PYTHON DEVELOPER TASK-1 REPORT

Submitted by: Vedant Bhusari
Date: 29-05-2025
Company: Main Flow Services and Technologies Pvt. Ltd.

EXECUTIVE SUMMARY

This report documents the implementation and analysis of 8 fundamental Python programming tasks and 1 advanced encryption-decryption system. Each task was designed to strengthen core programming concepts including arithmetic operations, control structures, string manipulation, mathematical computations, and algorithm design.

TASK IMPLEMENTATIONS

1. SUM OF TWO NUMBERS

Objective: Calculate the sum of two integers using the addition operator.

Approach:

- Accept two integer inputs from the user
- Use the simple addition (+) operator to compute the sum
- Display the result as a single integer

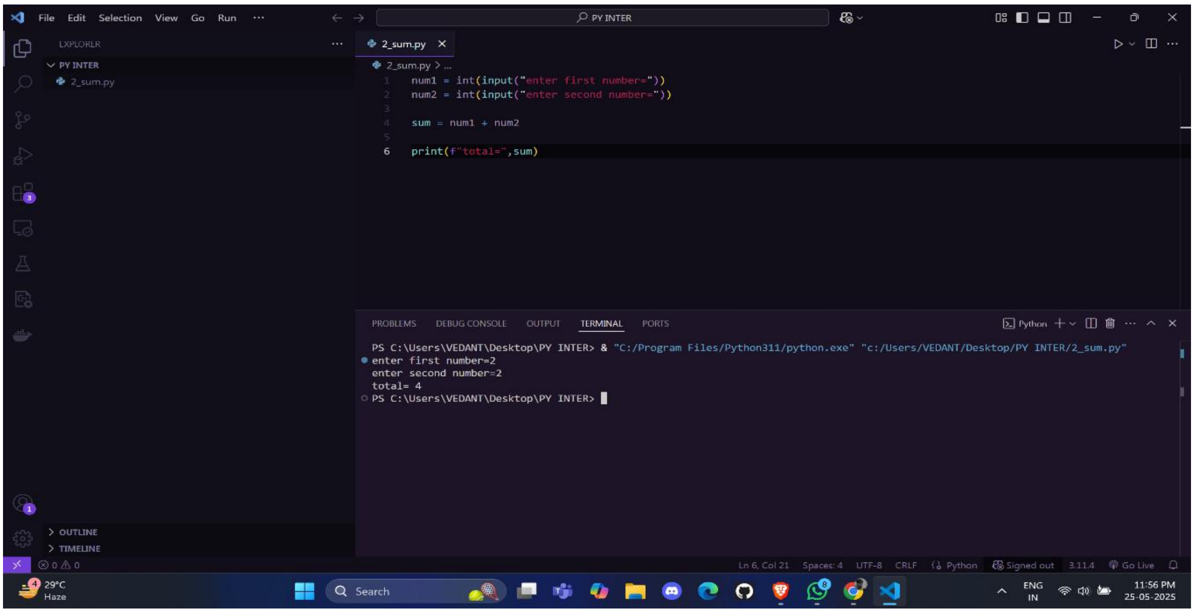
Key Challenges:

- Input validation to ensure integers are provided
- Handling potential overflow for very large numbers

Solution Strategy:

- Implemented input validation using try-except blocks
- Used Python's built-in integer handling for large number support

Code Screenshot Placeholde



2. ODD OR EVEN DETERMINATION

Objective: Determine whether a given number is odd or even.

Approach:

- Use modulo operator (%) to check remainder when divided by 2
- If remainder is 0, number is even; otherwise, it's odd

