

# AI ASSISTANT CODING

NAME : G. KARTHEEK

ROLL NO: 2403A53003

## ASSIGNMENT 8.2

### 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.

The screenshot shows a Google Colab notebook interface with the following content:

**Code Cell [7]:**

```
# Generate test cases for is_prime(n)
test_cases = [
    (0, False), # Edge case: 0
    (1, False), # Edge case: 1
    (2, True), # Edge case: 2 (smallest prime)
    (3, True), # Prime number
    (4, False), # Non-prime number
    (7, True), # Prime number
    (10, False), # Non-prime number
    (-5, False), # Edge case: negative number
    (101, True), # Large prime
    (100, False), # Large non-prime
    (997, True), # Another large prime
]
```

**Code Cell [6]:**

```
# Implement the is_prime(n) function
def is_prime(n):
    """
    Checks if a number is prime.

    Args:
        n: An integer.

    Returns:
        True if n is prime, False otherwise.
    """
    if n <= 1:
```

**Gemini Chat Panel:**

Generate test cases for a function `is_prime(n)` and then implement the function in Python. Requirements: only integers greater than 1 can be prime. Include edge cases like 0, 1, 2, negative numbers, and large primes

Empty cell

Sure, I will generate test cases and then implement the `is_prime(n)` function.

The tests for the `is_prime` function passed successfully! Would you like to add more test cases, or perhaps explore other prime-related problems like finding prime numbers within a range or generating prime factors?

What can I help you build?

Gemini can make mistakes so double-check it and use code with caution. [Learn more](#)

The screenshot shows a Google Colab notebook with two code cells. The first cell contains a function `is_prime(n)` that checks if a number is prime. The second cell contains test cases for the `is_prime` function, including edge cases like 0, 1, 2, negative numbers, and large primes. The output of the second cell shows that all test cases passed.

```

[6] True if n is prime, False otherwise.
if n <= 1:
    return False
if n <= 3:
    return True
if n % 2 == 0 or n % 3 == 0:
    return False
i = 5
while i * i <= n:
    if n % i == 0 or n % (i + 2) == 0:
        return False
    i += 6
return True

[5] # Test the function with the generated test cases
for number, expected in test_cases:
    result = is_prime(number)
    assert result == expected, f"For number {number}, expected {expected}, but got {result}"
    print(f"Test case {number}: Passed")

Test case 0: Passed
Test case 1: Passed
Test case 2: Passed
Test case 3: Passed
Test case 4: Passed
Test case 7: Passed
Test case 10: Passed
Test case -5: Passed
Test case 101: Passed
Test case 100: Passed
Test case 997: Passed

```

The Gemini chat interface on the right shows a conversation where the user asks for test cases for the `is_prime` function, and the AI provides a list of test cases and implements the function.

## 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

The screenshot shows a Google Colab notebook with two code cells. The first cell contains test cases for the `celsius_to_fahrenheit` and `fahrenheit_to_celsius` functions. The second cell contains the implementation of the `celsius_to_fahrenheit` function.

```

[11] # Generate test cases for celsius_to_fahrenheit(c)
celsius_to_fahrenheit_test_cases = [
    (0, 32.0), # Known pair: 0°C = 32°F
    (100, 212.0), # Known pair: 100°C = 212°F
    (25, 77.0), # Positive value
    (-10, 14.0), # Negative value
    (37.5, 99.5), # Decimal value
    (-40, -40.0), # Point where Celsius and Fahrenheit are equal
    ("abc", None), # Invalid input: string
    (None, None), # Invalid input: None
]

# Generate test cases for fahrenheit_to_celsius(f)
fahrenheit_to_celsius_test_cases = [
    (32, 0.0), # Known pair: 32°F = 0°C
    (212, 100.0), # Known pair: 212°F = 100°C
    (77, 25.0), # Positive value
    (14, -10.0), # Negative value
    (99.5, 37.5), # Decimal value
    (-40, -40.0), # Point where Celsius and Fahrenheit are equal
    ("abc", None), # Invalid input: string
    (None, None), # Invalid input: None
]

[9] # Implement the celsius_to_fahrenheit(c) function
def celsius_to_fahrenheit(c):
    """
    Converts Celsius to Fahrenheit.

    Args:
        c: Temperature in Celsius (number).
    """

```

The Gemini chat interface on the right shows a conversation where the user asks for test cases for the temperature conversion functions, and the AI provides a list of test cases and implements the functions.

