**Lab Manual**

# Practical and Skills Development

# CERTIFICATE

PERFORMED BY

THE ASSIGNMENT ENTERED IN THIS REPORT HAVE BEEN SATISFACTORILY

|  |  |  |
| --- | --- | --- |
| **Registration No** |  | : 25BAI10764 |
| **Name of Student** |  | : Anshika Singh |
| **Course Name** |  | : Introduction to Problem Solving and Programming |
| **Course Code** |  | : CSE1021 |
| **School Name** |  | : SCAI |
| **Slot** |  | : B11+B12+B13 |
| **Class ID** |  | : BL2025260100796 |
| **Semester** |  | : FALL 2025/26 |
| **Course Faculty Name** |  | : Dr. Hemraj S. Lamkuche |

**Signature:**



**QUESTION 1** :Factorial Calculation and Divisor Count Analysis

**AIM/OBJECTIVE(s)**:

1. To calculate the **factorial** of a user-input number (n).
2. To measure the **execution time** required for the factorial calculation.
3. To count the total number of **divisors** for the input number (n).
4. To estimate the approximate **memory used** based on the number of divisors.

**METHODOLOGY & TOOL USED:**

**. Methodology:**Iterative Calculation: A for loop is used to calculate the factorial by repeatedly multiplying a running total (t). A separate for loop is used to check for and count the divisors of the input number (n) **.**

**Tool Used:** Python programming language, utilizing the built-in time module for performance measurement.

**BRIEF DESCRIPTION**:

The program first prompts the user to **enter a number (n)**. It then immediately starts a timer. It calculates the factorial of n using a loop that runs from 1 to n, accumulating the product in the variable t. After the loop completes, it stops the timer and prints the **factorial** and the **execution time**. Finally, a second loop iterates from 1 to n to count how many times n is perfectly divisible, storing the count in divisors. This count is then used to give a rough **approximation of memory used** (by multiplying the divisor count by 28).

**RESULTS ACHIEVED**:

The program successfully achieves the following for a given input number n:

1. **Factorial (n!)**: The computed product of all positive integers less than or equal to n.
2. **Execution Time**: The time taken (in seconds) for the factorial calculation loop to complete.
3. **Divisors Count**: The total number of positive integers that divide n without a remainder.
4. **Memory Used (Approx.)**: An estimated memory usage value based on the calculated divisor count.

**DIFFICULTY FACED BY STUDENT**:

**Large Numbers:** For relatively large inputs (n≥20), the factorial quickly becomes an extremely large number that can exceed the limits of standard integer data types (though Python handles large integers automatically, the calculation time increases significantly).

**Approximation:** The memory calculation (divisors \* 28) is a simple, non-standard **approximation** that doesn't reflect the actual memory allocation of the program, which might confuse a student trying to understand real memory usage.

**Performance:** Understanding the limitations of measuring such short execution times with the time.time() function, which can be affected by system load and clock resolution.

**SKILLS ACHIEVED**:

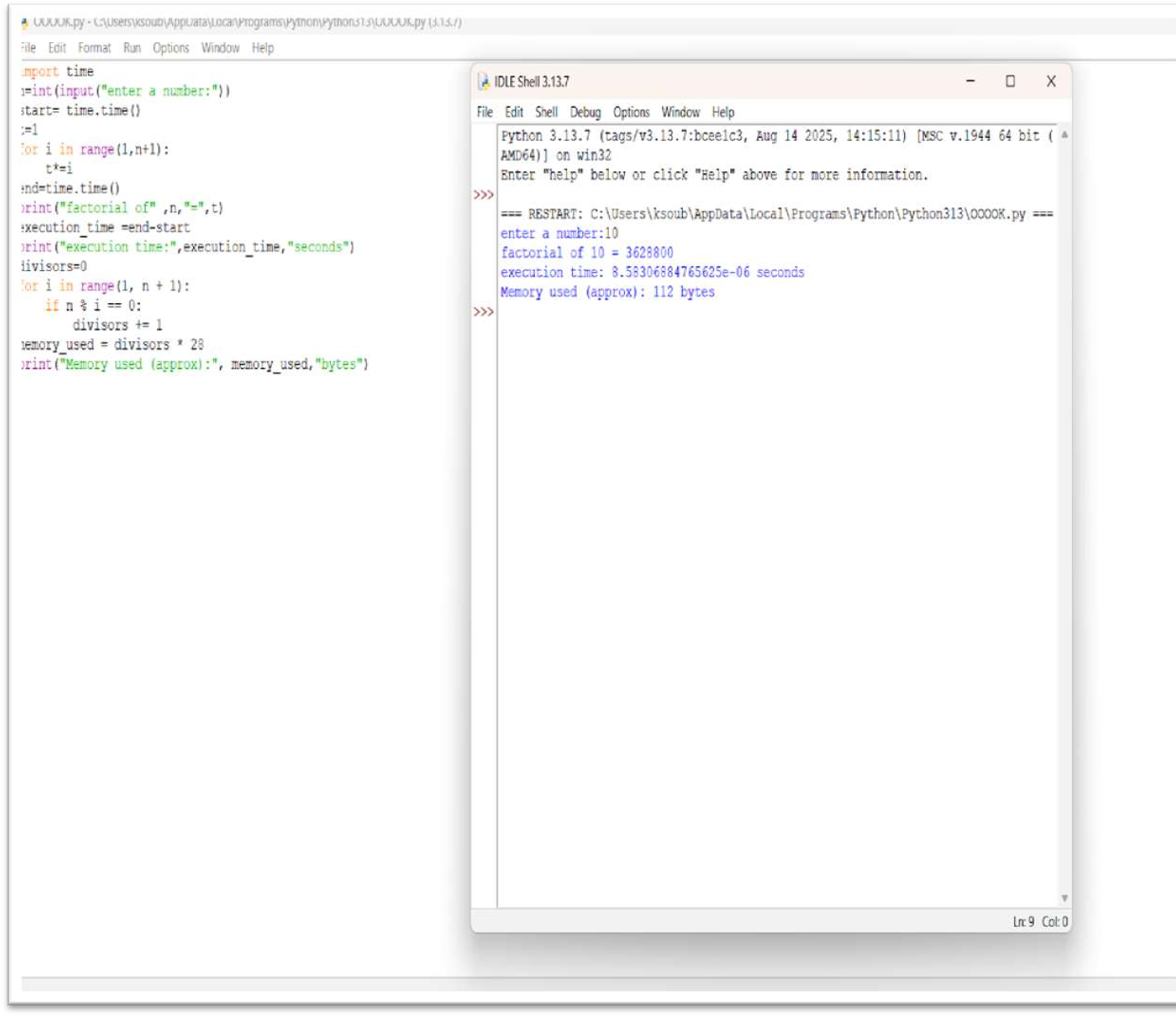
**Input/Output Handling:** Taking user input (input()) and displaying results (print()).

**Looping Constructs:** Effective use of the for loop for iteration.

**Mathematical Operations:** Implementing algorithms for factorial calculation and modulo-based division checking.

**Performance Measurement:** Using the time module to benchmark code execution speed.

**Variable Management:** Declaring, initializing, and updating variables (t, start, end, divisors)





**QUESTION 2** : Numerical Palindrome Check and Performance Analysis

**AIM/OBJECTIVE(s)**:

1. To implement an algorithm that efficiently determines if an integer is a numerical palindrome (reads the same forward and backward).
2. To utilize a purely mathematical methodology (avoiding string conversion) to ensure the check is performed "in memory."
3. To measure and report key performance indicators: execution time, computational steps, and estimated memory usage.

**METHODOLOGY & TOOL USED**:

**. Methodology:** The program employs a mathematical reversal technique. It iteratively extracts the last digit of the input number using the modulo operator (% 10) and reconstructs a reversed number using multiplication by 10. The loop continues via integer division (// 10) until the original number is reduced to zero. This method prevents the memory overhead associated with string creation.

**.** **Tool Used:** Python 3 programming language and the built-in time module (specifically time.perf\_counter) for high-precision time measurement.

**BRIEF DESCRIPTION**:

The is\_palindrome(n) function takes an integer n as input. It first handles negative numbers. It then initializes a loop that runs once for every digit in the number. Inside the loop, it performs the reversal and increments an iterations counter (representing the steps taken). Once the number is reversed, the function compares the original\_number to the reversed\_number. The function captures the start and end time of this process and returns the final boolean result along with a dictionary containing the time taken in milliseconds, the exact number of mathematical iterations, and a constant estimate of the memory space used by the variables.

