

ASSIGNMENT-8.3

NAME:THATIKONDA SASYA

BATCH:20

HTNO:2303A51346

TASK1:

```
import re

def is_valid_email(email: str) -> bool:
    if not isinstance(email, str) or not email:
        return False

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

    if '.' not in email:
        return False

    if not email[0].isalnum() or not email[-1].isalnum():
        return False

    if '..' in email:
        return False

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

    if not local or not domain:
        return False

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

    if '.' not in domain:
        return False

    pattern = r'^[A-Za-z0-9._]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}$'
    return bool(re.fullmatch(pattern, email))
```

```

def run_tests():

    # Valid Emails
    "user@example.com": True,
    "john.doe@gmail.com": True,
    "user_123@test.co.in": True,
    "a@b.com": True,

    # Invalid Emails
    "userexample.com": False,
    "user@com": False,
    "@example.com": False,
    "user.@example.com": False,
    ".user@example.com": False,
    "user@@example.com": False,
    "user@domain..com": False,
    "": False
}

print("Running Email Validation Test Cases:\n")

passed = 0
total = len(test_cases)

for email, expected in test_cases.items():
    result = is_valid_email(email)
    status = "PASS" if result == expected else "FAIL"
    print(f"{email:25} Expected: {expected} Got: {result} --> {status}")

    if status == "PASS":
        passed += 1


print("\nSummary:")
print(f"{passed}/{total} test cases passed.")

```

```
user@example.com      Expected: True  Got: True  --> PASS
john.doe@gmail.com    Expected: True  Got: True  --> PASS
user_123@test.co.in  Expected: True  Got: True  --> PASS
a@b.com               Expected: True  Got: True  --> PASS
userexample.com       Expected: False Got: False  --> PASS
user@com              Expected: False Got: False  --> PASS
@example.com          Expected: False Got: False  --> PASS
user.@example.com     Expected: False Got: False  --> PASS
.user@example.com     Expected: False Got: False  --> PASS
user@@example.com     Expected: False Got: False  --> PASS
user@domain..com      Expected: False Got: False  --> PASS
Expected: False  Got: False  --> PASS
```

Summary:

12/12 test cases passed.

All AI-generated test cases passed successfully 

JUSTIFICATION:

The function enforces all specified requirements, including the presence of "@" and ".", restriction of special characters at the beginning or end, and prevention of consecutive dots. By validating both local and domain parts carefully, the implementation ensures malformed emails are rejected while valid ones return True. All AI-generated test cases pass successfully, confirming the correctness of the solution.

TASK 2:

```
def run_tests():
    test_cases = [
        # Normal cases
        (95, "A"),
        (85, "B"),
        (75, "C"),
        (65, "D"),
        (50, "F"),

        # Boundary values
        (90, "A"),
        (80, "B"),
        (70, "C"),
        (60, "D"),
        (59, "F"),
        (100, "A"),
        (0, "F"),

        # Invalid inputs
        (-5, "Error"),
        (105, "Error"),
        ("eighty", "Error"),
    ]

    passed = 0

    for score, expected in test_cases:
        try:
            result = assign_grade(score)
```

```

        result = assign_grade(score)
    except Exception:
        result = "Error"

    status = "PASS" if result == expected else "FAIL"

    print(f"{str(score):<10} Expected: {expected:<6} Got: {result:<6} --> {status}")

    if status == "PASS":
        passed += 1

print("\nSummary:")
print(f"{passed}/{len(test_cases)} test cases passed.")

if passed == len(test_cases):
    print("All AI-generated test cases passed successfully ✅")

# =====
# Step 2: Implement Function
# (Written after defining tests)
# =====

def assign_grade(score):
    # Type validation
    if not isinstance(score, (int, float)):
        raise TypeError("Score must be a number")

```

```
# Type validation
if not isinstance(score, (int, float)):
    raise TypeError("Score must be a number")

# Range validation
if not 0 <= score <= 100:
    raise ValueError("Score must be between 0 and 100")

# Grade assignment
if score >= 90:
    return "A"
elif score >= 80:
    return "B"
elif score >= 70:
    return "C"
elif score >= 60:
    return "D"
else:
    return "F"

# =====
# Run Tests
# =====

if __name__ == "__main__":
    run_tests()
```

```

pattern = r'^([A-Za-z0-9._-]+@[A-Za-z0-9._-]+\.[A-Za-z]{2,})$'
95      Expected: A      Got: A      --> PASS
85      Expected: B      Got: B      --> PASS
75      Expected: C      Got: C      --> PASS
65      Expected: D      Got: D      --> PASS
50      Expected: F      Got: F      --> PASS
90      Expected: A      Got: A      --> PASS
80      Expected: B      Got: B      --> PASS
70      Expected: C      Got: C      --> PASS
60      Expected: D      Got: D      --> PASS
59      Expected: F      Got: F      --> PASS
100     Expected: A      Got: A      --> PASS
0       Expected: F      Got: F      --> PASS
-5      Expected: Error  Got: Error  --> PASS
105     Expected: Error  Got: Error  --> PASS
eighty  Expected: Error  Got: Error  --> PASS

```

JUSTIFICATION:

This grading system follows the Test-Driven Development (TDD) approach by defining comprehensive test cases before implementing the `assign_grade` function. Boundary values (60, 70, 80, 90) are explicitly tested to prevent off-by-one and range overlap errors. Invalid inputs such as negative numbers, out-of-range values, and non-numeric types are handled using proper exception handling to ensure robustness. All AI-generated test cases pass successfully, confirming correctness and reliability of the implementation.

