AIAC Assignment - 8.5

S.Suhas **2303A51453,B21**

Task Description #1 (Username Validator – Apply AI in Authentication Context)

- Task: Use AI to generate at least 3 assert test cases for a function is_valid_username(username) and then implement the function using Test-Driven Development principles.
- Requirements:
    - Username length must be between 5 and 15 characters.
    - Must contain only alphabets and digits.
    - Must not start with a digit.
    - No spaces allowed.

Example Assert Test Cases:

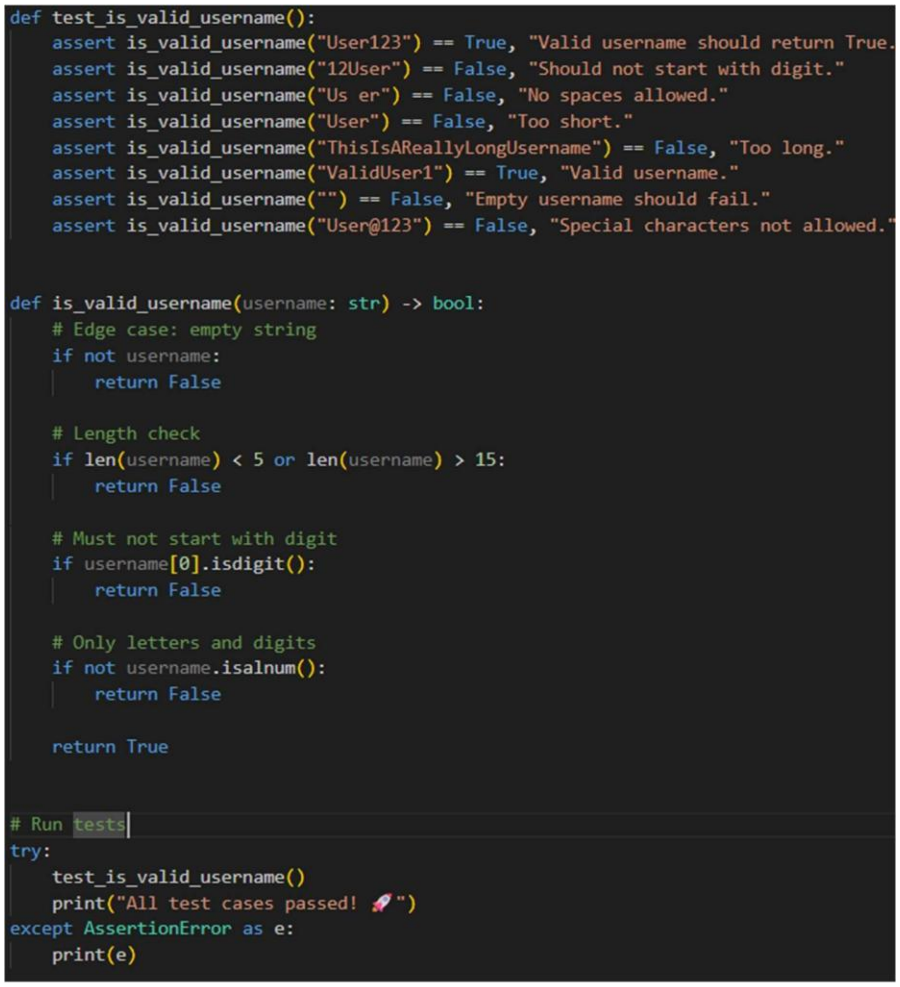
assert is_valid_username("User123") == True

assert is_valid_username("12User") == False

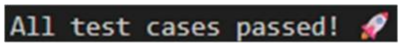
assert is_valid_username("Us er") == False

Expected Output #1:

- Username validation logic successfully passing all AI-generated test cases.


```
def test_is_valid_username():
    assert is_valid_username("User123") == True, "Valid username should return True.
    assert is_valid_username("12User") == False, "Should not start with digit."
    assert is_valid_username("Us er") == False, "No spaces allowed."
    assert is_valid_username("User") == False, "Too short."
    assert is_valid_username("ThisIsAReallyLongUsername") == False, "Too long."
    assert is_valid_username("ValidUser1") == True, "Valid username."
    assert is_valid_username("") == False, "Empty username should fail."
    assert is_valid_username("User@123") == False, "Special characters not allowed."


def is_valid_username(username: str) -> bool:
    # Edge case: empty string
    if not username:
        return False

    # Length check
    if len(username) < 5 or len(username) > 15:
        return False

    # Must not start with digit
    if username[0].isdigit():
        return False

    # Only letters and digits
    if not username.isalnum():
        return False

    return True


# Run tests
try:
    test_is_valid_username()
    print("All test cases passed! 🚀")
except AssertionError as e:
    print(e)
```

