**LABORATORY WORK #6.**

**Unit testing.**

**Purpose of the work:**learn how to plan and develop unit tests

import unittest

class Book:

def \_\_init\_\_(self, isbn, title, price):

self.isbn = isbn

self.title = title

self.price = price

class Inventory:

def \_\_init\_\_(self):

self.books = {}

self.stock = {}

def add\_book(self, book, quantity):

if len(book.isbn) != 13 or not book.isbn.isdigit():

raise ValueError("Invalid ISBN")

if not book.title.strip():

raise ValueError("Title cannot be empty")

if book.price <= 0 or book.price > 1000:

raise ValueError("Price must be between $0.01 and $1000")

self.books[book.isbn] = book

self.stock[book.isbn] = quantity

def update\_stock(self, isbn, quantity):

if quantity < 0 or quantity > 1000:

raise ValueError("Invalid Stocks Quantity")

self.stock[isbn] = quantity

class SearchEngine:

def search(self, query):

if not query.strip():

raise ValueError("Search query cannot be empty")

if len(query) > 100:

raise ValueError("Search query too long")

if any(char in "!@#$%;" for char in query):

raise ValueError("Invalid search characters")

return ["Result 1", "Result 2"]

class TestOnlineBookstore(unittest.TestCase):

def setUp(self):

self.inventory = Inventory()

self.search\_engine = SearchEngine()

def test\_add\_valid\_book(self):

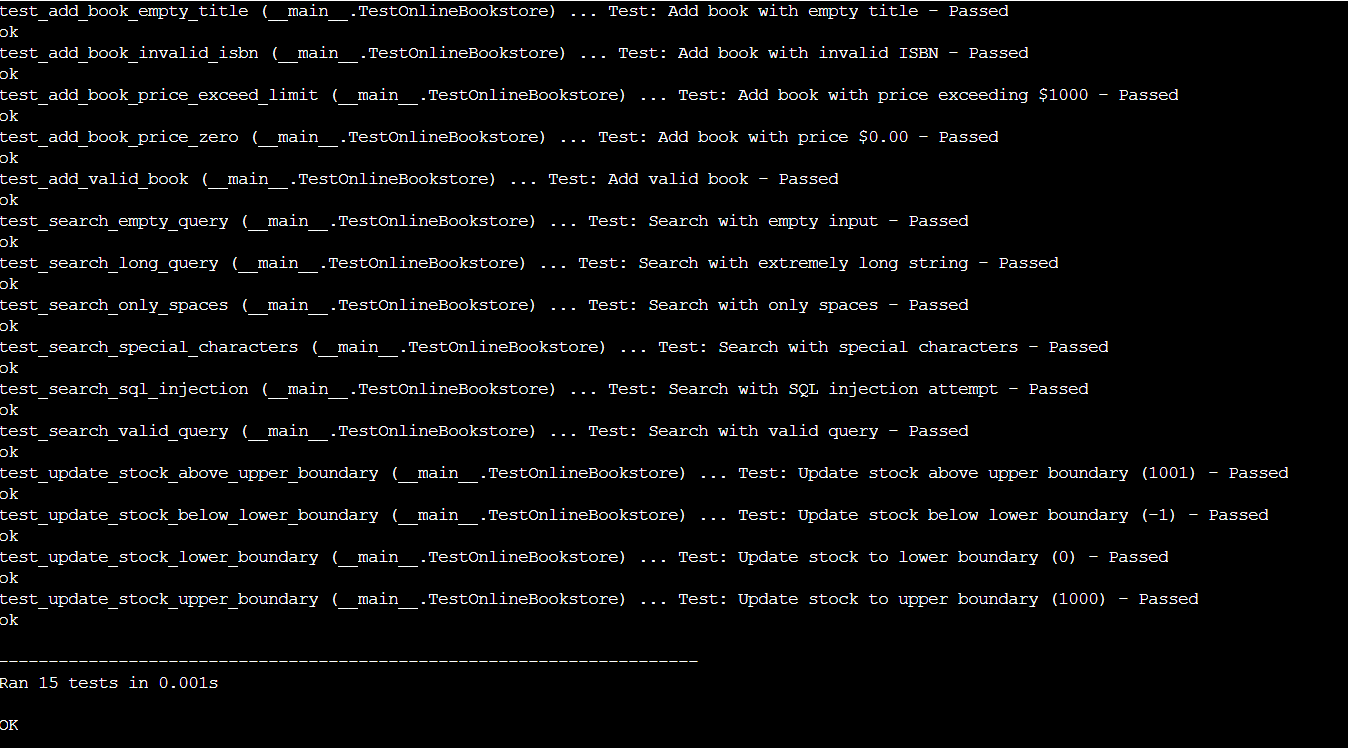
book = Book("9876543212345", "Sample Book", 19.99)

self.inventory.add\_book(book, 10)

self.assertIn(book.isbn, self.inventory.books)

self.assertEqual(self.inventory.stock[book.isbn], 10)

print("Test: Add valid book - Passed")



def test\_add\_book\_invalid\_isbn(self):

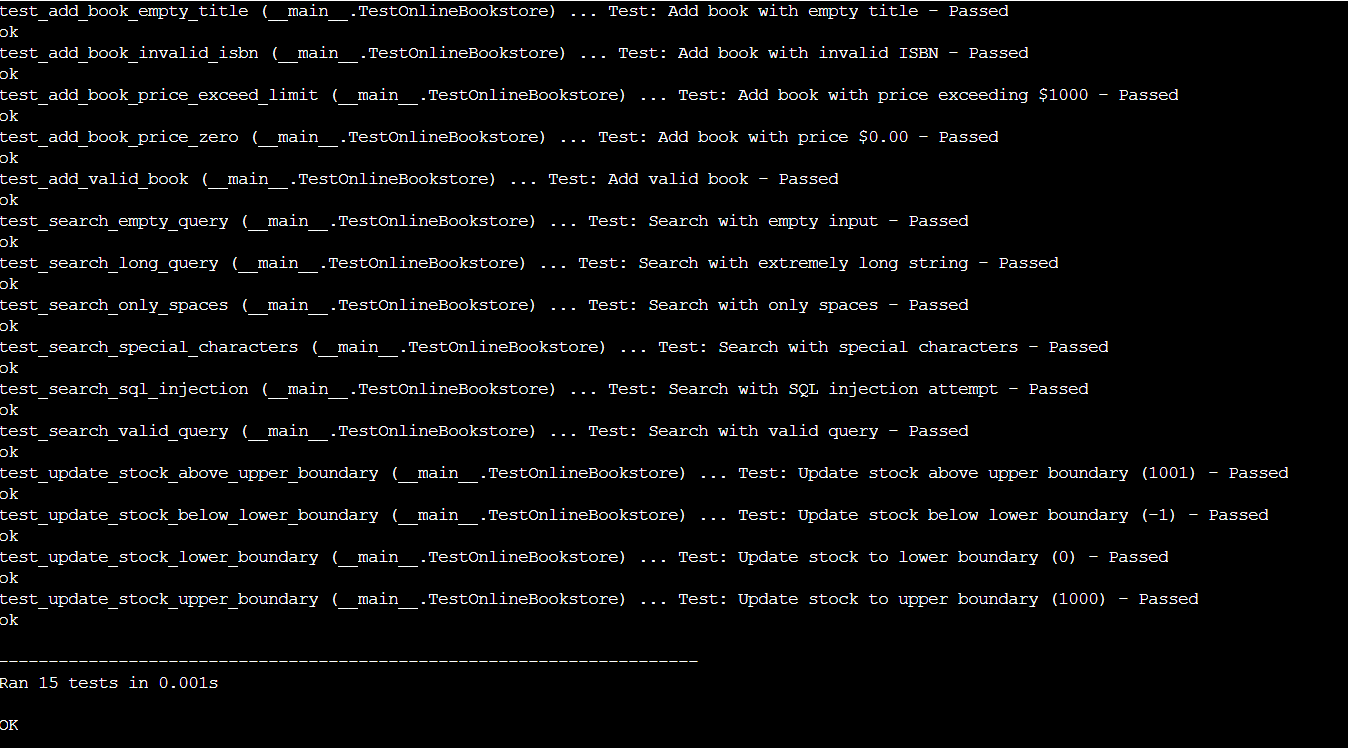
book = Book("12345", "Invalid ISBN Book", 15.00)

