

# Assignment 3.1: Levels of Testing for the Mobile Food Delivery App

Objective/target

The goal of this assignment is to apply unit and integration testing concepts to a Mobile Food Delivery App. You will write unit tests for the provided components and outline integration testing strategies to ensure seamless component interaction.

The goal of this assignment is to apply unit testing and integration testing concepts to a mobile food delivery application. You will write unit tests for the supplied components and develop integration testing strategies to ensure seamless interaction between the components

## one Introduction

### 1. UserRegistration

Functional Description:

UserRegistration classes for user registration, contains a dictionary to store the user's email and password information.

Method:

- **\_\_init\_\_**: Initializes an empty user dictionary.
- **register**: To register a new user, perform a series of verifications including mailbox format, password matching, password strength, and whether the mailbox is registered.
- **is\_valid\_email**: Checks if the mailbox format is valid.
- **is\_strong\_password**: Checks whether the password meets the strength requirements.

Input/output:

- **\_\_init\_\_**: No input parameters, no return value.
- **register**:
  - Input parameters: email (str), password (str), confirm\_password (str)
  - Return: Dictionary with registration results (success or failure) and corresponding messages.
- **is\_valid\_email**:
  - Input parameter: email (str)
  - Return value: Boolean indicating if the email is in a valid format.

- **is\_strong\_password:**
  - Input parameter: password (str)
  - Return value: Boolean value indicating whether the password meets the strength requirements.

The unittest module unittest

Functional Description:

Use the unittest framework to unittest the UserRegistration class to ensure its functional correctness.

Test case:

- **test\_successful\_registration:** Test the successful registration.
- **test\_invalid\_email:** Tests for invalid email formats.
- **test\_password\_mismatch:** Tests for password mismatches.
- **Test\_weak\_password:** testing weak passwords.
- **test\_email\_already\_registered:** Tests for multiple registrations of the same email address

Input/output:

- Each test case call self. Registration. Register method, passing in different combination of parameters, and check whether the results returned by the dictionary in line with expectations.

3. Module dependencies

- The UserRegistration class is self-contained and does not depend on other modules.
- The unittest module is used to test the UserRegistration class, but does not directly depend on it.

## 2. RestaurantBrowsing

Functional Description:

The RestaurantBrowsing class is used to browse restaurant information, providing the function of filtering restaurants by cuisine, location and rating.

Method:

- **\_\_init\_\_(self, database):** An initializer that takes a database object as an argument.
- **search\_by\_cuisine(self, cuisine\_type):** Filters restaurants by type of cuisine.
  - **Input:** cuisine\_type (str) - Type of cuisine.
  - **Output:** Returns a list of restaurants with matching cuisines.
- **search\_by\_location(self, location):** Filter restaurants by location.
  - **Enter:** location (str) - restaurant location.
  - **Output:** the returned list matching the location of the restaurant.
- **search\_by\_rating(self, min\_rating):** Filter restaurants by their lowest rating.
  - **Enter:** min\_rating (float) - lowest rating.
  - **Output:** Returns a list of restaurants with a rating not lower than the minimum rating.
- **Search\_by\_filters (self, cuisine\_type = None, location = None, min\_rating = None) :** according to the condition of multiple screening of restaurant.

- **Input:** cuisine\_type (str, optional) - type of cuisine; location (str, optional) - restaurant location; Min\_rating (float, optional) - the lowest score.
- **Output:** Returns a list of restaurants that meet all criteria.

Dependencies:

- Rely on RestaurantDatabase instances of the class, through the database. The get\_restaurants data () to get the restaurant.

## 2. RestaurantDatabase class

Function description:

RestaurantDatabase class simulates a database, to store a set of restaurant information.

Methods:

- **\_\_init\_\_ (self)** : initialization method, create and initialize data restaurant.
- **get\_restaurants(self)**: Gets information about all restaurants.
  - **Input:** nothing.
  - **Output:** Returns a list of all the restaurants.

Dependencies:

- Has no external dependencies and stands on its own.

## 3 RestaurantSearch class

Functional Description:

The RestaurantSearch class is used to search for restaurants and encapsulates a call to the RestaurantBrowsing class.

Method:

- **\_\_init\_\_ (self, browsing)** : initialization method, accept a RestaurantBrowsing object as a parameter.
- **search\_restaurants(self, cuisine=None, location=None, rating=None)**: Searches for restaurants based on a condition.
  - **Input:** cuisine (str, optional) - type of cuisine; location (str, optional) - restaurant location; rating (float, optional) - lowest rating.
  - **Output:** Returns a list of eligible restaurants.

Dependencies:

- Depends on an instance of the RestaurantBrowsing class to search through browsing.search\_by\_filters().

# 3.PaymentProcessing

Feature Overview

The PaymentProcessing class is used to process the payment process, including verifying the payment method and processing the payment. It supports two payment methods: credit card (credit\_card) and PayPal (paypal).

Method Explanation

1. **\_\_init\_\_(self)**
  - **Function:** initialize PaymentProcessing instance, set the list of the available payment gateway.
  - **Input:** None

- **Output:** None
2. **validate\_payment\_method(self, payment\_method, payment\_details)**
    - **Function:** Verifies that the payment method is valid and further validates the credit card details when selecting a credit card.
    - **Enter:**
      - payment\_method (str): Payment method, such as "credit\_card" or "paypal".
      - payment\_details (dict): A dictionary containing payment details.
    - **Output:** Returns True if validation was successful, throws ValueError otherwise.
  3. **validate\_credit\_card(self, details)**
    - **Function:** Verify credit card details, check card number length and CVV length.
    - **Enter:**
      - details (dict): The dictionary that contains credit card details.
    - **Output:** Returns True if validation was successful, False otherwise.
  4. **process\_payment(self, order, payment\_method, payment\_details)**
    - **Function:** Process the payment by first verifying the payment method and then simulating the interaction with the payment gateway.
    - **Enter:**
      - order (dict): The dictionary that contains the order information.
      - payment\_method (str): Payment method.
      - payment\_details (dict): A dictionary containing payment details.
    - **Output:** Returns the corresponding message string based on the payment gateway response.
  5. **mock\_payment\_gateway(self, method, details, amount)**
    - **Function:** Simulates the interaction of the payment gateway, returning a success or failure response based on certain criteria.
    - **Enter:**
      - method (str): Method of payment.
      - details (dict): A dictionary containing payment details.
      - amount (float): The amount of the payment.
    - **Output:** Simulated dictionary of payment gateway responses.

