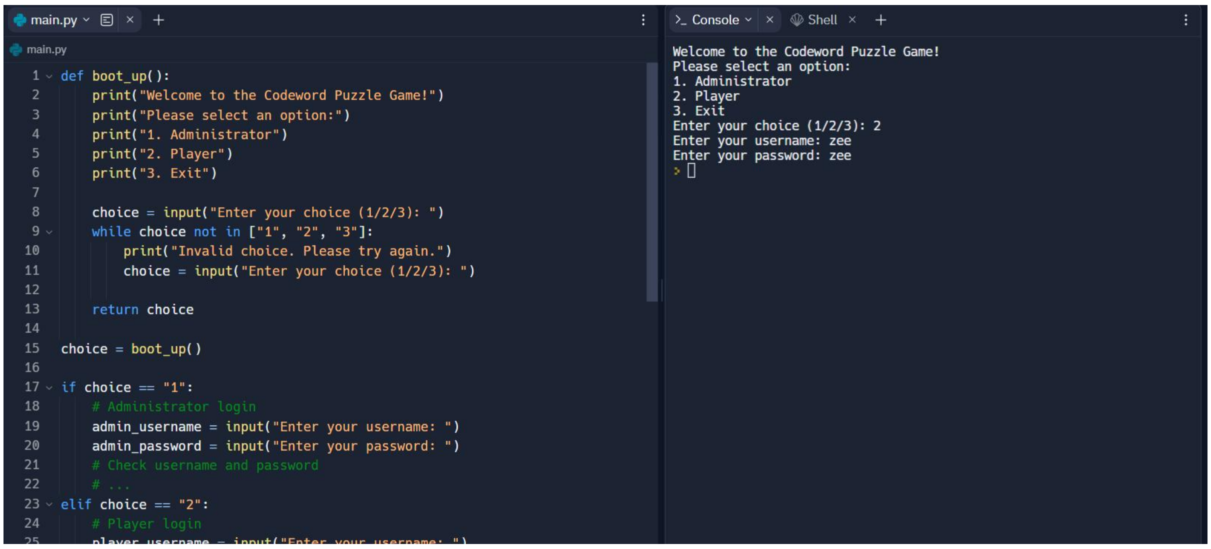
3 - Basic program logic :

3.1 Boot up -

This code defines a function called boot_up() that displays a menu of options for the user to select. The function returns the user's choice, which is then used to determine whether to login as an administrator or player, or to exit the program. If the user chooses to login, they are prompted to enter their username and password.

