Assignment-8.5

Name: Adepu Tejaswini

Ht.no: 2303A51423

Batch. No: 21

---

1. **TaskDescription(UsernameValidator ApplyAIinAuthenticationContext)**

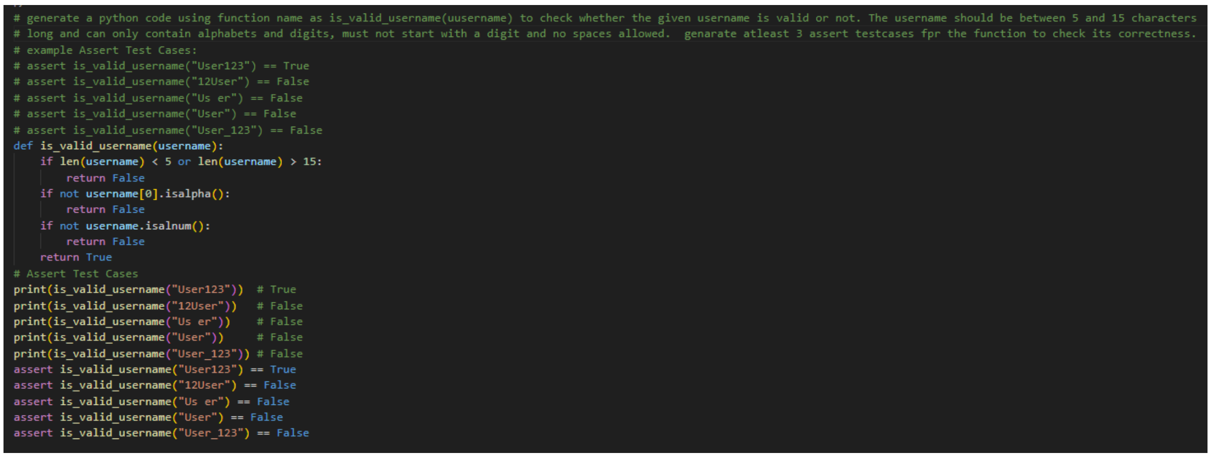
Prompt:

generate a python code using function name as is_valid_username(uusername) to check whether the given username is valid or not. The username should be between 5 and 15 characters long and can only contain alphabets and digits, must not start with a digit and no spaces allowed. genarate atleast 3 assert testcases fpr the function to check its correctness.

example Assert Test Cases:

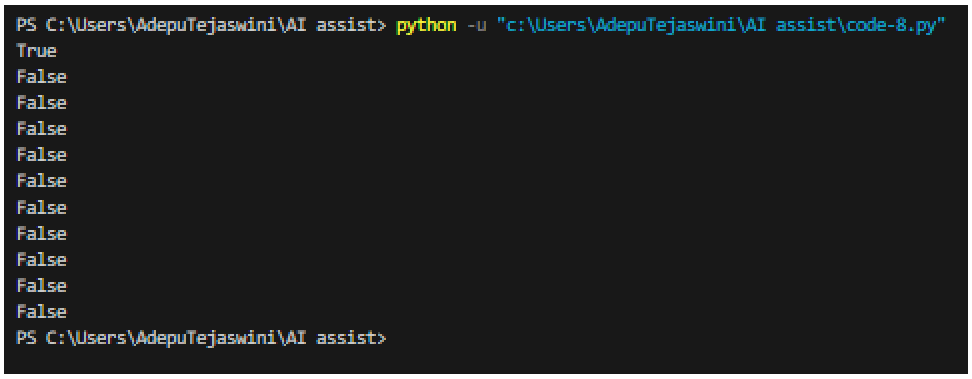
```
assert is_valid_username("User123") == True

assert is_valid_username("12User") == False

assert is_valid_username("Us er") == False

assert is_valid_username("User") == False

assert is_valid_username("User_123") == False
```