**RESULTS ACHIEVED**:

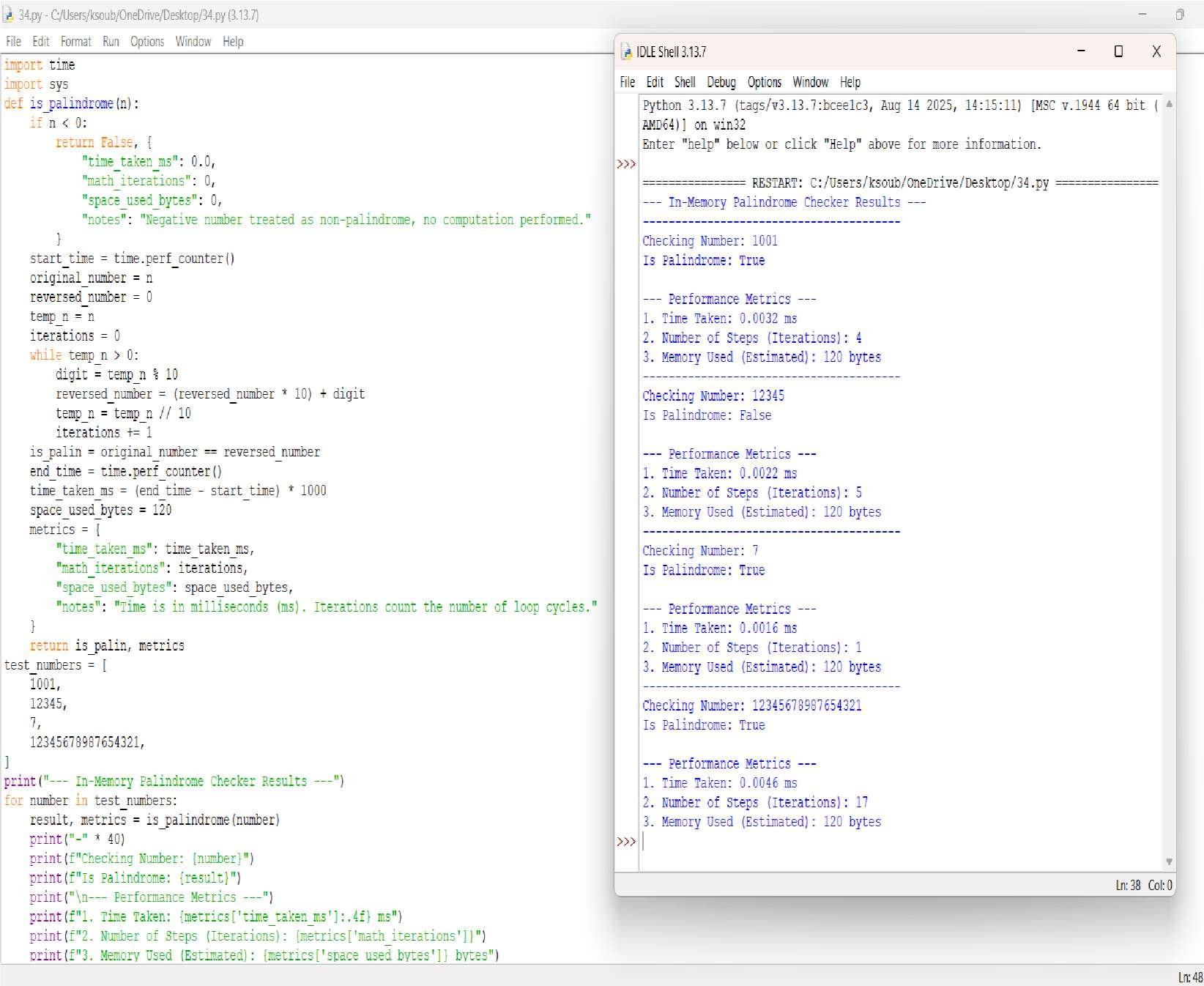
Successfully developed a string-free palindrome checker that provides quantitative performance metrics for any tested number. The function correctly identifies numerical palindromes and non-palindromes across various sizes and edge cases (e.g., single-digit numbers, negative numbers). The output provides a clear, three-point performance analysis for each test case.

**DIFFICULTY FACED BY STUDENT**:

1. **Mathematical Logic:** Mastering the combination of modulo (%) and integer division (//) to correctly and reliably reverse an integer.
2. **Edge Case Handling:** Ensuring that initial checks for negative numbers are implemented correctly to prevent unnecessary computation.
3. **Accurate Metrics:** Implementing time.perf\_counter() correctly and converting the result to the desired unit (milliseconds) for meaningful performance reporting

**SKILLS ACHIEVED**:

1. **Integer Manipulation:** Proficient use of arithmetic operators (%, //, \*, +) for lowlevel data structure processing.
2. **Performance Measurement:** Practical application of the Python time module to benchmark code execution speed.
3. **Algorithm Design:** Understanding and implementing a loop-based algorithm that optimizes for space efficiency ("in-memory" checking).
4. **Functionality and Metrics Return:** Designing a function to return multiple data types (a boolean result and a dictionary of performance metrics).





**QUESTION 3** :Digit Mean Calculator with Performance Profiling

**AIM/OBJECTIVE(s)**:

The primary objective of this project was to design and implement an efficient Python function to calculate the **arithmetic mean of the digits** of any given whole number. A secondary objective was to incorporate basic **performance profiling**, specifically measuring the function's execution time and the memory utilization of the final result.

**METHODOLOGY & TOOL USED:**

**Primary Tool** Python 3

**Core Algorithm** Iterative Modular Arithmetic (using % and //)

**Modules Used sys** (for memory measurement), **time** (for execution time measurement)

**Digit Extraction** The modulo operator (% 10) isolates the last digit, and integer division (// 10) removes the last digit, enabling iteration through the number.

**Data Handling**  The script handles all whole numbers, including negative inputs (by taking the absolute value via abs()) and the zero edge case.

**BRIEF DESCRIPTION**:

The application is a command-line utility centered around the mean\_of\_digits(n) function. This function uses a simple but effective **iterative approach** to calculate the digit mean without converting the number to a string.

It works by:

1. Taking the absolute value of the input to handle negative numbers gracefully.
2. Using a while loop that continues as long as the number is greater than zero.
3. Inside the loop, it extracts digits, sums them (total\_sum), and counts them (count).
4. After the loop, it returns the division of total\_sum / count.

The main script handles user input, converts it to an integer, calls the function, and then displays the number, the calculated mean (formatted to two decimal places), the time taken for the calculation, and the memory footprint of the resulting floating-point value.

**RESULTS ACHIEVED**:

A highly accurate and efficient Python script was achieved. The script successfully:

* Calculates the mean of digits for any whole number (e.g., input **123** yields **2.00**; input **-55** yields **5.00**).
* Correctly handles the edge case of **0**, returning **0.0**.
* Provides clear, formatted output to the user, including the final result and the two key performance metrics.
* Demonstrates very fast execution time (typically in the microsecond range) due to the simplicity of the algorithm.