assignment 8.2 x Untitled7.ipynb - Colab x Generate test cases x +

colab.research.google.com/drive/11dEW79jKDI4WyB5TEKf6AL3Pfx7yDvMO#scrollTo=e91297d3

File Edit View Insert Runtime Tools Help

Commands + Code + Text ▶ Run all

```
[9] ✓ 0s
# Implement the celsius_to_fahrenheit(c) function
def celsius_to_fahrenheit(c):
    """
    Converts Celsius to Fahrenheit.

    Args:
        c: Temperature in Celsius (number).

    Returns:
        Temperature in Fahrenheit (float) or None for invalid input.
    """
    if not isinstance(c, (int, float)):
        return None
    return (c * 9/5) + 32

# Implement the fahrenheit_to_celsius(f) function
def fahrenheit_to_celsius(f):
    """
    Converts Fahrenheit to Celsius.

    Args:
        f: Temperature in Fahrenheit (number).

    Returns:
        Temperature in Celsius (float) or None for invalid input.
    """
    if not isinstance(f, (int, float)):
        return None
    return (f - 32) * 5/9
```

[10]

Variables Terminal

9:35 PM Python 3

Gemini

Generate test cases for two functions: `celsius_to_fahrenheit(c)` and `fahrenheit_to_celsius(f)`. Requirements: validate known pairs (0°C = 32°F, 100°C = 212°F), include decimal values, and handle invalid inputs like strings or None. Then implement both functions in Python

Absolutely! I'll generate test cases for both `celsius_to_fahrenheit` and `fahrenheit_to_celsius`, including known pairs, decimal values, and invalid inputs. Then, I'll implement both functions in Python.

The temperature conversion functions and their test cases have been added and executed successfully. Would you like to add more test cases, or perhaps explore other unit conversions?

What can I help you build?

Gemini can make mistakes so double-check it and use code with caution. [Learn more](#)

assignment 8.2 x Untitled7.ipynb - Colab x Generate test cases x +

colab.research.google.com/drive/11dEW79jKDI4WyB5TEKf6AL3Pfx7yDvMO#scrollTo=e91297d3

File Edit View Insert Runtime Tools Help

Commands + Code + Text ▶ Run all

```
[10] ✓ 0s
# Test the celsius_to_fahrenheit function
print("Testing celsius_to_fahrenheit:")
for celsius, expected_fahrenheit in celsius_to_fahrenheit_test_cases:
    result = celsius_to_fahrenheit(celsius)
    assert result == expected_fahrenheit, f"For {celsius}°C, expected {expected_fahrenheit}°F, but got {result}°F"
    print(f"Test case {celsius}°C: Passed")

print("\nTesting fahrenheit_to_celsius:")
# Test the fahrenheit_to_celsius function
for fahrenheit, expected_celsius in fahrenheit_to_celsius_test_cases:
    result = fahrenheit_to_celsius(fahrenheit)
    assert result == expected_celsius, f"For {fahrenheit}°F, expected {expected_celsius}°C, but got {result}°C"
    print(f"Test case {fahrenheit}°F: Passed")
```

Testing celsius\_to\_fahrenheit:

- Test case 0°C: Passed
- Test case 100°C: Passed
- Test case 25°C: Passed
- Test case -10°C: Passed
- Test case 37.5°C: Passed
- Test case -40°C: Passed
- Test case abc°C: Passed
- Test case None°C: Passed

Testing fahrenheit\_to\_celsius:

- Test case 32°F: Passed
- Test case 212°F: Passed
- Test case 77°F: Passed
- Test case 14°F: Passed
- Test case 99.5°F: Passed
- Test case -40°F: Passed
- Test case abc°F: Passed
- Test case None°F: Passed

Variables Terminal

9:35 PM Python 3

Gemini

Generate test cases for two functions: `celsius_to_fahrenheit(c)` and `fahrenheit_to_celsius(f)`. Requirements: validate known pairs (0°C = 32°F, 100°C = 212°F), include decimal values, and handle invalid inputs like strings or None. Then implement both functions in Python

Absolutely! I'll generate test cases for both `celsius_to_fahrenheit` and `fahrenheit_to_celsius`, including known pairs, decimal values, and invalid inputs. Then, I'll implement both functions in Python.

The temperature conversion functions and their test cases have been added and executed successfully. Would you like to add more test cases, or perhaps explore other unit conversions?

What can I help you build?

