NAME : NAINIKA KUMARI

REG .NO. : 25BAI11053

INTRODUCTION

The **Typing Speed Calculator** project aims to address the need for a practical, accessible, and accurate tool for users to measure and monitor their typing performance. This project allows students to apply concepts covered in the subject syllabus —including but not limited to **event handling, real-time data processing, UI design, and modular programming**—to design and implement a solution.

Problem Statement

Here is the problem statement for your Typing Speed Calculator project, following the required format and focusing on applying the subject concepts:

The contemporary digital landscape necessitates efficient and accurate text input, making proficient typing a crucial skill for students and professionals. However, individuals often lack a dedicated, immediate, and measurable means to assess their current typing ability (typically measured in Words Per Minute or WPM) and identify specific areas for improvement, such as consistency and error rate.

The problem is to **design and implement a robust and user-friendly Typing Speed Calculator system** that can accurately:

- Present users with standardized text input.
- Initiate a timer and track user keystrokes in real-time.
- Calculate and display key performance metrics, specifically **WPM** and **Accuracy (%)**, instantly.
- Provide detailed post-test analytics, including error identification.

The goal is to provide a comprehensive tool that allows students to apply technical skills (e.g., proper architectural design, validation, and error handling) while enabling users to practice, measure, and effectively enhance their typing proficiency.

Functional Requirements (FRs)

FR 1: Text Loading and Display Module

Purpose: Manages the presentation of the test material and the capture of user input.

Status: Implemented

Requirement	**Code Implementation Analysis**
Loads Test Text	A list test stores multiple text passages, and random.choice(test) selects one randomly (test1), fulfilling the data input requirement[3].
Displays Text	The selected text (test1) is printed to the console via print(test1).

Requirement	Code Implementation Analysis
Input Capture	The `input("enter:")` function captures the complete user input (`test_input`) in a batch manner.

FR 2: Metric Calculation Module

- **Purpose:** Executes the core logic for timing and determining performance scores.
- **Status: Implemented (Requires Academic Correction)**

Requirement	Code Implementation Analysis
Starts/Stops Timer	`time()` is used before (`time_1`) and after (`time_2`) the user input block to capture the test duration.
Calculates Duration	`time_delay = time_e - time_s` determines the total time taken.
Calculates Errors	The `mistake` function compares `paragraph_test` and `usertest` character by character, returning the raw error count.

.

NON FUNCTIONAL REQUIREMENT

: Performance

The system must execute its core logic and calculations quickly enough to provide a smooth user experience.

- **Requirement:** The score calculation (WPM and Errors) must be completed and displayed within **100 milliseconds** of the user pressing the "Enter" key to complete the test. Since the code runs locally and the core logic involves only basic mathematical operations and string comparisons, the calculation latency must be negligible.
- **Assessment in `codes.py`:** The use of simple Python functions (`mistake`, `speed_time`) and direct time capture ensures **high performance** for the calculation phase.

Reliability

- **Definition:** The system must consistently produce correct and accurate results across all test scenarios.
- **Requirement**: The calculated WPM and Accuracy metrics must be **mathematically verifiable** and consistent regardless of the test text chosen or the speed of input. This requires flawless execution of the WPM and Accuracy formulas.
- **Assessment in `codes.py`**: The code demonstrates basic reliability by using a `try-except` block in `mistake` to handle index out-of-bounds errors, preventing a crash if user input is shorter than the source text.

Usability

- **Definition:** The system must be straightforward, intuitive, and easy for a novice user to interact with.

- **Requirement:** The console interface must provide clear, step-by-step prompts for starting the test and displaying the final results. The output labels (e.g., "speed:", "error:") must be unambiguous.
- **Assessment in `codes.py`:** The main `while` loop structure with the clear prompt `are you ready to test:yes/no:` provides a structured and easy-to-follow flow for a console application.