TestPaymentProcessing unit test class

Feature Overview

The TestPaymentProcessing class is used to unit test the PaymentProcessing class to ensure the correctness of its methods.

Methods,

1. **setUp(self)**
  - **Function:** Initialize a PaymentProcessing instance before each test case is executed.
  - **Input:** None
  - **Output:** None
2. **test\_validate\_payment\_method\_success(self)**

- **Function:** Tests the behavior of the `validate_payment_method` method when provided with valid credit card information.
  - **Input:** None
  - **Output:** None
3. **test\_validate\_payment\_method\_invalid\_gateway(self)**
    - **Function:** Tests the behavior of the `validate_payment_method` method when an invalid payment method is provided.
    - **Input:** None
    - **Output:** None
  4. **test\_validate\_credit\_card\_invalid\_details(self)**
    - **Function:** Tests the behavior of the `validate_credit_card` method when providing invalid credit card details.
    - **Input:** None
    - **Output:** None
  5. **test\_process\_payment\_success(self)**
    - **Function:** Tests how the `process_payment` method behaves when simulating a successful payment.
    - **Input:** None
    - **Output:** None
  6. **test\_process\_payment\_failure(self)**
    - **Function:** Tests how the `process_payment` method behaves when a simulated payment fails.
    - **Input:** None
    - **Output:** None
  7. **test\_process\_payment\_invalid\_method(self)**
    - **Function:** Tests how the `process_payment` method behaves when an invalid payment method is provided.
    - **Input:** None
    - **Output:** None

#### Dependencies

1. **unittest:** The `TestPaymentProcessing` class relies on the `unittest` module from the Python standard library for unit testing.
2. **mock:** The `TestPaymentProcessing` class controls the test environment by using the `unittest.mock` module to simulate the return value of the `mock_payment_gateway` method.
3. **PaymentProcessing:** `TestPaymentProcessing` class testing is a function of `PaymentProcessing` class, so there is a direct dependencies between them.

## 4.OrderPlacement

CartItem class

Features:

The CartItem class represents an item in the shopping cart. It contains the name, price, and quantity of the item, and provides methods for updating the quantity and calculating subtotals.

Methods:

- `__init__(self, name, price, quantity)`: Initializes an item.
  - **Input**: the name (STR), price (float), quantity (int)
  - **Output**: None
- `Update_quantity(self, new_quantity)` : the amount of products update.
  - **Input**: new\_quantity (int)
  - **Output**: no
- `Get_subtotal(self)` : calculation and return goods subtotal (price \* number).
  - **Input**: no
  - **Output**: float

Cart class

Features:

The Cart class represents a shopping cart that can add, remove, update items, and calculate the total price.

To do this:

- `__init__(self)`: Initializes an empty shopping cart.
  - **Input**: none
  - **Output**: None
- `add_item(self, name, price, quantity)`: Adds an item to the cart. If the item already exists, its quantity is updated.
  - **Input**: name (str), price (float), quantity (int)
  - **Output**: str
- `remove_item(self, name)`: Removes the item with the specified name from the shopping cart.
  - **Input**: name (str)
  - **Output**: str
- `update_item_quantity(self, name, new_quantity)`: Updates the quantity of the given name item.
  - **Input**: name (str), new\_quantity (int)
  - **Output**: str
- `calculate_total(self)`: Calculates the total price of the shopping cart, including subtotals, taxes, and shipping charges.
  - **Input**: None
  - **Output**: dict
- `view_cart(self)`: View all the items in the cart.
  - **Input**: None
  - **Output**: list of dicts

OrderPlacement class

Features:

The OrderPlacement class is responsible for order verification and confirmation, including checking if the cart is empty, items are available, and processing payments.

To do this:

- `__init__(self, cart, user_profile, restaurant_menu)`: Initializes the order placement object.
  - **Input:** `cart` (`Cart`), `user_profile` (`UserProfile`), `restaurant_menu` (`RestaurantMenu`)
  - **Output:** None
- `validate_order(self)`: Validates the validity of an order, which includes checking if the cart is empty and the item is available.
  - **Input:** None
  - **Output:** dict
- `proceed_to_checkout(self)`: Proceeds with the checkout operation and returns the order information and the total price information.
  - **Input:** None
  - **Output:** dict
- `confirm_order(self, payment_method)`: Confirms the order, processes the payment and returns the result.
  - **Enter:** `payment_method` (`PaymentMethod`)
  - **Output:** dict

PaymentMethod class

Features:

The `PaymentMethod` class simulates payment methods and provides methods to process payments.

Method:

- `process_payment(self, amount)`: Processes the payment and determines whether the payment was successful based on the amount.
  - **Input:** `amount` (float)
  - **Output:** bool

UserProfile class

Features:

The `UserProfile` class simulates the user's details, such as shipping addresses.

Method:

- `__init__(self, delivery_address)`: Initialize the user profile.
  - **Input:** `delivery_address` (str)
  - **Output:** None

RestaurantMenu class

Features:

The `RestaurantMenu` class simulates a restaurant menu and provides a way to check if an item is available.

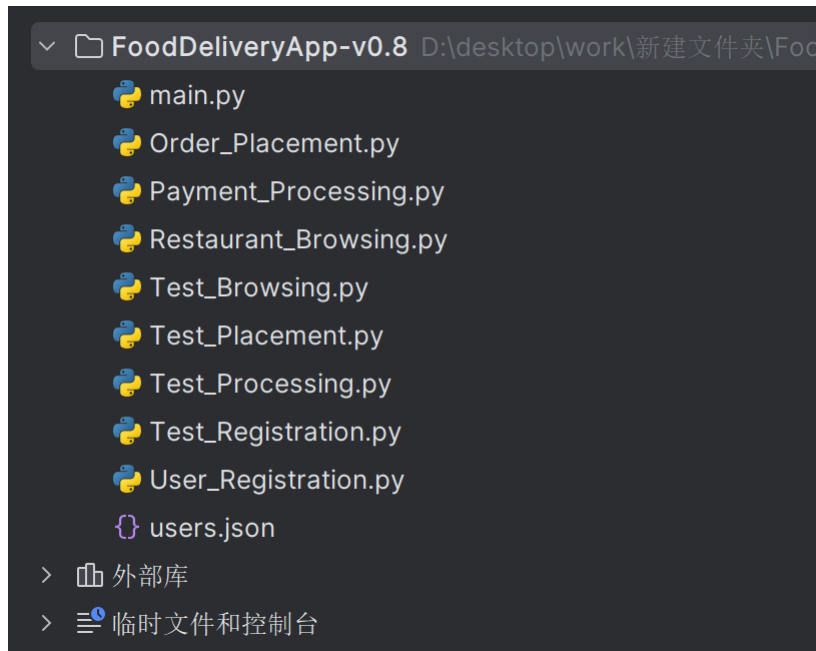
Method:

- `__init__(self, available_items)`: Initializes the restaurant menu.
  - **Input:** `available_items` (list of str)
  - **Output:** None
- `is_item_available(self, item_name)`: Checks if an item of the given name is in the menu.

