

## << AI ASSISTED CODING

### Task Description#1

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.

### Expected Output#1

- A working prime checker that passes AI-generated tests using edge coverage.

Prompt: Create a set of test cases with edge coverage for a Python function `is_prime(n)` that identifies prime numbers (integers greater than 1). Include tests for special cases like 0, 1, 2, negative numbers, and very large prime numbers. After generating the test cases, write the Python code for the `is_prime(n)` function and then execute the tests to verify the implementation."

### Code:

---

```
def is_prime(n):  
    """  
    Checks if a number is prime.  
  
    Args:  
        n: An integer.  
  
    Returns:  
        True if n is a prime number, False otherwise.  
    """  
    if not isinstance(n, int):  
        return False  
    if n <= 1:  
        return False  
    if n == 2:  
        return True  
    if n % 2 == 0:  
        return False  
    i = 3  
    while i * i <= n:  
        if n % i == 0:  
            return False  
        i += 2  
    return True
```



```
test_cases = [  
    (0, False),  
    (1, False),  
    (2, True),  
    (-1, False),  
    (-5, False),  
    (-10, False),  
    (3, True),  
    (5, True),  
    (7, True),  
    (11, True),  
    (4, False),  
    (6, False),  
    (8, False),  
    (9, False),  
    (10, False),  
    (999983, True),  
    (1000000, False),  
    (2.5, False),  
    ("abc", False),  
]
```


```
▶ print("Running tests...")
for number, expected_result in test_cases:
    actual_result = is_prime(number)
    if actual_result == expected_result:
        print(f"input {number}: Expected {expected_result}, Got {actual_result}")
    else:
        print(f"input {number}: Expected {expected_result}, Got {actual_result}")

print("\nNow you can enter a number to check if it's prime.")
user_input_str = input("Enter an integer to check if it's prime: ")

try:
    user_number = int(user_input_str)
    if is_prime(user_number):
        print(f"{user_number} is a prime number.")
    else:
        print(f"{user_number} is not a prime number.")
except ValueError:
    print("Invalid input. Please enter a valid integer.")
```

Output:

Running tests...



```
input 0: Expected False, Got False
input 1: Expected False, Got False
input 2: Expected True, Got True
input -1: Expected False, Got False
input -5: Expected False, Got False
input -10: Expected False, Got False
input 3: Expected True, Got True
input 5: Expected True, Got True
input 7: Expected True, Got True
input 11: Expected True, Got True
input 4: Expected False, Got False
input 6: Expected False, Got False
input 8: Expected False, Got False
input 9: Expected False, Got False
input 10: Expected False, Got False
input 999983: Expected True, Got True
input 1000000: Expected False, Got False
input 2.5: Expected False, Got False
input abc: Expected False, Got False
```

Now you can enter a number to check if it's prime.  
Enter an integer to check if it's prime: -5  
-5 is not a prime number.

#### Code Explanation:

- First, it checks if the number you give it is actually a whole number. If not, it immediately says "False" (it's not prime).
- If the number is 1 or less (like 0, 1, -5), it also says "False" because prime numbers have to be bigger than 1.
- If the number is exactly 2, it says "True" because 2 is the first prime number.
- If the number is bigger than 2 and is an even number (like 4, 6, 8), it says "False" because all even numbers bigger than 2 can be divided by 2.
- For all other numbers, it does a little check: it tries dividing the number by odd numbers (3, 5, 7, and so on) up to a certain point. If it finds any odd number that divides your number evenly, it means your number is not prime, and the helper says "False".
- If it tries dividing by all those odd numbers and doesn't find any that divide it evenly, then your number must be prime, and the helper says "True".

Observation:

- Functionality: The `is_prime` function appears to be correctly implemented and capable of identifying prime numbers based on the provided test cases.
- Test Coverage: The test cases cover a good range of scenarios, including edge cases (0, 1, 2, negative numbers), small primes and non-primes, a large prime, a large nonprime, and non-integer inputs. This provides good edge coverage.
- Test Results: All the provided test cases passed successfully, indicating that the current implementation of `is_prime` works as expected for these inputs.

Task Description#2 (Loops)

- 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

Prompt: Generate test cases for functions that convert Celsius to Fahrenheit and Fahrenheit to Celsius. Make sure the tests confirm that  $0^{\circ}\text{C}$  equals  $32^{\circ}\text{F}$  and  $100^{\circ}\text{C}$  equals  $212^{\circ}\text{F}$ . Also, include tests with decimal temperatures and inputs that are not valid numbers, like text or empty values."

