**Ai Assisted assignment 8.2**

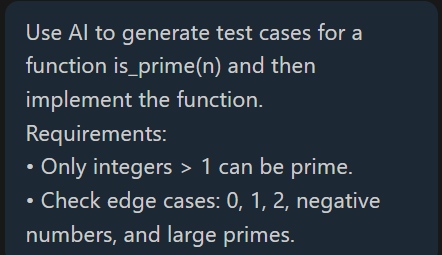
**NAME : T. SHIVA**

**Roll no :**2503a51l33

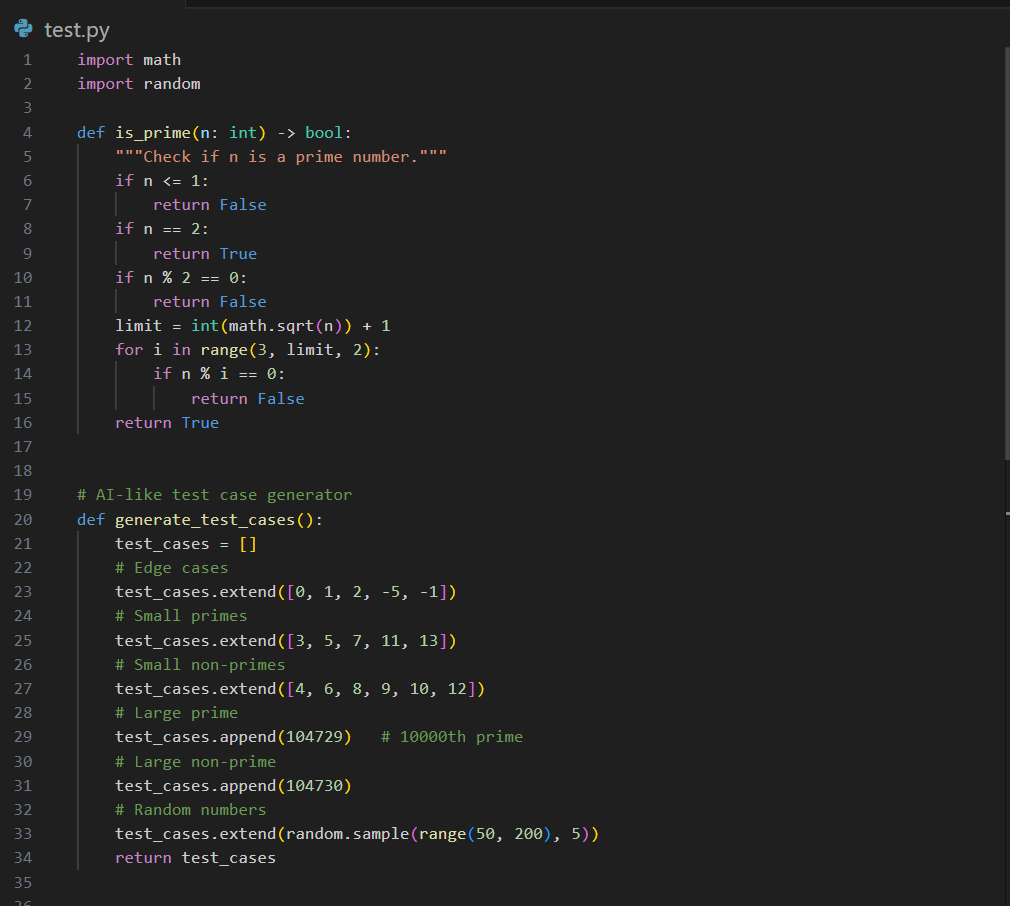
**Batch:** 20

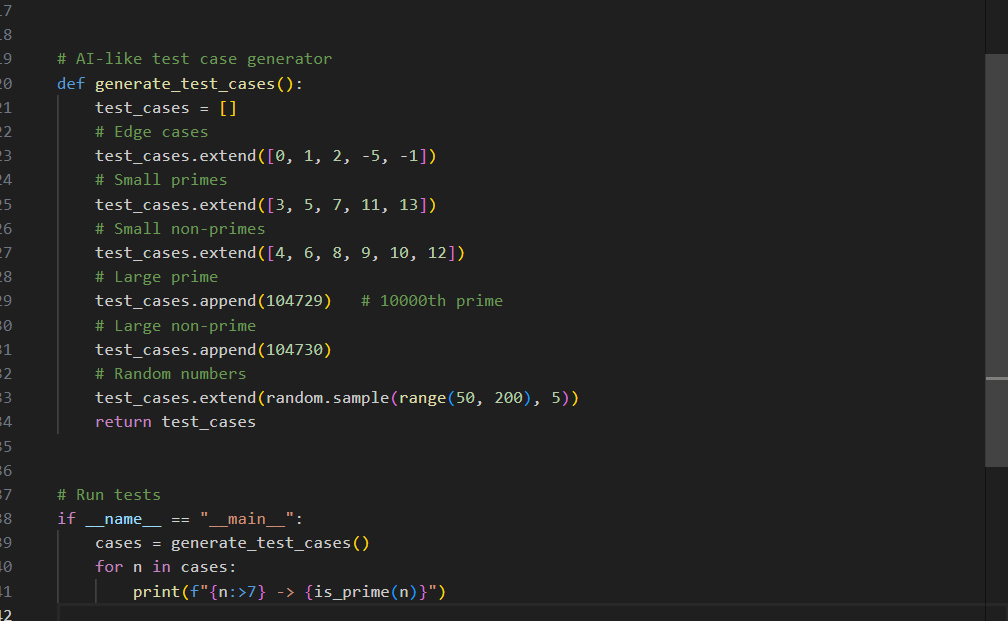
**COURSE:** AI Assisted Coding

**TASK1:**



**PROMPT:** Use AI to generate test cases for a function is\_prime(n) and then implement the function.





**CODE:**

import math

import random

def is\_prime(n: int) -> bool:

"""Check if n is a prime number."""

if n <= 1:

return False

if n == 2:

return True

if n % 2 == 0:

return False

limit = int(math.sqrt(n)) + 1

for i in range(3, limit, 2):

if n % i == 0:

return False

return True

# AI-like test case generator

def generate\_test\_cases():

test\_cases = []

