

Lab assignment-8.3

2303A51228

Batch-26

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:

Using unittest implement is_valid_email(email) following four TDD styles: Red Green Refactor cycle, behaviour driven tests for valid and invalid emails, boundary and edge case tests like starting or ending with special characters and multiple @, and error driven tests for wrong types, ensure all tests pass.

Code:

```
import unittest
from typing import Union

def is_valid_email(email: Union[str, int, None]) -> bool:

    if not isinstance(email, str):
        raise TypeError(f"Email must be a string, not {type(email).__name__}")

    if not email or email.startswith(('@', '.', '-')) or email.endswith('@', '.')):
```

```

        return False

    if email.count('@') != 1:
        return False

    local, domain = email.split('@')

    if not local or not domain or '.' not in domain:
        return False

    if domain.startswith('.') or domain.endswith('.'):
        return False

    if domain.count('..') > 0 or local.count(..) > 0:
        return False

    return True

```

```

class TestIsValidEmailRGR(unittest.TestCase):
    """Red Green Refactor cycle tests"""

    def test_valid_simple_email(self):
        self.assertTrue(is_valid_email("user@example.com"))

    def test_valid_email_with_numbers(self):
        self.assertTrue(is_valid_email("user123@example.com"))

    def test_invalid_no_at_sign(self):
        self.assertFalse(is_valid_email("userexample.com"))

    def test_invalid_no_domain_extension(self):
        self.assertFalse(is_valid_email("user@example"))

```

```

class TestIsValidEmailBDD(unittest.TestCase):
    """Behaviour Driven Development tests"""

    def test_valid_emails(self):
        valid_emails = [
            "john@example.com",
            "alice.smith@company.org",
            "bob_99@domain.co.uk",
            "test+tag@email.net"
        ]
        for email in valid_emails:
            with self.subTest(email=email):
                self.assertTrue(is_valid_email(email))

    def test_invalid_emails(self):
        invalid_emails = [
            "plainaddress",
            "user@",
            "@example.com",
            "user@@example.com",
            "user@.com",
            "user.example.com"
        ]
        for email in invalid_emails:
            with self.subTest(email=email):
                self.assertFalse(is_valid_email(email))

```

```
        self.assertFalse(is_valid_email(email))
```

```
class TestIsValidEmailBoundary(unittest.TestCase):
    """Boundary and edge case tests"""

    def test_starts_with_special_characters(self):
        self.assertFalse(is_valid_email("@user@example.com"))
        self.assertFalse(is_valid_email(".user@example.com"))
        self.assertFalse(is_valid_email("-user@example.com"))

    def test_ends_with_special_characters(self):
        self.assertFalse(is_valid_email("user@example.com."))
        self.assertFalse(is_valid_email("user@example.com@"))

    def test_multiple_at_signs(self):
        self.assertFalse(is_valid_email("user@@example.com"))
        self.assertFalse(is_valid_email("user@exam@ple.com"))

    def test_consecutive_dots(self):
        self.assertFalse(is_valid_email("user..name@example.com"))
        self.assertFalse(is_valid_email("user@exam..ple.com"))

    def test_empty_string(self):
        self.assertFalse(is_valid_email(""))

    def test_single_character_local_and_domain(self):
        self.assertTrue(is_valid_email("a@b.co"))
```

```
class TestIsValidEmailErrorDriven(unittest.TestCase):
    """Error driven tests for wrong types"""

    def test_none_type(self):
        with self.assertRaises(TypeError):
            is_valid_email(None)

    def test_integer_type(self):
        with self.assertRaises(TypeError):
            is_valid_email(12345)

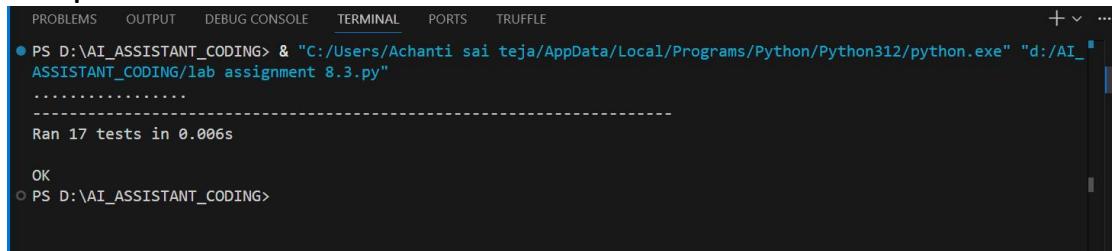
    def test_list_type(self):
        with self.assertRaises(TypeError):
            is_valid_email(["user@example.com"])

    def test_dict_type(self):
        with self.assertRaises(TypeError):
            is_valid_email({"email": "user@example.com"})

    def test_float_type(self):
        with self.assertRaises(TypeError):
            is_valid_email(123.45)
```

