**CS FINAL PROJECT** (Dishita, Rohan)

**Project Proposal: Election Data Management System**

The goal of this project is to develop a user-friendly Election Data Management System (EDMS) to design the collection, organization, and analysis of election-related data in an efficient way. This system will cater to various users involved in the election process, including campaign managers, researchers, and voters.

**Significance of the Project :**

* Transparency and Accountability: EDMS will promote transparency by providing a centralized platform for managing candidate information, contributions, and promotions. This allows for easier public access to election data, promoting trust and accountability.
* Informed Decision-Making: By enabling efficient data analysis, EDMS empowers voters to make informed decisions based on candidate details and campaign activities.
* Efficiency and Organization: EDMS streamlines data management for campaign managers and researchers. It eliminates manual data entry and facilitates data retrieval, saving time and resources.

**Installation and Instruction to use :**

* Availability: The initial release of EDMS will be a web-based application accessible through a standard web browser. This eliminates the need for complex installations.
* User Registration: Users will be required to register for an account with appropriate access levels based on their role (e.g., voter, campaign manager).
* Data Entry: The system will provide clear interfaces for entering candidate information, contributions (including contributor details), and promotion details (type, cost, dates).
* Data Access: Voters can search for candidates by name, district, or party. They can also view basic information about contributions and promotions.
* Data Analysis (Limited): The initial release will offer basic data visualization tools for candidates and campaign managers to view trends in contributions.

Overall, this EDMS project aims to provide a valuable tool for stakeholders in the election process, promoting transparency, informed decision-making, and efficient data management.

To design an application for managing candidate information, campaign contributions, and promotional activities, it's crucial to structure it well both in terms of software architecture and database design. Using Python data structures and algorithms efficiently will ensure that the application is scalable, and performs well under load. Here’s a detailed approach to designing this application:

**1.Database Schema Design**

The database schema needs to accommodate candidates, their campaign contributions, and promotional activities. Here’s a suggested schema:

Tables

* Candidates
  + CandidateID
  + Name
  + Party
  + District
* Contributions
  + ContributionID
  + CandidateID
  + Amount
  + Date
  + ContributorName
* Promotions
  + PromotionID
  + CandidateID
  + Type (e.g., Social Media, TV, Radio)
  + Cost
  + StartDate
  + EndDate

Relationships

* A Candidate can have multiple Contributions and Promotion

**2. Data Manipulation Using Python Lists and Dictionaries**

1. Incorporating Python's list and dictionary structures for efficient data manipulation and implementing various algorithms for data retrieval and analysis can significantly enhance the functionality of your application. Below, we provide sample implementations for searching, sorting, and potentially using graph algorithms to handle data related to candidates, contributions, and promotions.
2. **Search Algorithms**

For efficient retrieval of candidate information, you can implement a simple search algorithm to filter through a list of dictionaries representing candidates:

1. **Sorting Algorithms**

Sorting is crucial for organizing data like contributions. Here's a basic implementation using Python's built-in sorting capabilities:

**Functionalities and Test Results :**

Class: ElectionDatabase

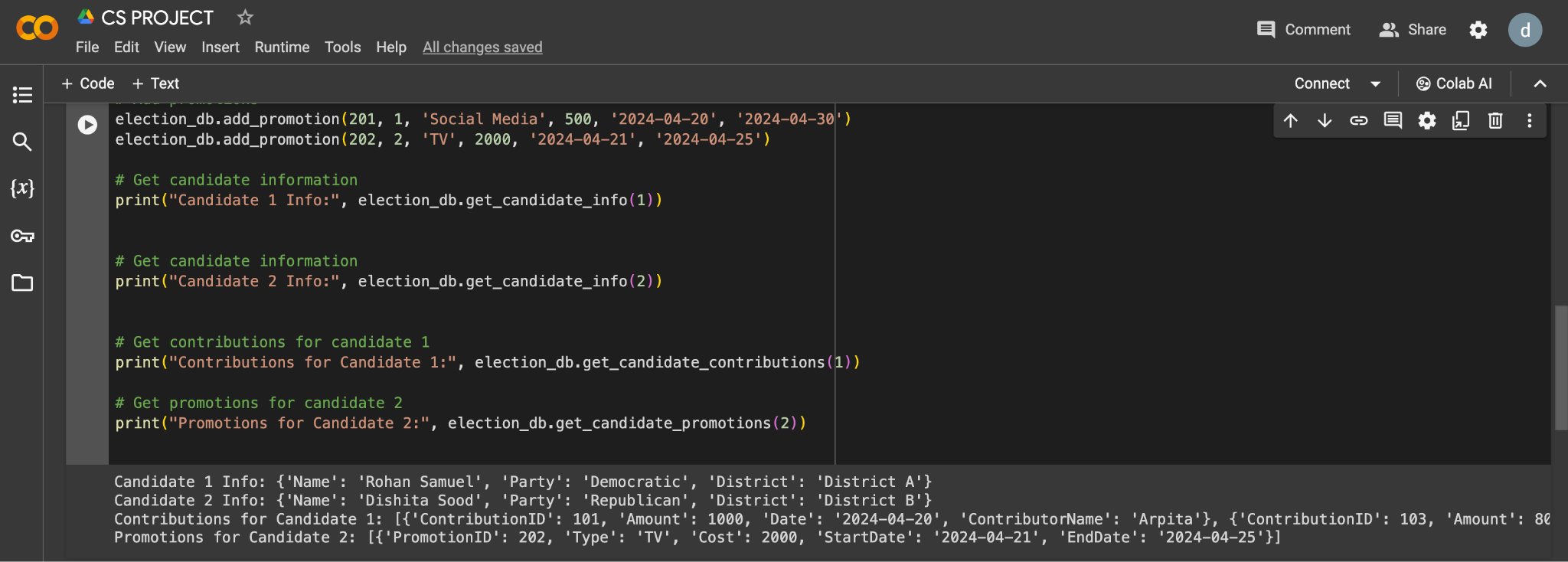
**1. Database Schema Design**

* This class creates a database to store information about candidates, their contributions, and promotions.
* It uses dictionaries to organize the data.
  + candidates: Stores candidate information using candidate ID as the key and a dictionary containing name, party, and district as the value.
  + contributions: Stores contributions for each candidate using candidate ID as the key and a list of dictionaries containing contribution details as the value.
  + promotions: Stores promotions for each candidate using candidate ID as the key and a list of dictionaries containing promotion details as the value.

Methods:

* \_\_init\_\_(self): Initializes the class and creates empty dictionaries for candidates, contributions, and promotions.
* add\_candidate(self, candidate\_id, name, party, district): Adds a new candidate to the candidates dictionary with the provided information.
* add\_contribution(self, contribution\_id, candidate\_id, amount, date, contributor\_name): Adds a contribution record for a specific candidate with details like amount, date, and contributor name. It first checks if the candidate has any contributions already and creates a new list if not.
* add\_promotion(self, promotion\_id, candidate\_id, type\_, cost, start\_date, end\_date): Adds a promotion record for a candidate with information about promotion type, cost, start and end date. Similar to add\_contribution, it checks for existing promotions first.
* get\_candidate\_info(self, candidate\_id): Retrieves and returns the information for a specific candidate using their ID.
* get\_candidate\_contributions(self, candidate\_id): Retrieves and returns a list of all contributions for a specific candidate.
* get\_candidate\_promotions(self, candidate\_id): Retrieves and returns a list of all promotions for a specific candidate.

**Test Result:**



**2. Data Manipulation Using Python Lists and Dictionaries**

1. **Incorporating Python's list and dictionary structures**

(i) aggregate\_contributions(contributions):

This function takes a list of contributions (dictionaries containing candidate information) and aggregates them by candidate ID. Here's what it does:

* aggregation = {}: Initializes an empty dictionary to store the aggregated data.
* for contribution in contributions: Loops through each contribution in the provided list.
* candidate\_id = contribution['candidate\_id']: Extracts the candidate ID from the current contribution dictionary.
* amount = contribution['amount']: Assuming 'amount' is a key in the contribution dictionary, it extracts the contribution amount.
* if candidate\_id in aggregation:: Checks if the candidate ID already exists in the aggregation dictionary.
  + aggregation[candidate\_id] += amount: If the ID exists, it adds the current contribution amount to the existing total for that candidate.
* else If the candidate ID is not found yet, it creates a new key-value pair in the aggregation dictionary with the candidate ID as the key and the current contribution amount as the value.
* return aggregation: Finally, the function returns the dictionary containing the aggregated contributions by candidate ID.