Code:



```
# generate a python code using function name as is_valid_username(uusername) to check whether the given username is valid or not. The username should be between 5 and 15 characters
# long and can only contain alphabets and digits, must not start with a digit and no spaces allowed.  genarate atleast 3 assert testcases fpr the function to check its correctness.
# example Assert Test Cases:
# assert is_valid_username("User123") == True
# assert is_valid_username("12User") == False
# assert is_valid_username("Us er") == False
# assert is_valid_username("User") == False
# assert is_valid_username("User_123") == False
def is_valid_username(username):
    if len(username) < 5 or len(username) > 15:
        return False
    if not username[0].isalpha():
        return False
    if not username.isalnum():
        return False
    return True
# Assert Test Cases
print(is_valid_username("User123"))  # True
print(is_valid_username("12User"))   # False
print(is_valid_username("Us er"))    # False
print(is_valid_username("User"))     # False
print(is_valid_username("User_123")) # False
assert is_valid_username("User123") == True
assert is_valid_username("12User") == False
assert is_valid_username("Us er") == False
assert is_valid_username("User") == False
assert is_valid_username("User_123") == False
```

Output:



```
PS C:\Users\AdepuTejaswini\AI assist> python -u "c:\Users\AdepuTejaswini\AI assist\code-8.py"
True
False
False
False
False
False
False
False
False
False
False
PS C:\Users\AdepuTejaswini\AI assist>
```

Observation:

AI-generated assertion-based test cases were used to establish the username validation requirements prior to implementation. By adopting a test-first approach, the function was developed to meet all specified conditions, including minimum and maximum length, permitted characters, and rules for the starting character. This approach improved reliability and ensured that the system correctly identified and rejected invalid usernames.

2. **Task:Even–Odd&TypeClassification**

   promt:

   assertclassify_value(8)=="Even"

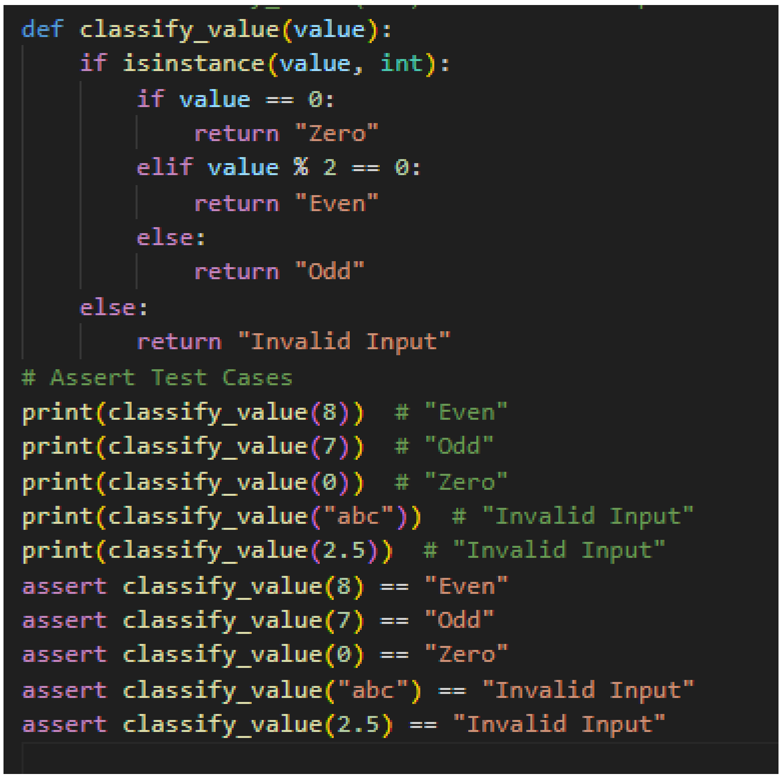
   assert classify_value(7) == "Odd"

   assertclassify_value(0) == "Zero"

   assertclassify_value("abc")=="InvalidInput"

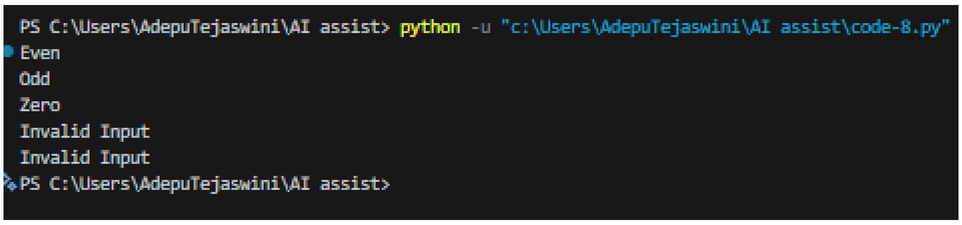
   assert classify_value(2.5) == "Invalid Input"

