Test Documentation for get_products(file_name)

get_products(file_name):
The get_products(file_name) function is responsible for loading product information from a specified file, typically in formats like CSV or JSON. It reads the file, extracts details such as the product's name, price, stock quantity, and any other relevant attributes, and returns this data in a structured format, such as a list or dictionary. This function enables the system to dynamically load and update the product catalog from external data sources, ensuring that users have access to the most current product information when shopping.

Valid Scenarios:

1. **Basic Valid Input**
   Description: Tests with a correctly formatted CSV containing multiple products.
   Expected Result: Correct parsing of products with accurate data for name, price, and units.

2. **Valid CSV with Different Product Types**
   Description: CSV containing different product types, such as food and electronics.
   Expected Result: Products should be parsed and stored correctly, with correct name, price, and units.

3. **Valid CSV with Correct Price and Units Data**
   Description: Tests that the CSV file correctly stores the price and units for products.
   Expected Result: The price and units should be parsed correctly for each product.

4. **CSV with Single Product**
   Description: Tests a single product entry in the CSV.
   Expected Result: Correct parsing of a single product's name, price, and units.

5. **CSV with Multiple Products, Some with Zero Units**
   Description: Tests a scenario where products have zero units.
   Expected Result: Products are parsed, and the unit value for each product is correctly handled.

6. **CSV with Large Numbers**
   Description: Tests products with large values for price and units.
   Expected Result: Large numbers should be correctly parsed and stored without overflow or errors.

7. **Valid CSV File with Multiple Lines and Same Product Names**
   Description: CSV with multiple entries of the same product name but different units.
   Expected Result: Correct handling of multiple entries for the same product, with proper units.

8. **CSV with Products Having Decimal Prices**
   Description: Tests parsing of products with decimal prices.
   Expected Result: Decimal values should be parsed correctly.

9. **Valid CSV with All Units Equal to Zero**
   Description: Tests with products where all units are zero.
   Expected Result: Units for all products should be parsed as zero.

10. **CSV with Maximum Length (Large Data Set)**
    Description: Tests how the function handles a large data set (1000 rows).
    Expected Result: Correct parsing of a large number of products.

Invalid Scenarios:

1. **Invalid File Path**
   Description: Tests how the function handles an invalid file path.
   Expected Result: A FileNotFoundError should be raised.

2. **Invalid CSV Format (Missing Columns)**
   Description: Tests handling of a CSV file that is missing one or more columns.
   Expected Result: A KeyError should be raised due to the missing columns.

3. **Invalid Price Format (Non-Numeric Price)**
   Description: Tests handling of a non-numeric price value.
   Expected Result: A ValueError should be raised due to invalid price formatting.

4. **Invalid Units Format (Non-Numeric Units)**
   Description: Tests handling of a non-numeric units value.
   Expected Result: A ValueError should be raised due to invalid units formatting.

Edge Cases:

1. **CSV with Empty Data**
   Description: Tests handling of an empty CSV file (only header, no data).
   Expected Result: Should return an empty list or no products.

2. **CSV with Extra Spaces in Data**
   Description: Tests how the function handles extra spaces in the CSV data (e.g., Apple , 2 , 10).
   Expected Result: Should still correctly parse the products by trimming extra spaces.

3. **Empty Product Name**
   Description: Tests a scenario where a product has an empty name.
   Expected Result: The product should be handled appropriately, either raising an error or being skipped.

Test Documentation for check_cart(user, cart)

check_cart(user, cart):
The check_cart(user, cart) function validates the status of a user's shopping cart before proceeding with a checkout. It checks if the cart contains any items and verifies that the total cost does not exceed the user's available wallet balance. If the cart is empty or if there are any invalid items, such as those with negative prices or incorrect quantities, it raises appropriate errors. This function ensures that the user can only proceed to checkout if the cart is valid and the user has enough funds to cover the purchase, helping to maintain a smooth shopping experience.

Valid Scenarios:

1. **Single product, sufficient funds**: The user has enough funds to purchase a single product.

2. **Multiple products, sufficient funds**: The user has enough funds to purchase multiple products.
3. **Cart with multiple units of one product, sufficient funds**: The user can purchase multiple units of the same product without exceeding the wallet balance.
4. **Single product with exact wallet balance**: The user's wallet balance is exactly equal to the total price of a single product.
5. **Empty cart, no money needed**: The cart is empty, so no funds are needed from the user.
6. **Cart with fractional product prices, sufficient funds**: The user can afford products with fractional prices.
7. **Product removal with correct price calculation**: The cart's price adjusts correctly when a product is removed, and the user can still afford the updated cart.
8. **Multiple products, total within budget**: The user has enough funds for the total price of multiple products in the cart.
9. **High wallet balance, multiple products**: The user has a large enough wallet balance to purchase luxury items or multiple expensive products.
10. **Cart with no negative prices**: All product prices in the cart are valid (no negative prices).

Invalid Scenarios:

1. **Insufficient funds**: The user does not have enough funds to purchase all items in the cart.
2. **Negative product price**: One or more products in the cart have a negative price, which is invalid.
3. **Cart is None (invalid cart object)**: The cart is not a valid object, leading to an AttributeError.
4. **User with missing wallet attribute**: The user object is missing the wallet attribute, leading to an AttributeError.

Edge Cases:

1. **Empty cart**: The scenario where the cart contains no items.
2. **Exact wallet balance**: The scenario where the total price of the cart exactly matches the user's wallet balance.

3. **Cart with fractional product prices**: The scenario where product prices are in fractions (e.g., $2.50).
4. **Cart with one product of high value**: The scenario where a single product's price is equal to or exceeds the user's wallet balance.
5. **User with zero wallet balance**: The scenario where the user has no funds in the wallet, and the cart is non-empty.

Test Cases:

1. **Test Case: test_valid_cart_case_1()**
   Description: Verifies that a user with sufficient funds can purchase a single product.
   Expected Outcome: Returns True when the cart is valid.

2. **Test Case: test_valid_cart_case_2()**
   Description: Verifies that a user with sufficient funds can purchase multiple products.
   Expected Outcome: Returns True when the cart is valid.

3. **Test Case: test_valid_cart_case_3()**
   Description: Verifies that a user with sufficient funds can purchase multiple units of a single product.
   Expected Outcome: Returns True when the cart is valid.

4. **Test Case: test_valid_cart_case_4()**
   Description: Verifies that a user can purchase a product if their wallet balance matches the product's price exactly.
   Expected Outcome: Returns True when the cart is valid.

5. **Test Case: test_valid_cart_case_5()**
   Description: Verifies that an empty cart doesn't require any funds.
   Expected Outcome: Returns True when the cart is valid.

6. **Test Case: test_valid_cart_case_6()**
   Description: Verifies that a user with sufficient funds can purchase products with fractional prices.
   Expected Outcome: Returns True when the cart is valid.

7. **Test Case: test_valid_cart_case_7()**
   Description: Verifies that when a product is removed from the cart, the total price is recalculated correctly.
   Expected Outcome: Returns True when the cart is valid.

8. **Test Case: test_valid_cart_case_8()**
   Description: Verifies that a user can purchase multiple products without exceeding their budget.
   Expected Outcome: Returns True when the cart is valid.

9. **Test Case: test_valid_cart_case_9()**
   Description: Verifies that a user with a high wallet balance can purchase luxury items or

multiple expensive products.
Expected Outcome: Returns True when the cart is valid.

10. **Test Case: test_valid_cart_case_10()**
    Description: Verifies that a cart with no negative product prices is valid.
    Expected Outcome: Returns True when the cart is valid.

Invalid Test Cases:

1. **Test Case: test_invalid_cart_case_1()**
   Description: Verifies that the cart raises an exception when the user has insufficient funds.
   Expected Outcome: Raises a ValueError for insufficient funds.

2. **Test Case: test_invalid_cart_case_2()**
   Description: Verifies that the cart raises an exception when one or more products have a negative price.
   Expected Outcome: Raises a ValueError for invalid product price.

3. **Test Case: test_invalid_cart_case_3()**
   Description: Verifies that the cart raises an exception if the cart object is None.
   Expected Outcome: Raises an AttributeError due to invalid cart object.

4. **Test Case: test_invalid_cart_case_4()**
   Description: Verifies that the cart raises an exception if the user object is missing the wallet attribute.
   Expected Outcome: Raises an AttributeError due to missing wallet attribute in the user object.

Test Documentation for checkout(user, cart)

checkout(user, cart):
The checkout(user, cart) function processes the checkout procedure for a user, ensuring that the user has sufficient funds in their wallet to complete the purchase of the items in their cart. It calculates the total cost of all the items in the cart and compares it to the user's available balance. If the user has enough funds, the function deducts the total cost from the user's wallet and empties the shopping cart. This function helps manage the finalization of purchases and maintains accurate records of both the user's balance and the cart's contents.

Valid Scenarios:

1. **Valid checkout with items in cart**
   Test Case: test_checkout_with_valid_items_in_cart
   Description: This test checks if the checkout function works correctly when the user has sufficient wallet balance to complete the purchase.
   Expected Outcome: The wallet balance is correctly updated, and the cart is cleared.

2. **Checkout with exact wallet balance**
   Test Case: test_checkout_with_exact_wallet_amount
   Description: This test verifies that the checkout function works when the total cost of the cart matches the exact wallet balance.
   Expected Outcome: The user's wallet balance becomes zero, and the cart is cleared.

3. **Checkout with multiple items**
   Test Case: test_checkout_multiple_items
   Description: This test validates the checkout process when the user buys multiple different items.
   Expected Outcome: The wallet balance is updated correctly, and the cart is emptied.

4. **Checkout with zero wallet balance**
   Test Case: test_checkout_with_zero_wallet_balance
   Description: This test ensures that the checkout function handles cases where the user's wallet balance is sufficient to exactly cover the cart total.
   Expected Outcome: The wallet balance becomes zero, and the cart is cleared.

5. **Checkout with high quantity of a single item**
   Test Case: test_checkout_with_high_quantity
   Description: This test checks if the checkout works correctly with a high quantity of a single item.
   Expected Outcome: The wallet balance is updated, and the cart is emptied.

6. **Checkout with large wallet balance**
   Test Case: test_checkout_large_wallet
   Description: This test validates the behavior when the user has a large wallet balance.
   Expected Outcome: The wallet balance is decreased by the total cost, and the cart is emptied.

7. **Checkout with multiple units of the same item**
   Test Case: test_checkout_multiple_units_of_same_item
   Description: This test checks if the checkout handles purchasing multiple units of the same product correctly.
   Expected Outcome: The wallet balance is updated correctly, and the cart is cleared.

8. **Checkout when cart is already empty**
   Test Case: test_checkout_when_cart_is_already_empty
   Description: This test ensures that when the cart is empty, no changes are made to the user's wallet or the cart.
   Expected Outcome: The wallet balance remains unchanged, and the cart stays empty.

9. **Checkout with wallet precision**
   Test Case: test_checkout_wallet_precision
   Description: This test checks if the checkout function handles floating-point precision in the user's wallet balance correctly.
   Expected Outcome: The user's wallet balance is updated correctly without rounding errors, and the cart is cleared.

10. **Checkout with discounted items**
    Test Case: test_checkout_with_discounted_items
    Description: This test verifies the checkout function works when items in the cart are discounted before checkout.
    Expected Outcome: The wallet balance is updated according to the discounted price, and the cart is cleared.

Invalid Scenarios:

1. **Insufficient wallet balance**
    Test Case: test_checkout_insufficient_wallet_balance
    Description: This test checks if the checkout function handles situations where the user doesn't have enough funds to complete the purchase.
    Expected Outcome: The wallet balance remains unchanged, and the cart is not cleared.

2. **Checkout with negative wallet balance**
    Test Case: test_checkout_with_negative_wallet_balance
    Description: This test ensures that the checkout function works even when the user's wallet balance is negative.
    Expected Outcome: The wallet balance remains negative, and the cart is not cleared.

3. **Invalid product units (negative value)**
    Test Case: test_checkout_with_invalid_product_units
    Description: This test verifies that the checkout function correctly handles invalid units for a product, such as negative quantities.
    Expected Outcome: The wallet balance remains unchanged, and the cart is not cleared.

4. **Invalid product units (negative value)**
    Test Case: test_checkout_with_invalid_product_units
    Description: This test checks if the checkout function handles the case where a product has an invalid quantity (negative units). The product should not be added to the cart for processing.
    Expected Outcome: The wallet balance remains unchanged, and the cart is not cleared because the product with invalid units (negative quantity) should not be processed.

Edge Cases:

1. **Empty cart during checkout**
    Test Case: test_checkout_empty_cart
    Description: This test checks if the checkout function behaves correctly when the cart is empty at the time of checkout.
    Expected Outcome: The wallet balance remains the same, and the cart stays empty.

2. **Checkout with exact cost and zero units in cart**
    Test Case: test_checkout_with_exact_wallet_amount
    Description: Similar to the first test case, but with the cart being checked to ensure that no

items or units are overlooked.
Expected Outcome: The user's wallet is reduced to zero, and the cart is emptied.