

Bethan Moyle Programming

1707952

by Bethan Moyle

Submission date: 29-Apr-2025 12:25PM (UTC+0100)

Submission ID: 257099815

File name: 57913_Bethan_Moyle_Bethan_Moyle_Programming_1707952_653609_1658103090.docx (424.62K)

Word count: 2735

Character count: 13694



Prifysgol Cymru
Y Drindod Dewi Sant
University of Wales
Trinity Saint David

Programming

Introduction to Computer Programming

Module Code: ACCA4020

BSC4F Computing (Computer Networks and Cyber Security) with Foundation Year

Bethan Moyle (1707952)

1707952@student.uwtsd.ac.uk

Table of Contents

Introduction	4
1a. Flowchart Analysis.....	4
1b. Identified Loops.....	5
Primary Loop (Outer Loop).....	5
Secondary Loop (Nested Loop).....	5
2a. Loops and Termination Conditions	6
Impact of early exit.....	6
2b. Boolean Variable.....	6
3 Conditional Statements.....	7
3a. Purpose of Conditional Statements	7
3b. Explanation of Specific Conditional Statements.....	7
If Num1 > 1.....	7
If Num1 % Num2 == 0	7
If Is_Correct == True	7
4a. Program	8
Generate_special_numbers as Plain Text	9
4b Test Cases	10
Start: 3 End:16.....	10
Start: 25, End: 42	10
5. Program Validation	11
a. Test Case 1:.....	11
b. Test Case 2:.....	11
c. Test Case 3:	12
d. Test Case 4:.....	12
e. Test Case 5:.....	13
6 Optimisation Techniques.....	13
7 Reflection on Learning Journey.....	14
Challenges	14
Solutions	14
Application of Concepts	14
8 Conclusion	14

9 References.....	15
-------------------	----

Introduction

This assignment explores a flowchart that outlines an algorithm for identifying prime numbers within a specified range. The analysis focuses on how loops interact with one another, the role of conditional statements, and the implementation of early exits. A Python program will be developed based on the logic and structure of the flowchart, including tests to validate its functionality. A reflection on the learning journey throughout the module is reviewed, highlighting key challenges encountered and proposed strategies for overcoming them.

1a. Flowchart Analysis

The purpose of the flow chart is to identify and collect all prime numbers within a set range defined by the user. The program begins by taking two inputs: a starting number (given to the variable **start**) and an ending number (given to the variable **end**). It then checks each number in that range to determine if it is prime. The flowchart includes two nested loops: the outer loop iterates through each number from start to end, and the inner loop checks if the current number has any divisors other than 1 or itself.

The flowchart uses several conditional checks implemented through if statements. The statements control the flow of the program and ensure correct prime numbers are identified. The first loops if conditions are to see if the number has a possibility of being prime before allowing it through to the second loop. If the number passes this initial filter, the program proceeds to check the divisibility of the number, by introducing new variables such as **Num2**. If the number is determined as prime it is added to the **Value[]** list and the program will move onto the next number until the value of **Num1** is larger than the value of **end+1**, at which point the program will then exit the loop and display the identified list of prime numbers.

However, a flaw in the flowchart's algorithm is that it fails to identify 2 as a prime number. When **Num1** = 2 and **Num2** = 2, the condition **Num1 > sqrt(Num1) + 1** evaluates as true, causing the algorithm to incorrectly mark 2 as non-prime. This is problematic because 2 is the only even prime number [1], divisible only by 1 and itself. To fix this, a special condition should be added to explicitly check if **Num1** equals 2 and handle it separately before applying the general logic.

1b. Identified Loops

The flowchart contains two nested loops, with one loop operating within the other. The outer loop runs first, and within it, the inner loop is executed. Although flowcharts don't use a unique symbol for loops, decision diamonds combined with arrows are used to indicate repetition [2]. The arrow will point back to a section of the code that has already been executed, this indicates the presence of a loop [2], where the process will repeat based on the decision made at that point.

Primary Loop (Outer Loop)

The first loop in the flowchart is the outer loop, which runs the program from the range of the **start** variable to the **end** variable specified by the user. It begins by checking whether **Num1** is less than **end+1**. Since **Num1** is initially set to the **start** value and increases by 1 with each iteration, this condition determines when the loop should exit and move on to the next block of code. If **Num1** exceeds the **end** value, the loop exits, the **Values** list is printed, and the program ends. However, if the condition is met, the loop moves on to another if statement that checks whether **Num1** is greater than 1, as value of 1 and below are not considered prime numbers[1]. If **Num1** is not greater than 1, it is incremented by 1, and the loop restarts. Once this condition finds number that meets the requirements, the program continues with the next steps in the prime checking process.

