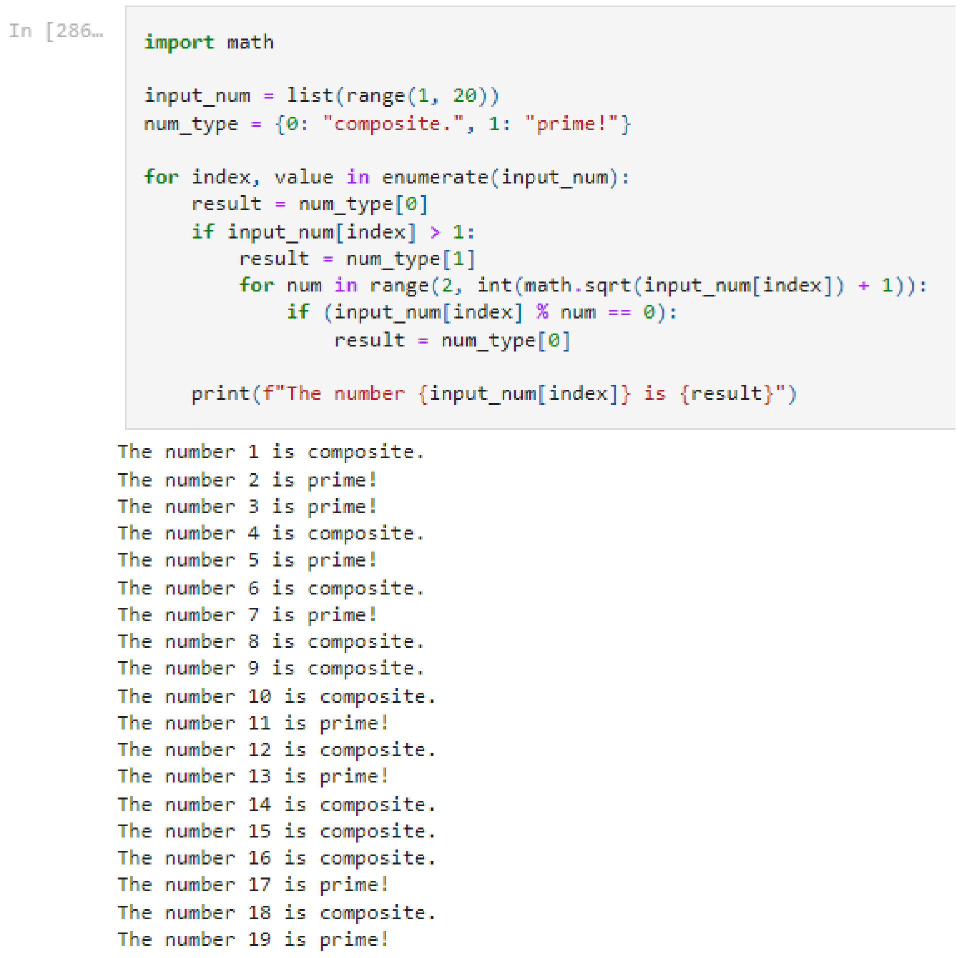
CS521 - Homework 3 - Alexander Bean

1. Prime or Composite?



```
In [286…

import math

input_num = list(range(1, 20))
num_type = {0: "composite.", 1: "prime!"}

for index, value in enumerate(input_num):
    result = num_type[0]
    if input_num[index] > 1:
        result = num_type[1]
        for num in range(2, int(math.sqrt(input_num[index]) + 1)):
            if (input_num[index] % num == 0):
                result = num_type[0]

    print(f"The number {input_num[index]} is {result}")
```

