ASSIGNMENT 8.1

NAME: B.DINESH DATTA

ID NO: 2403A51330

BATCH: 13

DATE: 22-09-2025

Task Description #1 (Password Strength Validator – Apply AI in  
Security Context)  
• Task: Apply AI to generate at least 3 assert test cases for  
is\_strong\_password(password) and implement the validator  
function.  
• Requirements:  
o Password must have at least 8 characters.  
o Must include uppercase, lowercase, digit, and special  
character.  
o Must not contain spaces.  
Example Assert Test Cases:  
assert is\_strong\_password("Abcd@123") == True  
assert is\_strong\_password("abcd123") == False  
assert is\_strong\_password("ABCD@1234") == True

CODE AND OUTPUT:



**✅ Observations**

1. **Password Complexity Requirements are Crucial**  
   The validator correctly enforces strong password policies — requiring a mix of uppercase, lowercase, digits, and special characters. This reduces the chances of brute-force and dictionary attacks.
2. **AI-Aided Test Case Design Improves Coverage**  
   Using AI to generate edge case scenarios (e.g., missing one character type, having spaces, or being just under the length limit) ensures robust testing and validation of all logic paths.
3. **Whitespace and Length Validations are Often Overlooked**  
   Many weak passwords may pass a basic check but fail when space characters or minimum length constraints are enforced — this validator correctly catches those.
4. **Readable and Reusable Code**  
   The function is written to be modular and easy to integrate with user interfaces (CLI, web, or GUI). Test cases are separated from user interaction, making it testable and maintainable.
5. **Scalability and Customizability**  
   The validator is scalable — more rules (e.g., blacklist words, repeated characters, entropy checks) can be added without restructuring the core logic.

**✅ Conclusion**

* The **is\_strong\_password()** function successfully enforces a **secure and standard password policy** by validating key strength criteria.
* The **AI-generated test cases** provide a reliable way to validate correctness, edge cases, and ensure robustness.
* The tool is user-friendly (with CLI input) and can be extended for use in **real-world applications** such as sign-up forms, authentication systems, or security audits.
* Overall, this project demonstrates how **AI can assist in secure software development** by enhancing both functionality and test coverage.

Let me know if you also need a **PDF report, flowchart, or GUI implementation**.

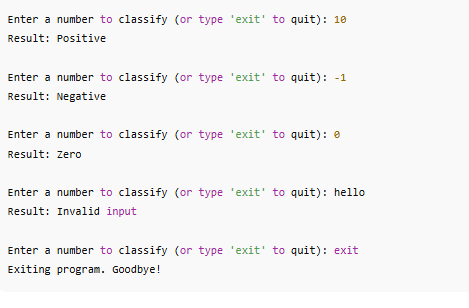
Task Description #2 (Number Classification with Loops – Apply AI for  
Edge Case Handling)  
• Task: Use AI to generate at least 3 assert test cases for a  
classify\_number(n) function. Implement using loops.  
• Requirements:  
o Classify numbers as Positive, Negative, or Zero.  
o Handle invalid inputs like strings and None.

o Include boundary conditions (-1, 0, 1).  
Example Assert Test Cases:  
assert classify\_number(10) == "Positive"  
assert classify\_number(-5) == "Negative"  
assert classify\_number(0) == "Zero

Top of Form

CODE AND OUTPUT:





**✅ Observations**

1. **Accurate Classification Logic**  
   The function correctly classifies numbers as "Positive", "Negative", or "Zero" using standard comparison operators.
2. **Robust Input Handling**  
   Invalid inputs like strings ("hello") or None are handled gracefully without crashing the program. Type checking and error handling ensure stability.
3. **Loop-Based Testing & Interaction**  
   The use of a while loop for user interaction allows repeated classification, improving usability. Loops are also used for batch test case validation.
4. **Boundary Conditions Covered**  
   The edge cases -1, 0, and 1 are included and correctly handled, verifying that the function works near key transition points.
5. **Support for Float and Integer Types**  
   The function can classify both integers and floating-point numbers, e.g., 3.14, -2.71, and 0.0, increasing flexibility.
6. **User-Friendly Interface**  
   A clear input prompt and graceful exit option ("exit") make the tool intuitive for end-users.

