
Code by Juan David Correa astropema.com
astropema@gmail.com Feb 2025

---

1. mean()

Library: statistics or numpy

Use: Calculates the arithmetic mean (average) of a dataset.

Code Example:##

In [31]:
```
import statistics

data = [10, 20, 30, 40, 50]
mean_value = statistics.mean(data)
print("Mean using statistics:", mean_value)

# Using NumPy
import numpy as np
mean_value_np = np.mean(data)
print("Mean using NumPy:", mean_value_np)
```

```
Mean using statistics: 30
Mean using NumPy: 30.0
```

median()

Library: statistics or numpy

Use: Calculates the median (middle value) of a dataset. If the dataset has an even number of elements, it returns the average of the two middle values.

Code Example:##

In [32]:
```
import statistics

data = [10, 20, 30, 40, 50]
median_value = statistics.median(data)
```

```
print("Median using statistics:", median_value)

# Using NumPy
import numpy as np
median_value_np = np.median(data)
print("Median using NumPy:", median_value_np)
```

```
Median using statistics: 30
Median using NumPy: 30.0
```

3. mode()

Library: statistics

Use: Finds the most common data point (mode) in a dataset. If the dataset has multiple modes, an exception may be raised in some libraries, while others handle it differently.

Code Example:

In [33]:
```
import statistics
from scipy import stats

# Dataset
data = [10, 20, 20, 30, 40, 50]

# Using statistics library
mode_value = statistics.mode(data)
print("Mode using statistics:", mode_value)

# Using SciPy with updated behavior
mode_result = stats.mode(data, keepdims=True)  # keepdims ensures compatibil
print("Mode using SciPy:", mode_result.mode[0], "with count:", mode_result.c
```

```
Mode using statistics: 20
Mode using SciPy: 20 with count: 2
```

4. stdev()

Library: statistics

Use: Calculates the standard deviation of a dataset, which measures the amount of variation or dispersion of the data.

Code Example:##

In [34]:
```
import statistics

data = [10, 20, 30, 40, 50]
stdev_value = statistics.stdev(data)
print("Standard Deviation using statistics:", stdev_value)
```

```
# Using NumPy
import numpy as np
stdev_value_np = np.std(data, ddof=1)  # ddof=1 for sample standard deviatic
print("Standard Deviation using NumPy:", stdev_value_np)
```

```
Standard Deviation using statistics: 15.811388300841896
Standard Deviation using NumPy: 15.811388300841896
```

5. variance()

Library: statistics or numpy

Use: Calculates the variance of a dataset, representing the average of the squared differences from the mean. It measures the spread of data.

Code Example:

In [35]:
```
import statistics

data = [10, 20, 30, 40, 50]
variance_value = statistics.variance(data)
print("Variance using statistics:", variance_value)

# Using NumPy
import numpy as np
variance_value_np = np.var(data, ddof=1)  # ddof=1 for sample variance
print("Variance using NumPy:", variance_value_np)
```

```
Variance using statistics: 250
Variance using NumPy: 250.0
```

6. skew()

Library: scipy.stats

Use: Calculates the skewness of a dataset, which measures the asymmetry of the data distribution around the mean. A positive value indicates a distribution with a longer tail on the right, while a negative value indicates a longer tail on the left.

Code Example:

In [36]:
```
from scipy.stats import skew

data = [10, 20, 20, 30, 40, 50]
skewness = skew(data)
print("Skewness of the data:", skewness)
```

```
Skewness of the data: 0.3053162697580519
```

7. kurtosis()

Library: scipy.stats

Use: Calculates the kurtosis of a dataset, which measures the "tailedness" of the distribution. Higher values indicate heavier tails, while lower values indicate lighter tails compared to a normal distribution.

Code Example:##

In [37]:
```
from scipy.stats import kurtosis

data = [10, 20, 20, 30, 40, 50]
kurt_value = kurtosis(data)
print("Kurtosis of the data:", kurt_value)
```

```
Kurtosis of the data: -1.151715976331361
```

8. percentile()

Library: numpy

Use: Calculates the percentile value for a given dataset. A percentile represents the value below which a given percentage of observations in a dataset fall.

Code Example:##

In [38]:
```
import numpy as np

data = [10, 20, 30, 40, 50]
percentile_25 = np.percentile(data, 25)  # 25th percentile (first quartile)
percentile_50 = np.percentile(data, 50)  # 50th percentile (median)
percentile_75 = np.percentile(data, 75)  # 75th percentile (third quartile)

print("25th Percentile:", percentile_25)
print("50th Percentile (Median):", percentile_50)
print("75th Percentile:", percentile_75)
```

```
25th Percentile: 20.0
50th Percentile (Median): 30.0
75th Percentile: 40.0
```

9. quantile()

Library: pandas

Use: Calculates the quantiles of a dataset. Quantiles divide data into equal-sized intervals. For instance, the 0.25 quantile is equivalent to the 25th percentile.

Code Example:##

In [39]:
```
import pandas as pd

data = [10, 20, 30, 40, 50]
series = pd.Series(data)

quantile_25 = series.quantile(0.25)  # 25th quantile
quantile_50 = series.quantile(0.50)  # 50th quantile (median)
quantile_75 = series.quantile(0.75)  # 75th quantile

print("25th Quantile:", quantile_25)
print("50th Quantile (Median):", quantile_50)
print("75th Quantile:", quantile_75)
```

```
25th Quantile: 20.0
50th Quantile (Median): 30.0
75th Quantile: 40.0
```

10. corrcoef()

Library: numpy

Use: Calculates the Pearson correlation coefficient, which measures the linear relationship between two variables. The result is a matrix where the diagonal represents the self-correlation (1.0), and the off-diagonal elements represent the correlation between variables.

Code Example:

In [40]:
```
import numpy as np

# Sample data for two variables
x = [10, 20, 30, 40, 50]
y = [15, 25, 35, 45, 55]

correlation_matrix = np.corrcoef(x, y)
print("Correlation Coefficient Matrix:\n", correlation_matrix)
print("Correlation between x and y:", correlation_matrix[0, 1])  # Extract s
```

```
Correlation Coefficient Matrix:
 [[1. 1.]
 [1. 1.]]
Correlation between x and y: 1.0
```

11. cov()

Library: numpy

Use: Calculates the covariance between two variables. Covariance measures the degree to which two variables change together. A positive covariance indicates that the variables increase together, while a negative covariance indicates an inverse relationship.

Code Example:

In [41]:
```
import numpy as np

# Sample data for two variables
x = [10, 20, 30, 40, 50]
y = [15, 25, 35, 45, 55]

covariance_matrix = np.cov(x, y)
print("Covariance Matrix:\n", covariance_matrix)
print("Covariance between x and y:", covariance_matrix[0, 1])  # Extract spe
```

```
Covariance Matrix:
 [[250. 250.]
 [250. 250.]]
Covariance between x and y: 250.0
```

12. zscore()

Library: scipy.stats

Use: Calculates the Z-score for each element in the dataset. The Z-score represents how many standard deviations an element is from the mean. This is useful for identifying outliers and understanding the relative position of data points.

Code Example:

In [42]:
```
from scipy.stats import zscore

data = [10, 20, 30, 40, 50]
z_scores = zscore(data)

print("Z-scores of the data:", z_scores)
```

```
Z-scores of the data: [-1.41421356 -0.70710678  0.          0.70710678  1.41
421356]
```

"" 13. ttest_ind() Library: scipy.stats

Use: Performs an independent two-sample t-test. This test compares the means of two independent samples to determine if there is a significant difference between them.

Code Example:

Explanation:

T-statistic: Indicates the difference between the sample means relative to the variation in the samples. P-value: Helps determine the significance of the results. A small p-value (e.g., < 0.05) suggests significant differences.

In [43]:
```python
from scipy.stats import ttest_ind

# Sample data for two groups
group1 = [10, 20, 30, 40, 50]
group2 = [15, 25, 35, 45, 55]

t_stat, p_value = ttest_ind(group1, group2)
print("T-statistic:", t_stat)
print("P-value:", p_value)
```

```
T-statistic: -0.5
P-value: 0.6305360755569764
```

14. linregress()

Library: scipy.stats

Use: Performs linear regression analysis, fitting a line to a dataset and returning the slope, intercept, correlation coefficient, p-value, and standard error of the estimate.

Code Example:

Explanation:

Slope: The rate of change of y with respect to x. Intercept: The value of y when x is 0. R-squared: Measures the goodness of fit (1.0 means a perfect fit). P-value: Tests the null hypothesis that the slope is zero. Standard Error: Measures the accuracy of the slope estimate.