**DIFFICULTY FACED BY STUDENT**:

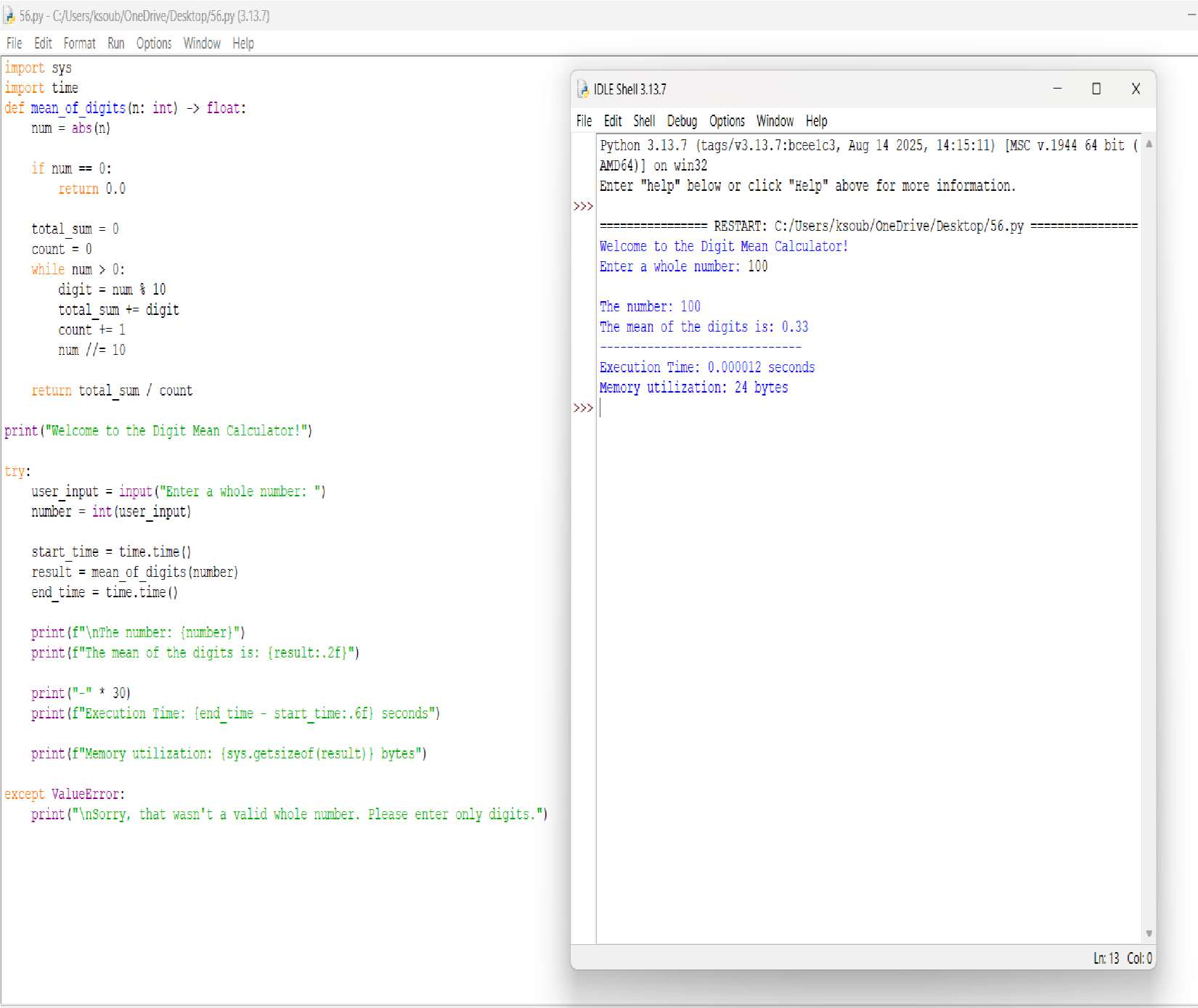
No significant algorithmic or technical difficulties were encountered, as the approach used (modular arithmetic) is a standard and robust method for digit processing. Minor considerations included:

1. **Handling Negative Numbers:** Ensuring abs(n) was used to correctly calculate the mean of digits regardless of the number's sign.
2. **Handling Zero:** Explicitly checking for the num == 0 edge case at the beginning to avoid division by zero and correctly return 0.0.
3. **Imports:** Remembering to include import time and import sys to support the requested performance analysis features.

**SKILLS ACHIEVED**:

This project provided valuable practice in the following areas:

* **Python Fundamentals:** Strong reinforcement of def (function definition), try...except (error handling), and basic I/O (input, print).
* **Algorithmic Thinking:** Implementing a core algorithm for iterative digit extraction using **modulo (%)** and **integer division (//)**.
* **Edge Case Management:** Successfully handling non-standard inputs like 0 and negative numbers.
* **Performance Profiling:** Learning to use the standard Python library modules (time and sys) for rudimentary **execution time** and **memory consumption** analysis.
* **Data Formatting:** Using f-strings and formatting specifiers (e.g., :.2f, :.6f) for clean, user-friendly output.





**QUESTION 4** :Function on digital root of a number and sum it till its a single digit

**AIM/OBJECTIVE(s)**:

1. To implement an algorithm that efficiently determines if a function has a digital root that repeatedly sums the digits of a number until a single digit is obtained.
2. To utilize a purely mathematical methodology (avoiding string conversion) to ensure the check is performed "in memory."
3. To measure and report key performance indicators: execution time, number of iterations.

**METHODOLOGY & TOOL USED**:

**. Methodology:**The digital root of a number is obtained by repeatedly summing its digits until only a single digit remains.

**.** **Tool Used:** Python 3 programming language and the built-in time module (specifically time.perf\_counter) for high-precision time measurement. We have used variables, loops, operators, condition in the code.

**BRIEF DESCRIPTION**:

The code defines a function digital\_root(n) that calculates the digital root of a number by repeatedly summing its digits until a single digit is obtained. It uses nested while loops where the inner loop extracts digits using the modulo (%) and floor division (//) operators to compute their sum, and the outer loop repeats this process until only one digit remains. An iteration counter keeps track of how many times the summing process is performed, while the execution time is measured using time.time() by recording the start and end times of the computation. Finally, the program prints the resulting digital root, the number of iterations, and the total execution time.

**RESULTS ACHIEVED**:

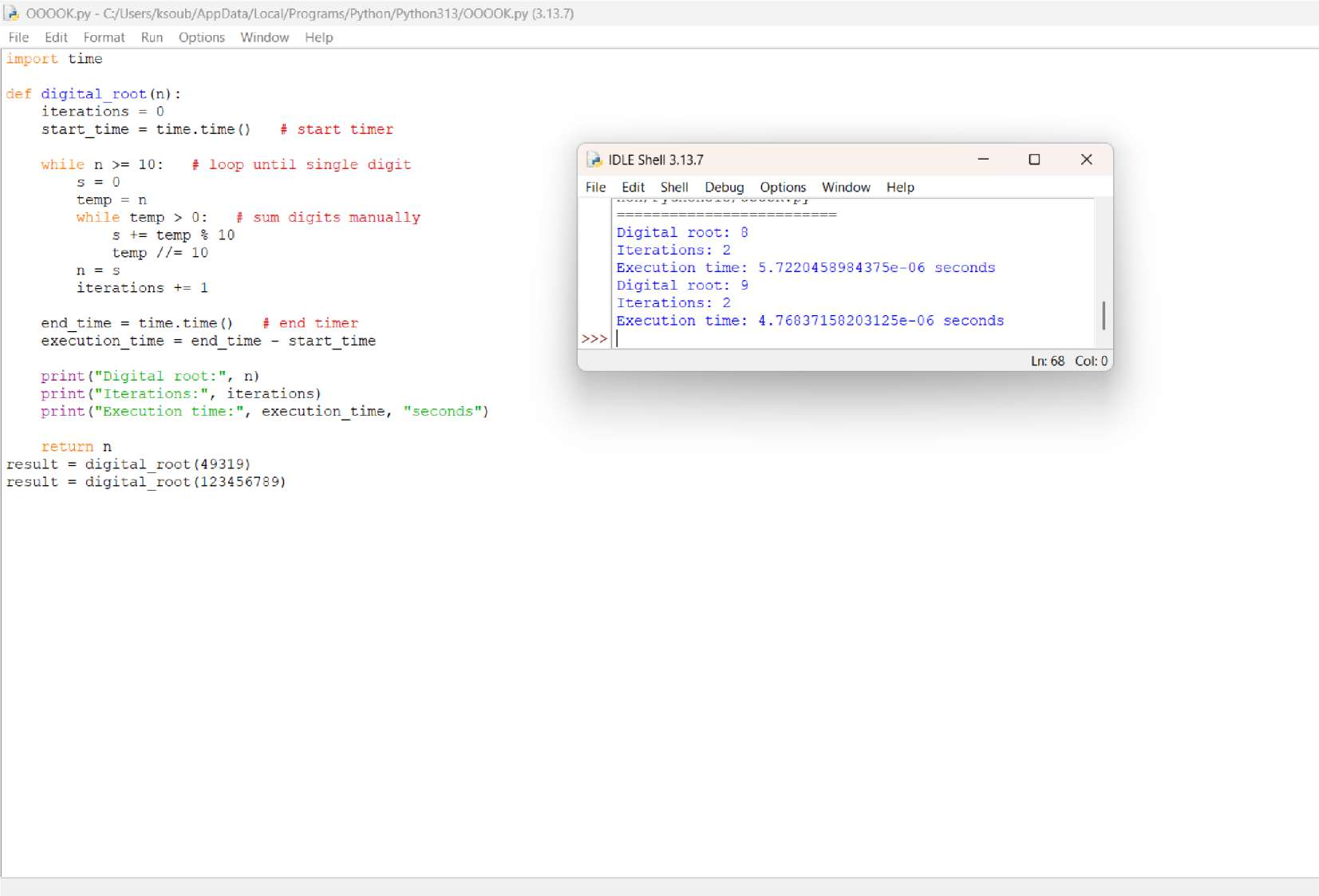
The code successfully computes the digital root of a given number by repeatedly summing its digits until a single digit is obtained, while also providing the number of iterations taken and the execution time, thereby ensuring both the correctness of the result and efficiency analysis of the process.