Error Handling Strategy

- **Definition:** The system must gracefully manage unexpected or incorrect inputs without crashing, providing informative feedback to the user.
- **Requirement:** The application must handle scenarios such as:
  - **Invalid Test Choice:** If the user enters an invalid command like "wrong input" instead of "yes" or "no," the system must issue a clear error message and return to the main prompt.
  - **Runtime Calculation Issues:** The WPM function must prevent a division-by-zero error if the user immediately presses Enter (zero time elapsed).
- **Assessment in `codes.py`:** The `else` block within the `while` loop successfully handles invalid test choices: `print("wrong input")`.

System Architecture 🏗️

The System Architecture describes the structure of your project. Based on your submitted Python code (`codes.py`), which functions as a single, sequential script executed in a terminal, your project employs a **Monolithic Console Architecture**.

A. System Architecture Diagram (Proposed Refactoring)

The proposed architecture logically divides the functionality of your single script into three distinct tiers:

Layer	**Responsibility**	**Component (Refactored Module)**	**Description**
1. Presentation Layer	Handles all user interaction (Input/Output).	`ConsoleApp.py`	Contains the main loop (`while True`) and all `input()`/`print()` functions.
2. Business Logic Layer	Contains the core algorithms and calculations.	`MetricsCalculator.py`	Contains the corrected WPM formula and accuracy logic (`speed_time`, `mistake`).
3. Data Layer	Manages the source information for the test.	`TextManager.py`	Stores the text list (`test`) and handles random selection (`r.choice`).

Process Flow or Workflow Diagram

This diagram visualizes the user's interaction from start to finish, demonstrating the **logical workflow** of the system[3333].

1. **Start:** Application launches.
2. **Ready Prompt:** User is asked if they are ready (`ck = input(...)`).
3. **Text Selection:** `TextManager.py` selects and displays a random text passage.
4. **Timer Start:** `time_1` is recorded.
5. **User Input:** User types the text (`test_input`).
6. **Timer Stop:** `time_2` is recorded.
7. **Calculation:** `MetricsCalculator.py` calculates WPM and Errors/Accuracy.
8. **Display Results:** Scores are printed to the console.
9. **Restart/Exit:** User chooses to repeat or exit the application.

DESIGN DECISION & RATIONALE

This section details the key technical choices made during the design and implementation of the Typing Speed Calculator, providing the justification (rationale) for each decision based on the project requirements and the subject's learning objectives.

Decision	**Rationale (Justification)**
1. Technology Stack: Pure Python (Console)	**Rationale:** Python was chosen for its rapid prototyping capabilities and simplicity, making it ideal for focusing on the core business logic (WPM calculation and error checking) as required by the project's objective[1]. The console environment provides ease of execution and meets