In [44]:
```python
from scipy.stats import linregress

# Sample data for two variables
x = [1, 2, 3, 4, 5]
y = [2, 4, 6, 8, 10]

slope, intercept, r_value, p_value, std_err = linregress(x, y)

print("Slope:", slope)
print("Intercept:", intercept)
print("R-squared:", r_value**2)
print("P-value:", p_value)
print("Standard Error:", std_err)
```

```
Slope: 2.0
Intercept: 0.0
R-squared: 1.0
P-value: 1.2004217548761408e-30
Standard Error: 0.0
```

The linregress() function from scipy.stats can only handle two vectors at a time: one for the independent variable (x) and one for the dependent variable (y). If you want to analyze more than two variables, you need to use a different method, such as multiple linear regression, which is supported by libraries like statsmodels or sklearn.

Example with Multiple Linear Regression Here's how you can perform regression analysis with more than two vectors using statsmodels:

In [45]:
```
!pip install statsmodels
```

```
Requirement already satisfied: statsmodels in /Users/obaozai/miniconda3/env
s/pygame39/lib/python3.9/site-packages (0.14.4)
Requirement already satisfied: numpy<3,>=1.22.3 in /Users/obaozai/miniconda
3/envs/pygame39/lib/python3.9/site-packages (from statsmodels) (2.0.2)
Requirement already satisfied: scipy!=1.9.2,>=1.8 in /Users/obaozai/minicond
a3/envs/pygame39/lib/python3.9/site-packages (from statsmodels) (1.13.1)
Requirement already satisfied: pandas!=2.1.0,>=1.4 in /Users/obaozai/minicon
da3/envs/pygame39/lib/python3.9/site-packages (from statsmodels) (2.2.3)
Requirement already satisfied: patsy>=0.5.6 in /Users/obaozai/miniconda3/env
s/pygame39/lib/python3.9/site-packages (from statsmodels) (1.0.1)
Requirement already satisfied: packaging>=21.3 in /Users/obaozai/miniconda3/
envs/pygame39/lib/python3.9/site-packages (from statsmodels) (24.2)
Requirement already satisfied: python-dateutil>=2.8.2 in /Users/obaozai/mini
conda3/envs/pygame39/lib/python3.9/site-packages (from pandas!=2.1.0,>=1.4->
statsmodels) (2.9.0.post0)
Requirement already satisfied: pytz>=2020.1 in /Users/obaozai/miniconda3/env
s/pygame39/lib/python3.9/site-packages (from pandas!=2.1.0,>=1.4->statsmodel
s) (2025.1)
Requirement already satisfied: tzdata>=2022.7 in /Users/obaozai/miniconda3/e
nvs/pygame39/lib/python3.9/site-packages (from pandas!=2.1.0,>=1.4->statsmod
els) (2025.1)
Requirement already satisfied: six>=1.5 in /Users/obaozai/miniconda3/envs/py
game39/lib/python3.9/site-packages (from python-dateutil>=2.8.2->pandas!=2.
1.0,>=1.4->statsmodels) (1.17.0)
```

In [46]:
```
import statsmodels.api as sm
print("Statsmodels successfully imported!")
```

```
Statsmodels successfully imported!
```

In [47]:
```
import numpy as np
import statsmodels.api as sm

# Sample data with 10 vectors (independent variables) and 1 dependent variab
np.random.seed(42)
X = np.random.rand(100, 10)  # 10 independent variables, 100 samples
y = np.random.rand(100)      # Dependent variable
```

```
# Adding a constant for the intercept
X = sm.add_constant(X)

# Fit the model
model = sm.OLS(y, X).fit()

# Summary of the model
print(model.summary())
```

OLS Regression Results

Dep. Variable:	y	R-squared:	0.075
Model:	OLS	Adj. R-squared:	-0.029
Method:	Least Squares	F-statistic:	0.7210
Date:	Sun, 09 Mar 2025	Prob (F-statistic):	0.703
Time:	12:35:23	Log-Likelihood:	-19.757
No. Observations:	100	AIC:	61.51
Df Residuals:	89	BIC:	90.17
Df Model:	10		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	0.5100	0.176	2.900	0.005	0.161	0.859
x1	0.0407	0.107	0.381	0.704	-0.171	0.253
x2	-0.2242	0.114	-1.969	0.052	-0.450	0.002
x3	0.1253	0.113	1.106	0.272	-0.100	0.350
x4	-0.0132	0.124	-0.106	0.916	-0.260	0.234
x5	0.0659	0.110	0.596	0.553	-0.154	0.285
x6	0.0529	0.109	0.487	0.628	-0.163	0.269
x7	0.0637	0.113	0.562	0.575	-0.161	0.289
x8	-0.0395	0.113	-0.349	0.728	-0.265	0.185
x9	0.0796	0.108	0.738	0.462	-0.135	0.294
x10	-0.0596	0.113	-0.525	0.601	-0.285	0.166

Omnibus:	20.889	Durbin-Watson:	2.017
Prob(Omnibus):	0.000	Jarque-Bera (JB):	4.987
Skew:	0.008	Prob(JB):	0.0826
Kurtosis:	1.906	Cond. No.	1

```
2.5
==============================================================================
==

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is corre
ctly specified.
```

In [48]:
```
X
```

Out[48]:
```
array([[1.        , 0.37454012, 0.95071431, ..., 0.86617615, 0.60111501,
        0.70807258],
       [1.        , 0.02058449, 0.96990985, ..., 0.52475643, 0.43194502,
        0.29122914],
       [1.        , 0.61185289, 0.13949386, ..., 0.51423444, 0.59241457,
        0.04645041],
       ...,
       [1.        , 0.66367117, 0.93682974, ..., 0.05142581, 0.49636625,
        0.59684285],
       [1.        , 0.33424389, 0.7709122 , ..., 0.43482734, 0.24640203,
        0.81910232],
       [1.        , 0.79941588, 0.69469647, ..., 0.13681863, 0.95023735,
        0.44600577]])
```

In [49]:
```
y
```

Out[49]:
```
array([0.18513293, 0.54190095, 0.87294584, 0.73222489, 0.80656115,
       0.65878337, 0.69227656, 0.84919565, 0.24966801, 0.48942496,
       0.22120944, 0.98766801, 0.94405934, 0.03942681, 0.70557517,
       0.92524832, 0.18057535, 0.56794523, 0.9154883 , 0.03394598,
       0.69742027, 0.29734901, 0.9243962 , 0.97105825, 0.94426649,
       0.47421422, 0.86204265, 0.8445494 , 0.31910047, 0.82891547,
       0.03700763, 0.59626988, 0.23000884, 0.12056689, 0.0769532 ,
       0.69628878, 0.33987496, 0.72476677, 0.06535634, 0.31529034,
       0.53949129, 0.79072316, 0.3187525 , 0.62589138, 0.88597775,
       0.61586319, 0.23295947, 0.02440078, 0.87009887, 0.02126941,
       0.87470167, 0.52893713, 0.9390677 , 0.79878324, 0.99793411,
       0.35071182, 0.76718829, 0.40193091, 0.47987562, 0.62750546,
       0.87367711, 0.98408347, 0.76827341, 0.41776678, 0.421357  ,
       0.7375823 , 0.23877715, 0.11047411, 0.35462216, 0.28723899,
       0.29630812, 0.23360775, 0.04209319, 0.01787393, 0.98772239,
       0.42777313, 0.38432665, 0.67964728, 0.21825389, 0.94996118,
       0.78634501, 0.089411  , 0.41758078, 0.87911831, 0.94473202,
       0.46740151, 0.61341139, 0.16703395, 0.99116863, 0.2316717 ,
       0.94273177, 0.64964665, 0.60773679, 0.51268851, 0.23066981,
       0.17652803, 0.22048621, 0.18643826, 0.77958447, 0.35012526])
```

15. describe()

Library: pandas

Use: Provides a summary of descriptive statistics for a dataset, including count, mean, standard deviation, minimum, maximum, and quartiles. It works on both numeric and non-numeric data, depending on the options used.

Code Example:

Explanation:

count: Number of observations. mean: Arithmetic mean. std: Standard deviation. min/max: Minimum and maximum values. 25%/50%/75%: Percentiles (quartiles).

In [50]:
```
import pandas as pd

# Sample dataset
data = {
    "Age": [25, 30, 35, 40, 45],
    "Salary": [40000, 50000, 60000, 70000, 80000]
}
df = pd.DataFrame(data)

# Generate descriptive statistics
summary = df.describe()
print("Descriptive Statistics:\n", summary)
```

```
Descriptive Statistics:
              Age        Salary
count   5.000000      5.000000
mean   35.000000  60000.000000
std     7.905694  15811.388301
min    25.000000  40000.000000
25%    30.000000  50000.000000
50%    35.000000  60000.000000
75%    40.000000  70000.000000
max    45.000000  80000.000000
```

16. probplot()

Library: scipy.stats

Use: Generates a probability plot to compare a dataset against a specified theoretical distribution (e.g., normal distribution). It is useful for visually assessing whether the data follows the expected distribution.

