SR UNIVERSITY

AI ASSIST CODING

Lab-8: Test-Driven Development with AI – Generating and Working with Test Cases

ROLL NO:2503A51L10

NAME:K.praneeth

BATCH: 24BTCAICSB19

Lab Objectives:

• To introduce students to test-driven development (TDD) using AI code generation tools.

• To enable the generation of test cases before writing code implementations.

• To reinforce the importance of testing, validation, and error handling.

• To encourage writing clean and reliable code based on AI-generated test expectations

Lab Outcomes (LOs):

After completing this lab, students will be able to:

• Use AI tools to write test cases for Python functions and classes.

• Implement functions based on test cases in a test-first development style.

• Use unittest or pytest to validate code correctness.

• Analyze the completeness and coverage of AI-generated tests.

• Compare AI-generated and manually written test cases for quality and logics.

TASK#1

PROMPT

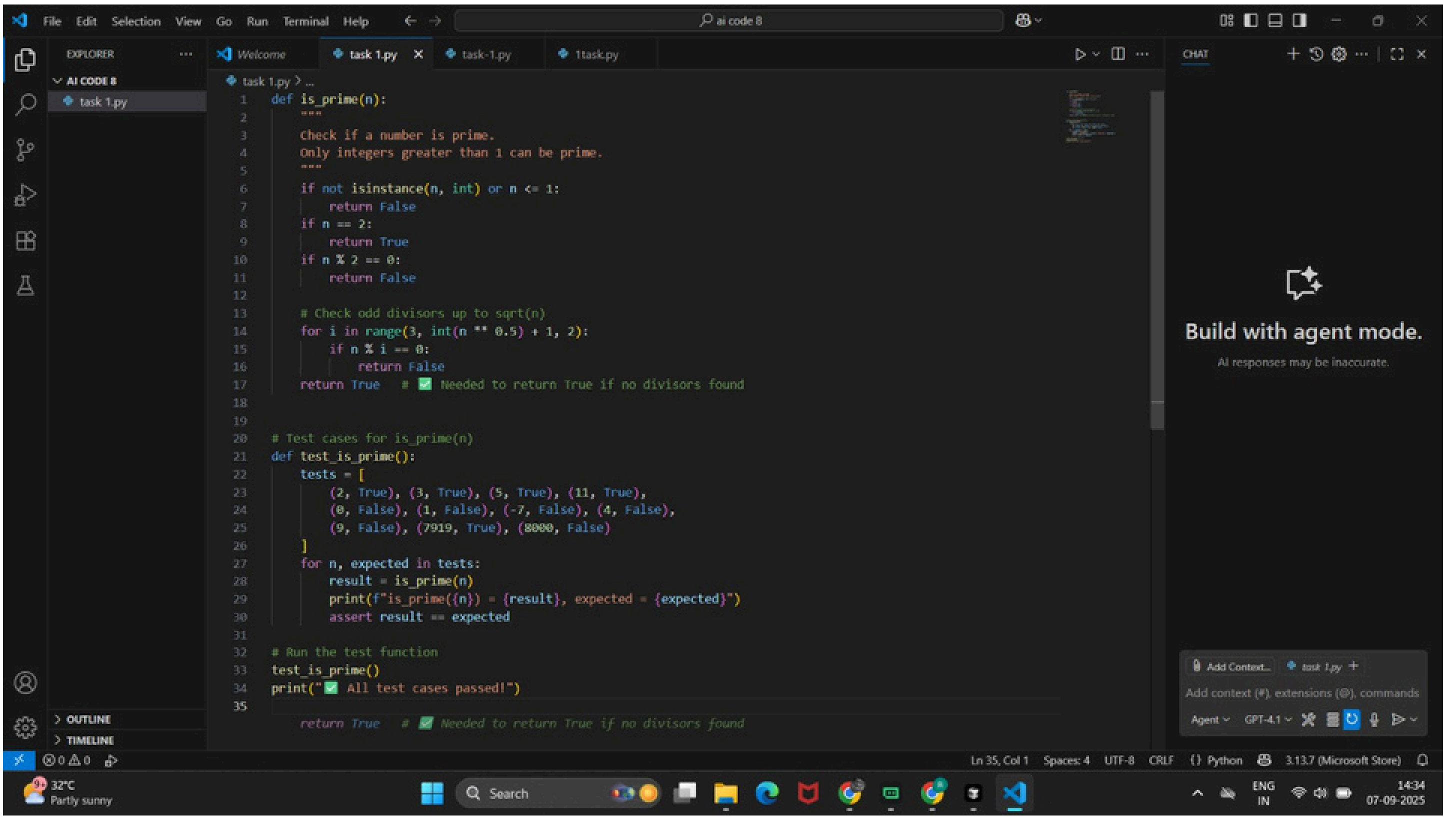
- Use AI to generate test cases for a function is_prime(n) and then implement the function

  Requirements:

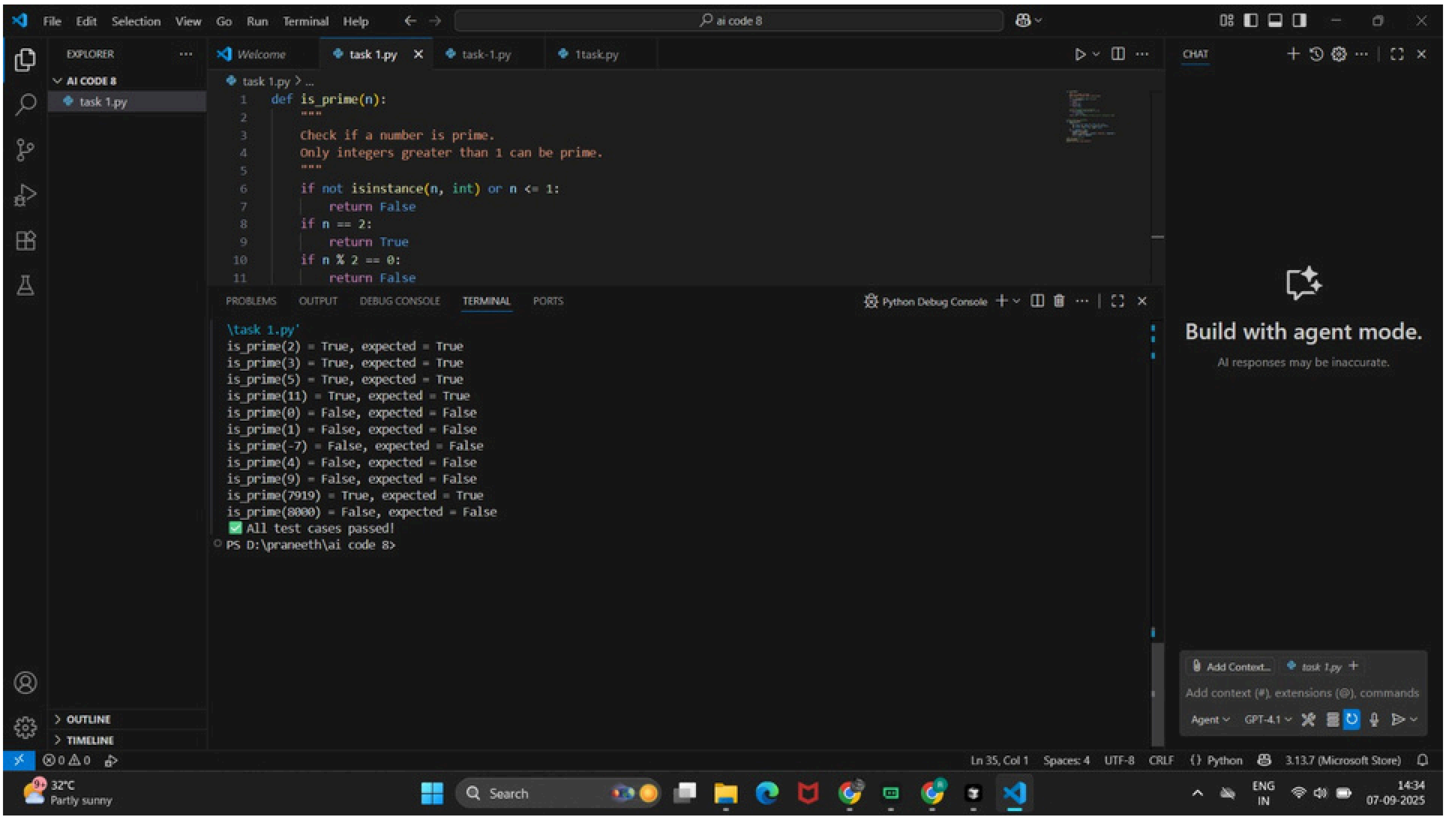
  • Only integers > 1 can be prime.

