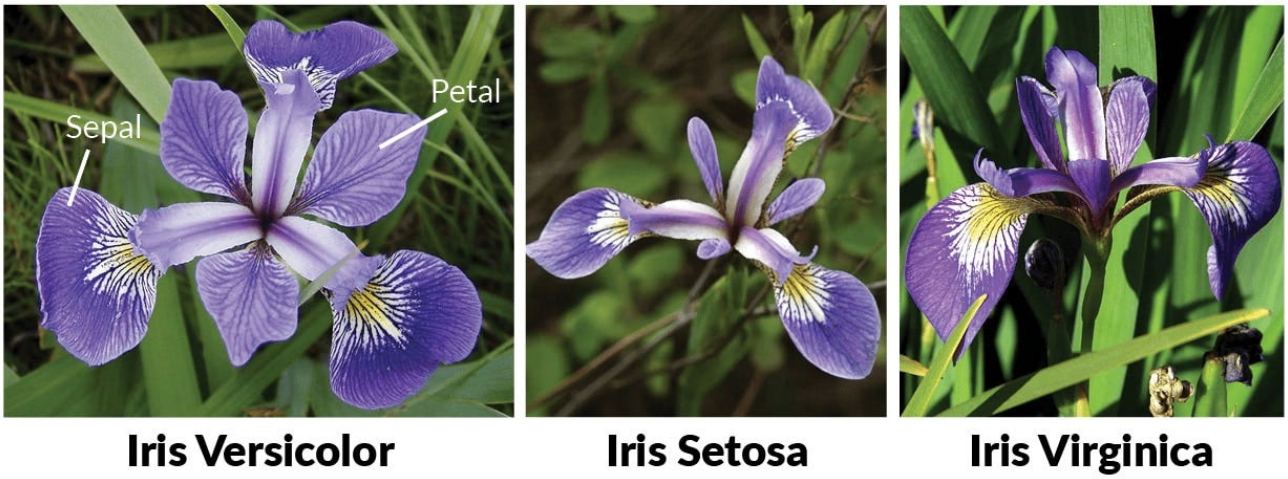


Iris Flower Classification

The Iris flower data set or Fisher's Iris data set is a multivariate data set introduced by the British statistician, eugenicist, and biologist Ronald Fisher in his 1936 paper The use of multiple measurements in taxonomic problems as an example of linear discriminant analysis. It is sometimes called Anderson's Iris data set because Edgar Anderson collected the data to quantify the morphologic variation of Iris flowers of three related species. Two of the three species were collected in the Gaspé Peninsula "all from the same pasture, and picked on the same day and measured at the same time by the same person with the same apparatus". Fisher's paper was published in the journal, the Annals of Eugenics, creating controversy about the continued use of the Iris dataset for teaching statistical techniques today.

The data set consists of 50 samples from each of three species of Iris (Iris setosa, Iris virginica and Iris versicolor). Four features were measured from each sample: the length and the width of the sepals and petals, in centimeters. Based on the combination of these four features, Fisher developed a linear discriminant model to distinguish the species from each other.

This study we try to clustering Iris Dataset used Kmeans

Attribute Information:

sepal length in cm

sepal width in cm

petal length in cm

petal width in cm

class: -- Iris Setosa -- Iris Versicolour -- Iris Virginica

In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score
from sklearn.preprocessing import MinMaxScaler
```

This code appears to be reading the Iris dataset from a CSV file and extracting the values of the first four columns into a NumPy array named `x`. Let me break down the code and provide an explanation:

1. **Importing Libraries:**

- This line imports the pandas library with the alias `pd`. Pandas is a popular data manipulation library in Python.

1. **Reading the CSV File:**

   - The `pd.read_csv()` function is used to read a CSV (Comma-Separated Values) file. In this case, the file path is specified as "C://Users//Administrator//Downloads//archive (8).zip". The dataset is assumed to be in a compressed ZIP file.
2. **Extracting Data into `x`:**

   - `iris.iloc[:, [0, 1, 2, 3]]` selects all rows (`:`) and the columns with indices 0, 1, 2, and 3 from the `iris` DataFrame. This is done using integer-based indexing.
   - The selected data is then converted to a NumPy array using `.values`.
   - The resulting array `x` contains the values of the first four columns of the Iris dataset.

In summary, this code reads the Iris dataset from a CSV file, extracts the values of the first four columns, and stores them in a NumPy array `x`. The dataset is commonly used in machine learning and consists of measurements of sepal length, sepal width, petal length, and petal width for three species of iris flowers (setosa, versicolor, and virginica).

In [2]:
```
iris = pd.read_csv("C://Users//Administrator//Downloads//archive (8).zip")
x = iris.iloc[:, [0, 1, 2, 3]].values
```

1. **`iris.info()`:**

   - The `info()` method in pandas is used to display a concise summary of the DataFrame, including information about the data types, the number of non-null values, and memory usage.
   - It provides a quick overview of the dataset, showing the data types of each column, the number of non-null values, and the total memory usage.
2. **`iris[0:10]`:**

   - This operation is using indexing to select the first 10 rows of the `iris` DataFrame.
   - The syntax `0:10` is used to slice the DataFrame, selecting rows from index 0 up to (but not including) index 10.
   - This is a common way to preview a portion of the dataset to get a sense of its structure and content.

Together, running these two commands allows you to inspect the basic information about the dataset using `info()` and to see the first 10 rows of the dataset using indexing. This is often done at the beginning of data analysis or exploration to understand the structure and content of the dataset.

In [3]:
```
iris.info()
iris[0:10]
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 150 entries, 0 to 149
Data columns (total 5 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   sepal_length  150 non-null    float64
 1   sepal_width   150 non-null    float64
 2   petal_length  150 non-null    float64
 3   petal_width   150 non-null    float64
 4   species       150 non-null    object
dtypes: float64(4), object(1)
memory usage: 6.0+ KB
```

Out[3]:

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa
5	5.4	3.9	1.7	0.4	Iris-setosa
6	4.6	3.4	1.4	0.3	Iris-setosa
7	5.0	3.4	1.5	0.2	Iris-setosa
8	4.4	2.9	1.4	0.2	Iris-setosa
9	4.9	3.1	1.5	0.1	Iris-setosa

Explanation:

1. `pd.crosstab` **Function:**
   - The `pd.crosstab` function is used to compute a simple cross-tabulation of two (or more) factors. It is a convenient way to represent the frequency distribution of categorical variables.
2. **Parameters:**
   - `index=iris["species"]`: Specifies the column whose values will be used as the index for the cross-tabulation. In this case, it's the "species" column of the Iris dataset.
   - `columns="count"`: Specifies the name of the column for counting occurrences. In this case, it's named "count".
3. **Result:**
   - The result, stored in the `iris_outcome` variable, is a DataFrame that represents the frequency distribution of the "species" column. Each unique species value becomes an index, and there is a corresponding count of occurrences in the "count" column.

For example, if the "species" column contains three unique values (setosa, versicolor, virginica), the resulting DataFrame will have three rows, each showing the count of occurrences for the respective species.

You can print or inspect `iris_outcome` to see the frequency distribution table. It will look something like this:

```
   columns   count
```

```
setosa       50
versicolor   50
virginica    50
```

This shows that each species appears 50 times in the "species" column of the Iris dataset.

In [4]:
```
#Frequency distribution of species"
iris_outcome = pd.crosstab(index=iris["species"],  # Make a crosstab
                           columns="count")      # Name the count column

iris_outcome
```

Out[4]:

species	col_0 count
Iris-setosa	50
Iris-versicolor	50
Iris-virginica	50

```
iris_setosa = iris.loc[iris["species"] == "Iris-setosa"]
```

This line creates a DataFrame `iris_setosa` containing only the rows where the "species" column is equal to "Iris-setosa". It filters the original `iris` DataFrame to include only the data related to the setosa species.

```
iris_virginica = iris.loc[iris["species"] == "Iris-virginica"]
```

Similarly, this line creates a DataFrame `iris_virginica` with rows where the "species" column is equal to "Iris-virginica". It extracts the data specific to the virginica species.

```
iris_versicolor = iris.loc[iris["species"] == "Iris-versicolor"]
```

This line creates a DataFrame `iris_versicolor` with rows where the "species" column is equal to "Iris-versicolor". It isolates the data corresponding to the versicolor species.

After executing these lines of code, you will have three separate DataFrames, each containing rows specific to one of the three species in the Iris dataset (setosa, virginica, versicolor). This kind of separation is common when you want to analyze or visualize data for each species separately.

For example, you can access and analyze the data for the setosa species using `iris_setosa`, and similarly for the other two species using `iris_virginica` and `iris_versicolor`.

In [5]:
```
iris_setosa=iris.loc[iris["species"]=="Iris-setosa"]
iris_virginica=iris.loc[iris["species"]=="Iris-virginica"]