with self.assertRaises(ValueError) as context:

self.inventory.add\_book(book, 5)

self.assertIn("Invalid ISBN", str(context.exception))

print("Test: Add book with invalid ISBN - Passed")



def test\_add\_book\_empty\_title(self):

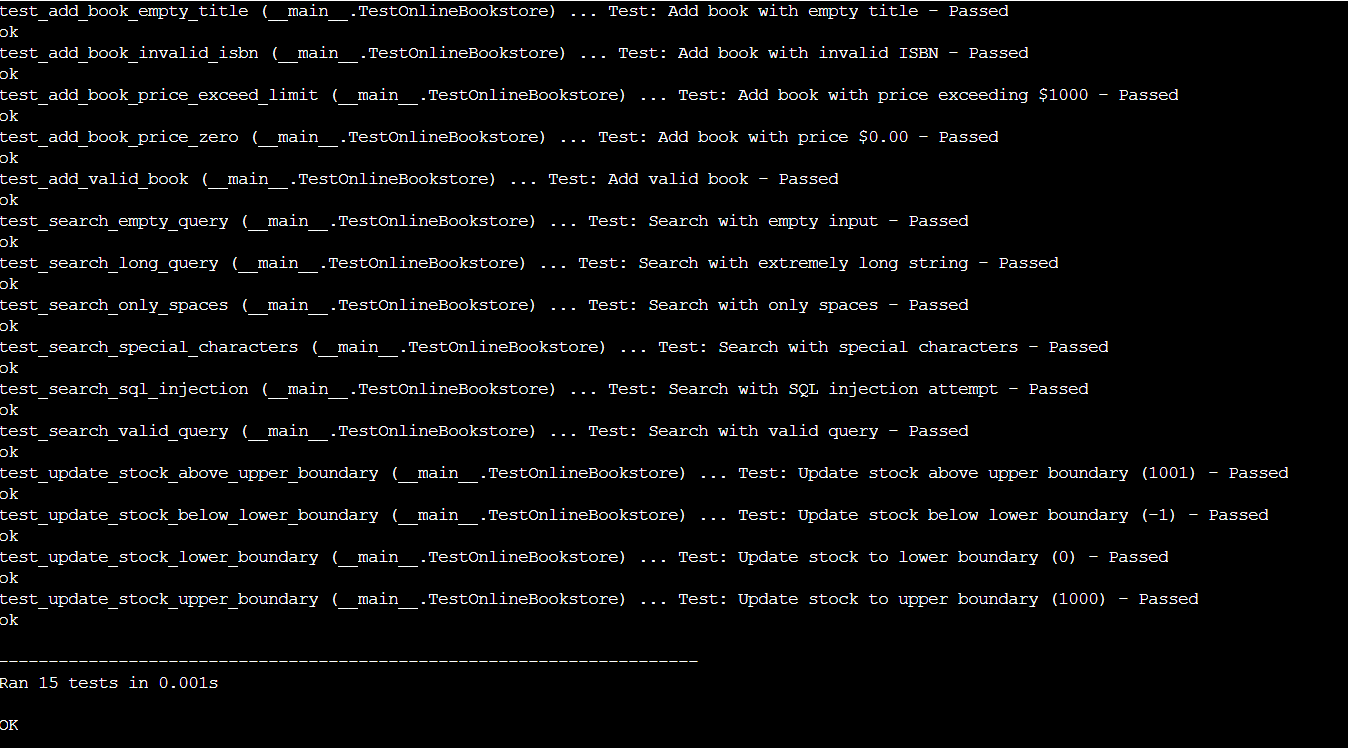
book = Book("9876543212345", " ", 25.00)

with self.assertRaises(ValueError) as context:

self.inventory.add\_book(book, 5)

self.assertIn("Title cannot be empty", str(context.exception))

print("Test: Add book with empty title - Passed")



def test\_add\_book\_price\_zero(self):