Output -



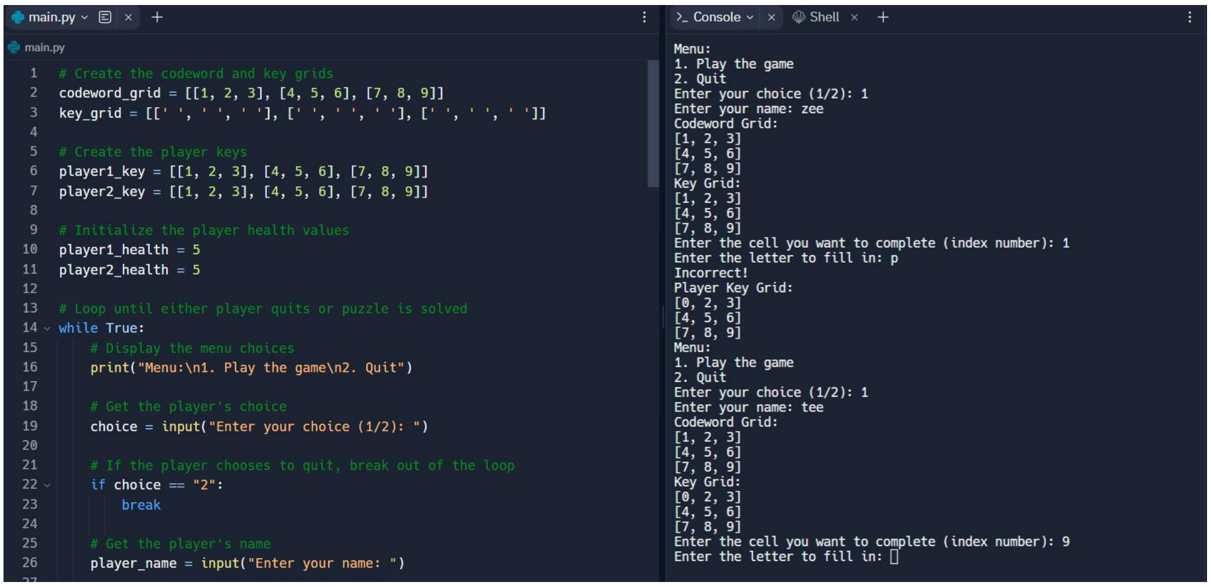
```python
def boot_up():
    print("Welcome to the Codeword Puzzle Game!")
    print("Please select an option:")
    print("1. Administrator")
    print("2. Player")
    print("3. Exit")

    choice = input("Enter your choice (1/2/3): ")
    while choice not in ["1", "2", "3"]:
        print("Invalid choice. Please try again.")
        choice = input("Enter your choice (1/2/3): ")

    return choice

choice = boot_up()

if choice == "1":
    # Administrator login
    admin_username = input("Enter your username: ")
    admin_password = input("Enter your password: ")
    # Check username and password
    # ...
elif choice == "2":
    # Player login
    player_username = input("Enter your username: ")
```

```
Welcome to the Codeword Puzzle Game!
Please select an option:
1. Administrator
2. Player
3. Exit
Enter your choice (1/2/3): 2
Enter your username: zee
Enter your password: zee
> 
```

3.2 - Player's Logic :

This code creates a word puzzle game where two players take turns filling in letters on a key grid to reveal a hidden message. Each player has a separate key grid that they must fill in correctly. The game checks if the player has enough health to play, displays the codeword and key grids, prompts the player to enter the cell they want to complete using index numbers, and checks if the letter they entered is correct. If the letter is correct, the player earns 2 health points and the letter is added to their key grid. If the letter is incorrect, the player loses 2 health points and the letter is removed from their key grid. The game ends when one player correctly fills in their entire key grid, and player statistics are displayed at the end of the game, including the number of correct and incorrect guesses and each player's health value.

