

## **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:**

The screenshot shows the Visual Studio Code editor with a Python file named `task1.py`. The code defines a function `is_prime(n)` that checks if a number is prime. It includes comments and test cases. The test cases are: `(2, True), (3, True), (5, True), (11, True), (0, False), (1, False), (-7, False), (4, False), (9, False), (7919, True), (8000, False)`. The function returns `True` if no divisors are found, and `False` otherwise. The test function `test_is_prime()` runs the test cases and prints the results. The output of the test function is: `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!

```
1 def is_prime(n):
2     """
3     Check if a number is prime.
4     Only integers greater than 1 can be prime.
5     """
6     if not isinstance(n, int) or n <= 1:
7         return False
8     if n == 2:
9         return True
10    if n % 2 == 0:
11        return False
12
13    # Check odd divisors up to sqrt(n)
14    for i in range(3, int(n ** 0.5) + 1, 2):
15        if n % i == 0:
16            return False
17    return True # Needed to return True if no divisors found
18
19
20 # Test cases for is_prime(n)
21 def test_is_prime():
22     tests = [
23         (2, True), (3, True), (5, True), (11, True),
24         (0, False), (1, False), (-7, False), (4, False),
25         (9, False), (7919, True), (8000, False)
26     ]
27     for n, expected in tests:
28         result = is_prime(n)
29         print(f"is_prime({n}) = {result}, expected = {expected}")
30         assert result == expected
31
32 # Run the test function
33 test_is_prime()
34 print("All test cases passed!")
35
36 return True # Needed to return True if no divisors found
```

The screenshot shows the Visual Studio Code editor with the same Python file `task1.py`. The terminal window is open, showing the output of the test function. The output is: `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!

```
\\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:

```
def is_prime(n):  
    if n < 2:  
        return False  
    for i in range(2, int(n**0.5) + 1):  
        if n % i == 0:  
            return False  
    return True  
  
# Test cases  
test_cases = [(2, True), (3, True), (5, True), (11, True), (0, False), (1, False), (-7, False), (4, False), (9, False), (7919, True), (8000, False)]  
all_passed = True  
for n, expected in test_cases:  
    result = is_prime(n)  
    if result != expected:  
        print(f'is_prime({n}) = {result}, expected = {expected}')  
        all_passed = False  
    else:  
        print(f'is_prime({n}) = {result}, expected = {expected}')  
if all_passed:  
    print('All test cases passed!')
```

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  $\sqrt{n}$** , 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:  $0^{\circ}\text{C} = 32^{\circ}\text{F}$ ,  $100^{\circ}\text{C} = 212^{\circ}\text{F}$ .
- Include decimals and invalid inputs like strings or None

CODE GENERATED:

```
1 def fixed_sort(lst):
2     """
3     Sort a list containing both numbers and strings
4     by converting all elements to strings.
5     """
6     return sorted(lst, key=lambda x: str(x))
7
8 # Example usage of fixed_sort
9 data = [3, "2", 1, "4"]
10 print("Sorted data:", fixed_sort(data)) # Output: ['1', '2', '3', '4']
11
12
13 def celsius_to_fahrenheit(c):
14     """
15     Convert Celsius to Fahrenheit.
16     Formula: F = C * 9/5 + 32
17     """
18     if not isinstance(c, (int, float)):
19         return None
20     return round(c * 9 / 5 + 32, 2)
21
22
23 def fahrenheit_to_celsius(f):
24     """
25     Convert Fahrenheit to Celsius.
26     Formula: C = (F - 32) * 5/9
27     """
28     if not isinstance(f, (int, float)):
29         return None
30     return round((f - 32) * 5 / 9, 2)
31
32
33 # AI-generated test cases
```

TC12: fahrenheit\_to\_celsius(None) = None | Expected: None | PASS  
PS D:\praneeth\ai code 8>