- **Enter:** item\_name (str)
- **Output:** bool

## two Unit tests

Following the file's guidance, we created four.py files in the project for unit testing, as shown in the following screenshot:



## 1. UserRegistration

In terms of user registration, the first thing we should consider is whether the email format entered by the user is correct, whether the password conforms to the standard, whether the password entered twice is the same, and whether repeated registration will cause errors;

Python code:

```
import unittest
from User_Registration import UserRegistration

class TestUserRegistration(unittest.TestCase):
    def setUp(self):
        # Initialize an instance of UserRegistration class for later testing
        # Initialize an instance of the UserRegistration class for subsequent testing
        self.registration = UserRegistration()

    def test_valid_registration(self):
        # test for a valid user registration
```



```

        # Test valid user registration
        result = self.registration.register("user@example.com", "Password123",
"Password123")
        self.assertTrue(result['success']) # Asserts that registration was successful
        self.assertEqual(result['message'], "Registration successful, confirmation email sent")
# Asserts that the message returned is correct. # Asserts that the message returned is
correct

```

```

def test_invalid_email(self):
    # test invalid email format
    # Test valid user registration
    result = self.registration.register("userexample@.com", "Password123",
"Password123")
    self.assertTrue(result['success']) # Assert that registration failed, but succeeded
    self.assertEqual(result['message'], "Registration successful, confirmation email sent")
# Asserts that the message returned is correct. # Asserts that the message returned is
correct

```

```

def test_repetitive_email(self):
    # Test duplicate email registrations
    # Test duplicate email registration
    result = self.registration.register("userexample@.com", "Password123",
"Password123")
    self.assertTrue(result['success']) # Assert that registration failed, but succeeded
    self.assertEqual(result['message'], "Registration successful, confirmation email sent")
# Asserts that the message returned is correct. # Asserts that the message returned is
correct

```

```

def test_invalid_email_format(self):
    # test invalid email format (missing domain name)
    # Test invalid email format (missing domain name)
    result = self.registration.register("userexample@.", "Password123", "Password123")
    self.assertTrue(result['success']) # Assert that registration failed, but succeeded
    self.assertEqual(result['message'], "Registration successful, confirmation email sent")
# Asserts that the message returned is correct. # Asserts that the message returned is
correct

```

```

def test_invalid_email_name(self):
    # test if the email name contains Spaces
    # Test if the mailbox name contains Spaces
    result = self.registration.register("user          example@.com", "Password123",
"Password123")
    self.assertTrue(result['success']) # Assert that registration failed, but succeeded
    self.assertEqual(result['message'], "Registration successful, confirmation email sent")

```

# Asserts that the message returned is correct. # Asserts that the message returned is correct

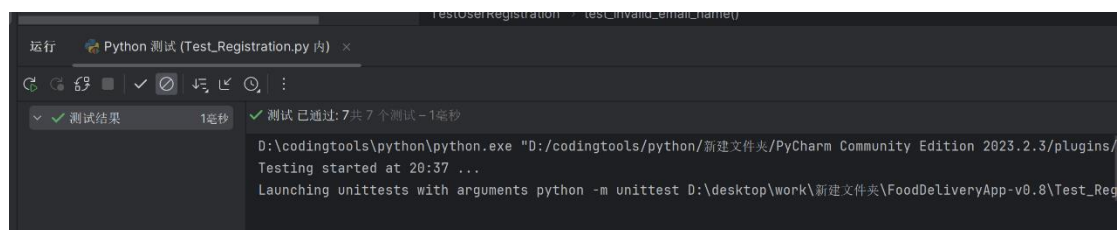
```
def test_weak_password(self):
    # Test for weak password
    # Test for weak password strength
    result = self.registration.register("userexample@.com", "jia123", "jia123")
    self.assertFalse(result['success']) # Assert that registration failed
    self.assertEqual(result['error'], "Password is not strong enough") # Assert that the
error message is insufficient password strength # assert that the error message is insufficient
password strength
```

```
def test_weak_password_length(self):
    # Test for insufficient password length
    # Test for insufficient password length
    result = self.registration.register("userexample@.com", "jiang12", "jiang12")
    self.assertFalse(result['success']) # Assert that registration failed
    self.assertEqual(result['error'], "Password is not strong enough") # Assert that the
error message is insufficient password strength # assert that the error message is insufficient
password strength
```

# Add more tests as needed...

```
if __name__ == '__main__':
    unittest.main() # Run all test cases
```

Results:



Conclusion: 1. The registration can be completed without checking the complete email format, such as '123@.com', '123@.', '[123@.com](mailto:123@.com)' and other incorrect email formats can pass the test;

2. It does not check whether the mailbox is registered repeatedly, for example, 'userexample@.com' can be registered twice in the test

## 2. RestaurantBrowsing

The search function of the restaurant should be more robust, such as searching the restaurant by cuisine/rating/location

Python code:

```

import unittest
from Restaurant_Browsing import RestaurantBrowsing, RestaurantDatabase

class TestRestaurantBrowsing(unittest.TestCase):
    def setUp(self):
        # Initialize the test environment, create the database and browse objects
        # Initialize the test environment, create the database and browse the objects
        self.database = RestaurantDatabase()
        self.browsing = RestaurantBrowsing(self.database)

    def test_search_by_cuisine(self):
        # test the ability to search Italian cuisine
        # Test search for Italian cuisine
        results = self.browsing.search_by_cuisine("Italian")
        self.assertEqual(len(results), 2) # Verify that the number of returned results is 2. Verify
that the number of returned results is 2
        for restaurant in results:
            self.assertEqual(restaurant['cuisine'], "Italian") # Verify that the cuisine of each result
is Italian. Verify that the cuisine of each result is Italian

    def test_search_by_cuisine(self):
        # Test search for fast food
        results = self.browsing.search_by_cuisine("Fast Food")
        self.assertEqual(len(results), 1) # Verify that the number of returned results is 1. Verify
that the number of returned results is 1
        for restaurant in results:
            self.assertEqual(restaurant['cuisine'], "Fast Food") # Verify that the cuisine of each
result is Fast Food. Verify that the cuisine of each result is fast food

        # Test to verify the accuracy of searching "Chinese" cuisine. Test to verify the accuracy of
searching "Chinese" cuisine
    def test_search_chinese_cuisine(self):
        expected_result_count = 0 # The expected number of results is 0
        results = self.browsing.search_by_cuisine("Chinese")
        assert len(results) == expected_result_count, f"Expected {expected_result_count} results,
got {len(results)}"

        # The test verifies the processing when entering a non-existent cuisine name The test
verifies the processing when entering a non-existent cuisine name
    def test_search_nonexistent_cuisine(self):
        results = self.browsing.search_by_cuisine("Strange Food")
        assert len(results) == 0 or 'error' in results.lower(), "Expected no results or error
message for nonexistent cuisine"