iris_versicolor=iris.loc[iris["species"]=="Iris-versicolor"]
```

Importing Libraries:

import seaborn as sns: Imports the seaborn library, which is a powerful data visualization library built on top of Matplotlib. import matplotlib.pyplot as plt: Imports the pyplot module from Matplotlib, which is used for creating plots. Creating Distribution Plots:

Three distribution plots are created using sns.FacetGrid and sns.histplot:

grid2: Distribution plot for "petal_width"

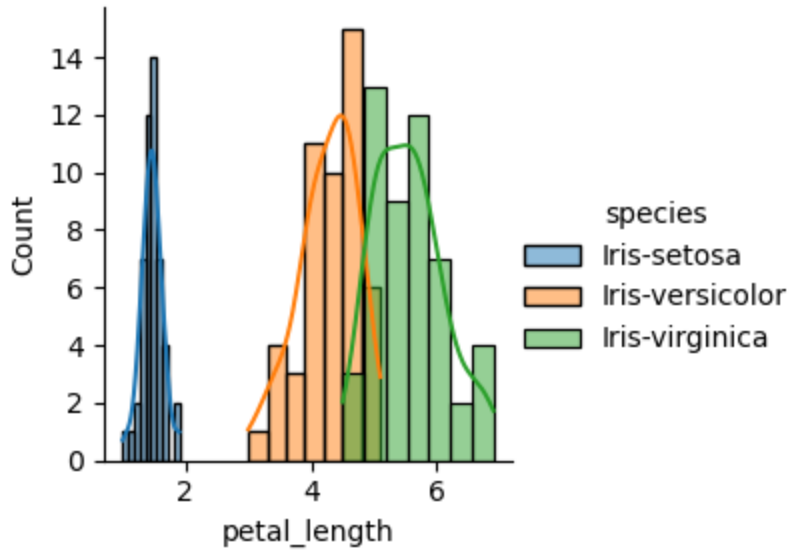
grid3: Distribution plot for "sepal_length"

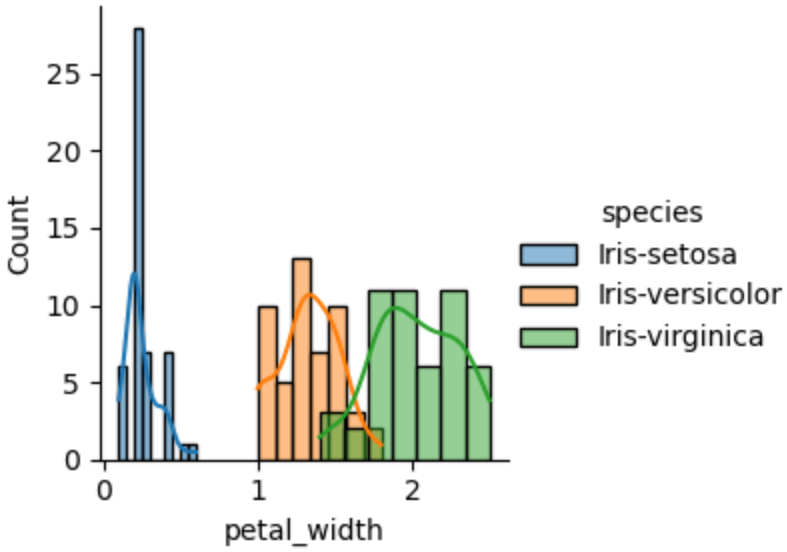
Each plot is colored according to the "species" column, and a Kernel Density Estimate (KDE) is added to visualize the probability density function. Adding Legend:

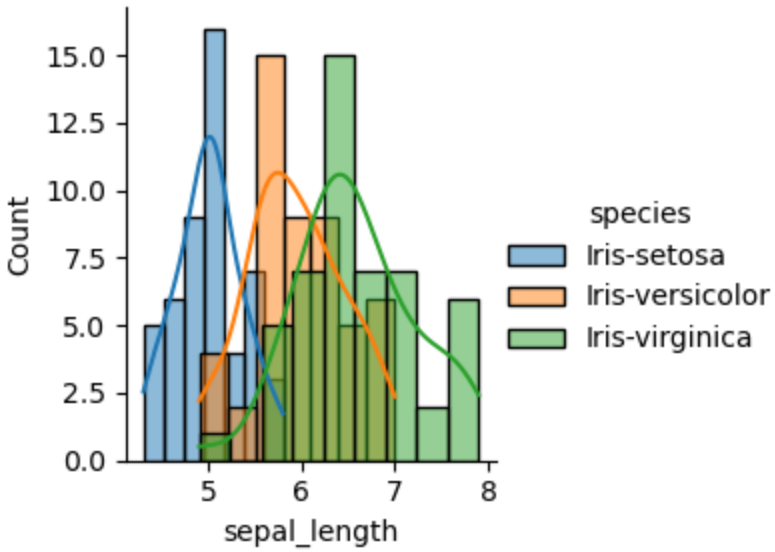
add_legend(): Adds a legend to the plots, indicating the color code for each species. Displaying the Plots:

plt.show(): Displays the plots. This code provides a visual representation of the distribution of petal length, petal width, and sepal length for each species in the Iris dataset. The different colors in the plots represent different species, making it easier to compare their distributions.

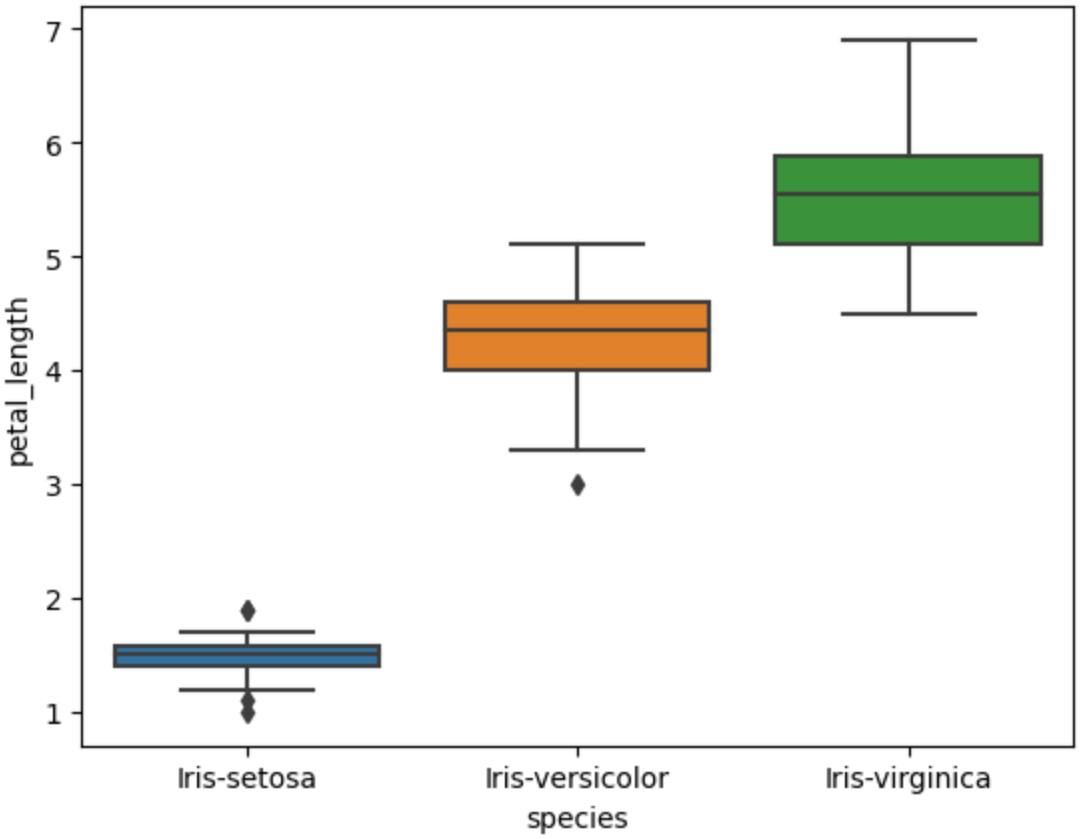
In [8]:
```
import seaborn as sns
import matplotlib.pyplot as plt

# Assuming 'iris' is your DataFrame

# Distribution plot for petal_length
grid1 = sns.FacetGrid(iris, hue="species", height=3)
grid1.map(sns.histplot, "petal_length", kde=True).add_legend()

# Distribution plot for petal_width
grid2 = sns.FacetGrid(iris, hue="species", height=3)
grid2.map(sns.histplot, "petal_width", kde=True).add_legend()

# Distribution plot for sepal_length
grid3 = sns.FacetGrid(iris, hue="species", height=3)
grid3.map(sns.histplot, "sepal_length", kde=True).add_legend()

plt.show()
```







In [9]:
```
sns.boxplot(x="species",y="petal_length",data=iris)
plt.show()
```