```
31
32
33 # AI-generated test cases
34 test_cases = [
35     # Celsius to Fahrenheit
36     ("C+F", 0, 32.0), # Known pair
37     ("C+F", 100, 212.0), # Known pair
38     ("C+F", 37.5, 99.5), # Decimal input
39     ("C+F", -40, -40.0), # Negative temperature
40     ("C+F", "abc", None), # Invalid string input
41     ("C+F", None, None), # None input
42
43     # Fahrenheit to Celsius
44     ("F+C", 32, 0.0), # Known pair
45     ("F+C", 212, 100.0), # Known pair
46     ("F+C", 99.5, 37.5), # Decimal input
47     ("F+C", -40, -40.0), # Negative temperature
48     ("F+C", "xyz", None), # Invalid string input
49     ("F+C", None, None), # None input
50 ]
51
52 # Run test cases
53 for i, (mode, input_val, expected) in enumerate(test_cases, 1):
54     if mode == "C+F":
55         result = celsius_to_fahrenheit(input_val)
56         func_name = "celsius_to_fahrenheit"
57     else:
58         result = fahrenheit_to_celsius(input_val)
59         func_name = "fahrenheit_to_celsius"
60
61     status = "PASS" if result == expected else "FAIL"
62     print(f"TC{i}: {func_name}({input_val}) = {result} | Expected: {expected} | {status}")
63
64
65 TC12: fahrenheit_to_celsius(None) = None | Expected: None | PASS
66 PS D:\praneeth\ai code 8>
```

OUTPUT OF THE CODE:

```
TC12: fahrenheit_to_celsius(None) = None | Expected: None | PASS
PS D:\praneeth\ai code > ^C
PS D:\praneeth\ai code > d;; cd 'd:\praneeth\ai code 8'; & 'c:\Users\SATHISH\AppData\Local\Microsoft\WindowsApps\python3.13.exe' 'c:\U
ser\SATHISH\.vscode\extensions\ms-python.debugpy-2025.10.0-win32-x64\bundle\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 > []
```

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

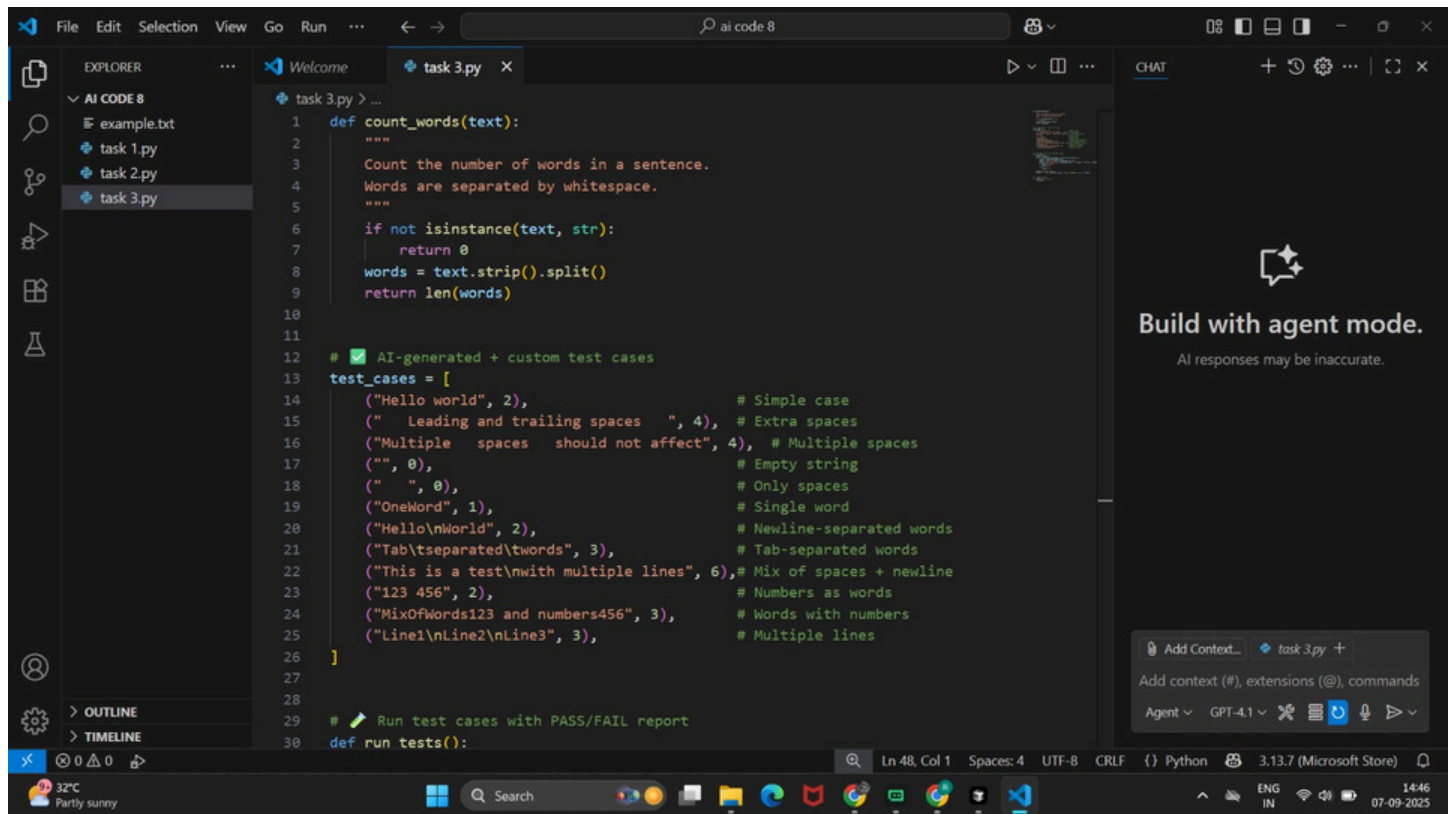
- Use AI to write test cases for a function count\_words(text) that returns the number of words in sentences

Requirements

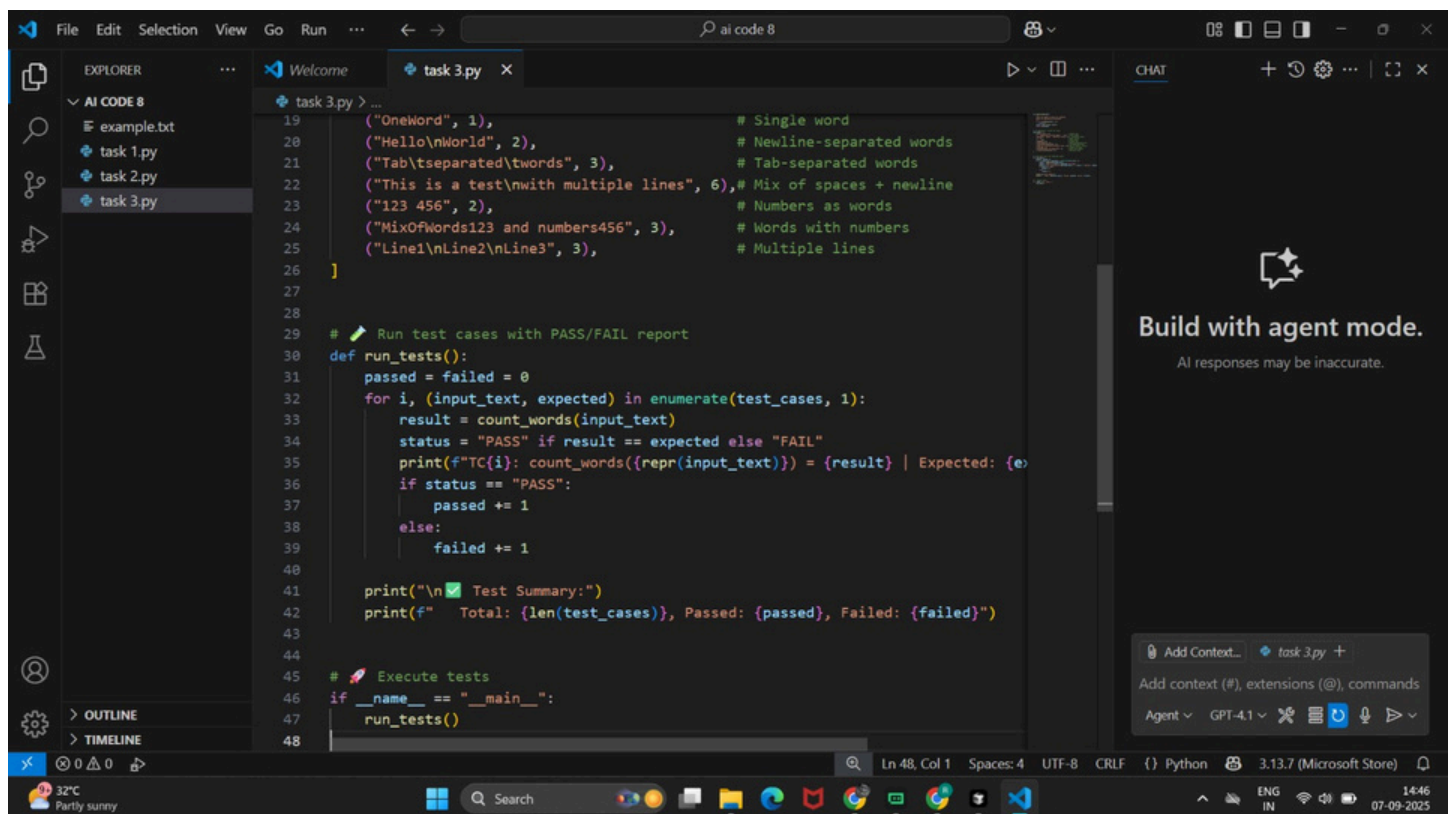
Handle normal text, multiple spaces, punctuation, and empty strings.

CODE GENERATED:





```
1 def count_words(text):
2     """
3     Count the number of words in a sentence.
4     Words are separated by whitespace.
5     """
6     if not isinstance(text, str):
7         return 0
8     words = text.strip().split()
9     return len(words)
10
11
12 # AI-generated + custom test cases
13 test_cases = [
14     ("Hello world", 2), # Simple case
15     (" Leading and trailing spaces ", 4), # Extra spaces
16     ("Multiple spaces should not affect", 4), # Multiple spaces
17     ("", 0), # Empty string
18     (" ", 0), # Only spaces
19     ("OneWord", 1), # Single word
20     ("Hello\nWorld", 2), # Newline-separated words
21     ("Tab\tseparated\twords", 3), # Tab-separated words
22     ("This is a test\nwith multiple lines", 6), # Mix of spaces + newline
23     ("123 456", 2), # Numbers as words
24     ("MixOfWords123 and numbers456", 3), # Words with numbers
25     ("Line1\nLine2\nLine3", 3), # Multiple lines
26 ]
27
28
29 # Run test cases with PASS/FAIL report
30 def run_tests():
```



```
19     ("OneWord", 1), # Single word
20     ("Hello\nWorld", 2), # Newline-separated words
21     ("Tab\tseparated\twords", 3), # Tab-separated words
22     ("This is a test\nwith multiple lines", 6), # Mix of spaces + newline
23     ("123 456", 2), # Numbers as words
24     ("MixOfWords123 and numbers456", 3), # Words with numbers
25     ("Line1\nLine2\nLine3", 3), # Multiple lines
26 ]
27
28
29 # Run test cases with PASS/FAIL report
30 def run_tests():
31     passed = failed = 0
32     for i, (input_text, expected) in enumerate(test_cases, 1):
33         result = count_words(input_text)
34         status = "PASS" if result == expected else "FAIL"
35         print(f"TC{i}: count_words({repr(input_text)}) = {result} | Expected: {e}")
36         if status == "PASS":
37             passed += 1
38         else:
39             failed += 1
40
41     print("\n✅ Test Summary:")
42     print(f"    Total: {len(test_cases)}, Passed: {passed}, Failed: {failed}")
43
44
45 # Execute tests
46 if __name__ == "__main__":
47     run_tests()
48
```

OUTPUT OF THE CODE:

```
task 3.py > ...
20 ("Hello\nWorld", 2),           # Newline-separated words
21 ("Tab\tseparated\twords", 3),   # Tab-separated words
22 ("This is a test\nwith multiple lines", 6), # Mix of spaces + newline
23 ("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()