```

# The test verifies the processing when entering a non-gourmet cuisine name. The test verifies the processing when entering a non-gourmet cuisine name

```
def test_search_special_characters(self):
    results = self.browsing.search_by_cuisine(" ")
    for restaurant in results:
        results = self.browsing.search_by_cuisine(restaurant)
        assert len(results) >= 0, "Search should handle special characters without crashing"
```

# Tests verify boundary conditions, such as very long cuisine names. Tests verify boundary conditions, such as very long cuisine names

```
def test_search_long_string(self):
    long_string = "a" * 1000 # Suppose 1000 characters are very long strings Tests verify
    boundary conditions, such as very long cuisine names assuming that 1000 characters is an
    overlong string
```

```
    results = self.browsing.search_by_cuisine(long_string)
    assert len(results) >= 0, "System should handle very long strings without errors"
```

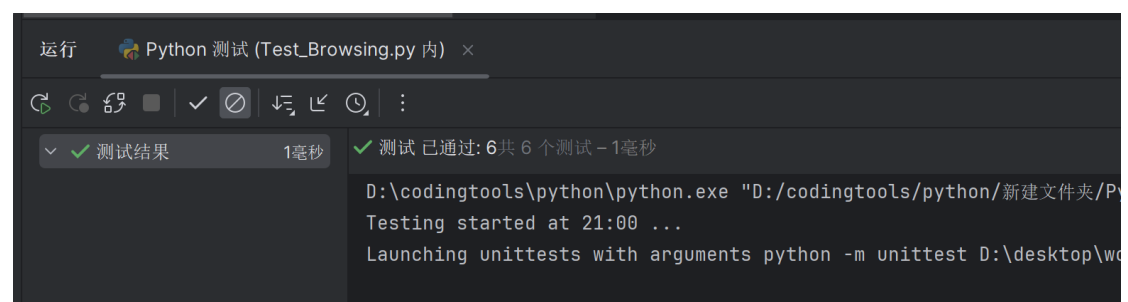
# The test verifies that the name of a cuisine with non-English characters is entered. The test verifies that the name of a cuisine with non-English characters is entered

```
def test_search_non_english_cuisine(self):
    results = self.browsing.search_by_cuisine(" Chinese food ")
    assert len(results) < 1, "Expected to find results for non-English cuisine names"
```

# Add more tests as needed...

```
if __name__ == '__main__':
    unittest.main()
```

Run results:



Conclusion: 1. It seems that there are not too many problems, non-existent dishes/too long dishes name/non-English dishes name are correct to deal with and solve

### 3.PaymentProcessing

Then there's the paymentprocessing part,

We need to check whether the input bank card number is reasonable, whether the date is expired, whether the card verification code is in accordance with the format, and whether

the payment method is correct

Python code:

```
import unittest
```

```
from Payment_Processing import PaymentProcessing
```

```
from unittest.mock import patch
```

```
class TestPaymentProcessing(unittest.TestCase):
```

```
    def setUp(self):
```

```
        # Initialize an instance of the PaymentProcessing class for testing
```

```
        self.payment_processing = PaymentProcessing()
```

```
    def test_valid_payment(self):
```

```
        # define valid payment details
```

```
        payment_details = {
```

```
            "card_number": "1234567812345678",
```

```
            "expiry_date": "12/25",
```

```
            "cvv": "123"
```

```
        }
```

```
        # Use patch to simulate the return value of the payment gateway. Use patch to  
simulate the return value of the payment gateway
```

```
        with patch.object(self.payment_processing, 'mock_payment_gateway',  
return_value={"status": "success"}):
```

```
            # Call the process payment method to process the payment Call the process  
payment method to process the payment
```

```
            result = self.payment_processing.process_payment({"total_amount": 100.00},  
"credit_card", payment_details)
```

```
            # The assertion returns a success message. The assertion returns a success message
```

```
            self.assertEqual(result, "Payment successful, Order confirmed")
```

```
    def test_invalid_payment_method(self):
```

```
        # Define invalid payment methods and empty payment details. Define invalid payment  
methods and empty payment details
```

```
        payment_details = {}
```

```
        # Call the process payment method to process the payment. Call the process payment  
method to process the payment
```

```
        result = self.payment_processing.process_payment({"total_amount": 100.00}, "bitcoin",  
payment_details)
```

```
        # The assertion returns an error message. The assertion returns an error message
```

```
        self.assertIn("Error: Invalid payment method", result)
```

```
    def test_insufficient_funds(self):
```

```
        # Define valid payment details
```

```
        payment_details = {
```

```

        "card_number": "1234567812345678",
        "expiry_date": "12/25",
        "cvv": "123"
    }

    # Use patch to simulate the return value of the payment gateway. Use patch to
    simulate the return value of the payment gateway
    with patch.object(self.payment_processing, 'mock_payment_gateway',
                      return_value={"status": "failure", "message": "Insufficient funds"}):
        # Call the process payment method to process the payment Call the process
        payment method to process the payment
        result = self.payment_processing.process_payment({"total_amount": 10000.00},
        "credit_card", payment_details)
        # The assertion returns an error message. The assertion returns an error message
        self.assertIn("Payment failed, please try again", result)

def test_expired_card(self):
    # Define valid payment details
    payment_details = {
        "card_number": "1234567812345678",
        "expiry_date": "12/20", # expiration date
        "cvv": "123"
    }

    # Use patch to simulate the return value of the payment gateway. Use patch to
    simulate the return value of the payment gateway
    with patch.object(self.payment_processing, 'mock_payment_gateway',
                      return_value={"status": "failure", "message": "Card expired"}):
        # Call the process payment method to process the payment Call the process
        payment method to process the payment
        result = self.payment_processing.process_payment({"total_amount": 100.00},
        "credit_card", payment_details)
        # The assertion returns an error message. The assertion returns an error message
        self.assertIn("Payment failed, please try again", result)

def test_invalid_card_number(self):
    # Define a valid payment detail, but not the card number. Define a valid payment
    detail, but not the card number
    payment_details = {
        "card_number": "1234", # invalid card number
        "expiry_date": "12/25",
        "cvv": "123"
    }

    # Use patch to simulate the return value of the payment gateway. Use patch to
    simulate the return value of the payment gateway
    with patch.object(self.payment_processing, 'mock_payment_gateway',