Code:

```

▶ def celsius_to_fahrenheit(c):
    """Converts Celsius to Fahrenheit."""
    if not isinstance(c, (int, float)):
        return None
    return (c * 9/5) + 32

def fahrenheit_to_celsius(f):
    """Converts Fahrenheit to Celsius."""
    if not isinstance(f, (int, float)):
        return None
    return (f - 32) * 5/9

celsius_test_cases = [
    (0, 32.0),
    (100, 212.0),
    (25.5, 77.9),
    ("abc", None),
    (None, None)
]

```

```

▶ fahrenheit_test_cases = [
    (32, 0.0),
    (212, 100.0),
    (77.9, 25.5),
    ("xyz", None),
    (None, None)
]

print("\nNow you can enter a temperature to convert.")

while True:
    user_input_temp = input("Enter a temperature (e.g., 25C or 77F), or type 'quit' to exit: ")
    if user_input_temp.lower() == 'quit':
        break

    if len(user_input_temp) > 1:
        unit = user_input_temp[-1].upper()
        try:
            temperature = float(user_input_temp[:-1])
            if unit == 'C':
                converted_temp = celsius_to_fahrenheit(temperature)
                if converted_temp is not None:
                    print(f"{temperature}°C is {converted_temp}°F")
            else:

```



```
if len(user_input_temp) > 1:
    unit = user_input_temp[-1].upper()
    try:
        temperature = float(user_input_temp[:-1])
        if unit == 'C':
            converted_temp = celsius_to_fahrenheit(temperature)
            if converted_temp is not None:
                print(f"{temperature}°C is {converted_temp}°F")
            else:
                print("Invalid input temperature.")
        elif unit == 'F':
            converted_temp = fahrenheit_to_celsius(temperature)
            if converted_temp is not None:
                print(f"{temperature}°F is {converted_temp}°C")
            else:
                print("Invalid input temperature.")
        else:
            print("Invalid unit. Please use 'C' for Celsius or 'F' for Fahrenheit.")
    except ValueError:
        print("Invalid temperature value. Please enter a number followed by C or F.")
else:
    print("Invalid input format. Please enter a number followed by C or F (e.g., 25C or 77F).")
```

Output:



```
Now you can enter a temperature to convert.
Enter a temperature (e.g., 25C or 77F), or type 'quit' to exit: 100c
100.0°C is 212.0°F
Enter a temperature (e.g., 25C or 77F), or type 'quit' to exit: 212F
212.0°F is 100.0°C
Enter a temperature (e.g., 25C or 77F), or type 'quit' to exit: abc
Invalid temperature value. Please enter a number followed by C or F.
Enter a temperature (e.g., 25C or 77F), or type 'quit' to exit: 45F
45.0°F is 7.222222222222222°C
Enter a temperature (e.g., 25C or 77F), or type 'quit' to exit: exit
Invalid temperature value. Please enter a number followed by C or F.
Enter a temperature (e.g., 25C or 77F), or type 'quit' to exit: quit
```

Code Explanation:

- Two Helpers: The code has two small programs (called "func ons") that help convert temperatures: one for Celsius to Fahrenheit, and one for Fahrenheit to Celsius.
- They Check: These helpers first check if you give them a proper number. If not, they say "I can't do that" (they return None).
- Example Tests: The code has lists of example temperatures and what the converted temperatures should be. These are used to check if the helpers are working right.
- Checking Answers: The code runs the helpers with these examples and checks if the answers match the correct ones. It prints if each check passes or fails.



- Try It Yourself: There's a part where you can type in a temperature (like "25C" or "77F").
- Gets Your Input: The code reads what you type and tries to figure out the number and if it's Celsius or Fahrenheit.
- Converts for You: It uses the right helper to convert your temperature and tells you the answer.
- Handles Mistakes: If you type something wrong, it tells you there's a problem.
- Quit Any me: You can type "quit" to stop the program.

Observation:

1. The Tools: The code has the instructions for these two temperature-changing tools.
2. Practice Examples: It has some examples with known answers to make sure the tools are working correctly.
3. Checking the Tools: The code uses the practice examples to test the tools and sees if they give the right answers.
4. Letting You Use the Tools: It then lets you tell it a temperature and which tool to use (Celsius to Fahrenheit or Fahrenheit to Celsius), and it will give you the result.

Task Description#3

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

Requirement

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

Expected Output#3

Accurate word count with robust test case validation

Prompt: Generate robust test cases for a Python function `count_words(text)` that determines the number of words in a given string. The tests should specifically cover standard sentences, strings with extra spaces, text containing punctuation, and empty or blank inputs."