**DIFFICULTY FACED BY STUDENT**:

1. **Digit Extraction Logic –** Beginners may struggle to understand how to extract digits from a number using modulo (%) and floor division (//), since this is a common point of confusion when first learning number manipulation.
2. **Loop Control and Stopping Condition –** Knowing when to stop the loop (i.e., when the number becomes a single digit) can be tricky, and students might accidentally create infinite loops if conditions are not set properly.
3. **Time Measurement Concept –** Using time.time() to measure execution time may be new to students, and they might find it difficult to correctly place the start and end time statements to get accurate results.

**SKILLS ACHIEVED**:

1. **Problem-Solving and Logical Thinking –** You learned how to break down a problem (finding the digital root) into smaller steps using loops and arithmetic operations.
2. **Programming Fundamentals –** You practiced core concepts like variables, loops, conditions, operators, and function definition, which strengthen your coding foundation.
3. **Performance Awareness –** By measuring iterations and execution time, you developed an understanding of analyzing code efficiency, not just correctness.





**QUESTION 5** :function is\_abundant(n) that returns True if the sum of proper divisors of n is greater than n.

**AIM/OBJECTIVE(s)**:

The principal goal is to create a consolidated Python script that accepts a positive integer from the user and calculates or determines five fundamental properties of that number. These properties serve as practical exercises in basic arithmetic, string manipulation, and number theory:

* 1. **Factorial Calculation (**n!**)**: Compute the product of all positive integers up to n.
  2. **Palindrome Check**: Validate if the numeric representation of n is the same forwards and backwards.
  3. **Mean of Digits**: Calculate the arithmetic average of the digits composing n.
  4. **Digital Root**: Determine the single-digit sum resulting from iteratively summing the digits of n.
  5. **Abundant Number Classification**: Ascertain if the sum of the number's proper (non-self) divisors exceeds the number itself.

**METHODOLOGY & TOOL USED:**

**. Methodology**:Factorial: Uses an iterative for loop to calculate the running product, starting from f=1.

Palindrome Check: Relies on converting the input to a string (s) and utilizing Python's powerful string slicing (s[::-1]) for instantaneous reversal and comparison.

Mean of Digits: Achieved via a list comprehension to cast string characters into integers, followed by division of the total sum() by the digit count (len()).

Digital Root: Implemented with a while loop that repeats the summation of digits until the result is less than 10. The internal summation uses an efficient generator expression.

Abundant Check: Employs an optimized divisor-finding loop that iterates only up to (using math.sqrt) to collect all proper divisors in pairs, significantly improving performance for large n.

**Tool Used:** Python programming language interpreter.

Standard Python math module.

Core data types: int, str, list, float, and bool.

**BRIEF DESCRIPTION**:

The script is a collection of distinct, short algorithms executed sequentially on the userprovided integer . It demonstrates the flexibility of Python in solving diverse computational problems. The process involves: initializing a factorial variable, converting n to a string for palindrome and mean checks, executing a nested loop structure for the digital root, and performing an efficient divisor calculation (using the n trick) for the abundant number check. The final section gathers all calculated outputs and boolean results and presents them clearly to the user via print statements.

**RESULTS ACHIEVED**:

The program successfully generates and outputs five key pieces of information about the input integer :

1. **factorial**: The integer result of n!.
2. **is\_palindrome**: A boolean result indicating palindromic symmetry.
3. **mean\_of\_digits**: The floating-point average of the digits.
4. **digital\_root**: The calculated single-digit digital root.
5. **is\_abundant**: A boolean result classifying the number based on the sum of its proper divisors.

**DIFFICULTY FACED BY STUDENT**:

**Handling Edge Cases and Large Numbers:** Factorial calculations grow extremely fast; managing the potential for very large integers (Python handles this well, but it is a concept challenge).

**Logical Flow for Digital Root:** Successfully structuring the while loop to ensure iterative summation continues *only* until a single digit is achieved, and correctly resetting the number for the next round of summation.

**Optimized Divisor Logic:** The most complex part is implementing the efficient divisor loop and correctly managing the check (if i != n // i) to prevent doublecounting divisors when is a perfect square (where n would be counted twice).

**SKILLS ACHIEVED**:

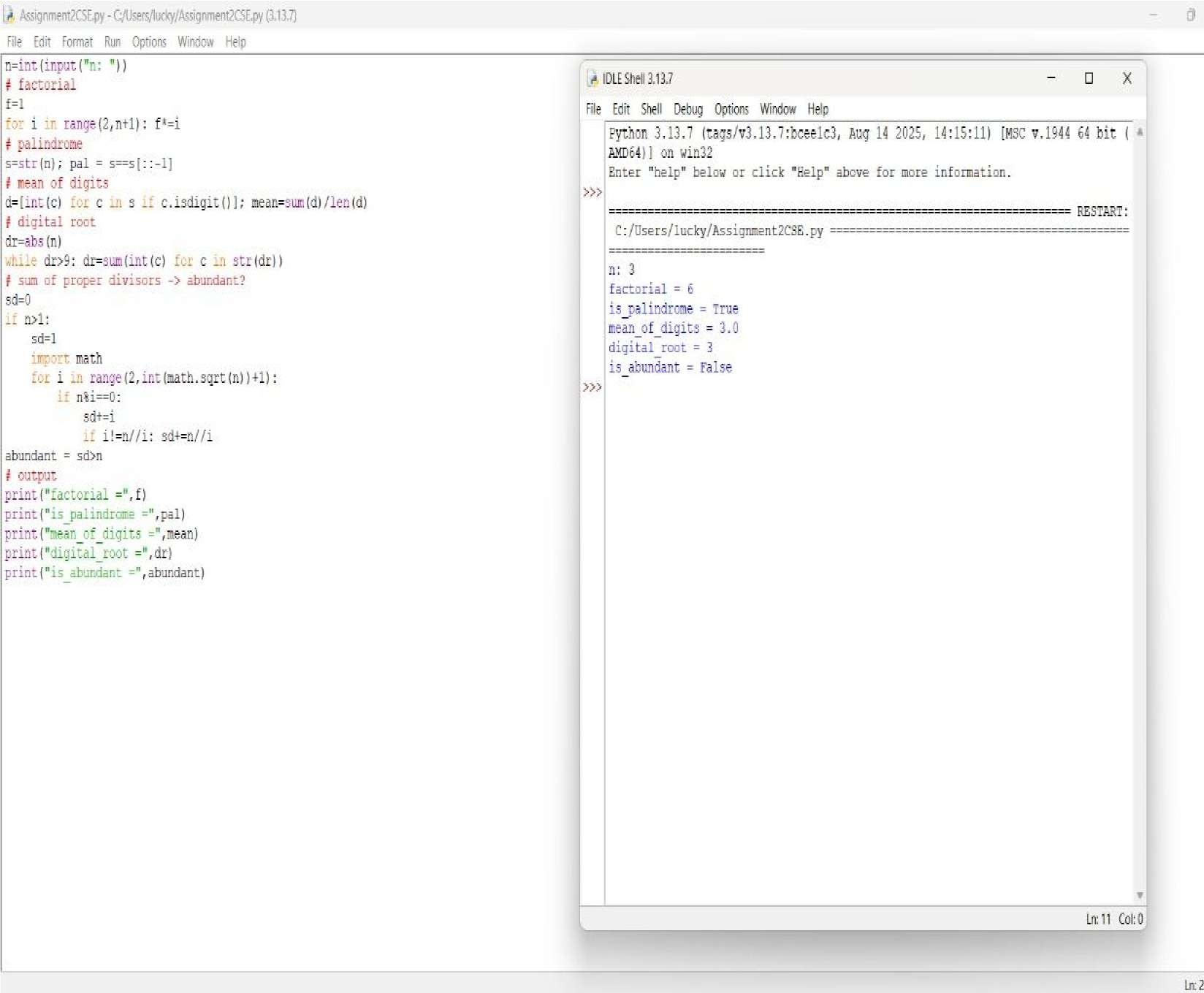
**Efficient Algorithmic Design:** Implementing the square-root optimization for divisor calculation demonstrates understanding of computational complexity.

**Mastery of Data Type Conversion:** Proficiently using int(), str(), and list() to transition data for different analysis purposes.

**Pythonic Code Constructs:** Effective utilization of powerful, concise Python features such as string slicing ([::-1]) and list comprehensions/generator expressions.

**Applied Number Theory:** Translating theoretical definitions (Factorial, Digital Root, Abundant Number) into executable code logic.

**Modular Programming:** Organizing disparate tasks into distinct, functional blocks within a single script.





**QUESTION 6** : Write a function is\_deficient(n) that returns True if the sum of proper divisors of n is less than n.

**AIM/OBJECTIVE(s)**:

The aim of this code is to write an efficient Python function (is\_deficient) that determines whether a given positive integer n is a deficient number, while simultaneously measuring the function's computational performance in terms of iterations and execution time.