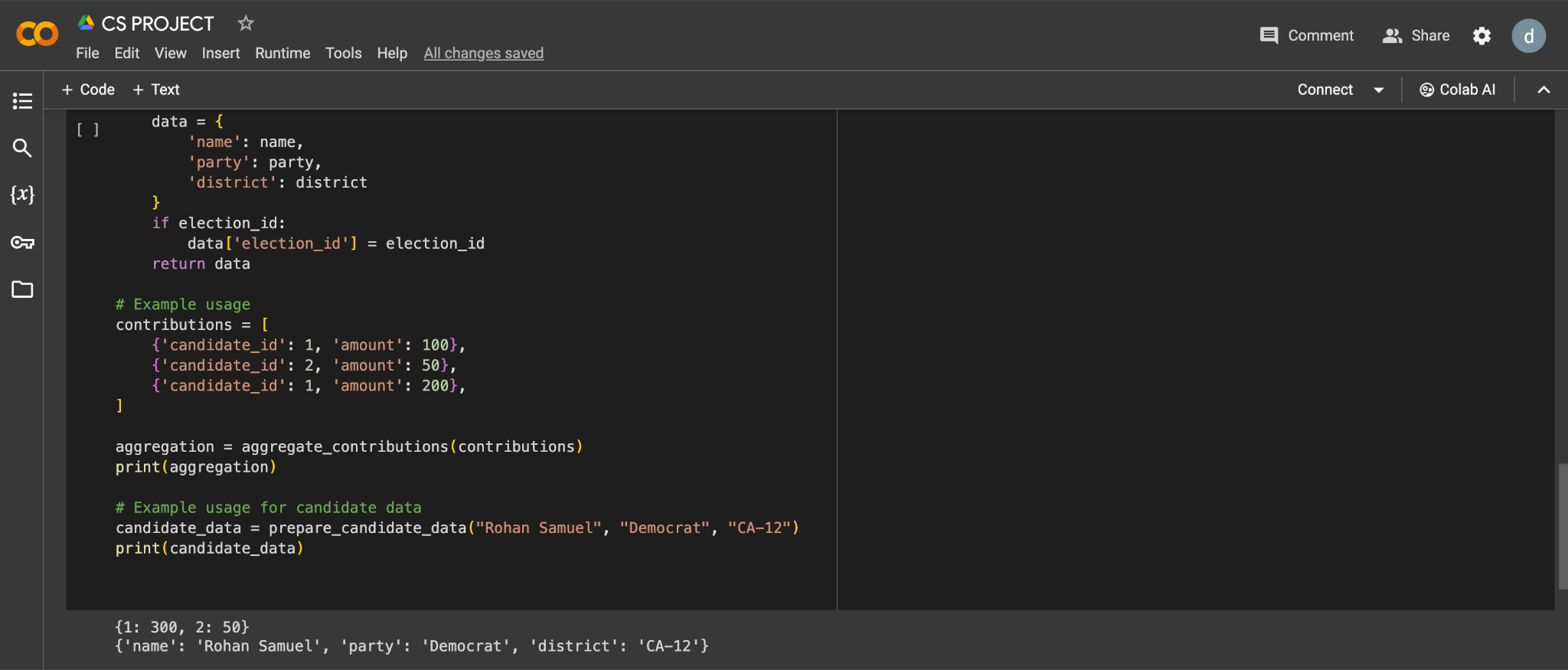
prepare\_candidate\_data(name, party, district, election\_id=None):

This function helps prepare data for a new candidate record. Here's what it does:

* data = {}: Initializes an empty dictionary to store the candidate information.
* data['name'] = name: Sets the 'name' key in the dictionary with the provided name argument.
* data['party'] = party: Sets the 'party' key with the provided party argument.
* data['district'] = district: Sets the 'district' key with the provided district argument.
* if election\_id: Checks if an optional election\_id argument is provided.
  + data['election\_id'] = election\_id: If provided, it adds the 'election\_id' key with the corresponding value.
* return data: Returns the dictionary containing the prepared candidate data.

