**Online Shopping Cart**

# 1. Project Description

Develop a console-based application that simulates the core functionality of an online shopping cart. Users of the application (simulating customers) will be able to:

* View a list of available products.
* Add products to their shopping cart.
* Update the quantity of items already in the cart.
* Remove items from the cart.
* View the current contents of their cart, including subtotal for each item and the grand total.
* Persist the product catalog and the shopping cart's state to files, so data is saved between application runs.

This project heavily utilizes Object-Oriented Programming (OOP) principles, focusing on inheritance for product types, encapsulation, and object composition.

# 2. Core OOP Concepts to Apply

* **Class & Object Design:** Model real-world entities (Product, CartItem, ShoppingCart) as Python classes.
* **Encapsulation:** Use private-like attributes (e.g., \_name, \_price) and expose them via public methods or @property decorators, ensuring data integrity.
* **Constructors (\_\_init\_\_):** Properly initialize object states.
* **Instance Methods:** Define behaviors specific to an object instance.
* **Inheritance:** Create a base Product class and derive specific product types (e.g., PhysicalProduct, DigitalProduct) to demonstrate polymorphism.
* **Polymorphism:** Use a common interface (e.g., display\_details() method) that behaves differently based on the actual product type.
* **Object Composition:** ShoppingCart will *contain* CartItem objects, and CartItem will *contain* a Product object.
* **Magic Methods (\_\_str\_\_, \_\_repr\_\_):** Implement these for clear object representation when printing or debugging.
* **Error Handling:** Use try-except blocks for user input validation and file operations, especially stock management.

# 3. Class Specifications & Coder Tasks

You are required to implement the following classes with the specified attributes and methods.

**3.1. Product Class (Base Class)**

Represents a generic product available for purchase.

* **Attributes (Private-like for encapsulation):**

* 1. \_product\_id: str - A unique identifier for the product.

○ \_name: str - The name of the product.

○ \_price: float - The price of the product per unit.

○ \_quantity\_available: int - The current stock quantity.

* **Properties:**

* 1. product\_id (read-only): Returns \_product\_id.

○ name (read-only): Returns \_name.

○ price (read-only): Returns \_price.

○ quantity\_available (read/write): Returns \_quantity\_available. The

setter should enforce non-negative values.

* **Methods:**

* 1. \_\_init\_\_(self, product\_id: str, name: str, price: float, quantity\_available: int):

■ Constructor. Initializes all attributes.

○ decrease\_quantity(self, amount: int) -> bool:

■ Decreases \_quantity\_available by amount. Returns True if successful, False if amount is invalid or stock is insufficient.

○ increase\_quantity(self, amount: int) -> None:

■ Increases \_quantity\_available by amount.

○ display\_details(self) -> str:

■ Returns a basic string detailing the product (ID, name, price, available quantity). This method will be overridden in child classes for polymorphism.

○ to\_dict(self) -> dict:

■ Returns a dictionary representation of the product's attributes, including a type key to distinguish between Product, PhysicalProduct, DigitalProduct for saving/loading.

## 3.2. PhysicalProduct Class

Inherits from Product, representing tangible goods.

* **Attributes (Specific to PhysicalProduct):**

* 1. \_weight: float - Weight in kilograms.
* **Properties:**

* 1. weight (read-only): Returns \_weight.
* **Methods:**

* 1. \_\_init\_\_(self, product\_id: str, name: str, price: float, quantity\_available: int, weight: float):

■ Constructor. Calls parent \_\_init\_\_ and initializes \_weight. ○ display\_details(self) -> str:

■ Overrides the parent method. Returns a string including product ID, name, price, available quantity, and weight.

○ to\_dict(self) -> dict:

■ Overrides parent. Returns dictionary including \_weight and type: 'physical'.

## 3.3. DigitalProduct Class

Inherits from Product, representing intangible goods (e.g., software, e-books).

* **Attributes (Specific to DigitalProduct):**

○ \_download\_link: str - A placeholder for a download URL.

* **Properties:**

○ download\_link (read-only): Returns \_download\_link.

* **Methods:**

○\_\_init\_\_(self, product\_id: str, name: str, price: float, quantity\_available: int, download\_link: str):

■ Constructor. Calls parent \_\_init\_\_ and initializes \_download\_link. ○ display\_details(self) -> str:

■ Overrides the parent method. Returns a string including product ID, name, price, and download link (as a placeholder).

○ to\_dict(self) -> dict:

■ Overrides parent. Returns dictionary including \_download\_link and type: 'digital'.

**3.4. CartItem Class**

Represents a single product with a specific quantity in the shopping cart.

* **Attributes (Private-like for encapsulation):**

○ \_product: Product - A reference to the Product object.

○ \_quantity: int - The quantity of this product in the cart.

* **Properties:**

○ product (read-only): Returns \_product.

○ quantity (read/write): Returns \_quantity. The setter should enforce non-negative values.

* **Methods:**

○ \_\_init\_\_(self, product: Product, quantity: int):

■ Constructor. Initializes \_product and \_quantity.

