

SR UNIVERSITY

AI ASSIST CODING

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

ROLL NO:2503A51L11

NAME:P.Susmija

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 VS Code interface with the following details:

- EXPLORER**: Shows "NO FOLDER OPENED". A tooltip says: "You have not yet opened a folder. Opening a folder will close all currently open editors. To keep them open, add a folder instead." An "Open Folder" button is present.
- EDITOR**: The code editor has the following content:


```
1 def is_prime(n):
2     """
3         Check if a number is prime.
4         Only integers greater than 1 can be prime.
5     """
6
7     if n <= 1:
8         return False
9     if n == 2:
10        return True
11     if n % 2 == 0:
12         return False
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
18 # AI-generated test cases
19 test_cases = [
20     (-7, False),    # Negative number
21     (0, False),     # Zero
22     (1, False),     # One
23     (2, True),      # Smallest prime
24     (3, True),      # Prime
25     (4, False),     # Even non-prime
26     (17, True),     # Prime
27     ...]
```
- CHAT**: A sidebar with the following text:

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
- TERMINAL**: The terminal shows the command "Ask GPT-4.1". Below it, a message says "Used 1 reference".
- OUTPUT**: A panel showing some test cases and their results.
- PROBLEMS**: A panel showing 8 problems.

The screenshot shows the VS Code interface with the following details:

- EXPLORER**: Shows "NO FOLDER OPENED". A tooltip says: "You have not yet opened a folder. Opening a folder will close all currently open editors. To keep them open, add a folder instead." An "Open Folder" button is present.
- EDITOR**: The code editor has the following content:


```
1 def is_prime(n):
2     return True
3 # AI-generated test cases
4 test_cases = [
5     (-7, False),    # Negative number
6     (0, False),     # Zero
7     (1, False),     # One
8     (2, True),      # Smallest prime
9     (3, True),      # Prime
10    (4, False),     # Even non-prime
11    (17, True),     # Prime
12    ...
13    # Run test cases
14    for i, (input_val, expected) in enumerate(test_cases, 1):
15        result = is_prime(input_val)
16        status = "PASS" if result == expected else "FAIL"
17        print(f"TC{i}: is_prime({input_val}) = {result} | Expected: {expected} | Status: {status}")
18
19 TC12: is_prime(8000) = False | Expected: False | PASS
20 PS C:\Users\Susmija>
```
- CHAT**: A sidebar with the following text:

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
- TERMINAL**: The terminal shows the command "Ask GPT-4.1". Below it, a message says "Used 1 reference".
- OUTPUT**: A panel showing some test cases and their results.
- PROBLEMS**: A panel showing 8 problems.

OUTPUT OF THE CODE:

PROBLEMS 8 OUTPUT DEBUG CONSOLE TERMINAL ...

TC12: is_prime(8000) = False | Expected: False | PASS

● PS C:\Users\Susmija> & C:/Users/Susmija/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/Susmija/Desktop/AI/task-1 cop.py"

TC1: is_prime(-7) = False | Expected: False | PASS

TC2: is_prime(0) = False | Expected: False | PASS

TC3: is_prime(1) = False | Expected: False | PASS

TC4: is_prime(2) = True | Expected: True | PASS

TC5: is_prime(3) = True | Expected: True | PASS

TC6: is_prime(4) = False | Expected: False | PASS

TC7: is_prime(17) = True | Expected: True | PASS

TC8: is_prime(18) = False | Expected: False | PASS

TC9: is_prime(97) = True | Expected: True | PASS

TC10: is_prime(100) = False | Expected: False | PASS

TC11: is_prime(7919) = True | Expected: True | PASS

TC12: is_prime(8000) = False | Expected: False | PASS

○ PS C:\Users\Susmija>

OBSERVATIONS:

Purpose of the Function

- 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:

The screenshot shows the Visual Studio Code interface with the following details:

- File Explorer:** Shows a folder named "NO FOLDER OPENED". A tooltip indicates: "Opening a folder will close all currently open editors. To keep them open, add a folder instead."
- Code Editor:** Displays Python code for temperature conversion. It includes functions for Celsius-to-Fahrenheit and Fahrenheit-to-Celsius conversions, along with AI-generated test cases. A specific comment "# AI-generated test cases" is present.
- Chat Panel:** A GPT-4.1 AI interface is shown, with the instruction: "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, negative numbers, and large primes".
- Bottom Status Bar:** Shows file paths (task-1.py, task-1 cop.py, 2task.py), line numbers (Ln 29, Col 48), and other system information.

The screenshot shows the Visual Studio Code interface with the following details:

- File Explorer:** Shows a folder named "NO FOLDER OPENED". A tooltip indicates: "Opening a folder will close all currently open editors. To keep them open, add a folder instead."
- Code Editor:** Displays the completed Python code. It includes the conversion functions and the AI-generated test cases from the previous screenshot, plus a new section at the bottom for running tests. The test section uses the `enumerate` function to iterate through the test cases and prints results.
- Chat Panel:** The GPT-4.1 AI interface remains active, providing the same requirements and edge cases as before.
- Bottom Status Bar:** Shows file paths (task-1.py, task-1 cop.py, 2task.py), line numbers (Ln 25, Col 45), and other system information.

OUTPUT OF THE CODE:

The screenshot shows a code editor interface with several tabs open. The tabs include 'task-1.py', 'task-1 cop.py', '2task.py', and 'edit.js'. The '2task.py' tab is active, displaying a Python function 'celsius_to_fahrenheit(c)' that converts Celsius to Fahrenheit using the formula $F = C * 9/5 + 32$. Below the function is a series of test cases (TC1 through TC12) comparing the function's output against expected values. The 'edit.js' tab shows some AI-generated JavaScript code for testing prime numbers. The left sidebar has sections for 'EXPLORER', 'PROBLEMS', 'OUTPUT', 'DEBUG CONSOLE', 'TERMINAL', 'PORTS', and 'CHART'. The bottom status bar shows file paths like 'C:\Users\Susmija> & C:/Users/Susmija/AppData/Local/Programs/Python/Python313/python.exe c:/Users/Susmija/Desktop/AI/2task.py', line and column counts ('Ln 29, Col 48'), and other system information.

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., $0^{\circ}\text{C} \rightarrow 32^{\circ}\text{F}$, $212^{\circ}\text{F} \rightarrow 100^{\circ}\text{C}$)
 - 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:

The screenshot shows the Visual Studio Code (VS Code) interface. The left sidebar has icons for file operations, search, and folder navigation. The main area displays a Python script named `3task.py`. The code defines a function `count_words` to count words in a sentence, handles edge cases like empty strings and single spaces, and includes a list of AI-generated test cases. A sidebar panel titled "CHAT" contains a prompt for generating test cases for a prime number function and some sample test cases. The status bar at the bottom shows "Spaces: 4" and "UTF-8".

```
C:/> Users > Susmija > Desktop > AI > 3task.py > ...
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 test cases
13 test_cases = [
14     ("Hello world", 2),                                     # Normal te
15     (" Leading and trailing spaces ", 4),                  # Extra spa
16     ("Multiple    spaces between words", 4),               # Multiple
17     ("Punctuation, should not: affect!", 5),              # Punctuati
18     ("", 0),                                              # Empty str
19     (" ", 0),                                             # Only spac
20     ("One", 1),                                           # Single wo
21     ("New\nline\tand\ttabs", 4),                            # Newlines
22     ("Mix of punctuation, spaces, and\nnewlines", 7),      # Complex f
23     ("123 456", 2)                                         # Numbers a
24 ]
25
26 # 🧪 Run test cases
27 for i, (input_val, expected) in enumerate(test_cases, 1):
28     result = count_words(input_val)
29     status = "PASS" if result == expected else "FAIL"
30     print(f"TC{i}: count_words({repr(input_val)}) = {result} | Expe
```

This screenshot shows the same VS Code environment as the first one, but the terminal output has been cleared or changed. The code and sidebar are identical to the first screenshot.

OUTPUT OF THE CODE:

```

def count_words(text):
    words = text.strip().split()
    return len(words)

```

PS C:\Users\Susmija> & C:/Users/Susmija/AppData/Local/Programs/Python/Python313/python.exe c:/Users/Susmija/Desktop/AI/3task.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 between words') = 4 | Expected: 4 | PASS
- TC4: count_words('Punctuation, should not: affect!') = 4 | Expected: 5 | FAIL
- TC5: count_words('') = 0 | Expected: 0 | PASS
- TC6: count_words(' ') = 0 | Expected: 0 | PASS
- TC7: count_words('One') = 1 | Expected: 1 | PASS
- TC8: count_words('New\nline\tand\ttabs') = 4 | Expected: 4 | PASS
- TC9: count_words('Mix of punctuation, spaces, and\nnewlines') = 6 | Expected: 7 | FAIL
- TC10: count_words('123 456') = 2 | Expected: 2 | PASS

PS C:\Users\Susmija>

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:

```
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)  
        print("TC2 FAIL")  
    except ValueError:  
        print("TC2 PASS")  
  
    # TC3: Valid withdrawal  
    try:  
        account.withdraw(30)  
        print("TC3 PASS")  
    except Exception as e:  
        print(f"TC3 FAIL: {e}")  
  
    # TC4: Negative withdrawal  
    try:  
        account.withdraw(-20)  
        print("TC4 FAIL")  
    except ValueError:  
        print("TC4 PASS")
```

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

Used 1 reference

Here are some test cases for `is_prime(n)`, followed by an implementation:

```
for is_prime(n)  
prime():  
    s_prime(2) == True      # Smallest prime  
    s_prime(3) == True      # Next prime  
    s_prime(5) == True      # Odd prime  
    s_prime(11) == True     # Prime  
    s_prime(8) == False     # Edge: zero  
    s_prime(1) == False     # Edge: one  
    s_prime(-7) == False    # Negative number  
    s_prime(4) == False     # Even non-prime  
    ...  
    ...
```

Add Context... 4taskss.py X
Add context (#), extensions (@), commands (/)
Ask v GPT-4.1 v

```
def run_tests():  
    # 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)  
        print("TC2 FAIL")  
    except ValueError:  
        print("TC2 PASS")  
  
    # TC3: Valid withdrawal  
    try:  
        account.withdraw(30)  
        print("TC3 PASS")  
    except Exception as e:  
        print(f"TC3 FAIL: {e}")  
  
    # TC4: Negative withdrawal  
    try:  
        account.withdraw(-20)  
        print("TC4 FAIL")  
    except ValueError:  
        print("TC4 PASS")
```

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

Used 1 reference

Here are some test cases for `is_prime(n)`, followed by an implementation:

```
for is_prime(n)  
prime():  
    s_prime(2) == True      # Smallest prime  
    s_prime(3) == True      # Next prime  
    s_prime(5) == True      # Odd prime  
    s_prime(11) == True     # Prime  
    s_prime(8) == False     # Edge: zero  
    s_prime(1) == False     # Edge: one  
    s_prime(-7) == False    # Negative number  
    s_prime(4) == False     # Even non-prime  
    ...  
    ...
```

Add Context... 4taskss.py X
Add context (#), extensions (@), commands (/)
Ask v GPT-4.1 v

The screenshot shows the Visual Studio Code interface with a dark theme. In the center, there is an editor window displaying Python code. The code defines a `run_tests()` function with several test cases (TC4, TC5, TC6, TC7) for an account class. A code completion tooltip is open over the `is_prime` call in the `TC8` test case, suggesting an implementation of the `is_prime` function. The tooltip contains the following code:

```
. for is_prime(n)
prime():
s_prime(2) == True      # Smallest prime
s_prime(3) == True      # Next prime
s_prime(5) == True      # Odd prime
s_prime(11) == True     # Prime
s_prime(0) == False     # Edge: zero
s_prime(1) == False     # Edge: one
s_prime(-7) == False    # Negative number
s_prime(4) == False     # Even non-prime
s_prime(13) == True     # Another prime
```

The status bar at the bottom indicates the code is in Python mode, version 3.13.7, and the date is 02-09-2025.

This screenshot is similar to the first one but shows more test cases added to the `run_tests()` function. The new test cases include TC5, TC6, TC7, and TC8. The code completion tooltip for `is_prime` now includes all these test cases. The status bar at the bottom indicates the code is in Python mode, version 3.13.7, and the date is 02-09-2025.

OUTPUT OF THE CODE:

The screenshot shows a terminal window with the following text:

```
PS C:\Users\Susmija> & C:/Users/Susmija/AppData/Local/Programs/Python/Python313/python.exe c:/Users/Susmija/Desktop/AI/4taskss.py
● TC1 PASS
TC2 PASS
TC3 PASS
TC4 PASS
TC5 PASS
TC6 PASS | Balance: 70
TC7 PASS
TC8 PASS
○ PS C:\Users\Susmija>
```

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:

```
C:\Users\Susmija\Desktop\AI> 5taskss.py > run_tests
1  def is_number_palindrome(num):
2      """
3          Checks if an integer reads the same backward.
4          Negative numbers are not considered palindromes.
5      """
6
7      if num < 0:
8          return False
9      return str(num) == str(num)[::-1]
# Test cases
10 def run_tests():
11     test_cases = [
12         (121, True),           # Simple palindrome
13         (123, False),          # Simple non-palindrome
14         (0, True),             # Zero is a palindrome
15         (-121, False),         # Negative number
16         (1221, True),          # Even-length palindrome
17         (12321, True),         # Odd-length palindrome
18         (10, False),            # Ends with zero
19         (1, True),              # Single-digit number
20         (1001, True),           # Palindrome with zeros inside
21         (-1, False),            # Negative single-digit
22     ]
23
24     for i, (input_val, expected) in enumerate(test_cases, 1):
25         result = is_number_palindrome(input_val)
26         status = "PASS" if result == expected else f"FAIL (got {res
27         print(f"TC{i}: is_number_palindrome({input_val}) → {expecte
28 run_tests()
```

OUTPUT OF THE 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.

► Function Logic:

- If the number is negative (`num < 0`), the function immediately returns False.
- Otherwise, the number is converted to a string, and the string is reversed using Python slicing (`[::-1]`).
- The original and reversed strings are compared:
 - If they are equal → return True (number is a palindrome).
 - If not → return 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.