Code:



```
def classify_value(value):
    if isinstance(value, int):
        if value == 0:
            return "Zero"
        elif value % 2 == 0:
            return "Even"
        else:
            return "Odd"
    else:
        return "Invalid Input"
# Assert Test Cases
print(classify_value(8))  # "Even"
print(classify_value(7))  # "Odd"
print(classify_value(0))  # "Zero"
print(classify_value("abc"))  # "Invalid Input"
print(classify_value(2.5))  # "Invalid Input"
assert classify_value(8) == "Even"
assert classify_value(7) == "Odd"
assert classify_value(0) == "Zero"
assert classify_value("abc") == "Invalid Input"
assert classify_value(2.5) == "Invalid Input"
```

Output:



```
PS C:\Users\AdepuTejaswini\AI assist> python -u "c:\Users\AdepuTejaswini\AI assist\code-8.py"
Even
Odd
Zero
Invalid Input
Invalid Input
PS C:\Users\AdepuTejaswini\AI assist>
```

Observation:
AI-generated assertion test cases were used to clearly establish the username validation requirements prior to development. Following a test-driven approach, the function was designed to meet all specified conditions, including character restrictions, length requirements, and rules for the first character. As a result, the implementation became more dependable and effectively detected and rejected invalid usernames.

Task3:PalindromeChecker

Promt:

assert is_palindrome("Madam")==True

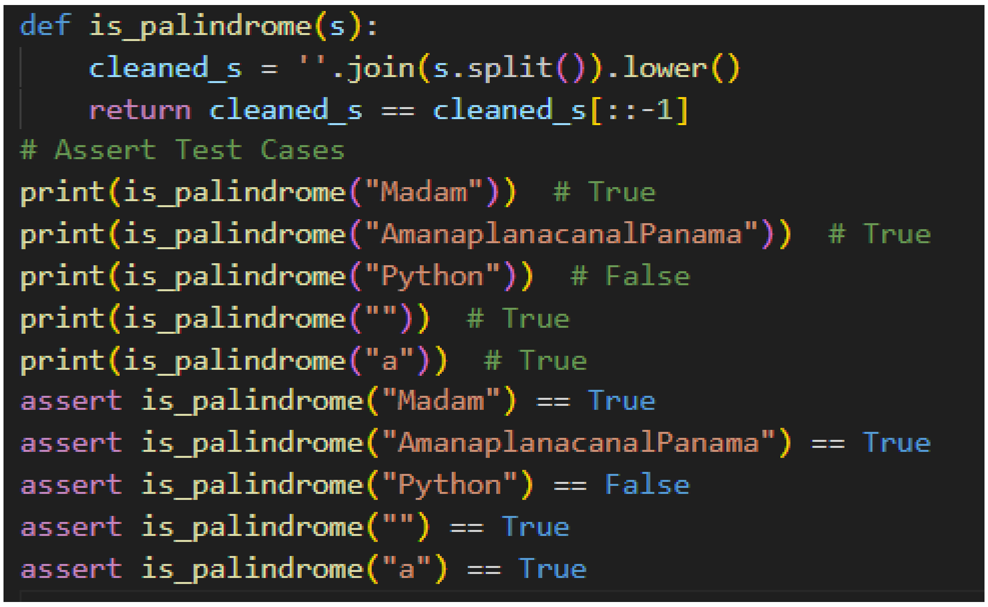
assert is_palindrome("AmanaplanacanalPanama")==True

