Programming 2 Midterm Part 1

Allowed Materials

This exam is closed book including other source code on your computer, and closed Internet.

The following resources are allowed:

Python Library documentation on your computer or at https://docs.python.org/3/library/

Github to view your exam repo, but no other Github repositories.

Procedure

1. Disconnect any auxiliary monitors. Use only one monitor during the exam.
2. Join the TAs voice channel on Discord and share your screen. Don't watch anyone else's screen.
3. Start recording a video using OBS or Webex.
4. Accept the assignment on Github and do it. https://classroom.github.com/a/wpN1hh1W
5. Submit a link to your video on this form: https://forms.gle/ZPYUUXgL2dN9oUcp9

Good luck.

What To Submit

1. Commit your work and push to the remote repository (on Github). Be careful to include all the Python files you add (sale.py, lineitem.py, test_lineitem.py) and files you modify. It's a good idea to check that Github has your final submission.

2. After you finish, rename your video file as Yourname-labexam1.mkv, upload to Google Drive, and share with these people:

j.brucker@ku.th

poomtum.r@ku.th

thanatibordee.s@ku.th

vitvara.v@ku.th

nabhan.s@ku.th

3. Please inform the TA monitoring your exam when you are done. Thanks.

TAs: Please record finish time on Google sheet.

Problem Description

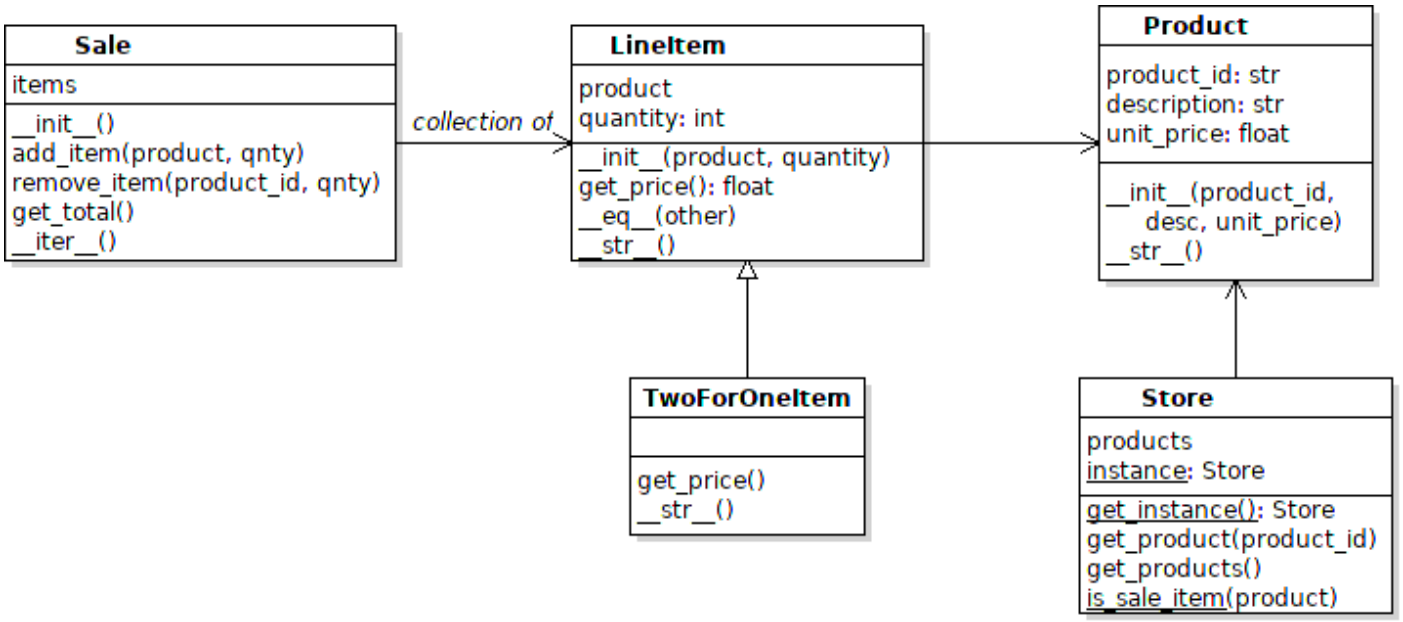
This is a simple sales application. The code for Store and Product are given.

You will write LineItem, Sale, and TwoForOneItem. Then add some code to main and write a unit test.

A Store manages a collection of Products. When a customer wants to buy some Product, the application will create a LineItem for each Product the customer buys and add it to a Sale.

A LineItem refers to a Product with a quantity. For example, to purchase 3 Bananas (product id 111), the app creates a LineItem for Banana (product) with a quantity of 3.

A Sale is a collection of LineItems -- everything the customer buys at one time, such as:
3 Bananas
2 Croissants
1 Drinking Water



Product - provided in Starter Code (product.py)

A Product is a kind of thing for sale, such as "Banana Cake".

Product has read-only (get) properties for its attributes.

product_id	(String) The product id.
description	(String) A description of the product.
unit_price	(float) the price of one unit of the product.
__str__()	returns the product description

Store - provided in starter code (store.py)

Store manages the Products in the store. Store is a singleton (we want only one Store), so you should create a store using the *class method* Store.get_instance().

get_instance()	A class method that returns a singleton instance of the Store.

get_product(product_id)	Get a Product having the requested product_id. Raises ValueError if no product with the requested product_id.
get_products()	Return a list view of all the products in the store
is_sale_item(product)	A class method that returns True if the product is on sale with "Buy 1 Get 1 Free" price.

