

SR-UNIVERSITY

ASSIGNMENT:8.3

HTNO:2303A51477

BTNO:10

Task 1: Email Validation using TDD

Scenario

You are developing a user registration system that requires reliable email input validation.

Requirements

- 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

Expected Output

- Python function for email validation
- All AI-generated test cases pass successfully
- Invalid email formats are correctly rejected
- Valid email formats return True

prompt :

Generate Python unittest test cases for a function

`is_valid_email(email)`.

The email must contain exactly one @ and at least one ..

It should not start or end with special characters (@, ., _, -).

It should reject multiple @, empty strings, and invalid inputs.

Include both valid and invalid email test cases.

Code:

```
import unittest

class TestEmailValidation(unittest.TestCase):
```

```

# 86 Valid Emails
def test_valid_emails(self):
    self.assertTrue(is_valid_email("user@example.com"))
    self.assertTrue(is_valid_email("john.doe@gmail.com"))
    self.assertTrue(is_valid_email("abc123@yahoo.co.in"))

# Missing '@'
def test_missing_at(self):
    self.assertFalse(is_valid_email("userexample.com"))

# Multiple '@'
def test_multiple_at(self):
    self.assertFalse(is_valid_email("user@@example.com"))

# Missing '.'
def test_missing_dot(self):
    self.assertFalse(is_valid_email("user@examplecom"))

# Starts with special character
def test_starts_with_special(self):
    self.assertFalse(is_valid_email("@example.com"))
    self.assertFalse(is_valid_email(".user@gmail.com"))

# Ends with special character
def test_ends_with_special(self):
    self.assertFalse(is_valid_email("user@gmail.com."))
    self.assertFalse(is_valid_email("user@gmail.com@"))

# Empty string
def test_empty(self):
    self.assertFalse(is_valid_email("")))

# Non-string input
def test_non_string(self):
    self.assertFalse(is_valid_email(123))

def is_valid_email(email):

    # Must be string and not empty
    if not isinstance(email, str) or email == "":
        return False

    # Must contain exactly one '@'

```

```

if email.count "@" != 1:
    return False

# Must contain at least one '.'
if "." not in email:
    return False

# Must not start or end with special characters
special_chars = "@._-"
if email[0] in special_chars or email[-1] in special_chars:
    return False

return True

if __name__ == "__main__":
    unittest.main()

```

OUTPUT:

The screenshot shows a code editor interface with a dark theme. On the left, there's a sidebar labeled 'PROJECTS' with a single item 'Assignment 8'. The main area displays a Python file named 'test1.py'. The code in the file is as follows:

```

ASSIGNMENT 8
test1.py

1 import re
2
3 class TestEmailValidation(unittest.TestCase):
4     def test_email(self):
5         self.assertTrue(is_valid_email("abc123@yahoo.co.in"))
6
7     # Missing '@'
8     def test_missing_at(self):
9         self.assertFalse(is_valid_email("userexample.com"))
10
11     # Multiple '@'
12     def test_multiple_at(self):
13         self.assertFalse(is_valid_email("user@example.com"))
14
15     # Missing '.'
16     def test_missing_dot(self):
17         self.assertFalse(is_valid_email("user@examplecom"))
18
19     # Starts with special character
20     def test_starts_with_special(self):
21         self.assertFalse(is_valid_email("@example.com"))
22         self.assertFalse(is_valid_email(".user@gmail.com"))
23
24     # Ends with special character
25     def test_ends_with_special(self):
26         self.assertFalse(is_valid_email("user@gmail.com."))
27         self.assertFalse(is_valid_email("user@gmail.com@"))
28
29     # Empty string
30
31
32
33

```

Below the code editor is a terminal window showing the execution of the test script:

```

32-x64\bundled\libs\debugpy\launcher` 62928` -- `C:\Users\ANDU RAHUL\OneDrive\Desktop\Documents\Desktop\ai lab\assignment 8\test1.py`
.....
Ran 8 tests in 0.002s
OK
PS C:\Users\ANDU RAHUL\OneDrive\Desktop\Documents\Desktop\ai lab\assignment 8> []

```

Analysis :

This program checks whether an email is valid or not.

