Ilyass Hmamou

ilyass.hmamou@snhu.edu

CS-499: Computer Science Capstone

4-2 Milestone Two: Enhancement Two: Algorithm and data structure

1- Artifact description:

For the improvement in this category of Algorithm and data structure, I continued working with the "Salvare Search For Rescue App." artifact. As a reminder, It was developed during the period between September 2023 and October 2023 as a component of the final project for CS-340: Client/Server development course.

The application enables users to engage with data from an existing animal shelter database through two distinct methods: an interactive datatable equipped with built-in filters, and an interactive map within a user-friendly interface. Constructed with the utilization of the Python language, pyMongo driver, Dash framework, and MongoDB, the app provides a comprehensive user experience.

2- Reason of choosing this artifact:

This artifact presents significant opportunities for advancement, allowing me to showcase my proficiency in analyzing ongoing projects, recognizing shortcomings, and implementing successful resolutions. In this specific refinement, my focus was on enhancing the algorithm responsible for updating the dashboard according to the user's selected filter. This improvement entailed modifications to the server-side filtering functionality and the utilization of a dictionary data structure to revamp the existing code. The objective was to enhance readability, maintainability, and reduce code duplication—principles integral to effective algorithm planning and efficient code composition.

3- Enhancement technical details:

The focus on this enhancement was to improve the Algorithm used on the server side filtering logic, which will lead to improve the efficiency of the code. This code improvement plan was around eliminating a nested if-elif block in the update_dashboard() method and replace it with a switch case like logic; this change required the usage of dictionary data structure since Python doesn't have a built-in switch case statement, therefore, I used a dictionary to map cases to functions, and provide a similar case of switch case. On this report, I will provide a deep dive on the technical details for this enhancement.

3-1: Why using dictionary to map cases instead of if-elif logic:

The refactored code using the dictionary approach is not necessarily more efficient in terms of computational performance. The primary advantage lies in improved readability, maintainability, and modularity. Let me explain:

1. Readability:

- The use of a dictionary to map cases to functions makes it clear which function corresponds to each case. This can enhance the readability of your code, especially when there are multiple cases.

2.Maintainability:

- If you need to add or modify cases, you can do so by simply updating the `filter_cases` dictionary. This makes the code easier to maintain and extend. In the original `if-elif-else` block, adding or modifying cases might involve changing multiple locations in the code, making it more error-prone.

3. Modularity:

- Each filter is encapsulated within its own function, promoting a more modular design. This makes it easier to understand each individual case and modify behavior without affecting the rest of the code.

4. Reduced Code Duplication:

- The dictionary approach eliminates the need to repeat the call to `pd.DataFrame(list(db.readAll(...)))` for each case, reducing code duplication.

5. Flexibility:

- The dictionary approach allows for the inclusion of more complex logic within each filter function if needed. This can be beneficial as the codebase evolves.

It's important to note that these advantages come from a design and maintenance perspective. In terms of runtime efficiency, the two approaches are likely to be similar, and the choice between them often depends on factors like codebase size, team preferences, and future extensibility

requirements. If performance is a critical concern, the focus should be on optimizing the logic within the filter functions rather than the structure of the control flow.

3-2: Existing code:

v1: original artifact code:

```
@app.callback([Output('datatable-id','data'),
               Output('datatable-id','columns')],
              [Input('filter-type', 'value')])
def update_dashboard(filter_type):
        #filter interactive data table with MongoDB queries
        if filter_type == 'WR':
            df = pd.DataFrame(list(db.readAll({
                "breed" : {"$in": ["Labrador Retriever Mix", "Chesapeake Bay Retriever", "Newfoundland"]},
                "sex_upon_outcome" : "Intact Female",
                "age_upon_outcome_in_weeks": {"$lte": 156, "$gte": 26}})))
        elif filter_type == 'MWR':
            df = pd.DataFrame(list(db.readAll({
                "breed" : {"$in": ["German Shephard", "Alaskan Malamute", "Old English Sheepdog",
                                   "Siberian Husky", "Rottweiler"]},
                "sex_upon_outcome" : "Intact Male",
                "age_upon_outcome_in_weeks": {"$lte": 156, "$gte": 26}})))
        elif filter_type == 'DRIT':
            df = pd.DataFrame(list(db.readAll({
                "breed" : {"$in": ["Doberman Pinscher", "German Shephard", "Golden Retriever",
                                   "Bloodhound", "Rottweiler"]},
                "sex_upon_outcome" : "Intact Male",
                "age_upon_outcome_in_weeks": {"$lte": 300, "$gte": 20}})))
        elif filter_type == 'All':
            df = pd.DataFrame.from_records(db.readAll({}))

        columns=[{"name": i, "id": i, "deletable": False, "selectable": True} for i in df.columns]
        data=df.to_dict('records')

        return (data,columns)
```

