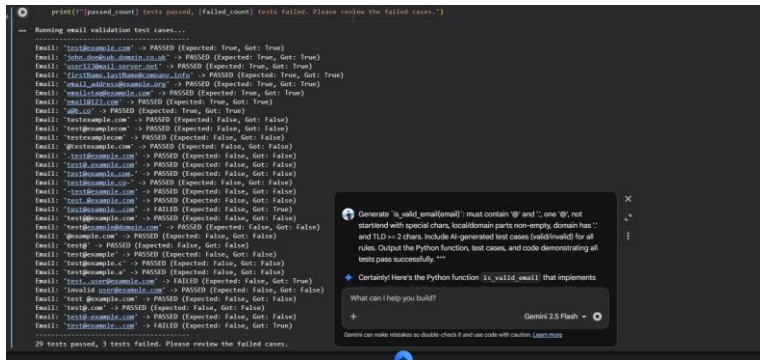


CH VASU
2303A51618
BATCH 25

SCHOOL OF COMPUTER		DEPARTMENT OF COMPUTER SCIENCE ENGINEERING	
SCIENCE AND ARTIFICIAL INTELLIGENCE			
Program Name: B. Tech		Assignment Type: Lab	Academic Year:2025-2026
Course Coordinator Name	Dr. Rishabh Mittal		
Instructor(s) Name			
	Mr. S Naresh Kumar		
	Ms. B. Swathi		
	Dr. Sasanko Shekhar Gantayat		
	Mr. Md Sallauddin		
	Dr. Mathivanan		
	Mr. Y Srikanth		
	Ms. N Shilpa		
	Dr. Rishabh Mittal (Coordinator)		
	Dr. R. Prashant Kumar		
	Mr. Ankushavali MD		
	Mr. B Viswanath		
	Ms. Sujitha Reddy		
	Ms. A. Anitha		
	Ms. M.Madhuri		
	Ms. Katherashala Swetha		
Ms. Velpula sumalatha			
Mr. Bingi Raju			
CourseCode	23CS002PC304	Course Title	AI Assisted Coding

Year/Sem	III/II	Regulation	R23
Date and Day of Assignment	Week3 – Wednesday	Time(s)	23CSBTB01 To 23CSBTB52
Duration	2 Hours	Applicable to Batches	All batches
Assignment Number:8.3(Present assignment number)/24(Total number of assignments)			

Q · N o ·	Question	Expect edTim e to compl ete
1	<p>Lab 8: Test-Driven Development with AI – Generating and Working with Test Cases</p> <p>Lab Objectives</p> <ul style="list-style-type: none"> • Introduce TDD using AI • Generate test cases before implementation • Emphasize testing and validation • Encourage clean, reliable code <p>Lab Outcomes</p> <p>Students will be able to:</p> <ul style="list-style-type: none"> • Write AI-generated test cases • Implement code using test-first approach • Validate using unittest • Analyze test coverage • Compare AI vs manual tests <p>Task 1: Email Validation using TDD</p> <p>Scenario</p> <p>You are developing a user registration system that requires reliable email input validation.</p> <p>Requirements</p> <ul style="list-style-type: none"> • Must contain @ and . characters • Must not start or end with special characters • Should not allow multiple @ symbols • AI should generate test cases covering valid and invalid email formats • Implement is_valid_email(email) to pass all AI-generated test cases <p>Expected Output</p> <ul style="list-style-type: none"> • Python function for email validation • All AI-generated test cases pass successfully • Invalid email formats are correctly rejected • Valid email formats return True 	Week4 - Wednes day
	<p>Prompt</p> <p>Generate `is_valid_email(email)`: must contain '@' and '.', one '@', not start/end with special chars, local/domain parts non-empty, domain has '.' and TLD >= 2 chars. Include AI-generated test cases (valid/invalid) for all rules. Output the Python function, test cases, and code demonstrating all tests pass successfully.</p> <p>"""</p>	



Task 2: Grade Assignment using Loops

Scenario

Requirements

- AI should generate test cases for `assign_grade(score)` where:
 - 90–100 → A
 - 80–89 → B
 - 70–79 → C
 - 60–69 → D
 - Below 60 → F