It uses a function called `is_valid_email(email)` to verify the format.

The function checks if the email contains exactly one @ and at least one ..

It ensures the email does not start or end with special characters (@ . _ -).

It rejects empty and non-string inputs.

`unittest` is used to test valid and invalid email cases.

If all tests pass, the program confirms correct validation.

This follows the Test-Driven Development (TDD) approach

Task 2: Grade Assignment using Loops

Scenario

You are building an automated grading system for an online examination platform.

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
- Include boundary values (60, 70, 80, 90)
- Include invalid inputs such as -5, 105, "eighty"
- Implement the function using a test-driven approach

Expected Output

- Grade assignment function implemented in Python
- Boundary values handled correctly
- Invalid inputs handled gracefully
- All AI-generated test cases pass

Prompt:

Write a Python function `assign_grade(score)`.

Use Test-Driven Development (write tests first).

If score is 90–100 → return A.

If score is 80–89 → return B.

If score is 70–79 → return C.

If score is 60–69 → return D.

If score is below 60 → return F.

Test boundary values: 60, 70, 80, 90.

Test invalid inputs: -5, 105, "eighty".

Make sure all tests pass.

Code:

```
import unittest  
class TestGradeAssignment(unittest.TestCase):  
  
    # Valid Grade Ranges
```

```

def test_grade_A(self):
    self.assertEqual(assign_grade(95), "A")
    self.assertEqual(assign_grade(90), "A")      # Boundary

def test_grade_B(self):
    self.assertEqual(assign_grade(85), "B")
    self.assertEqual(assign_grade(80), "B")      # Boundary

def test_grade_C(self):
    self.assertEqual(assign_grade(75), "C")
    self.assertEqual(assign_grade(70), "C")      # Boundary

def test_grade_D(self):
    self.assertEqual(assign_grade(65), "D")
    self.assertEqual(assign_grade(60), "D")      # Boundary

def test_grade_F(self):
    self.assertEqual(assign_grade(50), "F")
    self.assertEqual(assign_grade(0), "F")

# Invalid Inputs
def test_negative_score(self):
    self.assertEqual(assign_grade(-5), "Invalid")

def test_score_above_100(self):
    self.assertEqual(assign_grade(105), "Invalid")

def test_non_numeric_input(self):
    self.assertEqual(assign_grade("eighty"), "Invalid")

def assign_grade(score):

    # Check if input is a number
    if not isinstance(score, (int, float)):
        return "Invalid"

    # Check valid score range
    if score < 0 or score > 100:
        return "Invalid"

    # Grade Assignment

```

```

if 90 <= score <= 100:
    return "A"
elif 80 <= score <= 89:
    return "B"
elif 70 <= score <= 79:
    return "C"
elif 60 <= score <= 59:
    return "D"
else:
    return "F"

if __name__ == "__main__":
    unittest.main()

```

Output:

The screenshot shows the Visual Studio Code interface. The code editor displays a Python script named `task2.py` which contains a class `TestGradeAssignment` with six test methods: `test_grade_A`, `test_grade_B`, `test_grade_C`, `test_grade_D`, and `test_grade_F`. The test methods use `unittest.TestCase` and `assertEqual` to check grade assignments against expected values. The terminal at the bottom shows the command `python task2.py` being run, resulting in 8 tests running and an OK message.

```

1 import unittest
2
3
4 class TestGradeAssignment(unittest.TestCase):
5
6     # Valid Grade Ranges
7     def test_grade_A(self):
8         self.assertEqual(assign_grade(95), "A")
9         self.assertEqual(assign_grade(90), "A") # Boundary
10
11    def test_grade_B(self):
12        self.assertEqual(assign_grade(85), "B")
13        self.assertEqual(assign_grade(80), "B") # Boundary
14
15    def test_grade_C(self):
16        self.assertEqual(assign_grade(75), "C")
17        self.assertEqual(assign_grade(70), "C") # Boundary
18
19    def test_grade_D(self):
20        self.assertEqual(assign_grade(65), "D")
21        self.assertEqual(assign_grade(60), "D") # Boundary
22
23    def test_grade_F(self):
24        self.assertEqual(assign_grade(50), "F")
25        self.assertEqual(assign_grade(0), "F")
26