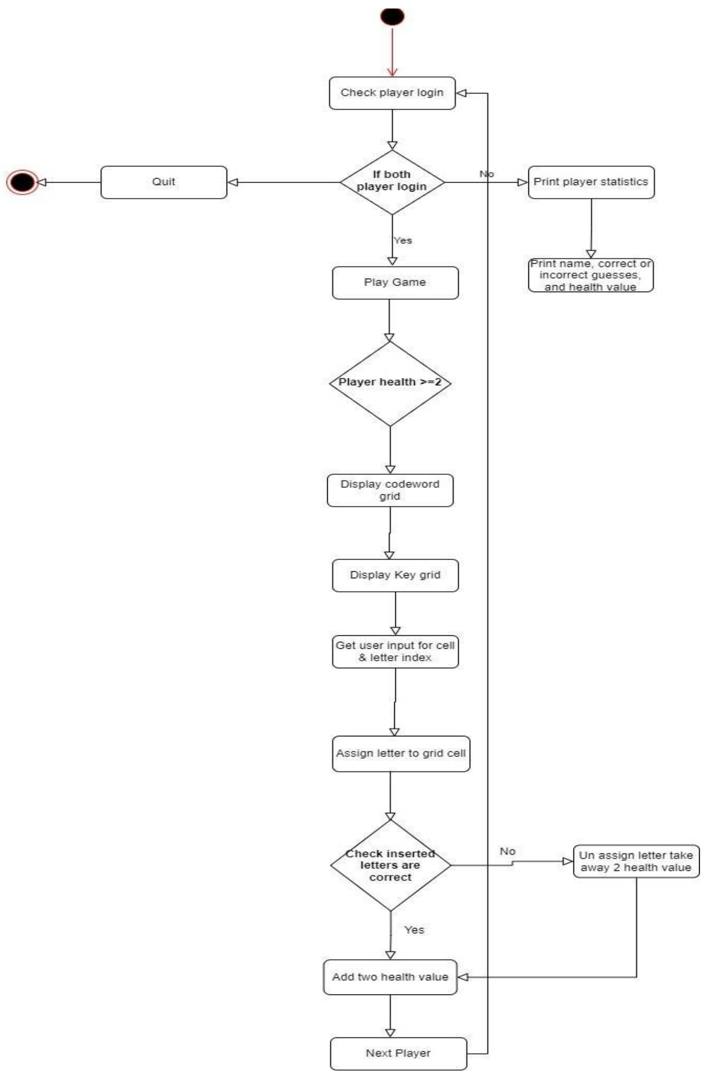
Output -



```python
# Create the codeword and key grids
codeword_grid = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
key_grid = [[' ', ' ', ' '], [' ', ' ', ' '], [' ', ' ', ' ']]

# Create the player keys
player1_key = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
player2_key = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

# Initialize the player health values
player1_health = 5
player2_health = 5

# Loop until either player quits or puzzle is solved
while True:
    # Display the menu choices
    print("Menu:\n1. Play the game\n2. Quit")

    # Get the player's choice
    choice = input("Enter your choice (1/2): ")

    # If the player chooses to quit, break out of the loop
    if choice == "2":
        break

    # Get the player's name
    player_name = input("Enter your name: ")
```

```
Menu:
1. Play the game
2. Quit
Enter your choice (1/2): 1
Enter your name: zee
Codeword Grid:
[1, 2, 3]
[4, 5, 6]
[7, 8, 9]
Key Grid:
[1, 2, 3]
[4, 5, 6]
[7, 8, 9]
Enter the cell you want to complete (index number): 1
Enter the letter to fill in: p
Incorrect!
Player Key Grid:
[0, 2, 3]
[4, 5, 6]
[7, 8, 9]
Menu:
1. Play the game
2. Quit
Enter your choice (1/2): 1
Enter your name: tee
Codeword Grid:
[1, 2, 3]
[4, 5, 6]
[7, 8, 9]
Key Grid:
[0, 2, 3]
[4, 5, 6]
[7, 8, 9]
Enter the cell you want to complete (index number): 9
Enter the letter to fill in: 
```

