Review the assignment due date

Open in GitHub Codespaces

(https://classroom.github.com/a/6mBBQM_-)
(https://classroom.github.com/open-in-codespaces?assignment_repo_id=17201212) Read the instructions carefully. Not following the instructions will result in not getting the credit you want for the assignment.

Make sure your output matches the sample run.

Objectives

- Implement a user-defined Protocol without explicitly inheriting from the protocol class.
- Use `isinstance()` to determine if an object belongs to a class.
- Use `issubclass()` to determine if one class is a subclass of another class.

Structure

Create or update the following things for Part 9:

- Protocol/Interface
  - Combinable in `combine.py`
- Classes
  - Candy
  - Cookie
  - Order
  - test_candy.py
  - test_cookie.py

Problem

Suppose you want to add functionality to your Dessert Shop application to combine items in an order that seem to be the same. For example, suppose the order contains two (2) items of Gummy Bears that both cost the same amount, $0.25/lb, but one if for 0.5 lbs and the other is for 1.25 lbs. It makes sense to combine those into 1 item like so: `Gummy Bears: 1.75 lbs @ $0.25/lb`

This would only make sense if both the item name and price per pound were the same. A store may sell one kind of Gummy Bear (blue) at *$0.35/lb* and another kind of Gummy Bear (green) at *$0.25/lb*. These would be different items and should not be combined.

Only identical Cookie items or identical Candy items can be combined. Trying to package two Ice Cream Sundaes into a single plastic boat wouldn't work, for example.

It is tempting to implement checks for combining by redefining `==` on combinable DessertItem objects, but this is not a good design idea:

1. We already defined all relational operators in Part 8 to do something else, including `==`.
2. The idea of combining to "same" or "similar" items, there's no concept of ordering, less than, greater than, or sorting, so it would not make sense to try to define that concept that way.

Combinable Protocol

You will define this protocol class in the file `combine.py`.

- Define the protocol class Combinable
- Define predicate method `can_combine(self, other: "Combinable")->bool`. It has no body.
- Define method `combine(self, other: "Combinable") -> "Combinable"`. It has no body.
- A precondition of calling combine successfully is that `can_combine()` would return `True` for the two items.

The Combinable Protocol would be called a pure Interface in other languages. It specifies methods, but no default implementations. In some places you see the type "Combinable" in quotes. This is necessary when you need to use the type being defined within its definition.

Changes to Candy class

Implement the Combinable protocol but **do not** not use inheritance.

- `can_combine(self, other:"Candy")` -> return True only if
    - other is an instance of Candy
    - other has the same name and same price per pound
    - otherwise False
- `combine(self, other: "Candy") -> "Candy"` -> add the weight of other to the weight of self destructively

Changes to Cookie class

Implement the Combinable protocol, but **do not** use inheritance.

- `can_combine(self, other:"Cookie") -> "Cookie"` -> return True only if
    - other is an instance of Cookie
    - other has the same name and same price per dozen
    - otherwise False
- `combine(self, other: "Cookie") -> "Cookie"` -> add the quantity of other to the quantity of self destructively

Changes to Order class

Python offers several ways to search a collection or list of items and potentially combine them.

One way is to add the following **mutually-exclusive checks** to Order's `add()` method:

1. if the new item is not Combinable or can_combine() returns False for all items in the order:
    i. add the new item to the order
2. if new item is Combinable:
    i. find the first item for which can_combine() returns True
    ii. combine the new item into existing item in the order

Test Cases

Add some new tests to test Combinable functionality.

New Candy Tests

Add one test for can_combine() that should correctly return True if both are candies.

Add one test for can_combine() that should correctly return False if the other item is not a Candy.

Add one test for combine() that should correctly combine two Candy items.

Add one test for combine() that should correctly not combine two items where one is a Candy and one is not. We expect this to fail.

New Cookie Tests

Add one test for can_combine() that should correctly return True if both are cookies.

Add one test for can_combine() that should correctly return False if the other item is not a Cookie.

Add one test for combine() that should correctly combine two Cookie items.

Add one test for combine() that should correctly not combine two items where one is a Cookie and one is not. We expect this to fail.

Key Program Requirements

1. Create the `Combinable` protocol in file `combinable.py`.
2. Implement the protocol in the `Candy` and `Cookie` classes.
3. pytest test cases have been created to test True and False cases for `can_combine`.
4. pytest test cases have been created to test cases where two items can be combined and cases where two items cannot be combined.
5. The PDF receipt shows orders with items combined as in the example run.
6. Your workspace contains all the files from Part 8 and:

- new file 'combine.py`
- the following are modified:
    - `dessert.py`
    - `test_candy.py`
    - `test_cookie.py`

If any of these are not correctly done, the most points you can receive is 50%.

Example Run

In this scenario, we show an abbreviated interaction with the menu because that did not change. We also show a receipt with **two** entries for Gummy Bears because the unit price is different.

1: Candy

2: Cookie

3: Ice Cream

4: Sunday

What would you like to add to the order? (1-4, Enter for done): 2

Enter the type of cookie: Oatmeal Raisin

Enter the quantity purchased: 4

Enter the price per dozen: 3.45

1: Candy

2: Cookie

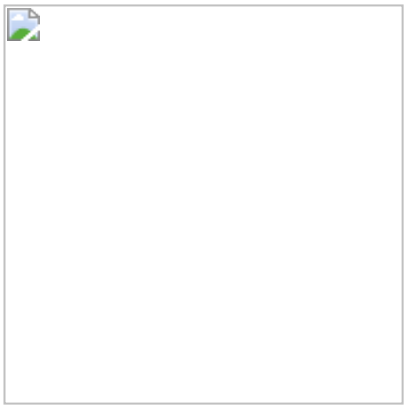
3: Ice Cream

4: Sunday

What would you like to add to the order? (1-4, Enter for done):

What form of payment will be used? (CASH, CARD, PHONE): Phone

Example receipt



Correctness

From your terminal, run `ruff check` on all of your .py source files and tesst files, like:

```
ruff check dessert.py
```

This will check for syntax errors, violations and many issues that could lead to bugs in your code. Code will be maually graded, so any score received are partial.

Style

From your terminal, run `ruff format` on all of your .py source files above to check the format of your code, like:

```
ruff format dessert.py
```

How to Submit

From your GitHub assignment repository page, click Submit and enter a nontrivial commit message.

Grading

This project is manually graded. Use the following rubric.

Criteria	Mastery (100 pts)	Developing (85 pts)	Beginning (70 pts)	Low (50 pts)

Criteria	Mastery (100 pts)	Developing (85 pts)	Beginning (70 pts)	Low (50 pts)
Implementation of the Combinable Protocol	- Properly defined in `combine.py` - All required methods present and correct	Defined in `combine.py` One method is missing or improperly implemented	`combine.py` is present but improperly defined	`combine.py` missing or no reference to Combinable
Candy and Cookie Class Implementation	Implements Combinable without inheritance All methods function correctly	Implements Combinable One or more methods have issues	Only Candy or Cookie implements Combinable	Neither Candy nor Cookie implements Combinable
Order Class Changes	Proper checks and full functionality in `add()`	Some modifications but minor issues present	Minimal modifications in `add()`	No changes related to Combinable in `add()`
Test Case Creation	All test cases created and correct	Most test scenarios covered but one or more missing or incorrect	Few test scenarios covered	No or minimal test cases related to Combinable

Students should aim for the Mastery level in all categories to ensure they have fully understood and implemented the concepts covered in the project. The overall project score is the average of the individual criteria scores.