Module 4: Pytest and Sphinx

JHU EP 605.256 – Modern Software Concepts in Python

Introduction

This assignment will introduce you to writing scalable unit tests using pytest and to making your application's documentation more easily available (and stylized) using sphinx. In this homework, we will construct an extensible pizza ordering application: a service that allows other developers to quickly build upon your restaurant template and extend to new menu items and processes.

Skills: Unittests, Object Oriented Programming, Code Documentation, Test-Driven Development

Assignment Overview

In this assignment you will:

1. Write a selection of unit tests using Pytest to answer the requirements of the given application
2. Write code to then address the unit tests and build the framework
3. Document your code service using Sphinx

After this assignment, you will have an extensible, scalable, documented pizza ordering service that will allow other developers to easily integrate ordering capabilities into a given restaurant.

1. Pizza Ordering Unit Tests

The pizza ordering package includes two core modules: **`pizza`** and **`order`**. Each customer begins the process by placing an **`order`**. An **`order`** must contain at least one **`pizza`**.

Our **`order`** code should look like the following:

```
Class Order
def __init__(self):
#Initializes a customer order
#Initialize order cost

def __str__(self):
#Print a customers complete order

def input_pizza(self, crust, sauce, cheese, toppings):
#Input the customers order for a given pizza
#Initialize the pizza object and attach to the order
#Update the cost

def order_paid(self):
#Set order as paid once payment has been collected
```

Our **`pizza`** code should look like the following:

```
Class Pizza
#Pizza objects and associated cost
def __init__(self, crust, sauce, cheese, toppings):
#Initializes a pizza
#Set pizza variables
#Set cost to create

def __str__(self):
#Print a pizza
#Print the cost of that pizza

def cost(self):
#Determine the cost of a pizza
```

Our **`pizza`** is built using the following cost associated variables:

Crust		Sauce		Topping	
Item	Cost	Item	Cost	Item	Cost
Thin	$5	Marinara	$2	Pineapple	$1
Thick	$6	Pesto	$3	Pepperoni	$2
Gluten Free	$8	Liv Sauce	$5	Mushrooms	$3

There is only one cheese option, and that is: Mozzarella cheese.

Collective **`cost`** is additive.

- E.g. a Thin Crust, Marinara, Mozzarella pizza with Pineapple will cost:

  $5 + $2 + $1 = $8

Each pizza must include at least one sauce and topping (only one crust option is allowed).

- E.g. a Thin Crust, Marinara, Liv Sauce, Mozzarella pizza with Pepperoni and Mushrooms will cost:

  $5 + $2 + $5 + $2 + $3 = $17

Each **`order`** must be paid for by customers. You may need additional functionality that is not listed within the structure above; however, you must include the above-described classes and methods.

Given this structure, we will now turn to unit tests. This code should be written using a Test-Driven Development (TDD) workflow. This means that we will start with our unit tests. Below is a list of unit tests, integration tests, and marks that need to be used for this assignment organized by component functionality.

Unit Tests

- Test **`order __init__()`**
  - Assert **`order`** should include an empty list of **`pizza`** objects
  - Assert **`order`** should have a zero **`cost`** until an order is input
  - Assert **`order`** should not have yet been **`paid`**
- Test **`order __str__()`**
  - Test **`order`** should return a string containing customer full order and cost
- Test **`order input_pizza()`**
  - Test method should update **`cost`**
- Test **`order order_paid()`**
  - Test method should update **`paid`** to true
- Test **`pizza __init__()`**
  - Test return an initialized **`pizza`**

    - Test `pizza` should have `crust (str)`, `sauce (list of str)`, `cheese (str)`, `toppings (list of str)`
    - Test `pizza` should return a non-zero `cost`
- Test `pizza __str__()`
    - Test `pizza` should return a string containing the pizza and cost
- Test `pizza cost()`
    - Test return of correct cost for an input pizza

Integration Tests

- Test that code can handle multiple `pizza` objects per `order`
    - Ensure multiple `pizza` objects within a given order result in an additively larger cost.