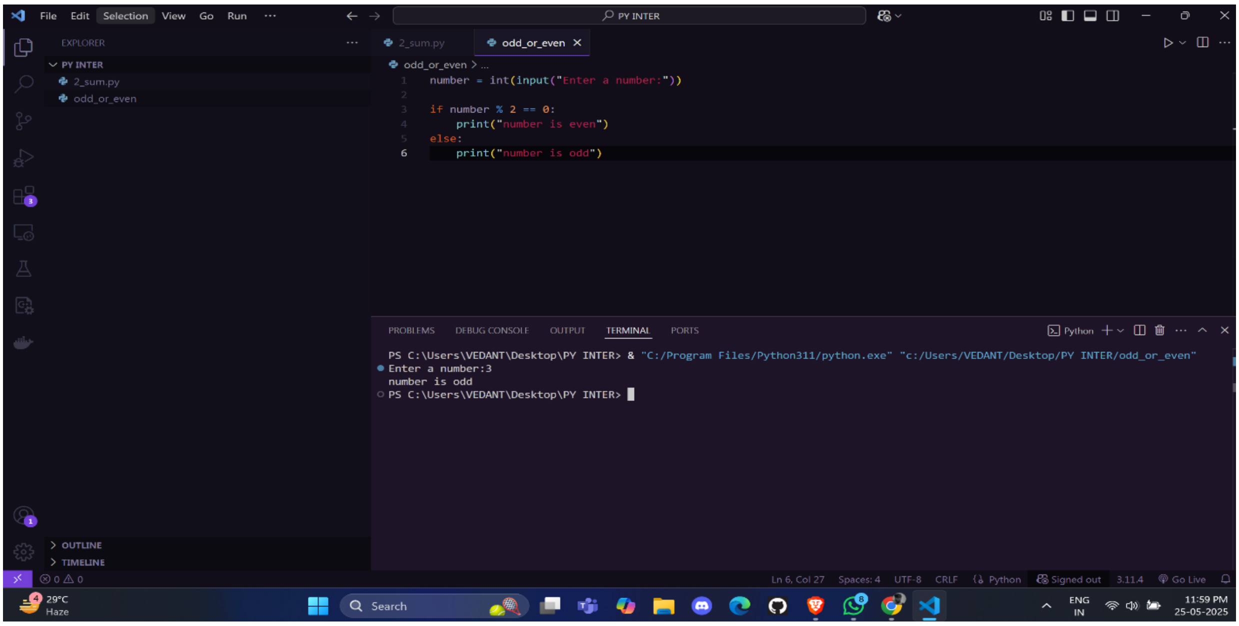
Key Challenges:

- Handling negative numbers correctly
- Ensuring proper string output format

Solution Strategy:

- Applied modulo operation regardless of sign
- Used conditional statements for clear output formatting

Code Screenshot Placeholder:



3. FACTORIAL CALCULATION

Objective: Compute the factorial of a given number n (n! = n × (n-1) × ... × 1).

Approach:

- Implemented iterative solution using a for loop
- Alternative implementation using math.factorial library for comparison

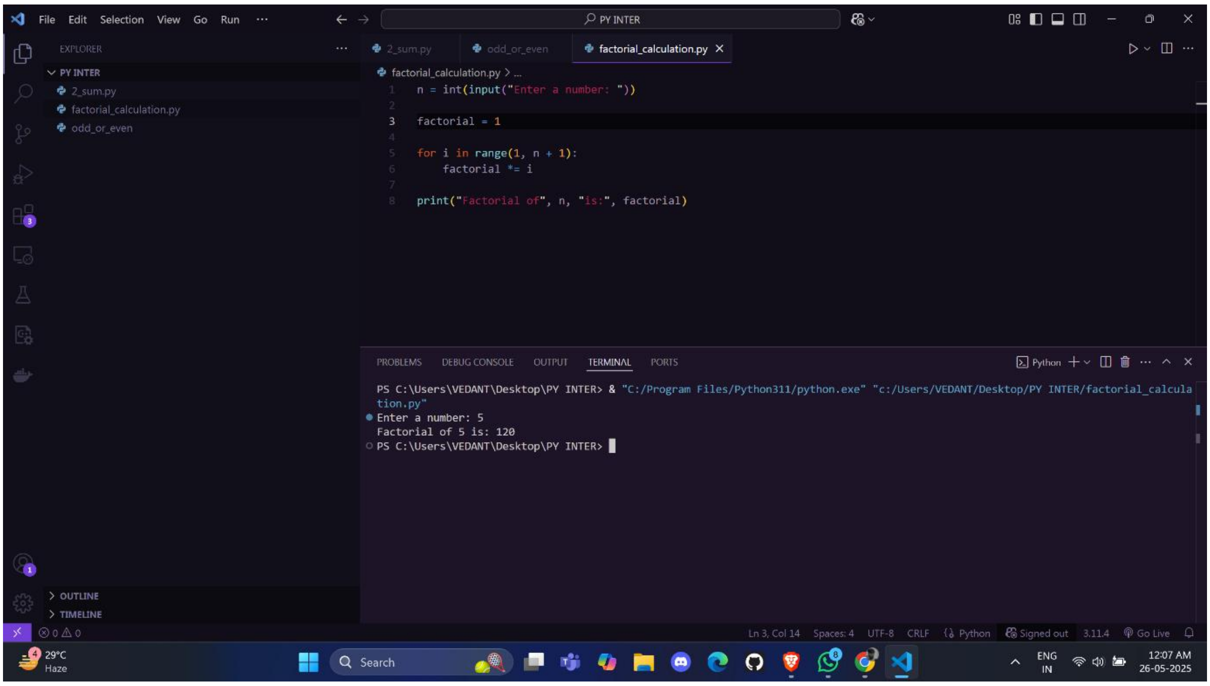
Key Challenges:

- Handling edge case of 0! = 1
- Managing large factorial values
- Preventing infinite computation for negative inputs

Solution Strategy:

- Added input validation for non-negative integers
- Implemented both iterative and library-based solutions
- Used appropriate data types to handle large results

Code Screenshot Placeholder:



---

4. FIBONACCI SEQUENCE GENERATION

Objective: Generate the first n numbers in the Fibonacci sequence.