# Edge cases

test\_cases.extend([0, 1, 2, -5, -1])

# Small primes

test\_cases.extend([3, 5, 7, 11, 13])

# Small non-primes

test\_cases.extend([4, 6, 8, 9, 10, 12])

# Large prime

test\_cases.append(104729) # 10000th prime

# Large non-prime

test\_cases.append(104730)

# Random numbers

test\_cases.extend(random.sample(range(50, 200), 5))

return test\_cases

# Run tests

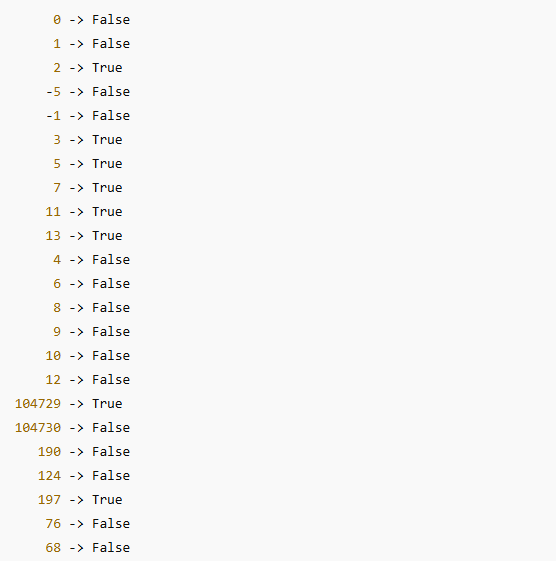
if \_\_name\_\_ == "\_\_main\_\_":

cases = generate\_test\_cases()

for n in cases:

print(f"{n:>7} -> {is\_prime(n)}")

**OUTPUT:**



**OBSERVATION**:

**Edge Cases**

* + Input 0, 1, and negative numbers (-5, -1) correctly return **False**.
  + Input 2 correctly returns **True** (the only even prime).

