# shopping\_cart.py

class Product:

def \_\_init\_\_(self, product\_id, name, price):

self.product\_id = product\_id

self.name = name

self.price = price

def \_\_repr\_\_(self):

return f"Product({self.product\_id}, {self.name}, {self.price})"

class Cart:

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

# Dictionary mapping product\_id to a tuple (Product, quantity)

self.items = {}

def add\_product(self, product, quantity=1):

if quantity <= 0:

raise ValueError("Quantity must be positive")

if product.product\_id in self.items:

current\_qty = self.items[product.product\_id][1]

self.items[product.product\_id] = (product, current\_qty + quantity)

else:

self.items[product.product\_id] = (product, quantity)

def remove\_product(self, product, quantity=1):

if product.product\_id not in self.items:

raise ValueError("Product not in cart")

if quantity <= 0:

raise ValueError("Quantity must be positive")

current\_qty = self.items[product.product\_id][1]

if quantity >= current\_qty:

del self.items[product.product\_id]

else:

self.items[product.product\_id] = (product, current\_qty - quantity)

def calculate\_total(self):

total = 0.0

for product, quantity in self.items.values():

total += product.price \* quantity

return total

def apply\_discount(self, discount\_rate):

if discount\_rate < 0 or discount\_rate > 100:

raise ValueError("Discount rate must be between 0 and 100")

total = self.calculate\_total()

discount\_amount = total \* (discount\_rate / 100.0)

return total - discount\_amount

class Order:

def \_\_init\_\_(self, cart, customer\_name):

self.cart = cart

self.customer\_name = customer\_name

self.total\_amount = cart.calculate\_total()

self.status = "Pending"

def process\_order(self):

if self.total\_amount <= 0:

raise ValueError("Cannot process order with zero total")

self.status = "Processed"

return True

class Inventory:

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

# Dictionary mapping product\_id to available quantity

self.stock = {}

def add\_stock(self, product, quantity):

if quantity <= 0:

raise ValueError("Quantity must be positive")

if product.product\_id in self.stock:

self.stock[product.product\_id] += quantity

else:

self.stock[product.product\_id] = quantity

def remove\_stock(self, product, quantity):

if product.product\_id not in self.stock or self.stock[product.product\_id] < quantity:

raise ValueError("Insufficient stock")

self.stock[product.product\_id] -= quantity

def check\_stock(self, product):

return self.stock.get(product.product\_id, 0)

class Coupon:

def \_\_init\_\_(self, code, discount\_rate):

if discount\_rate < 0 or discount\_rate > 100:

raise ValueError("Invalid discount rate")

self.code = code

self.discount\_rate = discount\_rate

def apply\_coupon(self, cart):

return cart.apply\_discount(self.discount\_rate)

Test Cases:

# TestA

import timeit

import concurrent.futures

from shopping\_cart import Product, Cart

def test\_calculate\_total():

cart = Cart()

product = Product(1, "Widget", 10.0)

# Populate the cart with 1000 units of the same product.

for \_ in range(1000):

cart.add\_product(product, 1)

# Measure the execution time of calculate\_total() over 1000 iterations.

total\_time = timeit.timeit(lambda: cart.calculate\_total(), number=1000)

print(f"Test A: 'calculate\_total()' took {total\_time:.6f} seconds over 1000 iterations.")

# TestB

def add\_and\_remove(cart, product, iterations):

for \_ in range(iterations):

try:

cart.add\_product(product, 1)

cart.remove\_product(product, 1)

except ValueError:

# In a race condition, removal might occur on a non-existent product.

pass

# Return the final total from the cart.

return cart.calculate\_total()

def test\_cart\_operations():

cart = Cart()

product = Product(1, "Widget", 10.0)

iterations\_per\_thread = 1000

num\_threads = 50 # Simulate 50 concurrent threads

with concurrent.futures.ThreadPoolExecutor(max\_workers=num\_threads) as executor:

futures = [executor.submit(add\_and\_remove, cart, product, iterations\_per\_thread)

for \_ in range(num\_threads)]

results = [future.result() for future in concurrent.futures.as\_completed(futures)]

# Ideally, if operations are balanced, the final cart total should be 0.

print("Test B: Final totals reported by threads:", results)

# -------------------------

# Main Execution

# -------------------------

if \_\_name\_\_ == "\_\_main\_\_":

print("Running Test A:")

test\_calculate\_total()

print("\nRunning Test B:")

test\_cart\_operations()