Secondary Loop (Nested Loop)

This loop checks whether the value of **Num2** is less than the square root of **Num1+1**. It uses the square root of the **num1** instead of its whole number for efficiency to check each number up to the number of **Num1**. This is because any factor greater than the square root of **Num1** would require a corresponding smaller factor, which would already have been checked.

If it is less than **Num2**, the program proceeds to check if **is_Correct** is set to **true**. Within the loop, the program checks whether **Num1** divided by **Num2** leaves a remainder of 0. If this condition is true, it means **Num1** is divisible by the value of **Num2**, so **is_Correct** is set to **False**, the loop breaks, and the program moves on to the next value of **Num1**. If the condition is false, **Num2** is incremented by 1, and the loop repeats as long as the condition **Num2 < sqrt(Num1) + 1** holds.

2a. Loops and Termination Conditions

The outer loop is responsible for checking whether **Num1** could potentially be a prime number, and it also checks whether the numbers in the range defined by the user has all been checked.

The inner loop is responsible for filtering out non-prime numbers by checking whether **Num1** is divisible by any number other than 1 and itself. If no such divisor is found, **is_Correct** remains **True**, and **Num1** is added to the **Values[]** list, which stores all identified prime numbers. However, if the loop finds a number that divides **Num1** evenly (without leaving a remainder), it means **Num1** is not prime, **is_Correct** is then set to **False**, the loop exits early, and **Num1** is not added to the list. This inner loop directly affects the outer loop's behaviour by deciding whether each **Num1** value qualifies as a prime. The early exit condition improves efficiency by avoiding unnecessary checks once a divisor is found.

Impact of early exit

If the outer loop exits early, it means that either the range of numbers from start to end+1 have all been checked or that the start and end range is not able to meet the conditions to continue through to the second loop. This helps to avoid unnecessary computations of numbers that do not meet the criteria to be considered possibly prime or are further than the defined range.

If the inner loop terminates early, it sets **is_Correct** to **False** and skips the remaining checks. This allows the outer loop to proceed to the next number. The early exit improves efficiency by avoiding unnecessary calculations and ensures that non-prime numbers are not added to the **Values[]** list.

2b. Boolean Variable

The Boolean variable used in the flowchart is **is_Correct**. Its purpose is to keep track of whether the current value of **Num1** is a prime number. It is initially set to true before any divisor checks are performed. This means that after passing the first two conditions, the program assumes **Num1** is prime unless a later condition proves otherwise. If a divisor is found (i.e., $\text{Num1} \% \text{Num2} == 0$), **is_Correct** is set to **False**.

If the condition **Num2 > sqrt(Num1) + 1** is false, the program checks whether **is_Correct** is still true before adding **Num1** to the **Values []** list. If it is, the number is added, otherwise, it is skipped.

3 Conditional Statements

3a. Purpose of Conditional Statements

In programming, the conditional statements are used to make decisions [3]. It allows certain blocks of code to execute only when a specific condition is met [3]. This controls the flow of the program by determining which parts of the code should run based on logic, data, or user input [3]. Conditional statements are especially useful for implementing decision-making logic [3]. For example, they can be used to validate user input by checking whether the user has entered a valid number or character and then respond accordingly by executing different sections of code. They are also essential for error handling, allowing the program to check for potential issues before they cause problems [3]. If statements help prevent programs from crashing or behaving unpredictably [3].

3b. Explanation of Specific Conditional Statements

If Num1 > 1

This conditional statement checks whether the value of **Num1** is greater than 1. This is important because, by definition, prime numbers are greater than 1. If **Num1** is less than 1, the condition evaluates to false, and the program typically does not check whether the number is prime. The program is structured to skip the prime-checking logic and increment **Num1** and continue looping until **Num1 > 1** is true. This ensures that the program only starts checking for prime numbers when **Num1** is in a valid range (i.e. greater than 1)

If Num1 % Num2 == 0

This conditional statement checks whether the remainder of dividing **Num1** by **Num2** is equal to 0. This is important for identifying non-prime numbers because prime numbers can only be divisible by 1 and itself. If **Num1** is divisible by any number other than 1 and itself leaving a remainder of 0, then **Num1** is not prime.