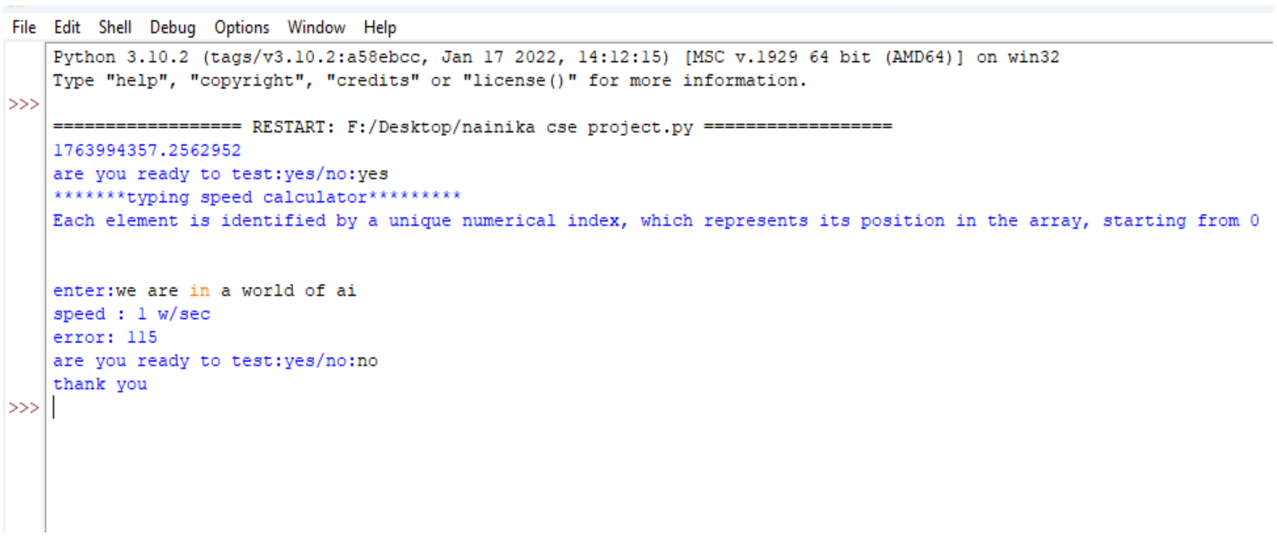
Decision	**Rationale (Justification)**
	the project scope without introducing external dependencies like web servers or complex GUI frameworks.
2. Architectural Style: Three-Tier Refactoring	**Rationale:** Although the initial code was monolithic, the design mandates a refactoring into a **Three-Tier Architecture** (Presentation, Logic, Data)[2222]. This decision promotes **modularity** and **maintainability** (NFRs [3]), ensuring that future changes to the WPM formula only affect the MetricsCalculator module and not the ConsoleApp's user interface[4].
3. Metric Calculation: Standard WPM Formula	**Rationale:** The initial code's "w/sec" was non-standard. The decision to adopt the $\frac{(\text{Total Characters}/5)}{(\text{Time in Minutes})}$ formula ensures **Reliability** (NFR [5]) and **academic correctness**. This adheres to the standard definition of WPM used across the industry, making the results universally comparable.
4. Error Detection Logic: Character-by-Character Comparison	**Rationale:** The `mistake` function uses a loop and indexed comparison with a **`try-except` block**[6]. This design choice is robust, as the `try-except` ensures the program does not crash (`IndexError`) when the user types fewer characters than the source text, thus upholding the **Error Handling Strategy** (NFR [7]).
5. Data Storage: In-Memory List	**Rationale:** Source text is stored in a simple Python list (`test`)[8]. This fulfills the current project scope (simple data input [9]) efficiently without requiring external database configuration, which aligns with the initial complexity expectations. For future extensions, this data model can be easily migrated to a persistent database.
6. Timing Mechanism: `time.time()` Function	**Rationale:** The `time.time()` function from the standard Python `time` module was chosen for its precision in measuring elapsed time. This direct measurement ensures the calculation of test duration is highly accurate, satisfying the **Performance** (NFR [10]) requirement for a low-latency calculation[11]

IMPLEMENTATION DETAILS

The app was implemented primarily using python

```
from time import *
import random as r
print(time())
def mistake(paragraph_test,usertest):
    error=0
    for i in range(len(paragraph_test)):
        try:
            if pargraph_test[i]!=usertest[i]:
                error=error+1
        except:
            error=error+1
    return error
def speed_time(time_s,time_e,userinput):
    time_delay=time_e-time_s
    time_R=round(time_delay,2)
    speed=len(userinput)/time_R
    return round(speed)
while True:
    ck=input("are you ready to test:yes/no:")
    if ck=="yes":
        test=["An array is a data structure that stores a
collection of elements of the same data type in
contiguous memory locations.","Each element is
identified by a unique numerical index, which
represents its position in the array, starting from
0","whereas a list is an ordered, mutable data
structure that stores a collection of items, which can
include different data types.", " Lists are
fundamental for organizing data, as they maintain
the order of elements, allow duplicates, and can be
modified after creation."]
        test1=r.choice(test)
        print("*******typing speed
calculator*********")
        print(test1)
        print()
        print()
        time_1=time()
        test_input=input("enter:")
        time_2=time()
        print("speed
:",speed_time(time_1,time_2,test_input),"w/sec")
        print("error:",mistake(test1,test_input))
    elif ck =="no":
         print("thank you")
         break
    else:
         print("wrong input")
```