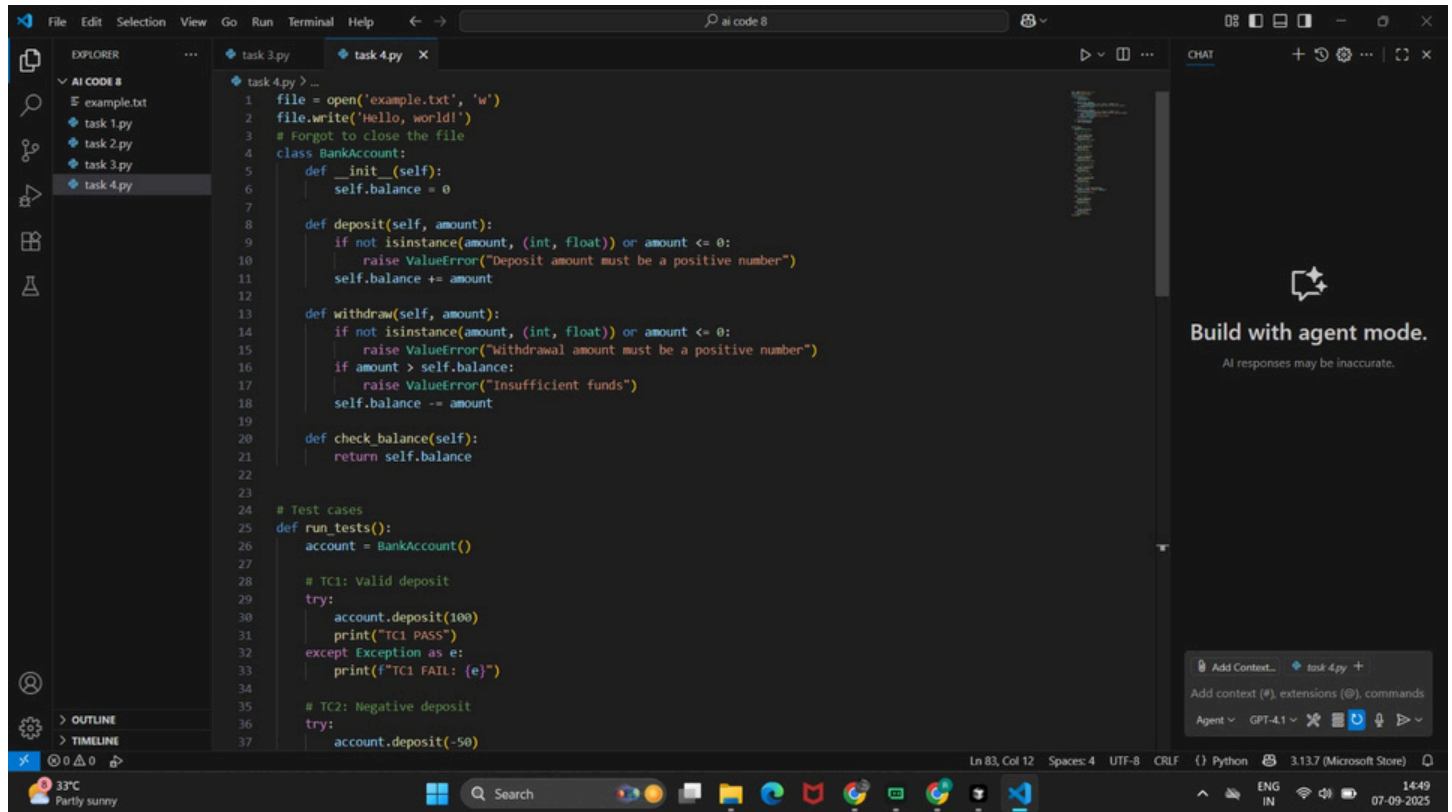
## Requirements:

- Negative deposits/withdrawals should raise an error.
- Cannot withdraw more than balance.