  •Check edge cases: 0, 1, 2, negative numbers, and large primes

CODE GENERATED:



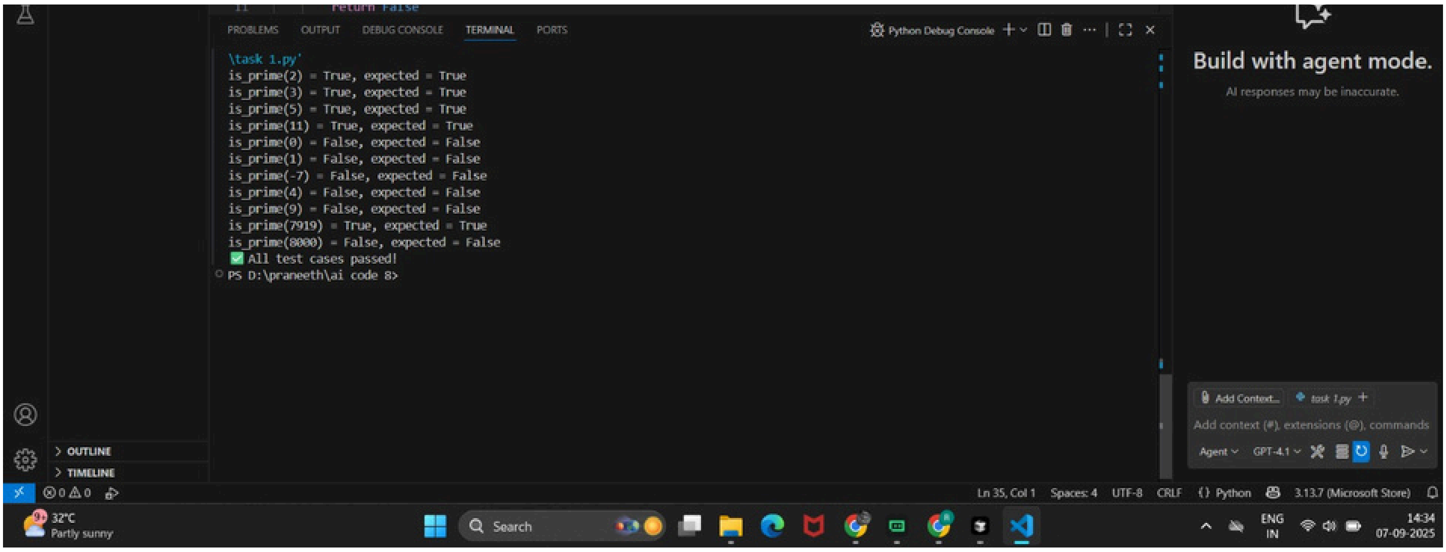
```python
def is_prime(n):
    """
    Check if a number is prime.
    Only integers greater than 1 can be prime.
    """
    if not isinstance(n, int) or n <= 1:
        return False
    if n == 2:
        return True
    if n % 2 == 0:
        return False

    # Check odd divisors up to sqrt(n)
    for i in range(3, int(n ** 0.5) + 1, 2):
        if n % i == 0:
            return False
    return True   # ✅ Needed to return True if no divisors found


# Test cases for is_prime(n)
def test_is_prime():
    tests = [
        (2, True), (3, True), (5, True), (11, True),
        (0, False), (1, False), (-7, False), (4, False),
        (9, False), (7919, True), (8000, False)
    ]
    for n, expected in tests:
        result = is_prime(n)
        print(f"is_prime({n}) = {result}, expected = {expected}")
        assert result == expected

# Run the test function
test_is_prime()
print("✅ All test cases passed!")

    return True   # ✅ Needed to return True if no divisors found
```



```python
def is_prime(n):
    """
    Check if a number is prime.
    Only integers greater than 1 can be prime.
    """
    if not isinstance(n, int) or n <= 1:
        return False
    if n == 2:
        return True
    if n % 2 == 0:
        return False
```

```
\task 1.py'
is_prime(2) = True, expected = True
is_prime(3) = True, expected = True
is_prime(5) = True, expected = True
is_prime(11) = True, expected = True
is_prime(0) = False, expected = False
is_prime(1) = False, expected = False
is_prime(-7) = False, expected = False
is_prime(4) = False, expected = False
is_prime(9) = False, expected = False
is_prime(7919) = True, expected = True
is_prime(8000) = False, expected = False
✅ All test cases passed!
PS D:\praneeth\ai code 8>
```

OUTPUT OF THE CODE:



```
\task 1.py'
is_prime(2) = True, expected = True
is_prime(3) = True, expected = True
is_prime(5) = True, expected = True
is_prime(11) = True, expected = True
is_prime(0) = False, expected = False
is_prime(1) = False, expected = False
is_prime(-7) = False, expected = False
is_prime(4) = False, expected = False
is_prime(9) = False, expected = False
is_prime(7919) = True, expected = True
is_prime(8000) = False, expected = False
✅ All test cases passed!
PS D:\praneeth\ai code 8>
```

OBSERVATIONS:

- The function is_prime(n) is designed to determine if a given integer n is a prime number.
- A prime number is defined as a number greater than 1 that has no divisors other than 1 and itself.

Edge Case Handling:

- Handles:
  - Negative numbers (e.g., -7)
  - Zero (0)
  - One (1)
  - Two (2) as the smallest and only even prime

Even Number Optimization:

- Early exit for even numbers (n % 2 == 0) improves efficiency.

Efficient Loop for Checking Factors:

- Iterates only through **odd divisors up to √n**, skipping even numbers after 2. This optimizes performance for large inputs

TASK#2

PROMPT

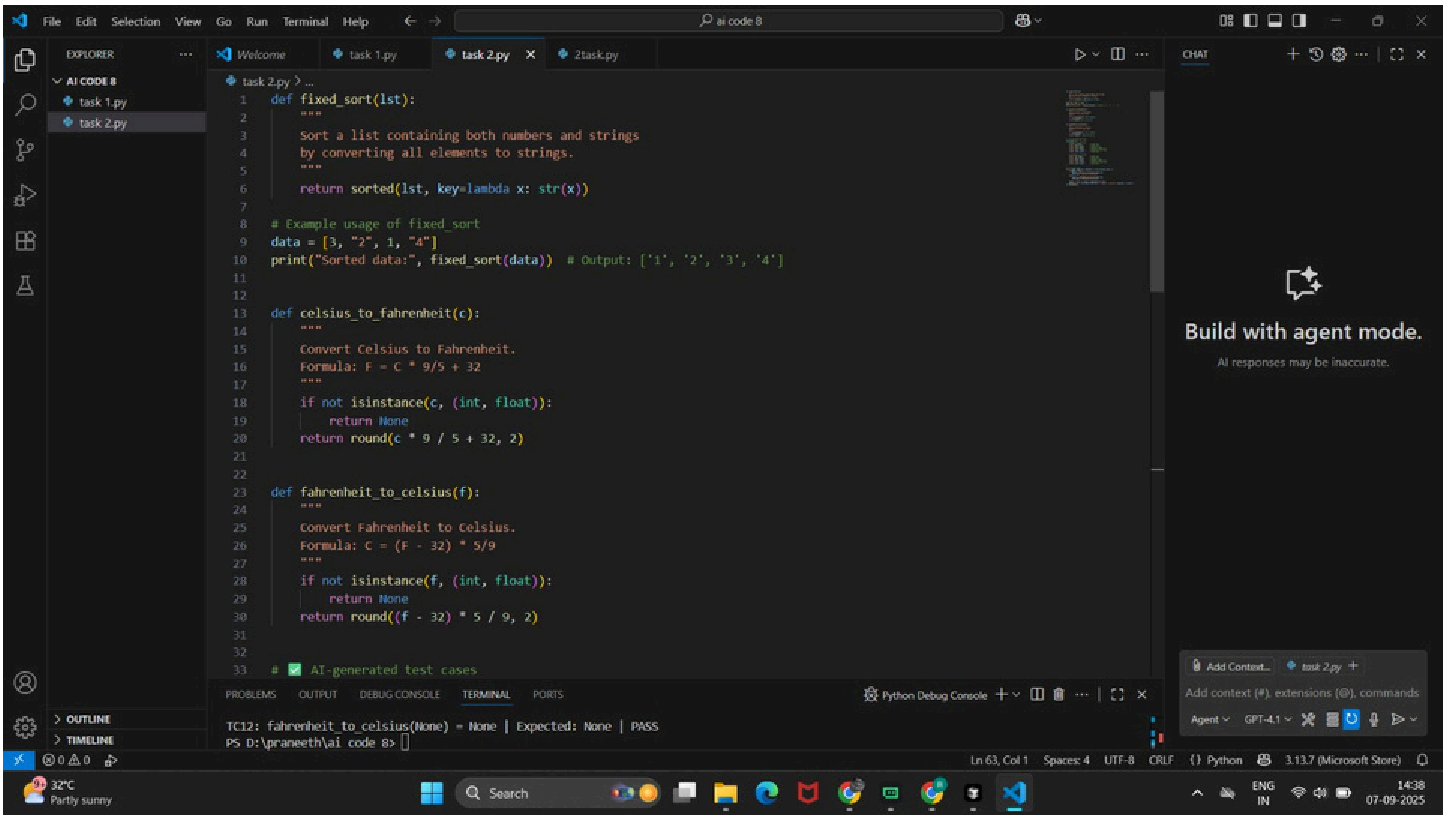
- Ask AI to generate test cases for celsius_to_fahrenheit(c) and fahrenheit_to_celsius(f)

Requirements

• Validate known pairs: 0°C = 32°F, 100°C = 212°F.