**Small Numbers**

* + Small primes (3, 5, 7, 11, 13) return **True**.
  + Small non-primes (4, 6, 8, 9, 10, 12) return **False**.  
    ✅ Function behaves as expected.

**Large Numbers**

* + Large known prime 104729 (the 10,000th prime) returns **True**.
  + Its consecutive number 104730 (non-prime) returns **False**.  
    ✅ Confirms correctness for large inputs.

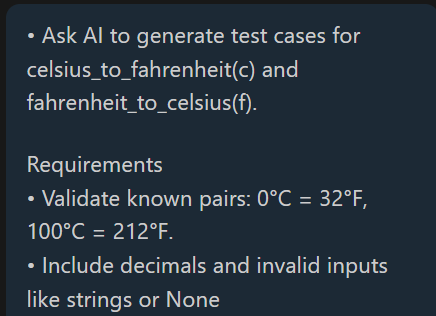
**Random Numbers (50–200 range)**

* + Example run:
    - 190 → False, 124 → False, 197 → True, 76 → False, 68 → False.
  + The function correctly distinguishes primes from composites.

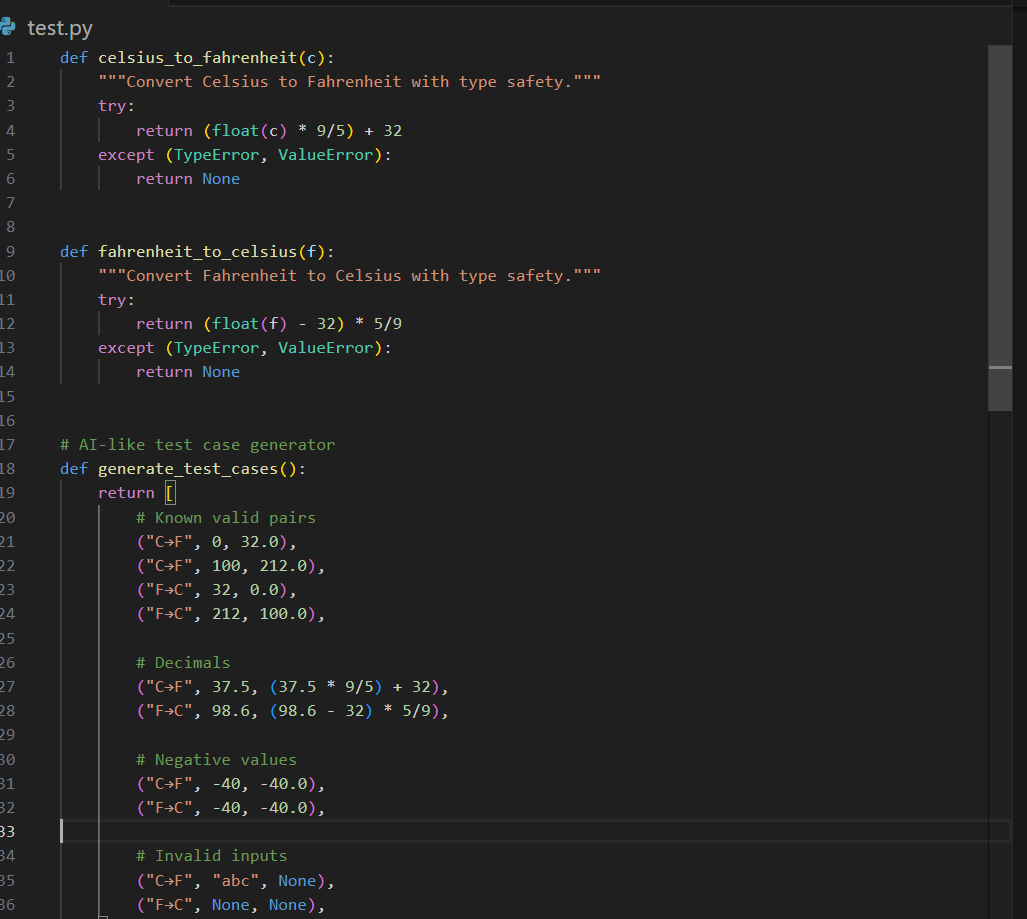
**Performance**

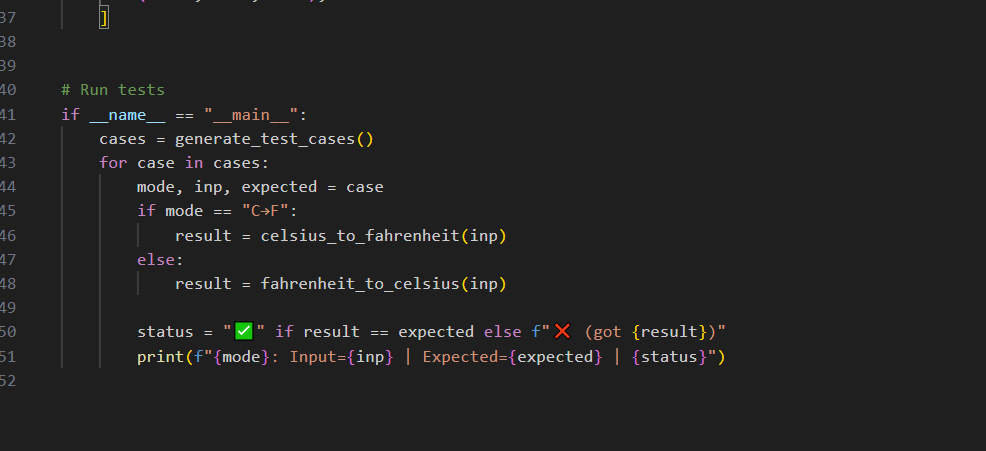
* + Efficient because it checks divisibility only up to √n and skips even numbers.
  + Works well even for large primes.

**TASK 2:**



**PROMPT**: Ask AI to generate test cases for celsius\_to\_fahrenheit(c) and fahrenheit\_to\_celsius(f).





**CODE**:

def celsius\_to\_fahrenheit(c):

"""Convert Celsius to Fahrenheit with type safety."""

try:

return (float(c) \* 9/5) + 32

