# Case Studies Part II

### Part 1: Testing

Testing your code is an important part of software development. Before you begin writing code, you need to have a plan in place to test that code. Review the specifications listed for both case studies in this document, and write a test plan that will demonstrate that you have successfully incorporated the specifications into your code. The test log should include the test you performed, the expected outcome, and a space for the actual outcome. **Submit your test plans to blackboard.**

### Part 2: Hierarchy Chart/Outline and Pseudocode

As your program gets larger and requires more functions, it gets more difficult to decide what the functions should be and how they should relate to each other. To help you manage your program, you can use a hierarchy chart to plan the modules of the program. For both programs, create a hierarchy chart or outline to plan how you will write the code. Examples of hierarchy charts and outlines is on page 129 of the text.

Once you have your hierarchy chart/outline created, write pseudocode for the programs. Examples of how to write pseudocode can be found on page 95 of the text. **Submit your hierarchy chart/outline and pseudocode to Blackboard.**

### Part 3: Write the Code

Using your test plan, hierarchy chart/outline, and your pseudocode, write the code for **ONE** of the two case studies. Insert a screenshot of the console from the program into this document. **Include screenshots of the program handling errors such as file not found, invalid integer, invalid float, etc. into this document. Submit this document to Blackboard.**

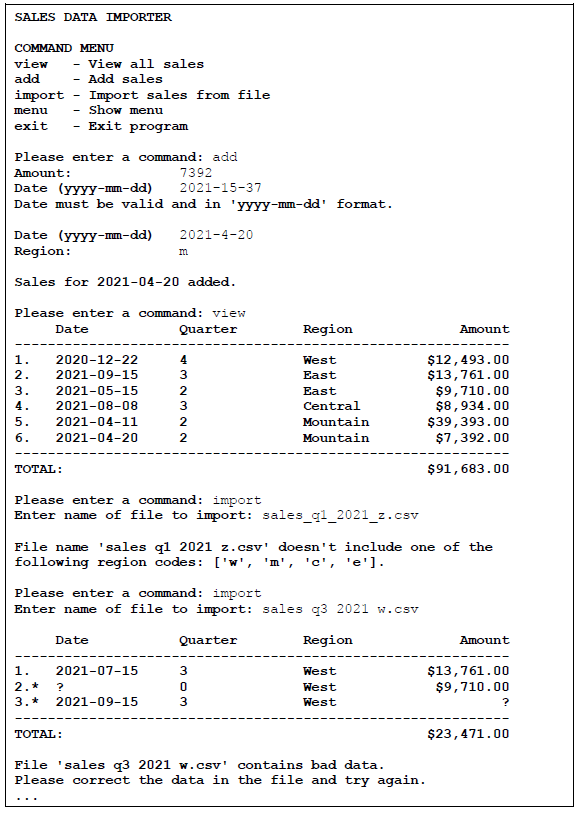
**Submit your completed test log to Blackboard.** Your test log should list the test you performed, the expected outcome, and the actual outcome.

**Submit your python code to Blackboard.** If you are submitting more than one file for the case study, zip the python files and submit the zip file.

These case studies take the place of a final exam. They are open book, open notes, open Google, but you may NOT collaborate with classmates on the solutions. By submitting this document to Blackboard, you are stating that the work contained in the file and in the programs is your own.

## Case Study 1: Data Importer

Create a program that imports sales data from one or more CSV files and allows the user to add sales data. Include error handling to check for invalid data entered into the program. The menu should look like this:



Specifications

1. The program should accept float entries for the amount.

2. Use the decimal module to make sure the program doesn’t yield incorrect total sales due to floating-point errors.

3. Store the functions for getting sales data from a user in a module named sales. Use docstrings and type hints to document the sales module.

4. Create a menu that allows the user to view existing sales data, add new sales data, and import sales data.

5. Calculate the quarter based on the month value:

* + 1, 2, 3: Quarter 1
  + 4, 5, 6: Quarter 2
  + 7, 8, 9: Quarter 3
  + 10, 11, 12: Quarter 4

6. Allow users to import sales data from a CSV file that contains sales amount, year, month, and day. Use a text file to keep track of imported files so a user can’t import a file more than once.

7. Store the functions for writing and reading the files in a separate module named db.py.

8. When the program starts, it should read the sales data from a file named all\_sales.csv.

9. When the program ends, it should write the sales data, including any imported data and any data entered by the user, to all\_sales.csv.

10. Don’t store the quarter value. Instead, calculate it from the month as needed.

11. Handle the exception that occurs if the program can’t find the all\_sales.csv file, the file the user is trying to import, or the text file that tracks imported files.

12. Handle the exceptions that occur if the user enters a string or float where an integer is expected, or a string where a float is expected.

13. Handle the exceptions that occur if the data in the imported file can’t be converted to a float or an int. Use asterisks and question marks to notify the user of bad imported data.

