ASSIGNMENT – 6.1

Srishanth
2403A51267
Batch-12 Task
1:
(Classes – Employee Management)

- Task: Use AI to create an Employee class with a ributes (name, id, salary) and a method to calculate yearly salary.

CODE:

```
class Employee:
    def __init__(self, name, id, salary):
        self.name = name
        self.id = id
        self.salary = salary

    def display_details(self):
        print(f"Employee Name: {self.name}")
        print(f"Employee ID: {self.id}")
        print(f"Monthly Salary: ${self.salary}")

    def calculate_bonus(self, bonus_percentage):
        bonus_amount = self.salary * (bonus_percentage / 100)
        print(f"Calculated Bonus: ${bonus_amount}")
        return bonus_amount

# Example usage:
employee1 = Employee("Alice", "E123", 5000)
employee1.display_details()
bonus = employee1.calculate_bonus(10) # Calculate 10% bonus
```

OUTPUT:

```
Employee Name: Alice
Employee ID: E123
Monthly Salary: $5000
Calculated Bonus: $500.0
```

Task 2 :
(Loops – Automorphic Numbers in a Range)

- Task: Prompt AI to generate a func on that displays all Automorphic numbers between 1 and 1000 using a for loop.

CODE:

```python
# Review of the existing find_automorphic_numbers function (for loop implementation)

def find_automorphic_numbers(start, end):
  """
  Finds and displays automorphic numbers within a given range.

  An automorphic number is a number whose square ends in the same digits
  as the number itself.

  Args:
    start: The beginning of the range (inclusive).
    end: The end of the range (inclusive).
  """
  print(f"Automorphic numbers between {start} and {end}:")
  for num in range(start, end + 1):
    square = num * num
    if str(square).endswith(str(num)):
      print(num)

# Correctness Analysis:
# The logic of squaring the number and checking if the string representation
# of the square ends with the string representation of the original number
# is a correct way to identify automorphic numbers. The `endswith()` string
# method is suitable for this check. The function iterates through the specified
# range (inclusive), so it covers all numbers in the range.

# Efficiency Analysis:
# The time complexity of this implementation is O(n * log10(n)), where n is the range (end - start + 1).
# The loop runs 'n' times. Inside the loop, the multiplication `num * num` takes constant time for typical integer sizes.
# Converting numbers to strings and performing the `endswith()` check takes time proportional to the number of digits in the number and its square.
# The number of digits in a number 'x' is approximately log10(x). The number of digits in x*x is approximately log10(x*x) = 2*log10(x).
# Therefore, the string operations take roughly O(log10(num)) time.
# Since the loop iterates from 'start' to 'end', the maximum value of 'num' is 'end'.
# So, for each iteration, the dominant operation related to the magnitude of the numbers is the string conversion and comparison, which is O(log10(end)).
# Thus, the overall time complexity is O((end - start + 1) * log10(end)).
# For the given range of 1 to 1000, this is highly efficient and acceptable. As the range grows, the time taken will increase roughly linearly with the :
```

OUTPUT:

```
Automorphic numbers between 1 and 1000:
1
5
6
25
76
376
625
```

Subtask:

Examine the exis ng code for correctness and discuss its efficiencency.

WHILE LOOP:

```python
def find_automorphic_numbers_while(start, end):
  """
  Finds and displays automorphic numbers within a given range using a while loop.

  An automorphic number is a number whose square ends in the same digits
  as the number itself.

  Args:
    start: The beginning of the range (inclusive).
    end: The end of the range (inclusive).
  """
  print(f"Automorphic numbers between {start} and {end} (using while loop):")
  num = start
  while num <= end:
    square = num * num
    if str(square).endswith(str(num)):
      print(num)
    num += 1

# Find automorphic numbers between 1 and 1000 using the while loop function
find_automorphic_numbers_while(1, 1000)
```

```
Automorphic numbers between 1 and 1000 (using while loop):
1
5
6
25
76
376
625
```

COMPARISION:

1. Syntax and Structure:

- For loop (find_automorphic_numbers): Uses `for num in range(start, end + 1):`
- The loop ini aliza on (`num = start`), condi on (`num <= end`), and itera on step (`num += 1`) are implicitly handled by the `range()` func on and the `for` statement.
- While loop (find_automorphic_numbers_while): Uses `while num <= end:` -

  Requires explicit ini aliza on of the loop variable (`num = start`) before the loop.
- Requires explicit upda ng of the loop variable (`num += 1`) within the loop body.
- The loop condi on (`num <= end`) is stated directly in the `while` statement.