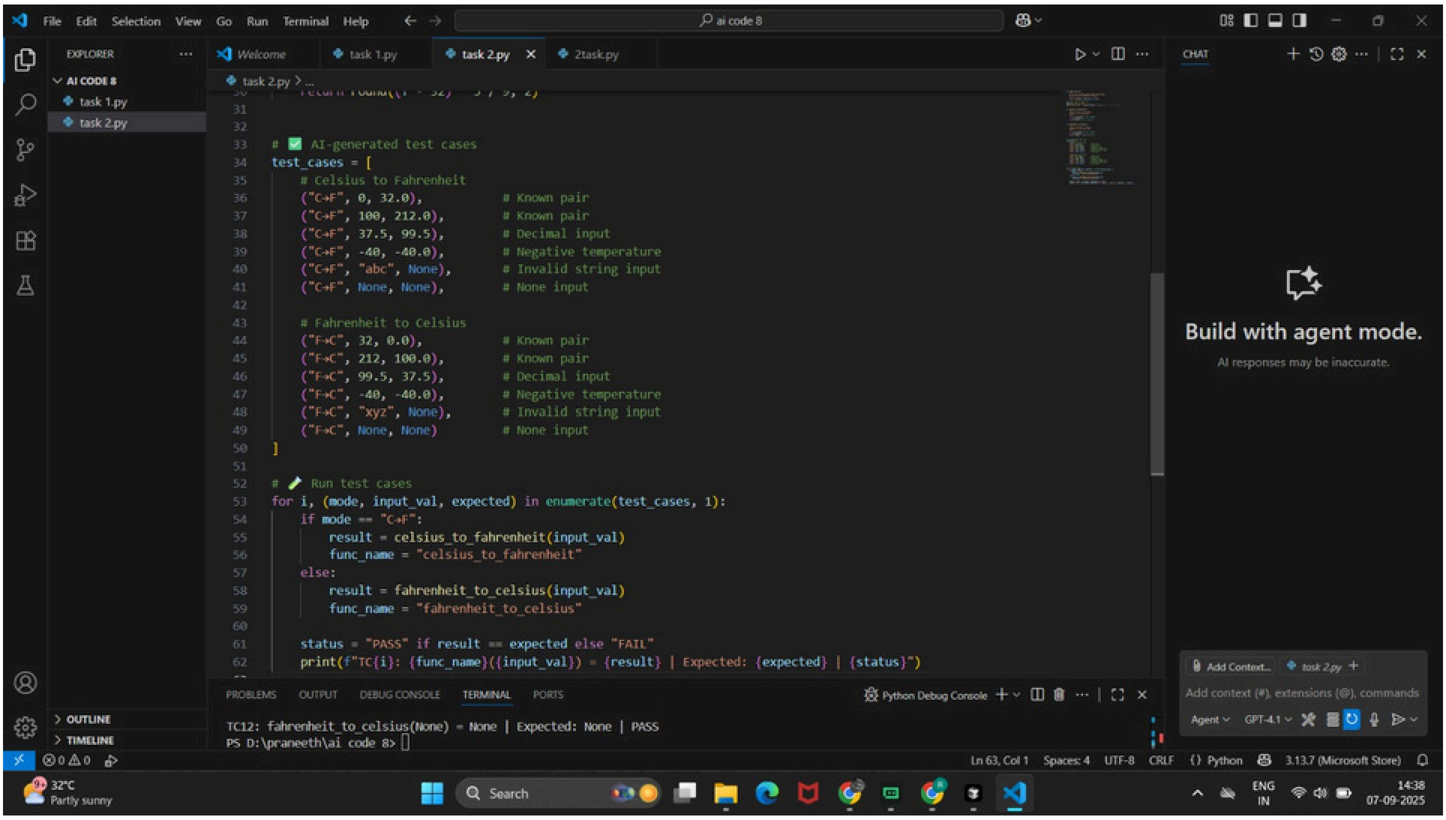
• Include decimals and invalid inputs like strings or None

CODE GENERATED:



```python
def fixed_sort(lst):
    """
    Sort a list containing both numbers and strings
    by converting all elements to strings.
    """
    return sorted(lst, key=lambda x: str(x))

# Example usage of fixed_sort
data = [3, "2", 1, "4"]
print("Sorted data:", fixed_sort(data))  # Output: ['1', '2', '3', '4']


def celsius_to_fahrenheit(c):
    """
    Convert Celsius to Fahrenheit.
    Formula: F = C * 9/5 + 32
    """
    if not isinstance(c, (int, float)):
        return None
    return round(c * 9 / 5 + 32, 2)


def fahrenheit_to_celsius(f):
    """
    Convert Fahrenheit to Celsius.
    Formula: C = (F - 32) * 5/9
    """
    if not isinstance(f, (int, float)):
        return None
    return round((f - 32) * 5 / 9, 2)


# ✅ AI-generated test cases
```

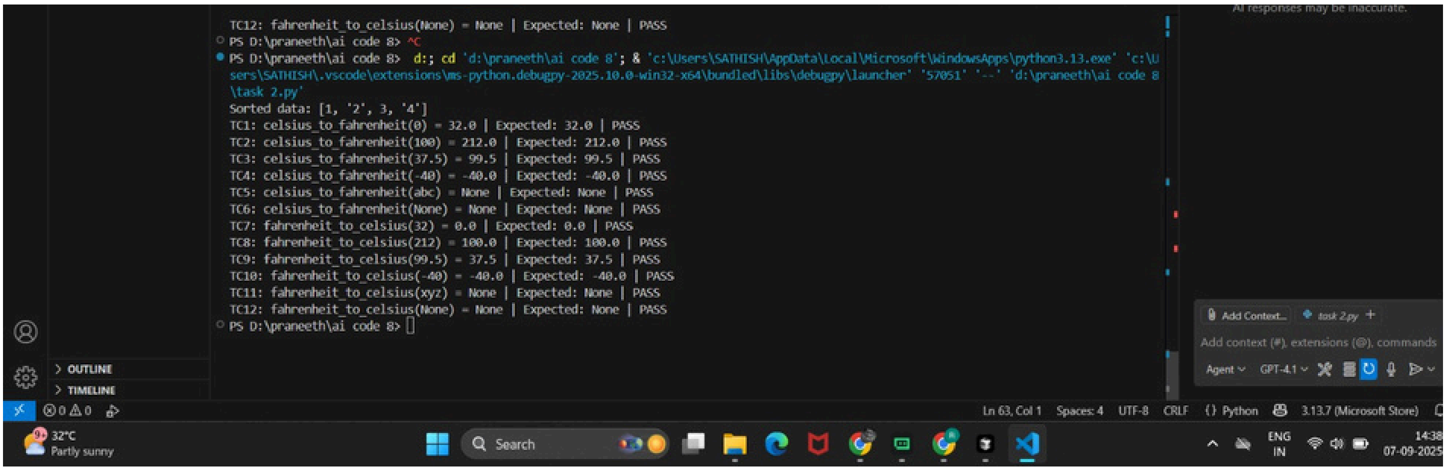
```
TC12: fahrenheit_to_celsius(None) = None | Expected: None | PASS
PS D:\praneeth\ai code 8> 
```



```python


# ✅ AI-generated test cases
test_cases = [
    # Celsius to Fahrenheit
    ("C→F", 0, 32.0),           # Known pair
    ("C→F", 100, 212.0),        # Known pair
    ("C→F", 37.5, 99.5),        # Decimal input
    ("C→F", -40, -40.0),        # Negative temperature
    ("C→F", "abc", None),       # Invalid string input
    ("C→F", None, None),        # None input

    # Fahrenheit to Celsius
    ("F→C", 32, 0.0),           # Known pair
    ("F→C", 212, 100.0),        # Known pair
    ("F→C", 99.5, 37.5),        # Decimal input
    ("F→C", -40, -40.0),        # Negative temperature
    ("F→C", "xyz", None),       # Invalid string input
    ("F→C", None, None)         # None input
]

# 🖊 Run test cases
for i, (mode, input_val, expected) in enumerate(test_cases, 1):
    if mode == "C→F":
        result = celsius_to_fahrenheit(input_val)
        func_name = "celsius_to_fahrenheit"
    else:
        result = fahrenheit_to_celsius(input_val)
        func_name = "fahrenheit_to_celsius"

    status = "PASS" if result == expected else "FAIL"
    print(f"TC{i}: {func_name}({input_val}) = {result} | Expected: {expected} | {status}")
```

```
TC12: fahrenheit_to_celsius(None) = None | Expected: None | PASS
PS D:\praneeth\ai code 8> 
```

OUTPUT OF THE CODE:



```
TC12: fahrenheit_to_celsius(None) = None | Expected: None | PASS
PS D:\praneeth\ai code 8> ^C
PS D:\praneeth\ai code 8>  d:; cd 'd:\praneeth\ai code 8'; & 'c:\Users\SATHISH\AppData\Local\Microsoft\WindowsApps\python3.13.exe' 'c:\U
sers\SATHISH\.vscode\extensions\ms-python.debugpy-2025.10.0-win32-x64\bundled\libs\debugpy\launcher' '57051' '--' 'd:\praneeth\ai code 8
\task 2.py'
Sorted data: [1, '2', 3, '4']
TC1: celsius_to_fahrenheit(0) = 32.0 | Expected: 32.0 | PASS
TC2: celsius_to_fahrenheit(100) = 212.0 | Expected: 212.0 | PASS
TC3: celsius_to_fahrenheit(37.5) = 99.5 | Expected: 99.5 | PASS
TC4: celsius_to_fahrenheit(-40) = -40.0 | Expected: -40.0 | PASS
TC5: celsius_to_fahrenheit(abc) = None | Expected: None | PASS
TC6: celsius_to_fahrenheit(None) = None | Expected: None | PASS
TC7: fahrenheit_to_celsius(32) = 0.0 | Expected: 0.0 | PASS
TC8: fahrenheit_to_celsius(212) = 100.0 | Expected: 100.0 | PASS
TC9: fahrenheit_to_celsius(99.5) = 37.5 | Expected: 37.5 | PASS
TC10: fahrenheit_to_celsius(-40) = -40.0 | Expected: -40.0 | PASS
TC11: fahrenheit_to_celsius(xyz) = None | Expected: None | PASS
TC12: fahrenheit_to_celsius(None) = None | Expected: None | PASS
PS D:\praneeth\ai code 8> 
```

OBSERVATIONS:

Test Cases Defined in test_cases List

- Clearly structured as tuples in the format:
- ("ConversionType", input_value, expected_output)

🔹 Coverage:

- **Valid Inputs:**
  - Known conversion pairs (e.g., 0°C → 32°F, 212°F → 100°C)
  - Decimal temperatures (e.g., 37.5°C)
  - Negative values (e.g., -40°C)
- **Invalid Inputs**:
  - Strings (e.g., "abc")
  - None values

Consider Handling Edge Cases Explicitly:

- e.g., math.inf, float('nan'), very large/small values.