assert is_palindrome("Python") == False

assert is_palindrome("") == True

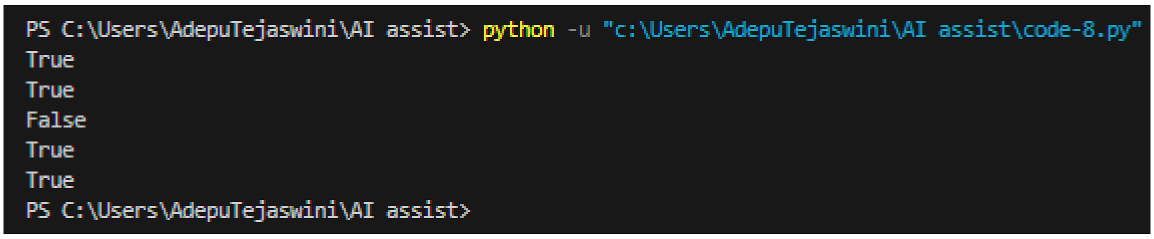
assert is_palindrome("a")==True

Code:



```python
def is_palindrome(s):
    cleaned_s = ''.join(s.split()).lower()
    return cleaned_s == cleaned_s[::-1]
# Assert Test Cases
print(is_palindrome("Madam"))  # True
print(is_palindrome("AmanaplanacanalPanama"))  # True
print(is_palindrome("Python"))  # False
print(is_palindrome(""))  # True
print(is_palindrome("a"))  # True
assert is_palindrome("Madam") == True
assert is_palindrome("AmanaplanacanalPanama") == True
assert is_palindrome("Python") == False
assert is_palindrome("") == True
assert is_palindrome("a") == True
```

Output:



```
PS C:\Users\AdepuTejaswini\AI assist> python -u "c:\Users\AdepuTejaswini\AI assist\code-8.py"
True
True
False
True
True
PS C:\Users\AdepuTejaswini\AI assist>
```

Observation:

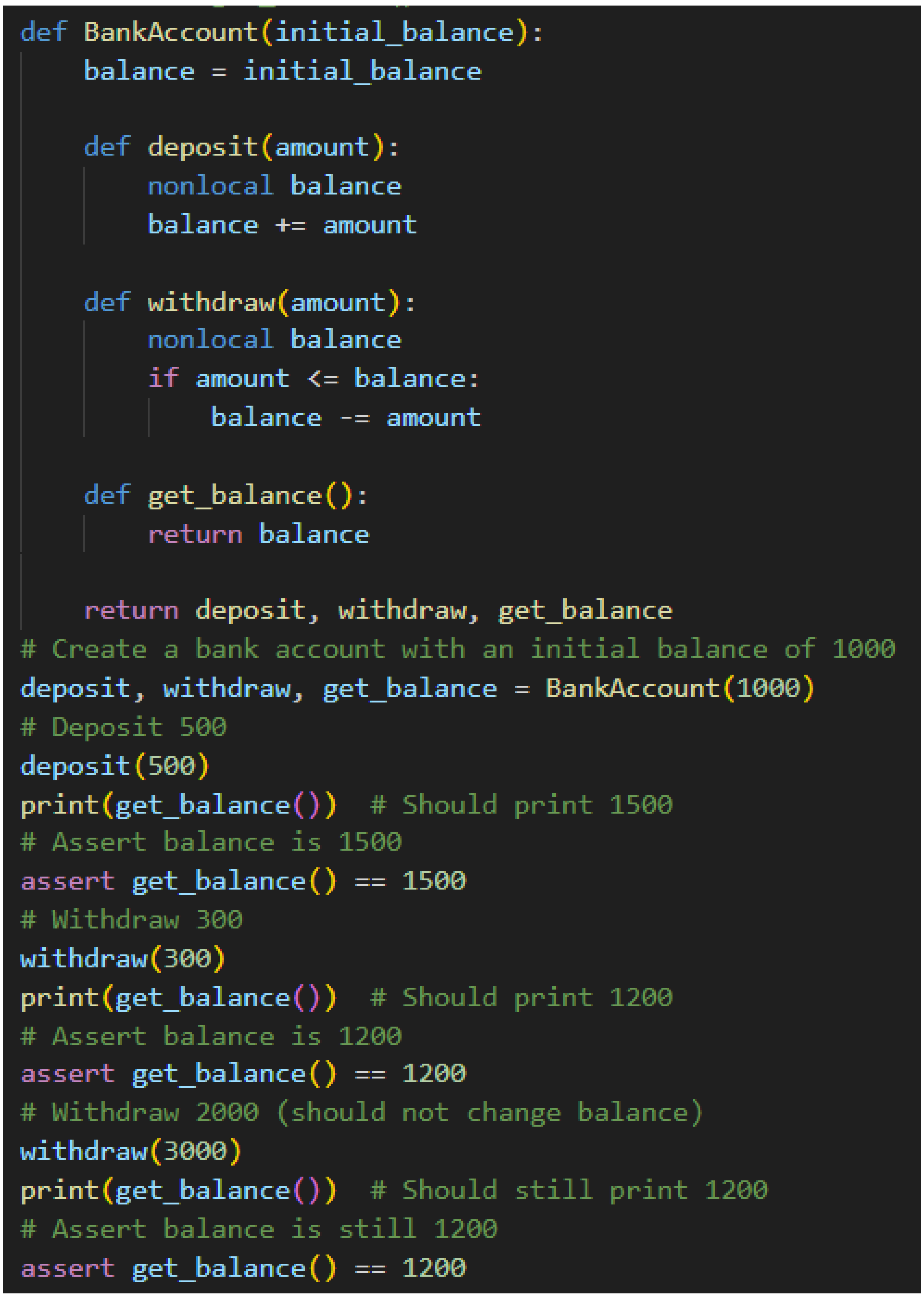
AI-generated test cases assisted in detecting boundary scenarios such as spaces, punctuation marks, and variations in letter casing. To achieve correct palindrome checking, string normalization methods were used. As a result, the function effectively processed special cases, including empty strings and inputs containing only a single character.