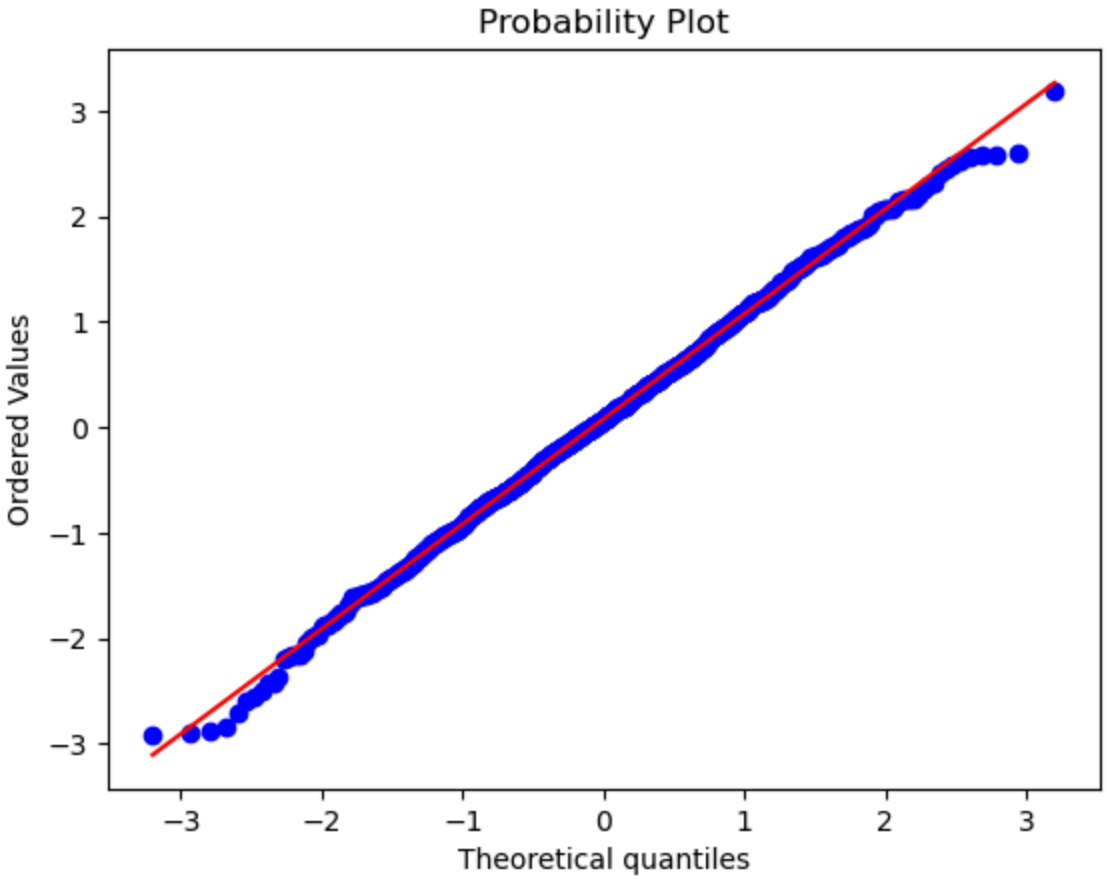
Code Example:

Output: A probability plot is displayed, showing the data points plotted against the theoretical quantiles. If the data follows the specified distribution, the points will lie close to a straight line.

In [51]:
```
import numpy as np
import scipy.stats as stats
import matplotlib.pyplot as plt
```

```python
# Generate sample data (normal distribution)
data = np.random.normal(loc=0, scale=1, size=1000)

# Generate the probability plot
stats.probplot(data, dist="norm", plot=plt)
plt.title("Probability Plot")
plt.show()
```



17. boxplot()

Library: matplotlib.pyplot or seaborn

Use: Creates a box plot (also called a whisker plot), which visualizes the distribution of a dataset and highlights its median, quartiles, and potential outliers.

Code Example: Using matplotlib:

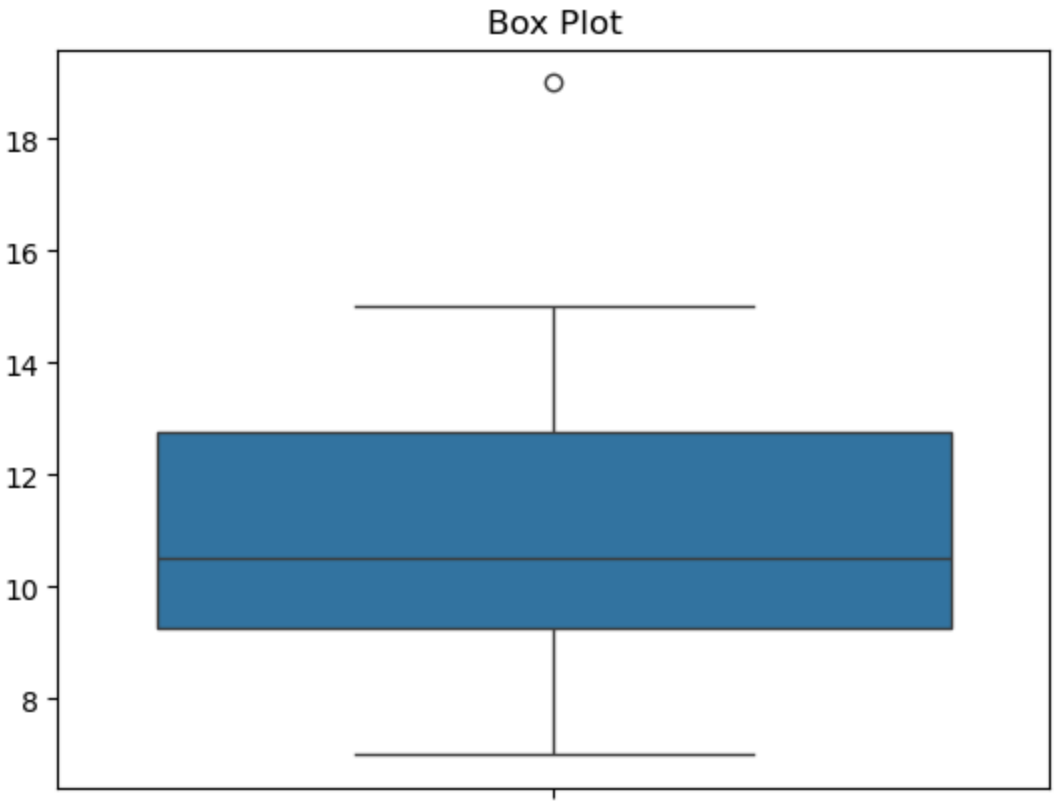
Output: A box plot is displayed with:

A box indicating the interquartile range (IQR: 25th to 75th percentile). A line within the box representing the median. Whiskers extending to 1.5 × IQR or the minimum/maximum values. Outliers plotted as individual points outside the whiskers.

In [52]:
```
import seaborn as sns

# Sample data
data = [7, 8, 8, 9, 10, 10, 10, 11, 11, 12, 13, 13, 15, 19]

# Generate a box plot
sns.boxplot(data=data)
plt.title("Box Plot")
plt.show()
```



18. histogram()

Library: matplotlib.pyplot

Use: Creates a histogram to visualize the distribution of a dataset by grouping data into bins and displaying the frequency of each bin.

Code Example:

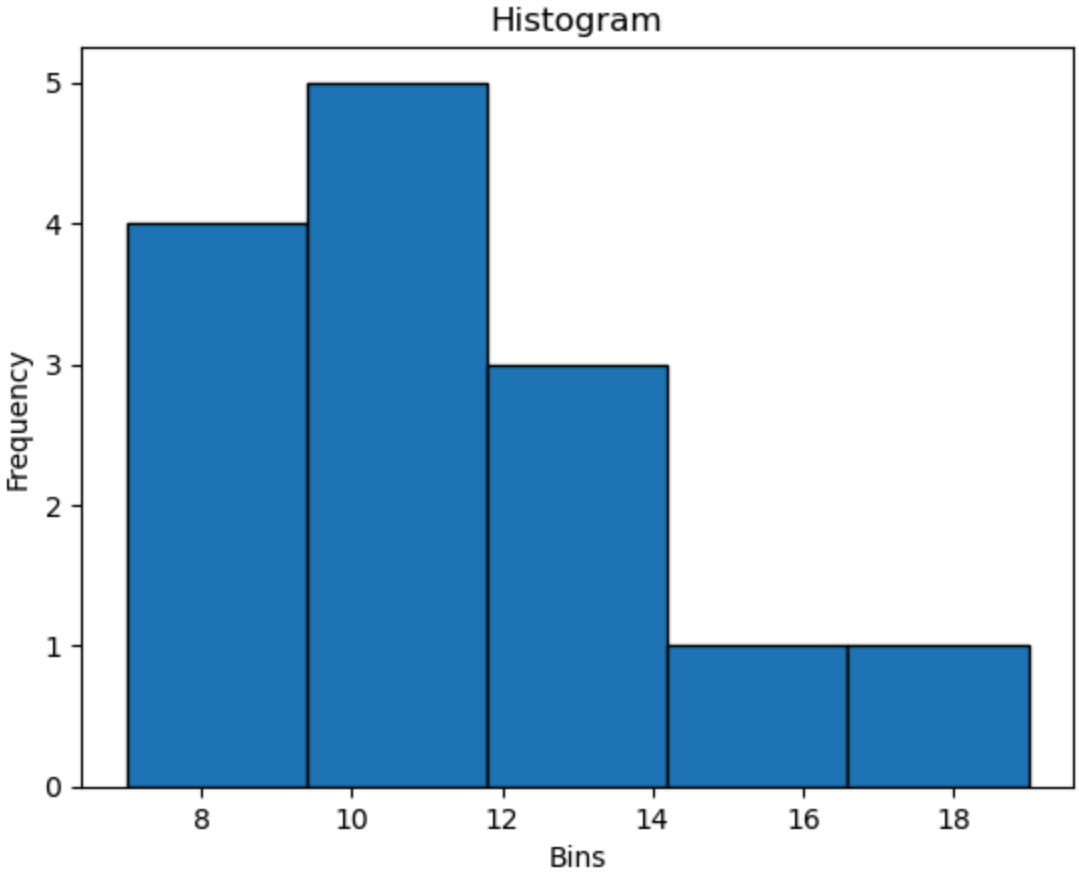
Output: A histogram is displayed with:

Bins on the x-axis representing ranges of values. Frequency on the y-axis representing the number of data points in each bin.

In [53]:
```
import matplotlib.pyplot as plt
```

```
# Sample data
data = [7, 8, 8, 9, 10, 10, 10, 11, 11, 12, 13, 13, 15, 19]

# Generate a histogram
plt.hist(data, bins=5, edgecolor="black")
plt.title("Histogram")
plt.xlabel("Bins")
plt.ylabel("Frequency")
plt.show()
```



19. kdeplot()

Library: seaborn

Use: Generates a Kernel Density Estimate (KDE) plot, which is a smoothed, continuous version of a histogram, often used to estimate the probability density function of a dataset.

Code Example:

Output: A KDE plot is displayed with a smooth curve representing the estimated probability density of the data.

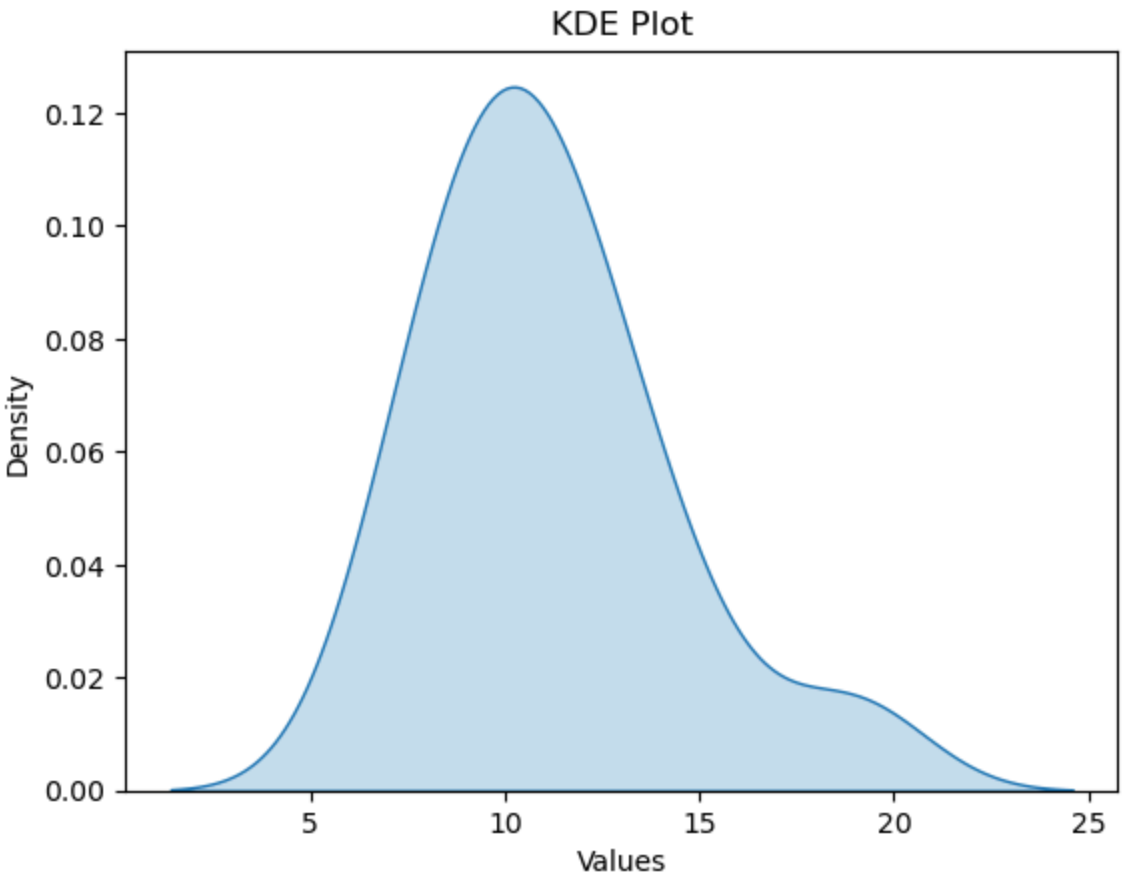
Key Features:

The fill=True option shades the area under the curve for better visualization. It provides an alternative to histograms, showing the distribution more smoothly.

In [54]:
```
import seaborn as sns
import matplotlib.pyplot as plt

# Sample data
data = [7, 8, 8, 9, 10, 10, 10, 11, 11, 12, 13, 13, 15, 19]

# Generate a KDE plot
sns.kdeplot(data=data, fill=True)
plt.title("KDE Plot")
plt.xlabel("Values")
plt.ylabel("Density")
plt.show()
```



20. heatmap()

Library: seaborn

Use: Generates a heatmap to visualize data in a matrix format with varying color intensities representing the values.

Code Example:

Output: A heatmap is displayed with:

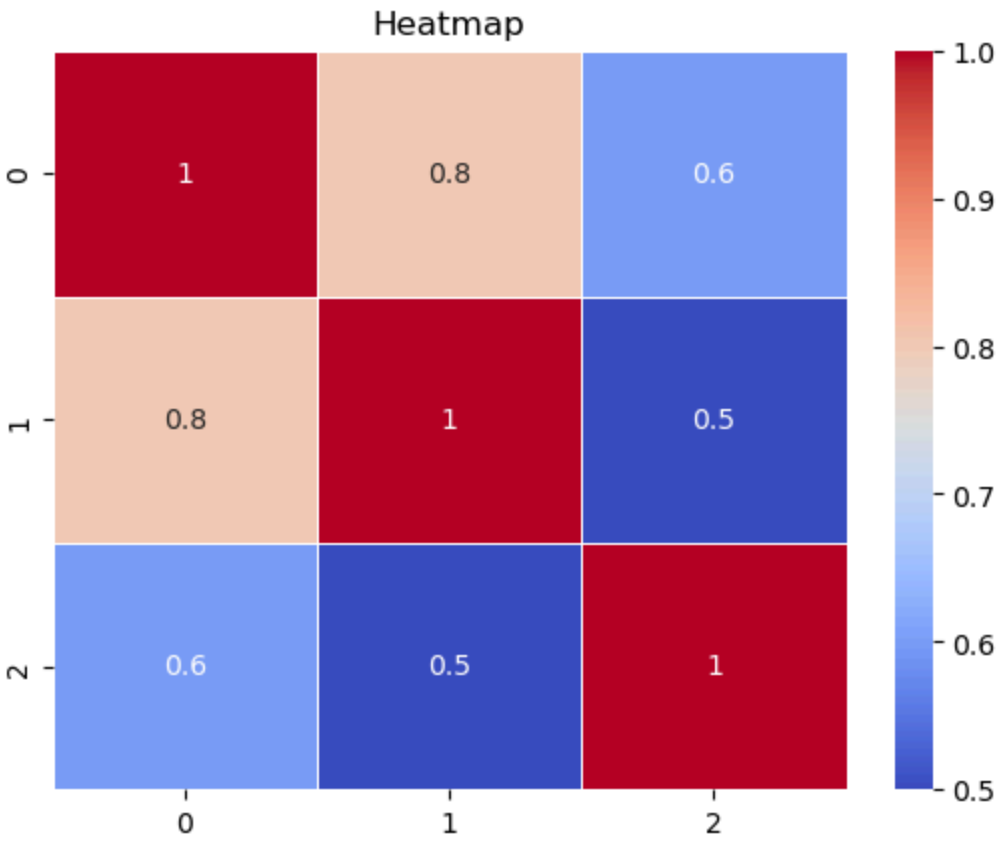
The matrix values annotated. A color scale representing the magnitude of values. Row and column relationships visually represented by color intensity. Key Features:

Useful for visualizing correlation matrices, confusion matrices, or any tabular data. Customizable color maps (e.g., coolwarm, viridis).

