Dessert Shop Part 3 -- CS 1420 Version

Read instructions carefully. Not following instructions will result in not getting the credit you want.

Your code from Part 2 is the starting point for Part 3.

Objectives

In Part 3 you will learn the following:

- Create an Abstract Base Class (ABC)
- Create abstract methods in a base class and concrete methods in subclasses
- Update existing classes to include additional methods
- Add new pytest test cases to existing test code
- Regression test existing methods through pytest test cases

Problem to solve

In Part 3, you add the ability to calculate the cost of any Dessert Item along with the associated tax. Also add the ability in the Order class to calculate the cost of all items in the order as well as the associated total tax.

Last, we provide example code (https://www.geeksforgeeks.org/creating-payment-receipts-using-python/) from Geeks for Geeks that will generate a PDF copy of the receipt using the reportlab module. An example PDF receipt is also provided. You will modify the example code to generate the receipt we want, and overwrite the file `receipt.pdf`.

To do this, you will make updates to your Dessert Shop system as described below.

Changes to DessertItem Class

- make it an **abstract class**
- add attribute tax_percent: float with the default value 7.25.
- add a new abstract method, calculate_cost(): float
- include a new method calculate_tax(): float that calculates and returns the actual tax for the item

Changes to All Dessert Classes Candy, Cookie, IceCream, Sundae

- add a method calculate_cost() that overrides the superclass method and returns the correct cost for the item

- Note: The cost of a Sundae is the cost of the ice cream plus the cost of the topping

Changes to Order Class

- add a new method, order_cost(), that calculates and returns the total cost for all items in the order
- add a new method, order_tax(), calculates and returns the total tax for all items in the order

Changes to receipt module

1. Add a function `make_receipt` that takes two input parameters:
    - data: list[list[str,int,float]]
    - out_file_name: str.
2. Put all the code for generating a receipt in the function `make_receipt`. No global code.
3. Add a `main()` method to `receipt.py` with conditional execution of main.
4. Example `DATA` for the receipt is a global constant in the code you are given. Move that declaration inside main(), then use that data to call your `make_receipt` function to visually test that it works.
5. In the TableStyle definition, change the `(4,4)` in GRID to `( len(DATA[0]), len(DATA) )`. Removes a hard-coded example value.
6. In the TableStyle definition, change the `3` in BACKGROUND to `len(DATA[0])-1`. Removes a hard-coded example value.

The destination file name for your manual test of `make_receipt` and for your dessert shop code in general is`receipt.pdf`. This is the value of the second parameter to `make_receipt`.

`make_receipt` should be called as the last line of `main()` in `dessertshop` module.

Changes to main()

- add a loop to main that generates the list-of-lists required by the receipt module to generate the receipt. Each row in the list includes the name of the dessert, the cost of each item and the tax for each item. Values should match what is shown in the example run below.
- add a row for subtotal of all the items in the order and the total tax for the order as shown in the scenario
- add a row for the total cost for the order (subtotal + total tax)
- add a row for the total number of items in the order as shown in the example
- call `receipt.make_receipt(data)` from `main()`.

New Test Cases to Add

- Modify your DessertItem test cases to use an instance of the Candy class. You have to test an abstract class by testing one of its concrete subclasses.
- Add new test cases to DessertItem test code that test the tax_percent attribute
- Add new test cases to test method calculate_cost for each respective dessert subclass.

- Add new test cases to test superclass method calculate_tax for each respective dessert subclass. Hint: Use the code that created example objects in main() from Part 2 as a source of test cases for each kind of object here.