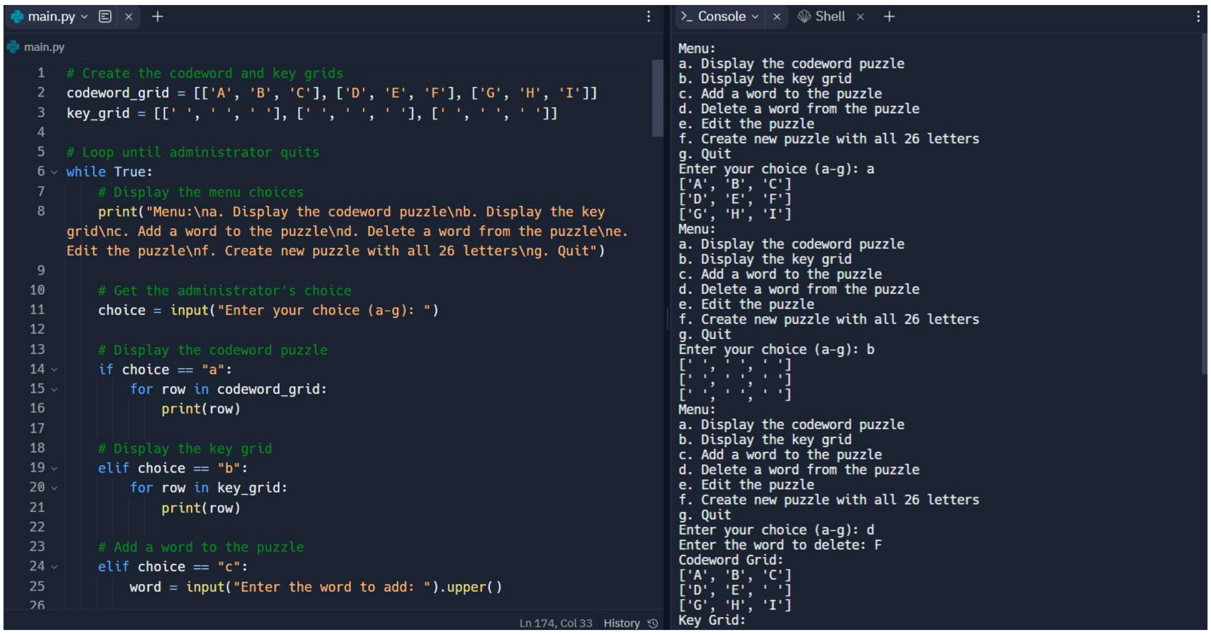
Flowchart as in Milestone 1 -


Check player login
If both player login
No
Print player statistics
Print name, correct or incorrect guesses, and health value
Quit
Yes
Play Game
Player health >=2
Display codeword grid
Display Key grid
Get user input for cell & letter index
Assign letter to grid cell
Check inserted letters are correct
No
Un assign letter take away 2 health value
Yes
Add two health value
Next Player

4 - Administrator logic :

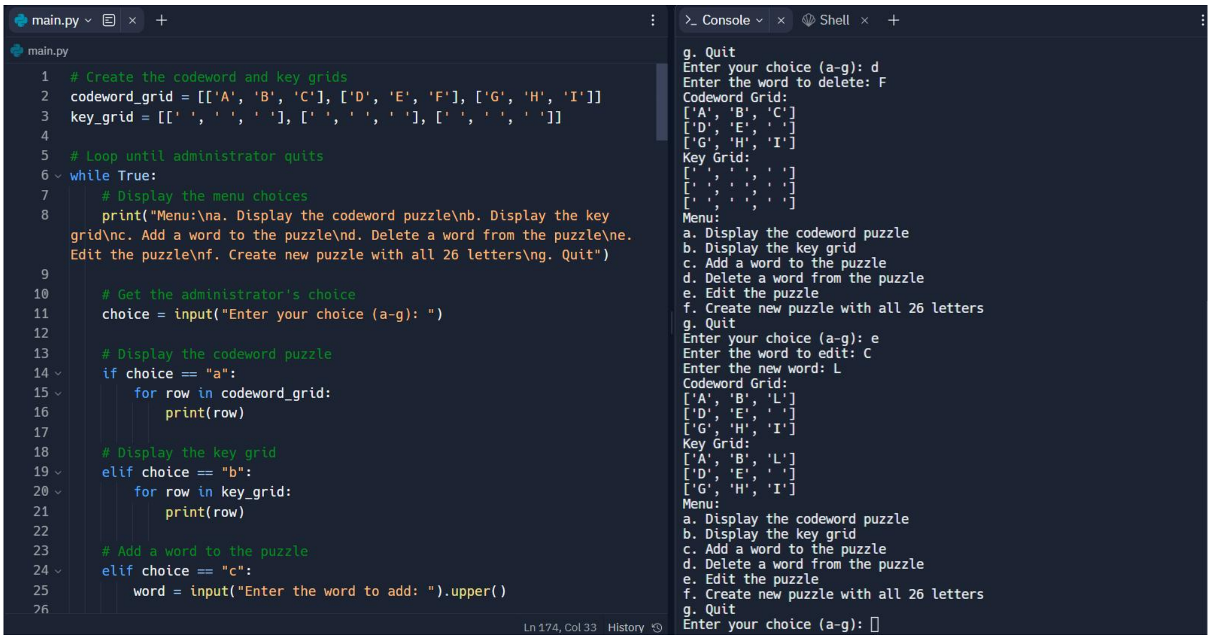
This code implements a menu-based system for an administrator to manage a word puzzle game. The menu provides options to display the codeword puzzle and key grid, add and delete words from the puzzle, edit the puzzle, and create a new puzzle with all 26 letters. The code handles input validation to ensure that only valid words can be added or edited, and updates both the codeword and key grids as necessary. The administrator can choose to quit the program at any time.