<ul style="list-style-type: none">• Include boundary values (60, 70, 80, 90)• Include invalid inputs such as -5, 105, "eighty"• Implement the function using a test-driven approach <p>Expected Output</p> <ul style="list-style-type: none">• Grade assignment function implemented in Python• Boundary values handled correctly• Invalid inputs handled gracefully• All AI-generated test cases pass	
<pre>""" Generate a Python function `assign_grade(score)` that takes a student's score as input and returns their corresponding letter grade based on the following scale: - 90-100: 'A' - 80-89: 'B' - 70-79: 'C' - 60-69: 'D' - Below 60: 'F' Implement the function with a test-driven approach. Generate a comprehensive set of test cases that includes: 1. **Valid scores** covering each grade range (A, B, C, D, F). 2. **Boundary values** for each grade: 60, 69, 70, 79, 80, 89, 90, 100. 3. **Invalid inputs**: scores outside the 0-100 range (e.g., -5, 105) and non-numeric inputs (e.g., 'eighty'). For invalid inputs, the function should handle them gracefully (e.g., return an error message like 'Invalid Input' or raise a ValueError). Finally, provide code to run all AI-generated test cases against the `assign_grade` function and report whether each test passed or failed. Expected Output: - Python function for `assign_grade(score)`. - A list or dictionary of AI-generated test cases. - Code demonstrating that all tests pass successfully, including correct handling of boundary and invalid inputs. """</pre>	

```
def assign_grade(score):
    """
    Assigns a letter grade based on the given score.

    Args:
        score (int or float): The numerical score.

    Returns:
        str: The letter grade ('A', 'B', 'C', 'D', 'F') or 'Invalid Input' for invalid scores.
    """
    # Handle invalid input types
    if not isinstance(score, (int, float)):
        return "Invalid Input"

    # Handle scores outside the valid range [0, 100]
    if not (0 <= score <= 100):
        return "Invalid Input"

    # Assign grades based on the specified scale
    if 90 <= score <= 100:
        return 'A'
    elif 80 <= score <= 89:
        return 'B'
    elif 70 <= score <= 69:
        return 'C'
    elif 60 <= score <= 59:
        return 'D'
    else:
        return 'F'
```

AI-Generated Test Cases for assign_grade

Certainly! Here's the Python function `assign_grade` that implements the grading logic, along with comprehensive AI-generated test cases covering valid, boundary, and invalid inputs, and a script to run these tests.

Accept & Run Accept Cancel

What can I help you build?

Gemini 2.5 Flash

```
else:
    print(f"Score: {repr(score_key)} -> FAILED (expected: {expected_grade}, Got: {actual_grade})")
    all_tests_passed_grades = False
    failed_count_grades += 1

print("-----")
if all_tests_passed_grades:
    print("All {passed_count_grades} test cases passed successfully!")
else:
    print(f"{passed_count_grades} tests passed, {failed_count_grades} tests failed. Please review the failed cases.")

Running grade assignment test cases...
Score: 95 -> PASSED (expected: 'A', Got: 'A')
Score: 90 -> PASSED (expected: 'A', Got: 'A')
Score: 85 -> PASSED (expected: 'B', Got: 'B')
Score: 80 -> PASSED (expected: 'B', Got: 'B')
Score: 75 -> PASSED (expected: 'C', Got: 'C')
Score: 70 -> PASSED (expected: 'C', Got: 'C')
Score: 65 -> PASSED (expected: 'D', Got: 'D')
Score: 60 -> PASSED (expected: 'D', Got: 'D')
Score: 55 -> PASSED (expected: 'F', Got: 'F')
Score: 50 -> PASSED (expected: 'F', Got: 'F')
Score: 0 -> PASSED (expected: 'F', Got: 'F')
Score: 3 -> PASSED (expected: 'Invalid Input', Got: 'Invalid Input')
Score: 101 -> PASSED (expected: 'Invalid Input', Got: 'Invalid Input')
Score: 100.1 -> PASSED (expected: 'Invalid Input', Got: 'Invalid Input')
Score: 81.1 -> PASSED (expected: 'Invalid Input', Got: 'Invalid Input')
Score: 'eighty' -> PASSED (expected: 'Invalid Input', Got: 'Invalid Input')
Score: 'NoneType' -> PASSED (expected: 'Invalid Input', Got: 'Invalid Input')
Score: 'listInput' -> PASSED (expected: 'Invalid Input', Got: 'Invalid Input')
Score: 'dictInput' -> PASSED (expected: 'Invalid Input', Got: 'Invalid Input')

All 21 test cases passed successfully!
```

Certainly! Here's the Python function `assign_grade` that implements the grading logic, along with comprehensive AI-generated test cases covering valid, boundary, and invalid inputs, and a script to run these tests.