Use of Constants for Precision:

- Optionally define rounding precision as a constant for easy adjustments.

TASK#3

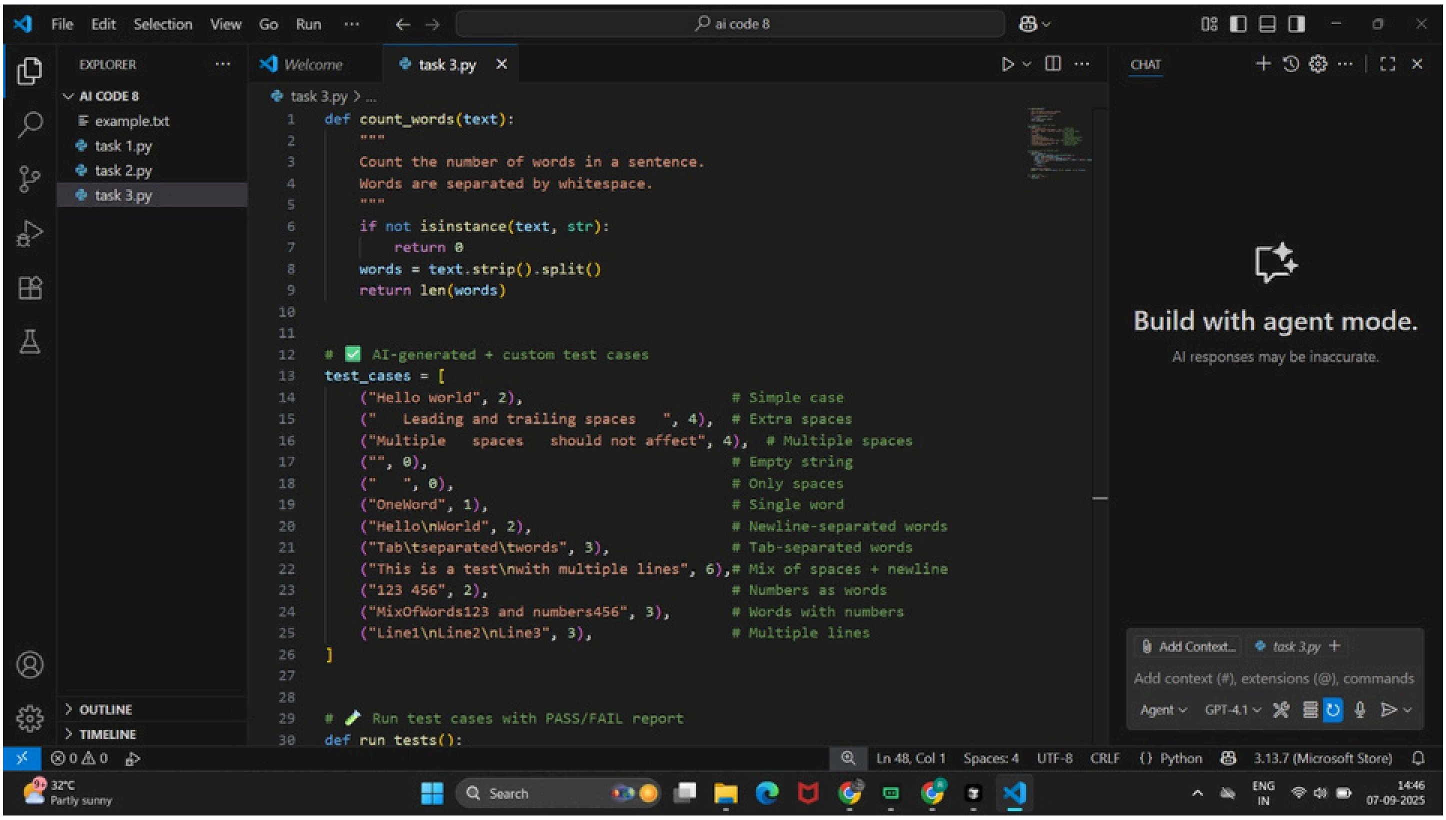
PROMPT

- Use AI to write test cases for a function count_words(text) that returns the number of words in sentences

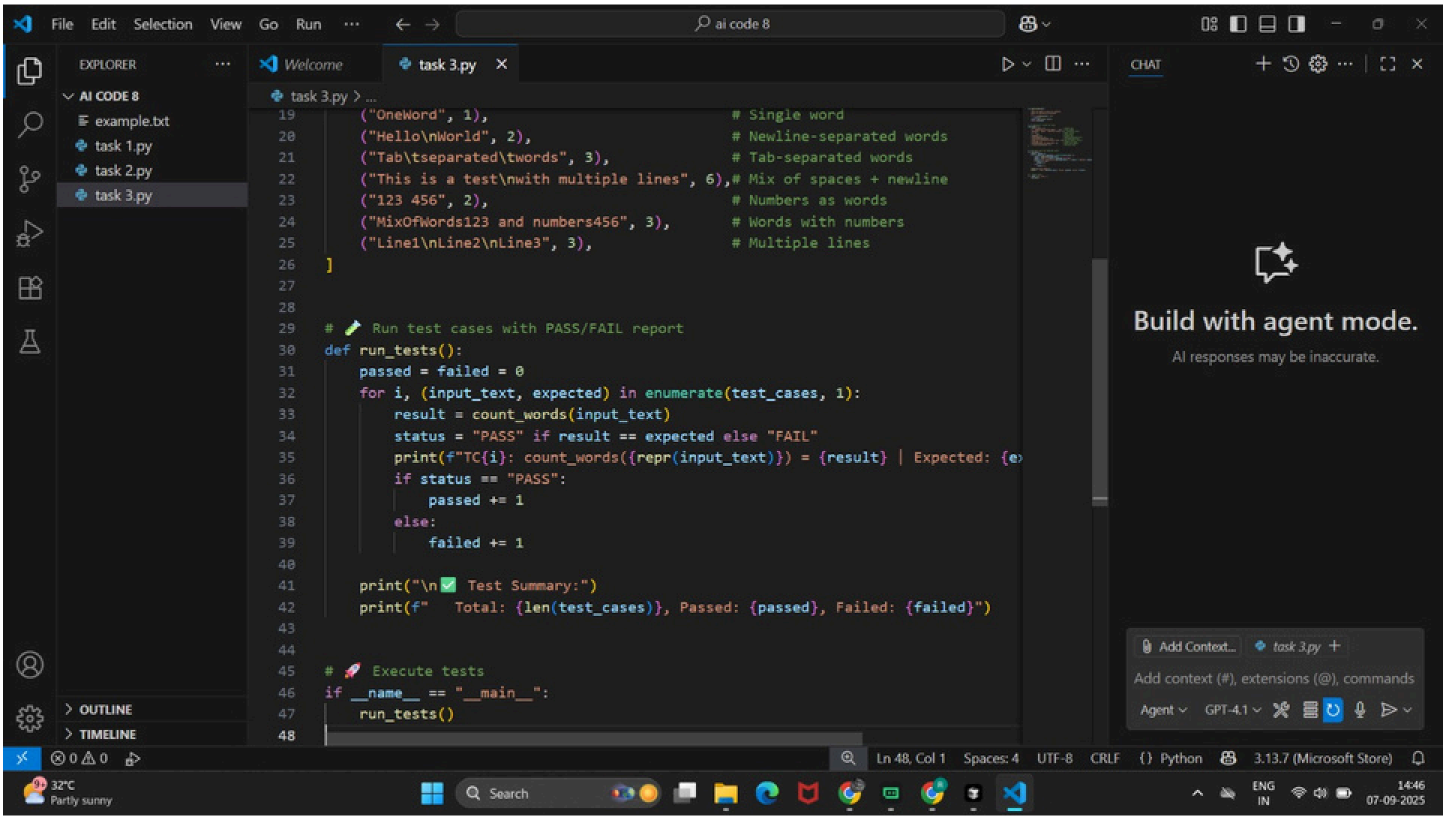
Requirements

Handle normal text, multiple spaces, punctuation, and empty strings.