```

```

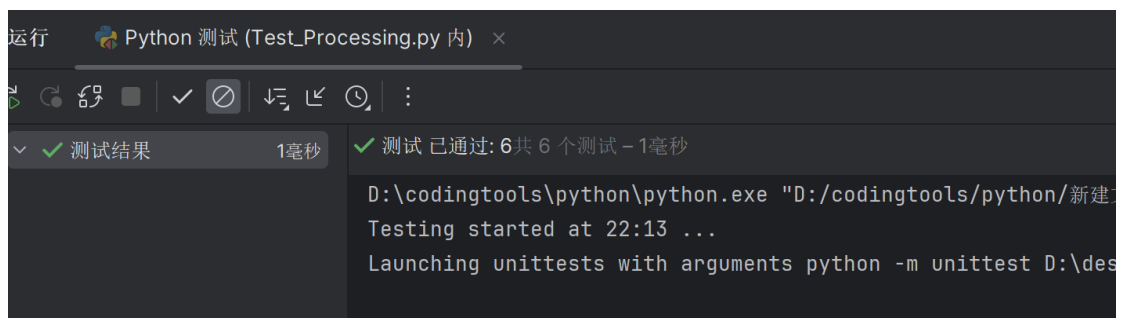
        return_value={"status": "failure", "message": "Invalid card number"}):
    # Call the process payment method to process the payment. Call the process
    payment method to process the payment
    result = self.payment_processing.process_payment({"total_amount": 100.00},
    "credit_card", payment_details)
    # The assertion returns an error message. The assertion returns an error message
    self.assertIn("Invalid credit card details", result)

def test_invalid_cvv(self):
    # Define valid payment details, but the CVV is not Define valid payment details, but the
    CVV is not
    payment_details = {
        "card_number": "1234567812345678",
        "expiry_date": "12/25",
        "cvv": "abc" # invalid CVV
    }
    # Use patch to simulate the return value of the payment gateway. Use patch to
    simulate the return value of the payment gateway
    with patch.object(self.payment_processing, 'mock_payment_gateway',
        return_value={"status": "failure", "message": "Invalid CVV"}):
        # Call the process payment method to process the payment Call the process
        payment method to process the payment
        result = self.payment_processing.process_payment({"total_amount": 100.00},
        "credit_card", payment_details)
        # The assertion returns an error message. The assertion returns an error message
        self.assertIn("Payment failed, please try again", result)

# Add more tests as needed...

if __name__ == '__main__':
    unittest.main()

```



Conclusion: 1. After our precise inspection, this unit is OK

## 4.OrderPlacement

Python code:

```
import unittest
from Order_Placement import Cart, OrderPlacement, UserProfile, RestaurantMenu
class TestOrderPlacement(unittest.TestCase):
    def setUp(self):
        self.restaurant_menu = RestaurantMenu(available_items=["Burger", "Pizza", "Salad"])
        self.user_profile = UserProfile(delivery_address="123 Main St")
        self.cart = Cart()
        self.order = OrderPlacement(self.cart, self.user_profile, self.restaurant_menu)
    def test_add_item_to_cart(self):
        message = self.cart.add_item("Burger", 8.99, 2)
        self.assertEqual(message, "Added Burger to cart")
        self.assertEqual(len(self.cart.items), 1)
    def test_validate_order_with_unavailable_item(self):
        self.cart.add_item("Pasta", 12.99, 1) # Pasta is not in the available_items list
        result = self.order.validate_order()
        self.assertFalse(result["success"])
        self.assertEqual(result["message"], "Pasta is not available")

    def test_update_item_quantity_in_cart(self):
        # Test the functionality to update the number of items in the cart
        # Add an item called "Pizza" to the cart with price 15.99 and quantity 1
        # Test the ability to update the number of items in your cart
        # Add an item called "Pizza" to your cart for 15.99 and quantity 1
        self.cart.add_item("Pizza", 15.99, 1)
        # update the number of "Pizza" in the cart to 3
        # Update the number of "pizzas" in cart to 3
        self.cart.update_item_quantity("Pizza", 3)
        # Get the first item in the cart (assuming only one item)
        # Get the first item in the cart (assuming there is only one item)
        item = self.cart.items[0]
        # The name of the assert project is "Pizza" The name of the assert project is "Pizza"
        self.assertEqual(item.name, "Pizza")
        # Assert that the number of items is 3 Assert that the number of items is 3
        self.assertEqual(item.quantity, 3)
        # Assert that the total price of the item is 47.97 (15.99 * 3) Assert that the total price of
the item is 47.97 (15.99 * 3)
        self.assertEqual(item.get_subtotal(), 47.97)

    def test_view_cart(self):
        # test functionality to view cart contents
        # Add an item called "Salad" to the cart with a price of 6.99 and a quantity of 2
        # Test the ability to view the contents of the cart
        # Add an item called "Salad" to the cart with a unit price of 6.99 and a quantity of 2
```