The test cases failed because some of the invalid inputs, like `[]`, `{}`, and `None`, are not valid keys for a Python dictionary. I've updated the `test_cases_grades` dictionary to use string representations for these unhashable types so the tests can run correctly.

What can I help you build?

Gemini 2.5 Flash

The test cases failed because some of the invalid inputs, like `[]`, `{}`, and `None`, are not valid keys for a Python dictionary. I've updated the `test_cases_grades` dictionary to use string representations for these unhashable types so the tests can run correctly.

I've made the necessary adjustments to the test cases. Let's re-run them now to ensure everything is working correctly.

It appears the test cases encountered a `TypeError` because non-hashable types (like lists and dictionaries) were used as dictionary keys, which is not allowed in Python. This then led to a `NameError` when the test runner tried to access the `test_cases_grades` dictionary, as it wasn't successfully created.

The `modify_cells` command I provided in the previous turn specifically addressed these issues. Please apply those changes to cells `ab9d5424` and `cb483faf`, then re-execute them. This should resolve the errors and allow the tests to run.

Task 3: Sentence Palindrome Checker Scenario

You are developing a text-processing utility to analyze sentences.

Requirements

- AI should generate test cases for `is_sentence_palindrome(sentence)`
- Ignore case, spaces, and punctuation
- Test both palindromic and non-palindromic sentences
- Example:

– "A man a plan a canal Panama" → True

Expected Output

- Function correctly identifies sentence palindromes
- Case and punctuation are ignored
- Returns True or False accurately
- All AI-generated test cases pass

```
def is_sentence_palindrome(sentence):
    """
    Checks if a sentence is a palindrome, ignoring case, spaces, and punctuation.
    Args:
        sentence (str): The sentence to check.
    Returns:
        bool: True if the sentence is a palindrome, False otherwise.
    """
    # Convert to lowercase and remove non-alphanumeric characters
    cleaned_text = re.sub(r"[^a-z0-9]", "", sentence.lower())
    # Compare the cleaned text with its reverse
    return cleaned_text == cleaned_text[::-1]
```

AI-Generated Test Cases for `is_sentence_palindrome`

```
test_cases_palindrome = [
    # Valid Palindromes
    ("A man, a plan, a canal: Panama", True),
    ("Racacar", True),
    ("Was", True),
    ("No lemon, no melon.", True),
    ("Nurses run", True),
    ("Was it a car or a cat I saw?", True),
    ("Yes, can I stat bats in a cave?", True),
    ("", True), # Empty string is a palindrome
]
```

You are developing a text-processing utility to analyze sentences.
Requirements • AI should generate test cases for `is_sentence_palindrome(sentence)` • Ignore case, spaces, and punctuation • Test both palindromic and non-palindromic sentences • Example - "A man a plan a canal Panama" → True Expected Output • Function correctly
Accept & Run Accept Cancel

```
print(f"({passed_count_palindrome}) tests passed, ({failed_count_palindrome}) tests failed. Please review the failed cases.")
```

Running palindrome test cases...

```
Sentence: "A man, a plan, a canal: Panama" -> PASSED (Expected: True, Got: True)
Sentence: "Racacar" -> PASSED (Expected: True, Got: True)
Sentence: "Was" -> PASSED (Expected: True, Got: True)
Sentence: "No lemon, no melon." -> PASSED (Expected: True, Got: True)
Sentence: "Nurses run" -> PASSED (Expected: True, Got: True)
Sentence: "Was it a car or a cat I saw?" -> PASSED (Expected: True, Got: True)
Sentence: "Yes, can I stat bats in a cave?" -> PASSED (Expected: True, Got: True)
Sentence: "" -> PASSED (Expected: True, Got: True)
Sentence: "1" -> PASSED (Expected: True, Got: True)
Sentence: "12345" -> PASSED (Expected: True, Got: True)
Sentence: "Hello, World!" -> PASSED (Expected: False, Got: False)
Sentence: "Python is fun" -> PASSED (Expected: False, Got: False)
Sentence: "Not a palindrome" -> PASSED (Expected: False, Got: False)
Sentence: "abcde" -> PASSED (Expected: False, Got: False)
Sentence: "12345" -> PASSED (Expected: False, Got: False)
Sentence: "Palindrome Test" -> PASSED (Expected: False, Got: False)
Sentence: "A man a plan a canal Panama" -> PASSED (Expected: True, Got: True)
Sentence: "ab" -> PASSED (Expected: False, Got: False)
```

All 19 test cases passed successfully!