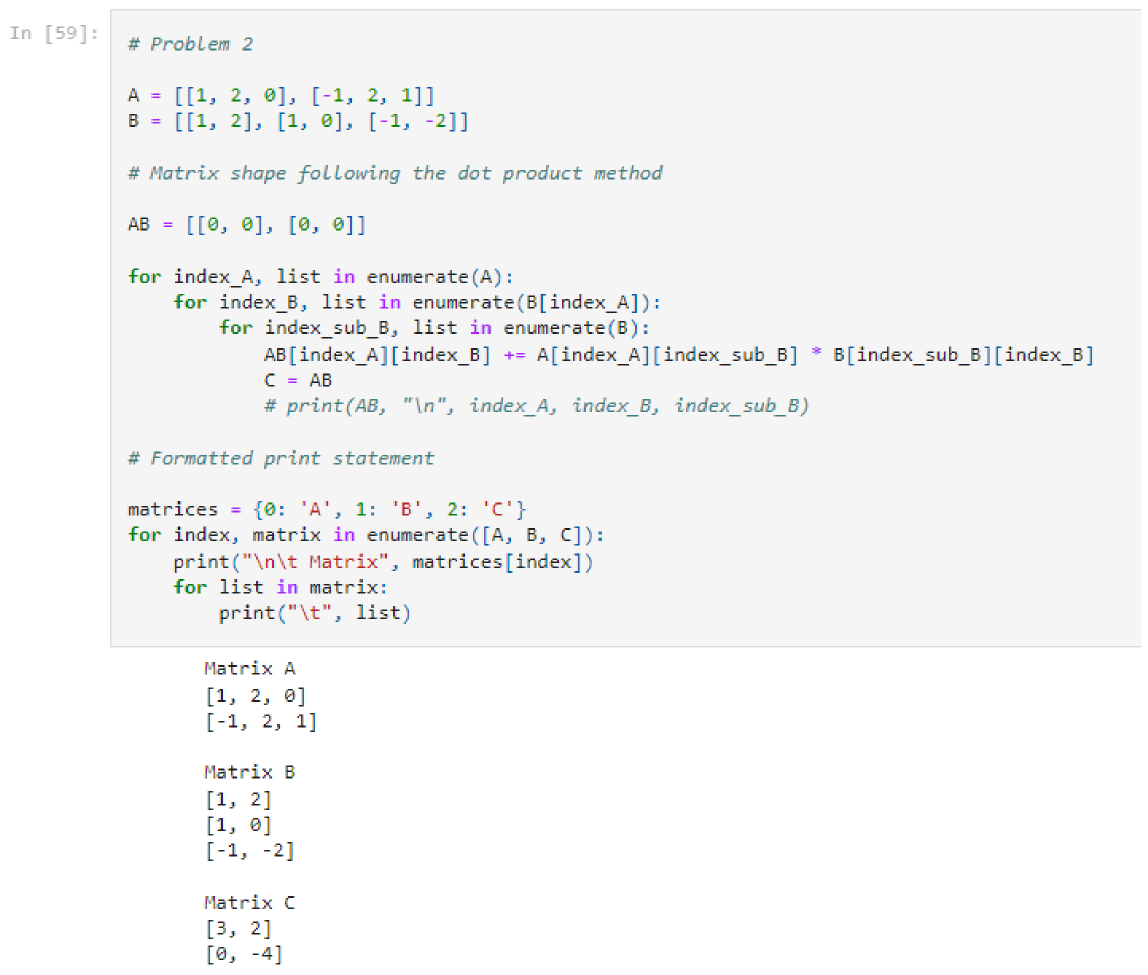
```
The number 1 is composite.
The number 2 is prime!
The number 3 is prime!
The number 4 is composite.
The number 5 is prime!
The number 6 is composite.
The number 7 is prime!
The number 8 is composite.
The number 9 is composite.
The number 10 is composite.
The number 11 is prime!
The number 12 is composite.
The number 13 is prime!
The number 14 is composite.
The number 15 is composite.
The number 16 is composite.
The number 17 is prime!
The number 18 is composite.
The number 19 is prime!
```

I began by listing a range of numbers to use for testing my program's functionality and correctness. I also created a dictionary for composite and prime assignment, which is referenced in the result variable.

I added conditional statements to begin processing the numbers for primality. The result variable begins at composite in case a number less than 1 is entered. Then, following the instructions, I created a statement to run through a range of numbers between 2 and the entered number's root. I converted the root to an integer because the range() method will raise an error if a non-int value is used.

While the question states that the program should not check numbers greater than the entered number's root, I added +1 to it in the condition. Without +1, I found the range would be slightly too low for some numbers to correctly be detected as composite. By manually increasing the range by 1, my program iterates through just enough loops to correctly detect and assign composite to the result.

2. Nested “For” Loops



In [59]:

```
# Problem 2

A = [[1, 2, 0], [-1, 2, 1]]
B = [[1, 2], [1, 0], [-1, -2]]

# Matrix shape following the dot product method

AB = [[0, 0], [0, 0]]

for index_A, list in enumerate(A):
    for index_B, list in enumerate(B[index_A]):
        for index_sub_B, list in enumerate(B):
            AB[index_A][index_B] += A[index_A][index_sub_B] * B[index_sub_B][index_B]
            C = AB
            # print(AB, "\n", index_A, index_B, index_sub_B)

# Formatted print statement

matrices = {0: 'A', 1: 'B', 2: 'C'}
for index, matrix in enumerate([A, B, C]):
    print("\n\t Matrix", matrices[index])
    for list in matrix:
        print("\t", list)
```

```
         Matrix A
         [1, 2, 0]
         [-1, 2, 1]

         Matrix B
         [1, 2]
         [1, 0]
         [-1, -2]

         Matrix C
         [3, 2]
         [0, -4]
```

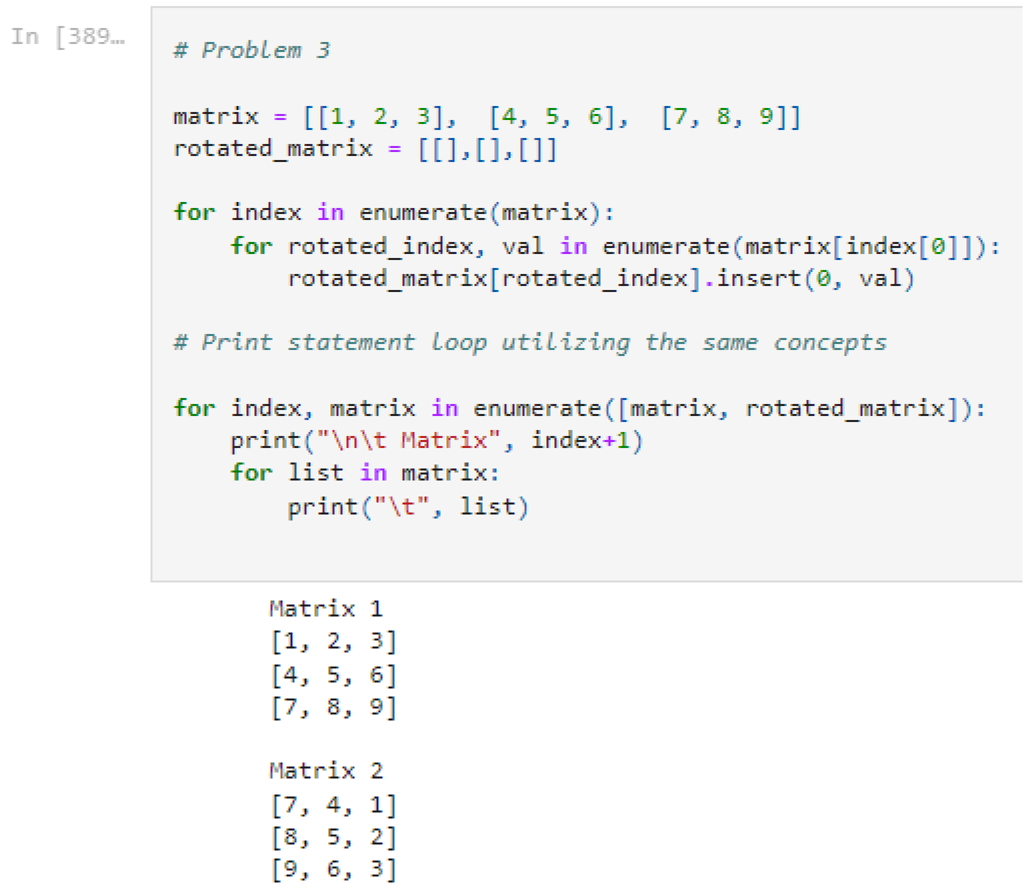
My first step was to determine the frame of the matrix. Following the dot product method, I got a 2x2 grid which I assigned to variable AB. I did this to allow for an easy insertion of numbers during my loops.

To program the formula Ax * Bx + Ay * By: I wrote nested for loops to go into the “rows” in matrix A, into the “columns” lists in matrix B (based on the length of the “rows” in matrix A), and finally into the “rows” in matrix B. This gave me access to the list values to code the dot product method.

For each set of items, it calculates Ax*Bx and adds it to the location in the AB frame. The location is determined by the X of A and Y of B. The loop iterates, then goes to the next index and does the same for Ay+By–or An+Bn rows there are in the “column”. It repeats this for each “row” in A.

I added a print statement at the bottom-most nested loop for visibility into my program. I used it to track the loop iteration and index sorting process. I commented it out so it can be re-added in and referred to.

3. “For” Loops with Enumeration



```
In [389…

# Problem 3

matrix = [[1, 2, 3],  [4, 5, 6],  [7, 8, 9]]
rotated_matrix = [[],[],[]]

for index in enumerate(matrix):
    for rotated_index, val in enumerate(matrix[index[0]]):
        rotated_matrix[rotated_index].insert(0, val)

# Print statement loop utilizing the same concepts

for index, matrix in enumerate([matrix, rotated_matrix]):
    print("\n\t Matrix", index+1)
    for list in matrix:
        print("\t", list)
```

```
         Matrix 1
         [1, 2, 3]
         [4, 5, 6]
         [7, 8, 9]

         Matrix 2
         [7, 4, 1]
         [8, 5, 2]
         [9, 6, 3]
```

My plan was to identify the items in the same “column” and insert them into lists based on those numbers, and will need to insert the values at the front of the list each time to produce the result. I created an empty list in the same format as the matrix list for easy insertion of the new values.

I used enumerate() twice: First, I used it to get an indexing value for the nested lists and their row value. When referring to it, it initially caused issues since enumerate() was producing a tuple. To get around this, I referred to the first index of the tuple value, which had the actual value and data type I needed. Second, in the nested loop, I used enumerate() to determine the “column” numbers of the sublist items.

Then I assigned the sublist items to the rotated_matrix lists based on their column value. I used insert() with position 0 so each new value was inserted at the beginning, which produced the rotating effect.

Then, in my print statement, I incorporated more nested for loops to practice the concepts, and to demonstrate the same concepts shown in the actual problem.

4. Two Digit Squaring Loop

In [10]:

```python
# Problem 4

import math

min = 10
max = int(math.sqrt(1000))

num_list = range(min, max)

print("Where:\nAB * AB = CAB\n---Results---\n")

for num in num_list:
    two_digit = str(num)
    three_digit = str(int(two_digit) ** 2)

    if two_digit[0] == three_digit[1] and two_digit[1] == three_digit[2]:
        print(f"{two_digit} * {two_digit} = {three_digit}")
```

```
Where:
AB * AB = CAB
---Results---

25 * 25 = 625
```

To do this, I defined a range of two-digit numbers for the program to check. However, when doing this, I thought back to Question 1–at some point, there will be two-digit numbers that, when squared, result in a four-digit number and do not need to be checked. So, to identify this, I imported the math module and got the square root of 1000, which was 31.6. I truncated the decimals and used 31 as the maximum value for the range. I assigned these numbers to variables for programming clarity.