Expected Output#4

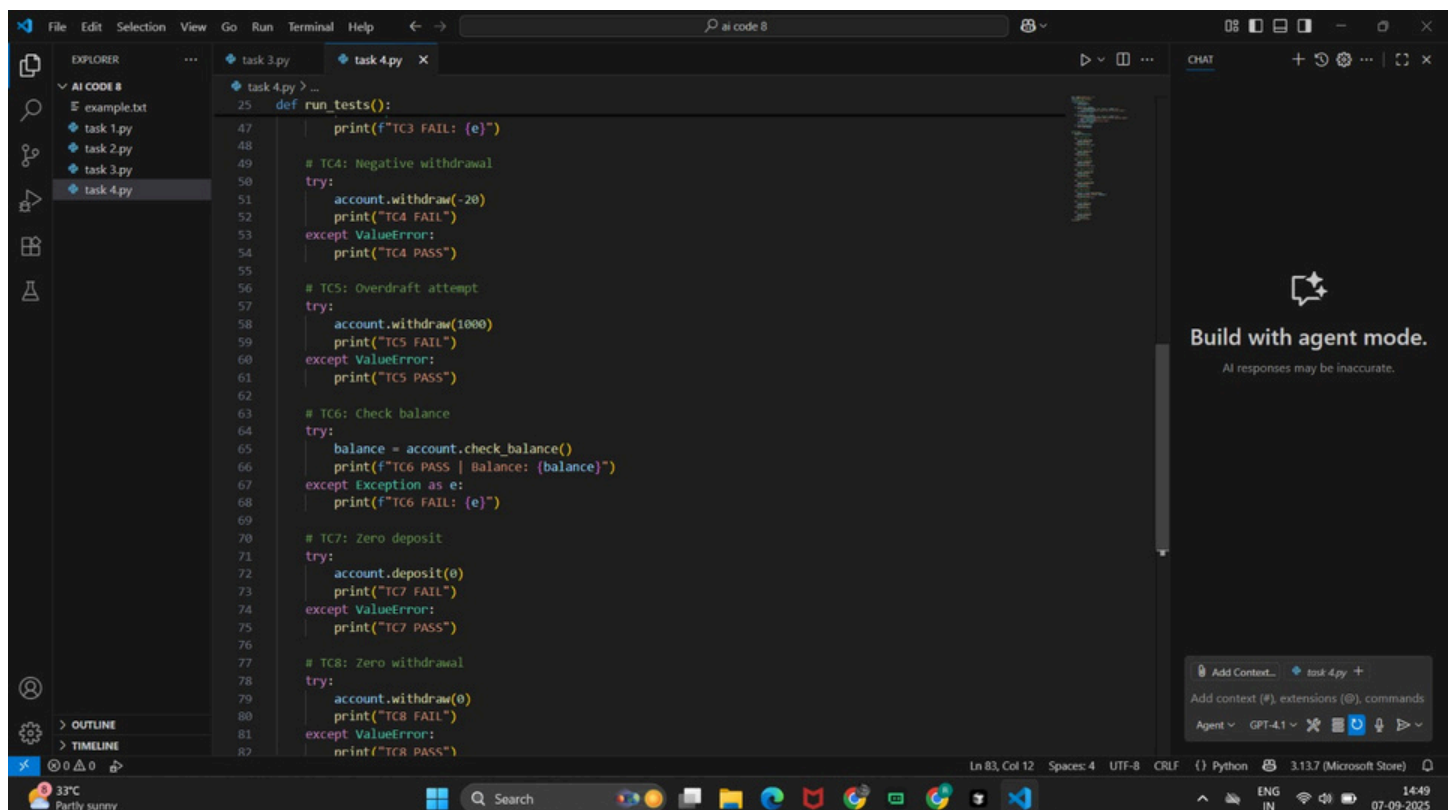
- AI-generated test suite with a robust class that handles all test cases.

## CODE GENERATED:



The screenshot shows the VS Code editor with a file named `task 4.py` open. The code defines a `BankAccount` class with methods `__init__`, `deposit`, `withdraw`, and `check_balance`. It also includes a `run_tests` function that tests the class with various inputs. The test cases include valid deposits, negative deposits, and valid withdrawals.

```
1 file = open('example.txt', 'w')
2 file.write('Hello, world!')
3 # Forgot to close the file
4 class BankAccount:
5     def __init__(self):
6         self.balance = 0
7
8     def deposit(self, amount):
9         if not isinstance(amount, (int, float)) or amount <= 0:
10             raise ValueError("Deposit amount must be a positive number")
11         self.balance += amount
12
13     def withdraw(self, amount):
14         if not isinstance(amount, (int, float)) or amount <= 0:
15             raise ValueError("Withdrawal amount must be a positive number")
16         if amount > self.balance:
17             raise ValueError("Insufficient funds")
18         self.balance -= amount
19
20     def check_balance(self):
21         return self.balance
22
23
24 # Test cases
25 def run_tests():
26     account = BankAccount()
27
28     # TC1: Valid deposit
29     try:
30         account.deposit(100)
31         print("TC1 PASS")
32     except Exception as e:
33         print(f"TC1 FAIL: {e}")
34
35     # TC2: Negative deposit
36     try:
37         account.deposit(-50)
```



The screenshot shows the continuation of the `run_tests` function in `task 4.py`. It includes test cases for negative withdrawals, overdraft attempts, check balance, zero deposits, and zero withdrawals.

```
47     print(f"TC3 FAIL: {e}")
48
49     # TC4: Negative withdrawal
50     try:
51         account.withdraw(-20)
52         print("TC4 FAIL")
53     except ValueError:
54         print("TC4 PASS")
55
56     # TC5: Overdraft attempt
57     try:
58         account.withdraw(1000)
59         print("TC5 FAIL")
60     except ValueError:
61         print("TC5 PASS")
62
63     # TC6: Check balance
64     try:
65         balance = account.check_balance()
66         print(f"TC6 PASS | Balance: {balance}")
67     except Exception as e:
68         print(f"TC6 FAIL: {e}")
69
70     # TC7: Zero deposit
71     try:
72         account.deposit(0)
73         print("TC7 FAIL")
74     except ValueError:
75         print("TC7 PASS")
76
77     # TC8: Zero withdrawal
78     try:
79         account.withdraw(0)
80         print("TC8 FAIL")
81     except ValueError:
82         print("TC8 PASS")
```



The screenshot shows the VS Code editor with a file named `task 4.py` open. The code defines a `run_tests()` function that tests various scenarios for a bank account. The terminal window shows the output of the tests, indicating that all tests passed (TC1 PASS through TC8 PASS) and the final balance is 70. The status bar at the bottom indicates the file is Python 3.13.7.

```
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}")
```

Terminal Output:

```
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\bundle\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:

This screenshot is a close-up of the terminal window from the previous image, showing the same output: TC1 PASS through TC8 PASS and the final balance of 70.

## 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.

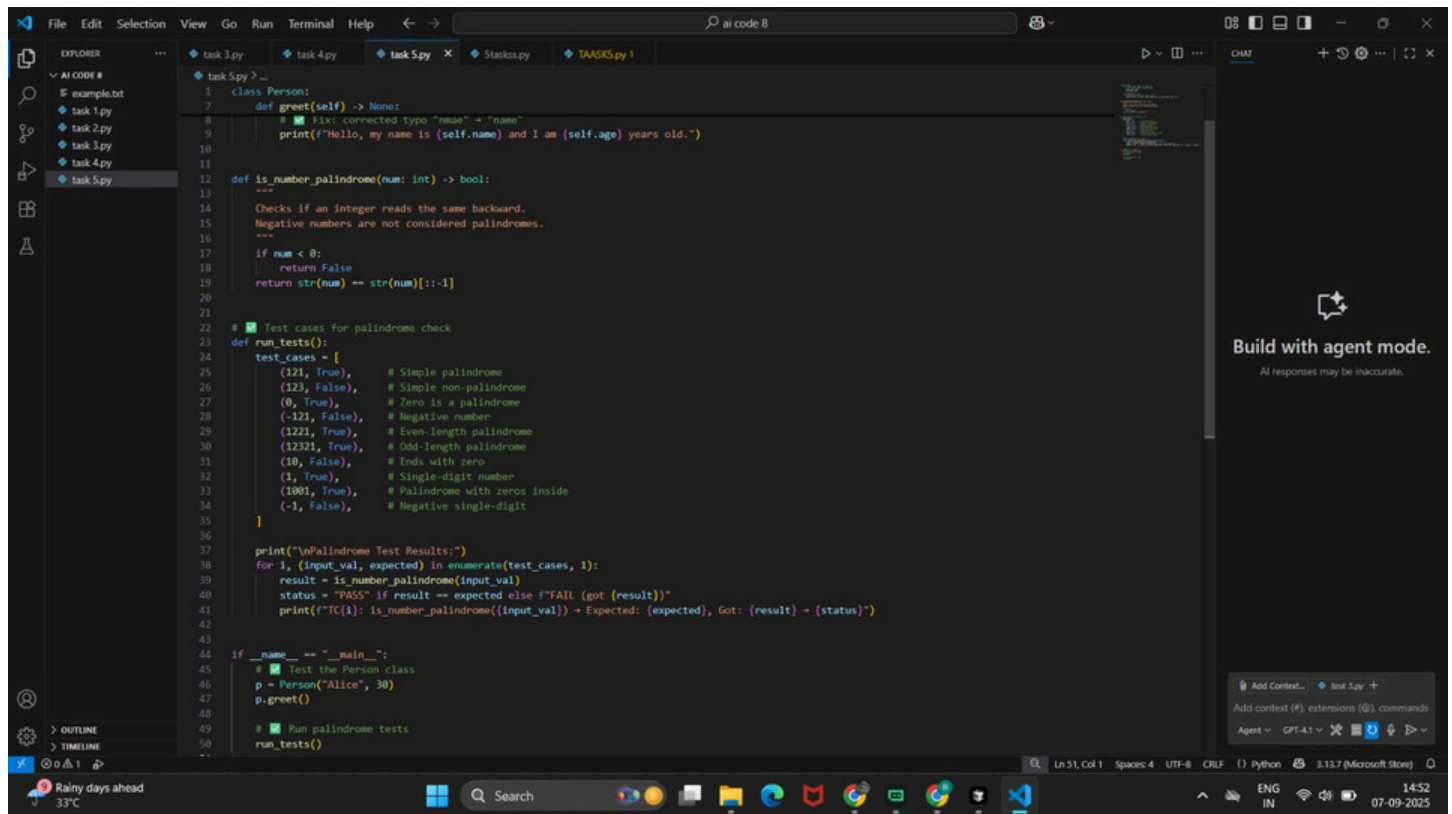
#### Examples:

121 → True

123 → False

0, negative numbers → handled

### CODE GENERATED:



```
1 class Person:
2     def greet(self) -> None:
3         # Fix: corrected type "name" -> "name"
4         print(f"Hello, my name is {self.name} and I am {self.age} years old.")
5
6
7
8
9
10
11
12 def is_number_palindrome(num: int) -> bool:
13     """
14     Checks if an integer reads the same backward.
15     Negative numbers are not considered palindromes.
16     """
17     if num < 0:
18         return False
19     return str(num) == str(num)[::-1]
20
21
22 # Test cases for palindrome check
23 def run_tests():
24     test_cases = [
25         (121, True), # Simple palindrome
26         (123, False), # Simple non-palindrome
27         (0, True), # Zero is a palindrome
28         (-121, False), # Negative number
29         (1221, True), # Even-length palindrome
30         (12321, True), # Odd-length palindrome
31         (10, False), # Ends with zero
32         (1, True), # Single-digit number
33         (1001, True), # Palindrome with zeros inside
34         (-1, False), # Negative single-digit
35     ]
36
37     print("\nPalindrome Test Results:")
38     for i, (input_val, expected) in enumerate(test_cases, 1):
39         result = is_number_palindrome(input_val)
40         status = "PASS" if result == expected else f"FAIL (got {result})"
41         print(f"TC{i}: is_number_palindrome({input_val}) -> Expected: {expected}, Got: {result} - {status}")
42
43
44 if __name__ == "__main__":
45     # Test the Person class
46     p = Person("Alice", 30)
47     p.greet()
48
49     # Run palindrome tests
50     run_tests()
```

The screenshot shows a VS Code editor with a Python file named `task 5.py`. The code defines a `Person` class and a `is_number_palindrome` function. The `Person` class has an `__init__` method that takes `name` and `age` as arguments and a `greet` method that prints a message. The `is_number_palindrome` function checks if a number is a palindrome. The terminal output shows the execution of the code, including the greeting message and the results of the palindrome tests.

```
1 class Person:
2     def __init__(self, name: str, age: int):
3         # Fix: use correct variable names
4         self.name = name
5         self.age = age
6
7     def greet(self) -> None:
8         # Fix: corrected typo "nmae" -> "name"
9         print(f"Hello, my name is {self.name} and I am {self.age} years old.")
10
11
12 def is_number_palindrome(num: int) -> bool:
13     """
14     Checks if an integer reads the same backward.
15     Negative numbers are not considered palindromes.
16     """
17     if num < 0:
```

Terminal Output:

```
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 >
```

## OUTPUT OF THE CODE:

This screenshot shows the terminal output from the previous code execution. It displays the greeting message and the results of the palindrome tests, confirming that the function works correctly for various inputs, including negative numbers and palindromes.

```
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 >
```

## 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.