Python script that combines all the sections of the K-Means clustering implementation, including comments to explain each part of the code. The explanation is provided at the end.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import load_iris

# Step 1: Load Iris dataset
iris = load_iris()
X = iris.data  # Features: sepal length, sepal width, petal length, petal w
y = iris.target  # Actual labels (for comparison)

# Step 2: Preprocess the Data (Standardization)
# Since K-Means is sensitive to the scale of the data, we scale it to have
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)  # Scaling the features

# Step 3: Define Euclidean Distance Function
# This function computes the Euclidean distance between two points (vectors
def euclidean_distance(x1, x2):
    return np.sqrt(np.sum((x1 - x2) ** 2))

# Step 4: Implement K-Means Algorithm
# This function performs the K-Means clustering.
def kmeans(X, K, max_iters=100):
    # Step 4a: Initialize centroids randomly by selecting K points from the
    np.random.seed(42)  # For reproducibility
    centroids = X[np.random.choice(X.shape[0], K, replace=False)]

    # Step 4b: Iterative process to assign points to clusters and update ce
    for i in range(max_iters):
        # Step 4b1: Assign each point to the nearest centroid based on Eucl
        labels = np.array([np.argmin([euclidean_distance(x, centroid) for c

        # Step 4b2: Recalculate centroids as the mean of the points in each
        new_centroids = np.array([X[labels == j].mean(axis=0) for j in rang

        # Step 4b3: Check for convergence (if centroids do not change)
        if np.all(centroids == new_centroids):
            print(f"Converged after {i+1} iterations.")
            break
        centroids = new_centroids

    return centroids, labels

# Step 5: Apply K-Means to the scaled Iris data
K = 3  # We know there are 3 classes in the Iris dataset
centroids, labels = kmeans(X_scaled, K)

# Output the final centroids and the first few cluster labels
print("Final Centroids:\n", centroids)
print("Labels for the first few data points:", labels[:10])

# Step 6: Visualize the Results
# Plotting the first two features (sepal length vs sepal width) for simplic
plt.figure(figsize=(8, 6))
plt.scatter(X_scaled[:, 0], X_scaled[:, 1], c=labels, cmap='viridis', marke

# Plot the centroids on top
plt.scatter(centroids[:, 0], centroids[:, 1], c='red', marker='x', s=200, l
plt.xlabel('Sepal Length (scaled)')
plt.ylabel('Sepal Width (scaled)')
plt.title('K-Means Clustering on Iris Dataset')
plt.legend()
plt.show()

# Step 7: Compare the Clustering Results with the Actual Labels
# Since K-Means is unsupervised, it does not know the actual class labels.
# We can compare the cluster labels with the true labels for an assessment.
print("True labels for the first few data points:", y[:10])
```

Explanation of the Code:

1. **Import Libraries**:
    - We import `pandas` for data manipulation, `numpy` for mathematical operations, `matplotlib` for visualization, and `StandardScaler` from `sklearn` for feature scaling.
    - The `load_iris()` function from `sklearn` loads the Iris dataset, which we use to test our implementation.

2. **Load and Preprocess the Data**:

   - We load the Iris dataset into `X` (features) and `y` (actual labels).
   - To ensure that all features contribute equally to the clustering process, we scale the data using `StandardScaler`. This standardizes the features to have a mean of 0 and a standard deviation of 1, making it easier for the K-Means algorithm to perform effectively.

3. **Define Euclidean Distance Function**:

   - The `euclidean_distance()` function calculates the distance between two points in space. We will use this to measure the proximity of each data point to the centroids during clustering.

4. **Implement the K-Means Algorithm**:

   - **Step 4a**: The centroids are initialized randomly by selecting `K` random data points from the dataset.
   - **Step 4b**: We enter the iterative process:
     - **Step 4b1**: Each data point is assigned to the nearest centroid based on the Euclidean distance.
     - **Step 4b2**: After assigning all points to clusters, we update each centroid by calculating the mean of the points assigned to that cluster.
     - **Step 4b3**: We check for convergence. If the centroids no longer change, the algorithm stops early.

5. **Apply K-Means to the Iris Dataset**:

   - We apply the `kmeans()` function to the scaled Iris data with `K=3` clusters (since the Iris dataset has 3 distinct species).
   - The final centroids and labels for each data point are printed out. The centroids represent the central point of each cluster, while the labels indicate the assigned cluster for each data point.

6. **Visualize the Clustering**:

   - Since we are using a 4-dimensional dataset (Iris dataset has 4 features), we visualize the first two features (sepal length vs sepal width) for simplicity.
   - We use `matplotlib` to create a scatter plot where each point is colored according to its assigned cluster. The centroids are marked with red 'X' markers.

7. **Compare Clustering Results with True Labels**:

    - Finally, we compare the K-Means clustering results (`labels`) with the true class labels (`y`). This comparison helps assess the performance of the algorithm. While K-Means doesn't know the true labels, you can visually inspect or measure how closely the clustering matches the actual species.

Key Notes:

- **Initialization**: The random initialization of centroids can sometimes lead to suboptimal clustering, especially if the initial centroids are chosen poorly. This can be mitigated using methods like **K-Means++**, but that's not implemented here.
- **Convergence**: The algorithm stops when the centroids do not change significantly after an iteration (convergence). If there is no convergence within the maximum number of iterations (`max_iters`), the algorithm will stop anyway.
- **Visualization**: Since we used the first two features for plotting, the clusters may look more distinct. In a higher-dimensional space, K-Means works in the full feature space, and visualizations can be more complex. Dimensionality reduction techniques like **PCA** or **t-SNE** are often used for such cases.

By running this code, you should see how the K-Means algorithm clusters the Iris dataset into 3 groups, and the visual representation should give you an idea of how well the algorithm performed.