```

```

soft'WindowsApps\python3.11.exe' 'c:\Users\ANDU RAHUL\.vscode\extensions\ms-python.debugpy-2025.18.0-win
32-x64\bundled\libs\debugpy\launcher' '53626' '--' 'C:\Users\ANDU RAHUL\OneDrive\Desktop\Documents\Desktop\ai lab\assignment 8>task2.py'
.....
Ran 8 tests in 0.002s
OK
PS C:\Users\ANDU RAHUL\OneDrive\Desktop\Documents\Desktop\ai lab\assignment 8>

```

Analysis :

The program gives grades based on marks.

It uses a function called `assign_grade(score)`.

It first checks if the input is valid.

If the score is less than 0 or more than 100, it returns Invalid.

90–100 → Grade A.

80–89 → Grade B.

70–79 → Grade C.

60–69 → Grade D.

Below 60 → Grade F.

Test cases check normal, boundary, and invalid values.

All tests must pass to confirm the program works correctly.

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

Prompt:

Write a Python function `is_sentence_palindrome(sentence)`.

Write test cases first (TDD method).

Ignore capital and small letters.

Ignore spaces.

Ignore punctuation marks.

Return True if the sentence is a palindrome.

Return False if it is not.

Test both correct and incorrect sentences.

Example: "A man a plan a canal Panama" → True.

All tests should pass.

Code:

```
import unittest
```

```
import string


class TestSentencePalindrome(unittest.TestCase):


    # Palindrome Sentences
    def test_palindrome_sentences(self):
        self.assertTrue(is_sentence_palindrome("A man a plan a canal
Panama"))
        self.assertTrue(is_sentence_palindrome("Madam In Eden, I'm
Adam"))
        self.assertTrue(is_sentence_palindrome("Was it a car or a cat I
saw?"))

    # Non-Palindrome Sentences
    def test_non_palindrome_sentences(self):
        self.assertFalse(is_sentence_palindrome("Hello world"))
        self.assertFalse(is_sentence_palindrome("Python is fun"))

    # Edge Cases
    def test_empty_string(self):
        self.assertTrue(is_sentence_palindrome(""))

    def test_single_character(self):
        self.assertTrue(is_sentence_palindrome("A"))

    def test_non_string_input(self):
        self.assertFalse(is_sentence_palindrome(123))

def is_sentence_palindrome(sentence):


    # Check if input is string
    if not isinstance(sentence, str):
        return False

    # Convert to lowercase
    sentence = sentence.lower()

    # Remove spaces and punctuation
    cleaned = ""
```

```
for char in sentence:
    if char.isalnum():      # keeps only letters and numbers
        cleaned += char

# Check palindrome
return cleaned == cleaned[::-1]

if __name__ == "__main__":
    unittest.main()
```

Output:

The screenshot shows the Visual Studio Code interface with the following details:

- EXPLORER**: Shows a tree view with "ASSIGNMENT 8" expanded, containing files: task2.py, task3.py, and tast1.py.
- Taskbar**: Shows tabs for "Welcome", "task1.py", "task2.py", and "task3.py".
- Code Editor**: Displays Python code for testing sentence palindromes. The code uses the unittest module to assert various sentence inputs as palindromes or non-palindromes.
- Terminal**: Shows the command line output of running the task3.py file, indicating the execution of tests and the result "Ran 5 tests in 0.003s".

```
task3.py > ...
1 import unittest
2 import string
3
4
5 class TestSentencePalindrome(unittest.TestCase):
6
7     # Palindrome Sentences
8     def test_palindrome_sentences(self):
9         self.assertTrue(is_sentence_palindrome("A man a plan a canal Panama"))
10        self.assertTrue(is_sentence_palindrome("Madam In Eden, I'm Adam"))
11        self.assertTrue(is_sentence_palindrome("Was it a car or a cat I saw?"))
12
13    # Non-Palindrome Sentences
14    def test_non_palindrome_sentences(self):
15        self.assertFalse(is_sentence_palindrome("Hello world"))
16        self.assertFalse(is_sentence_palindrome("Python is fun"))
17
18    # Edge Cases
19    def test_empty_string(self):
20        self.assertTrue(is_sentence_palindrome(""))
21
22    def test_single_character(self):
23        self.assertTrue(is_sentence_palindrome("A"))
24
25    def test_non_string_input(self):
26        self.assertFalse(is_sentence_palindrome(123))
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS Python Debug Console + ×