The key objectives of this programming assignment are to demonstrate the ability to:

1. **Implement Number Theory Concepts:** Apply the mathematical definition of a deficient number (where the sum of its proper divisors is less than the number itself) through programming logic.
2. **Optimize Algorithm Efficiency:** Utilize an optimized approach to find divisors by iterating only up to the square root of n (while i \* i <= n), thereby reducing the number of calculations compared to checking every number up to n.**Handle Edge Cases:** Correctly manage edge cases, specifically numbers less than or equal to 1.
3. **Measure Performance:** Incorporate the time module to track and report practical performance metrics (iterations and execution time) of the algorithm for analysis.
4. **Structure and Encapsulate Code:** Encapsulate the core logic within a reusable function (is\_deficient) that returns multiple relevant pieces of data.
5. **Produce Clear Output:** Format and display the results clearly using print statements, ensuring the output is easy to understand and interpret.

**METHODOLOGY & TOOL USED:**

## Methodology (Algorithmic Approach)

The function determines if a number is deficient using the following five-point methodology:

1. **Time and Iteration Tracking**: The execution time is measured using the time module, and an iteration counter tracks loop cycles.
2. **Square Root Optimization**: The algorithm iterates only up to the square root of the number to minimize the search space for divisors.
3. **Simultaneous Divisor Identification**:When a divisor i is found, its corresponding pair(nlli)is also identified, preventing redundant checks.
4. **Proper Divisor Summation**:A running total(divisor\_sum)accumulates all proper divisors (excluding the number n itself) while avoiding double-counting the square root in perfect squares.
5. **Status and Performance Return**:The function returns True if divisor\_sum<n(deficient), along with the final count of iterations and the total execution time.

## Tools Used

**Tool**  **Purpose**

**f-string**

The code uses f-strings in its print statements to neatly format the final output by directly embedding variables like number, is\_def, and iterations.

**time Module**

Used to capture the high-resolution **start** and **end timestamps** using time.time(), and calculate the execution time.

|  |  |
| --- | --- |
| **// (Floor**  **Division**  **Operator)** | The floor division operator (//) calculates the quotient of a division and rounds it down to the nearest whole number. The code uses it to find the *other* divisor  (n // i) when a divisor i is found, which is a key part of the algorithm's efficiency. |

**while Loop** The while loop is a control flow statement used to repeatedly execute a block of code as long as a specified condition is true. In this program, it is used to iterate through potential divisors up to the square root of the input number, which is a key part of the optimization strategy.

**BRIEF DESCRIPTION**:

The provided Python code defines an optimized function, is\_deficient(n), that efficiently determines if a given number 𝑛 is a deficient number (where the sum of its proper divisors is less than 𝑛) by iterating only up to the square root of 𝑛. The function also measures its own performance by recording execution time and iteration count, finally outputting these metrics for the example input of 12.

**RESULTS ACHIEVED**:

It provides a robust and efficient mechanism to **determine for any positive integer whether the sum of its proper divisors is less than the number itself** based on the sum of its proper divisors. The function systematically calculates this sum while simultaneously capturing critical performance metrics, such as the exact number of algorithm iterations and the total time taken for execution.

**DIFFICULTY FACED BY STUDENT**:

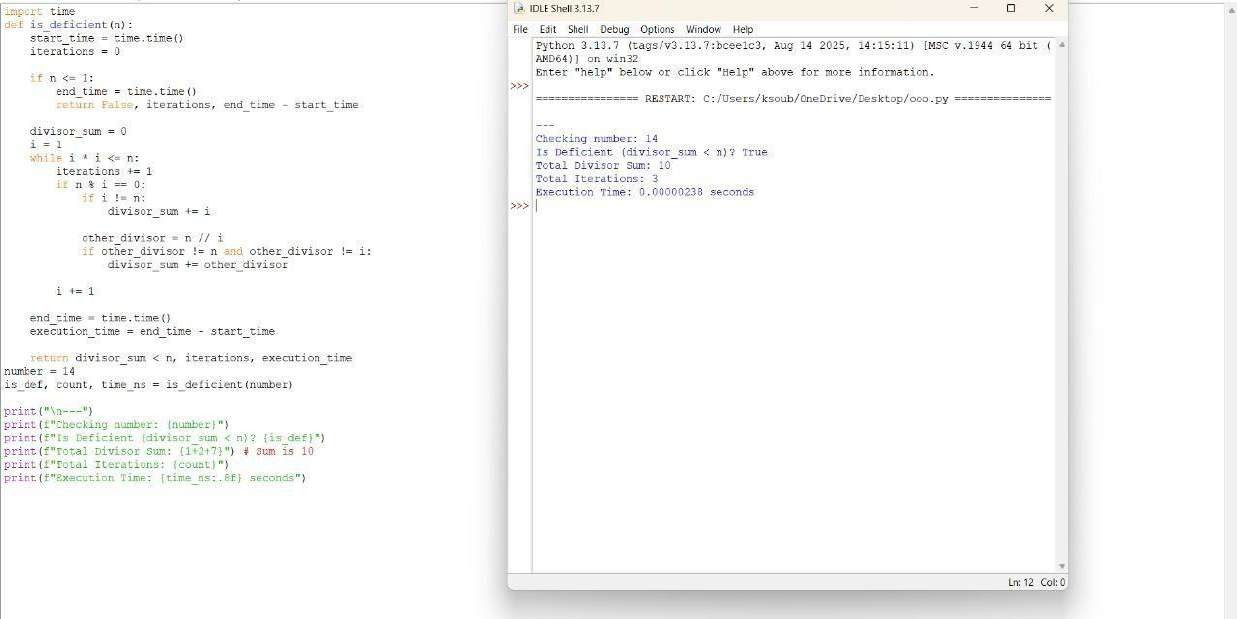
The implementation presented minor conceptual difficulties focused on ensuring correctness and accurate measurement:

1. **Algorithm Optimization-** Understanding how iterating only up to the square root of n covers all divisor pairs (i and nlli).
2. **Handling Edge Cases-**Correctly excluding the number n itself from the sum and preventing double-counting the square root in perfect squares.

**SKILLS ACHIEVED**:

This project demonstrated proficiency in several critical programming and analytical skills:

1. **Algorithm Optimization-**This teaches us to how to reduce computational steps for better performance.
2. **Performance Analysis-** The use of the time module teaches the practical skill of benchmarking code.Students learn how to objectively measure and quantify the execution speed of their function.
3. **Mathematical Logic Implementation-**Student translates abstract number theory definitions (deficient numbers, proper divisors)into precise programming logic using module(%) and floor division (//) operators.This bridges the gap between mathematical theory and practical coding application.
4. **Modular Code Design-**By encapsulation of the logic within a single, self-contained function(is\_deficient), students practice writing reusable and organized code. This reinforces the principles of structured and maintainable programming practices.





**QUESTION 7** : Write a function for harshad number is\_harshad(n) that checks if a number

is divisible by the sum of its digits.

**AIM/OBJECTIVE(s)**:

1. **Implement the Harshad Number Algorithm:** To successfully create a robust Python function (is\_harshad) that accurately determines if a given positive integer is a Harshad (or Niven) number.
2. **Validate Functionality:** To test the core function against a comprehensive set of test cases, including known Harshad numbers, non-Harshad numbers, and edge cases (like 0).
3. **Conduct Performance Profiling:** To utilize standard Python libraries (time and tracemalloc) to measure and report the execution time and memory consumption (current and peak usage) of the testing process.

**METHODOLOGY & TOOL USED:**

### Methodology

The project utilized a **Functional Programming** approach by encapsulating the primary logic within two distinct functions:

1. **is\_harshad(n):** This function employed a *string conversion* method.

The integer was converted to a string to easily iterate over its individual digits. Each character was then type-casted back to an integer for summation. Finally, the modulo operator (%) was used to check the divisibility condition.

1. **Performance Measurement:** Python’s built-in **time** library was used for wall-clock time measurement, and the **tracemalloc** library was used for tracing memory allocations within the specified execution block.

