

**Name : Ahmad Raza**

**Roll nob : 093**

1. **Class definition**
   * class MovieBudgetAnalyzer:  
     Declares a new class named MovieBudgetAnalyzer. A class groups data (movies) and functions that operate on that data.
2. **Constructor / initializer**
   * def \_\_init\_\_(self):  
     This method runs automatically when a new object of the class is created.
   * Inside \_\_init\_\_ the program defines self.movies, an instance attribute that holds the initial dataset:
   * self.movies = [
   * ("Dangal", 70000000),
   * ("Baahubali 2", 250000000),
   * ("PK", 85000000),
   * ("Kabhi Khushi Kabhie Gham", 55000000),
   * ("Sholay", 30000000),
   * ("Padmaavat", 215000000),
   * ("Sanju", 100000000)
   * ]

Each item is a tuple (name, budget) where name is a string and budget is an integer. self.movies is available to every method of the object.

1. **Method to add movies**
   * def add\_movies(self):  
     This method asks the user how many movies they want to add and appends those movies to self.movies.
   * num = int(input("How many movies do you want to add: "))  
     Reads user input and converts it to an integer (num). If the user types invalid text, int() will raise a ValueError.
   * for i in range(num):  
     Repeats the following block num times.
   * Inside the loop:
     + name = input("Enter movie name: ") reads the movie title (string).
     + budget = int(input("Enter movie budget: ")) reads the budget and converts it to an integer (again may raise ValueError on invalid input).
     + self.movies.append((name, budget)) stores the new movie as a tuple at the end of the list.
2. **Method to calculate average budget**
   * def calculate\_average\_budget(self):  
     Computes and returns the mean budget of all movies currently in self.movies.
   * total\_budget = 0 initializes a running total.
   * for movie in self.movies: loops over each movie tuple.
   * total\_budget += movie[1] adds the budget (second element of the tuple) to the total.
   * return total\_budget / len(self.movies) divides the total by the number of movies and returns the result. The returned value is a float. (If self.movies were empty, this would raise ZeroDivisionError, but the code starts with a non-empty initial list.)
3. **Method to find and print movies above average**
   * def find\_budget(self, average\_budget):  
     Accepts one argument, average\_budget, and prints all movies whose budget is greater than that average.
   * print("Movies with budget higher than average:") prints a header.
   * count = 0 initializes a counter for how many movies exceed the average.
   * for movie in self.movies: iterates through movies.
   * name, budget = movie unpacks the tuple into name and budget.
   * if budget > average\_budget: checks whether the current movie’s budget is strictly greater than the average.
   * print(f"{name} — higher by {budget - average\_budget}") prints the movie name and how much higher its budget is than the average. The difference is usually a float because average\_budget is a float.
   * count += 1 increments the counter for each movie above average.
   * After the loop, print("Total movies with budget higher than average:", count) prints the final count.
4. **Program execution (top-level code)**
   * analyzer = MovieBudgetAnalyzer() creates an instance of the class. This runs \_\_init\_\_ and sets up the initial movies list.
   * analyzer.add\_movies() calls the method that asks the user to add more movies (if any) and updates self.movies.
   * average = analyzer.calculate\_average\_budget() computes the average budget and stores it in average.
   * print("Average Budget:", average) prints the computed average.
   * analyzer.find\_budget(average) prints all movies with budgets higher than average and the total count.
5. **Data types and return values**
   * Movie budgets are stored as integers.
   * calculate\_average\_budget() returns a float (average).
   * When printing budget - average\_budget, the result will typically be a float.
6. **Edge cases and possible errors**
   * If the user inputs a non-integer for the number of movies or a movie budget, int() raises ValueError. You can handle this with try/except for a friendlier program.
   * If self.movies were ever empty, dividing by len(self.movies) would raise ZeroDivisionError. The current initial list prevents this, but it’s good to be aware of the risk.
   * The comparison uses > (strictly greater). Movies with budget exactly equal to the average will **not** be printed. Use >= if you want to include equals.
7. **Small suggested improvements (optional)**
   * Add input validation (wrap int() conversions in try/except and re-prompt on invalid input).
   * Format large numbers for readability: f"{value:,.0f}" will show 135,000,000 instead of 135000000.0.
   * Rename find\_budget to find\_high\_budget\_movies for clearer meaning.