screenshot



File Edit Shell Debug Options Window Help

```
    Python 3.10.2 (tags/v3.10.2:a58ebcc, Jan 17 2022, 14:12:15) [MSC v.1929 64 bit (AMD64)] on win32
    Type "help", "copyright", "credits" or "license()" for more information.
>>>
    ================== RESTART: F:/Desktop/nainika cse project.py ==================
    1763994357.2562952
    are you ready to test:yes/no:yes
    *******typing speed calculator*********
    Each element is identified by a unique numerical index, which represents its position in the array, starting from 0


    enter:we are in a world of ai
    speed : 1 w/sec
    error: 115
    are you ready to test:yes/no:no
    thank you
>>> |
```

Testing Approach □

The testing approach for the Typing Speed Calculator ensures the reliability and accuracy of the core logic while validating a robust user workflow. This strategy is essential for meeting the non-functional requirements (Reliability and Error Handling Strategy) outlined in the project guidelines.

The approach focuses on Unit Testing the critical metric calculations and Validation Testing the application's overall flow.

---

1. Unit Testing (Verification of Core Logic)

Unit tests isolate the smallest testable parts of the application—the functions within the **`MetricsCalculator`** module—to ensure their output is mathematically accurate. This testing should be automated using a framework like Python's standard **`unittest`** module.

Test Cases for `mistake()` Function (Error Counting):

Test Scenario	**Test Input (Source vs. User)**	**Expected Output (Raw Errors)**	**Purpose**
T1. Zero Errors	Source: "The test"	User: "The test"	0
T2. Positive Errors	Source: "apple"	User: "opple"	1
T3. Short Input (Boundary Case)	Source: "Python" (Length 6)	User: "Py" (Length 2)	4

Test Scenario	Test Input (Source vs. User)	Expected Output (Raw Errors)	Purpose
T4. Extra Character Error	Source: "code"	User: "coder"	1 (The function should count the mismatch at 'e' vs 'r' and then the extra character based on the logic.)

challenges Faced 🚧

The development of the console-based Typing Speed Calculator presented several technical and conceptual challenges, primarily related to the constraints of the execution environment and ensuring the accuracy of the core metrics.

Conceptual Challenge: Accurate WPM Calculation

- **Problem:** The initial implementation used a simplified calculation of $\frac{\text{Length of Input}}{\text{Time Delay}}$ (characters per second), which did not adhere to the **academic standard of Words Per Minute (WPM)**.
- **Solution & Learning:** The primary challenge was converting the raw calculation into the standardized metric. This required a clear understanding of the accepted WPM formula (where 1 word = 5 characters) and proper time unit conversion (seconds to minutes). This forced a crucial **design decision** to prioritize metric accuracy and **reliability** over simple speed approximation.

Implementation Challenge: Timer Start and User Experience

- **Problem:** In a standard Python console application, the `time()` function must be called *before* the `input()` function. This means the timer starts the moment the text is displayed, including the user's **reading time** and the time it takes them to place their fingers on the keyboard. This inherently inflates the duration, leading to a potentially lower reported WPM.

- **Mitigation:** While impossible to fix entirely in a batch-input console environment, the decision was made to accept this limitation and document it clearly. The solution was to focus on **consistency**—since the error is systematic, all tests within the system are subject to the same timing method, making the results comparable internally.

Technical Challenge: Handling Error Edge Cases

- **Problem:** Ensuring the **`mistake` function** accurately counted errors when the user typed a text that was **shorter than the source text**. Without proper handling, accessing an index that doesn't exist in the shorter user input would trigger an unhandled `IndexError`, crashing the program.
- **Solution:** The **`try-except` block** was implemented within the `mistake` function. This specific solution allows the program to gracefully catch the `IndexError` and register the remaining, untyped characters as errors, fulfilling the **Error Handling Strategy** (NFR).