Problem 1: LineItem

Write a LineItem class in **sale.py**. A LineItem represents one kind of thing a customer is buying, such as "3 Banana Cake". A LineItem has a quantity and a reference to a Product.

__init__(product, qnty)	Create a new LineItem with a reference to a product, and a quantity (qnty should be zero or positive).
product {property}	get the product reference. A read-only property (you cannot change the product).
quantity {get/set property}	get or set the quantity.
get_price()	Returns the price of this line item. The price is the Product unit price x quantity.
__eq__(other)	Two LineItems are equal if they have the same attributes.
__str__()	Returns the product description.

Example:

```
>>> p = Product('123', "SKE T-shirt", 120)
>>> shirts = LineItem(p, 2)   # buy 2 t-shirts
>>> shirts.quantity
2
>>> shirts.get_price()
240.0
>>> shirts.quantity = 1       # change the quantity
>>> shirts.get_price()
120.0
>>> str(shirts)
'SKE T-shirt'
```

Problem 2: Sale

Write a Sale class in **sale.py**. A Sale represents a sale of a bunch of items. Sale contains a collection of LineItems, but there can be only one LineItem for a specific Product, so if the same product is added multiple times, Sale should combine them into one LineItem.

__init__()	Create an empty Sale.
add_item(product, quantity)	add a quantity of a Product to the Sale. This either creates a new LineItem (if product is not yet in the sale) or updates the quantity of an existing LineItem. Only create 1 LineItem for each product (not 2

	LineItems for the same product). See Problem 4.2 for a later modification to this. If product_id is invalid, the Store will raise a ValueError.
remove_item(product_id, quantity)	remove some quantity of a product from the Sale using its product_id. There are several cases: a) if no LineItem for the given product_id then raise ValueError b) if the quantity is less than the amount of Product in the matching LineItem, then reduce the quantity in the LineItem c) if the quantity equals the amount of Product in the matching LineItem, then remove the LineItem from the sale c) if the quantity is *more* than the amount of Product in a matching LineItem, raise a ValueError
get_total()	Return the total price of the sale.
__iter__() -> Iterator[LineItem]	Return an Iterator for the LineItems in the Sale, so we have a way to see or print what is in the sale. Hint: a list or dict are Iterable, so you can use them to create the iterator that this method returns.

Problem 3: Complete print_sale in main.py

Complete the **print_sale** function to display all the items in the sale and the total price of the sale.

You must use the **iterator** provided by the Sale class. Do not access private attributes of Sale.

print_sale(sale)	Print a neatly formatted list of all items in the sale and the total price. Each line should show product_id, description, quantity, and price of the LineItem (not the product price). See the example below. You can use different column sizes.

```
>>> store = Store.get_instance()
>>> sale = Sale()
>>> for p in store.get_products():
...     print(p.product_id, p.description, " price", p.unit_price)
101  Drinking Water  price 9
102  Oishi Green Tea  price 16
...
111  Banana  price 8
112  Banana Cake  price 30
...

# Buy 3 drinking water
>>> water = store.get_product('101')
>>> sale.add_item(water, 3)
>>> sale.get_total()
27.0
# Buy 1 Banana (111) and 2 Banana Cake (112)
>>> p1 = store.get_product('111')
>>> p2 = store.get_prodcut('112')
```

```
>>> sale.add_item(p1, 1)
>>> sale.add_item(p2, 2)
>>> sale.get_total()
95.0
# Remove Banana Cake (use product_id to remove items)
>>> sale.remove_item('112', 2)
# Add 3 Bananas instead
>>> sale.add_item(p1, 3)
>>> main.print_sale(sale)
ID   Description          Qnty     Price
101  Drinking Water          3      27.0
112  Banana                  4      32.0
     Total                          59.0
```

After you complete Problem 4 this will print:

```
ID   Description          Qnty     Price
101  Drinking Water          3      27.0
112  Banana (2-for-1)        4      16.0
     Total                          43.0
```

Problem 4: TwoForOneItem

4.1 Write a subclass of LineItem that has "Buy 1 Get 1 Free" pricing.

`get_price()`	Get the *total* price for this LineItem, using buy-1-get-1-free pricing. If a Product has unit price 15 and a person buys qnty 4, then the price is 30 (buy 2 get 2 free), but if he buys 5 then the price is 45 (buy 3, get 2 free). If he buys 1 or 2 units, the price is 15.
`__str__()`	If the quantity is 1, it returns the product description (same as LineItem). If quantity > 1, return the product description followed by the string " (2-for-1)", such as "Banana (2-for-1)".
`other methods`	inherit from LineItem.

4.2 In Sale, modify add_item to ask the Store (is_sale_item) if a product is on sale. If it on sale, then create a TwoForOneItem instead of a regular LineItem.

Note About Decorator Pattern: There are 2 ways to use the Decorator Pattern - *class decorators* and *object decorators*. TwoForOneItem is a class decorator. In reality, an object decorator is more flexible, but always requires more code.

Problem 5: Unit Tests

Write unit tests for Sale using Python unittest (not Pytest). Test these cases:

- a new Sale is empty and has a total price 0
- if you add different quantities of a single product (create your own Product for tests) then the total sale price and quantity in the LineItem are always correct.

- if you add two different products multiple times each, then Sale only creates 2 line items (one for each product) instead of creating new line items each time, and the quantities and prices are correct.
- you can remove some quantity of a product and sale updates the quantity in the line item correctly
- you can remove the entire quantity of a product from the sale, then the line item is removed from the sale

For the unit tests, it is OK to access attributes of Sale. But better if you can write tests using only the methods of Sale.