```
soft\WindowsApps\python3.11.exe' 'c:\Users\ANDU RAHUL\.vscode\extensions\ms-python.debugpy-2025.18.0-win32-x64\bundled\libs\debugpy\launcher' '55999' '-' 'C:\Users\ANDU RAHUL\OneDrive\Desktop\Documents\Desktop\ai lab\assignment 8\task3.py'
.....
Ran 5 tests in 0.003s
OK
```

Analysis:

The program checks if a sentence is a palindrome.

It uses a function called `is_sentence_palindrome(sentence)`.

First, it checks if the input is a string.

It converts all letters to lowercase.

It removes spaces and punctuation marks.

It keeps only letters and numbers.

It reverses the cleaned sentence.

It compares the original cleaned sentence with the reversed one.

If both are the same → return True.

If not the same → return False.

Test cases check palindrome and non-palindrome sentences.

All tests must pass to confirm it works correctly.

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

Prompt:

Create a class `ShoppingCart`.

Use Test-Driven Development (write test cases first).

Add method `add_item(name, price)`.

Add method `remove_item(name)`.

Add method `total_cost()`.

Check item addition works correctly.

Check item removal works correctly.

Check total cost calculation is correct.

Handle empty cart case.

All test cases must pass.

Code:

```
import unittest

class TestShoppingCart(unittest.TestCase):

    def setUp(self):
        self.cart = ShoppingCart()

    def test_add_item(self):
        self.cart.add_item("Book", 500)
        self.assertEqual(self.cart.total_cost(), 500)
```

```

def test_remove_item(self):
    self.cart.add_item("Pen", 50)
    self.cart.remove_item("Pen")
    self.assertEqual(self.cart.total_cost(), 0)

def test_total_cost_multiple_items(self):
    self.cart.add_item("Shirt", 1000)
    self.cart.add_item("Shoes", 2000)
    self.assertEqual(self.cart.total_cost(), 3000)

def test_empty_cart(self):
    self.assertEqual(self.cart.total_cost(), 0)

def test_remove_non_existing_item(self):
    self.cart.add_item("Bag", 800)
    self.cart.remove_item("Shoes")    # not in cart
    self.assertEqual(self.cart.total_cost(), 800)

# =====
# Step 2: Implementation
# =====

class ShoppingCart:

    def __init__(self):
        self.items = []

    def add_item(self, name, price):
        self.items.append({"name": name, "price": price})

    def remove_item(self, name):
        for item in self.items:
            if item["name"] == name:
                self.items.remove(item)
                break

    def total_cost(self):
        total = 0
        for item in self.items:
            total += item["price"]
        return total

```