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

Output:



A screenshot of a terminal window titled "TERMINAL". The window shows the command "PS D:\AI_ASSISTANT_CODING> & "C:/Users/Achandi sai teja/AppData/Local/Programs/Python/Python312/python.exe" "d:/AI_ASSISTANT_CODING/lab_assignment_8.3.py"" being run. The output indicates that 17 tests were run in 0.006 seconds and resulted in "OK".

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS TRUFFLE
● PS D:\AI_ASSISTANT_CODING> & "C:/Users/Achandi sai teja/AppData/Local/Programs/Python/Python312/python.exe" "d:/AI_ASSISTANT_CODING/lab_assignment_8.3.py"
-----
Ran 17 tests in 0.006s
OK
○ PS D:\AI_ASSISTANT_CODING>
```

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:

Using unittest implement assign_grade(score) with four TDD approaches: Red Green Refactor, behaviour driven grade mapping A(90-100) B(80-89) C(70-79) D(60-69) F(<60), boundary tests for 60 70 80 90, and error driven tests for invalid inputs -5 105 and string values, ensure all tests pass.

Code:

```
import unittest
```

```

def assign_grade(score):
    """
    Assign a letter grade based on a numerical score.

    Args:
        score: numerical score (0-100)

    Returns:
        Letter grade (A, B, C, D, F)

    Raises:
        TypeError: if score is not a number
        ValueError: if score is outside 0-100 range
    """
    if not isinstance(score, (int, float)) or isinstance(score, bool):
        raise TypeError("Score must be a number")

    if score < 0 or score > 100:
        raise ValueError("Score must be between 0 and 100")

    if score >= 90:
        return 'A'
    elif score >= 80:
        return 'B'
    elif score >= 70:
        return 'C'
    elif score >= 60:
        return 'D'
    else:
        return 'F'

```

```

class TestAssignGrade(unittest.TestCase):
    """Test cases for assign_grade function using TDD approaches"""

    # Red Green Refactor: Basic grade mapping
    def test_grade_a(self):
        self.assertEqual(assign_grade(95), 'A')

    def test_grade_b(self):
        self.assertEqual(assign_grade(85), 'B')

    def test_grade_c(self):
        self.assertEqual(assign_grade(75), 'C')

    def test_grade_d(self):
        self.assertEqual(assign_grade(65), 'D')

    def test_grade_f(self):
        self.assertEqual(assign_grade(55), 'F')

    # Behaviour Driven: Complete grade range mapping
    def test_grade_a_range(self):
        self.assertEqual(assign_grade(90), 'A')
        self.assertEqual(assign_grade(100), 'A')

    def test_grade_b_range(self):
        self.assertEqual(assign_grade(80), 'B')
        self.assertEqual(assign_grade(89), 'B')

```

```

def test_grade_c_range(self):
    self.assertEqual(assign_grade(70), 'C')
    self.assertEqual(assign_grade(79), 'C')

def test_grade_d_range(self):
    self.assertEqual(assign_grade(60), 'D')
    self.assertEqual(assign_grade(69), 'D')

def test_grade_f_range(self):
    self.assertEqual(assign_grade(0), 'F')
    self.assertEqual(assign_grade(59), 'F')