Output:


```
All test cases passed! 🚀
```

Task Description #2 (Even–Odd & Type Classification – Apply AI for Robust Input Handling)

- Task: Use AI to generate at least 3 assert test cases for a function classify_value(x) and implement it using conditional logic and loops.
- Requirements:
  - If input is an integer, classify as "Even" or "Odd".
  - If input is 0, return "Zero".
  - If input is non-numeric, return "Invalid Input".

Example Assert Test Cases:

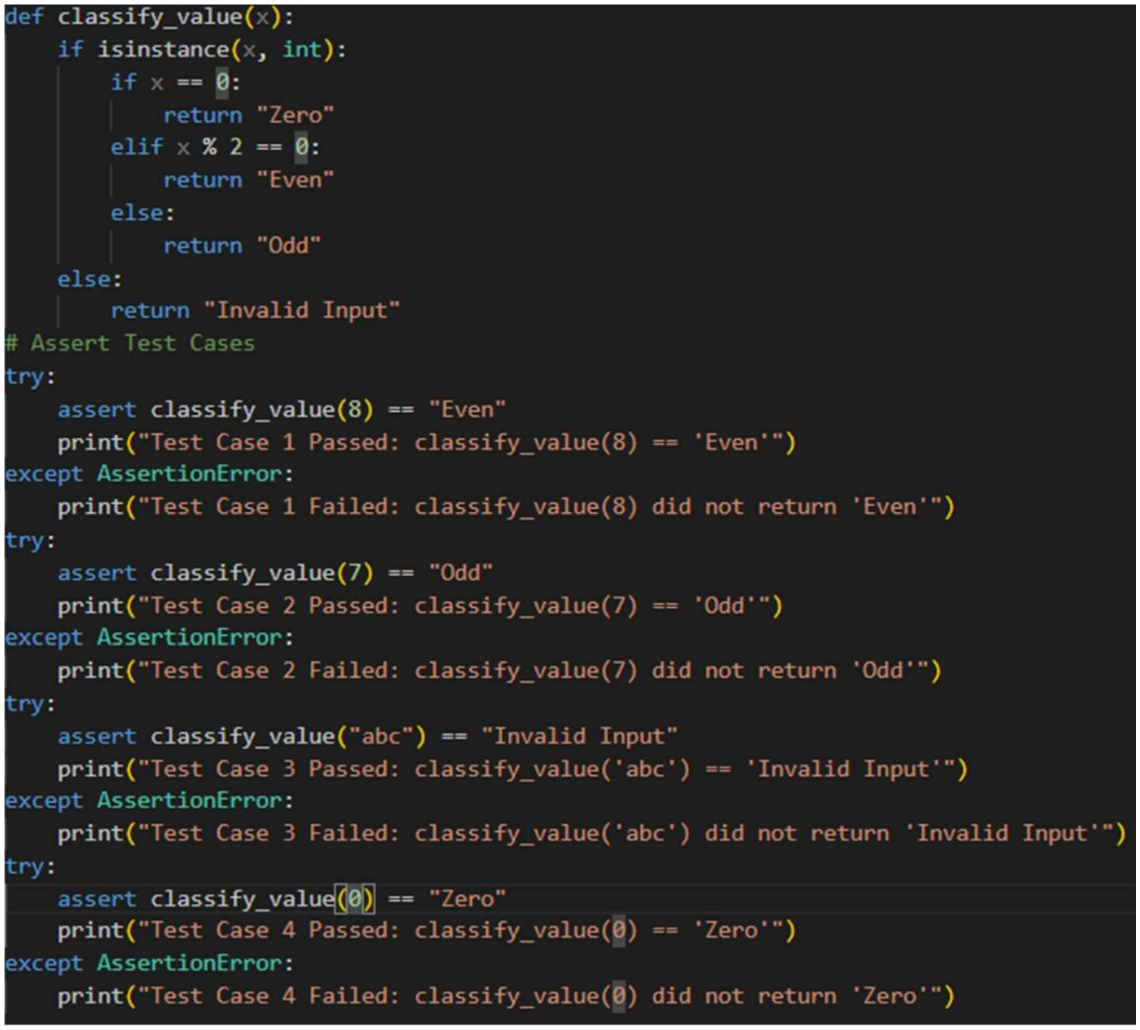
assert classify_value(8) == "Even"

assert classify_value(7) == "Odd"

assert classify_value("abc") == "Invalid Input"