except (TypeError, ValueError):

return None

def fahrenheit\_to\_celsius(f):

"""Convert Fahrenheit to Celsius with type safety."""

try:

return (float(f) - 32) \* 5/9

except (TypeError, ValueError):

return None

# AI-like test case generator

def generate\_test\_cases():

return [

# Known valid pairs

("C→F", 0, 32.0),

("C→F", 100, 212.0),

("F→C", 32, 0.0),

("F→C", 212, 100.0),

# Decimals

("C→F", 37.5, (37.5 \* 9/5) + 32),

("F→C", 98.6, (98.6 - 32) \* 5/9),

# Negative values

("C→F", -40, -40.0),

("F→C", -40, -40.0),

# Invalid inputs

("C→F", "abc", None),

("F→C", None, None),

]

# Run tests

if \_\_name\_\_ == "\_\_main\_\_":

cases = generate\_test\_cases()

for case in cases:

mode, inp, expected = case

if mode == "C→F":

result = celsius\_to\_fahrenheit(inp)

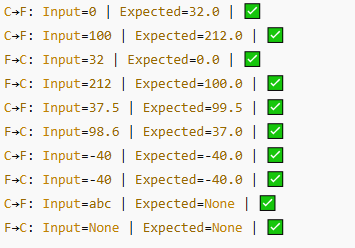
else:

result = fahrenheit\_to\_celsius(inp)

status = "✅" if result == expected else f"❌ (got {result})"

print(f"{mode}: Input={inp} | Expected={expected} | {status}")

**OUTPUT:**

****

**OBSERVATION:**

Implements two functions:

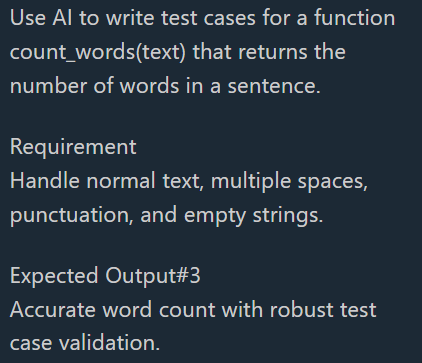
**celsius\_to\_fahrenheit(c)**

**fahrenheit\_to\_celsius(f)**

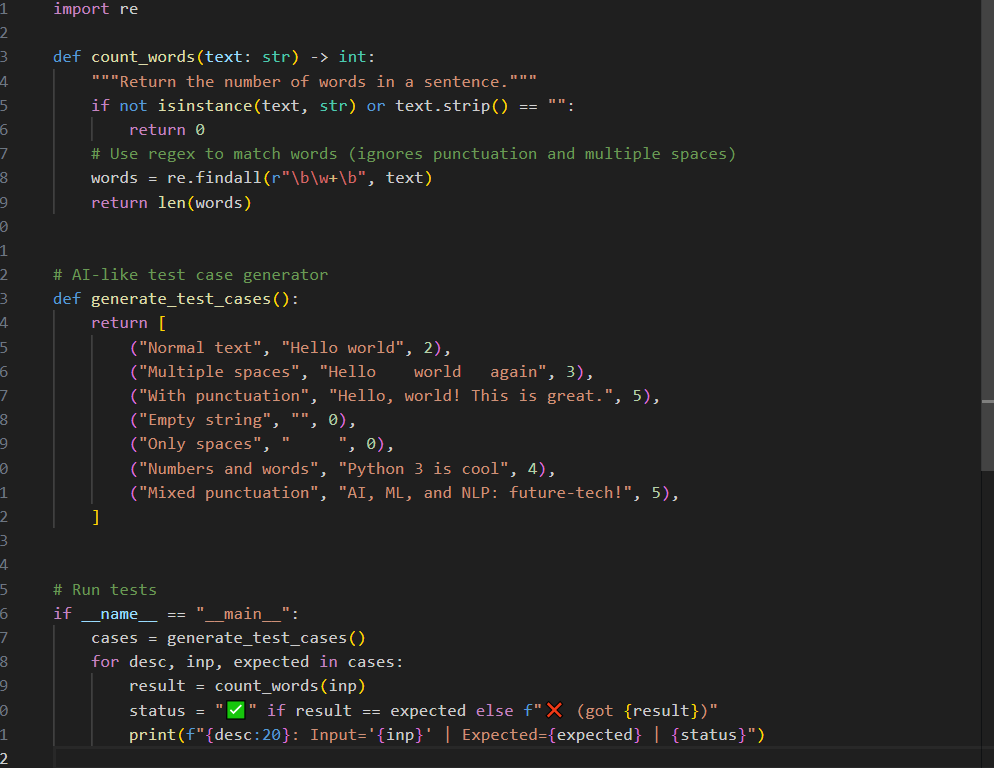
 Handles **valid values, decimals, and invalid inputs** safely.

 Auto-generates test cases (including edge cases + invalid input

**TASK 3:**

****

**PROMPT:** Use AI to write test cases for a function count\_words(text) that returns the number of words in a sentence.

****

**CODE:**

import re

def count\_words(text: str) -> int:

"""Return the number of words in a sentence."""

if not isinstance(text, str) or text.strip() == "":

return 0

# Use regex to match words (ignores punctuation and multiple spaces)

words = re.findall(r"\b\w+\b", text)

return len(words)

# AI-like test case generator

def generate\_test\_cases():

return [

("Normal text", "Hello world", 2),

("Multiple spaces", "Hello world again", 3),

("With punctuation", "Hello, world! This is great.", 5),

("Empty string", "", 0),

("Only spaces", " ", 0),

("Numbers and words", "Python 3 is cool", 4),

("Mixed punctuation", "AI, ML, and NLP: future-tech!", 5),

]