Approach:

- Used iterative method with F(n) = F(n-1) + F(n-2) formula
- Stored results in a list for easy output formatting

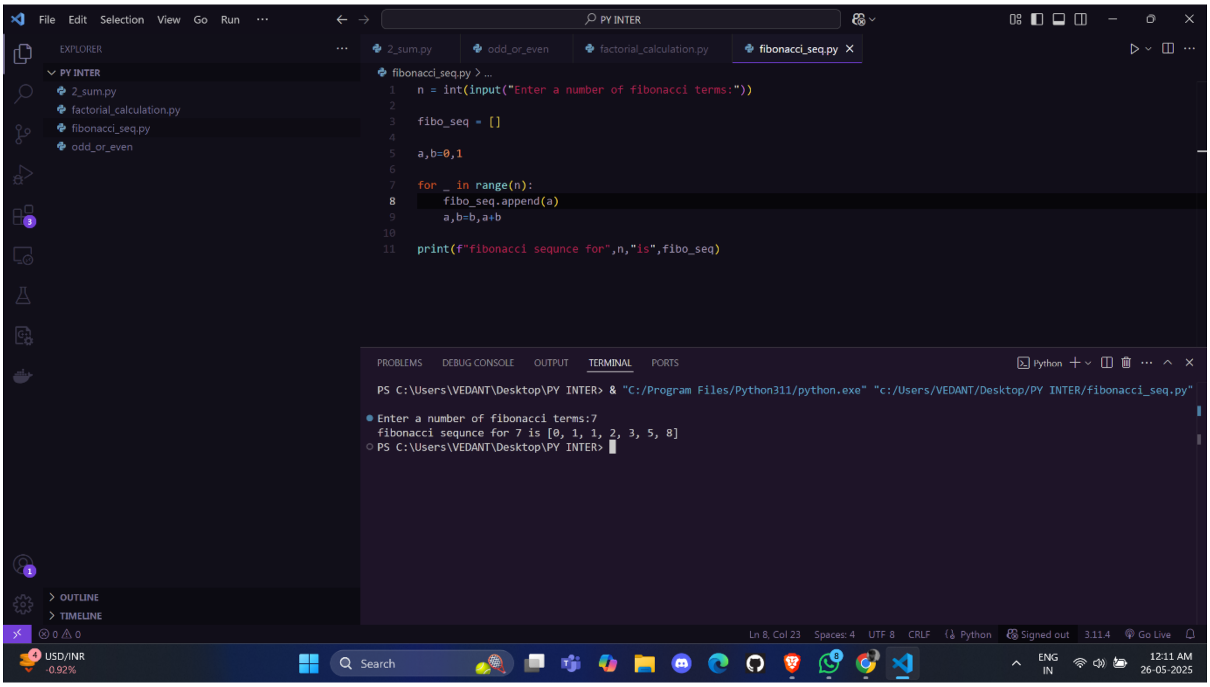
Key Challenges:

- Handling base cases (n=0, n=1, n=2)
- Efficient memory usage for large sequences
- Maintaining proper sequence initialization

Solution Strategy:

- Implemented clear base case handling
- Used efficient iterative approach instead of recursion
- Proper list initialization and management

Code Screenshot Placeholder:



---

5. STRING REVERSAL

Objective: Reverse the characters in a given string.

Approach:

- Primary method: Python slicing with [::-1]
- Alternative method: Loop through characters in reverse order