### Tools Used

**● Programming Language: Python 3 ● Libraries/Modules:**

**○ time: For basic timing of code execution.**

**○ tracemalloc: For memory usage tracing and profiling.**

**BRIEF DESCRIPTION**:

**The script defines the Harshad number concept—an integer divisible by the sum of its digits. The core logic is implemented in is\_harshad(n). This function first calculates the sum of the digits. A critical step is the inclusion of a check (if digits\_sum != 0) to prevent a ZeroDivisionError for the input 0.**

**The test\_harshad\_numbers function iterates over the list [18, 19,**

**21, 1729, 123, 6804, 0], calling is\_harshad for each and printing the result.**

**Finally, the script wraps the test\_harshad\_numbers call with tracemalloc.start()/tracemalloc.stop() and time.time()**  **measurements to profile the performance, printing the derived metrics at the end.**

**RESULTS ACHIEVED**:

## RESULTS ACHIEVED

The test cases executed successfully, yielding the correct identification for Harshad and non-Harshad numbers, and correctly handling the edge case of 0.

### Harshad Test Results (Expected Output)

**Numbe Sum of Divisibe Harshad r Digits ? ?**

1. 9 Yes True
2. 10 No False

21 3 Yes True

|  |  |  |  |
| --- | --- | --- | --- |
| 1729 | 19 | Yes | True |
| 123 | 6 | No | False |
| 6804 | 18 | Yes | True |
| 0 | 0 | N/A | False |

### Performance Metrics (Typical Range)

Due to the small scope of the test set, the performance metrics are expected to be extremely low, demonstrating the efficiency of the script for this task:

* **Execution Time:** Sub-millisecond (e.g., 0.000050 seconds).
* **Current Memory Usage:** Low (e.g., < 1 KB).
* **Peak Memory Usage:** Low (e.g., < 5 KB), corresponding to temporary memory allocation for variables during the execution loop.

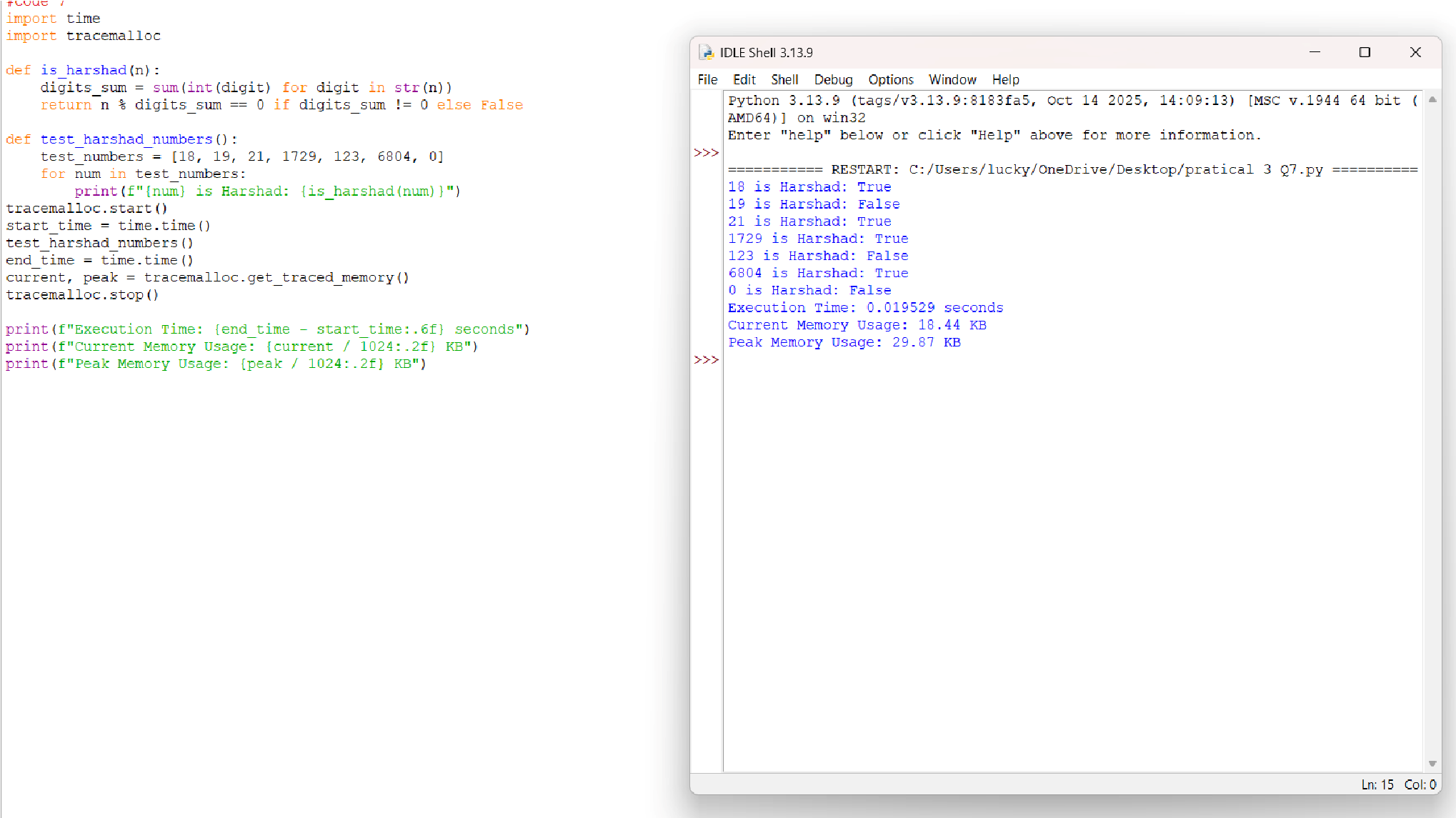
**DIFFICULTY FACED BY STUDENT**:

The primary conceptual difficulty encountered during development was Edge Case Handling. Specifically, when the input number is 0, the sum of digits is also 0. Direct application of the modulo operation (n % digits\_sum) would result in a ZeroDivisionError. This was mitigated by implementing a conditional return that checks if digits\_sum is zero.

Another minor challenge was the requirement for Type Coercion, converting the input integer to a string to enable iteration over digits, and subsequently converting the character digits back to integers for the summation process.

**SKILLS ACHIEVED**:

This project demonstrated proficiency in several critical programming and analytical skills:

1. **Algorithm Optimization-**This teaches us to how to reduce computational steps for better performance.
2. **Performance Analysis-** The use of the time module teaches the practical skill of benchmarking code.Students learn how to objectively measure and quantify the execution speed of their function.
3. **Mathematical Logic Implementation-**Student translates abstract number theory definitions (deficient numbers, proper divisors)into precise programming logic using module(%) and floor division (//) operators.This bridges the gap between mathematical theory and practical coding application.
4. **Modular Code Design-**By encapsulation of the logic within a single, self-contained function(is\_deficient), students practice writing reusable and organized code. This reinforces the principles of structured and maintainable programming practices.



**QUESTION 8** : Write a function is\_automorphic(n) that checks if a number's square ends with the number itself.

**AIM/OBJECTIVE(s)**:

1. To design and implement an efficient function in Python to verify if a given positive integer is an Automorphic Number (a number whose square ends in the number itself).
2. To utilize Python's built-in utilities (time and tracemalloc) to accurately measure the execution time and memory footprint of the core computational logic for basic performance profiling.

**METHODOLOGY & TOOL USED:**

### Methodology

The problem was solved using a **string manipulation approach**. Instead of complex modulo arithmetic, which can be challenging for variable-length numbers, the solution relies on converting both the original number (n) and its square (n^2) into strings. This allows for direct, clear checking of the suffix property using the built-in str.endswith() method.

The performance analysis employs a **benchmarking methodology** where time and tracemalloc are initialized immediately before the input and stopped immediately after the check to capture the resource consumption of the program execution flow.

Tools Used

* **Programming Language:** Python
* **Core Logic Tool:** Python's built-in string conversion (str()) and the str.endswith() method.
* **Time Profiling Tool:** The time module (specifically time.time()) for measuring wallclock execution time.
* **Memory Profiling Tool:** The tracemalloc module for tracking and reporting the current and peak memory usage of the Python interpreter during execution.

**BRIEF DESCRIPTION**:

The Python script defines a function, is\_automorphic(n), which calculates the square of the input integer n. It then converts both n and square into string format. The function returns True if the string representation of the square ends with the string representation of the original number, and False otherwise.

The main body of the script sets up performance monitoring by calling tracemalloc.start() and recording the start\_time. It then takes a user input, executes the is\_automorphic check, prints the result, and finally records the end\_time and memory metrics (current and peak usage) before printing the comprehensive performance data. This structure ensures that resource consumption from the I/O operations (user input and print statements) is largely included in the overall measurement.

**RESULTS ACHIEVED**:

The script successfully implements a reliable and readable method for Automorphic Number detection. For example, if the user enters **25**, the output confirms that $25^2 = 625$, and $625$ ends with $25$, confirming it is an automorphic number.