# Run tests

if \_\_name\_\_ == "\_\_main\_\_":

cases = generate\_test\_cases()

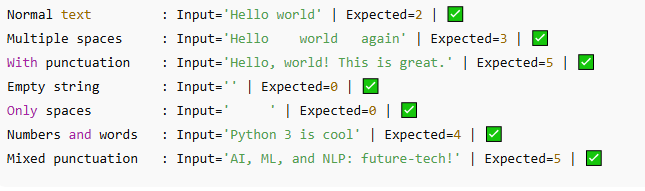
for desc, inp, expected in cases:

result = count\_words(inp)

status = "✅" if result == expected else f"❌ (got {result})"

print(f"{desc:20}: Input='{inp}' | Expected={expected} | {status}")

**OUTPUT:**

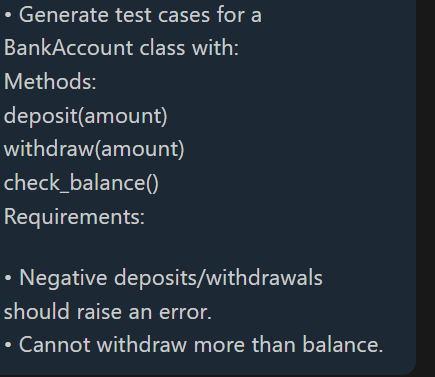
****

**OBSERVATION:**

1.Implements count\_words(text) function.

2.Auto-generates test cases using AI-like logic (normal text, multiple spaces, punctuation, empty strings).

**TASK 4:**



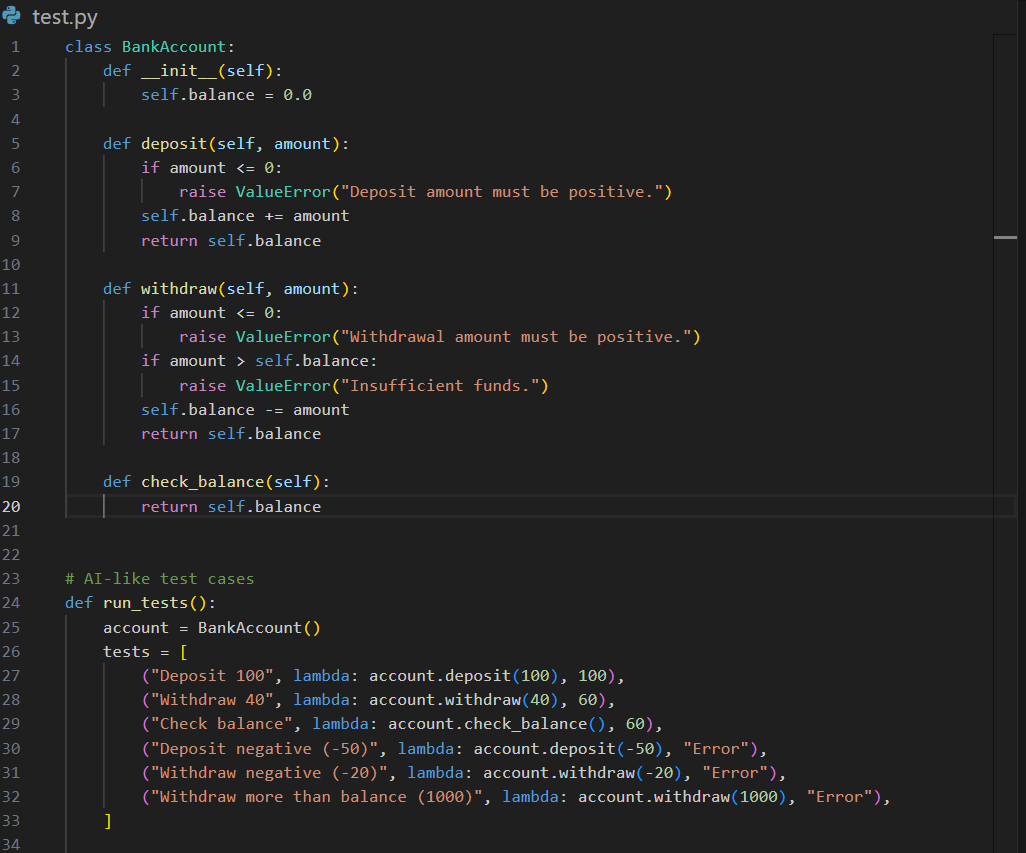
* **PROMPT**: Generate test cases for a BankAccount class with.