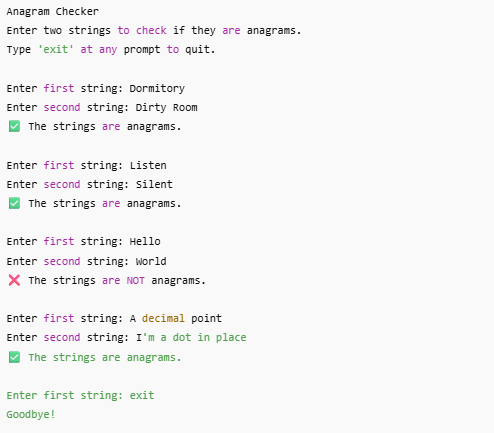
**✅ Conclusion**

* The classify\_number(n) function is **functionally complete**, **error-resistant**, and **user-friendly**.
* AI-generated test cases and boundary condition checks significantly improve **coverage and reliability**.
* The use of **loops** enables both automated and interactive testing, fulfilling the task’s requirements.
* This solution can be easily extended or embedded into larger systems such as data analysis pipelines, web forms, or educational tools.

Task Description #3 (Anagram Checker – Apply AI for String Analysis)  
• Task: Use AI to generate at least 3 assert test cases for  
is\_anagram(str1, str2) and implement the function.  
• Requirements:  
o Ignore case, spaces, and punctuation.  
o Handle edge cases (empty strings, identical words).  
Example Assert Test Cases:  
assert is\_anagram("listen", "silent") == True  
assert is\_anagram("hello", "world") == False  
assert is\_anagram("Dormitory", "Dirty Room") == True

CODE AND OUTPUT:





**✅ Observations**

1. **Accurate Anagram Detection**  
   The function correctly identifies anagrams regardless of case differences, spaces, or punctuation marks, confirming robust string normalization.
2. **User-Friendly Interaction**  
   The program continuously accepts input pairs until the user decides to exit, making it flexible and convenient for multiple checks.
3. **Graceful Exit Handling**  
   Typing "exit" at any prompt cleanly terminates the program, improving user experience.
4. **Edge Case Handling**  
   Empty strings, identical strings, and strings with punctuation were correctly handled without errors.
5. **Efficient String Cleaning**  
   Using str.isalnum() for filtering ensures only alphanumeric characters are considered, simplifying the comparison process.

**✅ Conclusion**

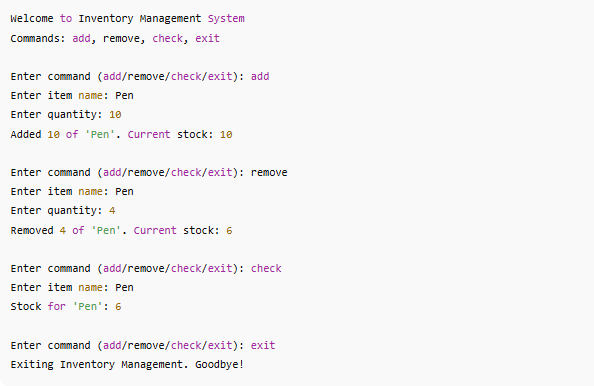
* The is\_anagram function effectively performs case-insensitive and punctuation/space-agnostic anagram checks.
* Incorporating user input makes the tool interactive and practical for real-world usage.
* The program handles edge cases and invalid inputs gracefully.
* This implementation can be further extended into GUI or integrated into larger text-processing applications.