In [55]:
```
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np

# Sample data (correlation matrix)
data = np.array([[1.0, 0.8, 0.6],
    [0.8, 1.0, 0.5],
    [0.6, 0.5, 1.0]])

# Generate a heatmap
sns.heatmap(data, annot=True, cmap="coolwarm", linewidths=0.5)
plt.title("Heatmap")
plt.show()
```



21. crosstab()

Library: pandas

Use: Creates a cross-tabulation (contingency table) to compute the frequency distribution of two or more categorical variables. It's often used for summarizing relationships between variables.

Code Example:

Explanation:

Rows represent the categories of one variable (Gender). Columns represent the categories of another variable (Purchased). The values are the frequencies of the occurrences.

In [56]:
```
import pandas as pd

# Sample data
data = {
    "Gender": ["Male", "Female", "Female", "Male", "Male"],
    "Purchased": ["Yes", "No", "Yes", "No", "Yes"]
}

df = pd.DataFrame(data)

# Generate a cross-tabulation
crosstab_result = pd.crosstab(df["Gender"], df["Purchased"])
print(crosstab_result)
```

```
Purchased  No  Yes
Gender
Female      1    1
Male        1    2
```

22. groupby()

Library: pandas

Use: Aggregates data based on one or more columns, allowing computation of summary statistics (e.g., mean, sum, count) for each group.

Code Example:

Explanation:

Groups the dataset by the Category column. Computes the mean of the Values column for each group. Key Features:

Can apply multiple aggregation functions (e.g., sum, count, max). Flexible for summarizing grouped data.

In [57]:
```python
import pandas as pd

# Sample data
data = {
    "Category": ["A", "A", "B", "B", "C"],
    "Values": [10, 20, 30, 40, 50]
}
df = pd.DataFrame(data)

# Group by 'Category' and calculate the mean
grouped_data = df.groupby("Category")["Values"].mean()
print(grouped_data)
```

```
Category
A    15.0
B    35.0
C    50.0
Name: Values, dtype: float64
```

"23. pivot_table() Library: pandas

Use: Creates a pivot table for summarizing data, similar to Excel pivot tables. It allows grouping data and applying aggregation functions flexibly.

Code Example: Explanation:

Index (Region): Rows of the pivot table. Columns (Product): Columns of the pivot table. Values (Sales): Values aggregated (here using sum). Key Features:

Can handle multiple aggregation functions. Fills missing values with fill_value.

In [58]:
```python
import pandas as pd

# Sample data
data = {
    "Region": ["North", "North", "South", "South", "East"],
    "Product": ["A", "B", "A", "B", "A"],
    "Sales": [100, 150, 200, 250, 300]
}
df = pd.DataFrame(data)

# Create a pivot table
pivot = pd.pivot_table(
    df,
    values="Sales",
    index="Region",
    columns="Product",
    aggfunc="sum",
    fill_value=0
)
print(pivot)
```

```
Product    A    B
Region
East     300    0
North    100  150
South    200  250
```

24. rolling()

Library: pandas

Use: Computes rolling statistics, such as moving averages, sums, or other aggregations, over a specified window size in a dataset.

Code Example:

Explanation:

The first two rows have NaN because the window size is 3, and there aren't enough values to calculate the rolling mean. The rolling mean is computed for every window of 3 consecutive rows. Key Features:

Highly customizable with functions like sum, mean, min, max. Supports time-series analysis and smoothing of data.

In [59]:
```python
import pandas as pd

# Sample data
data = {
    "Date": pd.date_range(start="2023-01-01", periods=7, freq="D"),
    "Values": [10, 20, 30, 40, 50, 60, 70]
}
df = pd.DataFrame(data)

# Compute a rolling mean with a window size of 3
df["Rolling Mean"] = df["Values"].rolling(window=3).mean()
print(df)
```

```
        Date  Values  Rolling Mean
0 2023-01-01      10           NaN
1 2023-01-02      20           NaN
2 2023-01-03      30          20.0
3 2023-01-04      40          30.0
4 2023-01-05      50          40.0
5 2023-01-06      60          50.0
6 2023-01-07      70          60.0
```

25. expanding()

Library: pandas

Use: Calculates cumulative statistics over the entire dataset, starting from the first observation and expanding to include all previous rows.

Code Example:

Explanation:

The mean starts with the first value (10), then includes the next value (average of 10 and 20 = 15), and so on. Provides a cumulative view of the data. Key Features:

Can compute cumulative sums, minimums, maximums, etc. Useful for tracking statistics as more data becomes available.

In [60]:
```
import pandas as pd

# Sample data
data = {
    "Values": [10, 20, 30, 40, 50]
}
df = pd.DataFrame(data)

# Compute an expanding mean
df["Expanding Mean"] = df["Values"].expanding().mean()
print(df)
```

```
   Values  Expanding Mean
0      10            10.0
1      20            15.0
2      30            20.0
3      40            25.0
4      50            30.0
```

26. resample()

Library: pandas

Use: Aggregates time-series data into different time frequencies (e.g., daily to monthly, hourly to daily) and applies aggregation functions like sum, mean, etc.

Code Example:

Explanation:

Resamples the data into 3-day intervals (3D). Aggregates the values in each interval using the sum function. Key Features:

Supports custom frequencies (D for daily, M for monthly, H for hourly, etc.). Allows applying custom aggregation functions.

In [61]:
```python
import pandas as pd

# Sample time-series data
data = {
    "Date": pd.date_range(start="2023-01-01", periods=10, freq="D"),
    "Values": [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
}
df = pd.DataFrame(data)
df.set_index("Date", inplace=True)

# Resample to 3-day frequency and calculate the sum
resampled = df.resample("3D").sum()
print(resampled)
```

```
            Values
Date
2023-01-01      60
2023-01-04     150
2023-01-07     240
2023-01-10     100
```

27. interpolate()

Library: pandas

Use: Fills missing values in a dataset using interpolation methods, such as linear, polynomial, or time-based interpolation.

Code Example: Explanation:

Linear interpolation fills missing values by calculating intermediate values based on the surrounding data points. Other methods include polynomial, spline, and time. Key Features:

Useful for handling missing data in time-series or continuous datasets. Supports various interpolation techniques.

In [62]:
```python
import pandas as pd

# Sample data with missing values
data = {
    "Values": [10, None, 30, None, 50]
}
df = pd.DataFrame(data)

# Fill missing values using linear interpolation
df["Interpolated"] = df["Values"].interpolate(method="linear")
print(df)
```

```
   Values  Interpolated
0    10.0          10.0
1     NaN          20.0
2    30.0          30.0
3     NaN          40.0
4    50.0          50.0
```

28. rank()

Library: pandas

Use: Assigns ranks to values in a dataset, with options for handling ties (average, minimum, maximum, or first occurrence).

Code Example:

Explanation:

Scores of 90 get an average rank of 4.5 because of a tie. The rank is based on the ascending order of the values. Key Features:

Flexible tie-breaking methods: average, min, max, first, or dense. Useful for ranking in competitions, grading systems, or sorting data based on priority.

In [63]:
```python
import pandas as pd

# Sample data
data = {
    "Scores": [90, 70, 80, 90, 60]
}
df = pd.DataFrame(data)

# Rank the scores
df["Rank"] = df["Scores"].rank(method="average")  # Method can be 'average',
print(df)
```

```
   Scores  Rank
0      90   4.5
1      70   2.0
2      80   3.0
3      90   4.5
4      60   1.0
```

29. diff()

Library: pandas

Use: Computes the difference between consecutive elements in a column or row, often used for finding changes or deltas in time-series data.

Code Example:

Explanation:

The first value is NaN because there's no preceding value to calculate the difference. Each subsequent value is the difference from the previous row. Key Features:

Can specify the number of periods to calculate differences (e.g., diff(periods=2)). Useful for detecting trends, calculating changes, or measuring growth rates.

In [64]:
```
import pandas as pd

# Sample data
data = {
    "Values": [10, 20, 30, 50, 80]
}
df = pd.DataFrame(data)

# Compute the difference between consecutive elements
df["Difference"] = df["Values"].diff()
print(df)
```

```
   Values  Difference
0      10         NaN
1      20        10.0
2      30        10.0
3      50        20.0
4      80        30.0
```

30. pct_change()

Library: pandas

Use: Calculates the percentage change between consecutive elements in a column or row. It is commonly used for analyzing financial or time-series data to measure relative change.