Code:

```

def count_words(text):
    """
    Counts the number of words in a string.

    Args:
        text: The input string.

    Returns:
        The number of words in the string, or 0 if the input is not a string or is empty/whitespace only.
    """
    if not isinstance(text, str):
        return 0
    text = text.strip()
    if not text: # Handle empty or whitespace-only strings
        return 0
    # Remove punctuation by replacing with spaces (optional, depending on definition of "word")
    # import string
    # text = text.translate(str.maketrans('', '', string.punctuation))
    words = text.split()
    return len(words)

```

```

test_cases = [
    ("This is a standard sentence.", 5),
    (" Sentence with spaces. ", 3),
    ("Sentence with extra spaces.", 4),
    ("Sentence, with punctuation!", 3), # Note: split() handles punctuation attached to words
    ("", 0),
    (" ", 0),
    (" Another, sentence with ! spaces. ", 5),
    (None, 0),
    (123, 0),
    ([], 0),
]

```

```

print("Running tests for count_words function...")
for text_input, expected_count in test_cases:
    actual_count = count_words(text_input)
    if actual_count == expected_count:
        print(f"input '{text_input}': Expected {expected_count}, Got {actual_count}")
    else:
        print(f"input '{text_input}': Expected {expected_count}, Got {actual_count}")

print("\nNow you can enter multiple lines of text to count words. Type 'quit' on a new line to finish.")

while True:
    try:
        line = input()
        if line.lower() == 'quit':
            break
        if line: # Only process non-empty lines (after 'quit' check)
            word_count = count_words(line)
            print(f"'{line}': {word_count} words")
    except EOFError: # Handle case where input stream is closed
        break

```

Output:

```

Running tests for count_words function...
input 'This is a standard sentence.': Expected 5, Got 5
input ' Sentence with spaces. ': Expected 3, Got 3
input 'Sentence with extra spaces.': Expected 4, Got 4
input 'Sentence, with punctuation!': Expected 3, Got 3
input '': Expected 0, Got 0
input ' ': Expected 0, Got 0
input ' Another, sentence with ! spaces. ': Expected 5, Got 5
input 'None': Expected 0, Got 0
input '123': Expected 0, Got 0
input '[]': Expected 0, Got 0

Now you can enter multiple lines of text to count words. Type 'quit' on a new line to finish.
Iam computer science student
'Iam computer science student': 4 words
bl ue pe n
'bl ue pe n': 4 words
take a nap
'take a nap': 3 words
quit

```

### Code Explanation:

- It first checks if what you gave it is text. If not, it says the word count is 0.
- It cleans up any extra spaces at the beginning or end of the text.
- If, after cleaning, there's no text left, it means there are no words, so it says the count is 0.
- Then, it splits the text into individual words based on where the spaces are.
- Finally, it counts how many words it found.
- Observation:
  - Core Logic: The count\_words function effectively uses the split() method to separate words based on spaces. This is a standard and generally efficient way to count words in many cases.
  - Handling Spaces: The function correctly handles multiple spaces between words and leading/trailing spaces by using .strip() and the default behavior of split().
  - Handling Empty/Blank Input: The code explicitly checks for empty strings and strings containing only whitespace after stripping, returning 0 for these cases as expected.
  - Handling Non-Strings: The function includes a check for non-string inputs and returns 0. The test cases show this works for None, integers, and lists.
  - Punctuation: The current implementation counts words with attached punctuation (like "Sentence," or "spaces.") as single words. This aligns with how split() works but might need adjustment depending on the desired definition of a "word".

- 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.

Prompt: Design test cases for a BankAccount class with methods for depositing, withdrawing, and checking the balance. Crucially, the tests should verify that attempting negative deposits or withdrawals raises an error, and that withdrawing an amount exceeding the current balance is not allowed."

Code:

```
import unittest
import sys

class BankAccount:
    def __init__(self, initial_balance=0):
        if initial_balance < 0:
            print("Warning: Initial balance cannot be negative. Setting to 0.")
            self.balance = 0
        else:
            self.balance = initial_balance

    def deposit(self, amount):
        if amount <= 0:
            print("Deposit amount must be positive.")
        else:
            self.balance += amount
            print(f"Deposited: {amount}. New balance: {self.balance}")

    def withdraw(self, amount):
        if amount <= 0:
            print("Withdrawal amount must be positive.")
        elif amount > self.balance:
            print("Insufficient funds.")
        else:
            self.balance -= amount
```



```
        self.balance -= amount
        print(f"Withdrew: {amount}. New balance: {self.balance}")

    def check_balance(self):
        return self.balance

class TestSimplifiedBankAccount(unittest.TestCase):

    def test_initial_balance(self):
        account = BankAccount(100)
        self.assertEqual(account.check_balance(), 100)

    def test_deposit(self):
        account = BankAccount()
        account.deposit(50)
        self.assertEqual(account.check_balance(), 50)

    def test_withdraw(self):
        account = BankAccount(100)
        account.withdraw(30)
        self.assertEqual(account.check_balance(), 70)
```

