CSc 110 Assignment 4:
Count-Driven Loops

Reminder: Your code is to be designed and written by only you and not to be shared with anyone else. See the Course Outline for details explaining the policies on Academic Integrity. Submissions that violate the Academic Integrity policy will be forwarded directly to the Computer Science Academic Integrity Committee.

All materials provided to you for this work are copyrighted, these and all solutions you create for this work cannot be shared in any form (digital, printed or otherwise). Any violations of this will be investigated and reported to Academic Integrity.

Learning Outcomes:

When you have completed this assignment, you should understand:

- How to define and call functions that use count-driven loops.

Getting started

1. Create a new file called `assignment4.py` and open it in your Wing editor
2. Design the functions according to the specifications in the Function Specifications section.

Submission

1. Double check your file before submission to avoid a **zero grade** for your submission for issues described in the Grading section of this document:
    1. Open and **run `assignment4.py`** in your Wing editor using the green arrow.
       You should see no errors and no output after the shell prompt.
       If there are errors, you must fix them before submitting.
       If there is output printed to the shell, find and remove any top-level print statements and/or top level function calls.
    2. At the shell prompt in Wing (>>>) type the following: **`import assignment4`**
       You should see no errors and no output after the shell prompt.
       If there are errors, you must fix them before submitting.
       If there is output printed to the shell, find and remove any top-level print statements and/or top level function calls.
    3. At the shell prompt in Wing (>>>), make calls to the required functions to ensure you have named them correctly. Ensure the function is exhibiting the expected behaviour when called and does not contain any unexpected output. You should be able to make calls to the following functions with expected input. Note: When making these calls, you must precede it with the module name ie. `assignment4.is_proper_divisor(2, 10)`
        i. is_proper_divisor
        ii. sum_of_proper_divisors
        iii. get_abundance
        iv. get_multiples
        v. print_multiplication_table
2. Upload your **`assignment4.py`** containing the completed function designs to BrightSpace

Grading:

- Late submissions will be given a **zero grade**.
- The files you submit must be named **`assignment4.py`**
  The filenames must be EXACT for them to work with our grading scripts. Errors in the filenames will result in a **zero grade**.
  Example mistakes often made by students include but are not limited to: spelling errors, different case letters, space characters and incorrect extension (not using .py)
- Your function names and the order of your function arguments must match EXACTLY as specified in this document or you will be given a **zero grade**.
- Your submission must not contain any print statements that are not required in the specification or any top-level calls to functions. This unexpected code can cause the automated tester to crash and will result in a **zero grade**.
- We will do **spot-check grading** in this course. That is, all submissions are graded BUT only a subset of your code might be graded. You will not know which portions of the code will be graded, so all of your code must be complete and adhere to specifications to receive marks.
- Your code must run without errors with Python 3.9. If you tested your configuration with setup.py file this would have verified you are using Python 3.9. Code that generates errors cannot be tested and will be given a **zero grade**.

Marks will be awarded for correctness, considering:

- the function signature matches the description given (has the name and arguments EXACTLY as specified)
- the function has the expected behaviour

and for code quality according to software engineering properties such as:

- documentation in docstring format: type hints, purpose, examples
- Test coverage – examples within the function docstring cover all boundary cases of all conditions within your function
- readability
  - use of whitespace
  - splitting complex computation or long statements across multiple lines
- meaningful variable names
  - lower case, starting with an alpha-character
- proper use of constants (avoid magic numbers)
  - defined above all function definitions
  - in UPPERCASE
- use of code constructs (functions, variables, conditions) to:
  - eliminate redundant code and redundant computation
  - make complex code easier to read

Function Specifications

Examples provided are for explanation purposes. You can/should create your own example data for additional testcases to demonstrate full test coverage.

1. Design a function called `is_proper_divisor` that takes two non-negative integer values n1 and n2. The function should determine whether n1 is a proper divisor of n2.
   What is the best type to return for a function like this? Make sure to make the BEST choice here.
   Recall the definition of a proper divisor, also referred to as a factor from math:
   https://www.mathsisfun.com/numbers/factors-all-tool.html

   NOTE: The following definition and clarifications are taken from the following publication:
   http://mfleck.cs.illinois.edu/building-blocks/version-1.0/number-theory.pdf

   - Definition:
     Suppose that *a* and *b* are integers. Then *a* divides *b* if *b* = *an* for some integer *n* then *a* is called a factor or divisor of *b*
   - Any non-zero whole number times 0 equals 0 so it is true that every non zero number is a factor of 0
   - because 0 * n is always zero, zero is not a factor of any non-zero value of n.
   - Zero is a factor of only one number: zero.

2. Design a function called `sum_of_proper_divisors` that takes a non-negative integer. The function should return the sum of the all the proper divisors of that number excluding itself.
   For example, `sum_of_proper_divisors(12)` should return 16
   since the proper divisors of 12, not including 12 are: 1, 2, 3, 4 and 6 which sum to 16.
   Be sure to make use of relevant helper functions!

3. Design a function called `get_abundance` that takes a non-negative integer. The function should return the abundance of that number. If the number passed to the function is not an abundant number, the function should return 0, otherwise the function should return the abundance of that number as described below.
   "In number theory, an abundant number or excessive number is a number that is smaller than the sum of its proper divisors. The integer 12 is the first abundant number. Its proper divisors are 1, 2, 3, 4 and 6 for a total of 16. The amount by which the sum exceeds the number is the abundance. The number 12 has an abundance of 4, for example."
   Taken from: https://en.wikipedia.org/wiki/Abundant_number
   Be sure to make use of relevant helper functions!

4. Design a function called `get_multiples` that returns a string containing a sequence of multiples separated by at least one space. The function takes the following three arguments in this order:
   - the number the sequence will begin with, assumed to be a non-negative integer that is a multiple of the second argument
   - the number representing the multiple, assumed to be a non-negative integer
   - the number of values that should be in the sequence returned as a string

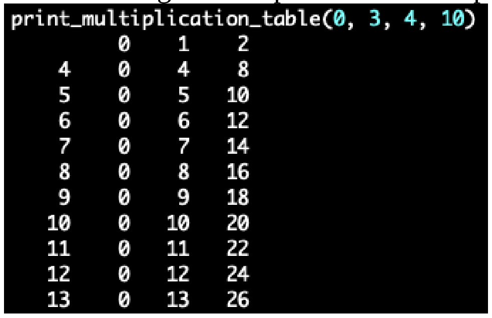
   The call `get_multiples(8, 2, 7)` should return the string: **`'8 10 12 14 16 18 20'`**
   since the sequence is to contain 7 numbers starting at 8, and subsequent numbers increasing by 2
   The call `get_multiples(9, 3, 6)` should return the string: **`'9 12 15 18 21 24'`**
   since the sequence is to contain 6 numbers starting at 9, and subsequent numbers increasing by 3

   Note you will need to convert the integers to strings using in order to add them to your result string.
   There must be at least one **space** between each pair of values in the string (no commas).

5. Design a function called `print_multiplication_table` that prints a multiplication table to the screen. The function takes the following four arguments in this order:
   - the number to start the horizonal axis at, assumed to be a non-negative number
   - the width of the multiplication table, assumed to be greater than 0
   - the number to start the vertical axis at, assumed to be a non-negative number
   - the height of the multiplication table, assumed to be greater than 0

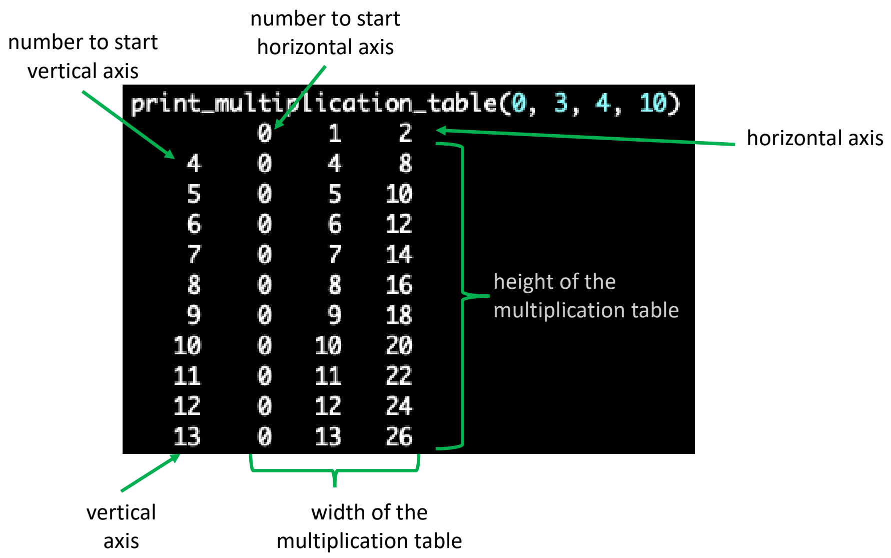
   Your implementation should call the `get_multiples` function as a helper.

   The following is a sample call with sample output:



```
print_multiplication_table(0, 3, 4, 10)
          0    1    2
     4    0    4    8
     5    0    5   10
     6    0    6   12
     7    0    7   14
     8    0    8   16
     9    0    9   18
    10    0   10   20
    11    0   11   22
    12    0   12   24
    13    0   13   26
```

   The following is the same example call with the image annotated to clarify the role of each of the arguments as described in the specification above.


number to start vertical axis
number to start horizontal axis
horizontal axis
height of the multiplication table
vertical axis
width of the multiplication table

```
print_multiplication_table(0, 3, 4, 10)
          0    1    2
     4    0    4    8
     5    0    5   10
     6    0    6   12
     7    0    7   14
     8    0    8   16
     9    0    9   18
    10    0   10   20
    11    0   11   22
    12    0   12   24
    13    0   13   26
```

   You will notice the output is nicely formatted in the example with the columns lined up. This is not a requirement of the assignment BUT the following must be done in order to pass the automated tester:
   - the output should be single spaced
   - each row of the table must be on its own line
   - each value printed must have at least one space after it before the next value is printed

   If you want to experiment with formatting on your own (this is not required)…
   - try out the integer width format specifier used by f-strings:

```
x = 22
print(f'the number is{x:5d}') # will print x in field of width 5
```

   - try out the `format` function that takes number and a format specifier and returns a string with the value in a string of the specified width:

```
x_as_a_string = format(x, '5d') # x_as_a_string will be: '   22'
```