Gemini can make mistakes so double-check it and use code with caution. [Learn more](#)

### 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 string

The screenshot shows a Google Colab notebook with the following content:

```
[41] ✓ Os
# Generate test cases for count_words(text)
count_words_test_cases = [
    ("This is a sentence.", 4), # Normal sentence
    (" Leading and trailing spaces.", 4), # Leading/trailing spaces
    ("Multiple spaces here.", 3), # Multiple spaces
    ("Sentence with punctuation!", 4), # Punctuation at the end
    ("", 0), # Empty string
    ("Onelword", 1), # Single word
    (" ", 0), # Only spaces
    ("Word,with,commas.", 3), # Punctuation within words
    ("Hyphenated-word", 1), # Hyphenated word
    ("Sentence. With. Multiple. Punctuation.", 4), # Multiple sentences with punctuation
    ("It's a contraction.", 3), # Contraction
    ("Word with a period.", 4), # Period at the end
    ("Word with a comma,", 3), # Comma at the end
    ("Word with a question mark?", 4), # Question mark at the end
    ("Word with an exclamation point!", 4), # Exclamation point at the end
    ("Word with multiple!!! punctuation...", 4), # Multiple punctuation marks
    (" Word with spaces and punctuation.", 4), # Combination of spaces and punctuation
    ("Word-with-hyphens-and-punctuation.", 1), # Hyphenated word with punctuation
    ("Another's contraction!", 2), # Contraction with punctuation
]
```

The Gemini chat interface on the right shows the following response:

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 string in python

Okay, I will generate test cases for the `count_words` function in a Python code cell, making sure to include cases with normal text, multiple spaces, punctuation, and empty strings as you requested.

I have generated the test cases for the `count_words` function as requested. Now, I can implement the function and then run the tests to ensure it works correctly with all the cases, including those with punctuation, multiple spaces, and empty strings. Would you like me to proceed with implementing the function and running the tests?

The screenshot shows a Google Colab notebook with the following content:

```
[42] ✓ Os
# Generate test cases for BankAccount class
import unittest

class TestBankAccount(unittest.TestCase):

    def test_initial_balance(self):
        account = BankAccount()
        self.assertEqual(account.check_balance(), 0)

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

    def test_deposit_zero(self):
        account = BankAccount()
        account.deposit(0)
        self.assertEqual(account.check_balance(), 0)

    def test_deposit_negative_raises_error(self):
        account = BankAccount()
        with self.assertRaises(ValueError):
            account.deposit(-50)
        self.assertEqual(account.check_balance(), 0) # Balance should remain unchanged

    def test_withdraw_positive(self):
        account = BankAccount()
        account.deposit(200)
        account.withdraw(50)
        self.assertEqual(account.check_balance(), 150)
```

The Gemini chat interface on the right shows the following response:

Generate test cases for a BankAccount class with methods `deposit(amount)`, `withdraw(amount)`, and `check_balance()`. Requirements: negative deposits or withdrawals should raise an error, and withdrawals cannot exceed the current balance. Then implement the class in Python

Okay, I can help you with that. I will generate test cases for a `BankAccount` class that include scenarios for valid and invalid deposits and withdrawals, as well as checking the balance. Then, I will implement the `BankAccount` class in Python, ensuring that it handles negative amounts and insufficient funds correctly.

The tests for the `BankAccount` class passed successfully! Although the cell execution shows as failed, the test results indicate that all 9 tests ran and passed.

Would you like to add more complex test cases, such as handling floating-point amounts or concurrency, or perhaps implement additional features like transferring funds between accounts?

```

[42] ✓ Os
def test_withdraw_zero(self):
    account = BankAccount()
    account.deposit(100)
    account.withdraw(0)
    self.assertEqual(account.check_balance(), 100)

def test_withdraw_negative_raises_error(self):
    account = BankAccount()
    account.deposit(100)
    with self.assertRaises(ValueError):
        account.withdraw(-50)
    self.assertEqual(account.check_balance(), 100) # Balance should remain unchanged

def test_withdraw_exceeds_balance_raises_error(self):
    account = BankAccount()
    account.deposit(100)
    with self.assertRaises(ValueError):
        account.withdraw(150)
    self.assertEqual(account.check_balance(), 100) # Balance should remain unchanged

def test_multiple_transactions(self):
    account = BankAccount()
    account.deposit(1000)
    account.withdraw(200)
    account.deposit(500)
    account.withdraw(300)
    self.assertEqual(account.check_balance(), 1000)

[43] ✓ Os
# Implement the BankAccount class
class BankAccount:
    """
    A simple bank account class.
    """
    def __init__(self):
        self._balance = 0

    def deposit(self, amount):
        """
        Deposits a positive amount to the account.

        Args:
            amount: The amount to deposit.

        Raises:
            ValueError: If the deposit amount is negative.
        """
        if amount < 0:
            raise ValueError("Deposit amount cannot be negative.")
        self._balance += amount

    def withdraw(self, amount):
        """
        Withdraws a positive amount from the account.

        Args:
            amount: The amount to withdraw.

        Raises:
            ValueError: If the withdrawal amount is negative or exceeds the balance.
        """

```

**Gemini**

6 Generate test cases for a BankAccount class with methods deposit(amount), withdraw(amount), and check\_balance(). Requirements: negative deposits or withdrawals should raise an error, and withdrawals cannot exceed the current balance. Then implement the class in Python

✦ Okay, I can help you with that. I will generate test cases for a BankAccount class that include scenarios for valid and invalid deposits and withdrawals, as well as checking the balance. Then, I will implement the BankAccount class in Python, ensuring that it handles negative amounts and insufficient funds correctly.

The tests for the BankAccount class passed successfully! Although the cell execution shows as failed, the test results indicate that all 9 tests ran and passed.

Would you like to add more complex test cases, such as handling floating-point amounts or concurrency, or perhaps implement additional features like transferring funds between accounts?

What can I help you build?

Gemini can make mistakes so double-check it and use code with caution. [Learn more](#)

```

[43] ✓ Os
# Implement the BankAccount class
class BankAccount:
    """
    A simple bank account class.
    """
    def __init__(self):
        self._balance = 0

    def deposit(self, amount):
        """
        Deposits a positive amount to the account.

        Args:
            amount: The amount to deposit.

        Raises:
            ValueError: If the deposit amount is negative.
        """
        if amount < 0:
            raise ValueError("Deposit amount cannot be negative.")
        self._balance += amount

    def withdraw(self, amount):
        """
        Withdraws a positive amount from the account.

        Args:
            amount: The amount to withdraw.

        Raises:
            ValueError: If the withdrawal amount is negative or exceeds the balance.
        """

```

**Gemini**

6 Generate test cases for a BankAccount class with methods deposit(amount), withdraw(amount), and check\_balance(). Requirements: negative deposits or withdrawals should raise an error, and withdrawals cannot exceed the current balance. Then implement the class in Python

✦ Okay, I can help you with that. I will generate test cases for a BankAccount class that include scenarios for valid and invalid deposits and withdrawals, as well as checking the balance. Then, I will implement the BankAccount class in Python, ensuring that it handles negative amounts and insufficient funds correctly.

The tests for the BankAccount class passed successfully! Although the cell execution shows as failed, the test results indicate that all 9 tests ran and passed.

Would you like to add more complex test cases, such as handling floating-point amounts or concurrency, or perhaps implement additional features like transferring funds between accounts?

What can I help you build?

Gemini can make mistakes so double-check it and use code with caution. [Learn more](#)

## Task Description#4

- 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.

The screenshot shows a Google Colab notebook titled 'Untitled7.ipynb'. The code is as follows:

```
[45] ✓ 0s
# Generate test cases for is_number_palindrome(num)
is_number_palindrome_test_cases = [
    (121, True), # Positive palindrome
    (123, False), # Positive non-palindrome
    (0, True), # Zero
    (5, True), # Single digit
    (-121, False), # Negative number (typically not considered palindromes)
    (1221, True), # Even number of digits palindrome
    (12321, True), # Odd number of digits palindrome
    (10, False), # Number ending in zero
    (101, True), # Palindrome with zero in middle
    (-101, False), # Negative palindrome
]

[46] ✓ 0s
# Implement the is_number_palindrome(num) function
def is_number_palindrome(num):
    """
    Checks if an integer is a palindrome.

    Args:
        num: An integer.

    Returns:
        True if the number is a palindrome, False otherwise.
    """
    # Negative numbers are not considered palindromes
    if num < 0:
        return False
    # Convert the number to a string
    num_str = str(num)
    # Check if the string reads the same forwards and backwards
    return num_str == num_str[::-1]
```

The Gemini chat interface on the right shows the following conversation:

6 Generate test cases for a function `is_number_palindrome(num)` that checks if an integer reads the same backward. Examples: `121 → True`, `123 → False`. Requirements: handle 0 and negative numbers gracefully. Then implement the function in Python

✦ Certainly! I can help you with that. I will generate test cases for the `is_number_palindrome` function, including positive palindromes, non-palindromes, single-digit numbers, zero, and negative numbers. Then, I will implement the function in Python, ensuring it handles these cases gracefully.

The `is_number_palindrome` function has been implemented and tested successfully with the generated test cases.

Would you like to explore variations of this problem, such as checking for palindromic strings, or perhaps work on a different type of coding challenge?

What can I help you build?

Gemini can make mistakes so double-check it and use code with caution. [Learn more](#)

The screenshot shows a Google Colab notebook titled 'Untitled7.ipynb'. The code is as follows:

```
[43] ✓ 0s
if amount < 0:
    raise ValueError("Withdrawal amount cannot be negative.")
if amount > self._balance:
    raise ValueError("Insufficient funds.")
self._balance -= amount

def check_balance(self):
    """
    Checks the current balance of the account.

    Returns:
        The current balance.
    """
    return self._balance

[44] ✓ 0s
# Run the test cases
if __name__ == '__main__':
    unittest.main(argv=['first-arg-is-ignored'], exit=False)

.....
Ran 9 tests in 0.012s
OK
```

The Gemini chat interface on the right shows the following conversation:

6 Generate test cases for a `BankAccount` class with methods `deposit(amount)`, `withdraw(amount)`, and `check_balance()`. Requirements: negative deposits or withdrawals should raise an error, and withdrawals cannot exceed the current balance. Then implement the class in Python

✦ Okay, I can help you with that. I will generate test cases for a `BankAccount` class that include scenarios for valid and invalid deposits and withdrawals, as well as checking the balance. Then, I will implement the `BankAccount` class in Python, ensuring that it handles negative amounts as well.

The tests for cell execution passed.

Would you like to point out any issues or like to transfer the balance?

Snip & Sketch

Snip saved to clipboard  
Select here to mark up and share the image

## 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

The screenshot shows a Google Colab notebook titled 'Untitled7.ipynb'. The interface includes a menu bar (File, Edit, View, Insert, Runtime, Tools, Help), a toolbar with icons for commands, code, text, and running, and a sidebar with navigation icons. The main editor area contains two code cells. Cell [45] defines a list of test cases for the `is_number_palindrome` function, including positive and negative palindromes, non-palindromes, zero, single digits, and numbers with zeros in the middle. Cell [46] implements the `is_number_palindrome` function. The function checks if a number is a palindrome by converting it to a string and comparing it with its reverse. It also handles negative numbers by returning False. The right sidebar shows the Gemini chat interface with a prompt to generate test cases and a response confirming the implementation and test results.

```
[45] OK
# Generate test cases for is_number_palindrome(num)
is_number_palindrome_test_cases = [
    (121, True),    # Positive palindrome
    (123, False),   # Positive non-palindrome
    (0, True),      # Zero
    (5, True),      # Single digit
    (-121, False),  # Negative number (typically not considered palindromes)
    (1221, True),   # Even number of digits palindrome
    (12321, True),  # Odd number of digits palindrome
    (10, False),    # Number ending in zero
    (101, True),    # Palindrome with zero in middle
    (-101, False),  # Negative palindrome
]

[46] OK
# Implement the is_number_palindrome(num) function
def is_number_palindrome(num):
    """
    Checks if an integer is a palindrome.

    Args:
        num: An integer.

    Returns:
        True if the number is a palindrome, False otherwise.
    """
    # Negative numbers are not considered palindromes
    if num < 0:
        return False
    # Convert the number to a string
```

The screenshot shows the same Google Colab notebook, but now with a third code cell [47] that tests the `is_number_palindrome` function. The test cell uses a loop to iterate over the test cases defined in cell [45], printing the results of each test. The output of the test is displayed in the terminal at the bottom of the notebook, showing that all test cases passed. The right sidebar shows the Gemini chat interface with a prompt to generate test cases and a response confirming the implementation and test results.

```
[47] OK
# Test the is_number_palindrome function
print("Testing is_number_palindrome:")
for number, expected in is_number_palindrome_test_cases:
    result = is_number_palindrome(number)
    assert result == expected, f"For number {number}, expected {expected}, but got {result}"
    print(f"Test case {number}: Passed")

Testing is_number_palindrome:
Test case 121: Passed
Test case 123: Passed
Test case 0: Passed
Test case 5: Passed
Test case -121: Passed
Test case 1221: Passed
Test case 12321: Passed
Test case 10: Passed
Test case 101: Passed
Test case -101: Passed
```