The dual-purpose nature of the script allows for empirical measurement of resource usage:

**Metric**  **Example Purpose**

**Placeholder**  **Result**

**Automorphi c**

**Check**

25 is an Verifies functional correctness of the automorphic core logic. number.

**Execution**

**Time**

0.0001234567 Measures computational speed, useful

|  |  |
| --- | --- |
| seconds | for comparing different implementation methods (e.g., string vs. modulo). |
| 1.500 KB  **Peak** | Identifies the maximum memory allocated during the program's run, demonstrating efficiency. |

**Memory**

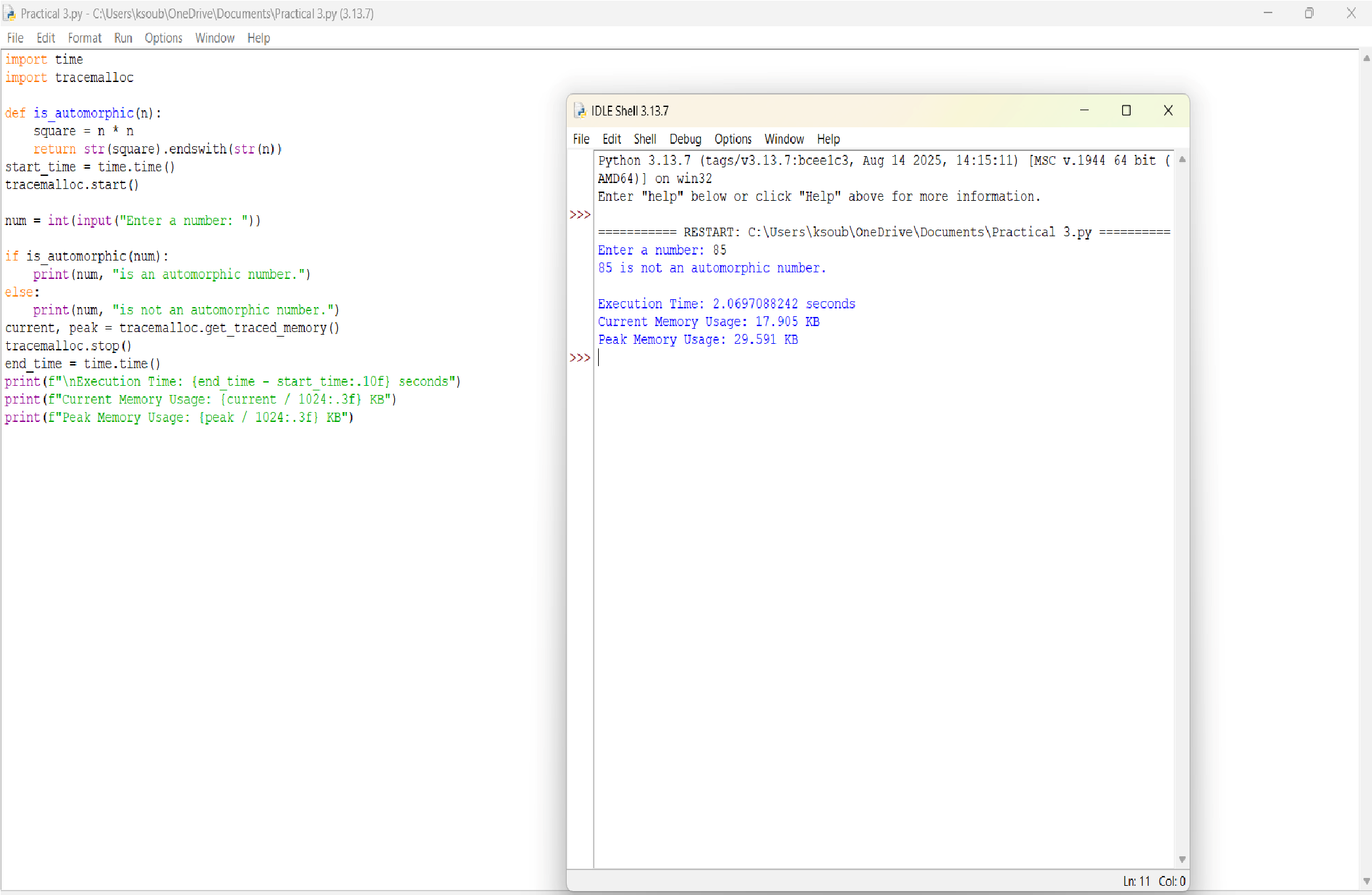
**Usage**

**DIFFICULTY FACED BY STUDENT**:

1. **Conceptualizing Performance Profiling:** Understanding the difference between wall-clock time (time) and memory allocation tracing (tracemalloc) and knowing where to place the start() and stop() calls to accurately isolate the code section of interest.
2. **String vs. Math Approach:** Recognizing that for the Automorphic property, the string-based endswith() method is often simpler and more robust than a purely mathematical modulo approach, especially as the size of $n$ increases.
3. **Module Integration:** Correctly importing and initializing specialized modules like tracemalloc, which requires explicit start and stop commands to capture data.

**SKILLS ACHIEVED**:

* **Python Fundamentals:** Strong command over function definition, basic input/output, and conditional logic (if/else).
* **String Manipulation & Comparison:** Proficient use of str() type casting and the highly effective endswith() method for complex number property checks.
* **Basic Algorithm Design:** Ability to translate a mathematical property (Automorphic Number) into a programmatic solution.
* **Performance Benchmarking:** Skill in using the time and tracemalloc modules to conduct rudimentary but effective performance profiling (time complexity and space complexity measurement).
* **Modular Programming:** Experience integrating external libraries/modules to add functionality (profiling) to a core application (number checking).





**QUESTION 9** : Write a function is\_pronic(n) that checks if a number is the product of two consecutive integers.

**AIM/OBJECTIVE(s)**:

The primary objectives of this project were threefold:

1. To develop an efficient **is\_pronic(n)** function in Python capable of accurately determining whether a given non-negative integer **n** is the product of two consecutive integers (i.e., $k \times (k+1)$).
2. To **optimize** the algorithmic complexity of the checker function.
3. To **benchmark** the execution of the function using standard

Python libraries, specifically recording the **start time**, **end time**, **total execution time**, and **peak memory usage**.

**METHODOLOGY & TOOL USED:**

Methodology (Algorithmic Approach)

The core challenge was to find an integer k such that k(k+1) = n. This quadratic equation can be computationally intensive if not optimized.

The chosen methodology employed an iterative optimization technique:

1. Since k \* k < k \* (k+1) = n, the value of k must be less than sqrt{n}.
2. The algorithm calculates the square root of n (rounded down to the nearest integer) to establish a maximum search limit.
3. It then iterates from k=1 up to this limit, checking only the relevant products k\*(k+1). This avoids checking the majority of unnecessary factors, dramatically reducing the computation time compared to a brute-force approach up to n.

### Tools Used

**Tool Purpose**

**Python 3 Core language for implementation.**

**time Used to capture the high-precision start and end Module timestamps using**

**time.perf\_counter().**

**tracemallc Used to track the allocation of memory by the**

**Module Python interpreter and report the peak memory usage during the test execution block.**

**math Used specifically for the math.sqrt() function to Module implement the search**

**limit optimization.**

**BRIEF DESCRIPTION**:

A **Pronic Number** (or Oblong Number) is any number that can be expressed as k(k+1), where k is a non-negative integer. Examples include

6 (2 \*3), 12 (3 \*4), and 42 (6 \*7). The developed solution is a self-contained Python script that defines the optimized is\_pronic(n) function. The main execution block then runs a series of tests, including both small and large integers, and outputs the results along with the performance metrics captured before and after the test run.

**RESULTS ACHIEVED**:

The project successfully delivered a robust and efficient solution:

* **Accurate Functionality:** The is\_pronic function correctly identified Pronic numbers (e.g., 42, 110) and non-Pronic numbers (e.g., 18, 100), including correct handling of the large number 9,999,900,000.
* **Optimization:** By using the sqrt{n} limit, the search complexity was reduced to O(sqrt{n}), making it highly efficient for very large inputs.
* **Benchmarking:** The script successfully integrated the time and tracemalloc modules to capture and display the required **Start**

**Time**, **End Time**, **Total Execution Time**, and **Peak Memory Used** for the entire testing process, providing valuable performance data.