Output -



```python
# Create the codeword and key grids
codeword_grid = [['A', 'B', 'C'], ['D', 'E', 'F'], ['G', 'H', 'I']]
key_grid = [[' ', ' ', ' '], [' ', ' ', ' '], [' ', ' ', ' ']]

# Loop until administrator quits
while True:
    # Display the menu choices
    print("Menu:\na. Display the codeword puzzle\nb. Display the key grid\nc. Add a word to the puzzle\nd. Delete a word from the puzzle\ne. Edit the puzzle\nf. Create new puzzle with all 26 letters\ng. Quit")

    # Get the administrator's choice
    choice = input("Enter your choice (a-g): ")

    # Display the codeword puzzle
    if choice == "a":
        for row in codeword_grid:
            print(row)

    # Display the key grid
    elif choice == "b":
        for row in key_grid:
            print(row)

    # Add a word to the puzzle
    elif choice == "c":
        word = input("Enter the word to add: ").upper()
```

```
Menu:
a. Display the codeword puzzle
b. Display the key grid
c. Add a word to the puzzle
d. Delete a word from the puzzle
e. Edit the puzzle
f. Create new puzzle with all 26 letters
g. Quit
Enter your choice (a-g): a
['A', 'B', 'C']
['D', 'E', 'F']
['G', 'H', 'I']
Menu:
a. Display the codeword puzzle
b. Display the key grid
c. Add a word to the puzzle
d. Delete a word from the puzzle
e. Edit the puzzle
f. Create new puzzle with all 26 letters
g. Quit
Enter your choice (a-g): b
[' ', ' ', ' ']
[' ', ' ', ' ']
[' ', ' ', ' ']
Menu:
a. Display the codeword puzzle
b. Display the key grid
c. Add a word to the puzzle
d. Delete a word from the puzzle
e. Edit the puzzle
f. Create new puzzle with all 26 letters
g. Quit
Enter your choice (a-g): d
Enter the word to delete: F
Codeword Grid:
['A', 'B', 'C']
['D', 'E', ' ']
['G', 'H', 'I']
Key Grid:
```



```python
# Create the codeword and key grids
codeword_grid = [['A', 'B', 'C'], ['D', 'E', 'F'], ['G', 'H', 'I']]
key_grid = [[' ', ' ', ' '], [' ', ' ', ' '], [' ', ' ', ' ']]

# Loop until administrator quits
while True:
    # Display the menu choices
    print("Menu:\na. Display the codeword puzzle\nb. Display the key grid\nc. Add a word to the puzzle\nd. Delete a word from the puzzle\ne. Edit the puzzle\nf. Create new puzzle with all 26 letters\ng. Quit")

    # Get the administrator's choice
    choice = input("Enter your choice (a-g): ")

    # Display the codeword puzzle
    if choice == "a":
        for row in codeword_grid:
            print(row)

    # Display the key grid
    elif choice == "b":
        for row in key_grid:
            print(row)

    # Add a word to the puzzle
    elif choice == "c":
        word = input("Enter the word to add: ").upper()
```

```
g. Quit
Enter your choice (a-g): d
Enter the word to delete: F
Codeword Grid:
['A', 'B', 'C']
['D', 'E', ' ']
['G', 'H', 'I']
Key Grid:
[' ', ' ', ' ']
[' ', ' ', ' ']
[' ', ' ', ' ']
Menu:
a. Display the codeword puzzle
b. Display the key grid
c. Add a word to the puzzle
d. Delete a word from the puzzle
e. Edit the puzzle
f. Create new puzzle with all 26 letters
g. Quit
Enter your choice (a-g): e
Enter the word to edit: C
Enter the new word: L
Codeword Grid:
['A', 'B', 'L']
['D', 'E', ' ']
['G', 'H', 'I']
Key Grid:
['A', 'B', 'L']
['D', 'E', ' ']
['G', 'H', 'I']
Menu:
a. Display the codeword puzzle
b. Display the key grid
c. Add a word to the puzzle
d. Delete a word from the puzzle
e. Edit the puzzle
f. Create new puzzle with all 26 letters
g. Quit
Enter your choice (a-g): 
```