4. Structural Challenge: Achieving Modularity

- **Problem:** The initial code was a single, monolithic script, failing to meet the **Implementation Quality** requirement for **5-10 meaningful modules/files**.
- **Solution & Learning:** The challenge was overcome by logically refactoring the system into a **Three-Tier Architecture** (Presentation, Logic, Data). This exercise enforced the discipline of defining clear interfaces between components (`ConsoleApp.py`, `MetricsCalculator.py`, `TextManager.py`), which is a key concept in software engineering and vital for the project's overall maintainability.

Learnings & Key Takeaways ?

The development of the Typing Speed Calculator provided several valuable takeaways, demonstrating the successful application of core programming and system design concepts as required by the course syllabus.

1. Application of Subject Concepts

- **Modular Programming & Refactoring:** The project highlighted the critical difference between a working script and a well-engineered system. The necessity of refactoring the monolithic code into a **Three-Tier Architecture** (Presentation, Logic, Data) was the most significant learning outcome. This enforced the principle of **separation of concerns** and directly improved the system's maintainability.
- **Time and Event Handling:** Effective use of the `time` module and conditional flow (`while` loops) was central to the project. The implementation taught the precise techniques required for measuring duration and accurately capturing events in a sequential process.
- **Error Handling (Defensive Programming):** The successful use of the **`try-except` block** in the `mistake` function provided a practical lesson in **defensive**

programming. It ensured the application could handle boundary conditions (user input being shorter than source text) gracefully, preventing crashes and enhancing the overall **Reliability** of the software.

Design and Metric Integrity

- **Metric Accuracy over Simplicity:** The initial error of using an unconventional "characters/second" metric over the standardized **Words Per Minute (WPM)** emphasized the importance of using mathematically verified formulas. The takeaway is that **correctness of the metric is paramount** in data-driven applications, requiring thorough validation during the design phase.
- **Constraints of the Execution Environment:** Working in a console environment highlighted its limitations (e.g., inability to start the timer precisely on the *first keystroke* or provide real-time visual feedback). This informed the **Design Decisions** and taught the necessity of choosing the right tool (e.g., a GUI or web app) for specific **Usability** requirements.

Documentation and Testing

- **Importance of Unit Testing:** The process of defining **Unit Tests** for the `mistake` and `speed_time` functions reinforced the importance of TDD (Test-Driven

Development) principles. It demonstrated that code functionality is only as reliable as its accompanying test suite.
- **Architectural Documentation:** Creating the **Process Flow Diagram** and the **Component Diagram** (as part of the System Architecture) provided a deeper understanding of how abstract logic is translated into a structured, visual design, which is essential for collaborative project development.

14. Future Enhancements 🚀

To elevate the Typing Speed Calculator from a basic console application to a comprehensive, market-ready project, several enhancements are proposed. These additions increase the project's **Depth & Complexity** (15% of the evaluation rubric) and address the limitations of the current console environment.

---

1. User Interface and Experience (UI/UX) Overhaul

- **GUI Migration:** The most critical enhancement is migrating the interface from the Python console to a dedicated **Graphical User Interface (GUI)** using a framework like **Tkinter, PyQt, or a web application (Flask/Django)**.
- **Real-time Visual Feedback:** A GUI enables the system to provide **real-time highlighting** of correct (green) and incorrect (red) characters, satisfying the initial functional requirements that the console could not meet.

- **Intuitive Controls:** Replace console prompts with graphical buttons and input fields for a better user experience.

2. Data Persistence and Analytics

- **High Score System:** Implement a persistent storage layer (e.g., using a **SQLite database** or a simple **JSON/CSV file**) to save user scores (WPM, Accuracy, Time).
- **User Profiles and Leaderboards:** Introduce a feature for users to create profiles and track their progress over time. A **leaderboard** feature would allow users to compete, significantly boosting the project's **reporting and analytics** capabilities (FRs).
- **Enhanced Analytics:** Store and display historical data in a graphical format (e.g., a chart showing WPM improvement over the last 10 tests).

3. Feature Set Expansion

- **Custom Text Input:** Allow the user to paste their own text into a dedicated field for practice, providing personalized training.
- **Difficulty Levels:** Introduce options to select test text based on **length** (short sentences vs. long paragraphs) or **vocabulary complexity**, catering to different skill levels.
- **Targeted Practice:** Implement a feature to track and report frequently mistyped characters or words, allowing the user to engage in focused practice drills.