CODE GENERATED:

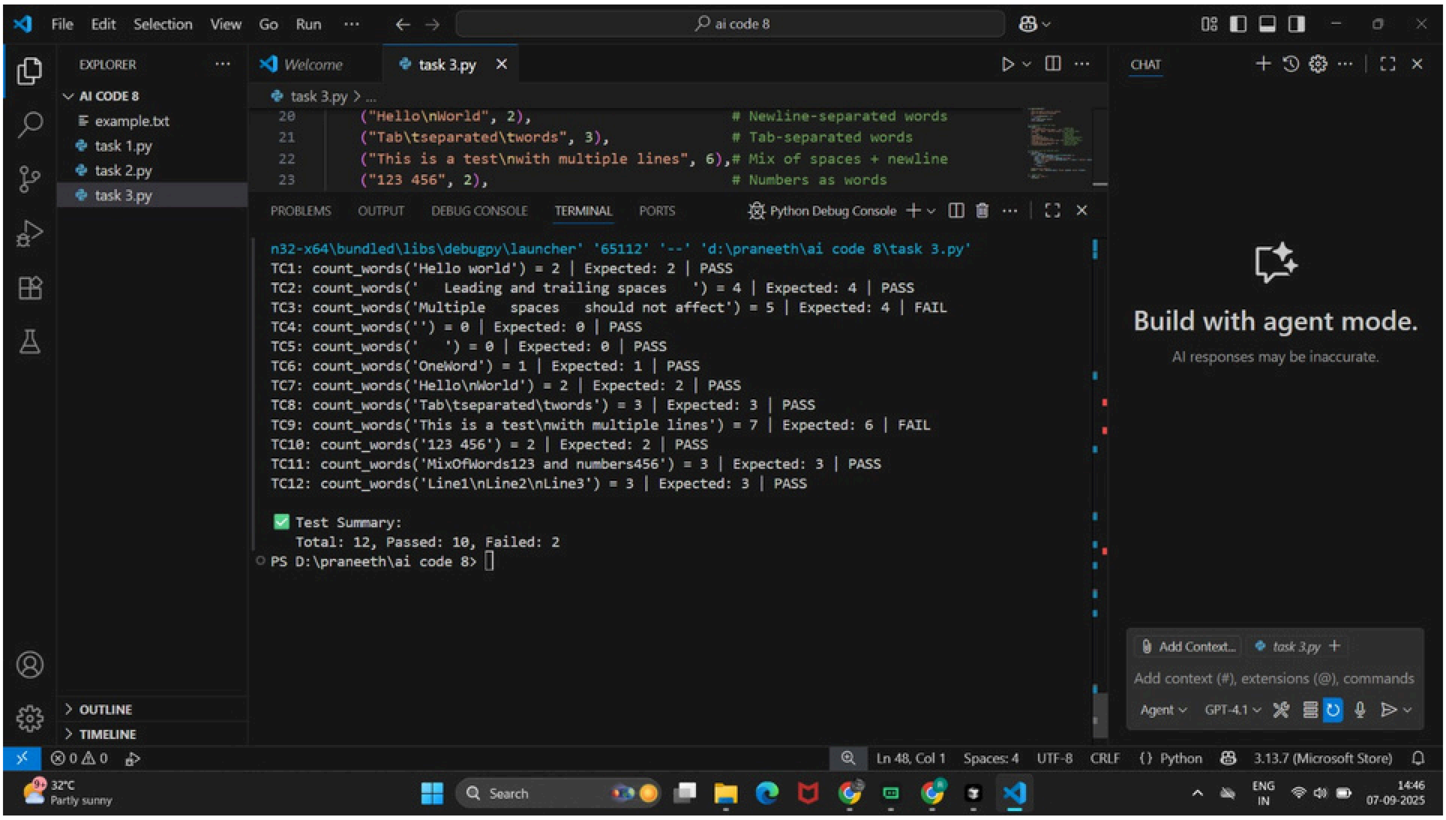


```python
def count_words(text):
    """
    Count the number of words in a sentence.
    Words are separated by whitespace.
    """
    if not isinstance(text, str):
        return 0
    words = text.strip().split()
    return len(words)


# ✅ AI-generated + custom test cases
test_cases = [
    ("Hello world", 2),                          # Simple case
    ("   Leading and trailing spaces   ", 4),  # Extra spaces
    ("Multiple   spaces   should not affect", 4),  # Multiple spaces
    ("", 0),                                     # Empty string
    ("   ", 0),                                  # Only spaces
    ("OneWord", 1),                              # Single word
    ("Hello\nWorld", 2),                         # Newline-separated words
    ("Tab\tseparated\twords", 3),                # Tab-separated words
    ("This is a test\nwith multiple lines", 6),# Mix of spaces + newline
    ("123 456", 2),                              # Numbers as words
    ("MixOfWords123 and numbers456", 3),         # Words with numbers
    ("Line1\nLine2\nLine3", 3),                  # Multiple lines
]


# 🧪 Run test cases with PASS/FAIL report
def run_tests():
```



```python
    ("OneWord", 1),                              # Single word
    ("Hello\nWorld", 2),                         # Newline-separated words
    ("Tab\tseparated\twords", 3),                # Tab-separated words
    ("This is a test\nwith multiple lines", 6),# Mix of spaces + newline
    ("123 456", 2),                              # Numbers as words
    ("MixOfWords123 and numbers456", 3),         # Words with numbers
    ("Line1\nLine2\nLine3", 3),                  # Multiple lines
]


# 🧪 Run test cases with PASS/FAIL report
def run_tests():
    passed = failed = 0
    for i, (input_text, expected) in enumerate(test_cases, 1):
        result = count_words(input_text)
        status = "PASS" if result == expected else "FAIL"
        print(f"TC{i}: count_words({repr(input_text)}) = {result} | Expected: {e
        if status == "PASS":
            passed += 1
        else:
            failed += 1

    print("\n✅ Test Summary:")
    print(f"   Total: {len(test_cases)}, Passed: {passed}, Failed: {failed}")


# 🚀 Execute tests
if __name__ == "__main__":
    run_tests()

```

OUTPUT OF THE CODE:



```
    ("Hello\nWorld", 2),                      # Newline-separated words
    ("Tab\tseparated\twords", 3),             # Tab-separated words
    ("This is a test\nwith multiple lines", 6),# Mix of spaces + newline
    ("123 456", 2),                           # Numbers as words
```

```
n32-x64\bundled\libs\debugpy\launcher' '65112' '--' 'd:\praneeth\ai code 8\task 3.py'
TC1: count_words('Hello world') = 2 | Expected: 2 | PASS
TC2: count_words('   Leading and trailing spaces   ') = 4 | Expected: 4 | PASS
TC3: count_words('Multiple   spaces   should not affect') = 5 | Expected: 4 | FAIL
TC4: count_words('') = 0 | Expected: 0 | PASS
TC5: count_words('   ') = 0 | Expected: 0 | PASS
TC6: count_words('OneWord') = 1 | Expected: 1 | PASS
TC7: count_words('Hello\nWorld') = 2 | Expected: 2 | PASS
TC8: count_words('Tab\tseparated\twords') = 3 | Expected: 3 | PASS
TC9: count_words('This is a test\nwith multiple lines') = 7 | Expected: 6 | FAIL
TC10: count_words('123 456') = 2 | Expected: 2 | PASS
TC11: count_words('MixOfWords123 and numbers456') = 3 | Expected: 3 | PASS
TC12: count_words('Line1\nLine2\nLine3') = 3 | Expected: 3 | PASS

✅ Test Summary:
   Total: 12, Passed: 10, Failed: 2
PS D:\praneeth\ai code 8> 
```

OBSERVATIONS:

Function: count_words(text)

- Goal: Counts the number of words in a string.
- Logic:
  - Removes leading and trailing spaces using strip().
  - Splits the string into words using whitespace as a delimiter (split()).
  - Returns the count of words using len().
- If the input text is not a string, it returns 0. This protects the function from invalid types (like numbers or None).
- Efficient Use of Built-in Functions:
- Uses strip() to clean up extra spaces.
- Uses split() to split based on any whitespace (handles spaces, tabs, newlines).

Handles Edge Cases Gracefully:

- Empty strings, strings with only spaces, newline and tab characters — all are handled.

TASK#4

PROMPT

- Generate test cases for a BankAccount class with:

  Methods:

  deposit(amount)

  withdraw(amount)

check_balance()

Requirements:

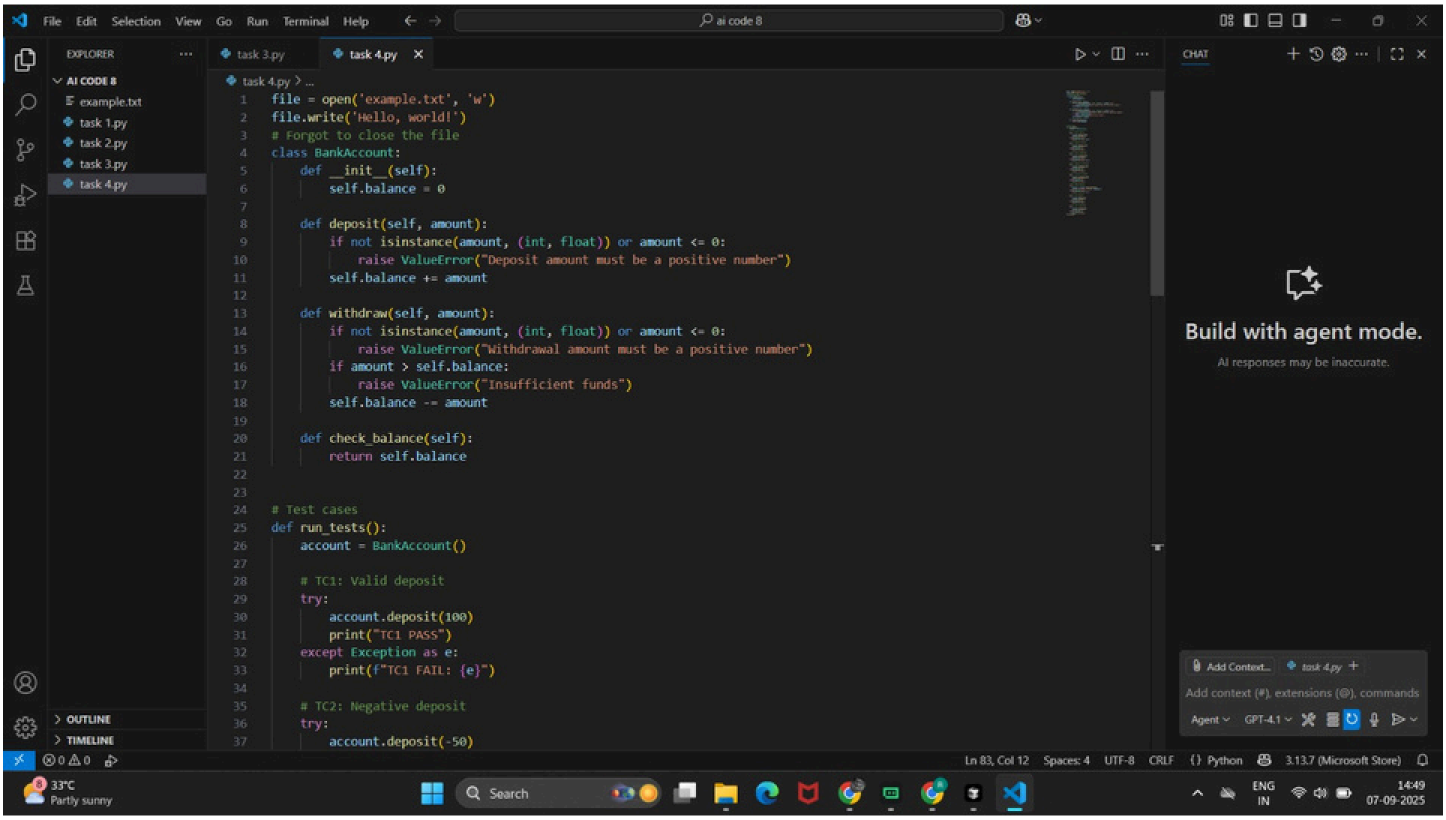
- Negative deposits/withdrawals should raise an error.