```
if __name__ == "__main__":
    unittest.main()
```

Output:

The screenshot shows the Visual Studio Code interface with the following details:

- EXPLORER:** Shows files in the project: task2.py, task3.py, task4.py, and task1.py.
- CODE EDITOR:** Displays the content of task4.py, which contains a class TestShoppingCart that inherits from unittest.TestCase. It includes several test methods: test_add_item, test_remove_item, test_total_cost_multiple_items, test_empty_cart, and test_remove_non_existing_item.
- TERMINAL:** Shows the command run in the terminal: "soft\WindowsApps\python3.11.exe" "c:\Users\ANDU RAHUL\.vscode\extensions\ms-python.debugpy-2025.18.0-win32-x64\bundled\libs\debugpy\launcher" "55999" "--" "c:\Users\ANDU RAHUL\OneDrive\Desktop\Documents\Desktop\ai_lab\assignment 8\task3.py". The output indicates that 5 tests were run in 0.003 seconds.

Analysis:

The program creates a shopping cart system.

Items are stored inside a list.

`add_item()` adds item name and price.

`remove_item()` removes item by name.

If item is not found, nothing happens.

`total_cost()` adds all item prices.

If cart is empty, total cost is 0.

Test cases check add, remove, and total cost.

Edge case: removing from empty cart.

All tests must pass to confirm correctness.

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

Prompt :

Write a Python function `convert_date_format(date_str)`.

Use Test-Driven Development (write test cases first).

Input format must be "YYYY-MM-DD".

Output format must be "DD-MM-YYYY".

Example: "2023-10-15" → "15-10-2023".

Test valid and invalid date inputs.

All test cases must pass.

Code:

```
import unittest

class TestDateConversion(unittest.TestCase):

    def test_valid_dates(self):
        self.assertEqual(convert_date_format("2023-10-15"),
"15-10-2023")
        self.assertEqual(convert_date_format("2000-01-01"),
"01-01-2000")

    def test_invalid_format(self):
        self.assertEqual(convert_date_format("15-10-2023"), "Invalid")
        self.assertEqual(convert_date_format("2023/10/15"), "Invalid")

    def test_non_string_input(self):
        self.assertEqual(convert_date_format(20231015), "Invalid")

    def test_incomplete_date(self):
        self.assertEqual(convert_date_format("2023-10"), "Invalid")

def convert_date_format(date_str):

    if not isinstance(date_str, str):
        return "Invalid"

    parts = date_str.split("-")
```

```

if len(parts) != 3:
    return "Invalid"

year, month, day = parts

# Basic length validation
if len(year) != 4 or len(month) != 2 or len(day) != 2:
    return "Invalid"

return f"{day}-{month}-{year}"

if __name__ == "__main__":
    unittest.main()

```

output:

The screenshot shows the Visual Studio Code interface. The left sidebar displays a project structure under 'EXPLORER' with files like task2.py, task3.py, task4.py, task5.py, and tast1.py. The main code editor area shows the content of task5.py, which contains a class TestDateConversion with several test methods. The terminal at the bottom shows the command 'python task5.py' being run, followed by the output of the test results.

```

1 import unittest
2
3
4
5 class TestDateConversion(unittest.TestCase):
6
7     def test_valid_dates(self):
8         self.assertEqual(convert_date_format("2023-10-15"), "15-10-2023")
9         self.assertEqual(convert_date_format("2000-01-01"), "01-01-2000")
10
11     def test_invalid_format(self):
12         self.assertEqual(convert_date_format("15-10-2023"), "Invalid")
13         self.assertEqual(convert_date_format("2023/10/15"), "Invalid")
14
15     def test_non_string_input(self):
16         self.assertEqual(convert_date_format(20231015), "Invalid")
17
18     def test_incomplete_date(self):
19         self.assertEqual(convert_date_format("2023-10"), "Invalid")
20
21     def convert_date_format(date_str):
22
23         if not isinstance(date_str, str):
24             return "Invalid"
25
26         parts = date_str.split("-")

```

```

soft\WindowsApps\python3.11.exe` 'c:\Users\ANDU RAHUL\.vscode\extensions\ms-python.debugpy-2025.18.0-win
32-x64\bundled\libs\debugpy\launcher' '62529' '--' 'C:\Users\ANDU RAHUL\OneDrive\Desktop\Documents\Desktop\ai lab\assignment 8\task5.py'
.....
-----
Ran 4 tests in 0.003s
OK
PS C:\Users\ANDU RAHUL\OneDrive\Desktop\Documents\Desktop\ai lab\assignment 8>

```

Analysis:

The function takes a date string as input.
It checks if the input is a string.
It verifies the format is YYYY-MM-DD.
It splits the date using "-".
It rearranges the parts into DD-MM-YYYY.
If the format is incorrect, return "Invalid".
Test cases check valid and invalid inputs.
All tests must pass to confirm correctness.

