Here's a detailed, line-by-line explanation of the \*\*input\*\* part of this code for each segment:

```python

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

```

1. \*\*`import pandas as pd`\*\*: Imports the `pandas` library, a popular library in Python for data manipulation and analysis, and aliases it as `pd`.

2. \*\*`import numpy as np`\*\*: Imports the `numpy` library, widely used for numerical operations in Python, and aliases it as `np`.

3. \*\*`import matplotlib.pyplot as plt`\*\*: Imports `matplotlib.pyplot`, a plotting library in Python, and aliases it as `plt`.

4. \*\*`import seaborn as sns`\*\*: Imports `seaborn`, a data visualization library built on top of `matplotlib` and designed to make plots more attractive and informative.

```python

data = pd.read\_csv("C:\\Users\\Asus\\Downloads\\sales\_data\_sample.csv", encoding='Latin-1')

```

5. \*\*`data = pd.read\_csv("C:\\Users\\Asus\\Downloads\\sales\_data\_sample.csv", encoding='Latin-1')`\*\*: Reads the CSV file located at the specified path using `pandas.read\_csv()`. The `encoding='Latin-1'` parameter specifies the encoding format, which is necessary here due to special Latin-1 characters that could cause errors with the default UTF-8 encoding. This loads the data into a DataFrame named `data`.

```python

data.head()

```

6. \*\*`data.head()`\*\*: Displays the first five rows of the DataFrame to quickly inspect the data.

```python

data.shape

```

7. \*\*`data.shape`\*\*: Displays the shape of the DataFrame, i.e., the number of rows and columns.

```python

data.isnull().sum()

```

8. \*\*`data.isnull().sum()`\*\*: Counts the number of missing (NaN) values in each column of the DataFrame, giving an overview of data completeness.

```python

data.drop(["ORDERNUMBER", "PRICEEACH", "ORDERDATE", "PHONE", "ADDRESSLINE1", "ADDRESSLINE2", "CITY", "STATE", "TERRITORY", "POSTALCODE", "CONTACTLASTNAME", "CONTACTFIRSTNAME"], axis=1, inplace=True)

```

9. \*\*`data.drop([...], axis=1, inplace=True)`\*\*: Drops specified columns (e.g., `ORDERNUMBER`, `PRICEEACH`, etc.) from the DataFrame. The `axis=1` specifies columns (vs rows), and `inplace=True` modifies the original DataFrame without creating a new copy.

```python

data.isnull().sum()

```

10. \*\*`data.isnull().sum()`\*\* (repeated): After dropping columns, this line is used again to check for any remaining NaN values in the DataFrame.

```python

data.describe()

```

11. \*\*`data.describe()`\*\*: Provides descriptive statistics (like mean, min, max) for numerical columns in the DataFrame, summarizing the central tendency, dispersion, and shape of the dataset's distribution.

```python

sns.countplot(data=data, x='STATUS')

```

12. \*\*`sns.countplot(data=data, x='STATUS')`\*\*: Plots a count plot using Seaborn, showing the frequency of different categories within the `STATUS` column.

```python

sns.histplot(x='SALES', hue='PRODUCTLINE', data=data, element="poly")

```

13. \*\*`sns.histplot(...)`\*\*: Creates a histogram for the `SALES` column, with different colors for each `PRODUCTLINE`, to observe the distribution of sales across different product lines.

```python

data['PRODUCTLINE'].unique()

```

14. \*\*`data['PRODUCTLINE'].unique()`\*\*: Displays unique values within the `PRODUCTLINE` column, identifying the distinct types of product lines.

```python

data.drop\_duplicates(inplace=True)

```

15. \*\*`data.drop\_duplicates(inplace=True)`\*\*: Removes duplicate rows from the DataFrame, with `inplace=True` modifying the original DataFrame.

```python

data.info()

```

16. \*\*`data.info()`\*\*: Provides a concise summary of the DataFrame, including the column data types and counts of non-null values.

```python

list\_cat = data.select\_dtypes(include=['object']).columns.tolist()

```

17. \*\*`list\_cat = data.select\_dtypes(include=['object']).columns.tolist()`\*\*: Creates a list of categorical columns by selecting columns with the data type `object` and converting the column names to a list. Categorical columns often contain text data or labels.

```python

for i in list\_cat:

sns.countplot(data=data, x=i)

plt.xticks(rotation=90)

plt.show()

```

18. \*\*`for i in list\_cat: ...`\*\*: Loops through each categorical column in `list\_cat`, creating a count plot for each. The `plt.xticks(rotation=90)` rotates x-axis labels by 90 degrees to improve readability.

```python

from sklearn import preprocessing

le = preprocessing.LabelEncoder()

```

19. \*\*`from sklearn import preprocessing`\*\*: Imports the `preprocessing` module from the `sklearn` (Scikit-learn) library, which contains functions to transform and normalize data.

20. \*\*`le = preprocessing.LabelEncoder()`\*\*: Initializes a `LabelEncoder`, which encodes categorical text labels as numeric values, suitable for machine learning.

```python

for i in list\_cat:

data[i] = le.fit\_transform(data[i])

```

21. \*\*`for i in list\_cat: data[i] = le.fit\_transform(data[i])`\*\*: Iterates over each categorical column in `list\_cat` and applies the `LabelEncoder` to convert text labels in each column to numeric codes.

```python

data['SALES'] = data['SALES'].astype(int)

```

22. \*\*`data['SALES'] = data['SALES'].astype(int)`\*\*: Converts the `SALES` column to integer data type, simplifying numeric computations.

```python

data.describe()

```

23. \*\*`data.describe()`\*\*: Re-displays descriptive statistics after the previous transformations, showing updated information for columns now formatted as integers.

```python

X = data[['SALES', 'PRODUCTCODE']]

```

24. \*\*`X = data[['SALES', 'PRODUCTCODE']]`\*\*: Selects the `SALES` and `PRODUCTCODE` columns to create the feature set `X`, which will be used for clustering.

```python

data.columns

```

25. \*\*`data.columns`\*\*: Lists all columns in the DataFrame.

```python

from sklearn.cluster import KMeans

kmeans = KMeans(n\_clusters=4, init='k-means++', random\_state=0).fit(X)

```

26. \*\*`from sklearn.cluster import KMeans`\*\*: Imports the `KMeans` clustering algorithm from `sklearn.cluster`.