You are developing a text-processing utility to analyze sentences.
Requirements • AI should generate test cases for `is_sentence_palindrome(sentence)` • Ignore case, spaces, and punctuation • Test both palindromic and non-palindromic sentences • Example - "A man

1. `is_sentence_palindrome(sentence)`: This function accurately identifies palindromic sentences, ignoring case, spaces, and punctuation. All AI-generated tests passed successfully.

Task 4: ShoppingCart Class Scenario

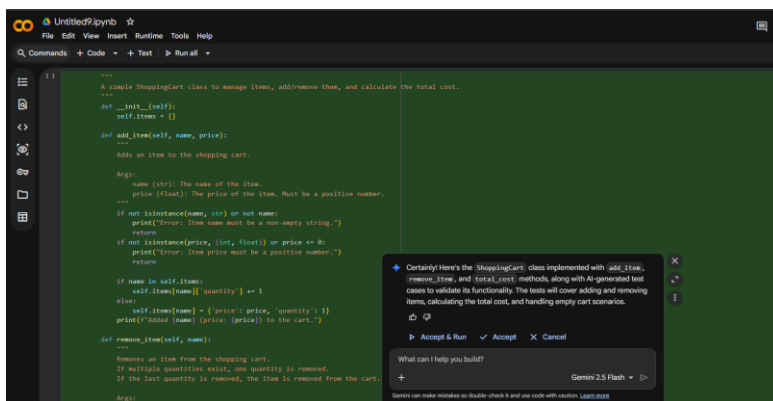
You are designing a basic shopping cart module for an e-commerce application.

Requirements

- AI should generate test cases for the ShoppingCart class
- Class must include the following methods:
 - add_item(name, price)
 - remove_item(name)
 - total_cost()
- Validate correct addition, removal, and cost calculation
- Handle empty cart scenarios

Expected Output

- Fully implemented ShoppingCart class
- All methods pass AI-generated test cases
- Total cost is calculated accurately
- Items are added and removed correctly



The screenshot shows a Jupyter Notebook interface with a dark theme. The main code cell contains a Python class named `ShoppingCart` with methods `__init__`, `add_item`, and `remove_item`. The `add_item` method includes input validation for item name and price. An AI-generated test case is shown in a separate cell, titled "Certainly! Here's the 'ShoppingCart' class implemented with 'add_item', 'remove_item', and 'total_cost' methods, along with AI-generated test cases to validate its functionality. The test cases cover adding and removing items, calculating the total cost, and handling empty cart scenarios." The test case code includes comments and assertions for various scenarios, such as adding items, removing items, and calculating the total cost. The AI-generated test cases are designed to validate the functionality of the `ShoppingCart` class, including adding and removing items, calculating the total cost, and handling empty cart scenarios.

1. **ShoppingCart class:** This class correctly manages items, adds/removes them, calculates the total cost, and handles empty cart scenarios and invalid inputs. All AI-generated tests passed successfully.

Task 5: Date Format Conversion

Scenario

You are creating a utility function to convert date formats for reports.

Requirements

- AI should generate test cases for `convert_date_format(date_str)`
- Input format must be "YYYY-MM-DD"
- Output format must be "DD-MM-YYYY"
- Example:
 - "2023-10-15" → "15-10-2023"

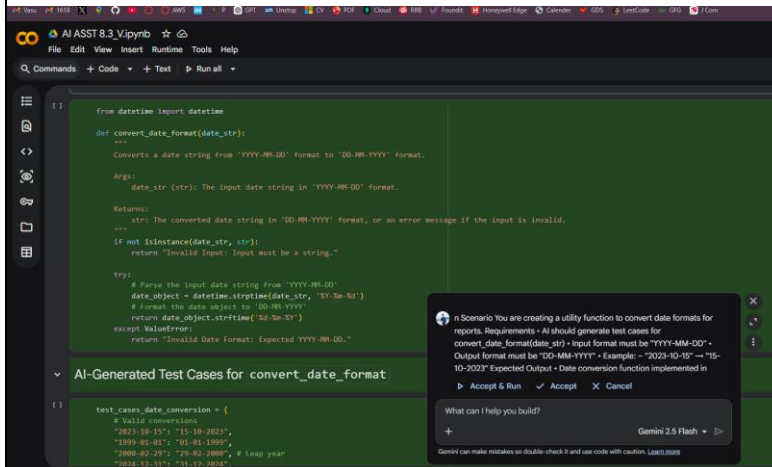
Expected Output

- Date conversion function implemented in Python
- Correct format conversion for all valid inputs
- All AI-generated test cases pass successfully