Code Example:

Explanation:

The first value is NaN because there's no previous value to compare. Each subsequent value is the percentage change from the previous value: For example, (20 - 10) / 10 = 1.0 (or 100%). Key Features:

Can specify the number of periods for comparison (e.g., pct_change(periods=2)). Useful for identifying relative growth, trends, or changes over time.

In [65]:
```python
import pandas as pd

# Sample data
data = {
    "Values": [10, 20, 30, 50, 80]
}
df = pd.DataFrame(data)

# Compute the percentage change
df["Percent Change"] = df["Values"].pct_change()
print(df)
```

```
   Values  Percent Change
0      10             NaN
1      20        1.000000
2      30        0.500000
3      50        0.666667
4      80        0.600000
```

31. corr()

Library: pandas

Use: Calculates the correlation matrix for numerical columns in a DataFrame. Correlation measures the relationship between two variables (e.g., Pearson, Kendall, or Spearman methods).

Code Example:

Explanation:

1.0: Perfect positive correlation (e.g., A and B). -1.0: Perfect negative correlation (e.g., A and C, B and C). Correlation values range from -1 to 1. Key Features:

Supports different correlation methods (pearson, kendall, spearman). Useful for identifying relationships between variables.

In [66]:
```python
import pandas as pd

# Sample data
data = {
    "A": [1, 2, 3, 4, 5],
    "B": [10, 20, 30, 40, 50],
    "C": [5, 4, 3, 2, 1]
}
df = pd.DataFrame(data)

# Compute the correlation matrix
correlation_matrix = df.corr(method="pearson")  # Method can be 'pearson', '
print(correlation_matrix)
```

```
     A    B    C
A  1.0  1.0 -1.0
B  1.0  1.0 -1.0
C -1.0 -1.0  1.0
```

32. covariance()

Library: pandas

Use: Calculates the covariance matrix for numerical columns in a DataFrame. Covariance measures the degree to which two variables vary together.

Code Example:

Explanation:

Diagonal values: Variance of individual columns (e.g., 2.5 for column A). Off-diagonal values: Covariance between columns (e.g., 25.0 for A and B, -2.5 for A and C). Key Features:

Useful for understanding relationships between variables, but values are not standardized (unlike correlation). Shows how two variables change together in absolute terms.

In [67]:
```python
import pandas as pd

# Sample data
data = {
    "A": [1, 2, 3, 4, 5],
    "B": [10, 20, 30, 40, 50],
    "C": [5, 4, 3, 2, 1]
}
df = pd.DataFrame(data)

# Compute the covariance matrix
covariance_matrix = df.cov()
print(covariance_matrix)
```

```
      A      B     C
A   2.5   25.0  -2.5
B  25.0  250.0 -25.0
C  -2.5  -25.0   2.5
```

""\33. value_counts() Library: pandas

Use: Counts the occurrences of unique values in a column. It is particularly useful for analyzing categorical data.

Code Example:

Explanation:

The output shows the frequency of each unique value in the Category column: A appears 3 times. C appears 3 times. B appears 2 times. Key Features:

Can sort values by frequency or index. Optionally includes missing values with dropna=False.

In [68]:
```
import pandas as pd

# Sample data
data = {
    "Category": ["A", "B", "A", "C", "B", "A", "C", "C"]
}
df = pd.DataFrame(data)

# Count unique values in the 'Category' column
value_counts = df["Category"].value_counts()
print(value_counts)
```

```
Category
A    3
C    3
B    2
Name: count, dtype: int64
```

34. unique()

Library: pandas

Use: Returns an array of unique values in a column or Series. It is useful for identifying distinct categories or values in a dataset.

Code Example:

Explanation:

The output shows all distinct values in the Category column without duplicates. Key Features:

Returns an array of unique values. Works for both numerical and categorical data.

In [69]:
```
import pandas as pd

# Sample data
data = {
    "Category": ["A", "B", "A", "C", "B", "A", "C", "C"]
}
df = pd.DataFrame(data)

# Get unique values in the 'Category' column
```

```
unique_values = df["Category"].unique()
print(unique_values)
```

```
['A' 'B' 'C']
```

"35. isin() Library: pandas

Use: Checks whether elements in a DataFrame or Series are contained in a specified list of values, returning a boolean mask. This is useful for filtering data based on specific criteria.

Code Example:

Explanation:

The isin() function checks if values in the Category column are either A or C and returns a boolean mask. The mask is then used to filter the DataFrame. Key Features:

Simplifies filtering operations with predefined sets of criteria. Works with both numeric and categorical data.

In [70]:
```
import pandas as pd

# Sample data
data = {
    "Category": ["A", "B", "A", "C", "B", "A", "C", "C"],
    "Values": [10, 20, 30, 40, 50, 60, 70, 80]
}
df = pd.DataFrame(data)

# Check if 'Category' is in a specific list of values
mask = df["Category"].isin(["A", "C"])
filtered_data = df[mask]
print(filtered_data)
```

```
  Category  Values
0        A      10
2        A      30
3        C      40
5        A      60
6        C      70
7        C      80
```

36. sample()

Library: pandas

Use: Randomly selects rows or columns from a DataFrame or Series. Useful for sampling data for testing or analysis.

Code Example:

Explanation:

n=3 specifies the number of rows to sample. random_state=42 ensures reproducibility of the random selection. Key Features:

Allows sampling by rows (axis=0) or columns (axis=1). Supports sampling with replacement using replace=True.

In [71]:
```
import pandas as pd

# Sample data
data = {
    "Category": ["A", "B", "A", "C", "B", "A", "C", "C"],
    "Values": [10, 20, 30, 40, 50, 60, 70, 80]
}
df = pd.DataFrame(data)

# Randomly sample 3 rows
sampled_data = df.sample(n=3, random_state=42)
print(sampled_data)
```

```
  Category  Values
1        B      20
5        A      60
0        A      10
```

37. describe_stats() (Custom summary using scipy.stats)

Library: scipy.stats

Use: Provides a detailed summary of statistics for a dataset, including count, mean, variance, skewness, and kurtosis.

Code Example:

Explanation:

Count: Number of observations. Mean: Arithmetic average of the data. Variance: Measure of the spread. Min/Max: Smallest and largest values in the dataset. Skewness: Asymmetry of the distribution (0 for symmetric). Kurtosis: "Tailedness" of the distribution. Key Features:

A compact way to get a full statistical summary. Particularly useful for quick exploratory data analysis.

In [72]:
```
from scipy.stats import describe

# Sample data
```

```
data = [10, 20, 30, 40, 50, 60, 70, 80]

# Compute descriptive statistics
stats_summary = describe(data)
print("Descriptive Statistics:")
print("Count:", stats_summary.nobs)
print("Mean:", stats_summary.mean)
print("Variance:", stats_summary.variance)
print("Min:", stats_summary.minmax[0])
print("Max:", stats_summary.minmax[1])
print("Skewness:", stats_summary.skewness)
print("Kurtosis:", stats_summary.kurtosis)
```

```
Descriptive Statistics:
Count: 8
Mean: 45.0
Variance: 600.0
Min: 10
Max: 80
Skewness: 0.0
Kurtosis: -1.2380952380952381
```

38. percentileofscore()

Library: scipy.stats

Use: Calculates the percentile rank of a score relative to the given dataset. It answers the question, "What percentage of data points are less than or equal to this score?"

Code Example: Explanation:

50 is greater than or equal to 62.5% of the data points. The function can use different kind arguments (e.g., rank, weak, strict) to specify the method of calculation. Key Features:

Useful for evaluating a score's relative standing in a dataset. Supports both discrete and continuous data.

In [73]:
```
from scipy.stats import percentileofscore

# Sample data
data = [10, 20, 30, 40, 50, 60, 70, 80]

# Calculate the percentile rank of the score 50
percentile_rank = percentileofscore(data, 50)
print("Percentile Rank of 50:", percentile_rank)
```

```
Percentile Rank of 50: 62.5
```

39. sem()

Library: scipy.stats

Use: Calculates the standard error of the mean (SEM) for a dataset, which quantifies the accuracy of the sample mean as an estimate of the population mean.

Code Example:

Explanation:

The SEM is calculated as the standard deviation divided by the square root of the sample size. Smaller SEM indicates more precise estimates of the population mean. Key Features:

Commonly used in inferential statistics and hypothesis testing. Assumes data is sampled from a normally distributed population.

In [74]:
```
from scipy.stats import sem

# Sample data
data = [10, 20, 30, 40, 50, 60, 70, 80]

# Calculate the standard error of the mean
standard_error = sem(data)
print("Standard Error of the Mean:", standard_error)
```

```
Standard Error of the Mean: 8.660254037844386
```

40. ttest_rel()

Library: scipy.stats

Use: Performs a paired (dependent) t-test to compare the means of two related samples. It is commonly used in before-and-after studies or matched-pair designs.