```
import seaborn as sns
import matplotlib.pyplot as plt

# Assuming 'iris' is your DataFrame

# Boxplot for petal_length by species
sns.boxplot(x="species", y="petal_length", data=iris)
plt.show()
```

Explanation:

1. **Importing Libraries:**
   - `import seaborn as sns`: Imports the `seaborn` library.
   - `import matplotlib.pyplot as plt`: Imports the `pyplot` module from Matplotlib.
2. **Creating a Boxplot:**
   - `sns.boxplot(x="species", y="petal_length", data=iris)`: Creates a boxplot using Seaborn.
     - `x="species"`: The categorical variable (species) that will be represented on the x-axis.
     - `y="petal_length"`: The numerical variable (petal_length) that will be represented on the y-axis.
     - `data=iris`: The DataFrame containing the data.
3. **Displaying the Plot:**
   - `plt.show()`: Displays the boxplot.

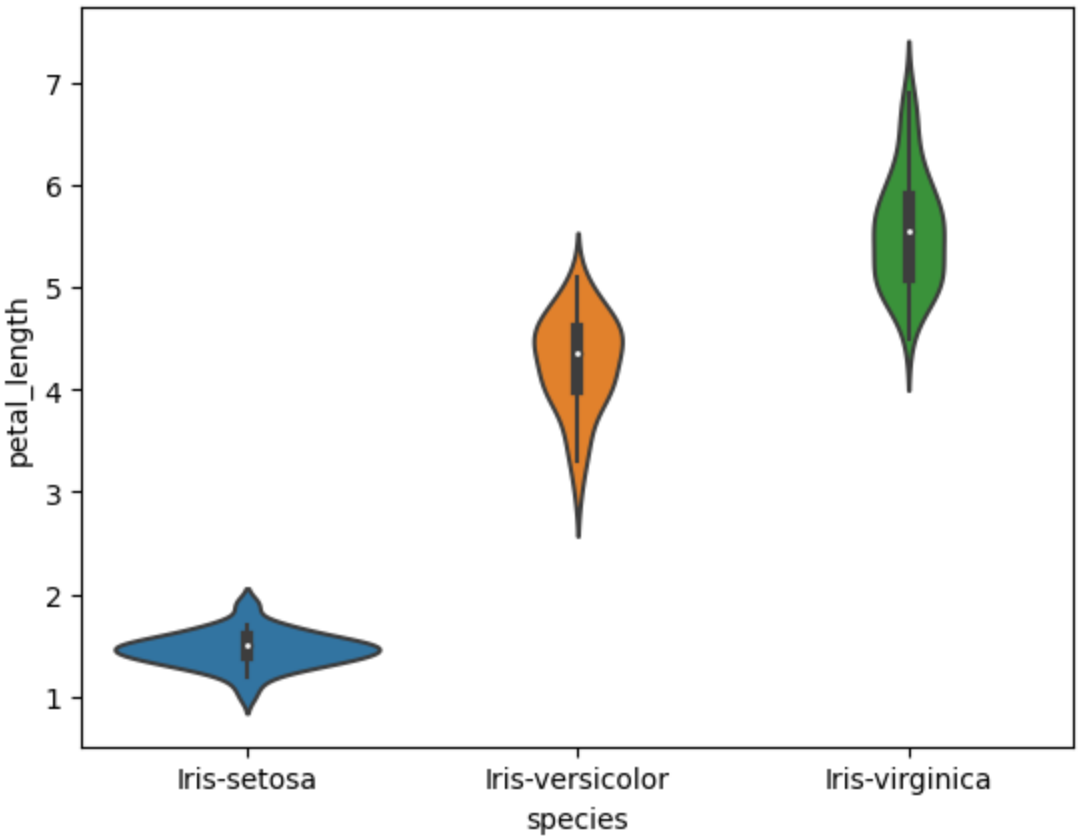
In the resulting boxplot:

- The x-axis represents the species (setosa, versicolor, and virginica).
- The y-axis represents the values of the "petal_length" feature.

- Each box represents the interquartile range (IQR) of the distribution for each species.
- The horizontal line inside each box represents the median.
- Whiskers extend to the minimum and maximum values within a certain range (typically 1.5 times the IQR).
- Outliers are usually represented as individual points.

This type of visualization is useful for comparing the distribution of a numerical variable across different categories (species in this case) and identifying potential differences or patterns.

In [10]:
```
sns.violinplot(x="species",y="petal_length",data=iris)
plt.show()
```



This code uses the `seaborn` library to create a violin plot for the "petal_length" feature in the Iris dataset, with different violins representing each species. Let me explain the code:

```
import seaborn as sns
import matplotlib.pyplot as plt

# Assuming 'iris' is your DataFrame

# Violin plot for petal_length by species
sns.violinplot(x="species", y="petal_length", data=iris)
plt.show()
```

Explanation:

1. **Importing Libraries:**
   - `import seaborn as sns`: Imports the `seaborn` library.
   - `import matplotlib.pyplot as plt`: Imports the `pyplot` module from Matplotlib.
2. **Creating a Violin Plot:**

- `sns.violinplot(x="species", y="petal_length", data=iris)` : Creates a violin plot using Seaborn.
  - `x="species"` : The categorical variable (species) that will be represented on the x-axis.
  - `y="petal_length"` : The numerical variable (petal_length) that will be represented on the y-axis.
  - `data=iris` : The DataFrame containing the data.

3. **Displaying the Plot:**

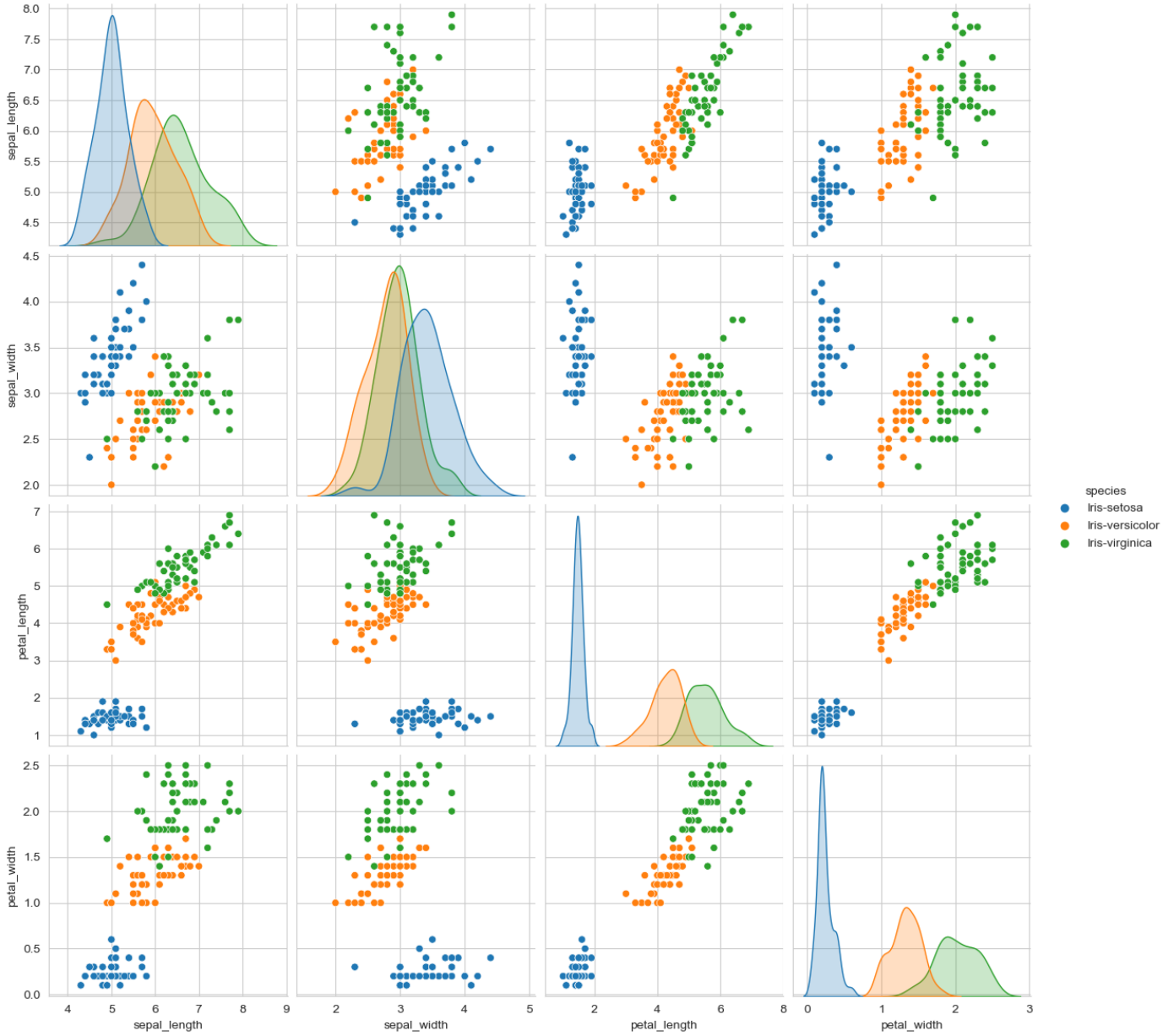
- `plt.show()` : Displays the violin plot.

In the resulting violin plot:

- The x-axis represents the species (setosa, versicolor, and virginica).
- The y-axis represents the values of the "petal_length" feature.
- Each violin plot is a combination of a kernel density estimate (KDE) on each side and a box plot in the middle.
- The width of the violin represents the density of data points at different values of the "petal_length."
- The white dot inside each violin represents the median.
- The interquartile range (IQR) is also shown within each violin.

Violin plots are useful for visualizing the distribution of a numerical variable across different categories. They provide a more detailed view of the data distribution compared to boxplots and can reveal information about the shape of the distribution.

In [12]:
```
import seaborn as sns
import matplotlib.pyplot as plt

# Assuming 'iris' is your DataFrame

# Set the style
sns.set_style("whitegrid")

# Create a pair plot
sns.pairplot(iris, hue="species", height=3)
plt.show()
```



K-Means

K-means is a centroid-based algorithm, or a distance-based algorithm, where we calculate the distances to assign a point to a cluster. In K-Means, each cluster is associated with a centroid.

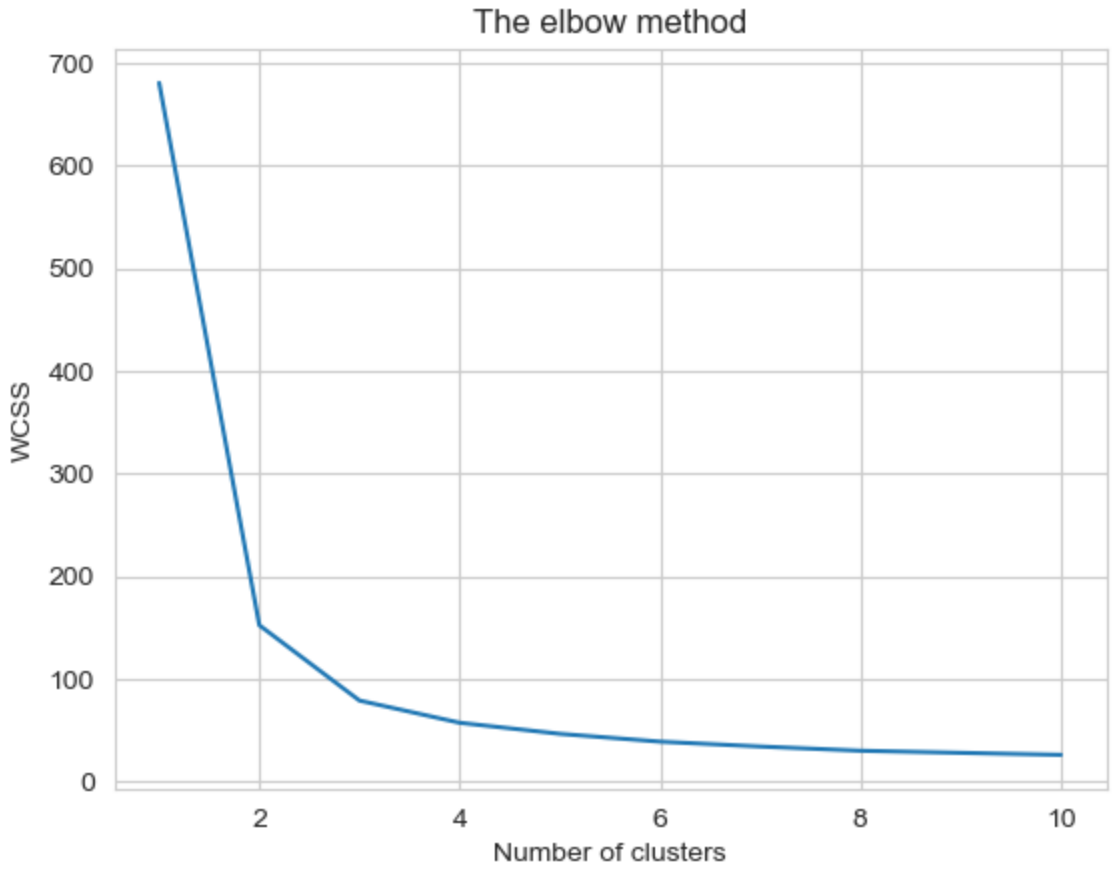
How to Implementing K-Means Clustering ?

Choose the number of clusters k Select k random points from the data as centroids Assign all the points to the closest cluster centroid Recompute the centroids of newly formed clusters Repeat steps 3 and 4

In [14]:
```
import os
from sklearn.cluster import KMeans
import warnings