# Boundary Tests
def test_boundary_60(self):
    self.assertEqual(assign_grade(60), 'D')
    self.assertEqual(assign_grade(59), 'F')

def test_boundary_70(self):
    self.assertEqual(assign_grade(70), 'C')
    self.assertEqual(assign_grade(69), 'D')

def test_boundary_80(self):
    self.assertEqual(assign_grade(80), 'B')
    self.assertEqual(assign_grade(79), 'C')

def test_boundary_90(self):
    self.assertEqual(assign_grade(90), 'A')
    self.assertEqual(assign_grade(89), 'B')

# Error Driven Tests: Invalid inputs
def test_invalid_negative_score(self):
    with self.assertRaises(ValueError):
        assign_grade(-5)

def test_invalid_score_over_100(self):
    with self.assertRaises(ValueError):
        assign_grade(105)

def test_invalid_string_input(self):
    with self.assertRaises(TypeError):
        assign_grade("85")

def test_invalid_none_input(self):
    with self.assertRaises(TypeError):
        assign_grade(None)

def test_invalid_boolean_input(self):
    with self.assertRaises(TypeError):
        assign_grade(True)

```

```

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

```

Output:

```

● PS D:\AI_ASSISTANT_CODING> & "C:/Users/Achanti sai teja/AppData/Local/Programs/Python/Python312/python.exe" "d:/AI_ASSISTANT_CODING/lab assignment 8.3.py"
.....
-----
Ran 19 tests in 0.004s
ok

```

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:

Using `unittest` implement `is_sentence_palindrome(sentence)` applying four TDD styles: Red Green Refactor, behaviour driven palindrome detection ignoring case spaces punctuation, boundary tests for empty and single word sentences, and error driven tests for non string input, ensure all tests pass.

Code:

```
import unittest
import re

def is_sentence_palindrome(sentence):
    """
    Check if a sentence is a palindrome, ignoring case, spaces, and punctuation.

    Args:
        sentence: A string to check

    Returns:
        Boolean indicating if the sentence is a palindrome

    Raises:
    """

    # Remove punctuation and convert to lowercase
    sentence = re.sub(r'[^\w\s]', '', sentence).lower()

    # Check if the sentence is equal to its reverse
    return sentence == sentence[::-1]
```

```

        TypeError: If input is not a string
"""
if not isinstance(sentence, str):
    raise TypeError("Input must be a string")

# Remove non-alphanumeric characters and convert to lowercase
cleaned = re.sub(r'[^a-zA-Z]', '', sentence.lower())

# Check if cleaned string is equal to its reverse
return cleaned == cleaned[::-1]

```

```

class TestIsSentencePalindrome(unittest.TestCase):
    """Test suite for is_sentence_palindrome function using TDD styles"""

    # Red-Green-Refactor Tests
    def test_simple_palindrome(self):
        """Test basic palindrome detection"""
        self.assertTrue(is_sentence_palindrome("racecar"))
        self.assertTrue(is_sentence_palindrome("A man a plan a canal Panama"))

    def test_non_palindrome(self):
        """Test non-palindrome detection"""
        self.assertFalse(is_sentence_palindrome("hello"))
        self.assertFalse(is_sentence_palindrome("python"))

    # Behaviour-Driven Tests
    def test_ignores_case(self):
        """Should ignore uppercase and lowercase"""
        self.assertTrue(is_sentence_palindrome("RaceCar"))
        self.assertTrue(is_sentence_palindrome("MaDaM"))

    def test_ignores_spaces(self):
        """Should ignore spaces between words"""
        self.assertTrue(is_sentence_palindrome("race car"))
        self.assertTrue(is_sentence_palindrome("a b a"))

    def test_ignores_punctuation(self):
        """Should ignore punctuation marks"""
        self.assertTrue(is_sentence_palindrome("Was it a car or a cat I saw?"))
        self.assertTrue(is_sentence_palindrome("A Santa at NASA"))

    def test_combined_case_spaces_punctuation(self):
        """Should handle all three simultaneously"""
        self.assertTrue(is_sentence_palindrome("Madam, I'm Adam!"))
        self.assertTrue(is_sentence_palindrome("Never odd or even."))