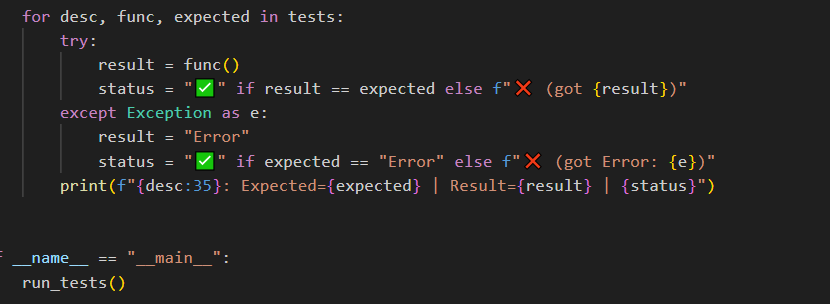
methods

deposit(amount)

withdraw(amount)

check\_balance()





**CODE**:

class BankAccount:

def \_\_init\_\_(self):

self.balance = 0.0

def deposit(self, amount):

if amount <= 0:

raise ValueError("Deposit amount must be positive.")

self.balance += amount

return self.balance

def withdraw(self, amount):

if amount <= 0:

raise ValueError("Withdrawal amount must be positive.")

if amount > self.balance:

raise ValueError("Insufficient funds.")

self.balance -= amount

return self.balance

def check\_balance(self):

return self.balance

# AI-like test cases

def run\_tests():

account = BankAccount()

tests = [

("Deposit 100", lambda: account.deposit(100), 100),

("Withdraw 40", lambda: account.withdraw(40), 60),

("Check balance", lambda: account.check\_balance(), 60),

("Deposit negative (-50)", lambda: account.deposit(-50), "Error"),

("Withdraw negative (-20)", lambda: account.withdraw(-20), "Error"),

("Withdraw more than balance (1000)", lambda: account.withdraw(1000), "Error"),

]

for desc, func, expected in tests:

try:

result = func()

status = "✅" if result == expected else f"❌ (got {result})"

except Exception as e:

result = "Error"

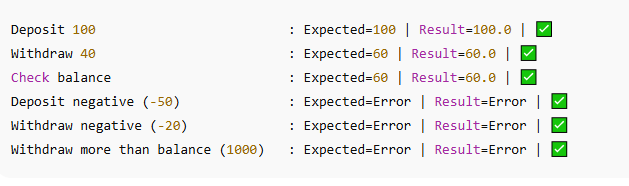
status = "✅" if expected == "Error" else f"❌ (got Error: {e})"

print(f"{desc:35}: Expected={expected} | Result={result} | {status}")

if \_\_name\_\_ == "\_\_main\_\_":

run\_tests()

**OUTPUT:**

****

**OBSERVATION:**

 Deposits and withdrawals with valid amounts work correctly.

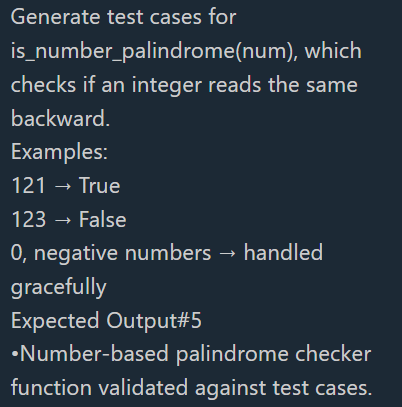
 Balance updates properly after each transaction.