TASK3:

```

import re

# =====
# Step 1: Define Tests (TDD)
# =====

def run_tests():
    test_cases = [
        # Palindromes
        ("A man a plan a canal Panama", True),
        ("Madam", True),
        ("Was it a car or a cat I saw?", True),
        ("No lemon, no melon", True),
        ("Able was I, ere I saw Elba", True),

        # Non-palindromes
        ("Hello world", False),
        ("OpenAI is great", False),
        ("Palindrome test", False),

        # Edge cases
        ("", True),          # Empty string is palindrome
        ("!!!", True),       # Only punctuation
    ]

    passed = 0

    for sentence, expected in test_cases:
        result = is_sentence_palindrome(sentence)
        status = "PASS" if result == expected else "FAIL"

        print(f"{sentence:<35} Expected: {expected:<5} Got: {result:<5} --> {status}")

```



```

        if status == "PASS":
            passed += 1

    print("\nSummary:")
    print(f"{passed}/{len(test_cases)} test cases passed.")

    if passed == len(test_cases):
        print("All AI-generated test cases passed successfully ✅")

# =====
# Step 2: Implement Function
# =====

def is_sentence_palindrome(sentence):
    # Remove punctuation and spaces, convert to lowercase
    cleaned = re.sub(r'[^A-Za-z0-9]', '', sentence).lower()

    # Compare with reverse
    return cleaned == cleaned[::-1]

# =====
# Run Tests
# =====

if __name__ == "__main__":
    run_tests()
    pattern = r'^[A-Za-z0-9._]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}$'
    A man a plan a canal Panama           Expected: 1      Got: 1      --> PASS
    Madam                                 Expected: 1      Got: 1      --> PASS
    Was it a car or a cat I saw?          Expected: 1      Got: 1      --> PASS
    No lemon, no melon                    Expected: 1      Got: 1      --> PASS
    Able was I, ere I saw Elba            Expected: 1      Got: 1      --> PASS
    Hello world                           Expected: 0      Got: 0      --> PASS
    OpenAI is great                       Expected: 0      Got: 0      --> PASS
    Palindrome test                       Expected: 0      Got: 0      --> PASS
    !!!                                   Expected: 1      Got: 1      --> PASS
    !!!                                   Expected: 1      Got: 1      --> PASS

```

JUSTIFICATION:

This implementation follows a test-driven approach by defining multiple palindromic and non-palindromic test cases before validating the function. It removes spaces, punctuation, and ignores case to ensure accurate sentence-level palindrome detection. All AI-generated test cases pass successfully, confirming correctness and robustness of the solution.

TASK4:

```
def run_tests():
    passed = 0
    total = 0

    # Test 1: Add items and calculate total
    total += 1
    cart = ShoppingCart()
    cart.add_item("Laptop", 50000)
    cart.add_item("Mouse", 1000)
    result = cart.total_cost()
    expected = 51000
    status = "PASS" if result == expected else "FAIL"
    print(f"Add Items Test      Expected: {expected} Got: {result} --> {status}")
    if status == "PASS": passed += 1

    # Test 2: Remove item
    total += 1
    cart.remove_item("Mouse")
    result = cart.total_cost()
    expected = 50000
    status = "PASS" if result == expected else "FAIL"
    print(f"Remove Item Test      Expected: {expected} Got: {result} --> {status}")
    if status == "PASS": passed += 1

    # Test 3: Remove non-existing item (should not crash)
    total += 1
    try:
        cart.remove_item("Keyboard")
        status = "PASS"
    except Exception:
        status = "FAIL"
    print(f"Remove Non-Exist Test --> {status}")
    if status == "PASS": passed += 1
```

JUSTIFICATION:

This implementation follows a test-driven approach by defining test cases to validate item addition, removal, and total cost calculation before finalizing the class logic. It ensures accurate cost computation, proper handling of empty cart scenarios, and safe removal of non-existing items without errors. All AI-generated test cases pass successfully, confirming correctness and reliability of the ShoppingCart class

TASK5:

```

import re
# =====
# Step 1: Define Test Cases (TDD)
# =====

def run_tests():
    test_cases = [
        # Valid dates
        ("2023-10-15", "15-10-2023"),
        ("2000-01-01", "01-01-2000"),
        ("1999-12-31", "31-12-1999"),
        ("2024-02-29", "29-02-2024"), # Leap year format valid

        # Invalid formats
        ("15-10-2023", "Error"),
        ("2023/10/15", "Error"),
        ("20231015", "Error"),
        ("abcd-ef-gh", "Error"),
        ("2023-13-01", "Error"), # Invalid month
        ("2023-00-10", "Error"), # Invalid month
        ("2023-10-32", "Error"), # Invalid day
    ]

    passed = 0

    for date_str, expected in test_cases:
        try:
            result = convert_date_format(date_str)
        except Exception:
            result = "Error"

        status = "PASS" if result == expected else "FAIL"