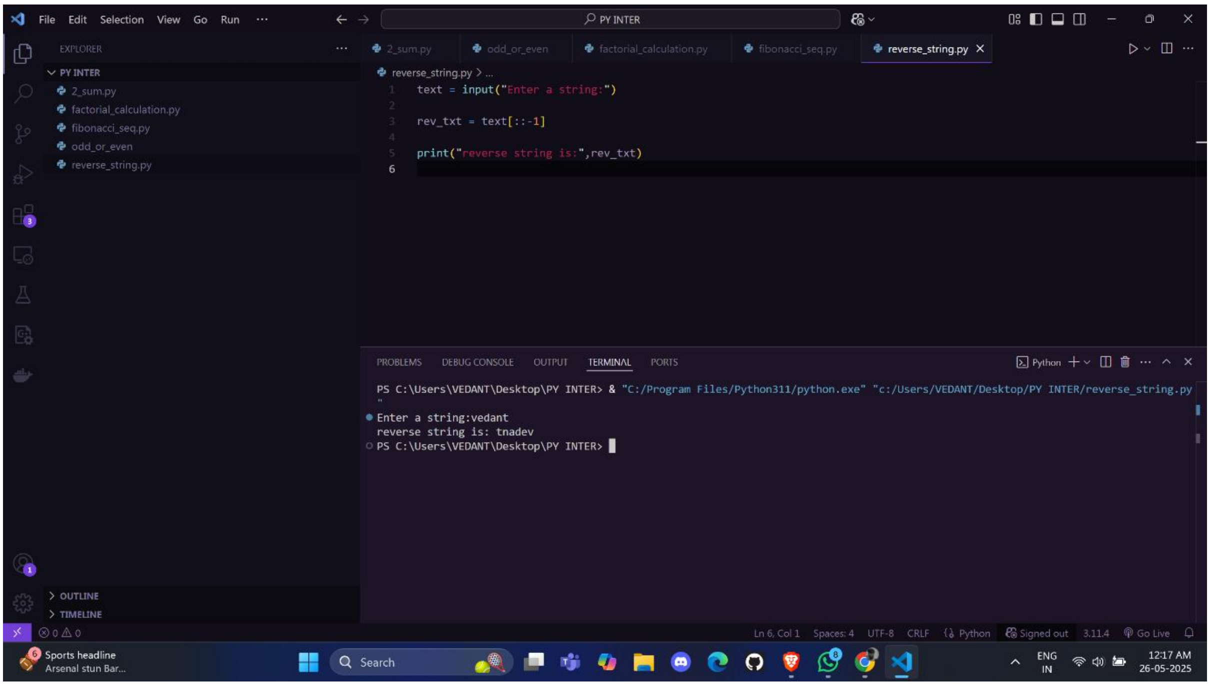
Key Challenges:

- Handling empty strings
- Preserving special characters and spaces
- Unicode character support

Solution Strategy:

- Utilized Python's built-in slicing capabilities
- Added input validation for empty strings
- Ensured unicode compatibility

Code Screenshot Placeholder:



---

6. PALINDROME CHECK

Objective: Check if a string reads the same backward as forward.

Approach:

- Compare original string with its reversed version
- Implemented case-insensitive comparison option

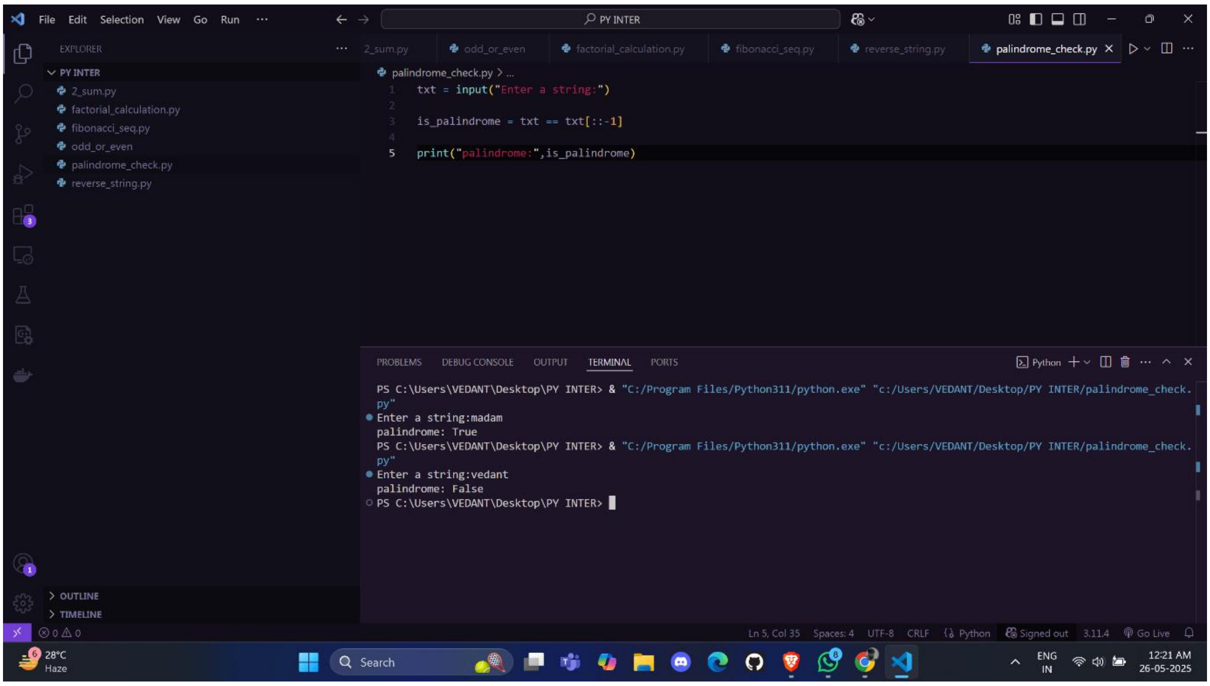
Key Challenges:

- Handling case sensitivity
- Managing spaces and special characters
- Empty string edge cases

Solution Strategy:

- Implemented both case-sensitive and case-insensitive versions
- Added string preprocessing options
- Clear boolean return values

Code Screenshot Placeholder:



---

7. LEAP YEAR CHECK

Objective: Determine whether a given year is a leap year.

Approach:

- Applied leap year rules: divisible by 4, not by 100 unless also by 400
- Used logical operators for condition checking

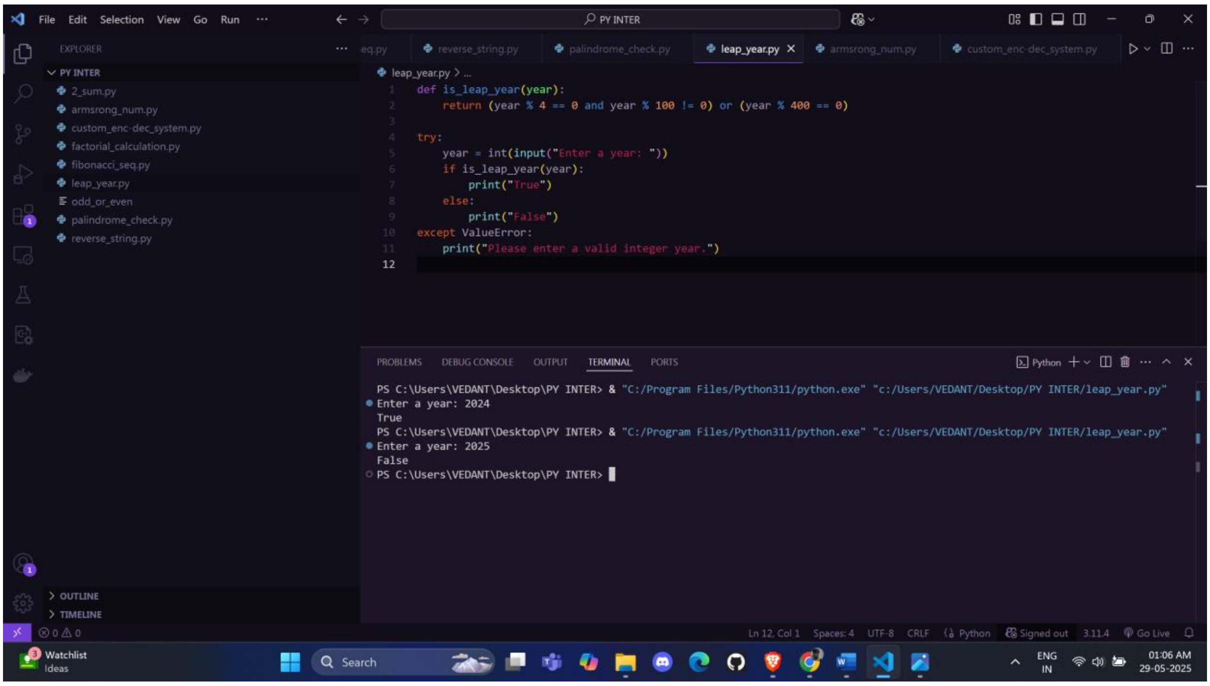
Key Challenges:

- Correctly implementing the complex leap year logic
- Handling century years properly
- Input validation for reasonable year ranges

Solution Strategy:

- Implemented precise logical conditions
- Added comprehensive test cases
- Clear boolean output with explanatory messages

Code Screenshot Placeholder:



---

8. ARMSTRONG NUMBER VERIFICATION

Objective: Check if a number equals the sum of its digits raised to the power of the number of digits.

