2):

        # The nested for loop will create the rows.
        for row in range(n1, n2):

            # To make the matrix of numbers easier to read, include a space
            # between each number in the rows until the loop reaches the
            # end of the row. You can override the default behavior of the
            # print() function (which inserts a new line character after
            # the print command runs) by using the "end=" "" parameter
            # inside the print() function.
              print(column*row, end=" ")

        # The row ends when the upper range value is encountered within the
        # nested for loop. The outer (column) for loop should insert a new line
        # to create the next row. Use the print() function new line default
        # behavior with an empty print() function:
        print()


# Call the function with 2 integer parameters.
matrix(1, 4)
```

```
# Should print:
# 1 2 3 4
# 2 4 6 8
# 3 6 9 12
# 4 8 12 16
```

- Predict the final value of a nested **for** loop with **range()** functions.

```
# For this example, the outer for loop uses an end of range index of
# 10. The value of index 10 will be 10-1, or 9.
for outer_loop in range(10):

    # Using the "outer_loop" variable as the end of range for the
    # inner loop, means the end of range index will be 9. The value
    # of index 9 will be 9-1, or 8.
    for inner_loop in range(outer_loop):

        # The printed result is the value of "inner_loop". Since
        # there aren’t any calculations in this loop, there is a
        # simple shortcut for solving what the final value printed
        # by the "inner_loop" will be. The solution is to simply use
        # the value of the "inner_loop" index, which is 8.
        print(inner_loop)
```

- Find and fix an error in a **for** loop with **range()** function.

```
# This function should count down by -2 from 11 to 1, so that it only
# prints odd numbers.

# This range(11, -2) tells the for loop to start at 11 and end at index
# position -2 (which corresponds to the numeric value of -1). Since the
# third incremental or decremental value is missing, the loop will
# increment by the default of +1 instead of the intended -2 decrement.
# Starting at index position 11 and incrementing by +1 will end the loop
# automatically, because the index is not counting down towards -2 as
# the end of the range.

# To fix this problem, the range() needs three parameters:
# First parameter should be the starting index position of 11.
# Second parameter should be the ending index position of 0 (value 1).
# Third parameter should be decrementing by -2.
# So, the range should be configured as range(11, 0, -2).

# Fix this loop with the corrected range parameters and click Run.
for n in range(11, -2):
    if n % 2 != 0:
        print(n, end=" ")

# Should print: 11, 9, 7, 5, 3, 1 once the problem is fixed.
```

Skill 2: Using while loops

- Use a **while** loop to print a sequence of numbers .

```
# For this example, the while loop will count down by threes starting
# from 18 and ending at 0.
starting_number = 18

# The while loop will continue to loop until it reaches 0.
while starting_number >= 0:

    # To make the sequence of numbers easier to read, include a space
    # between each number in the sequence. You can override the default
    # behavior of the print() function by using the "end=" parameter with
    # the print() function. The syntax for adding a space is: end=" "
    print(starting_number, end=" ")

    # Decrement the "starting_number" variable by -3.
    starting_number -= 3

# Should print 18 15 12 9 6 3 0
```

- Use a **while** loop to count the number of digits in a numerical value

```
# This function accepts a CEO's salary as a variable.
# It counts the number of digits in the salary and
# returns the sentence like:
# "The CEO has a 6-figure salary."
def X_figure(salary):

    # Initializes the counter as an integer.
    tally = 0

    # The if-statement checks if the variable "salary"
    # is equal to 0.
    if salary == 0:
        # If true, then it increments the counter to
        # show there is 1 digit in 0.
        tally += 1

    # The while loop starts to run while the "salary"
    # is greater than or equal to 1 (the loop will
    # not run if the "salary" is 0).
    while salary >= 1:

        # The body of the while loop counts the digits
        # in "salary" by counting the number of times
        # "salary" can be divided by 10 until "salary"
        # is no longer >= 1.
        salary = salary/10

        # Add 1 to the counter to tally the number of
        # times the loop runs.
        tally += 1

    # Return the results of the "tally" of the number
    # of digits in "salary".
    return tally

# Call the X_figure function with 1 parameter, converted to a string,
# inside a print function with additional strings.
print("The CEO has a " + str(X_figure(2300000)) + "-figure salary.")

# Should print"The CEO has a 7-figure salary."
```

Skill 3: Using while loops with if-else statements

- Use a function to accept two variable integers.
- Use nested **if-else** statements and **while** loops to count up or count down from the first variable to the second variable.

```
                # divider between numbers. The if-statement
                # above (if floor > exit) prevents the pipe
                # character from appearing after the "floor"
                # number is no longer greater than the "exit"

                # A pipe | character is added between each
                # floor number in the string variable
                # "elevator_direction" to provide a visual
            # If the "floor" number is still greater than
            # the exit floor number:
            if floor > exit:
            # "elevator_direction".
            elevator_direction += str(floor)

        # equal to the exit floor number:
        while floor >= exit:
            # The "floor" number is converted to a string
            # and is appended to the string variable
        elevator_direction = "Going down: "

        # While the "floor" number is greater than or

        # Then the "elevator_direction" string will be
        # initialized with the string "Going down: ".

    # If the passenger enters the elevator on a floor that
    # is higher than the destination floor:
    if enter > exit:
    # "floor" numbers.
    floor = enter
    elevator_direction = ""
    # print the floor numbers. The "elevator_direction"
    # variable will hold the string "Going up: " or
    # "Going down: " plus the count up or down of the
# number the passenger is going to "exit". Then, the function
# counts up or down from the two floor numbers.
def elevator_floor(enter, exit):
    # The "floor" variable will be used as a counter and to
# This function will accept two integer variables: the floor
# number that a passenger "enter"s an elevator and the floor
```

Reminder: Correct syntax is critical

Using precise syntax is critical when writing code in any programming language, including Python. Even a small typo can cause a syntax error and the automated Python-coded quiz grader will mark your code as incorrect. This reflects real life coding errors in the sense that a single error in spelling, case, punctuation, etc. can cause your code to fail. Coding problems caused by imprecise syntax will always be an issue whether you are learning a programming language or you are using programming skills on the job. So, it is critical to start the habit of being precise in your code now.

No credit will be given if there are any coding errors on the automated graded quizzes - including minor errors. Fortunately, you have 3 optional retake opportunities on the graded quizzes in this course. Additionally, you have unlimited retakes on practice quizzes and can review the videos and readings as many times as you need to master the concepts in this course.

Now, before starting the graded quiz, please review this list of common syntax errors coders make when writing code.

Common syntax errors:

- Misspellings
- Incorrect indentations
- Missing or incorrect key characters:
    - Parenthetical types - (curved), [square], { curly }
    - Quote types - "straight-double" or 'straight-single', “curly-double” or ‘curly-single’
    - Block introduction characters, like colons - :
- Data type mismatches
- Missing, incorrectly used, or misplaced Python reserved words
- Using the wrong case (uppercase/lowercase) - Python is a case-sensitive language

Resources

For additional Python practice, the following links will take you to several popular online interpreters and codepads:

- Welcome to Python
- Online Python Interpreter
- Create a new Repl
- Online Python-3 Compiler (Interpreter)
- Compile Python 3 Online
- Your Python Trinket