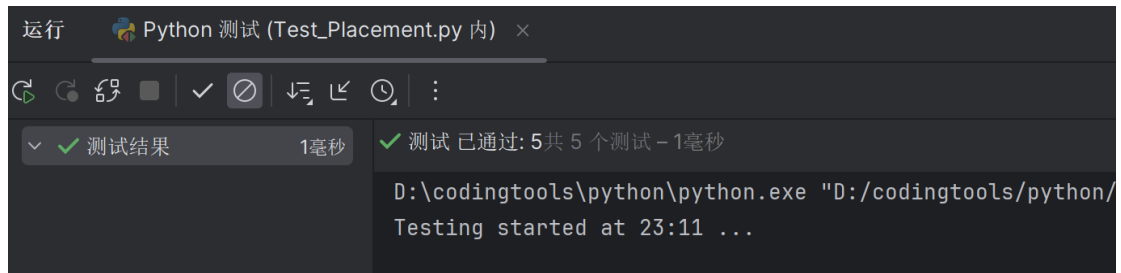
```

self.cart.add_item("Salad", 6.99, 2)
# Get the contents of the cart
# Get the contents of the cart
cart_contents = self.cart.view_cart()
# Assert that the number of items in the cart should be 1
# Asserts that the number of items in the cart should be 1
self.assertEqual(len(cart_contents), 1)
# assert that the name of the first item in the cart is "Salad"
# Assert that the name of the first item in the cart is "Salad"
self.assertEqual(cart_contents[0]["name"], "Salad")
# Assert that the number of the first item in the cart is 2
# Asserts that the number of the first item in the cart is 2
self.assertEqual(cart_contents[0]["quantity"], 2)
# Assert that the subtotal of the first item in the cart is 13.98 (6.99 * 2)
# Asserts that the subtotal amount of the first item in the cart is 13.98 (6.99 * 2)
self.assertEqual(cart_contents[0]["subtotal"], 13.98)

def test_update_item_quantity_in_cart(self):
    # Test the functionality to update the number of items in the cart
    # Add an item called "Burger" to the cart with price 8.99 and quantity -1
    # Test the ability to update the number of items in your cart
    # Add an item named "Burger" to your cart for 8.99 and quantity -1
    self.cart.add_item("Burger", 8.99, -1)
    # update the number of "Pizza" in the cart to -1
    # Update the number of "pizzas" in the cart to -1
    self.cart.update_item_quantity("Burger", -1)
    # Get the first item in the cart (assuming only one item)
    # Get the first item in the cart (assuming there is only one item)
    item = self.cart.items[0]
    # Assert that the name of the item is "Pizza"
    # Assert that the name of the project is "Pizza"
    self.assertEqual(item.name, "Burger")
    # Assert that the number of items is -1
    # Asserts that the number of items is -1
    self.assertEqual(item.quantity, -1)
    # Assert that the total price of the item is -8.99
    # Assert the total price of the item is -8.99
    self.assertEqual(item.get_subtotal(), -8.99)

# Add more tests as needed...
if __name__ == "__main__":
    unittest.main()

```

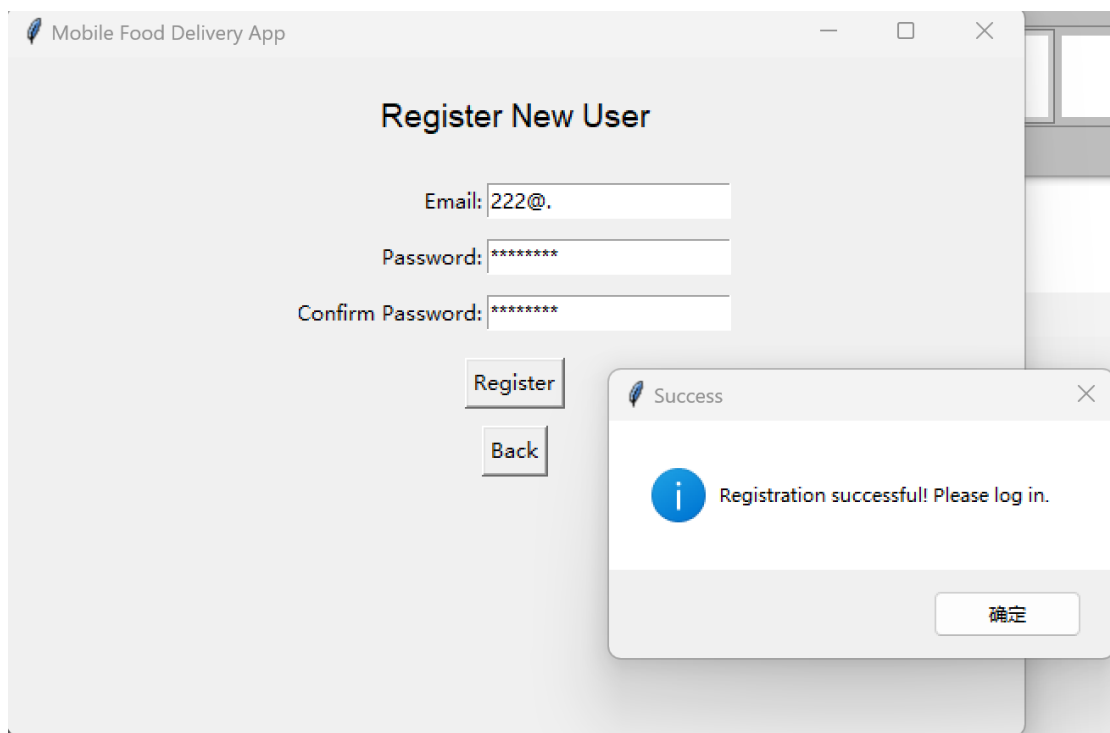


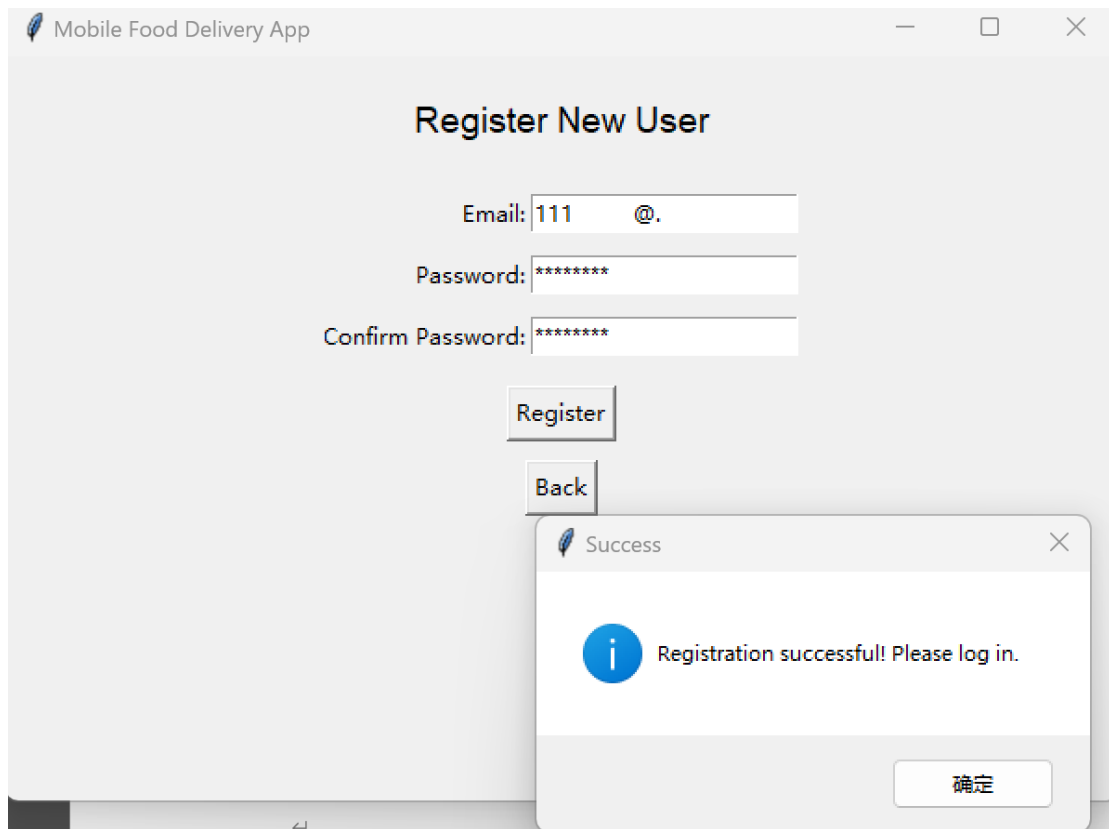
Conclusion: 1. When placing an order, you can enter a number of 0 or even a negative number to place an order, and you can re-place an order, and then an error will occur; Considering that there is no function to cancel the order, the amount required to place the order is detected in the subsequent modification. If the amount is 0 or even negative, it is not allowed to place the order

2.


### Three. Integration tests

In our integration tests, we found the following problems:





1. Incorrect email formats that pass (Spaces allowed in email names/incomplete domains after email)

 Checkout

—

□

×

Review your order:

Burger x114514433543533433457860 = \$1145144335435334368100352.00

Subtotal: \$1145144335435334368100352.00

Tax: \$114514433543533450231808.00

Delivery Fee: \$5.00

Total: \$1259658768978867885441024.00

Delivery Address: 123 Main St

Payment Method:


☒ Credit Card

☐ Paypal


For credit card enter a 16-digit card number:

1234567812345678

Confirm Order

 Order Confirmed

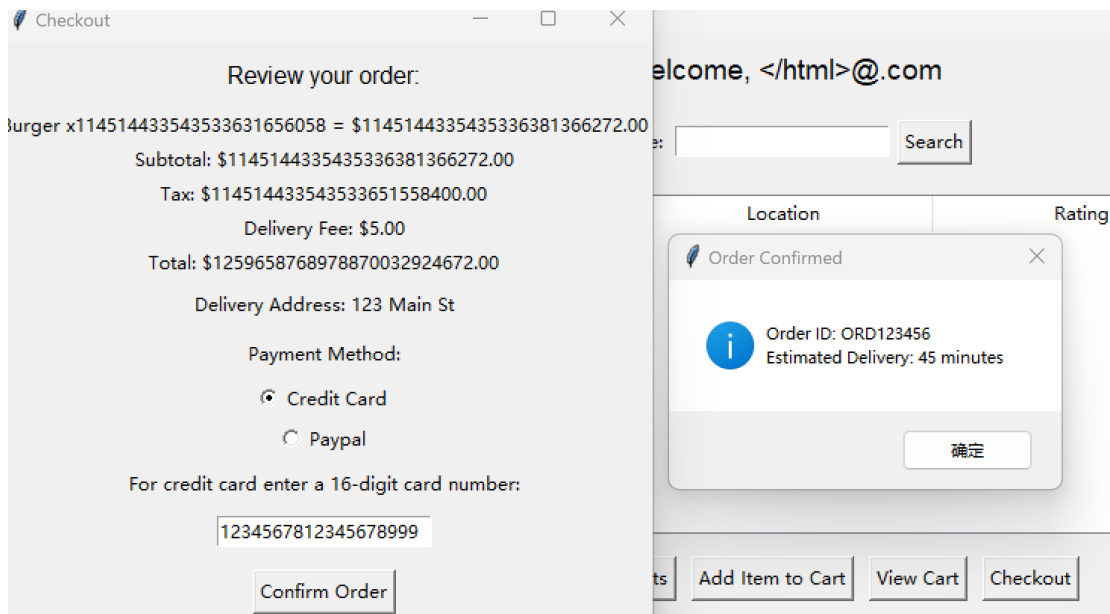
×



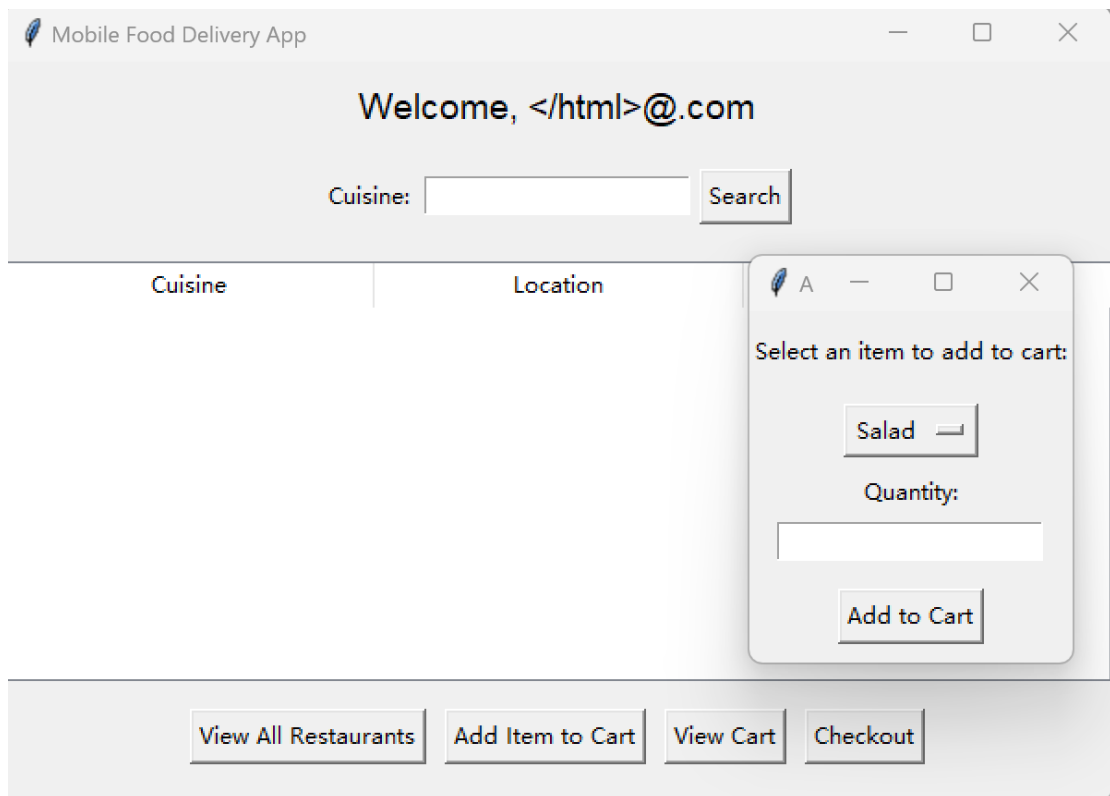
Order ID: ORD123456  
Estimated Delivery: 45 minutes

确定

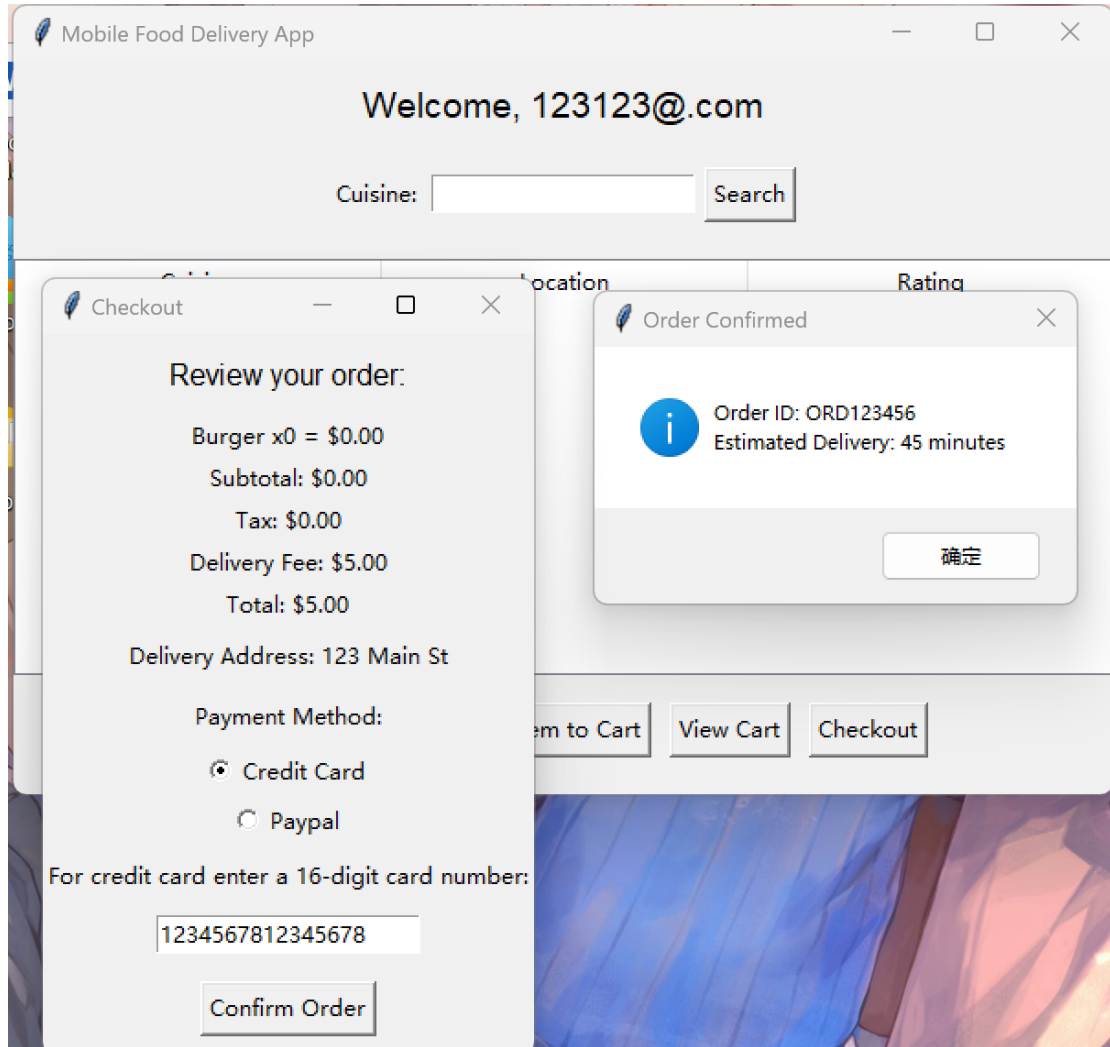




- When entering the bank card number, it does not check whether it is the specified number (too long or short numbers can be successfully placed).



- You can order directly if you don't have a restaurant selected



Mobile Food Delivery App

Checkout

Review your order:

Burger x-1 = \$-10.00

Subtotal: \$-10.00

Tax: \$-1.00

Delivery Fee: \$5.00

Total: \$-6.00

Delivery Address: 123 Main St

Payment Method:

☒ Credit Card

☐ Paypal

For credit card enter a 16-digit card number:

1234567812345678

Confirm Order

123123@.com

Search

Location	Rating
----------	--------

Item to Cart View Cart Checkout

6. You enter a negative number of dishes when ordering, making the amount of money you need to pay a negative number

## 4. Reflect and summarize

After completing this unit test, we have a much deeper understanding of unit testing in Python. By writing the `test_my_module.py` file, you learned how to import the necessary modules, create a test class, and write specific test methods. These steps helped me understand the basic process of unit testing and improved our programming skills.

Reflect on:

1. Code quality: While writing test cases, we found that certain boundary conditions were not covered, such as the case of negative inputs. This suggests that we need to take more careful consideration of various possible boundary cases when writing code.
2. Test coverage: While most of the functionality is covered by the test cases, some complex logical scenarios are not considered. More test cases need to be added in the future to improve the test coverage.

To summarize:

Through this unit testing exercise, we not only solidified our understanding of basic Python syntax and function calls, but also learned how to write and run effective unit tests. In the future development process, we will pay more attention to the improvement of code quality and the increase of test coverage to ensure the stability and reliability of the program. In addition, I also realize that choosing the right testing framework can greatly improve the development efficiency. I hope that in the future study and work, I can constantly optimize



and improve my testing skills and methods.