Output:

Code explanation:

- The BankAccount class simulates a simple bank account with methods for initializing the balance, depositing, withdrawing, and checking the balance. It includes error handling for negative initial balances, non-positive deposit/withdrawal amounts, and insufficient funds during withdrawal.
- The TestBankAccount class contains several test methods (starting with test\_) that create BankAccount objects and use unittest assertions (like assertEquals and assertRaises) to verify that the BankAccount methods behave as expected under different conditions, including the error cases you requested.
- The if \_\_name\_\_ == '\_\_main\_\_': block at the end allows you to run all tests by default or select a specific test to run by entering its corresponding number when prompted.

Observation:

- The BankAccount class was successfully implemented with `__init__`, `deposit`, `withdraw`, and `check_balance` methods.
- Error handling for negative deposits and withdrawals, as well as insufficient funds during withdrawal, was implemented using `ValueError`.
- Test cases were generated covering successful deposits and withdrawals, attempts at negative transactions, and overdrafts.
- The execution of the generated test cases confirmed that the BankAccount class correctly handles valid transactions and raises `ValueError` for invalid operations (negative deposits/withdrawals and overdrafts).
- An interactive user interface was added, allowing users to perform deposit, withdrawal, and balance checks with real-time feedback and error handling for invalid user input.

#### Task Description#5

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 gracefully

#### Expected Output#5

- Number-based palindrome checker function validated against test cases.

Prompt: "Generate test scenarios for a function `is_number_palindrome` which verifies if an integer is a palindrome. The tests should confirm correct behavior for positive, negative, and zero inputs, as well as clearly distinguish between numbers that are palindromes and those that are not."

Code:

```

▶ import unittest

def is_number_palindrome(num):
    """
    Checks if an integer reads the same backward.
    Handles negative numbers and zero gracefully.
    """
    if num < 0:
        return False # Negative numbers are not considered palindromes

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

class TestIsNumberPalindrome(unittest.TestCase):

    def test_positive_palindrome(self):
        self.assertTrue(is_number_palindrome(121))
        self.assertTrue(is_number_palindrome(1221))
        self.assertTrue(is_number_palindrome(7))

    def test_positive_non_palindrome(self):
        self.assertFalse(is_number_palindrome(123))
        self.assertFalse(is_number_palindrome(10))
        self.assertFalse(is_number_palindrome(12345))

```

```

    self.assertRaises(is_number_palindrome(12345))

def test_zero(self):
    self.assertTrue(is_number_palindrome(0))

def test_negative_numbers(self):
    self.assertFalse(is_number_palindrome(-121))
    self.assertFalse(is_number_palindrome(-1))

if __name__ == '__main__':
    print("Running unit tests:")
    unittest.main(argv=['first-arg-is-ignored'], exit=False)

    print("\n--- Check a specific number ---")
    try:
        user_input = input("Enter an integer to check if it's a palindrome: ")
        num_to_check = int(user_input)
        if is_number_palindrome(num_to_check):
            print(f"{num_to_check} is a palindrome.")
        else:
            print(f"{num_to_check} is not a palindrome.")
    except ValueError:
        print("Invalid input. Please enter an integer.")

```

➤ .....

Output:



```
.....
```

```
-----  
Ran 13 tests in 0.011s
```

```
OK
```

```
Running unit tests:
```

```
--- Check a specific number ---
```

```
Enter an integer to check if it's a palindrome: 22
```

```
22 is a palindrome.
```

#### Code explanation:

- The `is_number_palindrome(num)` function checks if an integer `num` is a palindrome by converting it to a string and comparing it to its reversed version. It specifically handles negative numbers by returning `False`.
- The `TestIsNumberPalindrome` class contains test methods to verify the function's behavior with positive palindromes and non-palindromes, zero, and negative numbers.
- The `if __name__ == '__main__':` block here runs all the unit tests for the palindrome function and then prompts the user to enter an integer to check if it's a palindrome using the `is_number_palindrome` function.

#### Observation:

- Two distinct functionalities: The notebook contains code for two separate tasks: a `BankAccount` class with tests and an `is_number_palindrome` function with tests.
- Unit Testing Framework: Both sections utilize the unittest framework for creating and running automated tests, which is a good practice for verifying code correctness.
- Error Handling: The `BankAccount` class includes basic error handling for invalid inputs (negative values, insufficient funds).
- Palindrome Logic: The `is_number_palindrome` function converts the number to a string to easily check for the palindrome property. It explicitly handles negative numbers.