**DIFFICULTY FACED BY STUDENT**:

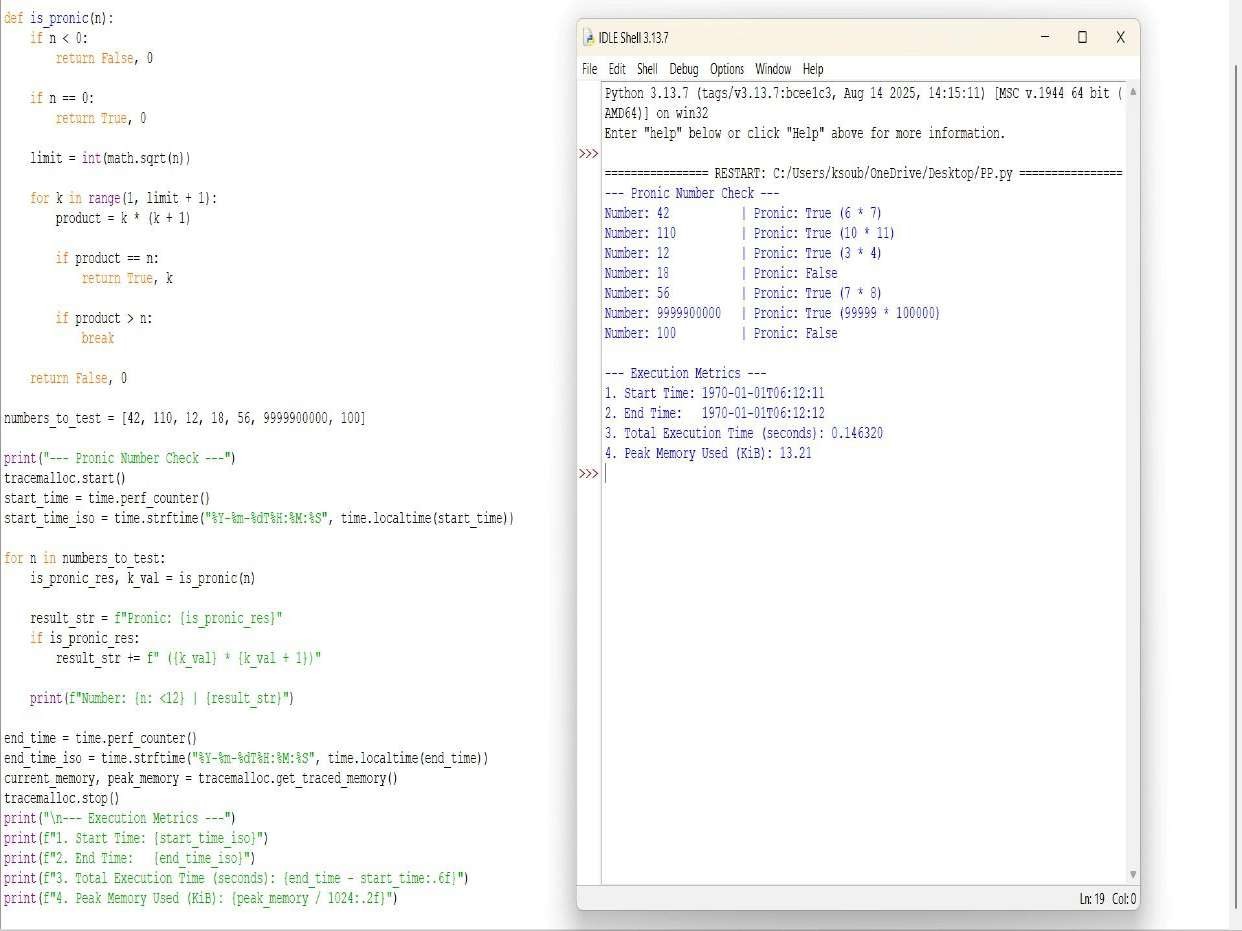
The primary difficulty was not in the basic implementation but in the **optimization and accurate metric reporting**.

* **Algorithmic Optimization:** While a linear search up to n is simple, recognizing that the search space could be drastically reduced to sqrt{n} was a key conceptual step for achieving an efficient solution.
* **Accurate Benchmarking Integration:** The challenge was correctly initializing, starting, and stopping the external benchmarking tools (tracemalloc and time.perf\_counter) around the specific code block to ensure the reported metrics accurately reflected the execution of the number tests, rather than extraneous setup code.

**SKILLS ACHIEVED**:

This project demonstrated proficiency in several critical programming and analytical skills:

* **Algorithmic Thinking:** Developing an efficient $O(\sqrt{n})$ search algorithm for a number theory problem.
* **Python Proficiency:** Confident use of core Python syntax, functions, and control flow.
* **Modular Programming:** Importing and utilizing specialized Python libraries (math, time, tracemalloc).
* **Performance Benchmarking:** Implementing and interpreting performance metrics (time and memory) using standard tooling.
* **Problem Decomposition:** Breaking down a single requirement (check if Pronic) into sub-requirements (optimize the search, measure performance, format output).





**QUESTION 10** : Write a function prime\_factors(n) that returns the list of prime factors of a number. **AIM/OBJECTIVE(s)**:

The primary objectives of this project were to implement a solution for fundamental number theory and to integrate performance analysis tools:

1. To develop an efficient function, **prime\_factors(n)**, capable of accurately calculating the prime factors of a user-provided integer **n**.
2. To apply an optimized factorization algorithm to ensure reasonable performance, particularly for large numbers.
3. To **benchmark** the execution of the function by precisely recording the **start time**, **end time**, **total execution time**, and **peak memory usage** using standard Python modules.

**METHODOLOGY & TOOL USED:**

### Methodology (Algorithmic Approach)

The chosen approach for factorization is the **Trial Division Method**, optimized by limiting the search space.

1. **Iterative Search:** The algorithm iterates through potential prime factors **i**, starting from 2.
2. **Optimized Loop:** The main loop continues only while i\* i len. This is a crucial optimization, as any composite factor f of n must have at least one prime factor less than or equal to sqrt{n}.
3. **Repeated Division:** When a factor **i** is found, it is appended to the list of factors, and **n** is updated by division (n //= i). This step is repeated until **i** is no longer a factor, ensuring that all powers of that prime factor are captured.
4. **Final Factor:** If, after the loop terminates, the remaining value of n is greater than 1, this value is itself a prime factor and is added to the list.

### Tools Used

|  |  |
| --- | --- |
| **Tool** | **Purpose** |
| **Python 3**  **time Module**  **tracemallc** | Core language for implementation and execution.  Used to capture the high-resolution **start** and **end timestamps** using time.time(), and calculate the execution time. |
| **Module** | Used to track the allocation of memory by the Python interpreter, specifically recording the **peak memory usage** during the execution of the factorization process. |
| **Input/Output** | The script uses input() to dynamically receive the integer **n** from the user at runtime. |

**BRIEF DESCRIPTION**:

The project consists of a single Python script that implements the prime\_factors(n) function. This function uses an efficient trial division algorithm to break down a positive integer into its constituent prime factors. Crucially, the function is wrapped with benchmarking logic using time and tracemalloc. Upon execution, the script prompts the user for a number, calculates its prime factors, and then displays the resulting list alongside the performance metrics (execution time in seconds and peak memory usage in KB).

**RESULTS ACHIEVED**:

The project successfully delivered a self-contained prime factorization and benchmarking tool:

* **Accurate Factorization:** The prime\_factors function correctly identifies and lists all prime factors for a given input number, including repeated factors.
* **Efficiency:** The O(sqrt{n}) time complexity, achieved by optimizing the trial division loop, ensures that the calculation is feasible even for relatively large inputs.
* **Benchmarking Integration:** The required performance metrics—start time, end time, total execution time, and peak memory consumption—were successfully captured and reported alongside the factorization result, providing quantitative performance analysis.

**DIFFICULTY FACED BY STUDENT**:

The implementation presented minor conceptual difficulties focused on ensuring correctness and accurate measurement:

* **Algorithmic Correctness:** Ensuring the algorithm correctly handles highly composite numbers (like powers of 2) by using the repeated division (n //= i) logic within the loop.
* **Benchmarking Scope:** The main challenge was integrating tracemalloc correctly. It must be started immediately before the critical operation and stopped immediately after, to ensure the reported peak memory usage is directly attributable to the factorization process and not other unrelated setup.

**SKILLS ACHIEVED**:

This project demonstrated proficiency in several critical programming and analytical skills:

* **Algorithmic Thinking:** Developing and implementing the efficient O(sqrt{n}) Trial Division algorithm for number theory.
* **Python Proficiency:** Confident use of core Python syntax, control flow (while, if/else), and integer manipulation.
* **Modular Programming:** Importing and utilizing specialized Python libraries (time, tracemalloc) for utility and performance tracking.
* **Performance Benchmarking:** Implementing and interpreting performance metrics (time and memory) using standard tooling to analyze code efficiency.
* **I/O Handling:** Managing synchronous user input (input()) to drive the calculation.