If Is_Correct == True

This condition checks whether the variable **Is_Correct** is set to **True**. The program uses this variable to track whether **Num1** has been identified as a prime number and has passed all checks. If **Is_Correct** is **True**, it means **Num1** was not divisible by any number other than itself or 1. When this condition is met the program will then add **Num1** to the **Values[]** list.

4a. Program

Generate_special_numbers

```
generate_special_numbers.py ×
1  import math
2
3  start=int(input("Enter Starting Number: "))
4  end=int(input("Enter End Number: "))
5
6  Value=[]
7  Num1= start
8
9  while Num1<end+1: #first loop
10     if Num1 > 1:
11         is_Correct = True
12         Num2 = 2
13         while Num2 < math.sqrt(Num1)+1: #second loop
14             if Num1 % Num2 == 0:
15                 is_Correct = False
16                 Num1=Num1+1
17             else:
18                 Num2 = Num2+1
19         else:
20             if is_Correct == True:
21                 Value.append(Num1) #adds value to list
22                 Num1=Num1+1
23         else:
24             Num1=Num1+1
25     else:
26         print(Value)
```

Generate_special_numbers as Plain Text

```
import math

start=int(input("Enter Starting Number: "))
end=int(input("Enter End Number: "))

Value=[]
Num1= start

while Num1<end+1: #first loop
    if Num1 > 1:
        is_Correct = True
        Num2 = 2
        while Num2 < math.sqrt(Num1)+1: #second loop
            if Num1 % Num2 == 0:
                is_Correct = False
                Num1=Num1+1
            else:
                Num2 = Num2+1
        else:
            if is_Correct == True:
                Value.append(Num1) #adds value to list
                Num1=Num1+1
            else:
                Num1=Num1+1
    else:
        print(Value)
```

4b Test Cases

Start: 3 End:16

The test case with the starting number 3 and end number 16, outputs numbers 3,5,7,11 and 13.

```
Enter Starting Number: 3
Enter End Number: 16
[3, 5, 7, 11, 13]

Process finished with exit code 0
|
```

Start: 25, End: 42

The test case with the starting number as 25 and end number 42, outputs numbers 29, 31, 37 and 41.

```
Enter Starting Number: 25
Enter End Number: 42
[29, 31, 37, 41]

Process finished with exit code 0
```

5. Program Validation

a. Test Case 1:

Start = 12, End = 22

```
Enter Starting Number: 12
Enter End Number: 22
[13, 17, 19]

Process finished with exit code 0
```

The program was executed for a range of numbers from 12 to 22. It evaluated each number within this range using conditional statements to determine whether it is a prime number. Once all prime numbers were identified, they were stored in a list. Finally, the program printed the list of prime numbers found in the given range, which were: 13, 17, and 19.

b. Test Case 2:

Start = 30, End = 8

```
Enter Starting Number: 30
Enter End Number: 8
[]

Process finished with exit code 0
```

When the program checks whether the starting value (30) is less than or equal to the end value plus one ($8 + 1 = 9$), the loop does not execute. As a result, the loop does not execute, and no numbers are evaluated or added to the **Value[]** list. Therefore, the program outputs an empty list.

c. Test Case 3:

Start = -11, End = 9

```
Enter Starting Number: -11
Enter End Number: 9
[3, 5, 7]

Process finished with exit code 0
```

In this scenario, the program does not enter the main prime-checking logic until the value of **Num1** is greater than 1. It increments **Num1** (-11) using **Num1 = Num1 + 1** until this condition is satisfied. Once **Num1** exceeds 1, the program proceeds to evaluate each number for primality. This continues until **Num1** reaches the end value of 10. During this process, the prime numbers identified and added to the list are 3, 5, and 7, which are then printed as the final output.

d. Test Case 4:

Start = 24, End = 27

```
Enter Starting Number: 24
Enter End Number: 27
[]

Process finished with exit code 0
```

Despite running through its validation steps for the given small range, the program does not identify any prime numbers. Since no values satisfy the prime condition, the **Values[]** list remains empty, and the program prints an empty output.

e. Test Case 5:

Start = 13, End = 13

```
Enter Starting Number: 13
Enter End Number: 13
[13]

Process finished with exit code 0
```

The program still performs its checks even though the start and end values of the range are both 13. Since the condition **Num1** > **end** + 1 is not initially met, the loop runs. The number 13 is evaluated and identified as a prime number, then added to the list. After this, **Num1** is incremented, causing the loop condition to fail, and the program exits, printing the list containing the single prime number 13.