Code Example:

Explanation:

T-statistic: The test statistic value for the paired t-test. P-value: The probability of observing a test statistic as extreme as the one computed, under the null hypothesis. Key Features:

Useful when the samples are not independent (e.g., measurements taken on the same subjects). Helps assess whether the mean difference between paired samples is statistically significant.

In [75]:
```
from scipy.stats import ttest_rel

# Sample data (before and after treatment)
before = [10, 20, 30, 40, 50]
after = [15, 25, 35, 45, 55]

# Perform a paired t-test
t_stat, p_value = ttest_rel(before, after)
print("T-statistic:", t_stat)
print("P-value:", p_value)
```

```
T-statistic: -inf
P-value: 0.0
```

```
/Users/obaozai/miniconda3/envs/pygame39/lib/python3.9/site-packages/scipy/stats/_axis_nan_policy.py:531: RuntimeWarning: Precision loss occurred in moment calculation due to catastrophic cancellation. This occurs when the data are nearly identical. Results may be unreliable.
  res = hypotest_fun_out(*samples, **kwds)
```

41. chisquare()

Library: scipy.stats

Use: Performs a chi-square goodness-of-fit test to determine if observed frequencies differ significantly from expected frequencies.

Code Example:

Explanation:

Chi-square Statistic: Measures the discrepancy between observed and expected frequencies. P-value: Determines whether the observed distribution significantly deviates from the expected. Key Features:

Commonly used to test for uniform distribution or adherence to a theoretical model. Requires non-negative observed and expected values.

In [76]:
```
from scipy.stats import chisquare

# Observed and expected frequencies
observed = [10, 20, 30]
expected = [15, 15, 30]

# Perform the chi-square test
chi2_stat, p_value = chisquare(f_obs=observed, f_exp=expected)
print("Chi-square Statistic:", chi2_stat)
print("P-value:", p_value)
```

```
Chi-square Statistic: 3.3333333333333335
P-value: 0.1888756028375618
```

42. f_oneway()

Library: scipy.stats

Use: Performs a one-way ANOVA test to determine if there are statistically significant differences between the means of two or more independent groups.

Code Example:

Explanation:

F-statistic: The ratio of between-group variance to within-group variance. P-value: Indicates whether the means of the groups are significantly different. Key Features:

Assumes the samples are independent and normally distributed with equal variances. Tests the null hypothesis that all group means are equal.

In [77]:
```
from scipy.stats import f_oneway

# Sample data for three groups
group1 = [10, 20, 30]
group2 = [15, 25, 35]
group3 = [10, 20, 30]

# Perform one-way ANOVA
f_stat, p_value = f_oneway(group1, group2, group3)
print("F-statistic:", f_stat)
print("P-value:", p_value)
```

```
F-statistic: 0.24999999999999997
P-value: 0.7865270823850706
```

43. kruskal()

Library: scipy.stats

Use: Performs the Kruskal-Wallis H-test, a non-parametric alternative to one-way ANOVA. It tests whether the medians of two or more independent samples are significantly different.

Code Example:

Explanation:

H-statistic: Test statistic based on ranked data. P-value: Indicates whether the medians of the groups are significantly different. Key Features:

Does not assume normality or equal variances. Suitable for ordinal data or non-normally distributed datasets.

In [78]:
```
from scipy.stats import kruskal

# Sample data for three groups
group1 = [10, 20, 30]
group2 = [15, 25, 35]
group3 = [10, 20, 30]

# Perform the Kruskal-Wallis H-test
h_stat, p_value = kruskal(group1, group2, group3)
print("H-statistic:", h_stat)
print("P-value:", p_value)
```

```
H-statistic: 0.6153846153846132
P-value: 0.7351414805916854
```

44. mannwhitneyu()

Library: scipy.stats

Use: Performs the Mann-Whitney U test, a non-parametric test for comparing the distributions of two independent samples. It is an alternative to the independent t-test when data does not meet normality assumptions.

Code Example: Explanation:

U-statistic: Test statistic indicating the rank sum comparison between two groups. P-value: Determines if the two groups differ significantly. Key Features:

Does not assume normality of the data. Can handle ordinal or non-normally distributed datasets. The alternative parameter specifies a two-sided, less, or greater test.

In [79]:
```
from scipy.stats import mannwhitneyu

# Sample data for two independent groups
group1 = [10, 20, 30, 40]
group2 = [15, 25, 35, 45]

# Perform the Mann-Whitney U test
u_stat, p_value = mannwhitneyu(group1, group2, alternative="two-sided")
print("U-statistic:", u_stat)
print("P-value:", p_value)
```

```
U-statistic: 6.0
P-value: 0.6857142857142857
```

45. wilcoxon()

Library: scipy.stats

Use: Performs the Wilcoxon signed-rank test, a non-parametric test for comparing two related (paired) samples. It is an alternative to the paired t-test when data does not meet normality assumptions.

Code Example:

Explanation:

W-statistic: Sum of the ranks of the differences between paired samples. P-value: Indicates whether there is a significant difference between the paired samples. Key Features:

Suitable for paired or matched samples that are not normally distributed. Tests the null hypothesis that the median difference between paired observations is zero.

In [80]:
```
from scipy.stats import wilcoxon

# Sample paired data (before and after treatment)
before = [10, 20, 30, 40, 50]
after = [15, 25, 35, 45, 55]

# Perform the Wilcoxon signed-rank test
w_stat, p_value = wilcoxon(before, after)
print("W-statistic:", w_stat)
print("P-value:", p_value)
```

```
W-statistic: 0.0
P-value: 0.0625
```

46. zmap()

Library: scipy.stats

Use: Calculates the z-scores of a dataset, standardizing the values relative to the mean and standard deviation of the population. It is useful for identifying how far a data point is from the mean.

Code Example:

Explanation:

Z-scores represent the number of standard deviations each value is from the mean of the population. Positive values indicate points above the mean, and negative values indicate points below. Key Features:

Useful for normalization and outlier detection. Relies on population mean and standard deviation for calculation

In [81]:
```
from scipy.stats import zmap

# Sample data
data = [10, 20, 30, 40, 50]
population = [15, 25, 35, 45, 55]

# Calculate z-scores for the sample data relative to the population
z_scores = zmap(data, population)
print("Z-scores:", z_scores)
```

```
Z-scores: [-1.76776695 -1.06066017 -0.35355339  0.35355339  1.06066017]
```

47. shapiro()

Library: scipy.stats

Use: Performs the Shapiro-Wilk test to check whether a dataset follows a normal distribution.

Code Example:

Explanation:

Shapiro-Wilk Statistic: Indicates how closely the data matches a normal distribution. P-value: If greater than 0.05, the null hypothesis (that the data is normally distributed) cannot be rejected. Key Features:

Suitable for small to medium-sized datasets. Tests the null hypothesis that the data comes from a normal distribution.

In [82]:
```
from scipy.stats import shapiro

# Sample data
data = [10, 12, 14, 16, 18, 20, 22, 24, 26]

# Perform the Shapiro-Wilk test
stat, p_value = shapiro(data)
print("Shapiro-Wilk Statistic:", stat)
print("P-value:", p_value)
```

```
Shapiro-Wilk Statistic: 0.9722884258803877
P-value: 0.913560953190048
```

48. anderson()

Library: scipy.stats

Use: Performs the Anderson-Darling test to evaluate whether a dataset follows a specified distribution (e.g., normal distribution). It provides critical values for comparison.

Code Example:

Explanation:

Statistic: Indicates how well the data matches the specified distribution. Critical Values: Pre-determined thresholds for significance levels. Compare the statistic to the critical values to determine normality: If the statistic is greater than a critical value, reject the null hypothesis at that significance level. Key Features:

More sensitive to differences in the tails of the distribution. Supports other distributions like exponential, gumbel, and logistic.

In [83]:
```
from scipy.stats import anderson

# Sample data
data = [10, 12, 14, 16, 18, 20, 22, 24, 26]

# Perform the Anderson-Darling test
result = anderson(data, dist='norm')  # 'norm' specifies a normal distributi
print("Anderson-Darling Statistic:", result.statistic)
print("Critical Values:", result.critical_values)
print("Significance Levels:", result.significance_level)
```

```
Anderson-Darling Statistic: 0.13676646631470213
Critical Values: [0.507 0.578 0.693 0.808 0.961]
Significance Levels: [15.  10.   5.   2.5  1. ]
```

49. kstest()

Library: scipy.stats

Use: Performs the Kolmogorov-Smirnov test to compare a dataset with a reference distribution or to test if two datasets are from the same distribution.

Code Example:

Explanation:

Statistic: The maximum difference between the empirical and reference distribution CDFs. P-value: If greater than 0.05, the null hypothesis (that the dataset matches the reference distribution) cannot be rejected. Key Features:

Non-parametric test, so it makes no assumptions about the data distribution. Useful for testing goodness-of-fit or comparing two datasets.