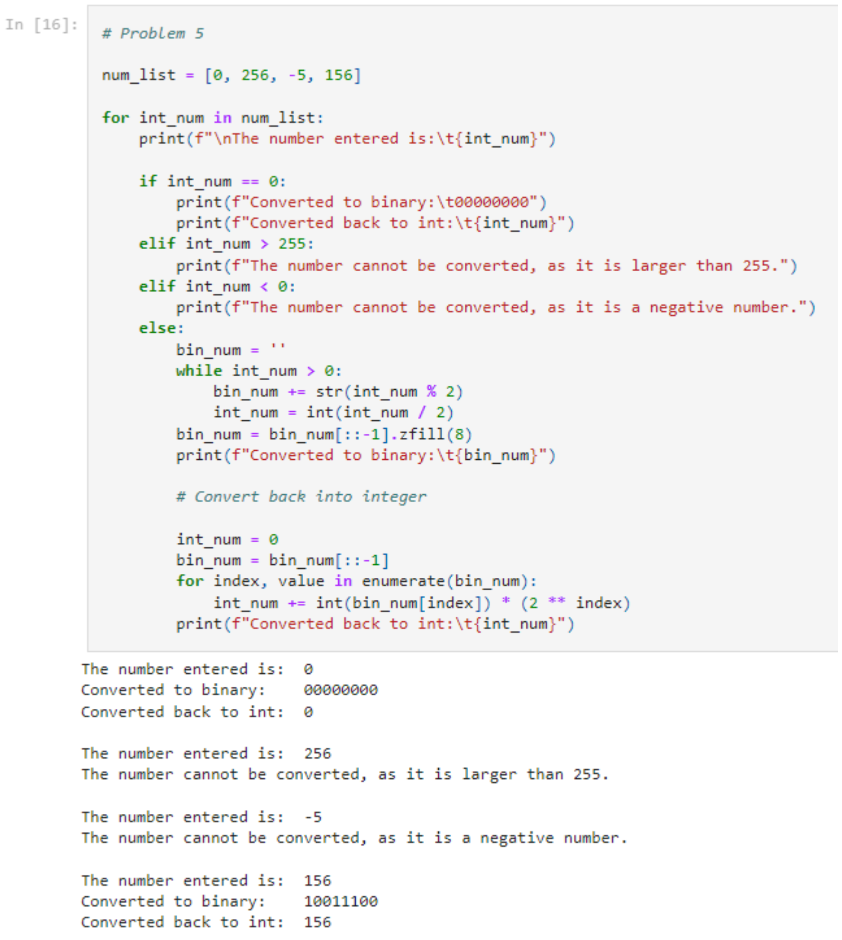
To determine what numbers are valid in the AB * AB = CAB equation, I decided to turn the numbers into strings and use indexing. I referred to the first and second indexes of the two-digit number, and the second and third indexes of the three-digit number. By doing so, I can compare A in AB to A in CAB, and B in AB to B in CAB. If both statements are true and match, it is a valid number and is printed. The loop then continues to cycle through all numbers within the range until the end.

4. Decimal to Binary Conversion

I created a list of numbers to demonstrate each if, elif, and else condition. I also added an additional elif statement for when the entered number is greater than 255 as that is the max for 8-bit binary strings.

I created bin_num as an empty string to assign the binary number to. Since the remainder of int_num will be a 1 or 0, I would take it and append it to the string.

Then I divided int_num by 2 following the format for binary number calculations, and the while loop iterates until int_num is less than 1.



```
In [16]:
# Problem 5

num_list = [0, 256, -5, 156]

for int_num in num_list:
    print(f"\nThe number entered is:\t{int_num}")

    if int_num == 0:
        print(f"Converted to binary:\t00000000")
        print(f"Converted back to int:\t{int_num}")
    elif int_num > 255:
        print(f"The number cannot be converted, as it is larger than 255.")
    elif int_num < 0:
        print(f"The number cannot be converted, as it is a negative number.")
    else:
        bin_num = ''
        while int_num > 0:
            bin_num += str(int_num % 2)
            int_num = int(int_num / 2)
        bin_num = bin_num[::-1].zfill(8)
        print(f"Converted to binary:\t{bin_num}")

        # Convert back into integer

        int_num = 0
        bin_num = bin_num[::-1]
        for index, value in enumerate(bin_num):
            int_num += int(bin_num[index]) * (2 ** index)
        print(f"Converted back to int:\t{int_num}")
```

```
The number entered is:  0
Converted to binary:    00000000
Converted back to int:  0

The number entered is:  256
The number cannot be converted, as it is larger than 255.

The number entered is:  -5
The number cannot be converted, as it is a negative number.

The number entered is:  156
Converted to binary:    10011100
Converted back to int:  156
```

Once the loop ends and I have the binary number, I use two processes to correctly flip/reverse it. In my testing, I found that if a user enters a smaller number (like 24), you will not get a full 8 digit result because it is smaller than 128, 64, and 32 and those binary places will not be filled with a 0. To solve this, I found the zfill() method, which adds leading zeros up to N number of spaces. So, since a binary number is 8 bits/spaces, I allocated 8 spaces to add leading zeroes if necessary. Then, I used indexing to step backwards through the string and produce the flipped result.

To convert it back into an integer, I reversed the indexing/step process to return bin_num to the original space. Then, I used enumerate() to get index numbers for each character in the string. Since binary numbers are exponential with base 2, I can use the assigned index as the exponent. Taking the value of bin_num in that index (either 1 or 0), I can multiply it by base 2 raised to the power of the index number (0-7). With that, I am able to correctly calculate the binary number back into the entered integer.