6 Optimisation Techniques

3 To optimise the program, the outer loop determines whether execution should continue based on the validity of the range. Loops reduce code repetition by allowing operations to be executed multiple times without duplicating code, improving both time and readability. Additionally, the nested loop allows for early exits during divisor checks. If a number is found to be non-prime, the loop can break immediately, avoiding unnecessary computations. Reusing the variables **Num1** and **Num2** also helps conserve memory and improve performance by eliminating the need for repeated resetting of the variables to their initial values each time.

Using **sqrt(Num1) + 1** further reduces computation time by limiting the number of checks needed to determine if **Num1** has any divisors. This is especially beneficial when dealing with large numbers, as it significantly narrows the range of potential factors.

7 Reflection on Learning Journey

Challenges

Loops presented a significant challenge, especially when determining the most appropriate type to use in varying scenarios. Mastery of these concepts tends to develop through consistent practice and experimentation. Beyond selecting the right loop structure, logical errors can also pose difficulties. For instance, in the process of creating the **generate_special_numbers** program, the initial version was free of syntax errors but contained flawed logic. Resolving these issues required stepping back, re-evaluating the approach, and identifying the root cause of the problem.

Solutions

To address challenges related to loops, a strategic approach involves practicing various loop types within real-world scenarios, gradually increasing complexity as comprehension strengthens. Revisiting foundational loop concepts and thoroughly understanding each type can provide a solid base before progressing to more advanced topics. When it comes to minimising logical errors, examining problems from multiple perspectives and testing different solutions can offer valuable insights into what works effectively and why certain approaches fall short.

Application of Concepts

In tackling this assignment, nested loops were applied to solve the problem. The initial approach involved using a range loop, attempting to preset the range with start and end numbers provided by the user. However, this method did not yield the expected results, as the program produced no output. To troubleshoot, the code was reviewed line by line, with sections commented out to isolate the issue. Replacing the **range loop** with a while loop to initiate the program allowed the code to function as intended.

8 Conclusion

The flowchart uses logic, nested loops and conditional statements to determine whether a number is a prime number. It contains identified prime numbers within a list and prints the final result. The program can self-correct if the user input range starts from below one or is higher than the end value. Optimisation techniques are used to make the code efficient, short and easy to read.

9 References

- [1] Bitesize, "What are Prime Numbers," BBC. [Online]. Available: <https://www.bbc.co.uk/bitesize/articles/z2q26fr#zp38mbk>
- [2] Unknown, "Flowcharts," Ada Computer Science. [Online]. Available: [https://adacomputerscience.org/concepts/design_flowcharts#:~:text=Some loops check their condition at the end \(these loops are called post-condition loops\).&text=To represent a post-condition loop in a decision symbol%2C as shown in Figure 2.](https://adacomputerscience.org/concepts/design_flowcharts#:~:text=Some loops check their condition at the end (these loops are called post-condition loops).&text=To represent a post-condition loop in a decision symbol%2C as shown in Figure 2.)
- [3] O. Jeremiah, "How to Use Conditional Statements in Python – Examples of if, else, and elif," free Code Camp. [Online]. Available: <https://www.freecodecamp.org/news/how-to-use-conditional-statements-if-else-elif-in-python/#:~:text=Conditional statements are an essential, to use them in practice.>

89/100

PAGE 1

PAGE 2

PAGE 3

PAGE 4



Comment 1

to identify

PAGE 5

PAGE 6



Comment 2

The answer is not complete

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11

PAGE 12

PAGE 13



Comment 3

THis sections lack clear explanation

PAGE 14



Comment 4

??

BETHAN MOYLE

89

QUESTION 1

Correctly describes the purpose of the flowchart [5] Identifies number and role of loops (outer and inner) [5]



The answer is incomplete

8

QUESTION 2

*Explains inner loop termination and its effect on the outer loop [4]
Identifies Boolean variable [2] and explains its role [4]*



Correct answer

10

QUESTION 3

General explanation of conditional statements [5] Explain Each conditional statement and its impact [15]



Correct answer

20

QUESTION 4

Correctly implements the program with valid logic [20] Tests with sample inputs and shows results [5]



Code is implemented correctly

25

QUESTION 5 [15]

Describes and explains five test case results [15]



All test cases explained with example.

15

QUESTION 6 [10]

Identifies optimizations and explains performance effectiveness [10]



Good effort; it lacks clarity

5

QUESTION 7 [10]

Reflection on learning with clear examples [10]



Basic mention of challenges

6