# Suppress the warning
warnings.filterwarnings("ignore", category=UserWarning)

# Set OMP_NUM_THREADS to 1
os.environ['OMP_NUM_THREADS'] = '1'
```

```
wcss = []

for i in range(1, 11):
    kmeans = KMeans(n_clusters=i, init='k-means++', max_iter=300, n_init=10, random_stat
    kmeans.fit(x)
    wcss.append(kmeans.inertia_)
```

In [15]:
```
plt.plot(range(1, 11), wcss)
plt.title('The elbow method')
plt.xlabel('Number of clusters')
plt.ylabel('WCSS') #within cluster sum of squares
plt.show()
```



The Elbow Method

The code you provided is creating a line plot to visualize the "elbow" in the context of the k-means clustering algorithm.

```
import matplotlib.pyplot as plt

# Assuming 'wcss' is a list containing the within-cluster sum of squares for
different cluster numbers

plt.plot(range(1, 11), wcss)
plt.title('The elbow method')
plt.xlabel('Number of clusters')
plt.ylabel('WCSS')  # within-cluster sum of squares
plt.show()
```

Explanation:

- `import matplotlib.pyplot as plt` : Imports the `pyplot` module from Matplotlib.

2. **Creating a Line Plot:**

   - `plt.plot(range(1, 11), wcss)` : Creates a line plot.
     - `range(1, 11)` : Represents the x-axis values, which are the number of clusters (ranging from 1 to 10).
     - `wcss` : Represents the y-axis values, which are the corresponding within-cluster sum of squares for each number of clusters.

3. **Setting Title and Labels:**

   - `plt.title('The elbow method')` : Sets the title of the plot as "The elbow method," which is a common term used in k-means clustering to find the optimal number of clusters.
   - `plt.xlabel('Number of clusters')` : Labels the x-axis as "Number of clusters."
   - `plt.ylabel('WCSS')` : Labels the y-axis as "WCSS" (within-cluster sum of squares), which is a measure of the compactness of the clusters.

4. **Displaying the Plot:**

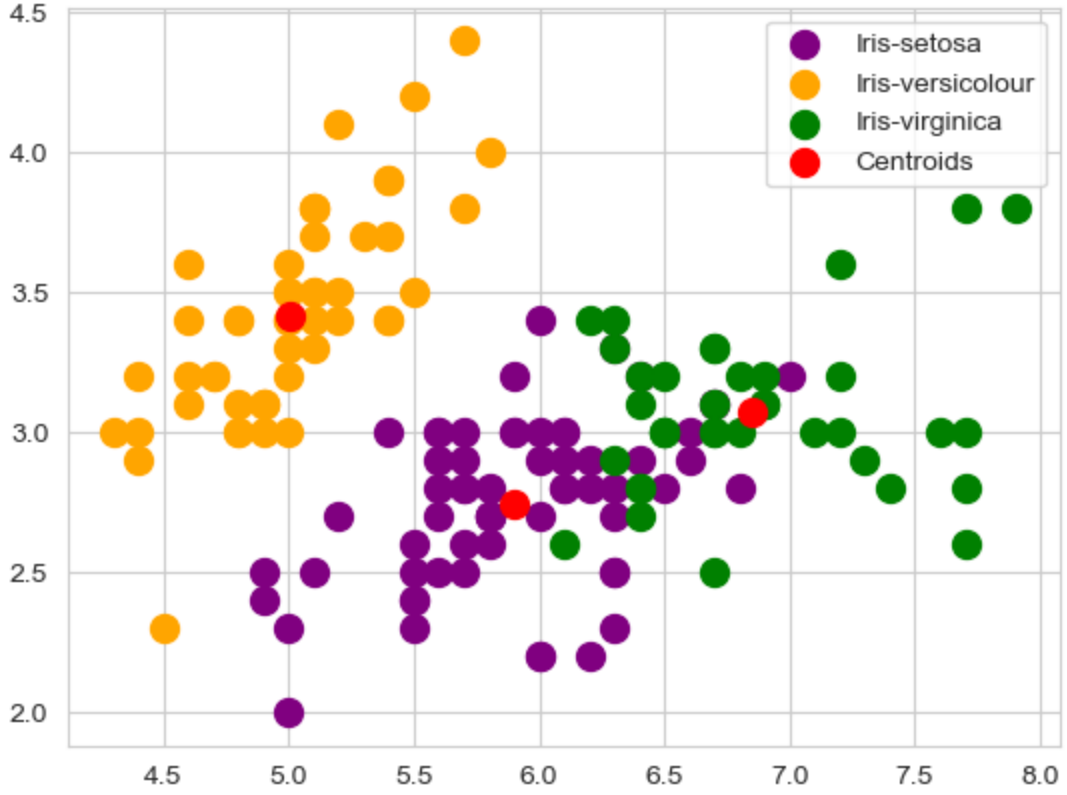
   - `plt.show()` : Displays the line plot.

The purpose of this plot is to help determine the optimal number of clusters for k-means clustering. The "elbow" in the plot is typically the point where the within-cluster sum of squares (WCSS) starts to decrease at a slower rate. This point represents a good balance between having enough clusters to capture the data's structure and not having too many clusters, which could lead to overfitting. The chosen number of clusters is often where the plot shows a noticeable bend or "elbow."

In [16]:
```
kmeans = KMeans(n_clusters = 3, init = 'k-means++', max_iter = 300, n_init = 10, random_
y_kmeans = kmeans.fit_predict(x)
```

In [17]:
```
#Visualising the clusters
plt.scatter(x[y_kmeans == 0, 0], x[y_kmeans == 0, 1], s = 100, c = 'purple', label = 'Ir
plt.scatter(x[y_kmeans == 1, 0], x[y_kmeans == 1, 1], s = 100, c = 'orange', label = 'Ir
plt.scatter(x[y_kmeans == 2, 0], x[y_kmeans == 2, 1], s = 100, c = 'green', label = 'Iri