These functions offer functionalities for processing contribution data and preparing candidate information for potential use with your election database class.

**Test Result:**



1. **Search Algorithms:**

The provided code defines a function search\_candidates that allows users to search for candidates by name. Here's a breakdown of its functionality:

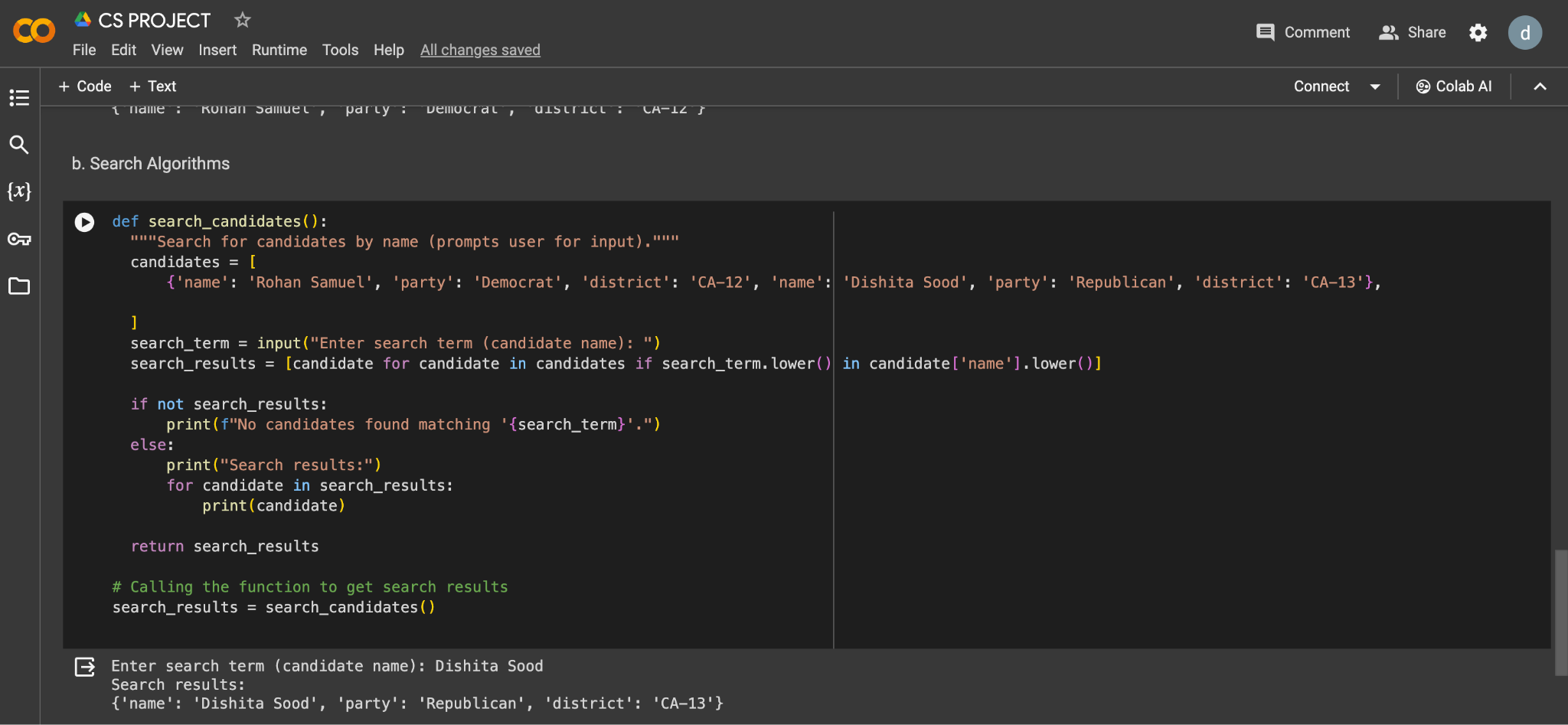
1. candidates = [] (placeholder): This line is a placeholder where you would define or load your actual list of candidates. Each candidate should be a dictionary containing information like name, party, and district.
2. search\_term = input("Enter search term (candidate name): "): Prompts the user to enter a search term (candidate name).
3. search\_results = [candidate for candidate in candidates if search\_term.lower() in candidate['name'].lower()]:
   * This line performs the search using list comprehension.
   * It iterates through each candidate in the candidates list.
   * It converts both the search term and candidate name to lowercase for case-insensitive search using lower().
   * It checks if the search term is present within the candidate's name using the in operator.
   * If a match is found, the candidate dictionary is added to the search\_results list.
4. if not search\_results:: Checks if the search\_results list is empty (no matches found).
   * print(f"No candidates found matching '{search\_term}'."): If no matches are found, it prints a message informing the user.
5. else:: If there are search results...
   * print("Search results:"): Prints a message indicating the search results.
   * for candidate in search\_results:: Loops through each candidate in the search\_results list.
     + print(candidate): Prints the details of each matching candidate.
6. return search\_results: The function returns the search\_results list. This can be useful if you want to process the search results further outside the function.

Note:

* Remember to replace the candidates = [] placeholder with your actual list of candidates or a method to load them.

This function provides a basic search functionality for finding candidates based on name. You can extend it to allow searching by other criteria like party or district

**Test Result:**

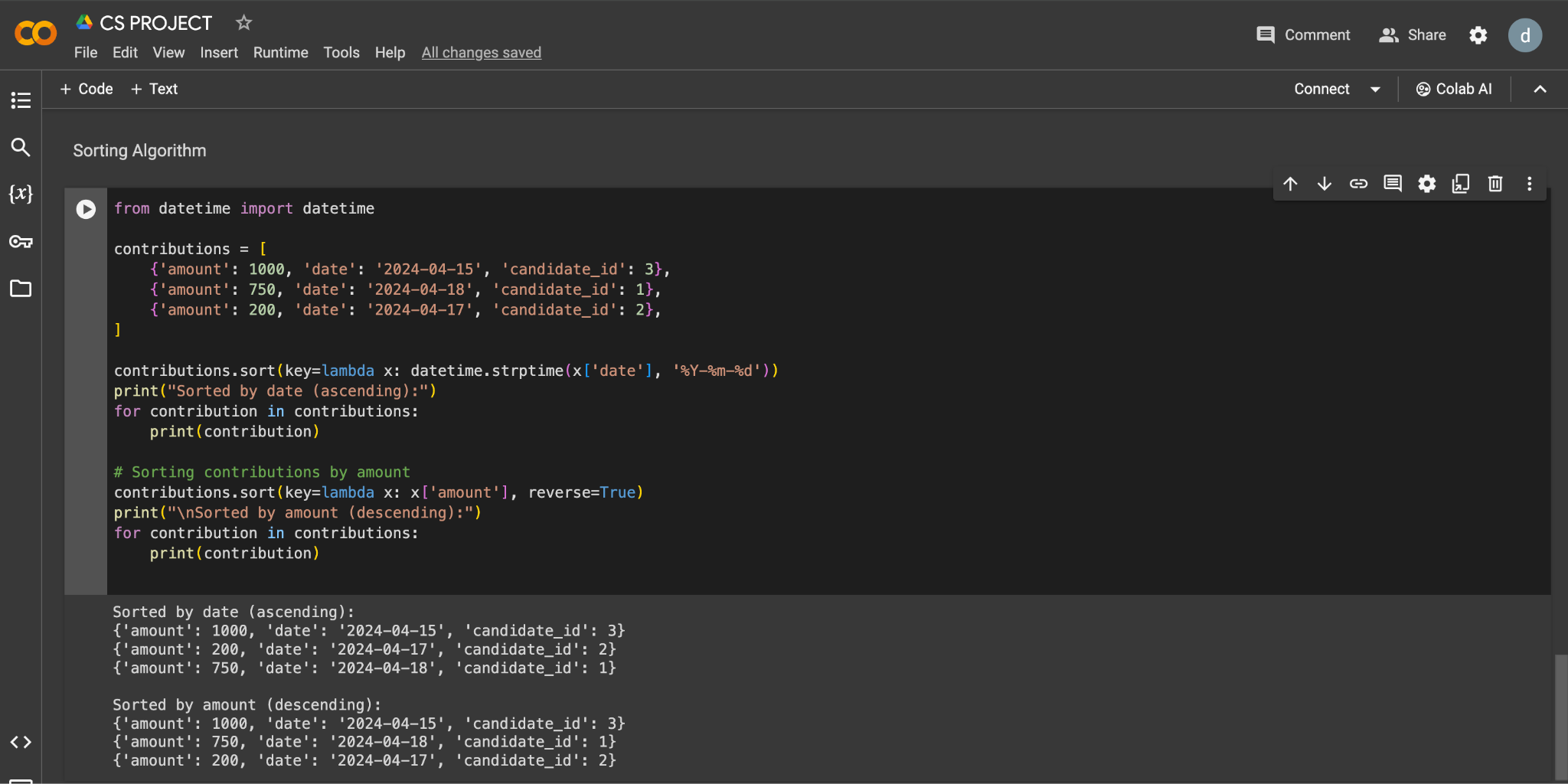


C. **Sorting Algorithms:**

The provided code demonstrates sorting contribution data based on date and amount. Here's a breakdown of its functionality:

1. Import and Data:
   * from datetime import datetime\*\*: Imports the datetime module for working with date and time.
   * contributions = [...]\*\*: This defines a list named contributions containing sample contribution data. Each contribution is a dictionary with 'amount', 'date', and 'candidate\_id' keys. Replace this with your actual contribution data.
2. Sorting by Date (Ascending):
   * contributions.sort(key=lambda x: datetime.strptime(x['date'], '%Y-%m-%d'))\*\*: This line sorts the contributions list based on the date.
     + sort method is used for sorting the list.
     + key argument specifies a function to define the sorting criteria.
     + The provided lambda function takes a contribution (x) as input.
     + Inside the lambda function:
       - datetime.strptime(x['date'], '%Y-%m-%d') converts the date string in the contribution dictionary (x['date']) to a datetime object using the specified format (%Y-%m-%d).
     + The sorting happens based on the resulting datetime object, effectively sorting by date in ascending order (earlier dates come first).
   * print("Sorted by date (ascending):"): Prints a message indicating the sorted order.
   * The for loop iterates through the sorted contributions list and prints each contribution dictionary.
3. Sorting by Amount (Descending):
   * contributions.sort(key=lambda x: x['amount'], reverse=True)\*\*: This line sorts the contributions list based on the amount in descending order.
     + Similar to the previous sort, it uses the sort method with a lambda function as the key.
