1. Head: `file_path = './boston_dataset.csv'`

Tail: `data = pd.read_csv(file_path)`
Explanation:
This section defines the path to the dataset file (`boston_dataset.csv`) and then loads the dataset into a pandas DataFrame. The `pd.read_csv(file_path)` function reads the CSV file from the given path and stores the data in the variable `data`. The DataFrame holds the dataset, and we can easily manipulate and analyze it.

---

2. Head: `print(data.head())`

Tail: `print("Missing Values:\n", data.isnull().sum())`
Explanation:
The `data.head()` function is used to print the first 5 rows of the dataset. This helps to quickly inspect the structure and contents of the data. The `data.isnull().sum()` function checks for any missing (NaN) values in the dataset and sums the count for each column, displaying it. If there are any missing values, it will show how many are in each column.

---

3. Head: `sns.set(rc={'figure.figsize': (11.7, 8.27)})`

Tail: `plt.show()`
Explanation:
The `sns.set()` function is used to set the default figure size for all Seaborn plots, which makes the visualizations easier to interpret. Here, the figure size is set to `(11.7, 8.27)` inches. Then, `sns.histplot(data['medv'], bins=30, kde=True)` creates a histogram to visualize the distribution of the target variable (`medv`), which represents the median value of owner-occupied homes in thousands of dollars. The `kde=True` parameter adds a kernel density estimate (KDE) curve on top of the histogram, providing an estimate of the probability density function. Finally, `plt.show()` renders the plot to the screen.

---

4. Head: `correlation_matrix = data.corr().round(2)`

Tail: `plt.show()`

Explanation:

Here, `data.corr()` computes the correlation matrix for all the numeric columns in the dataset. The correlation matrix shows the relationships between different features, helping to identify which variables are strongly correlated with the target variable. The `.round(2)` function rounds the correlation values to two decimal places for easier interpretation. After that, a heatmap is generated using `sns.heatmap()`, which visualizes the correlation matrix. The `annot=True` argument displays the correlation values on the heatmap, and `cmap='coolwarm'` defines the color palette. The plot is displayed using `plt.show()`.

---

5. Head: `features = ["rm", "lstat"]`

Tail: `plt.tight_layout()`

Explanation:

This line defines a list of features (`rm` for the average number of rooms per dwelling and `lstat` for the percentage of lower status population) to be used for prediction. The following code block generates scatter plots to visualize the relationships between each selected feature and the target variable (`medv`). It uses a loop to create individual plots for each feature. `plt.tight_layout()` adjusts the layout to prevent overlapping of the plots and makes the display cleaner.

---

6. Head: `X = data[["rm", "lstat"]]`

Tail: `Y = data["medv"]`

Explanation:

In this part, `X` contains the input features (independent variables), and `Y` contains the target variable (dependent variable). The `X` DataFrame includes the selected features, `rm` and `lstat`, while `Y` contains the target, `medv`. These are the variables that will be used to train the linear regression model.

---

7. Head: `X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.2, random_state=42)`

Tail: `model.fit(X_train, Y_train)`

Explanation:

This line splits the dataset into training and testing sets using `train_test_split()`. 80% of the data will be used for training the model, and the remaining 20% will be used for testing. The `random_state=42` ensures the same split each time the code runs, providing reproducibility. After the split, the model is trained using `model.fit(X_train, Y_train)`, where the input data (`X_train`) and the target values (`Y_train`) are used to fit the linear regression model.

---

8. Head: `Y_pred = model.predict(X_test)`

Tail: `mse = mean_squared_error(Y_test, Y_pred)`

Explanation:

After training the model, we use it to make predictions on the test data (`X_test`) with `model.predict(X_test)`. The predicted values are stored in `Y_pred`. Then, the `mean_squared_error()` function from the `sklearn.metrics` module is used to calculate the mean squared error (MSE) between the true target values (`Y_test`) and the predicted values (`Y_pred`). MSE gives an indication of how well the model's predictions match the actual data, with lower values indicating better performance.

---

9. Head: `print(f"Mean Squared Error: {mse:.2f}")`

Tail: (End of the code)

Explanation:

This line prints the Mean Squared Error (MSE) value, which represents the average squared difference between predicted and actual values. The `f"{mse:.2f}"` formats the MSE value to display two decimal places for easier reading. This final output provides a measure of the model's prediction accuracy.