```

```

        print(f"{date_str:<15} Expected: {expected:<12} Got: {result:<12} --> {status}")

        if status == "PASS":
            passed += 1

    print("\nSummary:")
    print(f"{passed}/{len(test_cases)} test cases passed.")

    if passed == len(test_cases):
        print("All AI-generated test cases passed successfully ✅")

# =====
# Step 2: Implement Function
# =====

def convert_date_format(date_str):
    # Validate type
    if not isinstance(date_str, str):
        raise TypeError("Date must be a string")

    # Strict format check YYYY-MM-DD
    pattern = r"^\d{4}-\d{2}-\d{2}$"
    if not re.match(pattern, date_str):
        raise ValueError("Invalid date format")

    year, month, day = date_str.split("-")

    month = int(month)
    day = int(day)

    # Basic range validation

```

```
# Basic range validation
if not (1 <= month <= 12):
    raise ValueError("Invalid month")

if not (1 <= day <= 31):
    raise ValueError("Invalid day")

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

# =====
# Run Tests
# =====

if __name__ == "__main__":
    run_tests()
```

```
# Basic range validation
if not (1 <= month <= 12):
    raise ValueError("Invalid month")


if not (1 <= day <= 31):
    raise ValueError("Invalid day")

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

# =====
# Run Tests
# =====

if __name__ == "__main__":
    run_tests()
```

```
2023-10-15      Expected: 15-10-2023   Got: 15-10-2023   --> PASS
2000-01-01      Expected: 01-01-2000   Got: 01-01-2000   --> PASS
1999-12-31      Expected: 31-12-1999   Got: 31-12-1999   --> PASS
2024-02-29      Expected: 29-02-2024   Got: 29-02-2024   --> PASS
15-10-2023      Expected: Error        Got: Error        --> PASS
2023/10/15      Expected: Error        Got: Error        --> PASS
20231015        Expected: Error        Got: Error        --> PASS
abcd-ef-gh      Expected: Error        Got: Error        --> PASS
2023-13-01      Expected: Error        Got: Error        --> PASS
2023-00-10      Expected: Error        Got: Error        --> PASS
2023-10-32      Expected: Error        Got: Error        --> PASS

Summary:
11/11 test cases passed.
All AI-generated test cases passed successfully 
```

JUSTIFICATION:

This solution follows a test-driven approach by defining multiple valid and invalid date test cases before finalizing the conversion logic. It strictly validates the input format (YYYY-MM-DD) and ensures accurate transformation to DD-MM-YYYY while handling incorrect formats gracefully. All AI-generated test cases pass successfully, confirming correctness and reliability of the date conversion utility.

```
# Test 4: Empty cart total
total += 1
empty_cart = ShoppingCart()
result = empty_cart.total_cost()
expected = 0
status = "PASS" if result == expected else "FAIL"
print(f"Empty Cart Test      Expected: {expected} Got: {result} --> {status}")
if status == "PASS": passed += 1

# Test 5: Add multiple items
total += 1
cart2 = ShoppingCart()
cart2.add_item("Item1", 100)
cart2.add_item("Item2", 200)
cart2.add_item("Item3", 300)
result = cart2.total_cost()
expected = 600
status = "PASS" if result == expected else "FAIL"
print(f"Multiple Items Test  Expected: {expected} Got: {result} --> {status}")
if status == "PASS": passed += 1

print("\nSummary:")
print(f"{passed}/{total} test cases passed.")

if passed == total:
    print("All AI-generated test cases passed successfully ✅")
```

```

# =====
# Step 2: Implement ShoppingCart
# =====

class ShoppingCart:
    def __init__(self):
        self.items = {}

    def add_item(self, name, price):
        if price < 0:
            raise ValueError("Price cannot be negative")

        self.items[name] = self.items.get(name, 0) + price

    def remove_item(self, name):
        if name in self.items:
            del self.items[name]

    def total_cost(self):
        return sum(self.items.values())

# =====
# Run Tests
# =====

if __name__ == "__main__":
    run_tests()

```

```

pattern = r'^[A-Za-z0-9._]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}$'
Add Items Test      Expected: 51000 Got: 51000 --> PASS
Remove Item Test    Expected: 50000 Got: 50000 --> PASS
Remove Non-Exist Test --> PASS
Empty Cart Test     Expected: 0 Got: 0 --> PASS
Multiple Items Test Expected: 600 Got: 600 --> PASS

Summary:
5/5 test cases passed.
All AI-generated test cases passed successfully ✓

```

JUSTIFICATION:

This implementation follows a test-driven approach by defining test cases to validate item addition, removal, and total cost calculation before finalizing the class logic. It ensures accurate cost

computation, proper handling of empty cart scenarios, and safe removal of non-existing items without errors. All AI-generated test cases pass successfully, confirming correctness and reliability of the ShoppingCart class.