     + The lambda function simply returns the amount directly from the contribution dictionary (x['amount']).
     + reverse=True argument specifies that the sorting should be in descending order (higher amounts first).
   * print("\nSorted by amount (descending):"): Prints a message indicating the sorted order.
   * The for loop again iterates through the sorted contributions list and prints each contribution dictionary.

**Test Result:**



**Project Issues:**

* Limited Data: The provided code snippets use sample data for candidates and contributions. In a real-world scenario, you would need to integrate the code with a database or other data source to manage actual election data.
* Security is important when handling sensitive data like contributions. Measures like user authentication and data encryption could be implemented.

**Limitations:**

* Basic Functionalities: The current code offers basic functionalities like adding, retrieving, and searching for candidate and contribution data. More advanced features like filtering by specific criteria or generating reports could be added.
* Scalability: The code might not be optimized for handling very large datasets. As the number of candidates and contributions grows, performance considerations and potential database solutions become more important.

**Application of Course Learnings:**

* Data Structures: The code effectively utilizes dictionaries to organize candidate and contribution data, demonstrating an understanding of data structures.
* Functions: The code defines functions for specific tasks like searching and sorting, showcasing the application of functions for modularity and code reusability.
* List comprehensions: The search functionality utilizes list comprehensions for concise and efficient filtering of data based on a condition.

**Overall Quality of Report**

The provided project demonstrates a good understanding of core programming concepts like data structures, functions, and data manipulation. Overall, this project lays a solid foundation for a comprehensive election data management system.

.