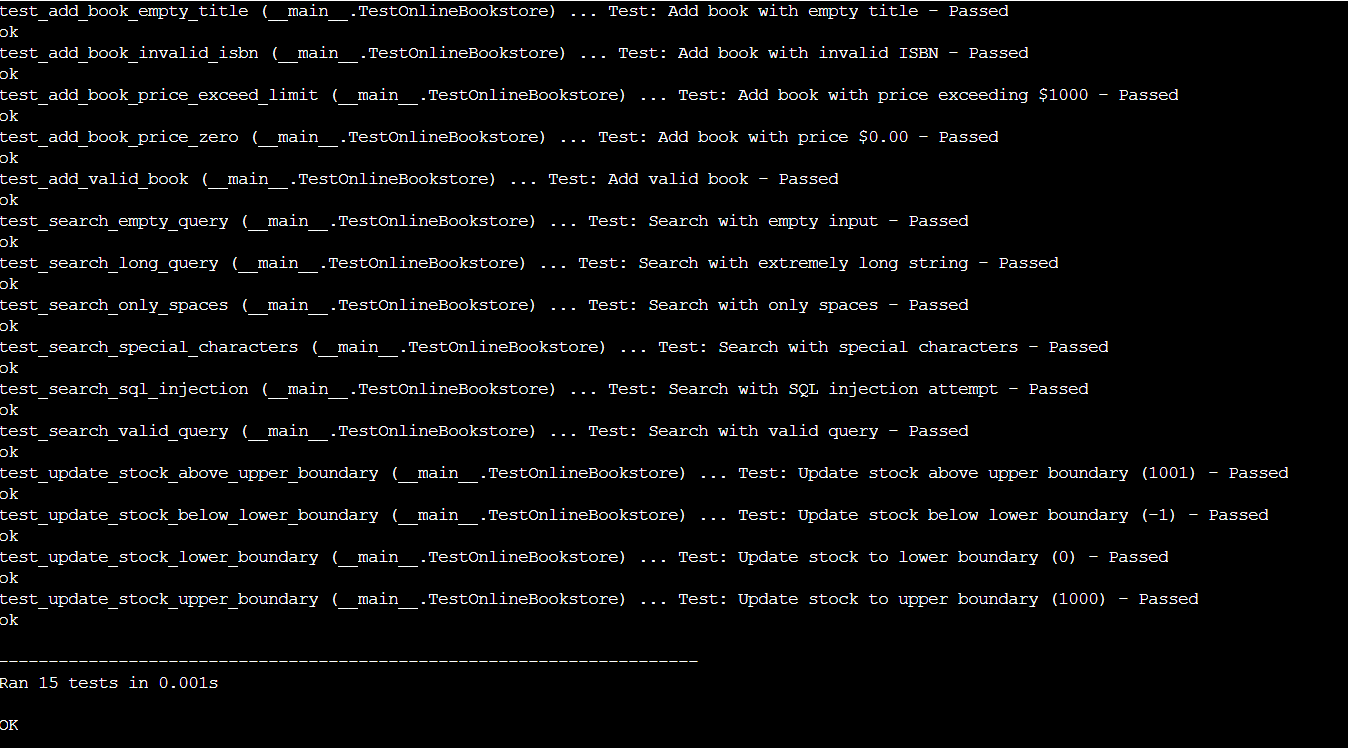
book = Book("9876543212345", "Free Book", 0.00)

with self.assertRaises(ValueError) as context:

self.inventory.add\_book(book, 5)

self.assertIn("Price must be between", str(context.exception))

print("Test: Add book with price $0.00 - Passed")



def test\_add\_book\_price\_exceed\_limit(self):

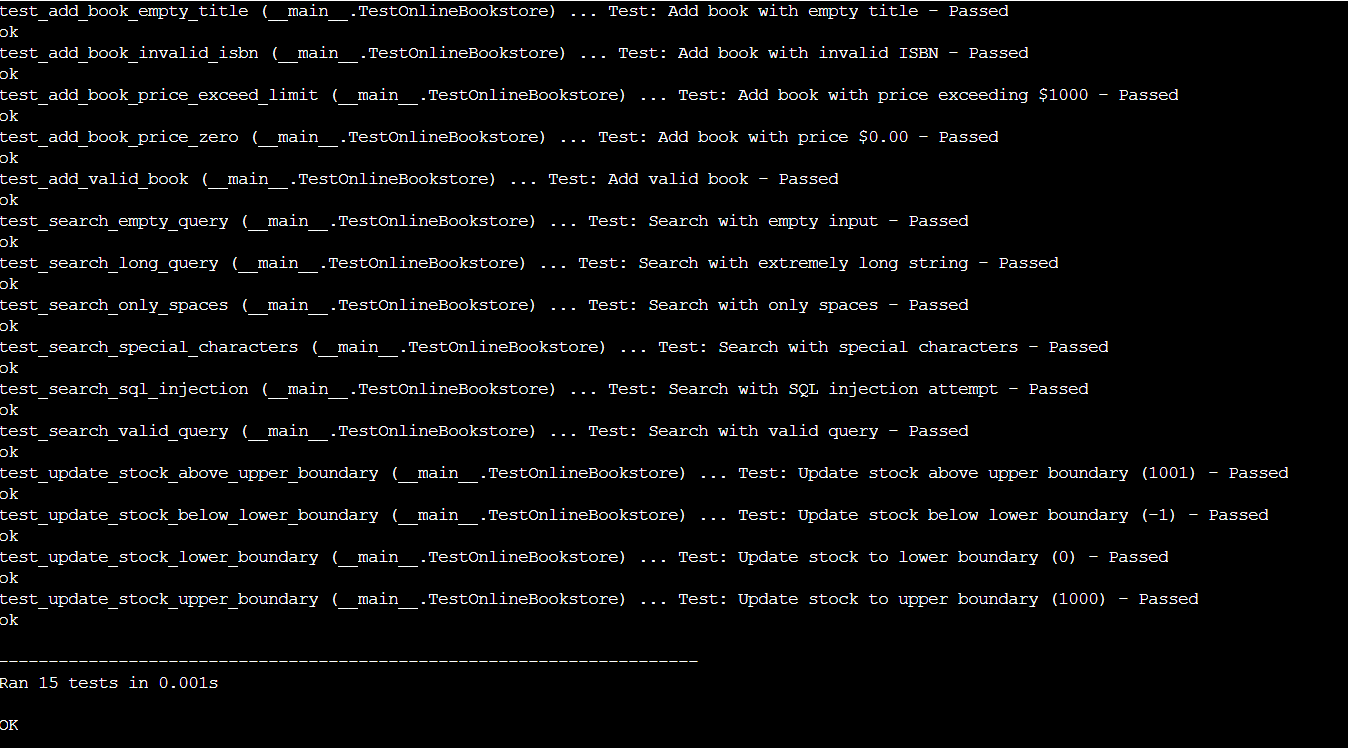
book = Book("9876543212345", "Expensive Book", 1001.00)

with self.assertRaises(ValueError) as context:

self.inventory.add\_book(book, 5)

self.assertIn("Price must be between", str(context.exception))

print("Test: Add book with price exceeding $1000 - Passed")