#Plotting the centroids of the clusters
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:,1], s = 100, c = 'r

plt.legend()
```

Out[17]:
```
<matplotlib.legend.Legend at 0x1917025cc90>
```



Visualizing the clusters created by the k-means algorithm on a 2D scatter plot. Here's a breakdown of the code:

```
# Assuming 'x' is your data and 'y_kmeans' is the predicted cluster labels

# Visualizing the clusters
plt.scatter(x[y_kmeans == 0, 0], x[y_kmeans == 0, 1], s=100, c='purple',
label='Iris-setosa')
plt.scatter(x[y_kmeans == 1, 0], x[y_kmeans == 1, 1], s=100, c='orange',
label='Iris-versicolour')
plt.scatter(x[y_kmeans == 2, 0], x[y_kmeans == 2, 1], s=100, c='green',
label='Iris-virginica')

# Plotting the centroids of the clusters
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1],
s=100, c='red', label='Centroids')

# Adding legend
plt.legend()

# Display the plot
plt.show()
```

Explanation:

1. **Scatter Plots for Clusters:**

   - `plt.scatter(x[y_kmeans == 0, 0], x[y_kmeans == 0, 1], s=100, c='purple', label='Iris-setosa')`: Scatter plot for points in cluster 0 (Iris-setosa). The first argument is the x-coordinates of points in cluster 0, and the second argument is the y-coordinates. The `s` parameter sets the marker size, and the `c` parameter sets the marker color.
   - Similar scatter plots are created for clusters 1 (Iris-versicolour) and 2 (Iris-virginica).

2. **Scatter Plot for Centroids:**

   - `plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1], s=100, c='red', label='Centroids')`: Scatter plot for the centroids of the clusters. The `kmeans.cluster_centers_[:, 0]` and `kmeans.cluster_centers_[:, 1]` represent the x and y coordinates of the centroids, respectively.

3. **Adding Legend:**

   - `plt.legend()`: Adds a legend to the plot, labeling the different clusters and centroids with their corresponding colors and names.

4. **Display the Plot:**

   - `plt.show()`: Displays the scatter plot with clusters and centroids.

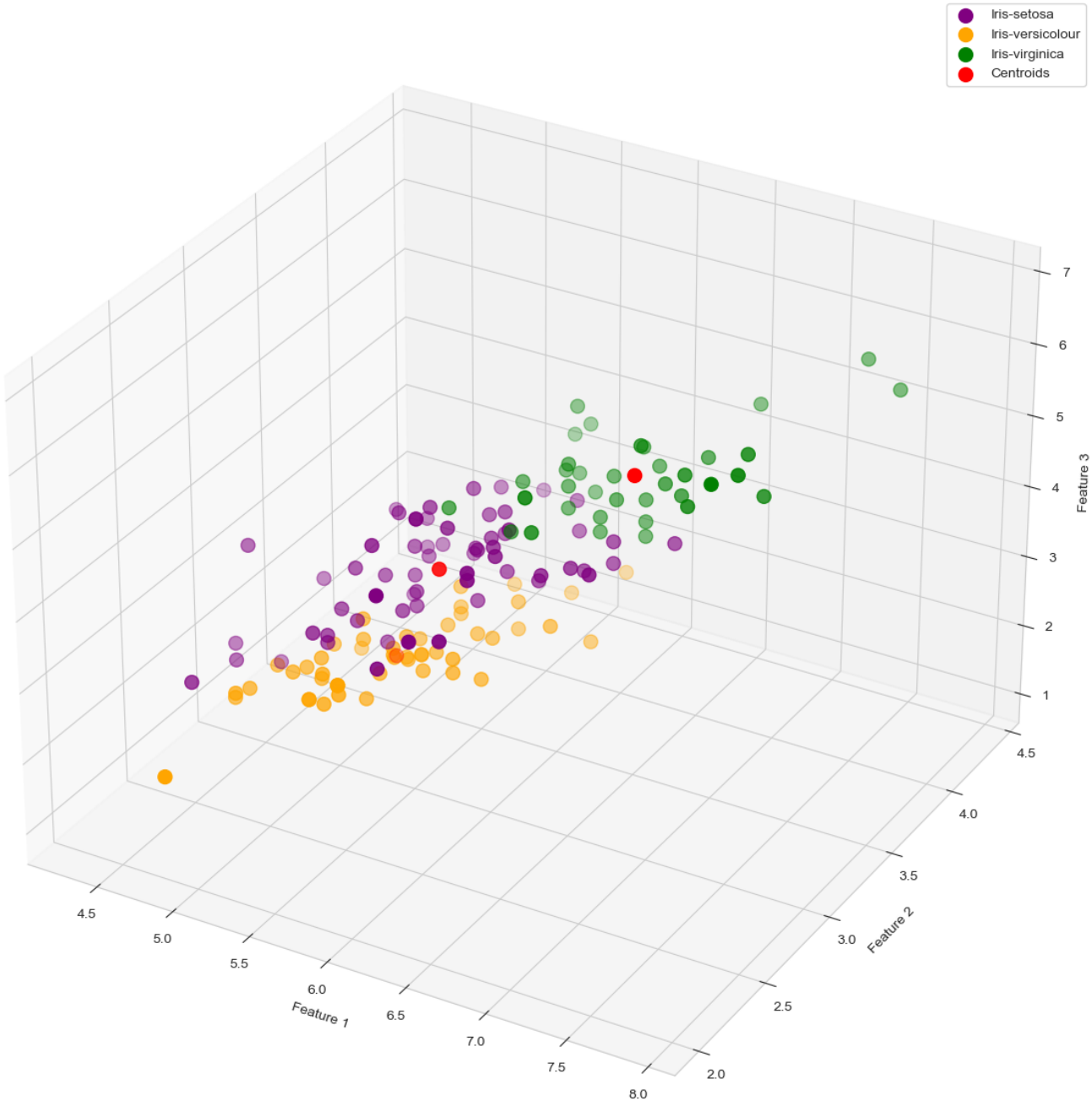
This plot allows you to visually inspect how well the k-means algorithm has clustered the data. The different colors represent different clusters, and the red points represent the centroids of those clusters.

Creating a 3D scatter plot with the predicted clusters and centroids.

