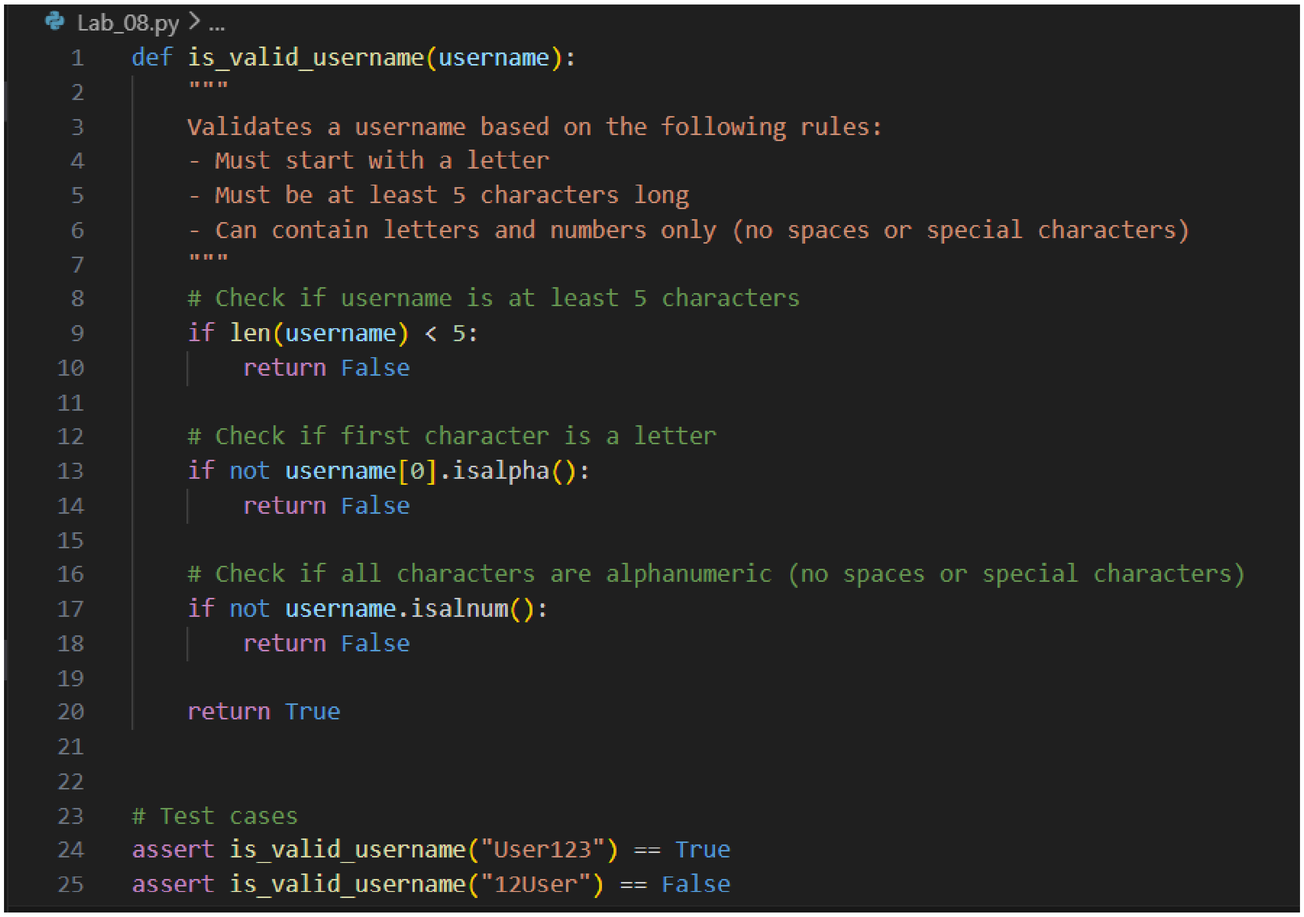
Assignment–8.5

Name : D.Akash
Batch : 21
Ht.No : 2303A51422

1. TaskDescription(UsernameValidator–ApplyAIinAuthenticationContext)
Promt:
#TestCases
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
assertis_valid_username("User_123")==False
valid usernames correctly.

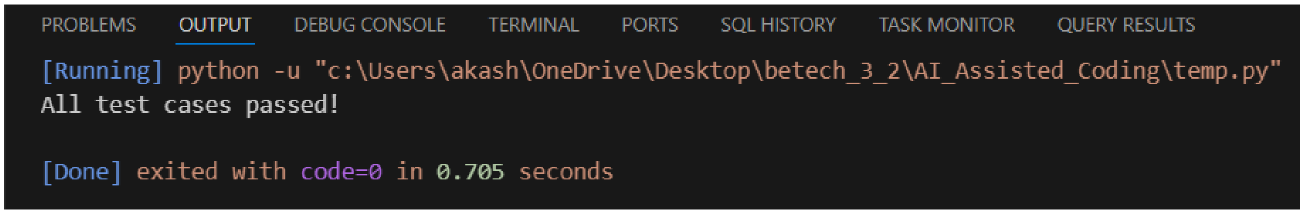
Code:



Lab_08.py > ...

```python
def is_valid_username(username):
    """
    Validates a username based on the following rules:
    - Must start with a letter
    - Must be at least 5 characters long
    - Can contain letters and numbers only (no spaces or special characters)
    """
    # Check if username is at least 5 characters
    if len(username) < 5:
        return False

    # Check if first character is a letter
    if not username[0].isalpha():
        return False

    # Check if all characters are alphanumeric (no spaces or special characters)
    if not username.isalnum():
        return False

    return True


# Test cases
assert is_valid_username("User123") == True
assert is_valid_username("12User") == False
```

Output:

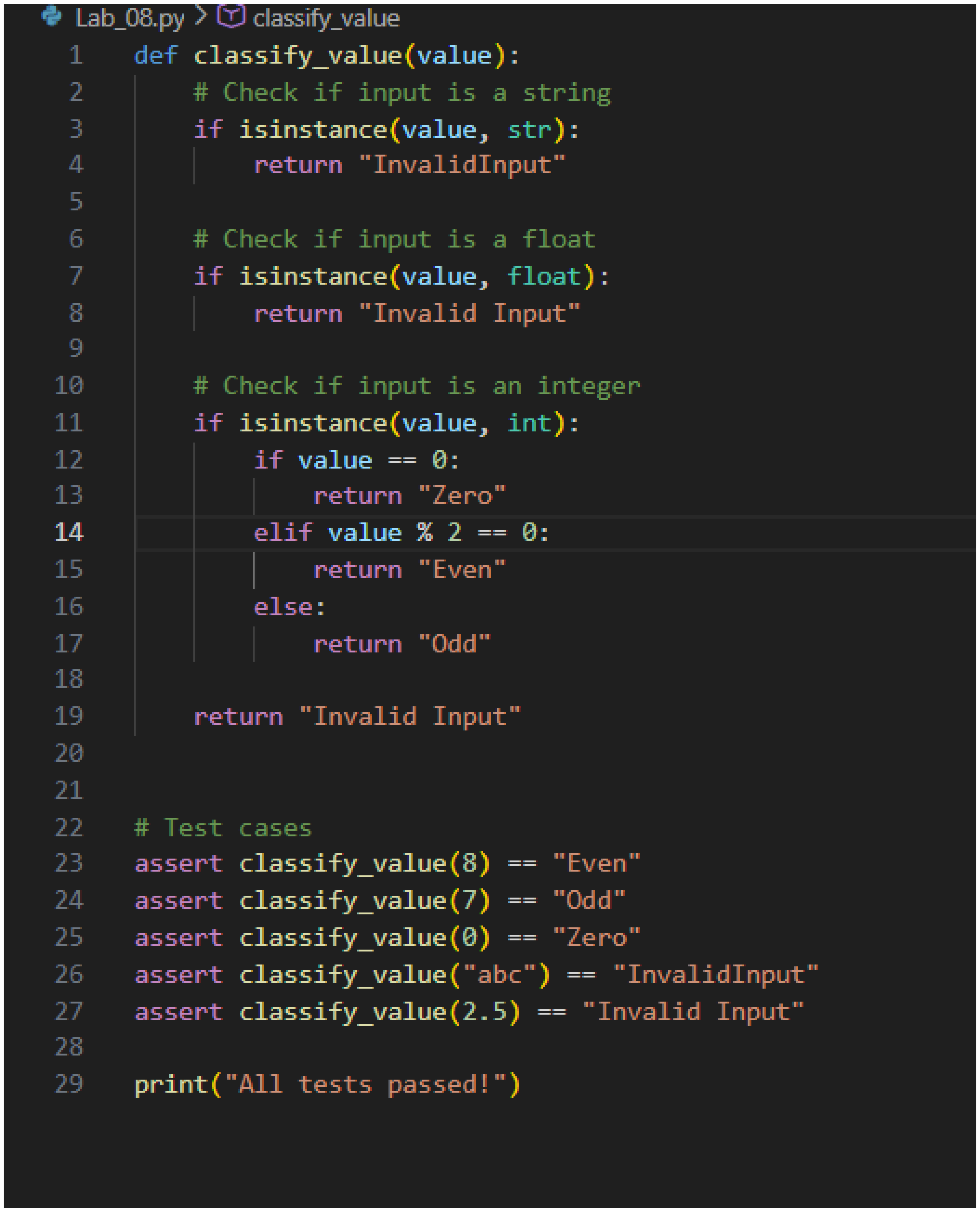


```
[Running] python -u "c:\Users\akash\OneDrive\Desktop\betech_3_2\AI_Assisted_Coding\temp.py"
All test cases passed!

[Done] exited with code=0 in 0.705 seconds
```

Observation:

AI-generated assert test cases helped define the user name validation rules before coding. By writing tests first, the function was implemented to satisfy all constraints such as length limits, allowed characters, and starting character rules . This ensured the function was reliable and handled invalid usernames correctly.

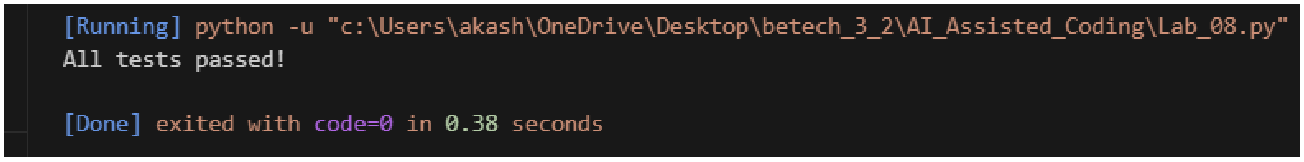
2. **Task:Even–Odd&TypeClassification**
promt:
assertclassify_value(8)=="Even"
assert classify_value(7) == "Odd"
assertclassify_value(0) == "Zero"
assertclassify_value("abc")=="InvalidInput"
assert classify_value(2.5) == "Invalid Input"
code:



Lab_08.py > classify_value

```python
def classify_value(value):
    # Check if input is a string
    if isinstance(value, str):
        return "InvalidInput"

    # Check if input is a float
    if isinstance(value, float):
        return "Invalid Input"

    # Check if input is an integer
    if isinstance(value, int):
        if value == 0:
            return "Zero"
        elif value % 2 == 0:
            return "Even"
        else:
            return "Odd"

    return "Invalid Input"


# Test cases
assert classify_value(8) == "Even"
assert classify_value(7) == "Odd"
assert classify_value(0) == "Zero"
assert classify_value("abc") == "InvalidInput"
assert classify_value(2.5) == "Invalid Input"

print("All tests passed!")
```

Output:



```
[Running] python -u "c:\Users\akash\OneDrive\Desktop\betech_3_2\AI_Assisted_Coding\Lab_08.py"
All tests passed!

[Done] exited with code=0 in 0.38 seconds
```

Observation:

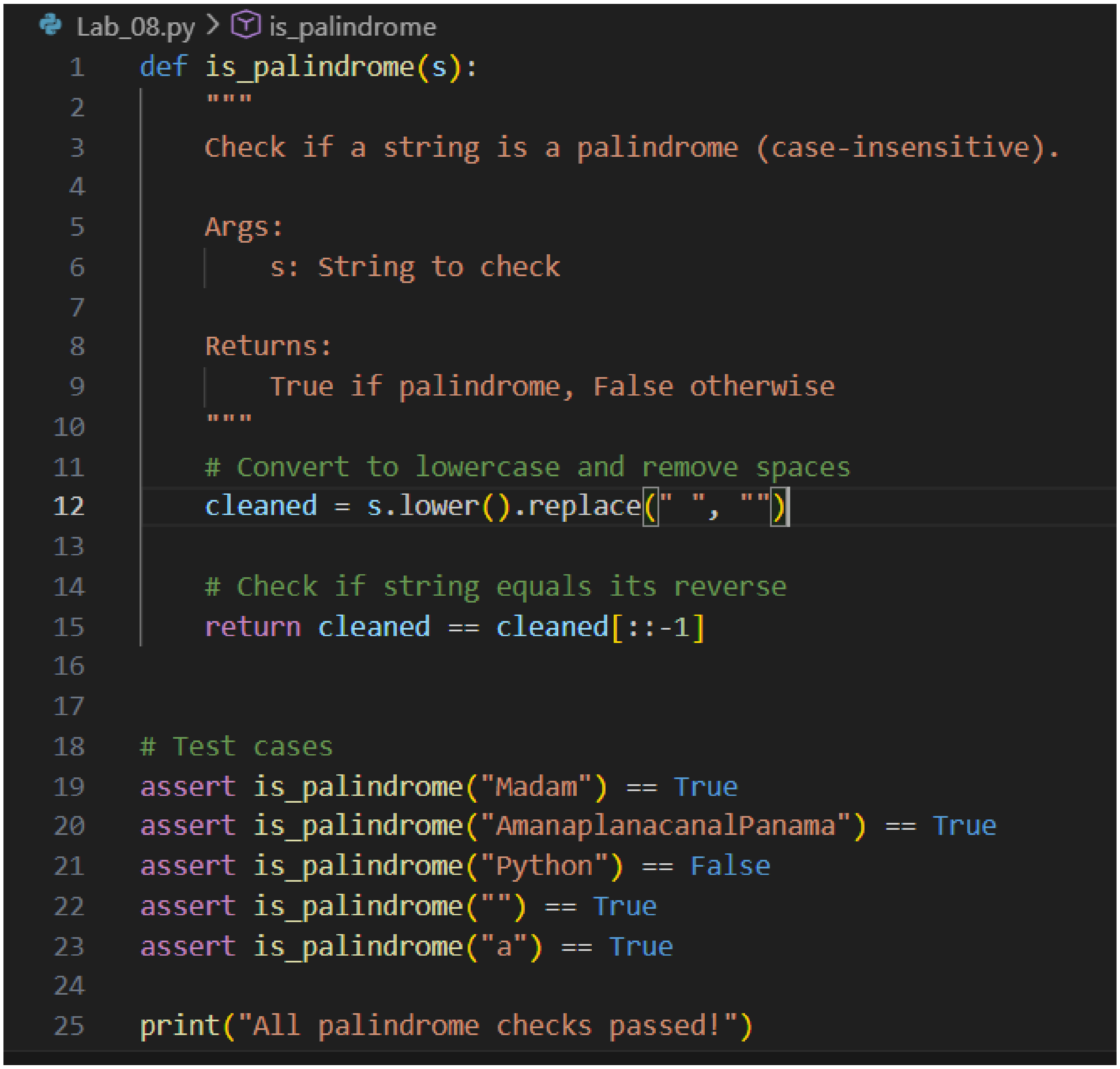
AI-generated assert test cases helped define the username validation rules before coding. By writing tests first, the function was implemented to satisfy all constraints such as length limits, allowed characters, and starting character rules . This ensured the function was reliable and handled invalid usernames correctly.

Task3:PalindromeChecker

Promt:

assertis_palindrome("Madam")==True
assertis_palindrome("AmanaplanacanalPanama")==True assert
is_palindrome("Python") == False
assert is_palindrome("") == True
assertis_palindrome("a")==True

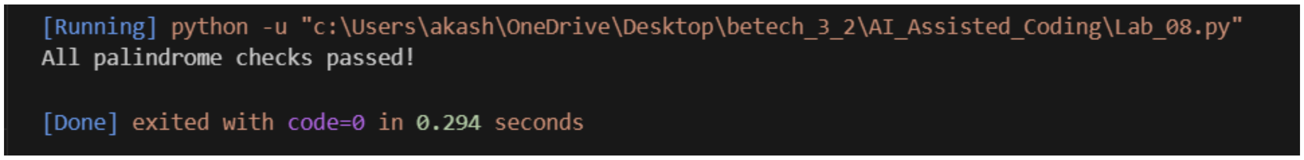
Code:



Lab_08.py > is_palindrome

```python
def is_palindrome(s):
    """
    Check if a string is a palindrome (case-insensitive).

    Args:
        s: String to check

    Returns:
        True if palindrome, False otherwise
    """
    # Convert to lowercase and remove spaces
    cleaned = s.lower().replace(" ", "")

    # Check if string equals its reverse
    return cleaned == cleaned[::-1]


# Test cases
assert is_palindrome("Madam") == True
assert is_palindrome("AmanaplanacanalPanama") == True
assert is_palindrome("Python") == False
assert is_palindrome("") == True
assert is_palindrome("a") == True

print("All palindrome checks passed!")
```

Output:



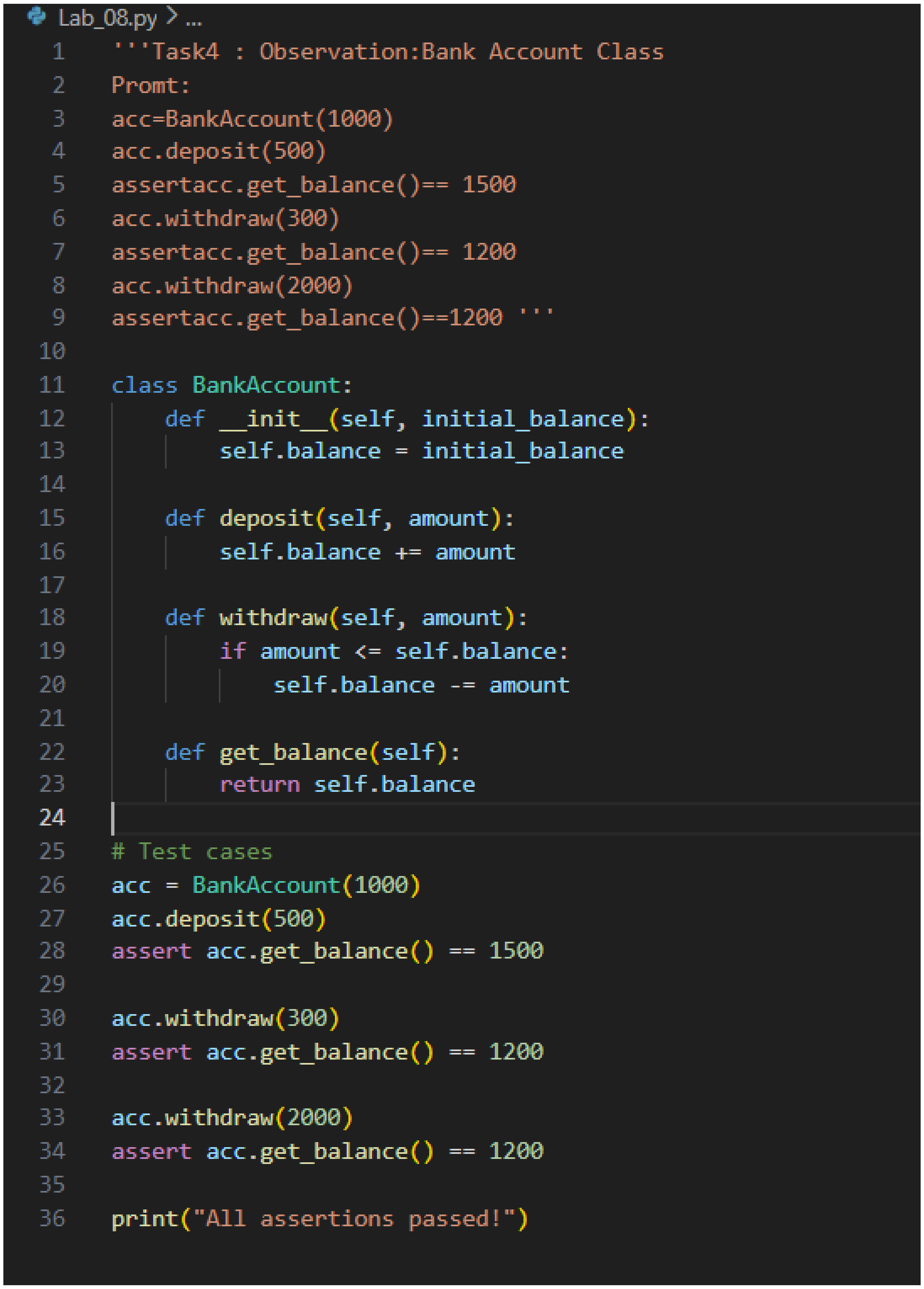
```
[Running] python -u "c:\Users\akash\OneDrive\Desktop\betech_3_2\AI_Assisted_Coding\Lab_08.py"
All palindrome checks passed!

[Done] exited with code=0 in 0.294 seconds
```

Observation:
AI-generated tests helped identify edge cases likes paces, punctuation, and case differences. String normalization techniques were applied to ensure accurate palindrome detection. The function successfully handled empty strings and single-character inputs.

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
```

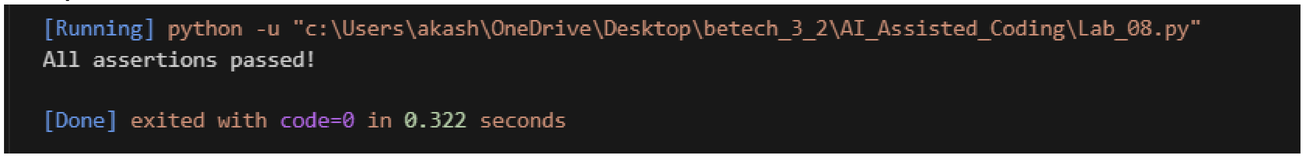
Code:



Lab_08.py > ...

```python
'''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 '''

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

    def deposit(self, amount):
        self.balance += amount

    def withdraw(self, amount):
        if amount <= self.balance:
            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

acc.withdraw(2000)
assert acc.get_balance() == 1200

print("All assertions passed!")
```

Output:



```
[Running] python -u "c:\Users\akash\OneDrive\Desktop\betech_3_2\AI_Assisted_Coding\Lab_08.py"
All assertions passed!

[Done] exited with code=0 in 0.322 seconds
```

Observation:

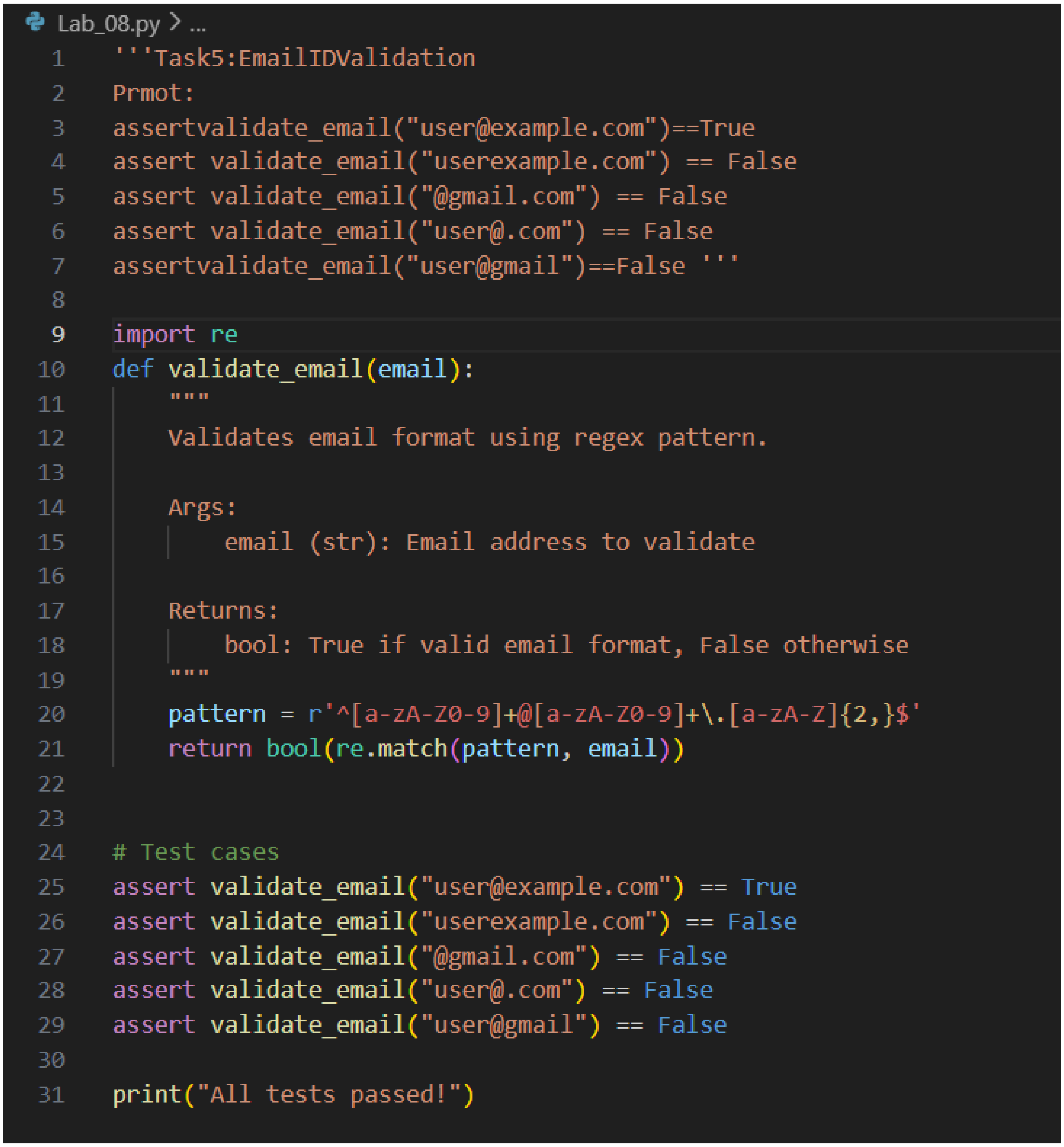
AI-generated test cases helped design object-oriented methods before implementation . The class correctly handled deposits,withdrawals,and balance retrieval. Test-driven development ensured correct behavior and reduced logical errors in financial operations.

Task5:EmailIDValidation

Prmot:

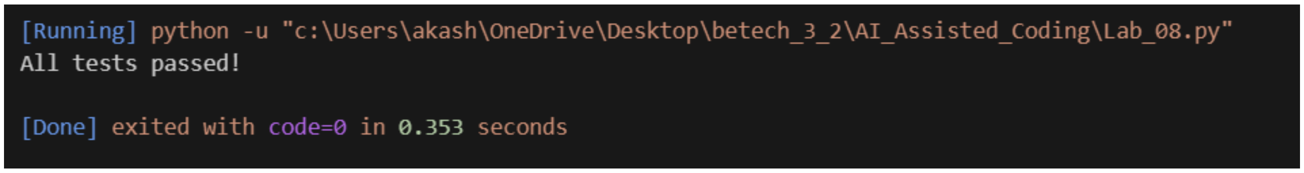
assertvalidate_email("user@example.com")==True
assert validate_email("userexample.com") == False
assert validate_email("@gmail.com") == False
assert validate_email("user@.com") == False
assertvalidate_email("user@gmail")==False

Code:



```
Lab_08.py > ...
'''Task5:EmailIDValidation
Prmot:
assertvalidate_email("user@example.com")==True
assert validate_email("userexample.com") == False
assert validate_email("@gmail.com") == False
assert validate_email("user@.com") == False
assertvalidate_email("user@gmail")==False '''

import re
def validate_email(email):
    """
    Validates email format using regex pattern.

    Args:
        email (str): Email address to validate

    Returns:
        bool: True if valid email format, False otherwise
    """
    pattern = r'^[a-zA-Z0-9]+@[a-zA-Z0-9]+\.[a-zA-Z]{2,}$'
    return bool(re.match(pattern, email))


# Test cases
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

print("All tests passed!")
```

Output:



```
[Running] python -u "c:\Users\akash\OneDrive\Desktop\betech_3_2\AI_Assisted_Coding\Lab_08.py"
All tests passed!

[Done] exited with code=0 in 0.353 seconds
```

Observation:

AI test cases guided the validation rules for email format.The function correctly checked for required symbols and invalid formats . Edgecases such as missing symbols and improper placement were handled effectively, improving data validation reliability