ENGG1003 - Lab Week 7

Brenton Schulz

1 Introduction

This lab covers Python functions and a prime number generating algorithm. The intention is that it is completed during your Zoom lab and while waiting to have your assignment graded.

2 Tasks

Task 1: Basic Functions

Write three separate Python functions which perform as follows:

- Function 1 takes no argument and always returns the value `42`
- Function 2 takes a single, `x` argument and returns its square, `x**2`
- Function 3 takes two arguments, `x` and `y` and returns their product `x*y`

Write these functions in the same script and include "test code" below them which calls the functions with synthetic test data (ie: any numbers for which you already know the answer) and prints the results.

You will need to create your own names for the functions. This exercise is purely intended as practice implementing correct function `def` syntax so the exact names are not important. Choose *anything* which gets the job done without error or ambiguity.

Task 2: Intermediate Functions

Modify your Assignment 1 code so that the distance calculation (either the equirectangular approximation or haversine formula) is implemented in a function.

Test if your function is correct with synthetic data and compare your result to that of an online calculator, eg: `https://www.nhc.noaa.gov/gccalc.shtml`

Task 3: Prime Number Generation: Sieve of Eratosthenes

This task is intended as an algorithmically complex task. Start by reading the task then completing an example ***by hand*** *to get a "feel" for the algorithm.*

The *Sieve of Eratosthenes* is an algorithm for generating prime numbers. The method is attributed to the ancient Greek scholar Eratosthenes of Cyrene (276-194 BC).

If implemented by hand, the algorithm requires the following steps:

1. Write a list of integers from 2 to some "large" number, N
2. Observe that 2 is prime and that any multiple of 2 would *not* be prime
3. Moving down the list, cross out every multiple of 2
4. Choosing the next highest number which is not crossed out, 3, move down the list and cross out every multiple of 3
5. Choosing the next highest number which is not crossed out, 5, move down the list and cross out every multiple of 5
6. Repeat the pattern until you no longer cross out any values
7. All numbers which were not crossed out are prime

Example:

Listing integers from 2 to 16:
`2 3 4 5 6 7 8 9 10 11 12 13 14 15 16`

Crossing out every multiple of 2:
2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ 9 ~~10~~ 11 ~~12~~ 13 ~~14~~ 15 ~~16~~

Crossing out every multiple of 3:
2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ ~~9~~ ~~10~~ 11 ~~12~~ 13 ~~14~~ ~~15~~ ~~16~~

Crossing out every multiple of 5 (ok, they're all crossed out by now already so we can stop):
2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ ~~9~~ ~~10~~ 11 ~~12~~ 13 ~~14~~ ~~15~~ ~~16~~

All the numbers left over are prime:
`2 3 5 7 11 13`

Task: Write a Python script which uses this method to find all prime numbers strictly less than some number N. Declare N as a variable near the start of your script, debugging with a small value (eg: 16) then choosing a large value (eg: 1 million) for further testing.

Since we need to remember a "long" list of numbers (and whether or not they are "crossed out") this is a perfect application for an array.

To complete this task, declare an array full of ones with `np.ones()`. This function behaves identically to `np.zeros()` except it fills the array with 1's instead of 0's.

The array should be initialised to all 1's to indicate that, at the start of the algorithm, nothing has been crossed out. As elements are "crossed out" they should be set to zero.

The list of indices will form the list of integers. The array elements (the numbers stored at each index) indicate whether that index has been "crossed out" or not.

The array includes indices 0 and 1 which are integers not considered by the original algorithm. Your code may either ignore them, force the array elements to zero (0 and 1 are *not* prime) or treat them as special cases when printing the result (ie: have an explicit `if index == 0 or index == 1:` statement).

Your program should implement the following pseudocode:

```
BEGIN
  N = 16
  numbers[] is an array with every element initialised to 1

  FOR index = 2, 3, 4, ..., N
    IF numbers[index] is 1
      FOR j = 2*index, 3*index, 4*index, ... N
        numbers[j] = 0
      ENDFOR
    ENDIF
  ENDFOR
  Print every index for which numbers[index] is 1
END
```

Note the complex flow control structure: there is a *nested* `for` loop where the inner loop's execution is conditional (ie: inside an `if` statement).

Also observe that the inner `for` loop loops over `n*index`. Carefully consider how this can be achieved with the `range()` function (or, indeed, if it is even possible). There are many ways to implement this loop but they are all non-trivial and require some thought.

Extension 1: Declare the `numbers` array of type `numpy.byte` to minimise RAM usage. This uses an eighth of the RAM of the `np.float64` default.

Extension 2: The Wikipedia article covering the Sieve of Eratosthenes describes several optimisations. Implement the one which only crosses out `index` values starting at `index**2` (instead of `2*index`) and the algorithm exit condition of not attempting to cross out values larger than $\sqrt{N}$. How much faster does this make the code execute?