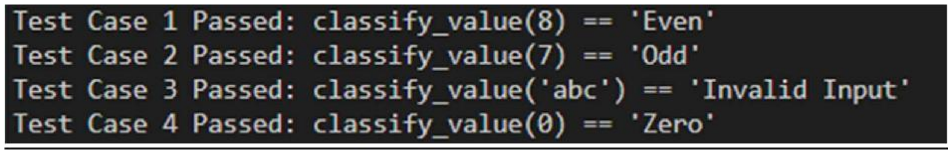
Expected Output #2:

- Function correctly classifying values and passing all test cases.



```python
def classify_value(x):
    if isinstance(x, int):
        if x == 0:
            return "Zero"
        elif x % 2 == 0:
            return "Even"
        else:
            return "Odd"
    else:
        return "Invalid Input"
# Assert Test Cases
try:
    assert classify_value(8) == "Even"
    print("Test Case 1 Passed: classify_value(8) == 'Even'")
except AssertionError:
    print("Test Case 1 Failed: classify_value(8) did not return 'Even'")
try:
    assert classify_value(7) == "Odd"
    print("Test Case 2 Passed: classify_value(7) == 'Odd'")
except AssertionError:
    print("Test Case 2 Failed: classify_value(7) did not return 'Odd'")
try:
    assert classify_value("abc") == "Invalid Input"
    print("Test Case 3 Passed: classify_value('abc') == 'Invalid Input'")
except AssertionError:
    print("Test Case 3 Failed: classify_value('abc') did not return 'Invalid Input'")
try:
    assert classify_value(0) == "Zero"
    print("Test Case 4 Passed: classify_value(0) == 'Zero'")
except AssertionError:
    print("Test Case 4 Failed: classify_value(0) did not return 'Zero'")
```

OUTPUT:



```
Test Case 1 Passed: classify_value(8) == 'Even'
Test Case 2 Passed: classify_value(7) == 'Odd'
Test Case 3 Passed: classify_value('abc') == 'Invalid Input'
Test Case 4 Passed: classify_value(0) == 'Zero'
```

Task Description #3 (Palindrome Checker – Apply AI for String Normalization)

- Task: Use AI to generate at least 3 assert test cases for a function is_palindrome(text) and implement the function.
- Requirements:
  - Ignore case, spaces, and punctuation.
  - Handle edge cases such as empty strings and single characters.

Example Assert Test Cases:

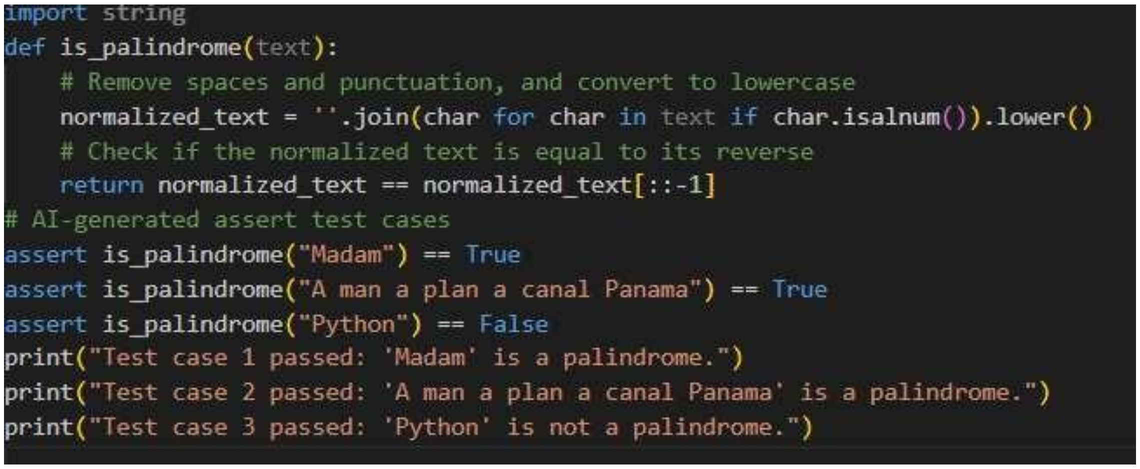
assert is_palindrome("Madam") == True

assert is_palindrome("A man a plan a canal Panama") == True

assert is_palindrome("Python") == False

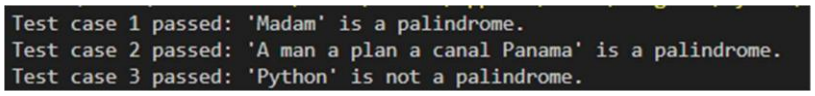
Expected Output #3:

- Function correctly identifying palindromes and passing all AI-generated tests.



```python
import string
def is_palindrome(text):
    # Remove spaces and punctuation, and convert to lowercase
    normalized_text = ''.join(char for char in text if char.isalnum()).lower()
    # Check if the normalized text is equal to its reverse
    return normalized_text == normalized_text[::-1]
# AI-generated assert test cases
assert is_palindrome("Madam") == True
assert is_palindrome("A man a plan a canal Panama") == True
assert is_palindrome("Python") == False
print("Test case 1 passed: 'Madam' is a palindrome.")
print("Test case 2 passed: 'A man a plan a canal Panama' is a palindrome.")
print("Test case 3 passed: 'Python' is not a palindrome.")
```

OUTPUT:



```
Test case 1 passed: 'Madam' is a palindrome.
Test case 2 passed: 'A man a plan a canal Panama' is a palindrome.
Test case 3 passed: 'Python' is not a palindrome.
```

Task Description #4 (BankAccount Class – Apply AI for Object-Oriented Test-Driven Development)

- Task: Ask AI to generate at least 3 assert-based test cases for a BankAccount class and then implement the class.
- Methods:
    - deposit(amount)
    - withdraw(amount)
    - get_balance()

Example Assert Test Cases:

acc = BankAccount(1000)

acc.deposit(500)