27. \*\*`kmeans = KMeans(n\_clusters=4, init='k-means++', random\_state=0).fit(X)`\*\*: Initializes and fits a KMeans model on `X` with four clusters, using the `k-means++` method for cluster center initialization. `random\_state=0` ensures reproducibility of results.

Here's a detailed, line-by-line explanation of the remaining \*\*input\*\* part of this code:

```python

kmeans.labels\_

```

1. \*\*`kmeans.labels\_`\*\*: Returns the cluster labels for each data point in the dataset after the KMeans model has been fitted. Each label is an integer representing a specific cluster, where the cluster numbers (e.g., `0`, `1`, `2`, `3`) correspond to the cluster that each data point was assigned to.

```python

array([2, 2, 2, ..., 3, 0, 2])

```

2. \*\*`array([2, 2, 2, ..., 3, 0, 2])`\*\*: An example output of `kmeans.labels\_`, showing a sample array of labels where each element corresponds to the cluster assignment of a data point in the dataset. For instance, the first three data points belong to cluster `2`, and so on.

```python

kmeans.inertia\_

```

3. \*\*`kmeans.inertia\_`\*\*: The `inertia\_` attribute in KMeans returns the sum of squared distances between each data point and its nearest cluster center. It is a measure of how tightly grouped the clusters are; a lower inertia generally indicates more compact clusters.

```python

1043164092.8545694

```

4. \*\*`1043164092.8545694`\*\*: An example output of `kmeans.inertia\_`, showing the calculated inertia value for the current clustering configuration.

```python

kmeans.n\_iter\_

```

5. \*\*`kmeans.n\_iter\_`\*\*: This returns the number of iterations KMeans performed before it converged. KMeans iterates until it either meets a maximum number of iterations or achieves convergence, where the cluster centers no longer change significantly.

```python

15

```

6. \*\*`15`\*\*: An example output for `kmeans.n\_iter\_`, indicating that the KMeans algorithm took 15 iterations to converge.

```python

kmeans.cluster\_centers\_

```

7. \*\*`kmeans.cluster\_centers\_`\*\*: This attribute gives the coordinates of the cluster centers in the feature space. Each element in this array corresponds to the center of one cluster in terms of the selected features (`SALES` and `PRODUCTCODE` in this case).

```python

array([[1913.93425926, 63.19907407],

[8023.78238342, 28.35751295],

[3489.45517241, 55.50640394],

[5371.72523364, 40.62616822]])

```

8. \*\*`array([...])`\*\*: An example output of `kmeans.cluster\_centers\_`, where each sub-array represents the x and y coordinates (here, `SALES` and `PRODUCTCODE`) of each cluster’s center in the dataset.

```python

# getting the size of the clusters

from collections import Counter

Counter(kmeans.labels\_)

```

9. \*\*`from collections import Counter`\*\*: Imports the `Counter` class from Python’s `collections` module, which provides a way to count occurrences of elements in an iterable (like `kmeans.labels\_`).

10. \*\*`Counter(kmeans.labels\_)`\*\*: Creates a counter object that counts the occurrences of each unique cluster label in `kmeans.labels\_`. This gives the number of data points that belong to each cluster.

```python

Counter({0: 1078, 2: 1015, 3: 537, 1: 193})

```

11. \*\*`Counter({...})`\*\*: An example output showing the number of data points in each cluster. In this example, cluster `0` has `1078` points, cluster `2` has `1015` points, etc.

```python

Hence the Number of Clusters to be chosen will be 4 according to the elbow method

```

12. \*\*`Hence the Number of Clusters to be chosen will be 4 according to the elbow method`\*\*: This is a comment explaining that based on the elbow method analysis, 4 is a suitable number of clusters, which aligns with the clustering results obtained.

```python

sns.scatterplot(data=X, x="SALES", y="PRODUCTCODE", hue=kmeans.labels\_)

```

13. \*\*`sns.scatterplot(...)`\*\*: Uses Seaborn’s `scatterplot` to visualize the clustered data points. The `data=X` argument specifies that `X` is the data source, with `SALES` plotted on the x-axis and `PRODUCTCODE` on the y-axis. The `hue=kmeans.labels\_` parameter colors the points based on their assigned cluster, allowing the clusters to be visually distinguished.

```python

plt.scatter(kmeans.cluster\_centers\_[:,0], kmeans.cluster\_centers\_[:,1], marker="X")

```

14. \*\*`plt.scatter(...)`\*\*: Adds the cluster centers to the existing scatter plot. `kmeans.cluster\_centers\_[:,0]` and `kmeans.cluster\_centers\_[:,1]` represent the x and y coordinates of each cluster center, respectively. The `marker="X"` parameter indicates that the cluster centers will be marked with an "X" on the plot.

```python

plt.legend()

```

15. \*\*`plt.legend()`\*\*: Adds a legend to the plot, which provides a key for understanding the color coding of clusters and marks.

```python

plt.show()

```

16. \*\*`plt.show()`\*\*: Displays the scatter plot, showing the data points grouped by clusters and the centers of each cluster marked by an "X."