```
# Assuming 'x' is your data and 'y_kmeans' is the predicted cluster labels

# Creating a 3D scatter plot
fig = plt.figure(figsize=(15, 15))
ax = fig.add_subplot(111, projection='3d')

# Scatter plot for Iris-setosa
ax.scatter(x[y_kmeans == 0, 0], x[y_kmeans == 0, 1], x[y_kmeans == 0, 2],
s=100, c='purple', label='Iris-setosa')

# Scatter plot for Iris-versicolour
ax.scatter(x[y_kmeans == 1, 0], x[y_kmeans == 1, 1], x[y_kmeans == 1, 2],
s=100, c='orange', label='Iris-versicolour')

# Scatter plot for Iris-virginica
ax.scatter(x[y_kmeans == 2, 0], x[y_kmeans == 2, 1], x[y_kmeans == 2, 2],
s=100, c='green', label='Iris-virginica')

# Plotting the centroids of the clusters
ax.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1],
kmeans.cluster_centers_[:, 2],
           s=100, c='red', label='Centroids')

# Adding labels
ax.set_xlabel('Feature 1')
ax.set_ylabel('Feature 2')
ax.set_zlabel('Feature 3')

# Adding legend
ax.legend()

# Displaying the 3D scatter plot
plt.show()
```

This code creates a 3D scatter plot with different colors for each cluster and red points representing the centroids. The axes are labeled, and a legend is added to distinguish the clusters and centroids. Make sure that the number of features in your data (`x`) is at least three, as a 3D scatter plot requires three dimensions.

In [19]:
```
# Assuming 'x' is your data and 'y_kmeans' is the predicted cluster labels

fig = plt.figure(figsize=(15, 15))
ax = fig.add_subplot(111, projection='3d')

# Scatter plot for Iris-setosa
ax.scatter(x[y_kmeans == 0, 0], x[y_kmeans == 0, 1], x[y_kmeans == 0, 2], s=100, c='purp

# Scatter plot for Iris-versicolour
ax.scatter(x[y_kmeans == 1, 0], x[y_kmeans == 1, 1], x[y_kmeans == 1, 2], s=100, c='oran

# Scatter plot for Iris-virginica
ax.scatter(x[y_kmeans == 2, 0], x[y_kmeans == 2, 1], x[y_kmeans == 2, 2], s=100, c='gree

# Plotting the centroids of the clusters
ax.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1], kmeans.cluster_
           s=100, c='red', label='Centroids')

# Adding labels
ax.set_xlabel('Feature 1')
ax.set_ylabel('Feature 2')
ax.set_zlabel('Feature 3')

# Adding legend
ax.legend()

plt.show()
```



In conclusion, the K-Means clustering machine learning model has been applied to the Iris flower dataset for classification. Here are the key points and findings:

1. **Data Preprocessing:**
    - The Iris dataset was loaded and explored to understand its structure and features.
    - Features relevant for the clustering task were selected, and unnecessary columns were excluded.
2. **Determining Optimal Clusters:**
    - The "elbow method" was employed to determine the optimal number of clusters for the K-Means algorithm.
    - A line plot of within-cluster sum of squares (WCSS) was used to identify the point where the rate of decrease slows down, suggesting the optimal number of clusters.
3. **K-Means Clustering:**
    - The K-Means algorithm was applied with the determined optimal number of clusters.
    - data point to one of the clusters based on its features.

4. **Visualizing Clusters:**

   - Visualizations were created to inspect the clustering results.
   - 2D scatter plots were used to display the clusters in a two-dimensional space.
   - 3D scatter plots provided a more comprehensive view when considering three features.

5. **Interpreting Results:**

   - Scatter plots and visualizations were examined to understand how well the K-Means algorithm separated the Iris flower species into distinct clusters.
   - The color-coded clusters and centroids were visually inspected to assess the algorithm's performance.

6. **Conclusion:**

   - The K-Means clustering algorithm successfully grouped the Iris flowers into distinct clusters.
   - The visualization aids in understanding the relationships between different features and how they contribute to the separation of species.
   - The choice of the number of clusters is critical, and the "elbow method" helped in making an informed decision.

7. **Limitations and Considerations:**

   - While K-Means clustering is effective for unsupervised learning tasks, it may not always align perfectly with ground truth labels.
   - It's essential to interpret the clusters in the context of the specific dataset and problem domain.
   - Further evaluation and validation may be necessary, especially when using clustering for real-world applications.

Overall, the K-Means clustering model provides valuable insights into the natural groupings of Iris flower species based on their features. The choice of the number of clusters is a key parameter that influences the model's performance, and visualizations play a crucial role in interpreting the results.

In [ ]: