Approach and Concept:
The code for the vending machine problem is to return the least number of notes back to the customer after buying an item. In my code, I applied the dynamic programming concept to find the number of notes that can be given as change for a specific payment. The reason why I chose to apply Dynamic Programming as my approach is because this topic was part of my syllabus when I was studying in Monash. Dynamic Programming is a very powerful optimization technique in computer science.

Core Idea and Strategy for Dynamic Programming:

1. **Overlapping Subproblems:** The problem of finding the minimum number of notes for a change can be broken down into smaller subproblems (i.e., finding the minimum number of notes for all amounts less than the given change). These subproblems overlap, as the solution to a larger problem depends on the solutions to its smaller subproblems.

2. **Memoization:** The solutions to these subproblems are stored in the memo and memo_notes arrays. This ensures that each subproblem is solved only once and its result is reused if needed, saving computation time.

3. **Optimal Substructure:** The problem exhibits optimal substructure, meaning the optimal solution to the problem can be constructed from the optimal solutions of its subproblems. This is evident in how we calculate memo[i] and memo_notes[i] for all i.

4. **Building the Solution:** Finally, we can now build the solution to the original problem (i.e., finding the least amount of notes back to the customers) by using the solutions to the subproblems stored in memo and memo_notes.

Summary of the Python Code:
The vending_machine function takes in two input arguments.
The first input argument is the payment variable, the payment variable represents the amount of money given to the vending machine.
The second input argument is the name of the selected beverage.

The function calculates how much change needs to be given by subtracting the price of the beverage from the payment. Then initialize two memos. The first memo will store the minimum number of notes needed for each amount of change from 0 to change value. memo_notes will store the last note used for each amount of change. After building up the memo and memo_notes, for each amount of change from the smallest note to change, it finds the minimum number of notes needed and the last note used. Now, calculate which notes to return as change. It starts with the total change and subtracts each note used from it until no change is left. The notes used are stored in change_notes. Finally, the code returns two things, the minimum number of notes needed for the total change and a list of notes to be given as change.