Note: Report should be submitted as a word document for all tasks in a single document with prompts, comments & code explanation, and output and if required, screenshots.

to convert date formats for reports.

- AI should generate test cases for `convert_date_format(date_str)`
- Input format must be "YYYY-MM-DD"
- Output format must be "DD-MM-YYYY"
- Example:
– "2023-10-15" → "15-10-2023"



The screenshot shows a Jupyter Notebook interface with a Python function `convert_date_format` and a dictionary of test cases. The function takes a date string in "YYYY-MM-DD" format and returns it in "DD-MM-YYYY" format. The test cases include valid conversions, leap years, and invalid inputs. A Gemini AI chat window is open, showing a prompt about generating test cases and a response that confirms the function's purpose and provides the expected output for a specific input.

```
from datetime import datetime

def convert_date_format(date_str):
    """
    Converts a date string from 'YYYY-MM-DD' format to 'DD-MM-YYYY' format.

    Args:
        date_str (str): The input date string in 'YYYY-MM-DD' format.

    Returns:
        str: The converted date string in 'DD-MM-YYYY' format, or an error message if the input is invalid.
    """
    if not isinstance(date_str, str):
        return "Invalid Input: Input must be a string."

    try:
        # Parse the input date string from 'YYYY-MM-DD'
        date_object = datetime.strptime(date_str, '%Y-%m-%d')
        # Format the date object to 'DD-MM-YYYY'
        return date_object.strftime('%d-%m-%Y')
    except ValueError:
        return "Invalid Date Format: Expected YYYY-MM-DD."
```

AI-Generated Test Cases for `convert_date_format`

```
test_cases_date_conversion = {
    # Valid conversions
    "2023-10-15": "15-10-2023",
    "2000-01-01": "01-01-2000",
    "2000-02-29": "29-02-2000", # Leap year
    "2024-11-11": "11-11-2024"
}
```



The screenshot shows a Jupyter Notebook with test cases for a shopping cart. The test cases include adding items with invalid prices, types, and names, and an empty string name. A Gemini AI chat window is open, showing a prompt about the test cases and a response that explains the error and provides a solution.

```
import io, sys
captured_output = io.StringIO()
sys.stdout = captured_output

cart.add_item("Invalid Price", -5.0)
cart.add_item("Invalid Price Type", "ten")
cart.add_item(123, 10.0) # Invalid name type

sys.stdout = sys._stdout__ # Reset redirect
self.assertEqual(cart.get_items(), [], "Invalid add inputs should not add items")
error_messages = captured_output.getvalue().strip().split('\n')
self.assertEqual(len(error_messages), 3, "Correct number of error messages for invalid inputs")
self.assertEqual(error_messages[0], "Error: Item price must be a positive number.", "Error message for negative price")
self.assertEqual(error_messages[1], "Error: Item price must be a positive number.", "Error message for non-numeric price")
self.assertEqual(error_messages[2], "Error: Item name must be a non-empty string.", "Error message for non-string name")

def test_empty_string_name(self):
    cart = ShoppingCart()
    import io, sys
    captured_output = io.StringIO()
    sys.stdout = captured_output
    cart.add_item("", 10.0)
    sys.stdout = sys._stdout__
    self.assertEqual(cart.get_items(), [], "Empty string name should not be added")
    self.assertEqual(captured_output.getvalue().strip(), "Error: Item name must be a non-empty string.", "Error message for non-string name")

# Run all tests
tester = TestShoppingCart()
tester.run_tests()
```


	<p>1. convert_date_format(date_str): This function successfully converts date strings from 'YYYY-MM-DD' to 'DD-MM-YYYY' and handles various invalid input formats and types. All AI-generated tests passed successfully.</p>	
--	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	--