Study Guide: Week 3 Graded Quiz

It is time to prepare for the Week 3 Graded Quiz. Please review the following items from this module before beginning the Week 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 Week 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=" ")**

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

```

Run

Reset

- 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.

```
# 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

```

Run

Reset

```
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.

```python
# 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)

```

Run

Reset

```
0
0
1
0
1
2
0
1
2
3
0
1
2
3
4
0
1
2
3
4
5
0
1
2
3
4
5
6
0
1
2
3
4
5
6
7
0
1
2
3
4
5
6
7
8
```

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

```python
# 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, 0, -2):
    if n % 2 != 0:
        print(n, end=" ")

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

Run

Reset

```
11 9 7 5 3 1
```

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
```

Run

Reset

```
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."
```

Run

Reset

```
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.

```
# This function will accept two integer variables: the floor
# number that a passenger "enter"s an elevator and the floor
# 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
    # 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
    # "floor" numbers.
    floor = enter
    elevator_direction = ""

    # If the passenger enters the elevator on a floor that
    # is higher than the destination floor:
    if enter > exit:

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

        # While the "floor" number is greater than or
        # 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".
            elevator_direction += str(floor)

            # If the "floor" number is still greater than
            # the exit floor number:
            if floor > exit:

                # A pipe | character is added between each
                # floor number in the string variable
                # "elevator_direction" to provide a visual
                # 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"
```

Run

Reset

```
Going up: 1 | 2 | 3 | 4
Going down: 6 | 5 | 4 | 3 | 2
```

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