    # Boundary Tests
    def test_empty_string(self):
        """Empty string is a palindrome"""
        self.assertTrue(is_sentence_palindrome(""))

    def test_single_character(self):
        """Single character is always a palindrome"""
        self.assertTrue(is_sentence_palindrome("a"))
        self.assertTrue(is_sentence_palindrome("z"))

    def test_single_word(self):
        """Single word palindrome and non-palindrome"""
        self.assertTrue(is_sentence_palindrome("noon"))
        self.assertFalse(is_sentence_palindrome("world"))

```

```

def test_spaces_only(self):
    """String with only spaces is a palindrome"""
    self.assertTrue(is_sentence_palindrome("   "))

def test_punctuation_only(self):
    """String with only punctuation is a palindrome"""
    self.assertTrue(is_sentence_palindrome("!@#$%"))

# Error-Driven Tests
def test_non_string_integer(self):
    """Should raise TypeError for integer input"""
    with self.assertRaises(TypeError):
        is_sentence_palindrome(123)

def test_non_string_list(self):
    """Should raise TypeError for list input"""
    with self.assertRaises(TypeError):
        is_sentence_palindrome(['a', 'b', 'a'])

def test_non_string_none(self):
    """Should raise TypeError for None input"""
    with self.assertRaises(TypeError):
        is_sentence_palindrome(None)

def test_non_string_boolean(self):
    """Should raise TypeError for boolean input"""
    with self.assertRaises(TypeError):
        is_sentence_palindrome(True)

```

```

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

```

Output:

```

PS D:\AI ASSISTANT CODING> & "C:/Users/Achanti sai teja/AppData/Local/Programs/Python/Python312/py
ASSISTANT_CODING/lab_assignment_8.3.py"
-----
Ran 15 tests in 0.004s
OK

```

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:

Using unittest implement ShoppingCart class with add_item remove_item total_cost using four TDD types: Red Green Refactor, behaviour driven tests for add remove and total cost, boundary tests for empty cart and single item, and error driven tests for invalid price and missing items, ensure all tests pass.

Code:

```
import unittest
from unittest.mock import Mock

class ShoppingCart:
    """A simple shopping cart implementation."""

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

    def add_item(self, name, price, quantity=1):
        """Add an item to the cart."""
        if price < 0:
            raise ValueError("Price cannot be negative")
        if not isinstance(name, str) or not name.strip():
            raise ValueError("Item name must be a non-empty string")
        if quantity <= 0:
            raise ValueError("Quantity must be positive")

        self.items.append({
            'name': name,
            'price': price,
            'quantity': quantity
        })

    def remove_item(self, name):
        """Remove an item from the cart."""
        initial_length = len(self.items)
        self.items = [item for item in self.items if item['name'] != name]
```

```

        if len(self.items) == initial_length:
            raise ValueError(f"Item '{name}' not found in cart")

    def total_cost(self):
        """Calculate the total cost of items in the cart."""
        return sum(item['price'] * item['quantity'] for item in self.items)

```

```

class TestShoppingCartBDD(unittest.TestCase):
    """Behaviour Driven Development tests."""

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

    def test_add_item_successfully(self):
        self.cart.add_item("Apple", 1.50, 2)
        self.assertEqual(len(self.cart.items), 1)

    def test_remove_item_successfully(self):
        self.cart.add_item("Apple", 1.50)
        self.cart.remove_item("Apple")
        self.assertEqual(len(self.cart.items), 0)

    def test_total_cost_calculation(self):
        self.cart.add_item("Apple", 1.50, 2)
        self.cart.add_item("Banana", 0.75, 3)
        self.assertAlmostEqual(self.cart.total_cost(), 5.25)

```

```

class TestShoppingCartBoundary(unittest.TestCase):
    """Boundary tests."""

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

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

    def test_single_item_in_cart(self):
        self.cart.add_item("Apple", 1.50, 1)
        self.assertEqual(len(self.cart.items), 1)
        self.assertAlmostEqual(self.cart.total_cost(), 1.50)

    def test_zero_price_item(self):
        self.cart.add_item("Free Sample", 0, 1)
        self.assertAlmostEqual(self.cart.total_cost(), 0)

```

```

class TestShoppingCartErrorHandler(unittest.TestCase):
    """Error-driven tests."""

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

    def test_negative_price_raises_error(self):
        with self.assertRaises(ValueError):
            self.cart.add_item("Apple", -1.50)

    def test_invalid_item_name_raises_error(self):

```

```

        with self.assertRaises(ValueError):
            self.cart.add_item("", 1.50)

    def test_zero_quantity_raises_error(self):
        with self.assertRaises(ValueError):
            self.cart.add_item("Apple", 1.50, 0)

    def test_remove_nonexistent_item_raises_error(self):
        with self.assertRaises(ValueError):
            self.cart.remove_item("Nonexistent")

    def test_remove_item_from_empty_cart_raises_error(self):
        with self.assertRaises(ValueError):
            self.cart.remove_item("Apple")

```

```

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

```

Output:

```

● PS D:\AI_ASSISTANT_CODING> & "C:/Users/Achanti sai teja/AppData/Local/Programs/Python/Python312/python.exe" "d:/AI_ASSISTANT_CODING/lab_assignment 8.3.py"
.....
-----
Ran 11 tests in 0.004s
OK
● PS D:\AI_ASSISTANT_CODING>

```

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:

Using unittest implement convert_date_format(date_str) converting YYYY-MM-DD to DD-MM-YYYY using four TDD styles: Red Green Refactor, behaviour driven correct conversion, boundary tests for leap year and month day limits, and error driven tests for wrong format and invalid input, ensure all tests pass.

Code:

```
import unittest
from datetime import datetime

def convert_date_format(date_str):
    """Convert date from YYYY-MM-DD to DD-MM-YYYY format."""
    if not isinstance(date_str, str):
        raise TypeError("Input must be a string")

    if len(date_str) != 10 or date_str[4] != '-' or date_str[7] != '-':
        raise ValueError("Invalid date format. Expected YYYY-MM-DD")

    try:
        year, month, day = date_str.split('-')

        if not (year.isdigit() and month.isdigit() and day.isdigit()):
            raise ValueError("Date components must be numeric")

        year, month, day = int(year), int(month), int(day)

        if month < 1 or month > 12:
            raise ValueError("Month must be between 01 and 12")

        if day < 1 or day > 31:
            raise ValueError("Day must be between 01 and 31")

        # Validate actual date
        datetime(year, month, day)

    return f"{day:02d}-{month:02d}-{year}"

    except ValueError as e:
        raise ValueError(str(e))
```

```
class TestConvertDateFormat(unittest.TestCase):

    # Red Green Refactor Style
    def test_basic_conversion(self):
        """Test basic date conversion from YYYY-MM-DD to DD-MM-YYYY"""
        self.assertEqual(convert_date_format("2023-05-15"), "15-05-2023")

    def test_single_digit_month_and_day(self):
        """Test conversion with zero-padded output"""
        self.assertEqual(convert_date_format("2023-01-05"), "05-01-2023")