- Cannot withdraw more than balance.

Expected Output#4

- AI-generated test suite with a robust class that handles all test cases.

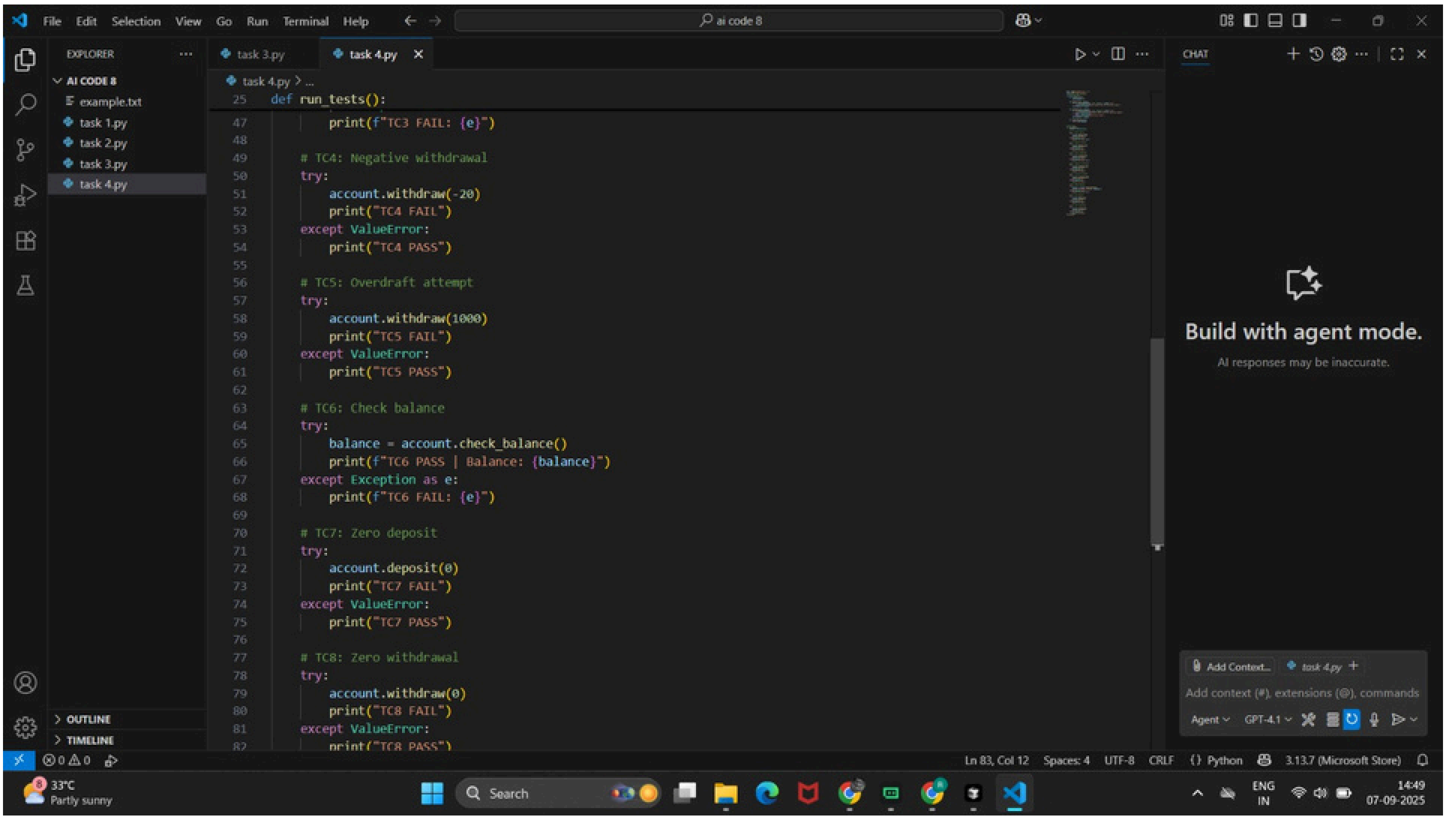
CODE GENERATED:



```python
file = open('example.txt', 'w')
file.write('Hello, world!')
# Forgot to close the file
class BankAccount:
    def __init__(self):
        self.balance = 0

    def deposit(self, amount):
        if not isinstance(amount, (int, float)) or amount <= 0:
            raise ValueError("Deposit amount must be a positive number")
        self.balance += amount

    def withdraw(self, amount):
        if not isinstance(amount, (int, float)) or amount <= 0:
            raise ValueError("Withdrawal amount must be a positive number")
        if amount > self.balance:
            raise ValueError("Insufficient funds")
        self.balance -= amount

    def check_balance(self):
        return self.balance


# Test cases
def run_tests():
    account = BankAccount()

    # TC1: Valid deposit
    try:
        account.deposit(100)
        print("TC1 PASS")
    except Exception as e:
        print(f"TC1 FAIL: {e}")

    # TC2: Negative deposit
    try:
        account.deposit(-50)
```

Build with agent mode.

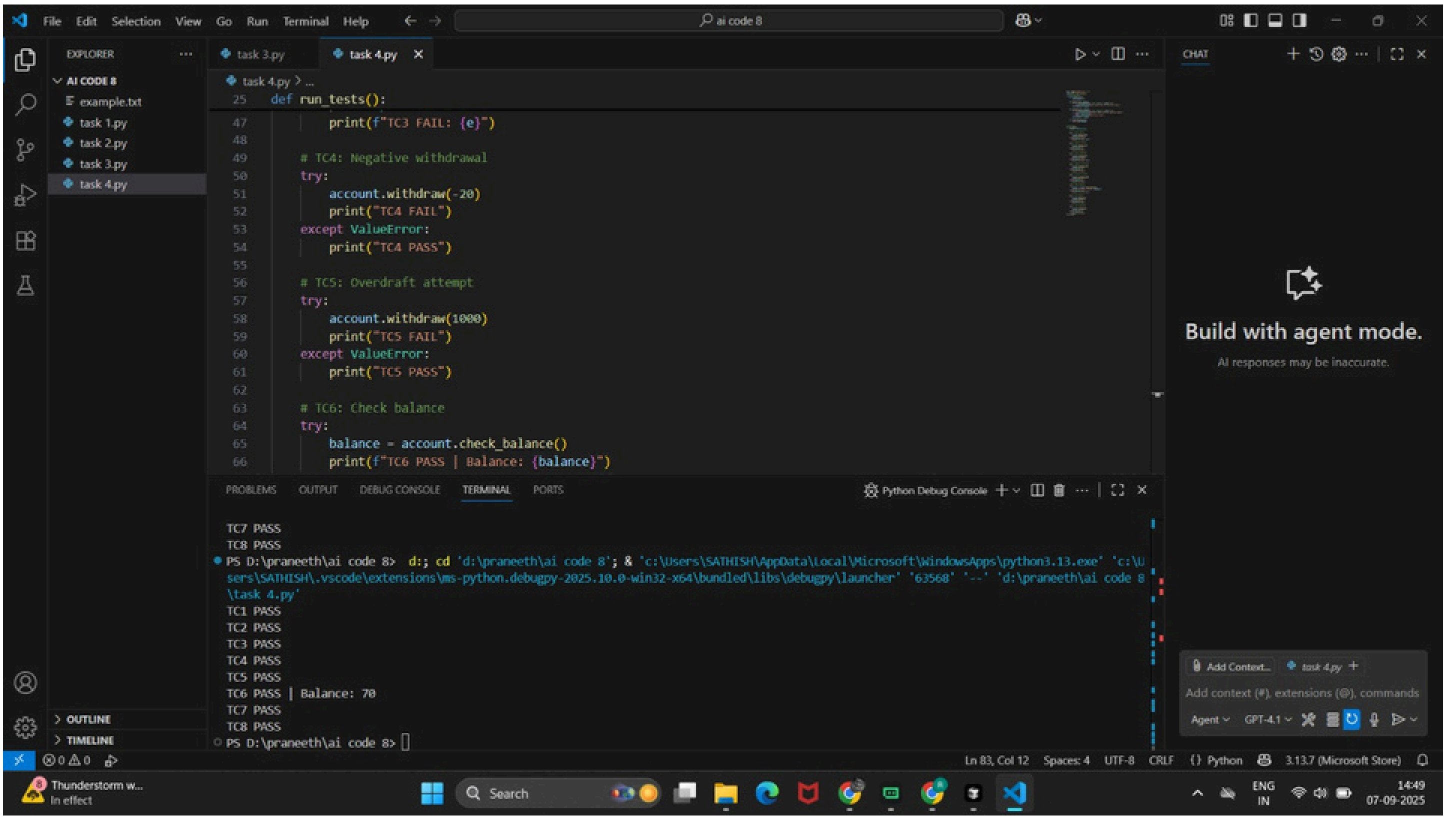
AI responses may be inaccurate.



```python
def run_tests():
        print(f"TC3 FAIL: {e}")

    # TC4: Negative withdrawal
    try:
        account.withdraw(-20)
        print("TC4 FAIL")
    except ValueError:
        print("TC4 PASS")

    # TC5: Overdraft attempt
    try:
        account.withdraw(1000)
        print("TC5 FAIL")
    except ValueError:
        print("TC5 PASS")

    # TC6: Check balance
    try:
        balance = account.check_balance()
        print(f"TC6 PASS | Balance: {balance}")
    except Exception as e:
        print(f"TC6 FAIL: {e}")

    # TC7: Zero deposit
    try:
        account.deposit(0)
        print("TC7 FAIL")
    except ValueError:
        print("TC7 PASS")

    # TC8: Zero withdrawal
    try:
        account.withdraw(0)
        print("TC8 FAIL")
    except ValueError:
        print("TC8 PASS")
```