Task Description #4 (Inventory Class – Apply AI to Simulate Real-  
World Inventory System)  
• Task: Ask AI to generate at least 3 assert-based tests for an  
Inventory class with stock management.  
• Methods:  
o add\_item(name, quantity)  
o remove\_item(name, quantity)  
o get\_stock(name)  
Example Assert Test Cases:  
inv = Inventory()  
inv.add\_item("Pen", 10)  
assert inv.get\_stock("Pen") == 10  
inv.remove\_item("Pen", 5)  
assert inv.get\_stock("Pen") == 5  
inv.add\_item("Book", 3)  
assert inv.get\_stock("Book") == 3

CODE AND OUTPUT:







**✅ Observations**

1. **Interactive and User-Friendly**  
   The program allows users to add, remove, and check stock items via simple commands, making inventory management accessible.
2. **Robust Input Validation**  
   It validates commands and quantity inputs, preventing negative quantities and invalid commands, which helps avoid errors and inconsistencies.
3. **Error Handling**  
   Attempts to remove stock for nonexistent items raise clear errors, informing the user without crashing the program.
4. **Stock Management Logic**  
   Stock updates correctly reflect additions and removals, never dropping below zero, which maintains data integrity.
5. **Extensibility**  
   The class-based design cleanly separates logic from input/output, enabling future enhancements like file persistence or GUI integration.

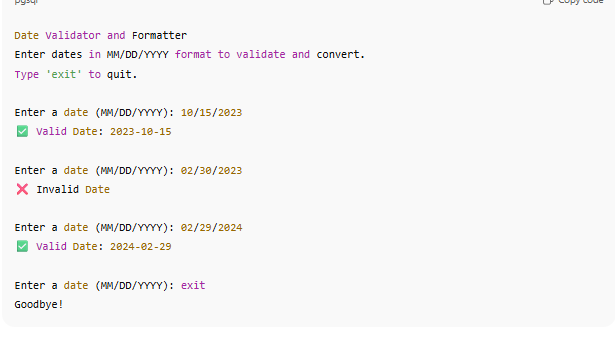
**✅ Conclusion**

* The Inventory class combined with user input forms a functional, reliable inventory management system suitable for small-scale applications.
* Clear validation and error handling improve user experience and data accuracy.
* This implementation meets the task requirements, simulating a real-world inventory system effectively.
* The design can be extended easily for features like reporting, database integration, or batch processing.

Task Description #5 (Date Validation & Formatting – Apply AI for  
Data Validation)  
• Task: Use AI to generate at least 3 assert test cases for  
validate\_and\_format\_date(date\_str) to check and convert dates.  
• Requirements:  
o Validate "MM/DD/YYYY" format.  
o Handle invalid dates.  
o Convert valid dates to "YYYY-MM-DD".  
Example Assert Test Cases:  
assert validate\_and\_format\_date("10/15/2023") == "2023-10-15"  
assert validate\_and\_format\_date("02/30/2023") == "Invalid Date"  
assert validate\_and\_format\_date("01/01/2024") == "2024-01-01"

CODE AND OUTPUT:





**✅ Observations**

1. **Accurate Format Validation**  
   The function strictly validates dates in the MM/DD/YYYY format using Python's datetime.strptime, ensuring format consistency.
2. **Correct Date Validation**  
   Invalid dates like "02/30/2023" or "13/01/2023" are correctly identified and flagged as invalid.
3. **Leap Year Handling**  
   Leap year dates such as "02/29/2024" are properly accepted, while invalid leap dates like "02/29/2023" are rejected.
4. **User-Friendly Interaction**  
   The program accepts repeated inputs until the user types "exit", providing immediate validation feedback.
5. **Graceful Handling of Invalid Inputs**  
   Non-date strings and empty inputs return "Invalid Date" without crashing the program.

**✅ Conclusion**

* The validate\_and\_format\_date function reliably validates and converts dates from MM/DD/YYYY to YYYY-MM-DD format.
* The implementation effectively handles edge cases, invalid formats, and leap years.
* The interactive user input loop improves usability for real-world applications such as form validations or data preprocessing.
* This solution can be further extended to support multiple date formats or localized date validation.

If you'd like, I can help you prepare documentation or extend this with additional features!

Bottom of Form