    # Behaviour Driven Tests
    def test_correct_conversion_spring_date(self):
```

```

"""Given a spring date, convert correctly"""
self.assertEqual(convert_date_format("2023-03-20"), "20-03-2023")

def test_correct_conversion_summer_date(self):
    """Given a summer date, convert correctly"""
    self.assertEqual(convert_date_format("2023-06-15"), "15-06-2023")

def test_correct_conversion_autumn_date(self):
    """Given an autumn date, convert correctly"""
    self.assertEqual(convert_date_format("2023-09-30"), "30-09-2023")

def test_correct_conversion_winter_date(self):
    """Given a winter date, convert correctly"""
    self.assertEqual(convert_date_format("2023-12-25"), "25-12-2023")

# Boundary Tests - Leap Year and Month/Day Limits
def test_leap_year_feb_29(self):
    """Test leap year February 29th during leap year"""
    self.assertEqual(convert_date_format("2020-02-29"), "29-02-2020")

def test_non_leap_year_feb_29(self):
    """Test February 29th during non-leap year raises error"""
    with self.assertRaises(ValueError):
        convert_date_format("2023-02-29")

def test_january_first(self):
    """Test boundary: first day of year"""
    self.assertEqual(convert_date_format("2023-01-01"), "01-01-2023")

def test_december_31st(self):
    """Test boundary: last day of year"""
    self.assertEqual(convert_date_format("2023-12-31"), "31-12-2023")

def test_april_30_days(self):
    """Test month with 30 days"""
    self.assertEqual(convert_date_format("2023-04-30"), "30-04-2023")

def test_april_31_invalid(self):
    """Test day 31 in 30-day month raises error"""
    with self.assertRaises(ValueError):
        convert_date_format("2023-04-31")

# Error Driven Tests - Wrong Format and Invalid Input
def test_wrong_format_slash_separator(self):
    """Test error handling for wrong separator (slash)"""
    with self.assertRaises(ValueError):
        convert_date_format("2023/05/15")

def test_wrong_format_no_separator(self):
    """Test error handling for missing separator"""
    with self.assertRaises(ValueError):
        convert_date_format("20230515")

def test_invalid_month_13(self):
    """Test error handling for invalid month"""
    with self.assertRaises(ValueError):
        convert_date_format("2023-13-15")

def test_invalid_month_00(self):
    """Test error handling for month 0"""
    with self.assertRaises(ValueError):

```

```

        convert_date_format("2023-00-15")

def test_invalid_day_32(self):
    """Test error handling for day 32"""
    with self.assertRaises(ValueError):
        convert_date_format("2023-05-32")

def test_invalid_day_00(self):
    """Test error handling for day 0"""
    with self.assertRaises(ValueError):
        convert_date_format("2023-05-00")

def test_non_string_input(self):
    """Test error handling for non-string input"""
    with self.assertRaises(TypeError):
        convert_date_format(20230515)

def test_non_string_list_input(self):
    """Test error handling for list input"""
    with self.assertRaises(TypeError):
        convert_date_format(["2023", "05", "15"])

def test_non_numeric_year(self):
    """Test error handling for non-numeric year"""
    with self.assertRaises(ValueError):
        convert_date_format("abcd-05-15")

def test_non_numeric_month(self):
    """Test error handling for non-numeric month"""
    with self.assertRaises(ValueError):
        convert_date_format("2023-ab-15")

def test_non_numeric_day(self):
    """Test error handling for non-numeric day"""
    with self.assertRaises(ValueError):
        convert_date_format("2023-05-ab")

def test_empty_string(self):
    """Test error handling for empty string"""
    with self.assertRaises(ValueError):
        convert_date_format("")

def test_wrong_length(self):
    """Test error handling for incorrect length"""
    with self.assertRaises(ValueError):
        convert_date_format("2023-5-15")

```

```

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

```

Output:

```

PS D:\AI_ASSISTANT_CODING> & "C:/Users/Achanti sai teja/AppData/Local/Programs/Python/Python312/python.exe" "d:/AI_ASSISTANT_CODING/lab assignment 8.3.py"
-----
Ran 25 tests in 0.005s
OK
PS D:\AI_ASSISTANT_CODING>

```