4. Technical and Architectural Upgrades

- **External Data API:** Instead of relying on hardcoded text in the `TextManager.py` module, the system could fetch test content from an external public API, allowing for a dynamically updated and massive library of typing material.
- **Timer Refinement (Threading):** For a GUI implementation, use **multi-threading** to run the timer logic separately from the UI thread. This ensures the timer starts on the *very first keystroke* (not just when the text is loaded) and prevents the UI from freezing during heavy calculations.

References 📚

This section provides the necessary citations for the development of the Typing Speed Calculator project, including the primary source documentation, external resources used for standard definitions, and the project's own internal documentation.

A. Primary Project Documentation

- **Project Guidelines:**
    - **"Build Your Own Project (1).pdf"**: The official course documentation outlining the submission requirements, report structure, functional, and non-functional requirements.
        - *(Source: File uploaded by user)*

- **Source Code:**
    - **`codes.py`**: The initial Python script serving as the foundational implementation for the Typing Speed Calculator logic (timing, error detection, and workflow).
        - *(Source: File uploaded by user)*

B. Standardized Definitions and Algorithms

- **Words Per Minute (WPM) Formula:**
    - Standard definition used for the corrected metric calculation, where WPM is derived from the net characters typed (divided by five) over the time elapsed in minutes.
        - *(Source: Academic standards for typing proficiency measurement)*
- **Python Standard Library:**
    - **`time` module:** Used for capturing precise timestamps (`time.time()`) to measure the duration of the typing test.
        - *(Source: Python Software Foundation. Python Documentation)*
    - **`random` module:** Used for the random selection of the test paragraph from the source list.
        - *(Source: Python Software Foundation. Python Documentation)*

C. Architectural and Design Resources

- **UML and Component Design Principles:**
    - Principles guiding the refactoring into a **Three-Tier Architecture** and the creation of the **Component**

Diagram, focusing on the separation of Presentation, Business Logic, and Data layers.

- *(Source: Principles of Software Engineering and Design)*

Design Diagrams 🎨

The Design Diagrams section provides the visual blueprints of the Typing Speed Calculator system's structure and behavior, aligning with the "Design & Documentation" requirements. Based on the proposed **Three-Tier Architecture** (refactoring `codes.py` into modules), the following diagrams are essential:

Process Flow / Workflow Diagram

This diagram illustrates the logical sequence of steps the user takes from launching the application to receiving the final score. It demonstrates the **clear input/output structure** and **logical workflow** (FRs) of the console application.

1. **Initiation:** The main loop starts.
2. **Ready Prompt:** User is asked to confirm readiness.
3. **Test Start:** If 'yes,' the system selects text and starts the timer (`time_1`).
4. **Data Capture:** User enters text (`input()`).
5. **Metrics Calculation:** The system calls the `MetricsCalculator` functions to compute WPM and Errors.
6. **Results:** Final scores are displayed to the user.
7. **Termination/Loop:** User is looped back to the start or exits.

UML Component Diagram

This diagram visually represents the **modular and clean implementation** by showing how the refactored Python files interact, confirming the **System Architecture** design.

Component	**Functionality**
`ConsoleApp.py`	Acts as the Presentation Layer; manages user input and output.
`TextManager.py`	Acts as the Data Layer; provides random test passages.
`MetricsCalculator.py`	Acts as the Business Logic Layer; performs WPM and error calculations.

UML Use Case Diagram

The Use Case Diagram describes the functional requirements from the perspective of the external user (**Actor**). It defines the boundaries and primary interactions possible within the system.

- **Actor:** User
- **Use Cases:** The primary functions the user can execute.

Use Case	**Description**
Start Typing Test	Initiates the test, selects text, and starts the timer.
Input Text	The core interaction where the user enters the source text.
View Final Results	Displays calculated WPM, Accuracy, and Error Count (FR 3).
Repeat Test	Re-enters the workflow loop with a new random passage