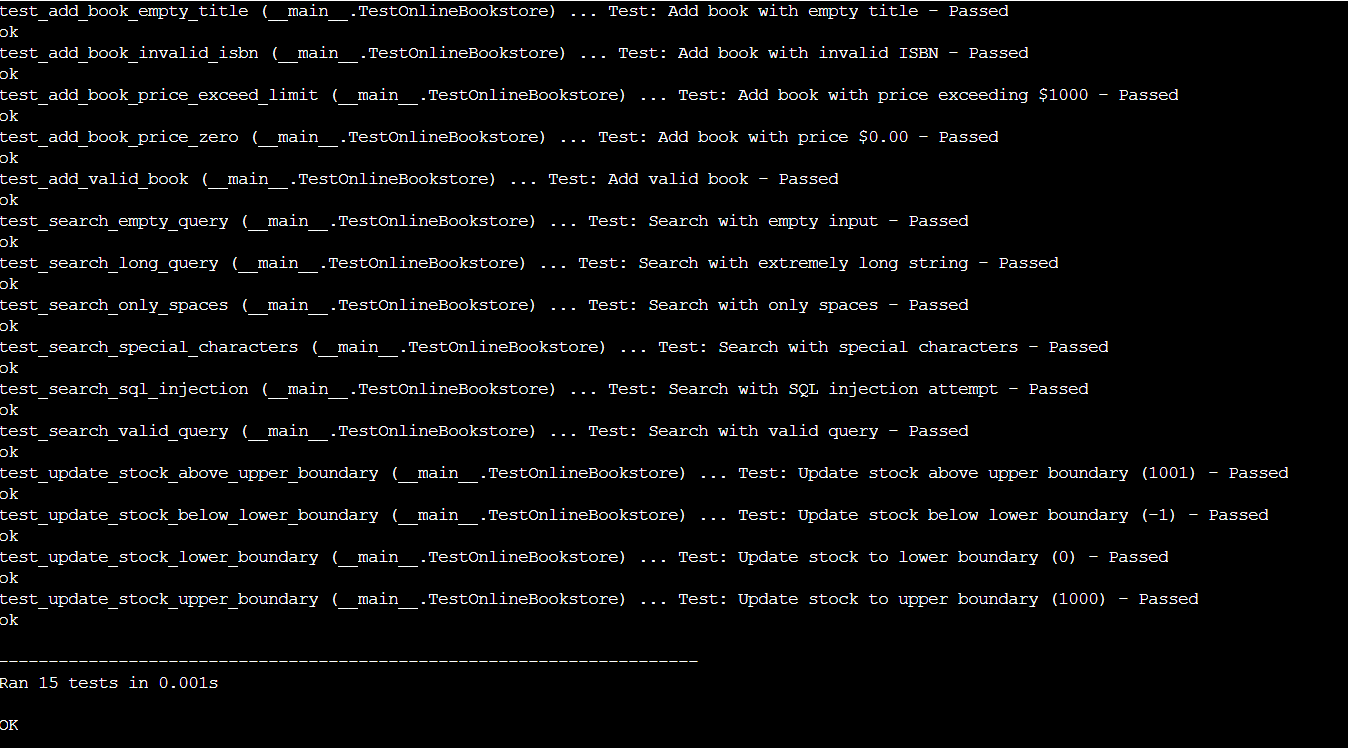
def test\_update\_stock\_lower\_boundary(self):

self.inventory.add\_book(Book("9876543212345", "Test Book", 10.00), 10)

self.inventory.update\_stock("9876543212345", 0)

self.assertEqual(self.inventory.stock["9876543212345"], 0)

print("Test: Update stock to lower boundary (0) - Passed")



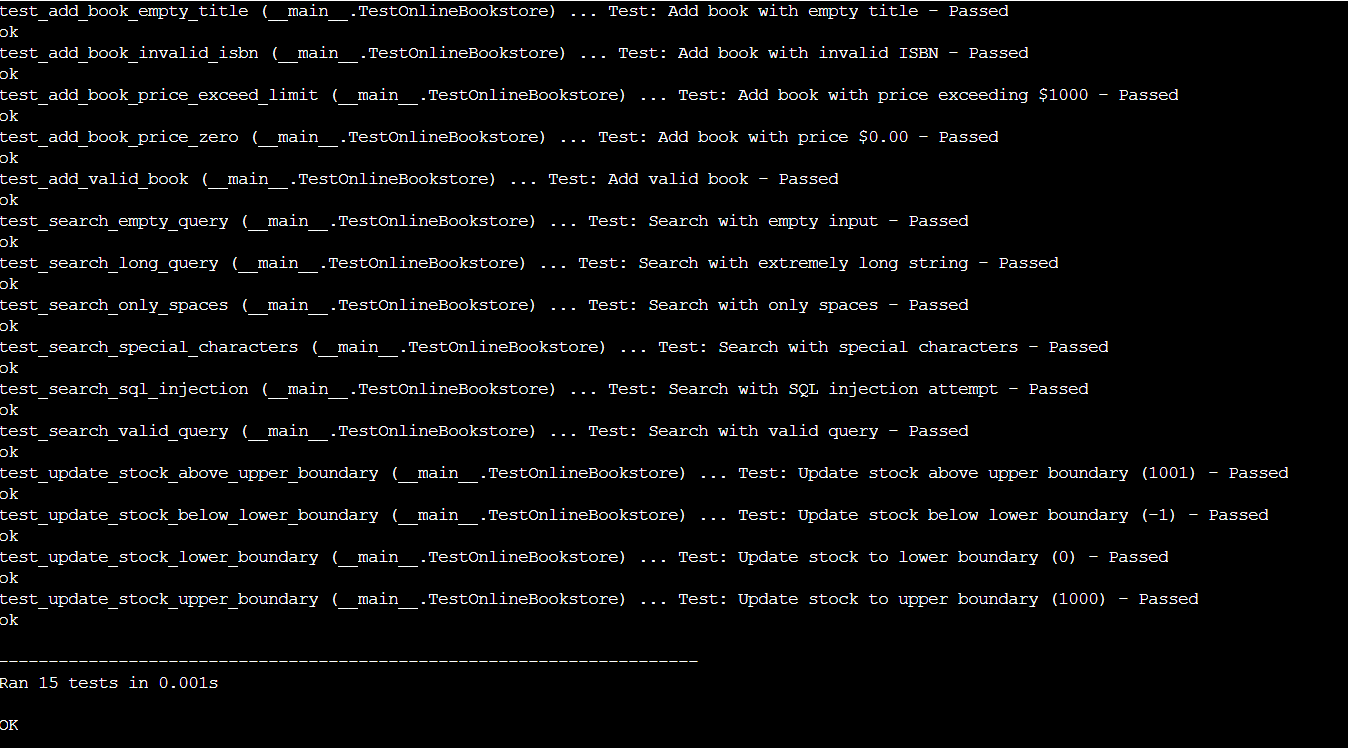
def test\_update\_stock\_upper\_boundary(self):

self.inventory.add\_book(Book("9876543212345", "Test Book", 10.00), 10)

self.inventory.update\_stock("9876543212345", 1000)

self.assertEqual(self.inventory.stock["9876543212345"], 1000)

print("Test: Update stock to upper boundary (1000) - Passed")



def test\_update\_stock\_below\_lower\_boundary(self):