2. Readability and Conciseness:

- For loop: Generally considered more readable and concise when itera ng over a known sequence or range.
- The structure `for item in sequence:` clearly indicates itera on over a collec on.
- For itera ng through a range, the `range()` func on makes the start and end points immediately clear.
- While loop: Can be slightly less concise for simple range itera on as it requires managing the loop variable manually.

- More suitable when the number of itera ons is not known in advance or the loop termina on depends on a complex condi on.
- For this specific task (itera ng through a defined range), the for loop is arguably more idioma c and slightly more readable.

3. Performance and Efficiency:

- For loop and While loop implementa ons for this task are expected to have virtually iden cal performance.
- Both loops execute the same core logic (squaring, string conversion, endswith check) for each number in the range.
- The underlying opera ons within the loops are the same.
- Any minor differences in overhead are negligible for typical ranges like 1 to 1000.
- Both implementa ons have a me complexity of O((end - start + 1) * log10(end)) as discussed previously.

4. Advantages and Disadvantages for this Task:

- For loop:
- Advantage: More natural and concise syntax for itera ng over a fixed range. Less prone to infinite loops due to forge ng to update the loop variable.
- Disadvantage: Less flexible than a while loop if the itera on logic becomes more complex and doesn't fit the simple 'iterate over a sequence' pa ern.
- While loop:
- Advantage: More flexible when the loop condi on is not directly ed to itera ng through a simple range or sequence. Useful for condi ons that change dynamically.
- Disadvantage: Requires more careful management of the loop variable (ini aliza on and update), increasing the poten al for off-by-one errors or infinite loops if not handled correctly.
- For this specific task, the advantage of flexibility offered by the while loop is not par cularly needed, making the for loop a slightly be er fit in terms of simplicity and conciseness.

Conclusion: For finding automorphic numbers within a fixed range, the `for` loop implementa on is generally preferred due to its cleaner syntax and reduced risk of errors related to loop variable management, although both implementa ons are equally efficient.

Task 3 :

(Condi onal Statements – Online Shopping Feedback Classifica on)

- Task: Ask AI to write nested if-elif-else condi ons to classify online shopping feedback as Posi ve, Neutral, or Nega ve based on a numerical ra ng (1–5).

CODE:

```python
def classify_feedback(rating):
  """
  Classifies online shopping feedback as Positive, Neutral, or Negative
  based on a numerical rating from 1 to 5 using nested if-elif-else.

  Args:
    rating: A numerical rating from 1 to 5.

  Returns:
    A string indicating the feedback classification ("Positive", "Neutral",
    "Negative"), or "Invalid rating" if the input is outside the 1-5 range.
  """
  if 1 <= rating <= 5:
    if rating >= 4:
      return "Positive"
    elif rating == 3:
      return "Neutral"
    else: # rating is 1 or 2
      return "Negative"
  else:
    return "Invalid rating"

# Example usage:
print(f"Rating 5: {classify_feedback(5)}")
print(f"Rating 3: {classify_feedback(3)}")
print(f"Rating 1: {classify_feedback(1)}")
print(f"Rating 0: {classify_feedback(0)}")
print(f"Rating 6: {classify_feedback(6)}")
```

OUTPUT:

```
Rating 5: Positive
Rating 3: Neutral
Rating 1: Negative
Rating 0: Invalid rating
Rating 6: Invalid rating
```

Task 4 (Loops – Prime Numbers in a Range)

- Task: Generate a func on using AI that displays all prime numbers within a user-specified range (e.g., 1 to 500).

CODE:

```
def find_prime_numbers_for(start, end):
  """
  Finds and displays prime numbers within a given range using a for loop.

  Args:
    start: The beginning of the range (inclusive).
    end: The end of the range (inclusive).
  """
  print(f"Prime numbers between {start} and {end} (using for loop):")
  for num in range(start, end + 1):
    # Prime numbers are greater than 1
    if num > 1:
      is_prime = True
      # Check for factors from 2 up to the number itself (exclusive)
      for i in range(2, num):
        if (num % i) == 0:
          is_prime = False
          break # Not prime, so break the inner loop
      if is_prime:
        print(num)

# Find prime numbers between 1 and 500 using the for loop function
find_prime_numbers_for(1, 500)
```

OUTPUT:

```
Prime numbers between 1 and 500 (using for loop):
2
3
5
7
11
13
17
19
23
29
31
37
41
43
47
53
59
61
67
71
73
79
83
89
97
101
103
107
109
113
127
131
137
139
149
151
157
163
167
173
179
181
191
193
197
199
211
223
227
229
```

Subtask:

Examine the generated code for correctness and discuss its efficiency, par cularly focusing on the prime-checking logic.

```python
# Review of the existing find_prime_numbers_for function (for loop implementation)

def find_prime_numbers_for(start, end):
  """
  Finds and displays prime numbers within a given range using a for loop.

  Args:
    start: The beginning of the range (inclusive).
    end: The end of the range (inclusive).
  """
  print(f"Prime numbers between {start} and {end} (using for loop):")
  for num in range(start, end + 1):
    # Prime numbers are greater than 1
    if num > 1:
      is_prime = True
      # Check for factors from 2 up to the number itself (exclusive)
      for i in range(2, num):
        if (num % i) == 0:
          is_prime = False
          break # Not prime, so break the inner loop
      if is_prime:
        print(num)

# Correctness Analysis:
# The function correctly handles numbers greater than 1.
# It correctly identifies 2 as prime because the inner loop range(2, 2) is empty,
# and is_prime remains True.
# For numbers greater than 2, it checks for divisibility by all integers from 2 up to num - 1.
# If any divisor is found, the number is marked as not prime, and the inner loop breaks.
# If no divisor is found after checking all numbers up to num - 1, the number is considered prime.
# This logic is correct for identifying prime numbers.
# Edge case: Numbers less than or equal to 1 are correctly excluded as non-prime.

# Efficiency Analysis:
# The outer loop iterates through each number from start to end, which is O(end - start + 1) iterations.
# The inner loop iterates from 2 up to num - 1. In the worst case (when num is prime or has no small factors),
# this inner loop runs approximately num - 2 times, which is O(num) iterations.
# For each iteration of the inner loop, the modulo operation (num % i) is performed, which is a constant time operation.
# Therefore, the time complexity of checking a single number 'num' for primality is O(num).
# The overall time complexity of the find_prime_numbers_for function is the sum of the complexities
# for checking each number from start to end. In the worst case (e.g., checking many prime numbers),
# this is approximately the sum of O(num) for each num from start to end.
# This results in a time complexity of roughly O(end^2) in the worst case for the range 1 to end.
# For a range of 1 to 500, this O(n^2) approach is acceptable. However, for much larger ranges,
# this approach becomes inefficient as the time taken increases quadratically with the upper limit of the range.
# For example, checking primality up to 10,000 would be significantly slower than up to 500.
```

Subtask:Create an op mized version of the func on, for example, by using the square root method for prime checking.

```python
[11] import math

def find_prime_numbers_optimized(start, end):
  """
  Finds and displays prime numbers within a given range using an optimized
  approach (checking factors up to the square root).

  Args:
    start: The beginning of the range (inclusive).
    end: The end of the range (inclusive).
  """
  print(f"Prime numbers between {start} and {end} (optimized):")
  for num in range(start, end + 1):
    # Prime numbers are greater than 1
    if num > 1:
      # Handle the special case of 2
      if num == 2:
        print(num)
        continue # Move to the next number

      # Check for factors from 2 up to the square root of the number
      is_prime = True
      # We only need to check up to the integer part of the square root
      limit = int(math.sqrt(num))
      for i in range(2, limit + 1):
        if (num % i) == 0:
          is_prime = False
          break # Not prime, so break the inner loop
      if is_prime:
        print(num)

# Find prime numbers between 1 and 500 using the optimized function
find_prime_numbers_optimized(1, 500)
```

output:

```
Prime numbers between 1 and 500 (optimized):
2
3
5
7
11
13
17
19
23
29
31
37
41
43
47
53
59
61
67
71
73
79
83
89
97
101
103
107
109
113
127
131
137
139
149
151
157
163
167
173
179
181
191
193
197
199
211
223
227
229
233
239
241
251
257
263
```

```
229
233
239
241
251
257
263
269
271
277
281
283
293
307
311
313
317
331
337
347
349
353
359
367
373
379
383
389
397
401
409
419
421
431
433
439
443
449
457
461
463
467
479
487
491
499
```

Subtask:

Discuss the differences in efficiency between the ini al and op mized versions. Comparison of Efficiency: Ini al (O(n^2)) vs. Op mized (O(n * sqrt(n))) Prime Finding:

----

1. Time Complexity Comparison:

- Ini al (for loop): O(end^2)
- The inner loop checks for divisibility up to `num - 1` for each number `num`, leading to a quadra c rela onship with the upper limit of the range (`end`).
- Op mized (square root method): O(end * sqrt(end))
- For each number `num`, the inner loop checks for divisibility up to `sqrt(num)`. The maximum value of `sqrt(num)` is `sqrt(end)`.
- The overall complexity is the sum of checking each number, which is approximately propor onal to `end * sqrt(end)`.

2. Why checking up to the square root is sufficient:

To determine if a number `num` is prime, we need to check if it has any divisors other than 1 and itself.

If `num` has a divisor `i` such that `1 < i < num`, then `num = i * j` for some integer `j` where `1 < j < num`.

If `i` is a divisor, then `j = num / i` is also a divisor.

Consider the rela onship between `i` and `j`:

- If `i < sqrt(num)`, then `j = num / i > num / sqrt(num) = sqrt(num)`. So, if there's a divisor less than the square root, there's a corresponding divisor greater than the square root.
- If `i > sqrt(num)`, then `j = num / i < num / sqrt(num) = sqrt(num)`. So, if there's a divisor greater than the square root, there's a corresponding divisor less than the square root.
- If `i = sqrt(num)`, then `num` is a perfect square, and `i = j = sqrt(num)` is a divisor.

This means that if a number `num` has any divisor other than 1 and itself, it must have at least one divisor less than or equal to its square root.

Therefore, to check for primality, we only need to test for divisibility by integers from 2 up to the integer part of `sqrt(num)`.

3. Contribu on to overall performance improvement:

The reduc on in the number of checks for each number significantly improves performance, especially for larger ranges.

For a number `num`, the ini al method performs approximately `num` checks in the worst case.

The op mized method performs approximately `sqrt(num)` checks in the worst case.

The number of checks is reduced from linear (propor onal to `num`) to sublinear (propor onal to `sqrt(num)`).

As the range (`end`) increases, the difference between `end^2` and `end * sqrt(end)` grows substan ally.

For example:

- If end = 100, ini al checks ~ 100^2 = 10,000. Op mized checks ~ 100 * sqrt(100) = 100 * 10 = 1,000.

- If end = 1000, ini al checks ~ 1000^2 = 1,000,000. Op mized checks ~ 1000 * sqrt(1000) ≈ 1000 * 31.6 ≈ 31,600.

- If end = 10000, ini al checks ~ 10000^2 = 100,000,000. Op mized checks ~ 10000 * sqrt(10000) = 10000 * 100 = 1,000,000.

The op mized method performs significantly fewer checks for each number, and this saving accumulates over the en re range, leading to a much faster execu on me for larger inputs.

Task 5 (Classes – Library System)

- Task: Use AI to build a Library class with methods to add_book(), issue_book(), and display_books().

CODE:

```python
class Library:
    def __init__(self):
        self.books = []  # List to store books (e.g., as dictionaries or objects)

    def add_book(self, book):
        """Adds a book to the library."""
        self.books.append(book)
        print(f"Book '{book}' added to the library.")

    def issue_book(self, book_title):
        """Issues a book from the library."""
        if book_title in self.books:
            self.books.remove(book_title)
            print(f"Book '{book_title}' issued successfully.")
        else:
            print(f"Book '{book_title}' not available in the library.")

    def display_books(self):
        """Displays all books currently in the library."""
        if self.books:
            print("Books in the library:")
            for book in self.books:
                print(f"- {book}")
        else:
            print("The library is currently empty.")

# Example Usage:
my_library = Library()

my_library.add_book("The Hitchhiker's Guide to the Galaxy")
my_library.add_book("Pride and Prejudice")
my_library.add_book("1984")

my_library.display_books()

my_library.issue_book("Pride and Prejudice")
my_library.issue_book("Moby Dick") # Trying to issue an unavailable book

my_library.display_books()
```

OUTPUT:

```
Book 'The Hitchhiker's Guide to the Galaxy' added to the library.
Book 'Pride and Prejudice' added to the library.
Book '1984' added to the library.
Books in the library:
- The Hitchhiker's Guide to the Galaxy
- Pride and Prejudice
- 1984
Book 'Pride and Prejudice' issued successfully.
Book 'Moby Dick' not available in the library.
Books in the library:
- The Hitchhiker's Guide to the Galaxy
- 1984
```