Build with agent mode.

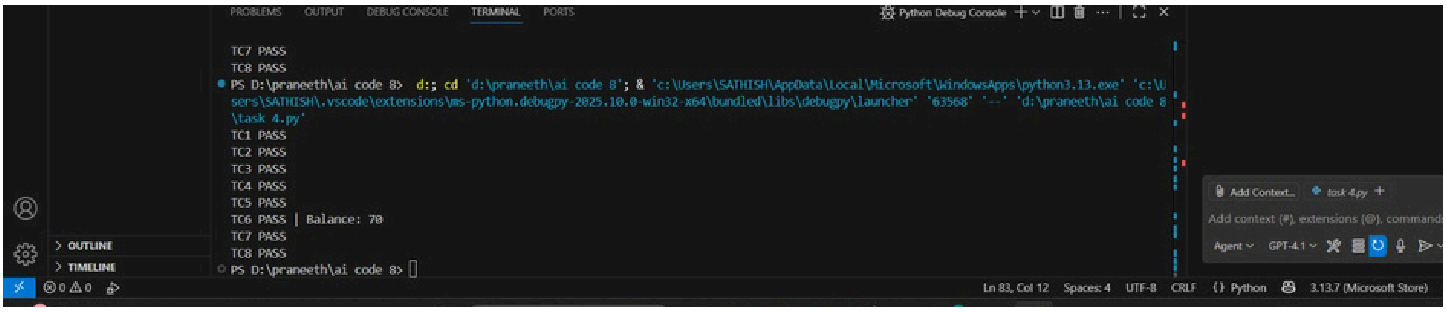
AI responses may be inaccurate.



```python
def run_tests():
        print(f"TC3 FAIL: {e}")

    # TC4: Negative withdrawal
    try:
        account.withdraw(-20)
        print("TC4 FAIL")
    except ValueError:
        print("TC4 PASS")

    # TC5: Overdraft attempt
    try:
        account.withdraw(1000)
        print("TC5 FAIL")
    except ValueError:
        print("TC5 PASS")

    # TC6: Check balance
    try:
        balance = account.check_balance()
        print(f"TC6 PASS | Balance: {balance}")
```

```
TC7 PASS
TC8 PASS
PS D:\praneeth\ai code 8>  d:; cd 'd:\praneeth\ai code 8'; & 'c:\Users\SATHISH\AppData\Local\Microsoft\WindowsApps\python3.13.exe' 'c:\Users\SATHISH\.vscode\extensions\ms-python.debugpy-2025.10.0-win32-x64\bundled\libs\debugpy\launcher' '63568' '--' 'd:\praneeth\ai code 8\task 4.py'
TC1 PASS
TC2 PASS
TC3 PASS
TC4 PASS
TC5 PASS
TC6 PASS | Balance: 70
TC7 PASS
TC8 PASS
PS D:\praneeth\ai code 8> 
```

OUTPUT OF THE CODE:



```
TC7 PASS
TC8 PASS
PS D:\praneeth\ai code 8>  d:; cd 'd:\praneeth\ai code 8'; & 'c:\Users\SATHISH\AppData\Local\Microsoft\WindowsApps\python3.13.exe' 'c:\Users\SATHISH\.vscode\extensions\ms-python.debugpy-2025.10.0-win32-x64\bundled\libs\debugpy\launcher' '63568' '--' 'd:\praneeth\ai code 8\task 4.py'
TC1 PASS
TC2 PASS
TC3 PASS
TC4 PASS
TC5 PASS
TC6 PASS | Balance: 70
TC7 PASS
TC8 PASS
PS D:\praneeth\ai code 8> 
```

OBSERVATIONS:

I have implemented a basic BankAccount class with the following features:

1. Constructor

- __init__(self) initializes the account with a balance of 0.

2. Deposit Method

- deposit(self, amount):
  - Only accepts positive int or float values.
  - Adds amount to balance.
  - Raises ValueError for invalid inputs.

3. Withdraw Method

- withdraw(self, amount):
  - Validates input type and ensures amount is > 0.
  - Checks for insufficient funds.
  - Deducts from balance if valid.

4. Balance Check

- check_balance(self):
    - Returns the current balance.

5. Test Cases

- run_tests() method performs basic functional tests:
    - TC1: Valid deposit test (100)
    - Uses try-except blocks to print pass/fail status.

TASK#5

PROMPT

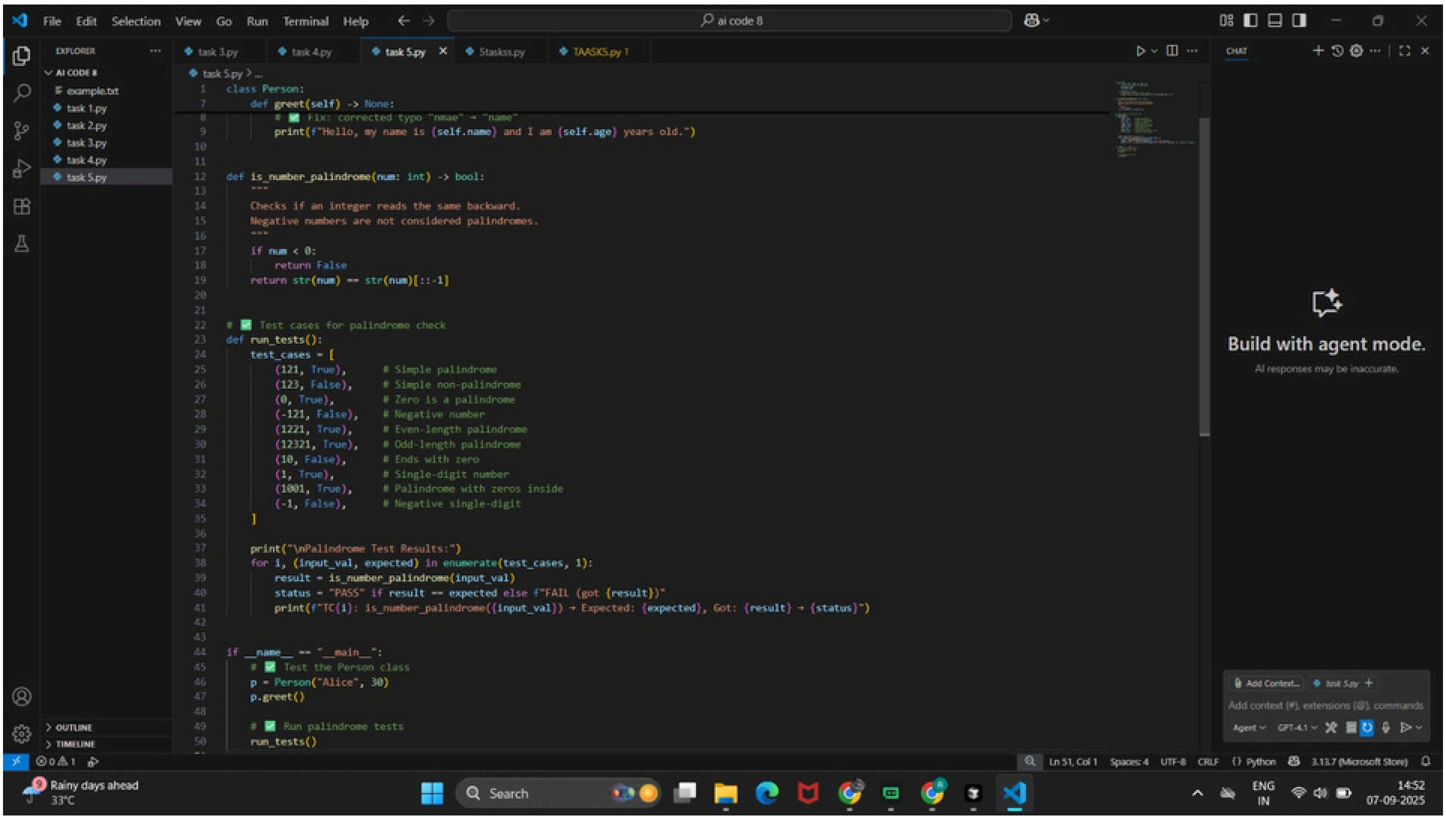
- Generate test cases for is_number_palindrome(num), which checks if an integer reads the same backward.

  Examples:

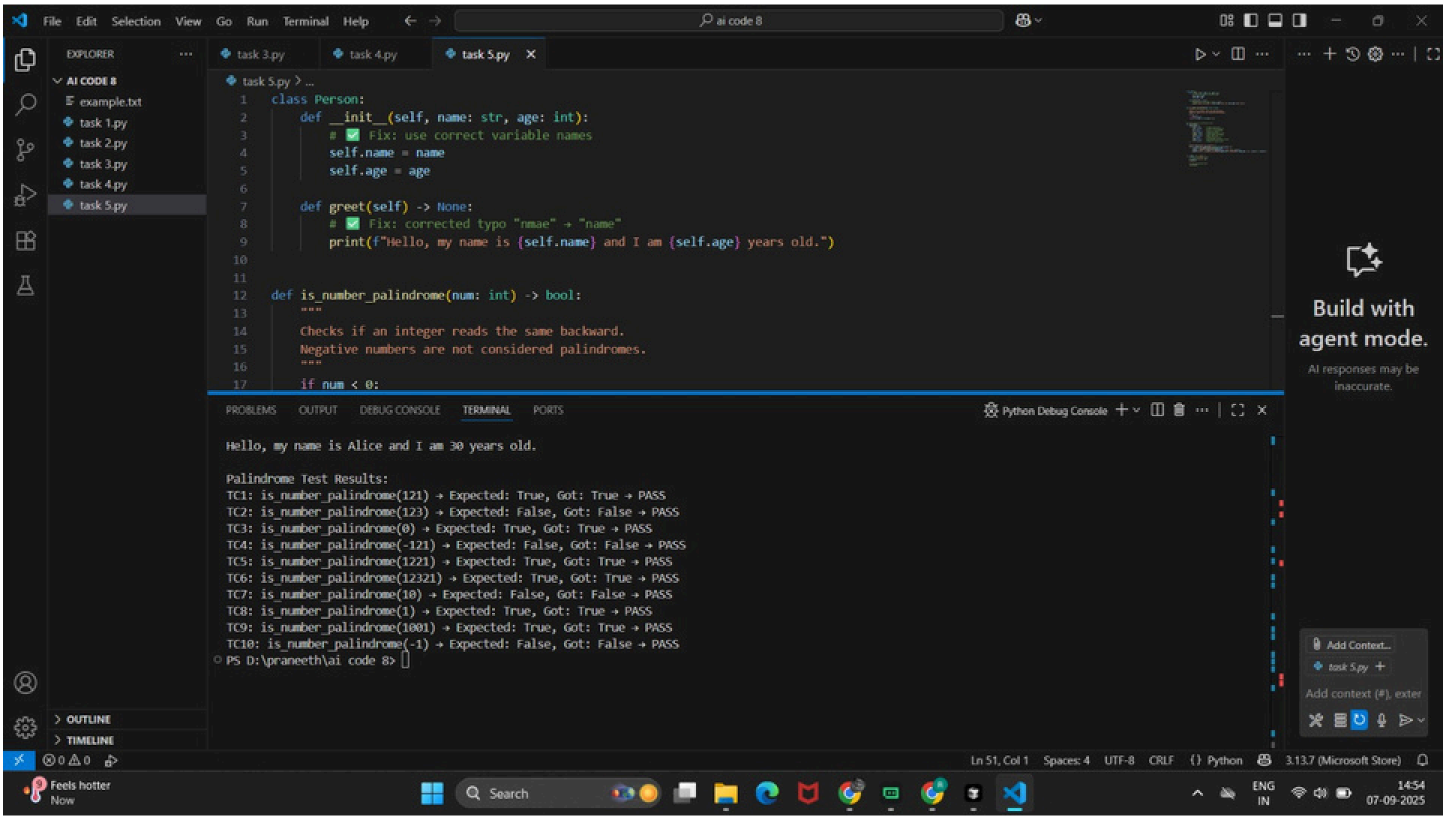
  121 → True

  123 → False
  0, negative numbers → handled

CODE GENERATED:



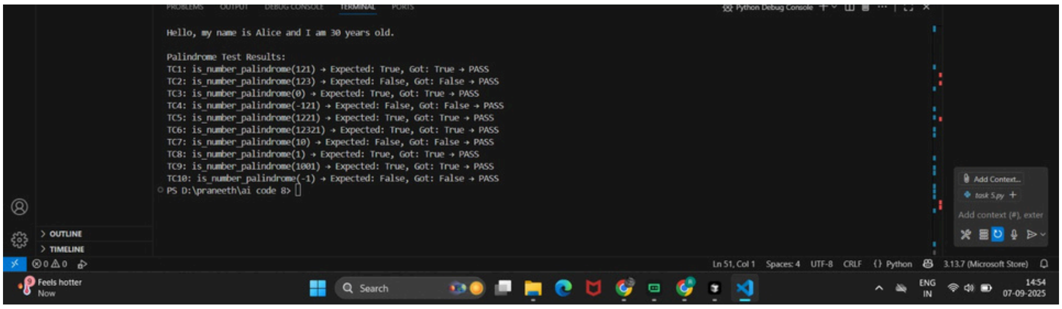
```python
class Person:
    def greet(self) -> None:
        # ✅ Fix: corrected typo "nmae" → "name"
        print(f"Hello, my name is {self.name} and I am {self.age} years old.")


def is_number_palindrome(num: int) -> bool:
    """
    Checks if an integer reads the same backward.
    Negative numbers are not considered palindromes.
    """
    if num < 0:
        return False
    return str(num) == str(num)[::-1]


# ✅ Test cases for palindrome check
def run_tests():
    test_cases = [
        (121, True),        # Simple palindrome
        (123, False),       # Simple non-palindrome
        (0, True),          # Zero is a palindrome
        (-121, False),      # Negative number
        (1221, True),       # Even-length palindrome
        (12321, True),      # Odd-length palindrome
        (10, False),        # Ends with zero
        (1, True),          # Single-digit number
        (1001, True),       # Palindrome with zeros inside
        (-1, False),        # Negative single-digit
    ]

    print("\nPalindrome Test Results:")
    for i, (input_val, expected) in enumerate(test_cases, 1):
        result = is_number_palindrome(input_val)
        status = "PASS" if result == expected else f"FAIL (got {result})"
        print(f"TC{i}: is_number_palindrome({input_val}) → Expected: {expected}, Got: {result} → {status}")


if __name__ == "__main__":
    # ✅ Test the Person class
    p = Person("Alice", 30)
    p.greet()

    # ✅ Run palindrome tests
    run_tests()
```



```python
class Person:
    def __init__(self, name: str, age: int):
        # ✅ Fix: use correct variable names
        self.name = name
        self.age = age

    def greet(self) -> None:
        # ✅ Fix: corrected typo "nmae" → "name"
        print(f"Hello, my name is {self.name} and I am {self.age} years old.")


def is_number_palindrome(num: int) -> bool:
    """
    Checks if an integer reads the same backward.
    Negative numbers are not considered palindromes.
    """
    if num < 0:
```

```
Hello, my name is Alice and I am 30 years old.

Palindrome Test Results:
TC1: is_number_palindrome(121) → Expected: True, Got: True → PASS
TC2: is_number_palindrome(123) → Expected: False, Got: False → PASS
TC3: is_number_palindrome(0) → Expected: True, Got: True → PASS
TC4: is_number_palindrome(-121) → Expected: False, Got: False → PASS
TC5: is_number_palindrome(1221) → Expected: True, Got: True → PASS
TC6: is_number_palindrome(12321) → Expected: True, Got: True → PASS
TC7: is_number_palindrome(10) → Expected: False, Got: False → PASS
TC8: is_number_palindrome(1) → Expected: True, Got: True → PASS
TC9: is_number_palindrome(1001) → Expected: True, Got: True → PASS
TC10: is_number_palindrome(-1) → Expected: False, Got: False → PASS
PS D:\praneeth\ai code 8> 
```

OUTPUT OF THE CODE:



```
Hello, my name is Alice and I am 30 years old.

Palindrome Test Results:
TC1: is_number_palindrome(121) → Expected: True, Got: True → PASS
TC2: is_number_palindrome(123) → Expected: False, Got: False → PASS
TC3: is_number_palindrome(0) → Expected: True, Got: True → PASS
TC4: is_number_palindrome(-121) → Expected: False, Got: False → PASS
TC5: is_number_palindrome(1221) → Expected: True, Got: True → PASS
TC6: is_number_palindrome(12321) → Expected: True, Got: True → PASS
TC7: is_number_palindrome(10) → Expected: False, Got: False → PASS
TC8: is_number_palindrome(1) → Expected: True, Got: True → PASS
TC9: is_number_palindrome(1001) → Expected: True, Got: True → PASS
TC10: is_number_palindrome(-1) → Expected: False, Got: False → PASS
PS D:\praneeth\ai code 8> 
```

OBSERVATIONS:

1. **Negative numbers** are **not considered palindromes** because the minus sign (-) is not mirrored.
2. The function **converts the number to a string**, reverses it, and compares it with the original string.
3. If the reversed string matches the original, the function returns True; otherwise, it returns False.

A list of test cases is created in the form of tuples: (input, expected_output). The function loops through each test case:

- Calls the palindrome-checking function with the test input.
- Compares the returned result with the expected value.
- Prints whether the test case passed or failed along with relevant information.