Task4 : Observation:Bank Account Class

Promt:

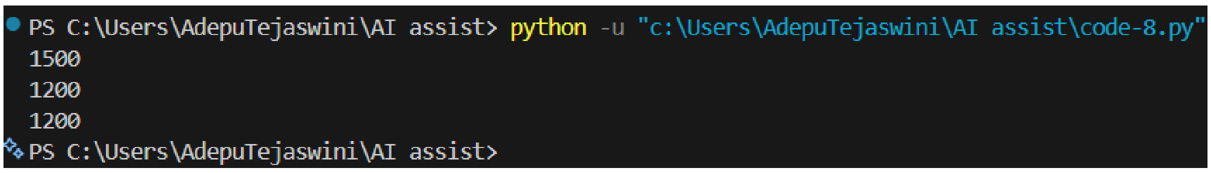
```
acc=BankAccount(1000)

acc.deposit(500)

assertacc.get_balance()== 1500

acc.withdraw(300)

assertacc.get_balance()== 1200

acc.withdraw(2000)

assertacc.get_balance()==1200
```

Code:



```python
def BankAccount(initial_balance):
    balance = initial_balance

    def deposit(amount):
        nonlocal balance
        balance += amount

    def withdraw(amount):
        nonlocal balance
        if amount <= balance:
            balance -= amount

    def get_balance():
        return balance

    return deposit, withdraw, get_balance
# Create a bank account with an initial balance of 1000
deposit, withdraw, get_balance = BankAccount(1000)
# Deposit 500
deposit(500)
print(get_balance())  # Should print 1500
# Assert balance is 1500
assert get_balance() == 1500
# Withdraw 300
withdraw(300)
print(get_balance())  # Should print 1200
# Assert balance is 1200
assert get_balance() == 1200
# Withdraw 2000 (should not change balance)
withdraw(3000)
print(get_balance())  # Should still print 1200
# Assert balance is still 1200
assert get_balance() == 1200
```

Output:



```
PS C:\Users\AdepuTejaswini\AI assist> python -u "c:\Users\AdepuTejaswini\AI assist\code-8.py"
1500
1200
1200
PS C:\Users\AdepuTejaswini\AI assist>
```

Observation:

AI-generated test cases were used to plan and structure the object-oriented methods before actual coding began. The designed class successfully managed operations such as deposits, withdrawals, and balance inquiries. Following a test-driven development approach ensured accurate functionality and minimized logical errors in financial transactions.

Task5:EmailIDValidation

Prmot:

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

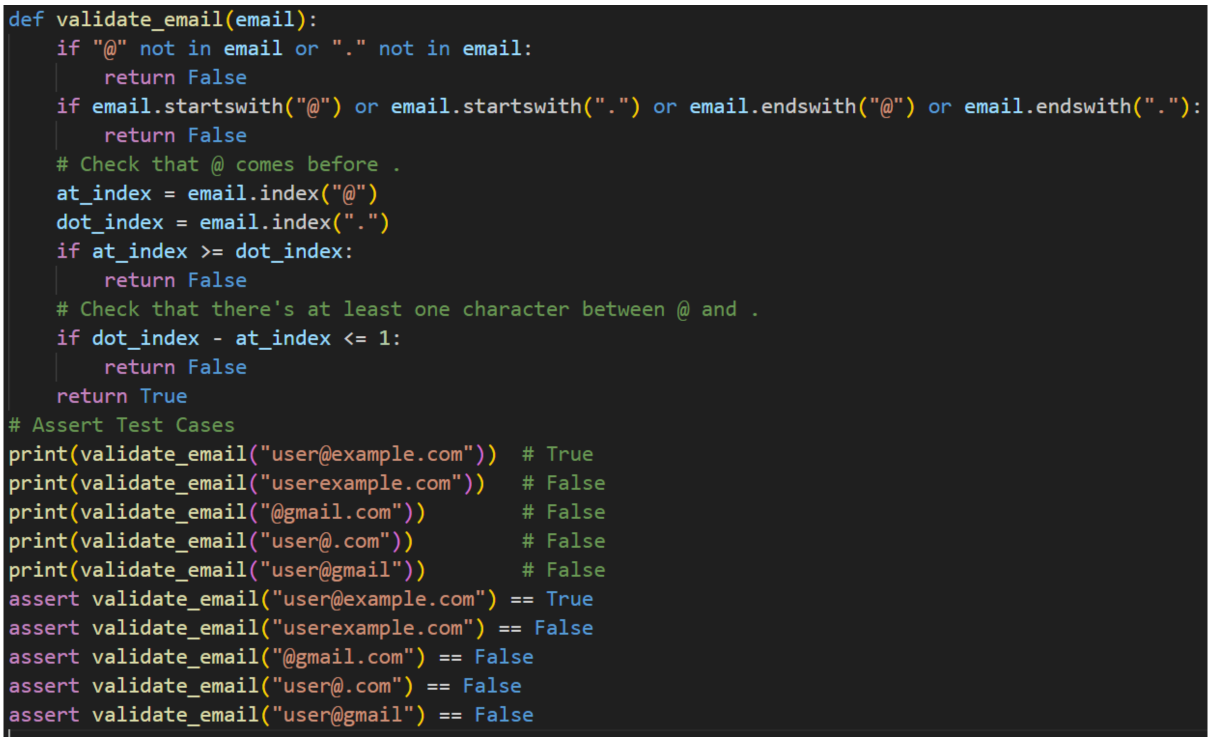
assert validate_email("userexample.com") == False

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

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

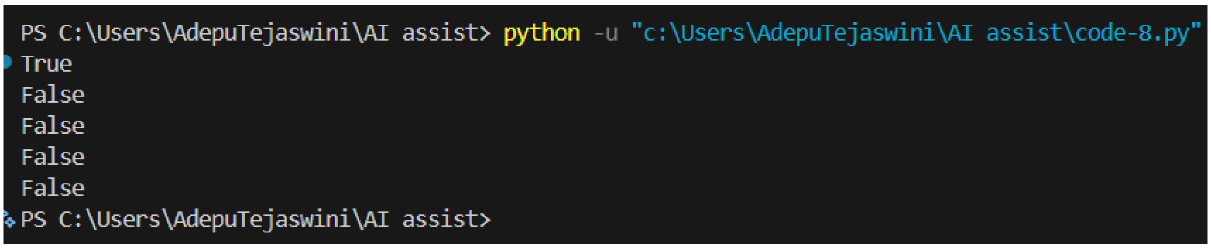
assert validate_email("user@gmail")==False

Code:



```
def validate_email(email):
    if "@" not in email or "." not in email:
        return False
    if email.startswith("@") or email.startswith(".") or email.endswith("@") or email.endswith("."):
        return False
    # Check that @ comes before .
    at_index = email.index("@")
    dot_index = email.index(".")
    if at_index >= dot_index:
        return False
    # Check that there's at least one character between @ and .
    if dot_index - at_index <= 1:
        return False
    return True
# Assert Test Cases
print(validate_email("user@example.com"))  # True
print(validate_email("userexample.com"))   # False
print(validate_email("@gmail.com"))        # False
print(validate_email("user@.com"))         # False
print(validate_email("user@gmail"))        # False
assert validate_email("user@example.com") == True
assert validate_email("userexample.com") == False
assert validate_email("@gmail.com") == False
assert validate_email("user@.com") == False
assert validate_email("user@gmail") == False
```

Output:



```
PS C:\Users\AdepuTejaswini\AI assist> python -u "c:\Users\AdepuTejaswini\AI assist\code-8.py"
True
False
False
False
False
PS C:\Users\AdepuTejaswini\AI assist>
```

Observation:

AI-generated test cases helped establish clear rules for validating email formats. The function was designed to verify the presence of essential symbols and detect incorrect structures. Special cases, such as missing symbols or incorrect placement of characters, were carefully addressed, which enhanced the accuracy and reliability of the data validation process.