Updating this if elif block to switch case

V2: post enhancement one artifact code:

```
def update_dshboard(selected_filter, btn1, btn2):
    #if Disaster Rescue or Individual Tracking option selected
    if (selected_filter == 'drit'):
        # call the readAll method from the AnimalShelter class
        # pass parameters identifying the selected type of rescue
        df = pd.DataFrame(list(db.readAll(
                {
                    "animal_type":"Dog",
                    "breed":{"$in":[
                        "Doberman Pinscher","German Shepherd","Golden Retriever","Bloodhound","Rottweiler"]},
                    "sex_upon_outcome":"Intact Male",
                    "age_upon_outcome_in_weeks": {"$gte":20},
                    "age_upon_outcome_in_weeks":{"$lte":300}
                }
            )
        ))
    # if Mountain or Wilderness Rescue option is selected
    elif (selected_filter == 'mwr'):
        # call the readAll method from the AnimalShelter class
        # pass parameters identifying the selected type of rescue
        df = pd.DataFrame(list(db.readAll(
                {
                    "animal_type":"Dog",
                    "breed":{"$in":[
                        "German Shepherd","Alaskan Malamute","Old English Sheepdog",
                        "Siberian Husky","Rottweiler"]},
                    "sex_upon_outcome":"Intact Male",
                    "age_upon_outcome_in_weeks":{"$gte":26},
                    "age_upon_outcome_in_weeks":{"$lte":156}
                }
            )
```

Updating if elif block to switch case

```
                }
            )
        ))
    # if Water Rescue option is selected
    elif (selected_filter == 'wr'):
        # call the readAll method from the AnimalShelter class
        # pass parameters identifying the selected type of rescue
        df = pd.DataFrame(list(db.readAll(
                {
                    "animal_type":"Dog",
                    "breed":{"$in":["Labrador Retriever Mix","Chesapeake Bay Retriever","Newfoundland"]},
                    "sex_upon_outcome":"Intact Female",
                    "age_upon_outcome_in_weeks":{"$gte":26},
                    "age_upon_outcome_in_weeks":{"$lte":156}
                }
            )
        ))
    # higher number of button clicks to determine filter type
    elif (int(btn1) > int(btn2)):
        # call the readAll method from the AnimalShelter class
        # pass parameters animal_type = Cat
        df = pd.DataFrame(list(db.readAll({"animal_type":"Cat"})))
    elif (int(btn2) > int(btn1)):
        # call the readAll method from the AnimalShelter class
        # pass parameters animal_type = Dog
        df = pd.DataFrame(list(db.readAll({"animal_type":"Dog"})))
    else:
        # when reset button is clicked, call readAll method with no parameters to return all the data
        df = pd.DataFrame.from_records(db.readAll({}))

    data = df.to_dict('records')
```

updating if elif block to switch case

3-3: The New code:

Let's walk through the refactored code step by step:

3-3-1. Filter Functions:

First, we define individual functions for each filter type. These functions return the filter criteria based on the selected filter type.

```
disaster_rescue_filter():
return {
    "animal_type": "Dog",
    "breed": {"$in": ["Doberman Pinscher", "German Shepherd", "Golden Retriever", "Bloodhound", "Rottweiler"]},
    "sex_upon_outcome": "Intact Male",
    "age_upon_outcome_in_weeks": {"$gte": 20, "$lte": 300}
}
```

```
mountain_wilderness_rescue_filter():
return {
    "animal_type": "Dog",
    "breed": {"$in": ["German Shepherd", "Alaskan Malamute", "Old English Sheepdog", "Siberian Husky", "Rottweil
    "sex_upon_outcome": "Intact Male",
    "age_upon_outcome_in_weeks": {"$gte": 26, "$lte": 156}
}
```

```
water_rescue_filter():
return {
    "animal_type": "Dog",
    "breed": {"$in": ["Labrador Retriever Mix", "Chesapeake Bay Retriever", "Newfoundland"]},
    "sex_upon_outcome": "Intact Female",
    "age_upon_outcome_in_weeks": {"$gte": 26, "$lte": 156}
}
```

```
cat_filter():
return {"animal_type": "Cat"}
```

```
dog_filter():
return {"animal_type": "Dog"}
```

```
reset_filter():
return {}
```

3-3-2. Dictionary Mapping:

Next, we create a dictionary (`filter_cases`) to map filter types to their corresponding functions. Then, refactor the main function that updates the dashboard (`update_dashboard`) to use the dictionary to get the appropriate filter function based on the selected filter type.

```
update_dashboard(selected_filter, btn1, btn2):
filter_cases = {
    'drit': disaster_rescue_filter,
    'mwr': mountain_wilderness_rescue_filter,
    'wr': water_rescue_filter,
    'cat': cat_filter,
    'dog': dog_filter,
    'reset': reset_filter
}

selected_filter_fn = filter_cases.get(selected_filter, reset_filter)
df = pd.DataFrame(list(db.readAll(selected_filter_fn())))

data = df.to_dict('records')
return data
```

explanation:

- `selected_filter` is the user-selected filter type.

