KIT103/KMA155 Practical 06: Divisibility and finding prime numbers

This week we'll apply some common divisibility rules to check the divisibility of *really* big numbers stored as strings and, if you have time, try to find some prime numbers.

Preparation

Start Spyder and save the temp.py file in the editor with a new name as you will save your work in it.

1 Common divisibility rules and huge numbers

If you want to check if a number n is divisible by another number d then the simplest way to express that in a program is `n % d == 0`. That is, that the remainder from dividing n by d is zero. But what do you do if the number is really large? Python has excellent support for large integers, but many programming languages do not (for instance, the largest integer Java can represent is 18,446,744,073,709,600,000, which may be big, but is certainly not the largest integer imaginable).[1]

Given numbers expressed as strings of the characters '0' through '9' we can use the common divisibility rules (Section 5.2 of the Maths lecture slides and summary notes [from p42 of the notes]) to determine if a really big number is divisible by 2, 3, 4, 5, 7, 9 or 11. We'll do two examples here and some more will be in the assignment.

Task: Define a function `divisible_by_5` that takes a string-valued parameter, examines the last character and returns `True` if that character is '0' or '5'.

Tips:
- For a string `s`, `s[i]` is the character at position 0 ≤ i < `len(s)`. If `i` is negative then it counts back from the end of the string, so `s[-1]` is the last character in s.
- Use `int(s)` to convert a string or character to the integer it represents.

Task: Define another function `divisible_by_9` that takes a string representing an integer, calculates its digital sum (the sum of its digits) and returns `True` if that is divisible by 9.

1.1 Generating test cases

Apart from writing down random strings of digits and manually applying the appropriate divisibility rule, how can you generate test cases for your functions above? Because Python *does* support large integers, you can generate some test inputs that are easier to verify prior to using them as test cases:

1. To generate a large number, consider any of these three approaches:

    1. Type random digits and, for each number *n* you create, test if *d*|*n* directly: `n % d == 0`
    2. *or* Type random digits and multiply that by *d* to create a positive test case
    3. *or* Type random digits, multiply by *d*, and add a value between 1 and *d*–1 to create a negative test case
2. Convert *n* to a string: `str(n)`
3. Pass that to your divisible_by_*d*() function

1.2 Optional challenge: Divisibility by 7

Task: Define another function `divisible_by_7` that takes a string representing an integer and reduces it to two digits by applying the following rule repeatedly: remove the last digit, double it, and subtract that value from the number formed from the remaining digits. If the final result, converted to an integer, is divisible by 7 then the function should return `True`.

Now you've done that, or even if you've implemented the suggested solution (either is fine), is this divisibility rule a good one to implement in a program?

2 (if time) Your first prime number generator

Based on your knowledge of the definition of a prime number, complete the following function to generate prime numbers up to a specified value of *n*.

```
def primes(n):
    '''Generates a list of prime numbers up to n'''
    primes = [] #list of primes
    for k in range(2, n+1):
        #Add code to test if k is prime
        #If it is then call the following to add it to the list:
        #primes.append(k)
    return primes
```

2.1 How fast is that?

Add the following import to your source file:

```
import time
```

And then define this function (or some variation of it), which will time the execution of your prime generating function:

```
def time_primes1(n, show_msg=True):
    start = time.clock()
    total = len(primes(n))
    elapsed = time.clock() - start
    if show_msg:
        print('Took {0:g} seconds to produce the {1} primes up to
{2}'.format(elapsed, total, n))
    return elapsed
```

Call `time_primes1` with the value of *n* up to which it should generate primes (using your `primes()` implementation), and optionally a Boolean value (of `False`) if you want to suppress the message about how long it took to run.

Try running it in a loop with values of *n* from [10, 100, 1000, ..., 100000].

---

1. Note that the *largest integer imaginable* is not, as you may have thought as a child, $\infty + 1$↩