Sample Run (Ed Note: Replace with PDF output)

Your generated PDF receipt format and values should match what is here. |Name| Item Cost | Tax | |--------|-|-| |Candy Corn|$0.38|$0.03| |Gummy Bears|$0.09|$0.01| |Chocolate Chip|$2.00|$0.14| |Pistachio|$1.58|$0.11| |Vanilla|$3.36|$0.24| |Oatmeal Raisin|$0.58|$0.04| |----------| |Order Subtotals|$7.99|$0.57| |Order Total||$8.56| |Total items in the order||6|

Key Requirements

1. attribute `tax_percent` is in DessertItem class
2. method `calculate_cost` is abstract in DessertItem and concrete in all inheriting subclasses
3. method `calculate_tax` is concrete in DessertItem and is **NOT** overrriden in any inheriting subclasses
4. pytest test cases have been created or modified as described above
5. PDF receipt output file should look similar to the sample run shown above
6. Your workspace should have the following 7 files:
    - dessert.py
    - dessertshop.py
    - test_dessert.py
    - test_candy.py
    - test_cookie.py
    - test_icecream.py
    - test_sundae.py

This way you don't end up with one huge test file.

Correctness

From your terminal, run `ruff check` on each of the 7 files above, such as: `ruff check dessert.py` `ruff check dessertshop.py` `ruff check test_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 each of the 7 files above to check the format of your code, like:

```
ruff format dessert.py
ruff format dessertshop.py
ruff format test_dessert.py
```

How to Submit

From your Github assignment repository page, click Commit and enter a nontrivial commit message.

Grading

Criteria	Mastery (100 points)	Proficient (85 points)	Developing (70 points)	Beginning (60 points)	Not Demonstrated (50 points)
Tax_percent attribute	Tax_percent attribute is correctly implemented in DessertItem class.	Tax_percent attribute is implemented in DessertItem class, but there are some minor errors.	Tax_percent attribute is present but not correctly implemented.	Tax_percent attribute is attempted but fundamentally flawed.	Tax_percent attribute is not implemented.
Abstract calculate_cost method	Correctly implemented as an abstract method in DessertItem and correctly overridden in all inheriting subclasses.	Correctly implemented as an abstract method in DessertItem, but there are some errors in the implementation in subclasses.	Calculate_cost method is present but not correctly implemented as an abstract method or correctly overridden.	Calculate_cost method is attempted but fundamentally flawed.	Calculate_cost method is not implemented.
Calculate_tax method	Correctly implemented in DessertItem and not overridden in any inheriting subclasses.	Correctly implemented in DessertItem, but there are some minor errors.	Calculate_tax method is present but not correctly implemented.	Calculate_tax method is attempted but fundamentally flawed.	Calculate_tax method is not implemented.
Test cases	All required pytest test cases have been created and all tests pass.	All required pytest test cases have been created but some tests do not pass.	Some required pytest test cases are missing or there are significant errors.	Test cases are attempted but fundamentally flawed.	No or very few correct pytest test cases.

Criteria	Mastery (100 points)	Proficient (85 points)	Developing (70 points)	Beginning (60 points)	Not Demonstrated (50 points)
Receipt generation	PDF receipt is correctly generated and matches the example in both format and values.	PDF receipt is generated, but there are some errors in the format or values.	Some attempt has been made to generate the receipt, but it is significantly incorrect.	Receipt generation is attempted but fundamentally flawed.	No attempt has been made to generate the receipt.
Code organization	Code is correctly divided into the 7 required files.	Code is divided into files, but not exactly as specified.	Some attempt has been made to divide the code into files, but there are significant errors.	Code organization is attempted but fundamentally flawed.	Code is not divided into separate files.
`ruff checks`	Code passes all `ruff checks` for syntax errors, violations, and potential bugs.	Code passes most `ruff checks` for syntax errors, violations, and potential bugs.	Code passes some `ruff checks` for syntax errors, violations, and potential bugs.	Code passes few `ruff checks` for syntax errors, violations, and potential bugs.	Code fails `ruff checks` for syntax errors, violations, and potential bugs.

Students should strive for mastery level. Lower than that indicates areas where more practice is needed or more learning is needed. Code score is the average of the individual feature scores.

Total score is 1/4 * style score + 3/4 * code_score.