Gregoria Ramirez

CS 499 Capstone Enhancement and Narrative

Algorithms and Data Structures

Module 4

July 27, 2025

DR Troy Hawk

For this category, I selected my Animal Shelter Dashboard project originally built for CS 340: Client-Server Development in February 2025. This project is a web-based dashboard built in Python using the Dash framework and connected to a MongoDB database. The dashboard allows users to explore shelter outcome data using dropdown and radio button filters and visualizes the results in a searchable table and interactive charts.

I included this artifact in my ePortfolio because it demonstrates how I applied structured logic and efficient control flow to solve real-world filtering and visualization problems. In the original version, the *update\_dashboard()* function used repetitive *if* blocks and direct filtering that made it harder to manage. In the enhanced version, I reorganized this function to apply layered filters that only trigger when conditions are met. I also used pandas’ vectorized operations to make filtering more efficient and added input validation so the dashboard would not break when dropdowns were empty or when combinations did not match any records.

**Examples Before and After Enhancements:**

**Original Code** *(Before Enhancement):*

# Filtering data directly on original DataFrame multiple times

if filter\_type == 'water':

filtered\_df = df[(df['breed'].str.contains('Labrador Retriever')) &

(df['outcome\_type'] == 'Euthanasia')]

**This approach:**

1. Repeatedly filters the full DataFrame
2. Overwrites previous filter results without preserving the original data
3. Does not handle empty or missing filter inputs properly

**Enhanced Code** (After Enhancement):

def update\_dashboard(filter\_type, selected\_colors, selected\_breeds):

filtered\_df = df.copy()

if filter\_type == 'water':

filtered\_df = filtered\_df[

(filtered\_df['breed'].str.contains('Labrador Retriever', na=False)) &

(filtered\_df['outcome\_type'] == 'Euthanasia') &

(filtered\_df['animal\_type'] == 'Dog')

]

elif filter\_type == 'mount':

filtered\_df = filtered\_df[

(filtered\_df['outcome\_type'] == 'Transfer') &

(filtered\_df['animal\_type'] == 'Cat') &

(filtered\_df['sex\_upon\_outcome'].str.contains('Female', na=False)) &

(filtered\_df['age\_upon\_outcome\_in\_weeks'].between(52, 260))

]

**# Additional filters that apply safely and independently**

if selected\_colors:

filtered\_df = filtered\_df[filtered\_df['color'].isin(selected\_colors)]

if selected\_breeds:

filtered\_df = filtered\_df[filtered\_df['breed'].isin(selected\_breeds)]

#  **Return a clean, predictable structure for use in the dashboard**

return filtered\_df.to\_dict('records')

**Key Enhancements:**

1. Added layered (chained) filter conditions that apply sequentially and only when relevant, improving clarity and control flow.
2. Used *df.copy()* to create a separate working dataset, preserving the original data and preventing unintended side effects.
3. Implemented safer filtering methods such as *.str.contains(..., na=False*) and *.isin()* to avoid runtime errors caused by missing or null data.
4. Handled multiple filter combinations properly so that filters work together instead of conflicting or overwriting each other.
5. Returned a clean, consistent data structure with .*to\_dict('records')* that the dashboard can reliably use.

These changes not only improved runtime efficiency but also reduced the amount of data being processed in each step. For example, filtering a 3,000-row DataFrame down to just 150–300 relevant records per interaction significantly improved responsiveness, especially during rapid user input. These improvements enhanced both performance and user experience. The dashboard now reacts more smoothly to filter changes and is more resilient to unusual inputs or empty selections.

Although I focused on improving the filtering logic and performance, I also added logical input checks to prevent the dashboard from producing empty or invalid datasets when users select rare breeds, uncommon rescue types, or when the dataset has missing values. By catching these cases early in the filtering process, I ensured predictable and stable results in the interface.

**Alignment to Program Outcomes:**

This enhancement aligns with Outcome 3: 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. The trade-off here was between performance and readability. Chained filters and vectorized methods provided speed improvements but required careful commenting and structure to keep the code easy to follow.

Enhancing this artifact helped me better understand the importance of organizing logic for both readability and efficiency. One challenge I faced was making sure multiple filters could work together without interfering with each other’s results. I tested various combinations to ensure the application behaved correctly in all cases. Handling missing values or null fields required deliberate use of *na=False* to prevent runtime errors in string filtering. Another challenge was maintaining clarity while introducing more flexible logic, which I solved by grouping related filtering steps and adding inline comments.

This process improved my confidence in using control flow and data structures to solve real-world problems. It also reinforced how thoughtful, planned logic design can lead to stable, responsive, and professional-grade applications.