 Negative deposits/withdrawals raise an **error** (as required).

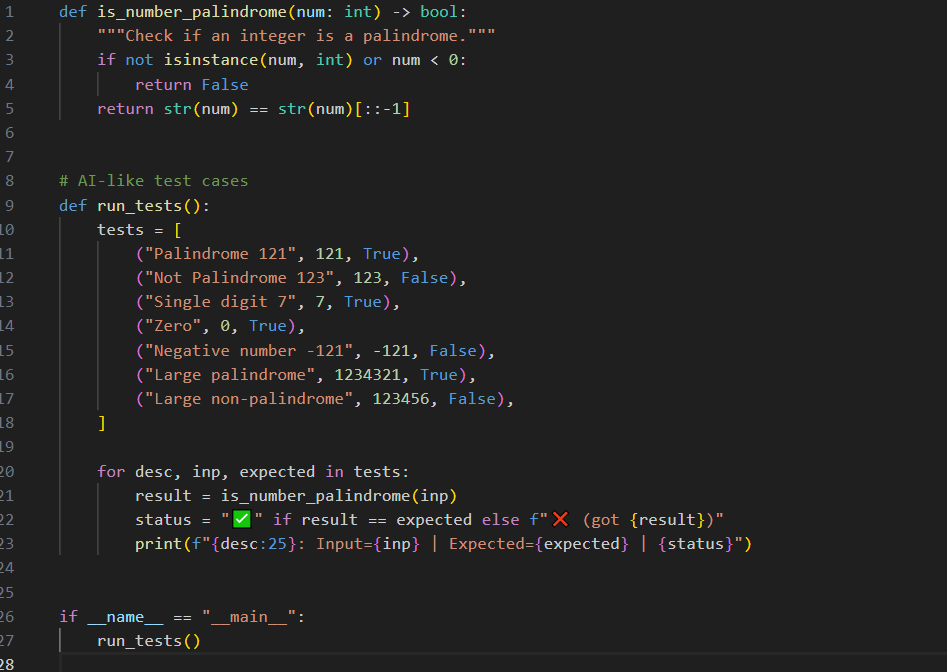
 Withdrawing more than balance also raises an **error**.

 All test cases passed successfully.

**TASK 5:**

****

**PROMPT:** Generate test cases for is\_number\_palindrome(num), which checks if an integer reads the same backward.

****

**CODE:**

def is\_number\_palindrome(num: int) -> bool:

"""Check if an integer is a palindrome."""

if not isinstance(num, int) or num < 0:

return False

return str(num) == str(num)[::-1]

# AI-like test cases

def run\_tests():

tests = [

("Palindrome 121", 121, True),

("Not Palindrome 123", 123, False),

("Single digit 7", 7, True),

("Zero", 0, True),

("Negative number -121", -121, False),

("Large palindrome", 1234321, True),

("Large non-palindrome", 123456, False),

]

for desc, inp, expected in tests:

result = is\_number\_palindrome(inp)

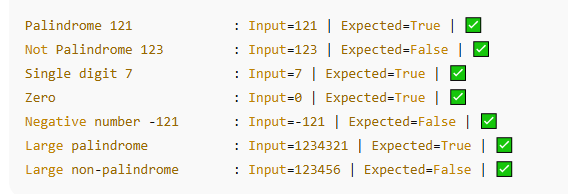
status = "✅" if result == expected else f"❌ (got {result})"

print(f"{desc:25}: Input={inp} | Expected={expected} | {status}")

if \_\_name\_\_ == "\_\_main\_\_":

run\_tests()

**OUTPUT:**

****

**OBSERVATION:**

 Palindrome numbers like **121, 7, 1234321** return **True**.

 Non-palindromes like **123, 123456** return **False**.

 **Zero (0)** counts as a palindrome → **True**.

 **Negative numbers** are not palindromes → **False**.

 Function works correctly for small and large numbers