Marks

- Include a mark for tests that involve `order`
- Include a mark for tests that involve `pizza`

All tests should be run if marks for `order`, `pizza` are called. There should not exist a test that is not assigned to `order`, `pizza.`

2. Write Code to Address Unit Test

A common way to organize requirements is into a **SHALL**, **SHOULD, SHALL NOT** list. Requirements for your homework assignment are under this structure and give you development freedom over how to implement – if it conforms to the Shall/Should/Shall Not list.

SHALL: A high priority requirement that must be implemented as an essential part of this requirement and will otherwise result in an unsatisfactory "program correctness" grade within our assignment matrix within the syllabus.

SHOULD: A low priority requirement that will result in a good/excellent "program correctness" grade if implemented.

SHALL NOT: A forbidden component of this assignment. This list does not include items specifically mentioned on the syllabus under academic integrity, but the expectation is that those areas are similarly respected.

For this assignment your solution:

- **SHALL** follow the template and rules described in the unit test section, including all classes and methods in the creation of code
- **SHALL** use Python 3.10+
- **SHALL** place code and tests under a folder named `module_4`
- **SHALL** keep unit tests and source code in separate folders
- **SHALL** include a README
- **SHALL** include a requirements.txt
- **SHALL** rely upon pytest as your test framework

- **SHOULD** include appropriate documentation
- **SHOULD** include two separate files: **`order.py, pizza.py`**
- **SHOULD** include three separate test files: **`test_order.py, test_pizza.py, test_integration.py`**
- **SHOULD** include a **`pytest.ini`** under **`module_4`** that defines pytest markers

This order described below:

- **`Order 1:`**
  - **`Pizza(thin, pesto, mozzarella, mushrooms)`**
  - **`Pizza(thick, marinara, mozzarella, mushrooms)`**
- **`Order 2:`**
  - **`Pizza(gluten_free, marinara, mozzarella, pineapple)`**
  - **`Pizza(thin, liv_sauce, pesto, mozzarella, mushrooms, pepperoni)`**

Should result in:

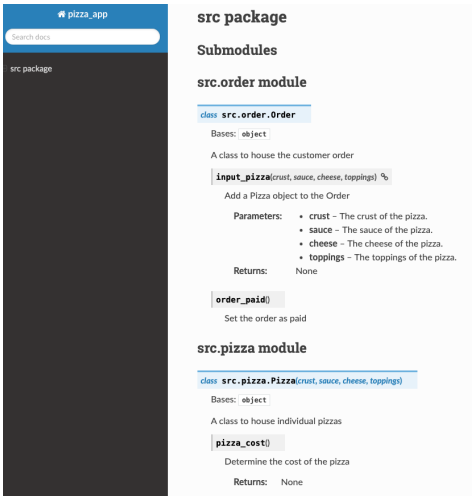
```
Customer Requested:
Crust: thin, Sauce: ['pesto'], Cheese: mozzarella, Toppings: ['mushroom'], Cost: 11
Crust: thick, Sauce: ['marinara'], Cheese: mozzarella, Toppings: ['mushroom'], Cost: 11

Customer Requested:
Crust: gluten_free, Sauce: ['marinara'], Cheese: mozzarella, Toppings: ['pineapple'], Cost: 11
Crust: thin, Sauce: ['liv_sauce', 'pesto'], Cheese: mozzarella, Toppings: ['mushroom', 'pepperoni'], Cost: 18
```

3. Build application documentation in Sphinx

Within lecture, we provided instruction on how to integrate a github project with sphinx documentation. Follow this process to connect your github repo, and then build and publish sphinx documentation for your pizza application.

Please include the HTML generated for your pizza application under **`module_4`**

Your documentation should look like this:



And it should be available on Read the Docs

4. Deliverables

1. The SSH URL to your GitHub repository
2. **`order.py, pizza.py`** under **`module_4/src`**
3. **`test_order.py, test_pizza.py, test_integration`** under **`module_4/tests`**
4. **`pytest.ini`** under **`module_4`**
5. README under **`module_4`**
6. requirements.txt under **`module_4`**
7. Link to sphinx read the docs documentation for your pizza application.
8. Sphinx generated HTML associated with your application under **`module_4`**

Please remember to submit to both canvas and commit to your private github!

Please let us know if you have any questions via Teams or email!