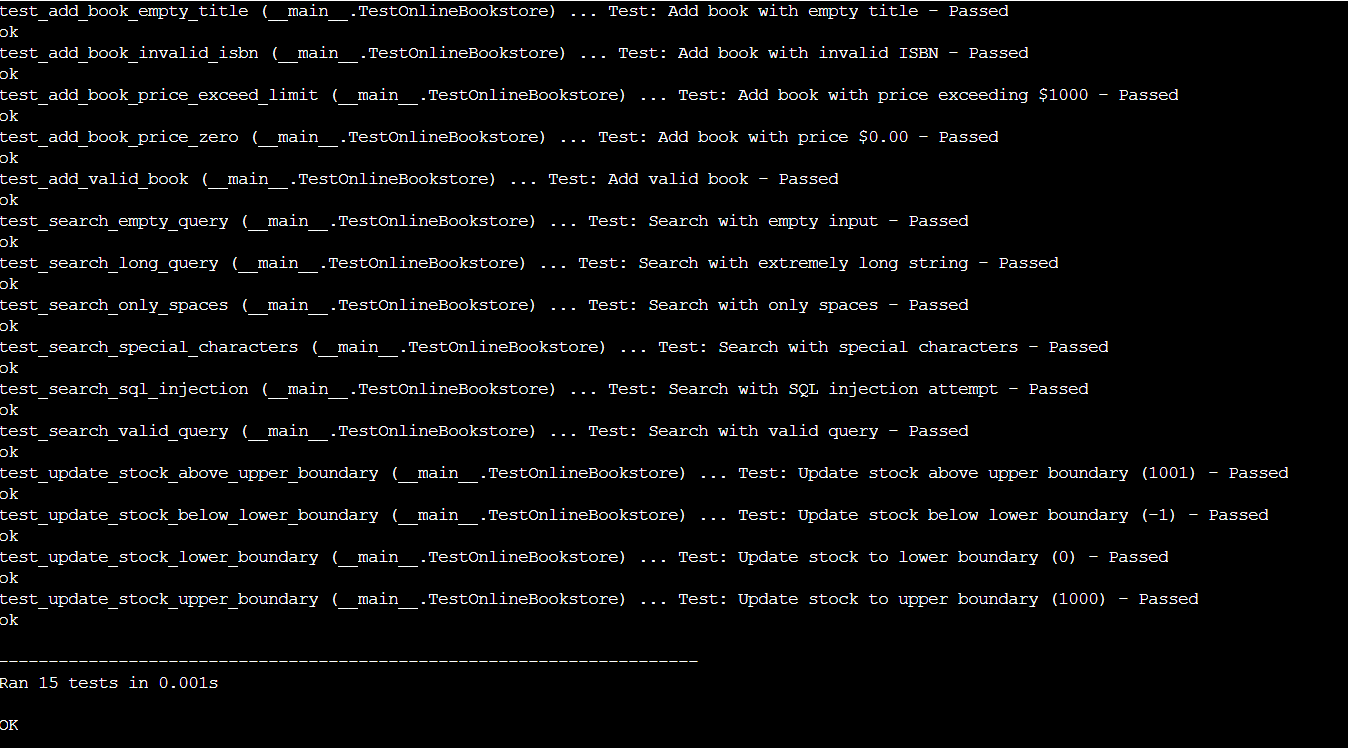
self.inventory.add\_book(Book("9876543212345", "Test Book", 10.00), 10)

with self.assertRaises(ValueError) as context:

self.inventory.update\_stock("9876543212345", -1)

self.assertIn("Invalid Stocks Quantity", str(context.exception))

print("Test: Update stock below lower boundary (-1) - Passed")



def test\_update\_stock\_above\_upper\_boundary(self):

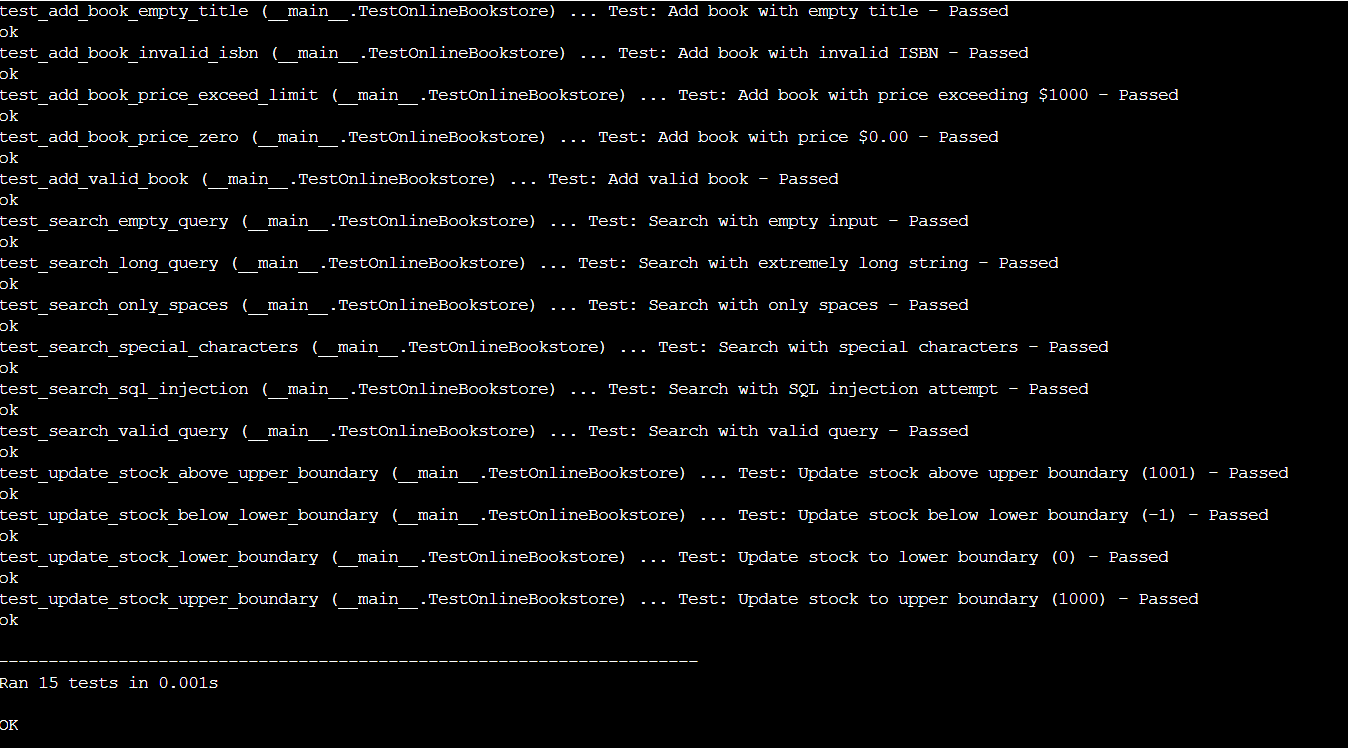
self.inventory.add\_book(Book("9876543212345", "Test Book", 10.00), 10)

with self.assertRaises(ValueError) as context:

self.inventory.update\_stock("9876543212345", 1001)

self.assertIn("Invalid Stocks Quantity", str(context.exception))

print("Test: Update stock above upper boundary (1001) - Passed")



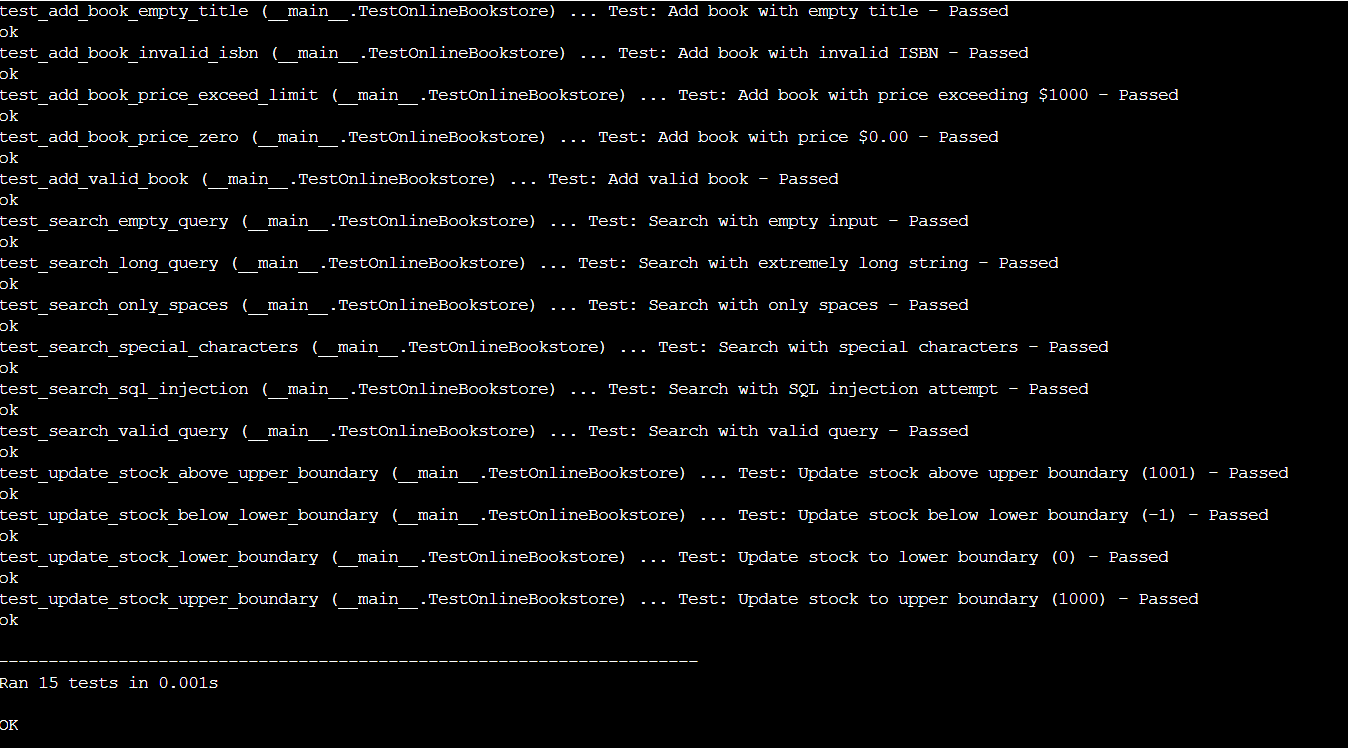
def test\_search\_valid\_query(self):

results = self.search\_engine.search("valid query")

self.assertIsInstance(results, list)

self.assertTrue(len(results) > 0)

print("Test: Search with valid query - Passed")



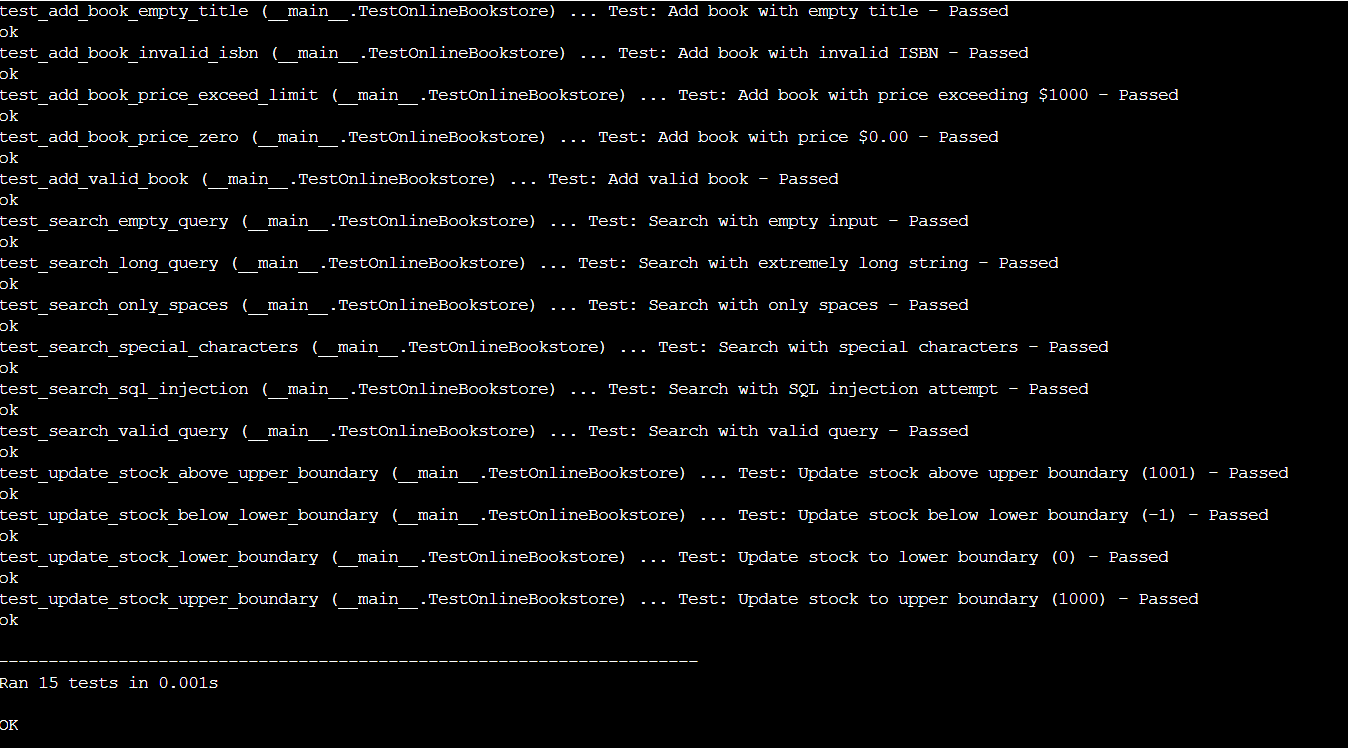
def test\_search\_special\_characters(self):

with self.assertRaises(ValueError) as context:

self.search\_engine.search("!@#$%")

self.assertIn("Invalid search characters", str(context.exception))

print("Test: Search with special characters - Passed")



def test\_search\_long\_query(self):

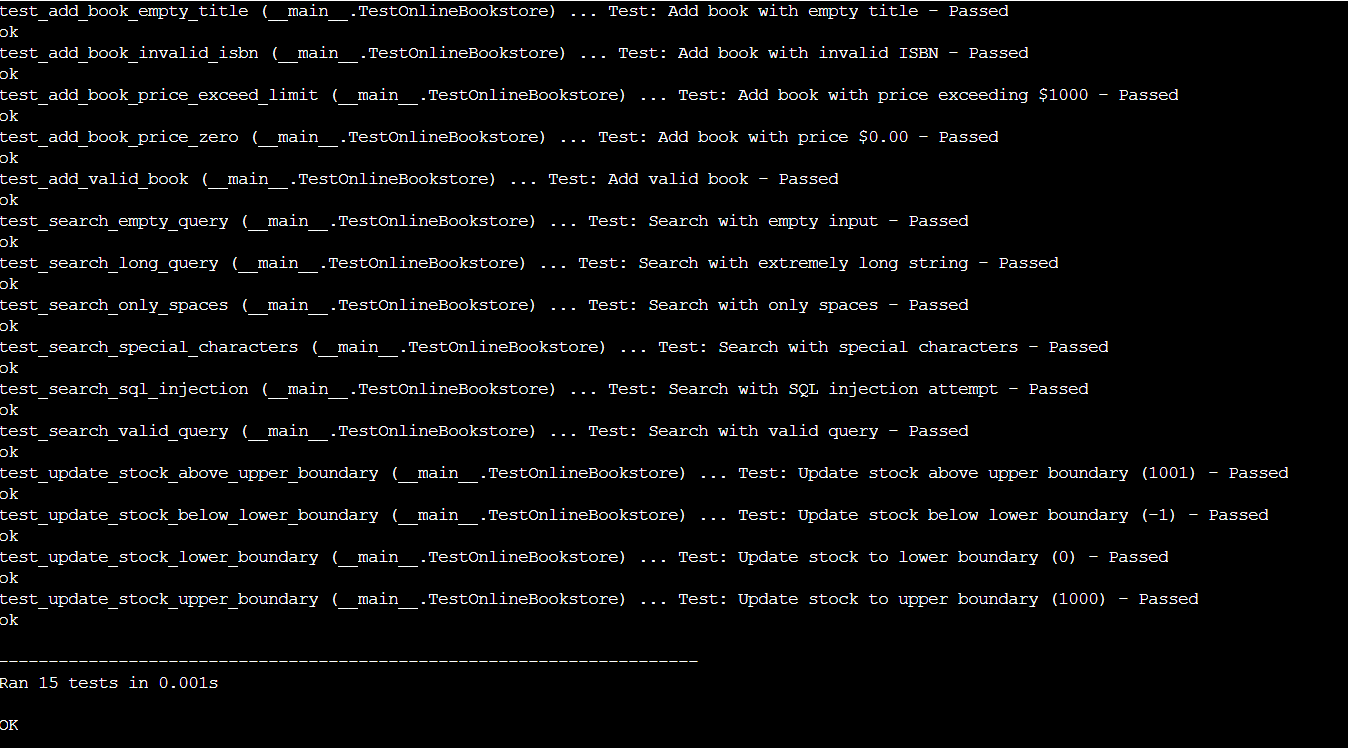
long\_query = "a" \* 101

with self.assertRaises(ValueError) as context:

self.search\_engine.search(long\_query)

self.assertIn("Search query too long", str(context.exception))

print("Test: Search with extremely long string - Passed")



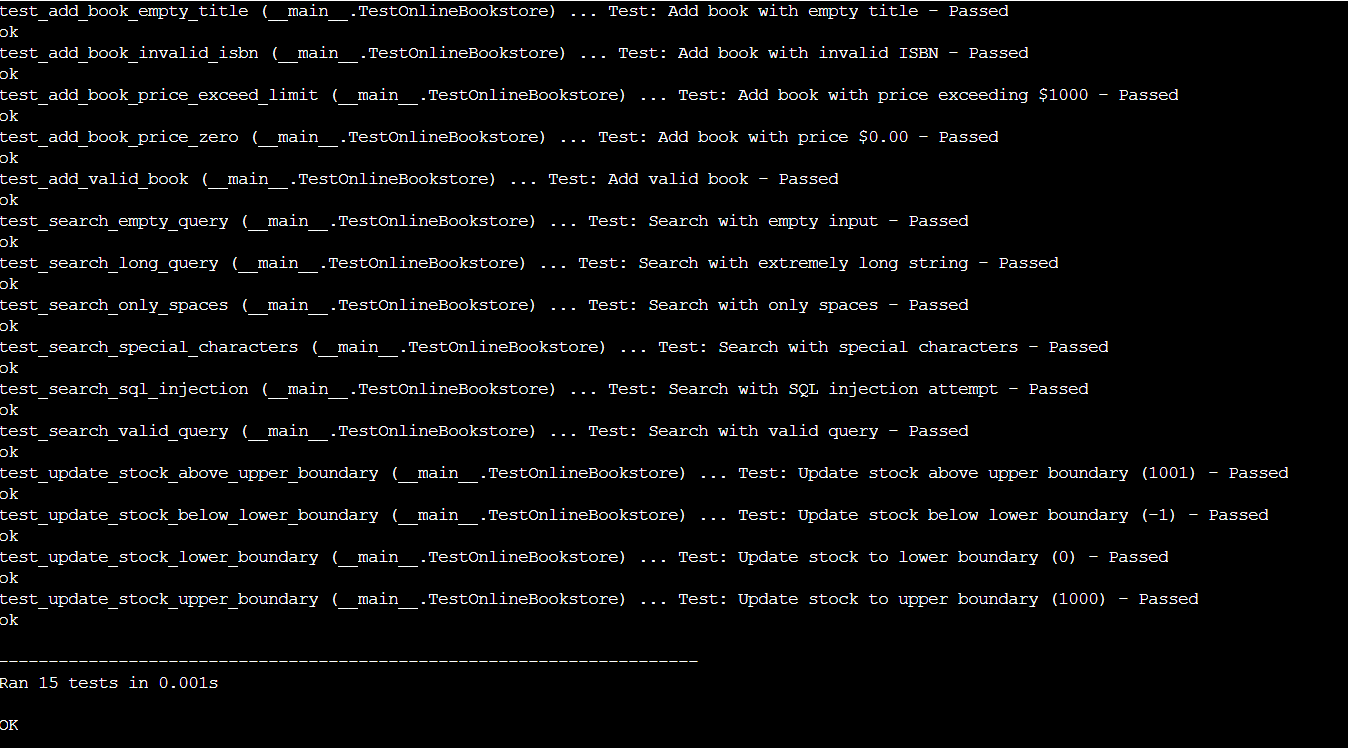
def test\_search\_sql\_injection(self):

with self.assertRaises(ValueError) as context:

self.search\_engine.search("; DROP TABLE BOOKS")

self.assertIn("Invalid search characters", str(context.exception))

print("Test: Search with SQL injection attempt - Passed")



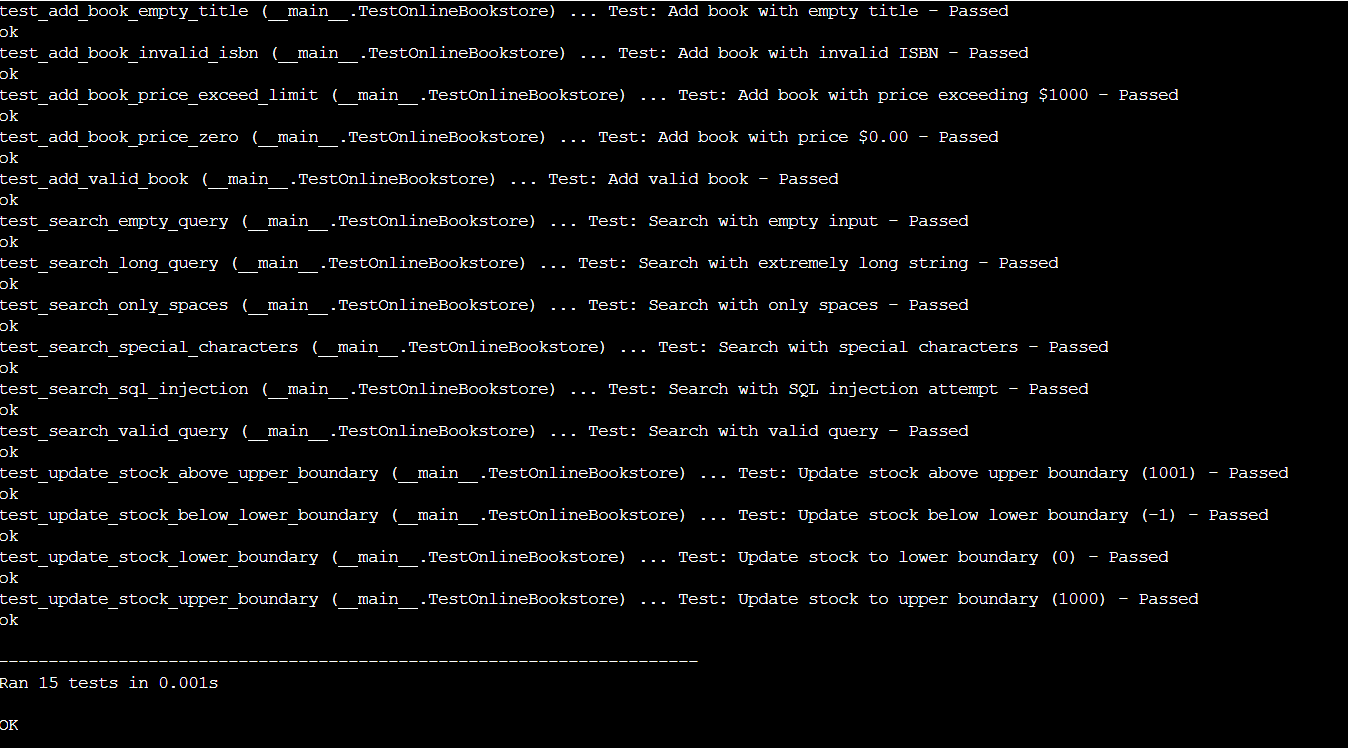
def test\_search\_empty\_query(self):

with self.assertRaises(ValueError) as context:

self.search\_engine.search("")

self.assertIn("Search query cannot be empty", str(context.exception))

print("Test: Search with empty input - Passed")



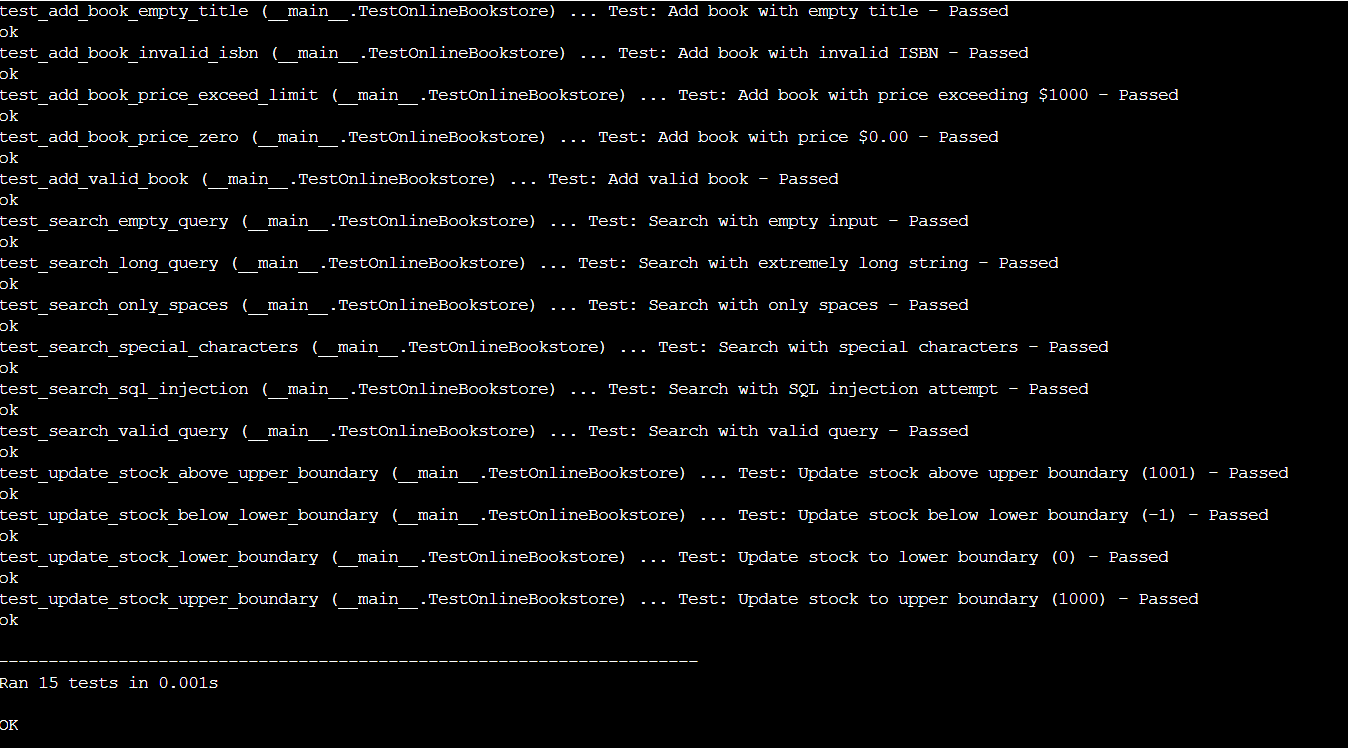
def test\_search\_only\_spaces(self):

with self.assertRaises(ValueError) as context:

self.search\_engine.search(" ")

self.assertIn("Search query cannot be empty", str(context.exception))

print("Test: Search with only spaces - Passed")



if \_\_name\_\_ == '\_\_main\_\_':

unittest.main(verbosity=2)