5 - Extension :

Based on the initial project requirements, I have brainstormed several possible extension ideas that I believe would enhance the game of codewords and demonstrate my independent learning and creativity. Here are my top five extension suggestions:

i) Use of more advanced data structures/coding techniques (dictionary, classes etc…):

Explanation: Using advanced data structures and coding techniques can make the game more efficient, maintainable, and extensible. For example, using classes can help to organize the code and make it more modular, while using dictionaries can allow for faster lookups and updates.

Practical Implementation: Define classes for the game, player, and AI opponent, and use dictionaries to store the game state and key grid. Use inheritance and polymorphism to make the code more modular and to minimize duplication. For example, the game and AI opponent classes could inherit from a common base class, and the player class could inherit from the game class. Use debugging tools and strategies such as print statements and breakpoint debugging to test and troubleshoot the code.

ii) Extend the game by generating new games automatically using a selection of words from either a hand written list or (even better) read from a downloaded file of words found online/a dictionary coding library:

Explanation: Allowing for automatically generated games can make the game more dynamic and interesting, as players will encounter new puzzles every time they play. Reading in words from a file or online source can also allow for a greater variety of word choices.

Practical Implementation: Implement a feature that allows the game to read in a file of words and generate a new game based on a random selection of words from the file. Allow players to choose from different word lists or categories, such as animals, cities, or sports teams. Use file I/O and error handling to ensure that the input is valid and that the game can be generated successfully. Use version control tools such as Git to manage changes to the code and to collaborate with other team members.

iii) Reading and writing to a file of words/player statistics:

Explanation: Storing player statistics and game progress can allow players to track their performance over time and to resume a game at a later time. Reading in words from a file can also allow for greater customization and variety in the game.

Practical Implementation: Implement a feature that allows the game to read in a file of words or player statistics, and write out new files as necessary. Store player statistics such as win/loss record, average time to solve a puzzle, and high scores. Allow players to save and load their game progress, so they can resume a game at a later time. Use file I/O and error handling to ensure that the input is valid and that the data is saved and loaded correctly. Use unit testing tools such as Pytest to test the functionality of the file I/O and error handling code.

iv) Plotting a graph of some player activities:

Explanation: Plotting player activities can provide players with a visual representation of their progress and can motivate them to improve their performance. It can also allow for comparisons between different players and different games.

Practical Implementation: Implement a feature that tracks player activities, such as time spent playing, number of games won/lost, and average time to solve a puzzle. Plot the data using a graphing library such as Matplotlib or Plotly, allowing players to visualize their progress and compare their performance to others. Implement features such as filtering by time period, player name, or game difficulty, to allow for more customized analyses. Use data cleaning and preprocessing techniques such as outlier detection and normalization to ensure that the data is accurate and meaningful.

v) Develop the whole code in Pygame:

Explanation: Using Pygame to create a graphical user interface can make the game more engaging and immersive, and can allow for greater customization and interactivity.

Practical Implementation: Use Pygame to create a graphical user interface for the game, including buttons, text fields, and graphical elements such as the codeword and key grids. Implement animations and sound effects to make the game more engaging and immersive. Allow players to customize the look and feel of the game, such as choosing from different color schemes or themes. Use Pygame's event handling and game loop functionality to respond to player input and update the game state. Use code profiling tools such as cProfile or PyCharm's built-in profiler to identify and optimize performance bottlenecks in the Pygame code.