* The program should validate the user’s entries as follows:

14. The sales amount should be greater than zero.

15. The sales month should be between 1 and 12.

16. The sales day should be between 1 and 31 **for months 1, 3, 5, 10, and 12**.

17. For months 4, 6, 9, and 11, the maximum day value is 30.

18. For month 2, the maximum day value is 28 (don’t handle leap years).

19. The sales year should be greater than or equal to 2000 and less than or equal to 9999.

20. Use the locale module to display the sales amount using formatting that’s appropriate for your current locale.

21. Use f-strings to format the widths and alignment of the columns in the display table.

22. Use string operations to ensure the name of an imported file follows this format: *sales\_qn\_yyyy\_r.csv*, where n represents the sales quarter, yyyy represents the sales year, and r represents the sales region.

23. Valid regions are w for West, m for Mountain, c for Central, and e for East.

24. The program should get the sales region code from the filename and display and store it along with the other sales data.

25 The program should also get a valid sales region from the user when they add sales data with the ‘add’ command.

26 The imported CSV files should have one value for the sales date in YYYY-MM-DD format, rather than individual year, month, and day values.

27. The year, month, and day should be entered by the user as a single entry in YYYYMM-DD format, rather than as individual year, month, and day values.

28. When a user enters an invalid date or when an imported file contains an invalid date, the user should be given the opportunity to correct the date.

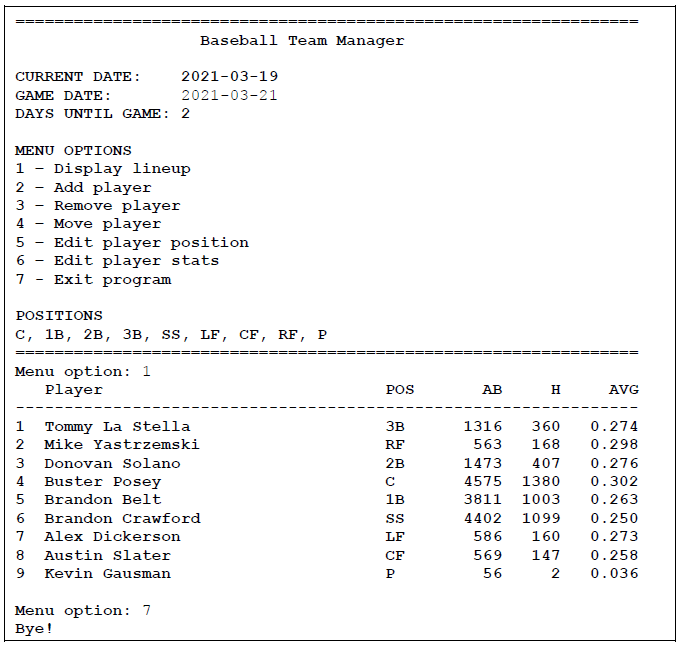
29. Store the sales data as a list of dictionaries rather than a list of lists. To save the sales data to a CSV file, you’ll need to convert it back to a list of lists.

30. Store the valid regions in a dictionary with the region code (w, m, c, or e) as the key and the region name (West, Mountain, Central, or East) as the value.

31. Use the regions dictionary to display the region name rather than the region code.

## Case Study 2 – Baseball Team Management

Create a program that lets the manager of a baseball team keep the data for each player and list and display the lineup for a baseball game. Include error handling to check for invalid data entered into the program. The menu should look like this:



**Specifications**

1. The formula for calculating batting average is: **average = hits / at\_bats**

2. The program should round batting average to a maximum of three decimal places.

3 Use functions to organize the code to make it more reusable, easier to read, and easier to maintain.

4. If the user enters an invalid menu option, display an error message and display the menu again so the user can clearly see the valid menu options.

5. Make sure the user can’t enter data that doesn’t make sense (such as a negative number of hits or at bats, or the player having more hits than at bats).

6. When entering/editing positions, the program should always require the user to enter a valid position.

7. Store the functions for writing and reading the file of players in a separate module named db.py.

8. Handle the exception that occurs if the program can’t find the data file.

9. Handle the exceptions that occur if the user enters a string where an integer is expected.

10 . Handle the exception that occurs if the user enters zero for the number of at bats. In that case, the player’s batting average should be 0.0.

11. Use the repetition operator to display separator lines that use 64 characters.

12. Use spaces, not tabs, to align the columns of data for the players.

13. Make sure the program always displays the batting average with 3 decimal places.

14. Display the positions by processing the tuple of valid positions. (C, 1B, 2B, etc).

15. When the program starts, use the YYYY-MM-DD format to display the current date and to get the date of the next game from the user.

16. Make sure the user enters the date in the correct format.

17. Only display the number of days until the game if the game is in the future. Don’t display the number if the game date is in the past or the user doesn’t enter a date.

18. Use a dictionary to store the data for each player. To get this to work, you need to modify the functions that read and write the data to the file so they work correctly with a list of dictionaries. That’s because the previous version of this program used the csv module to work with a list of lists, but the csv module doesn’t work with a list of dictionaries.