○ calculate\_subtotal(self) -> float:

■ Returns the subtotal for this cart item (product.price \* quantity). ○ \_\_str\_\_(self) -> str:

■ Returns a user-friendly string representation, e.g., "Item: [Product Name], Quantity: [Qty], Price: $[Price], Subtotal:

$[Subtotal]".

○ to\_dict(self) -> dict:

■ Returns a dictionary representation of the CartItem, including product\_id and quantity, useful for saving to file.

## 3.5. ShoppingCart Class

Manages the collection of CartItem objects and provides shopping cart functionalities.

* **Attributes (Private-like for encapsulation):**

* 1. \_items: dict[str, CartItem] - A dictionary where keys are product\_id

(from the Product object) and values are CartItem objects.

○ \_product\_catalog\_file: str - Filename for saving/loading product data (e.g., 'products.json').

○ \_cart\_state\_file: str - Filename for saving/loading shopping cart state (e.g., 'cart.json').

* **Methods:**

* 1. \_\_init\_\_(self, product\_catalog\_file: str = 'products.json', cart\_state\_file: str = 'cart.json'):

■ Initializes empty \_items dictionary.

■ Sets data file names.

■ Calls \_load\_catalog() to load available products.

■ Calls \_load\_cart\_state() to load previous cart state.

○ \_load\_catalog(self) -> dict[str, Product]: (Private Helper Method)

■ Loads product data from \_product\_catalog\_file.

■ Handles FileNotFoundError.

■ **Important:** Reconstructs Product objects (including

PhysicalProduct, DigitalProduct) based on their type field from

the loaded dictionary data.

■ Returns the loaded product catalog.

○ \_load\_cart\_state(self) -> None: (Private Helper Method)

■ Loads cart item data from \_cart\_state\_file.

■ Handles FileNotFoundError.

■ **Important:** Reconstructs CartItem objects, referencing the correct

Product objects from the loaded catalog.

○\_save\_catalog(self, catalog: dict[str, Product]) -> None:

(Private Helper Method)

■ Saves the given catalog to \_product\_catalog\_file. Iterates and calls to\_dict() on each Product.

○ \_save\_cart\_state(self) -> None: (Private Helper Method)

■ Saves the current \_items (shopping cart content) to

\_cart\_state\_file. Iterates and calls to\_dict() on each CartItem. ○add\_item(self, product\_id: str, quantity: int) -> bool:

■ Checks if the product exists in the loaded catalog and if there's enough stock.

■ If successful, adds a new CartItem or updates the quantity of an existing one. Decreases product stock.

■ Returns True on success, False otherwise. Calls \_save\_cart\_state().

○ remove\_item(self, product\_id: str) -> bool:

■ Removes a CartItem entirely from the cart. Increases product stock back.

■ Returns True if removed, False if not found. Calls \_save\_cart\_state().

○update\_quantity(self, product\_id: str, new\_quantity: int) -> bool:

■ Updates the quantity of an existing CartItem. Adjusts product stock accordingly (difference between old and new quantity).

■ Handles invalid new\_quantity (e.g., negative, exceeding available stock).

■ Returns True on success, False otherwise. Calls \_save\_cart\_state().

○ get\_total(self) -> float:

■ Calculates and returns the grand total of all items in the cart.

○ display\_cart(self) -> None:

■ Prints a formatted list of all items in the cart, their quantities, subtotals, and the grand total.

○ display\_products(self) -> None:

■ Prints details of all available products in the catalog using their display\_details() method (demonstrating polymorphism).

# 4. Coder Tasks (Implementation Steps)

1. **Set up Project Structure:** Create shopping\_cart.py (or similar).
2. **Implement Product Class:** Write the base Product class with all specified attributes, properties, and methods.
3. **Implement PhysicalProduct and DigitalProduct Classes:** Implement these classes, ensuring they correctly inherit from Product and override display\_details() and to\_dict().
4. **Implement CartItem Class:** Write the CartItem class with its attributes, properties, and methods.
5. **Implement ShoppingCart Class (Core Logic):** 
   1. Implement \_\_init\_\_, \_load\_catalog, \_load\_cart\_state, \_save\_catalog, \_save\_cart\_state.

■ **Crucial for \_load\_catalog and \_load\_cart\_state:** When loading data from JSON, you'll need to reconstruct Product and CartItem objects from their dictionary representations. For Product, use the type field to decide whether to create a Product, PhysicalProduct, or DigitalProduct instance. For CartItem, you'll need to look up the

Product object from your loaded catalog.

○ Implement add\_item, remove\_item, update\_quantity. These methods must interact correctly with the stock (\_quantity\_available) of the Product objects in the catalog.

○ Implement get\_total, display\_cart, display\_products.

1. **Create Console Interface:** 
   1. Develop a main() function or a run() method in the ShoppingCart class.

○ Implement a main while loop that presents a menu to the user (e.g., "1. View

Products", "2. Add Item to Cart", "3. View Cart", "4. Update Quantity", "5. Remove Item", "6. Checkout (dummy)", "7. Exit").

○ Use input() to get user choices and data.

○ Call the appropriate ShoppingCart methods based on user input.

○ Include clear print() statements for feedback and displaying results.