CS5001 HW4

Instructions

- Write your name as well as your NU ID on your assignment. Please number your problems.
- Submit both results and your code.
- Give complete answers. Do not just give the final answer; instead, show steps you went through to get there and explain what you are doing. Do not leave out critical intermediate steps.
- This assignment must be submitted electronically through Gradescope by October 15th, 2024 (Tuesday) by 9:00 AM. (Because Monday is Columbus Day. We are not going to have a HW due on vacation.)
- All of your codes must be commented.

Written Questions

1. **Name 3 kinds of exceptions.**
2. **In python, in order to handle errors, you can either ask permission or ask forgiveness. Give one example for each of the two cases.**
3. **Explain what is try-except statement? Under what circumstances might you need to use it?**

Coding Questions

1. Handling Multiple Discounts in the Smart Cart System

Whole Foods Market is introducing a discount system for bulk purchases. Customers can apply different discounts based on the total quantity of items purchased. However, discounts may occasionally be entered incorrectly, or the discount list may not match the items list.

Your task is to write a Python function called `apply_discounts` that calculates the total discounted price for a list of items, handling potential errors in input.

The function should take three arguments:

- `prices`: A list of item prices, where each price is a number.
- `quantities`: A list of item quantities, where each quantity is a number.
- `discounts`: A list of discount percentages (as decimals, e.g., 0.10 for 10% off), where each discount should correspond to an item.

The function should perform the following:

- Check if the lengths of `prices`, `quantities`, and `discounts` are the same. If not, raise a `ValueError` with the message: "Error: Prices, quantities, and discounts list length mismatch."
- For each item, calculate its total cost by multiplying the price by the quantity and then applying the discount. If a price, quantity, or discount is invalid (e.g., not a number), catch the `ValueError` or `TypeError`.
- For invalid prices, print: "Warning: Invalid item price skipped." For invalid quantities, print: "Warning: Invalid item quantity skipped." For invalid discounts, print: "Warning: Invalid discount skipped." Continue to the next item.
- After processing all items, return the total discounted cost.

2. Error Handling in an Autonomous Driving System Using Bounding Box Detection

In an autonomous driving system, bounding boxes are often used to detect and locate objects within the camera's field of view. A bounding box is defined by two pairs of coordinates `(x1, y1)` and `(x2, y2)`, where `(x1, y1)` represents the top-left corner of the box and `(x2, y2)` represents the bottom-right corner. These coordinates define the area in which an object, such as a pedestrian or vehicle, is detected.

In this exercise, you will create a Python program that manages detected objects using bounding boxes. You will also handle potential errors that can occur when manipulating bounding boxes in an autonomous driving system.

Bounding Box Data Structure

Each detected object will be stored in a dictionary, where the key is the object's unique identifier (such as an integer ID) and the value is a dictionary representing the bounding box:

```
bounding_boxes = {
    1: {'x1': 50, 'y1': 100, 'x2': 200, 'y2': 250},
    2: {'x1': 300, 'y1': 400, 'x2': 350, 'y2': 450},
}
```

Tasks and Error Handling

Implement the following functionalities with appropriate error handling mechanisms:

- **Add a new detected object:** Add a bounding box for a new object by specifying the coordinates `x1`, `y1`, `x2`, and `y2`. If the coordinates are invalid (e.g., `x2` is less than `x1` or `y2` is less than `y1`), raise a `ValueError` and display an appropriate error message.
- **Update an existing bounding box:** Update the coordinates of an existing bounding box by its object ID. If the ID does not exist, raise a `KeyError`. Handle cases where the coordinates provided are invalid, just as in the add function.
- **Remove a detected object:** Remove a bounding box by its object ID. If the ID does not exist, raise a `KeyError` and display a message that the object was not found.
- **Display all detected objects:** Display the list of all detected objects along with their bounding box coordinates.
- **Handle invalid input types:** Ensure that the bounding box coordinates are integers. If a user inputs non-integer values, raise a `TypeError` with an appropriate error message.

3. Collision Detection in AR Systems

In AR/VR systems, collision detection is a key technique used to determine whether two or more virtual objects intersect or come into contact. For example, in a VR game, collision detection ensures that a player cannot walk through walls or objects. In an augmented reality setting, it is used to determine if a virtual object is overlapping with a real-world object, allowing for accurate interaction between the virtual and real worlds.

Collision detection often relies on defining the boundaries of objects using geometric shapes such as spheres, cubes, or more complex meshes. In this exercise, you will work with simplified bounding spheres to represent objects and detect collisions between them. A bounding sphere is defined by a center point `(x, y, z)` and a radius `r`.

Bounding Sphere Data Structure

Each object in the AR/VR environment will be stored in a dictionary, where the key is the object's unique identifier (such as an integer ID), and the value is a dictionary representing the bounding sphere:

```
objects = {
    1: {'x': 50, 'y': 100, 'z': 200, 'r': 15},
    2: {'x': 60, 'y': 110, 'z': 210, 'r': 20},
}
```

Tasks and Error Handling

Implement the following functionalities with appropriate error handling mechanisms:

- **Add a new object:** Add a bounding sphere for a new object by specifying the center coordinates `x`, `y`, `z`, and the radius `r`. If the radius is negative or zero, raise a `ValueError` and display an appropriate error message.
- **Check for collisions between two objects:** Implement a function to check if two objects (identified by their object IDs) collide. A collision occurs if the distance between the centers of two spheres is less than or equal to the sum of their radii. If one or both object IDs do not exist, raise a `KeyError` and handle it appropriately. If the objects are not colliding, return a message indicating no collision.
- **Update the bounding sphere of an existing object:** Update the position or radius of an object by its ID. Ensure that the radius is positive and handle errors for non-existent object IDs with a `KeyError`.
- **Remove an object:** Remove an object from the environment by its ID. Handle cases where the object ID does not exist by raising a `KeyError`.
- **Display all objects:** Display the list of all objects with their center coordinates and radii.
- **Handle invalid input types:** Ensure that the coordinates and radii are integers or floats. If non-numeric values are input, raise a `TypeError` with an appropriate error message.