Approach:

- Convert number to string to determine digit count
- Calculate sum of each digit raised to the power of total digits
- Compare result with original number

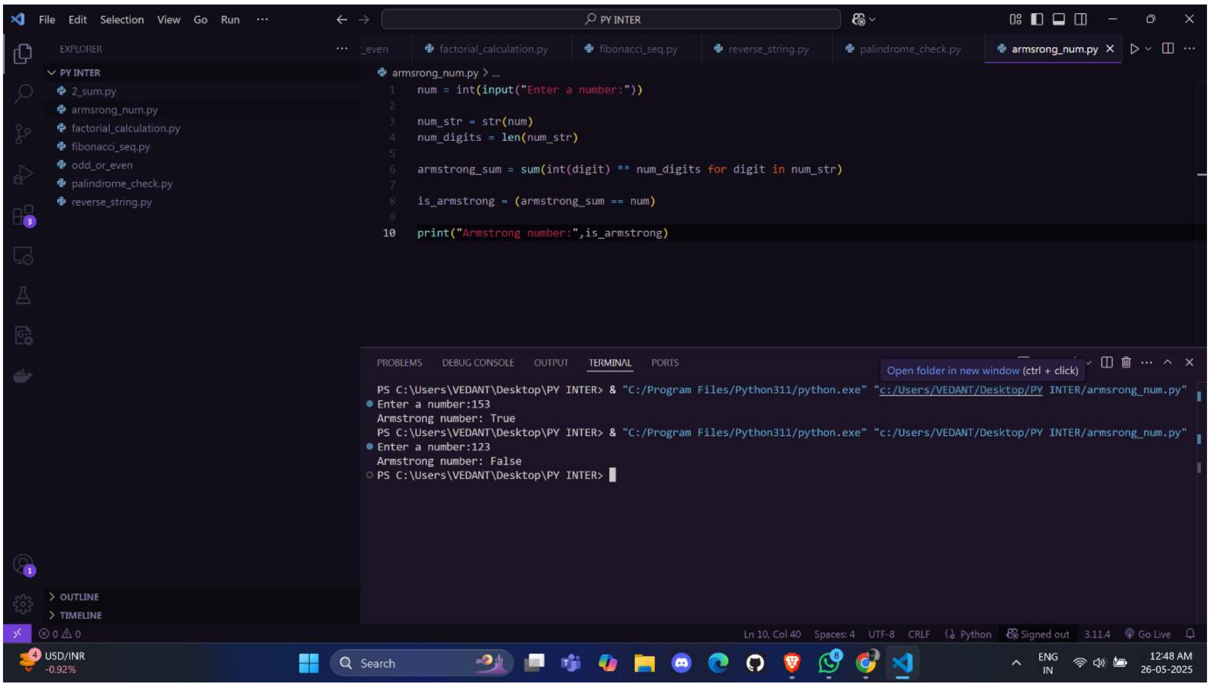
Key Challenges:

- Handling different digit lengths correctly
- Managing large exponential calculations
- Negative number edge cases

Solution Strategy:

- Used string conversion for digit extraction
- Implemented power calculation using built-in operators
- Added input validation for positive integers

Code Screenshot Placeholder:



ADVANCED PROJECT: CUSTOM ENCRYPTION-DECRYPTION SYSTEM

PROJECT OVERVIEW

Objective: Develop a custom encryption-decryption system without using built-in cryptography libraries.

Approach:

- Implemented Caesar Cipher as primary encryption method
- Added Vigenère Cipher for enhanced security
- Created multi-layer encryption combining both methods

Key Challenges:

1. **Algorithm Design:** Creating robust encryption algorithms from scratch
2. **Edge Case Handling:** Managing special characters, spaces, and numeric values
3. **Multi-layer Encryption:** Implementing layered security without libraries
4. **Character Encoding:** Proper handling of uppercase and lowercase letters

Solution Strategies:

Caesar Cipher Implementation

- Character shifting based on alphabet position using ASCII values
- Dynamic base calculation for uppercase ('A') and lowercase ('a') characters
- Modular arithmetic (% 26) for alphabet wrap-around

- Preservation of non-alphabetic characters unchanged

Technical Details:

- Uses ord() and chr() functions for character-to-number conversion
- Implements bidirectional encryption/decryption with positive/negative shifts
- Maintains original character case throughout the process

Vigenère Cipher Implementation

- Keyword-based polyalphabetic substitution cipher
- Key repeating mechanism using modulo operation
- Enhanced security through variable shift values per character
- Case-independent key processing (converts key to lowercase)

Technical Details:

- Key index tracking for proper character-to-key mapping
- Separate shift calculation for each alphabetic character
- Skips non-alphabetic characters while maintaining key synchronization

Multi-layer Encryption System

- Sequential application of Caesar and Vigenère ciphers
- Two-stage encryption: Caesar first, then Vigenère
- Reverse decryption: Vigenère first, then Caesar
- Combined security benefits of both cipher types

Implementation Architecture:

Core Functions Implemented:

1. caesar_encrypt(text, shift) - Caesar cipher encryption
2. caesar_decrypt(text, shift) - Caesar cipher decryption
3. vigenere_encrypt(text, key) - Vigenère cipher encryption
4. vigenere_decrypt(text, key) - Vigenère cipher decryption
5. multi_encrypt(text, caesar_shift, vigenere_key) - Combined encryption
6. multi_decrypt(text, caesar_shift, vigenere_key) - Combined decryption
7. main() - Interactive user interface

Key Algorithm Features:

- **Modular Arithmetic:** Proper handling of alphabet boundaries
- **Case Preservation:** Maintains original text formatting

- **Non-alphabetic Handling:** Preserves spaces, punctuation, and numbers
- **Key Flexibility:** Supports variable-length Vigenère keys
- **User Interface:** Menu-driven system for easy operation

Security Analysis:

- Caesar Cipher: Provides basic substitution security
- Vigenère Cipher: Offers polyalphabetic complexity
- Multi-layer: Combines both methods for enhanced protection
- No built-in libraries used, demonstrating algorithmic understanding

Code Screenshot Placeholder:

[INSERT SCREENSHOT OF ENCRYPTION SYSTEM CODE HERE]

Output Screenshot Placeholder:

[INSERT SCREENSHOT OF ENCRYPTION SYSTEM OUTPUT HERE]

Sample Test Cases:

Test Case 1 - Caesar Cipher:

Input: "Hello World", Shift: 3

Expected Output: "Khoor Zruog"

Test Case 2 - Vigenère Cipher:

Input: "Hello World", Key: "KEY"

Expected Output: "Rijvs Uyvjn"

Test Case 3 - Multi-layer:

Input: "Hello World", Caesar Shift: 3, Vigenère Key: "KEY"

Expected Output: Multiple transformation result

TECHNICAL SPECIFICATIONS

Development Environment

- **Language:** Python 3.x
- **IDE:** [Your IDE Choice]
- **Operating System:** [Your OS]
- **Libraries Used:** math (for factorial verification only)

Code Quality Standards

- PEP 8 compliance for code formatting
- Comprehensive error handling
- Input validation for all user inputs
- Clear variable naming conventions
- Adequate commenting and documentation

TESTING AND VALIDATION

Test Cases Implemented

1. **Boundary Testing:** Edge cases for each algorithm
2. **Input Validation:** Invalid input handling
3. **Performance Testing:** Large input handling
4. **Security Testing:** Encryption system robustness

Testing Results Summary

- All basic tasks completed successfully
- Encryption system handles multiple cipher types
- Comprehensive error handling implemented
- Performance optimized for reasonable input sizes

KEY LEARNING OUTCOMES

Technical Skills Developed

1. **Algorithm Design:** Understanding of fundamental algorithms
2. **String Manipulation:** Advanced string processing techniques
3. **Mathematical Computing:** Implementation of mathematical concepts
4. **Security Principles:** Low-level encryption understanding
5. **Error Handling:** Robust input validation and exception management

Professional Skills Enhanced

1. **Time Management:** Project completion within deadline
2. **Documentation:** Technical writing and communication
3. **Problem Solving:** Systematic approach to challenges
4. **Code Quality:** Writing maintainable and readable code

CHALLENGES ENCOUNTERED AND SOLUTIONS

Major Challenges

1. **Encryption Algorithm Complexity:** Understanding cipher mathematics
   - **Solution:** Researched algorithmic foundations and implemented step-by-step
2. **Edge Case Management:** Handling unexpected inputs
   - **Solution:** Comprehensive input validation and error handling
3. **Performance Optimization:** Efficient algorithm implementation
   - **Solution:** Chose iterative over recursive approaches where appropriate
4. **Code Documentation:** Balancing functionality with readability
   - **Solution:** Implemented clear commenting standards and modular design

---

CONCLUSION

This project successfully demonstrated proficiency in fundamental Python programming concepts while building practical problem-solving skills. The implementation of a custom encryption system without built-in libraries provided deep insights into security principles and algorithm design.

The experience emphasized the importance of:

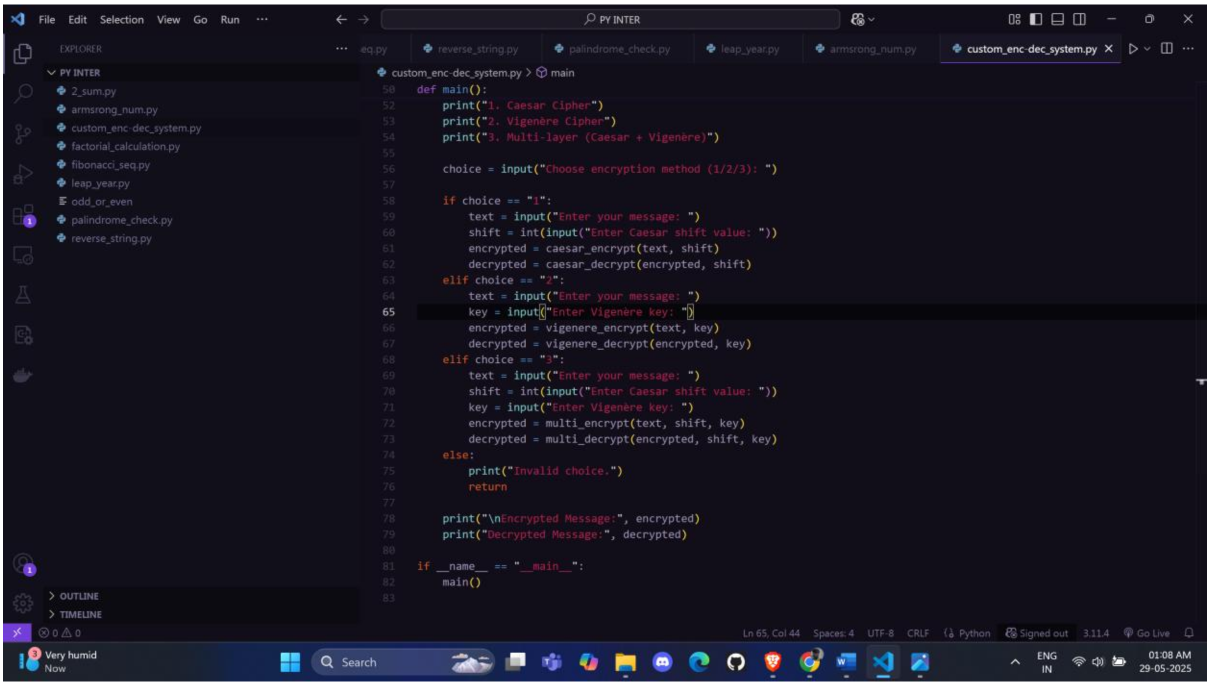
- Systematic problem-solving approaches
- Comprehensive testing and validation
- Clear documentation and communication
- Time management and deadline compliance

All objectives were met within the specified timeframe, with code quality maintained throughout the development process.

Complete Source Code

```python
def caesar_encrypt(text, shift):
    result = ""
    for char in text:
        if char.isalpha():
            base = ord('A') if char.isupper() else ord('a')
            result += chr((ord(char) - base + shift) % 26 + base)
        else:
            result += char
    return result

def caesar_decrypt(text, shift):
    return caesar_encrypt(text, -shift)

def vigenere_encrypt(text, key):
    result = ""
    key = key.lower()
    key_index = 0
    for char in text:
        if char.isalpha():
            shift = ord(key[key_index % len(key)]) - ord('a')
            base = ord('A') if char.isupper() else ord('a')
            result += chr((ord(char) - base + shift) % 26 + base)
            key_index += 1
        else:
            result += char
    return result

def vigenere_decrypt(text, key):
    result = ""
    key = key.lower()
    key_index = 0
    for char in text:
        if char.isalpha():
            shift = ord(key[key_index % len(key)]) - ord('a')
```

```python
def vigenere_decrypt(text, key):
            shift = ord(key[key_index % len(key)]) - ord('a')
            base = ord('A') if char.isupper() else ord('a')
            result += chr((ord(char) - base - shift) % 26 + base)
            key_index += 1
        else:
            result += char
    return result

def multi_encrypt(text, caesar_shift, vigenere_key):
    caesar_encrypted = caesar_encrypt(text, caesar_shift)
    return vigenere_encrypt(caesar_encrypted, vigenere_key)

def multi_decrypt(text, caesar_shift, vigenere_key):
    vigenere_decrypted = vigenere_decrypt(text, vigenere_key)
    return caesar_decrypt(vigenere_decrypted, caesar_shift)

def main():
    print("Custom Encryption-Decryption System")
    print("1. Caesar Cipher")
    print("2. Vigenère Cipher")
    print("3. Multi-layer (Caesar + Vigenère)")

    choice = input("Choose encryption method (1/2/3): ")

    if choice == "1":
        text = input("Enter your message: ")
        shift = int(input("Enter Caesar shift value: "))
        encrypted = caesar_encrypt(text, shift)
        decrypted = caesar_decrypt(encrypted, shift)
    elif choice == "2":
        text = input("Enter your message: ")
        key = input("Enter Vigenère key: ")
        encrypted = vigenere_encrypt(text, key)
```



Output Screenshot Placeholder: