KIT103/KMA155 Programming Assignment 3: Number Theory 1

Enter your answers to these questions in the accompanying Python script file `kit103_assign3.py`. As in the previous assignment, your answers will be function definitions and the script file already contains function 'stubs' that you will complete. Include your name and student ID in the places indicated in the header comment.

Submit your completed script file to the *Programming Assignment 3 (Number Theory 1)* assignment folder on MyLO by **1500 (3pm) Wednesday 20 September 2017**.

> Test your solutions thoroughly. Your submission is expected to run without failure (even if it doesn't produce the correct answer for each question). If we have to correct your submission in order for it to run then the maximum total mark you can receive will be 3/5 (1.5/2.5 if in KMA155).

KMA155 students will be assessed primarily on Questions 1 (for $n \in \{2, 3, 4\}$) and Question 3, which are marked with an asterisk (*), but may attempt all questions if they wish.

Question 1: Divisibility of really, really big integers (2 marks, *)

In many programming languages the native integer data types have an upper limit on their maximum value. To test the divisibility of numbers larger than can be represented natively it can be convenient to instead work with strings of digits, as character strings can grow indefinitely.

Task: There are incomplete functions in the assignment script file that are intended to test large numbers (defined by strings of the characters '0' through '9') for divisibility by various values. Complete the implementations of these functions by *applying the common divisibility rules* for 2, 3, 4, and 11 (the functions are named `divisible_by_`$n \in \{2, 3, 4, 11\}$). KMA155 students must implement the rules for divisibility by 2, 3 and 4.

> **Hints:**
> - Your functions will be tested with strings, like '1234', not integers such as 1234. These are not equivalent: the first occupies more than 64 bits in memory (4 × 2-byte characters), while the small integer value is likely only 32 bits.
> - Practical 6 (week 7) contains some similar examples and describes how to access individual characters in a string by index (which may count backwards from the end of the string). String *slicing* may also be useful in up to two of the above functions.
> - You might not need to convert any part of the string to a number (but if you've applied the common divisibility rules and have reached the point where that makes sense then it's fine for you to do so).

Question 2: GCD from a prime factorisation (2 marks)

Both the math and programming lecture streams have described how to calculate *gcd*(*a*, *b*) using the prime factorisations of *a* and *b*. Your task is to complete the stub function `q2_factor_gcd(a, b)` to implement this behaviour. In your solution you should use the Python `Counter` class to represent a multiset/bag, and make use of the included helper function `factor_list` to generate a list of prime factors for each input.

Question 3: Are a and b coprime (i.e., relatively prime)? (1 mark, *)

In the assignment script file there is a stub function `q3_coprime(a, b)` that is intended to return `True` if `a` and `b` are relatively prime, `False` otherwise (coprime is another term for relatively prime, and provides a shorter function name).

Based on your knowledge, the math lecture notes and programming lab exercises, implement q3_coprime. You may, in fact probably should, implement at least one additional helper function, which must use the most direct approach to deriving its answer (i.e., you may not reuse the answer to any other question in this assignment). With that helper function available the body of q3_coprime could be as short as one line of code.

Information on string slicing

The following information on string slicing (essentially, taking a substring from a string) may be helpful in some parts of Question 1. For a string `s`:

```
s[start:end] # characters start through end-1
s[start:]    # characters start through the rest of the string
s[:end]      # characters from the beginning through end-1
s[:]         # a copy of the whole string
```

And you may optionally include a step value:

```
s[start:end:step] # start through not past end, by step
```

So, for example, `s[2::3]` copies every third character in `s` starting with the third character (i.e., position 2) up to the end (because the end position was omitted).