- `filter_cases.get(selected_filter, reset_filter)` retrieves the corresponding filter function from the dictionary. If the selected filter is not found, it defaults to the `reset_filter` function.

- The function then fetches data from the database using the selected filter criteria and converts it to a format suitable for the dashboard.

3-3-3. Update Buttons Function:

The function that resets the clicks of the cat and dog filter buttons (`update_buttons`) has been modified to use `PreventUpdate` when the reset button is clicked.

```
def update_buttons(reset):
    if reset:
        return 0, 0
    else:
        raise PreventUpdate
```

Explanation:

- If the reset button is clicked (`reset` is truthy), it returns 0 for both buttons to reset their click counts.

- If the reset button is not clicked, it raises `PreventUpdate` to prevent undesired updates.

3-3-4. Dash Callbacks:

Finally, the Dash callbacks are simplified to directly call the respective functions.

```
# This callback add interactive dropdown filter option to the dashboard to find dogs per category
# or interactive button filter option to the dashboard to find all cats or all dogs
@app.callback(
    Output('datatable-id', 'data'),
    [Input('filter-type', 'value'),
     Input('submit-button-one', 'n_clicks'),
     Input('submit-button-two', 'n_clicks')]
)
def callback_update_dashboard(selected_filter, btn1, btn2):
    return update_dashboard(selected_filter, btn1, btn2)
```

```
# This callback reset the clicks of the cat and dog filter button
@app.callback(
    [Output('submit-button-one', 'n_clicks'),
     Output('submit-button-two', 'n_clicks')],
    [Input('reset-buttons', 'n_clicks')]
)
def callback_update_buttons(reset):
    return update_buttons(reset)
```

These callbacks directly call the `update_dashboard` and `update_buttons` functions, improving readability and maintaining the separation of concerns.

In summary, the refactoring uses functions and a dictionary to organize and streamline the code, making it more modular, readable, and maintainable. Each function has a specific responsibility, and the dictionary provides an efficient way to handle multiple filter cases.

3-4: Time complexity, Optimization and Efficiency comparison between original and refactored code

3-4-1. Time complexity:

In Python, both a switch-like structure using a dictionary and a series of `if-elif` statements have similar time complexities for average and worst-case scenarios.

a. Switch Case using Dictionary:

- The dictionary lookup in Python has an average time complexity of O(1). This means that, on average, the time it takes to look up a value in a dictionary is constant, regardless of the size of the dictionary.

- The worst-case time complexity for a dictionary lookup is O(n), where n is the size of the dictionary. However, in practical scenarios, dictionary lookups are highly optimized, and the worst-case scenario is rarely encountered.

b. Original if-elif Structure:

- The original `if-elif` structure has a linear time complexity O(k), where k is the number of conditions. In the worst case, the time complexity is O(k) because it needs to evaluate each condition until it finds a match.

- For a small number of conditions, the linear time complexity is not a significant concern, but as the number of conditions grows, the code's performance could degrade.

Conclusion:

In terms of time complexity, both approaches are reasonable, and the difference is often negligible. The primary advantage of using a dictionary is improved readability and maintainability, especially when dealing with a large number of conditions. The dictionary allows for direct mapping of cases to functions without the need for sequential evaluation.

Therefore, the switch case model using a dictionary is preferred in Python for its readability and maintainability, even if the time complexity is similar to the original `if-elif` structure. The benefits of code organization and ease of understanding often outweigh minor differences in time complexity for these types of structures.

Southern New Hampshire University

3-4-2. Optimization

Let's compare the optimization levels of the switch case using a dictionary versus the original `if-elif` structure in terms of readability, maintainability, and potential for code optimization:

A- Switch Case using Dictionary:

Advantages:

1. Readability and Maintainability:

- The dictionary-based approach is often more readable, especially when dealing with a large number of conditions. Each case is explicitly defined in a dictionary, making it easy to understand.

2. Ease of Modification:

- Adding or modifying cases is straightforward. You can extend the dictionary without modifying the core logic, promoting a clean and modular design.

3. Reduced Repetition:

- If there are patterns or similarities between cases, you can reduce redundancy, as shown in the previous example with the `age_filter` function.

Disadvantages:

1. Limited to Equality Checks:

- The dictionary-based approach is effective when conditions are based on equality checks. If more complex conditions are required, additional logic might be needed within the functions.

B-Original if-elif Structure:

Advantages:

1. Explicit Control Flow:

- The original `if-elif` structure provides explicit control flow, making it easy to understand the sequence of conditions.

2. Easier to Debug:

- Debugging might be more straightforward in an `if-elif` structure because you can see the sequence of conditions and identify which one is being triggered.

Disadvantages:

1. Readability and Maintainability:

- As the number of conditions grows, the `if-elif` structure can become less readable and more error-prone.

2. Potential for Redundancy:

- There's a higher potential for redundancy, especially if conditions share common elements. This can lead to maintenance challenges.

Conclusion:

The switch case using a dictionary is generally favored in Python for its readability and maintainability advantages. While both approaches have similar time complexities, the switch case using a dictionary allows for a cleaner organization of code, easier modification, and reduced redundancy.

In practice, the switch case using a dictionary is often considered a more "Pythonic" and idiomatic approach when dealing with multiple conditions. However, the best approach depends on the specific context and requirements of your application.

3-3-4. Efficiency:

In terms of code efficiency, both the switch case using a dictionary and the original `if-elif` structure are likely to have similar performance characteristics, as the primary time complexity driver is the database query operation, and both structures have similar average time complexities.

Here are some considerations for efficiency:

A-Switch Case using Dictionary:

Efficiency Advantages:

1. Constant Time Dictionary Lookup:

- Dictionary lookups in Python have an average time complexity of O(1), providing efficient and constant-time access to functions based on the selected filter.

2. Readability and Maintainability:

- The cleaner and more modular design facilitated by the dictionary-based approach can lead to more maintainable code, which is beneficial in terms of development efficiency.

Efficiency Disadvantages:

1. Potential Memory Overhead:

- The dictionary-based approach may have a slightly higher memory overhead due to the need to store the dictionary itself. However, this is often negligible unless the dictionary is extremely large.

B-Original if-elif Structure:

Efficiency Advantages:

1. Explicit Control Flow:

- The original `if-elif` structure might have a more explicit control flow, which can be beneficial for understanding the sequence of conditions.

Efficiency Disadvantages:

1. Linear Time Complexity:

- The `if-elif` structure has a linear time complexity O(k), where k is the number of conditions. In the worst case, it needs to evaluate each condition sequentially until it finds a match. This might be less efficient for a large number of conditions.

C- Conclusion:

In terms of efficiency, the differences between the switch case using a dictionary and the original `if-elif` structure are likely to be minimal in most practical scenarios. The primary considerations should be code readability, maintainability, and ease of modification.

The switch case using a dictionary is often favored in Python for its clean and idiomatic design. The efficiency gains, if any, are likely to be marginal compared to the benefits of improved code organization and readability. Ultimately, the choice between the two approaches should consider the specific requirements and preferences of your project.

4- Course objective achieved:

This enhancement showcases two course outcomes:

1- 'Design and evaluate computing solutions that solve a given problem using algorithmic principles and computer science practices and standards appropriate to its solution, while managing the trade-offs involved in design choices'

2- ‘Design, develop, and deliver professional-quality oral, written, and visual communications that are coherent, technically sound, and appropriately adapted to specific audiences and contexts‘

These results were attained by analyzing the existing code and pinpointing algorithmic shortcomings, such as the utilization of nested if-elif clauses, the code exhibiting a waterfall structure, and a deficiency in coding efficiency resulting in poor readability and maintainability. These issues address the first outcome of devising and assessing computational solutions that resolve a given problem through algorithmic principles. Subsequently, these concerns were rectified by introducing a more efficient code implementation and furnishing a comprehensively documented explanation, as illustrated in the step-by-step walkthrough outlined above. This addresses the secondary outcome.

The skills utilized to attain this result encompass the following:

1. Evaluating the current code and pinpointing areas of concern, such as scalability, maintainability, and readability.

2. Devising an algorithmic solution that seamlessly translates into code to address the identified issues.

3. Recognizing the significance of code efficiency and the need for comprehensive documentation.

4. Proficiency in generating technical documentation outlining the necessary modifications, coupled with a step-by-step elucidation of the proposed solution.

5- Reflection:

Throughout this improvement process, I gained an understanding of the crucial importance of implementing code that is scalable and maintainable. When initially creating the Artifact for my previous course, my primary goal was to complete the project for the course requirements. I did not give much consideration to the quality of the code or the potential need to revisit it for improvements or enhancements. Upon revisiting the project during the capstone course, I encountered challenges in making meaningful updates due to inadequate documentation and poor code readability. This experience underscored the lesson that it is essential to consistently prioritize code quality, maintainability, and scalability. Even if immediate enhancements or improvements are not apparent during the initial implementation, code should be clean and structured with the awareness that someone may need to work on it in the future.