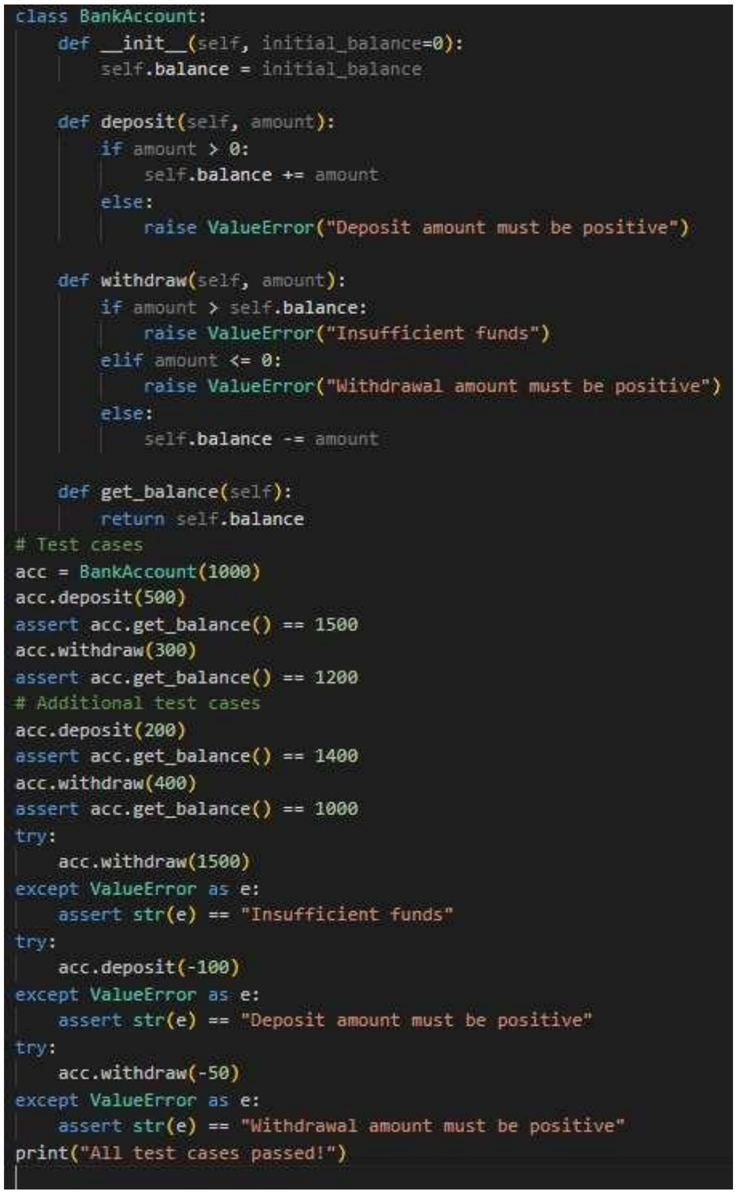
assert acc.get_balance() == 1500

acc.withdraw(300)

assert acc.get_balance() == 1200

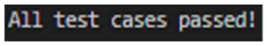
Expected Output #4:

Fully functional class that passes all AI-generated assertions



```python
class BankAccount:
    def __init__(self, initial_balance=0):
        self.balance = initial_balance

    def deposit(self, amount):
        if amount > 0:
            self.balance += amount
        else:
            raise ValueError("Deposit amount must be positive")

    def withdraw(self, amount):
        if amount > self.balance:
            raise ValueError("Insufficient funds")
        elif amount <= 0:
            raise ValueError("Withdrawal amount must be positive")
        else:
            self.balance -= amount

    def get_balance(self):
        return self.balance
# Test cases
acc = BankAccount(1000)
acc.deposit(500)
assert acc.get_balance() == 1500
acc.withdraw(300)
assert acc.get_balance() == 1200
# Additional test cases
acc.deposit(200)
assert acc.get_balance() == 1400
acc.withdraw(400)
assert acc.get_balance() == 1000
try:
    acc.withdraw(1500)
except ValueError as e:
    assert str(e) == "Insufficient funds"
try:
    acc.deposit(-100)
except ValueError as e:
    assert str(e) == "Deposit amount must be positive"
try:
    acc.withdraw(-50)
except ValueError as e:
    assert str(e) == "Withdrawal amount must be positive"
print("All test cases passed!")
```

OUTPUT:



```
All test cases passed!
```

Task Description #5 (Email ID Validation – Apply AI for Data Validation)

- Task: Use AI to generate at least 3 assert test cases for a function validate_email(email) and implement the function.
- Requirements:
  - Must contain @ and .
  - Must not start or end with special characters.
  - Should handle invalid formats gracefully.

Example Assert Test Cases:

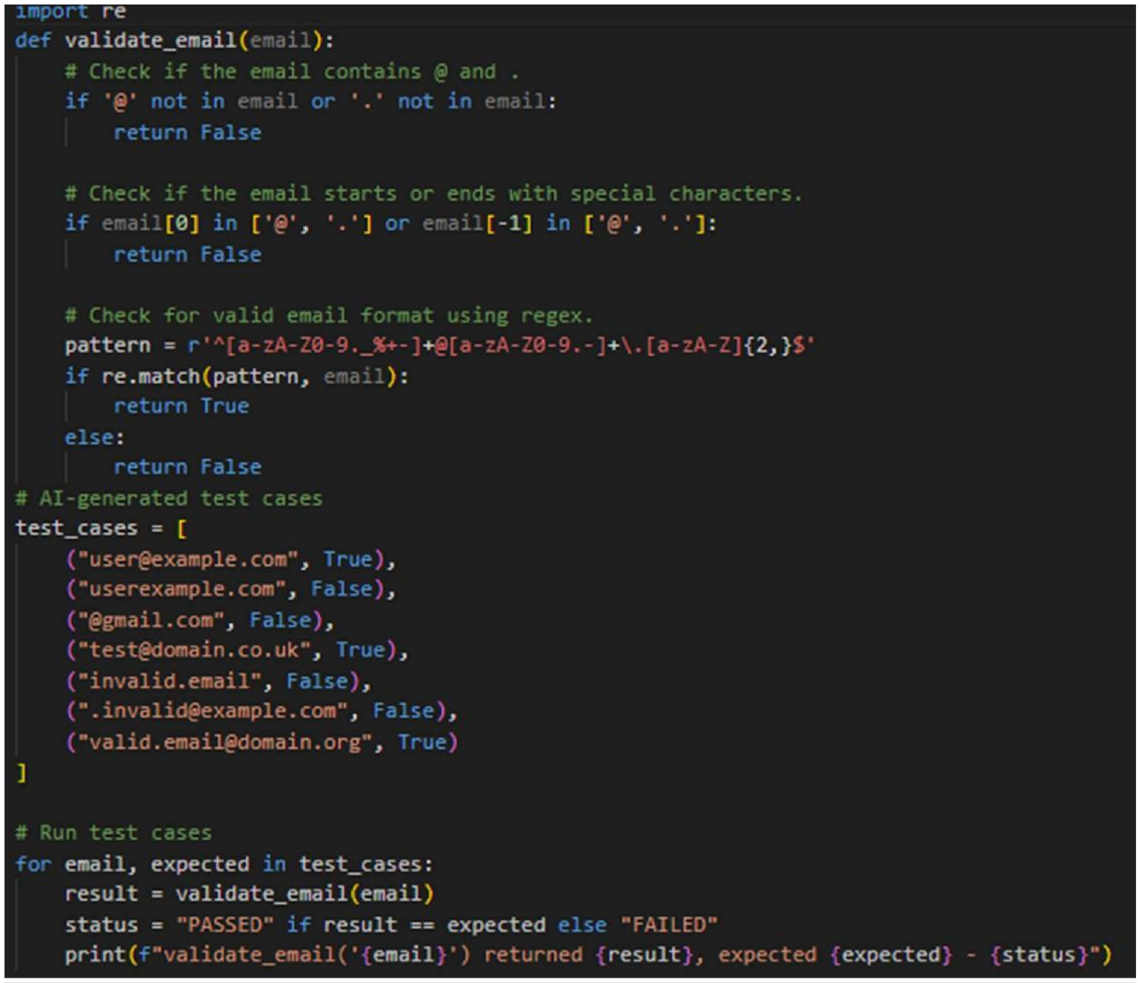
assert validate_email("user@example.com") == True

assert validate_email("userexample.com") == False

assert validate_email("@gmail.com") == False

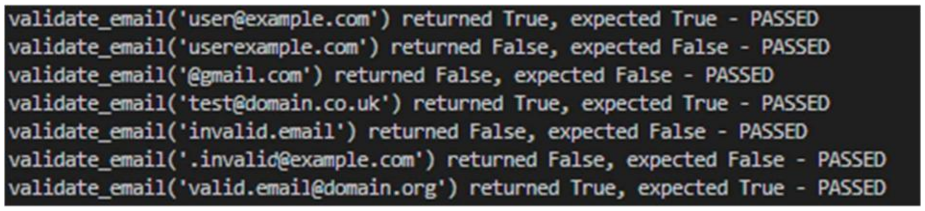
Expected Output #5:

- Email validation function passing all AI-generated test cases and handling edge cases correctly.



```python
import re
def validate_email(email):
    # Check if the email contains @ and .
    if '@' not in email or '.' not in email:
        return False

    # Check if the email starts or ends with special characters.
    if email[0] in ['@', '.'] or email[-1] in ['@', '.']:
        return False

    # Check for valid email format using regex.
    pattern = r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'
    if re.match(pattern, email):
        return True
    else:
        return False
# AI-generated test cases
test_cases = [
    ("user@example.com", True),
    ("userexample.com", False),
    ("@gmail.com", False),
    ("test@domain.co.uk", True),
    ("invalid.email", False),
    (".invalid@example.com", False),
    ("valid.email@domain.org", True)
]

# Run test cases
for email, expected in test_cases:
    result = validate_email(email)
    status = "PASSED" if result == expected else "FAILED"
    print(f"validate_email('{email}') returned {result}, expected {expected} - {status}")
```

OUTPUT:



```
validate_email('user@example.com') returned True, expected True - PASSED
validate_email('userexample.com') returned False, expected False - PASSED
validate_email('@gmail.com') returned False, expected False - PASSED
validate_email('test@domain.co.uk') returned True, expected True - PASSED
validate_email('invalid.email') returned False, expected False - PASSED
validate_email('.invalid@example.com') returned False, expected False - PASSED
validate_email('valid.email@domain.org') returned True, expected True - PASSED
```