In [84]:
```
from scipy.stats import kstest

# Sample data
data = [10, 12, 14, 16, 18, 20, 22, 24, 26]

# Perform the Kolmogorov-Smirnov test
stat, p_value = kstest(data, 'norm', args=(20, 5))  # Test against a normal
print("Kolmogorov-Smirnov Statistic:", stat)
print("P-value:", p_value)
```

```
Kolmogorov-Smirnov Statistic: 0.23258904586104773
P-value: 0.634883574914318
```

50. levene()

Library: scipy.stats

Use: Performs Levene's test for equality of variances. It tests whether two or more groups have equal variances, making it robust to non-normal distributions.

Code Example:

Explanation:

Levene Statistic: Indicates the variance equality between groups. P-value: If greater than 0.05, the null hypothesis (that all groups have equal variances) cannot be rejected. Key Features:

Robust to deviations from normality. Can use different methods ('median', 'mean', 'trimmed') for testing, specified via the center parameter.

In [85]:
```
from scipy.stats import levene

# Sample data for three groups
group1 = [10, 12, 14, 16, 18]
group2 = [20, 22, 24, 26, 28]
group3 = [30, 32, 34, 36, 38]

# Perform Levene's test
stat, p_value = levene(group1, group2, group3)
print("Levene Statistic:", stat)
print("P-value:", p_value)
```

```
Levene Statistic: 0.0
P-value: 1.0
```

51. bartlett()

Library: scipy.stats

Use: Performs Bartlett's test for equality of variances. It tests whether the variances of two or more groups are equal, assuming that the data is normally distributed.

Code Example:

Explanation:

Bartlett Statistic: Measures the equality of variances. P-value: If greater than 0.05, the null hypothesis (that variances are equal) cannot be rejected. Key Features:

Suitable for normally distributed data. Less robust to deviations from normality compared to Levene's test.

In [86]:
```
from scipy.stats import bartlett

# Sample data for three groups
group1 = [10, 12, 14, 16, 18]
group2 = [20, 22, 24, 26, 28]
group3 = [30, 32, 34, 36, 38]

# Perform Bartlett's test
stat, p_value = bartlett(group1, group2, group3)
print("Bartlett Statistic:", stat)
print("P-value:", p_value)
```

```
Bartlett Statistic: 0.0
P-value: 1.0
```

52. spearmanr()

Library: scipy.stats

Use: Calculates the Spearman rank-order correlation coefficient and p-value to measure the monotonic relationship between two datasets. It is a non-parametric alternative to Pearson's correlation.

Code Example:

Explanation:

Spearman Correlation Coefficient: Ranges from -1 to 1, indicating the strength and direction of the monotonic relationship. 1: Perfect positive monotonic relationship. -1: Perfect negative monotonic relationship. 0: No monotonic relationship. P-value: Tests the null hypothesis that there is no correlation. Key Features:

Suitable for non-linear relationships and ordinal data. Assumes no specific distribution for the data.

In [87]:
```
from scipy.stats import spearmanr

# Sample data
x = [10, 20, 30, 40, 50]
y = [15, 25, 35, 45, 55]

# Compute Spearman's correlation
correlation, p_value = spearmanr(x, y)
print("Spearman Correlation Coefficient:", correlation)
print("P-value:", p_value)
```

```
Spearman Correlation Coefficient: 0.9999999999999999
P-value: 1.4042654220543672e-24
```

53. kendalltau()

Library: scipy.stats

Use: Calculates Kendall's Tau, a non-parametric measure of the strength and direction of association between two variables. It is based on the number of concordant and discordant pairs.

Code Example:

Explanation:

Kendall's Tau: Ranges from -1 to 1, where: 1: Perfect agreement (all pairs concordant). -1: Perfect disagreement (all pairs discordant). 0: No association. P-value: Tests the null hypothesis that there is no association between the variables. Key Features:

Robust to outliers and suitable for ordinal or non-normally distributed data. Works well for smaller datasets or those with many tied ranks.

In [88]:
```
from scipy.stats import kendalltau

# Sample data
x = [10, 20, 30, 40, 50]
y = [15, 25, 35, 45, 55]

# Compute Kendall's Tau
correlation, p_value = kendalltau(x, y)
print("Kendall's Tau:", correlation)
print("P-value:", p_value)
```

```
Kendall's Tau: 0.9999999999999999
P-value: 0.016666666666666666
```

54. skewtest()

Library: scipy.stats

Use: Tests the null hypothesis that a dataset has a skewness equal to zero (i.e., it is symmetric).

Code Example:

Explanation:

Skewness Test Statistic: Measures the asymmetry of the dataset. P-value: If greater than 0.05, the null hypothesis (that the data is symmetric) cannot be rejected. Key Features:

Assumes the dataset is drawn from a normal distribution. Useful for detecting asymmetry in datasets.

In [89]:
```
from scipy.stats import skewtest

# Larger sample data
data = [10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30]

# Perform the skewness test
try:
    stat, p_value = skewtest(data)
    print("Skewness Test Statistic:", stat)
    print("P-value:", p_value)
except ValueError as e:
    print("Error:", e)
```

```
Skewness Test Statistic: 1.0282686128790464
P-value: 0.30382349108673234
```

55. kurtosistest()

Library: scipy.stats

Use: Tests the null hypothesis that a dataset has a kurtosis equal to the kurtosis of a normal distribution (kurtosis = 3).

Important Note: Like skewtest(), this function requires at least 20 samples for reliable results. It will throw an error if the sample size is too small.

Explanation Kurtosis Test Statistic: Measures the "tailedness" of the data relative to a normal distribution. P-value: If greater than 0.05, the null hypothesis (that the data has normal kurtosis) cannot be rejected. Key Features:

Assumes the data comes from a normal distribution. Tests whether the data's kurtosis is consistent with a normal distribution.

In [90]:
```
from scipy.stats import kurtosistest

# Sample data
data = [10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42,

# Perform the kurtosis test
try:
    stat, p_value = kurtosistest(data)
    print("Kurtosis Test Statistic:", stat)
    print("P-value:", p_value)
except ValueError as e:
    print("Error:", e)
```

```
Kurtosis Test Statistic: -1.7058104152122062
P-value: 0.0880433833252835
```

56. friedmanchisquare()

Library: scipy.stats

Use: Performs the Friedman test, a non-parametric test for comparing the mean ranks of three or more related groups. It is often used for repeated measures.

Explanation Friedman Chi-Square Statistic: Tests whether the distributions of the groups are significantly different. P-value: If greater than 0.05, the null hypothesis (that the distributions are identical) cannot be rejected. Key Features:

Does not assume normality or equal variances. Useful for analyzing repeated measures or matched data.

In [91]:
```
from scipy.stats import friedmanchisquare

# Sample data: three related groups
group1 = [10, 20, 30]
group2 = [15, 25, 35]
group3 = [10, 15, 25]

# Perform the Friedman test
stat, p_value = friedmanchisquare(group1, group2, group3)
print("Friedman Chi-Square Statistic:", stat)
print("P-value:", p_value)
```

```
Friedman Chi-Square Statistic: 5.636363636363634
P-value: 0.05971441573218535
```

57. binom_test()

Library: scipy.stats

Use: Performs an exact test for the probability of success in a Bernoulli experiment. It tests whether the observed success proportion differs from the expected proportion.

Explanation P-value: If less than 0.05, the null hypothesis (that the observed success rate matches the expected success rate) is rejected. Parameters: successes: Number of observed successes. trials: Total number of trials. expected_prob: The hypothesized probability of success. alternative: Can be "two-sided", "less", or "greater".

In [92]:
```
from scipy.stats import binomtest

# Observed successes and total trials
successes = 15
trials = 20
expected_prob = 0.5  # Expected probability of success

# Perform the binomial test
result = binomtest(successes, trials, expected_prob, alternative="two-sided"
print("P-value:", result.pvalue)
```

```
P-value: 0.04138946533203125
```

from scipy.stats import binomtest

Observed successes and total trials

successes = 15 trials = 20 expected_prob = 0.5 # Expected probability of success

Perform the binomial test

result = binomtest(successes, trials, expected_prob, alternative="two-sided") print("P-value:", result.pvalue)

Explanation pmf: Computes the probability of a specific outcome given the number of trials and probabilities for each category. Outcome: Represents the number of occurrences in each category. Key Features:

Useful for modeling scenarios with multiple outcomes (e.g., rolling a die, voting distributions). Assumes the probabilities sum to 1 and that outcomes are non-negative integers.

In [93]:
```
from scipy.stats import multinomial

# Define parameters
n = 10  # Total number of trials
probs = [0.2, 0.5, 0.3]  # Probabilities for each category

# Probability of a specific outcome
outcome = [2, 5, 3]  # Counts for each category
probability = multinomial.pmf(outcome, n, probs)
print("Probability of outcome", outcome, ":", probability)
```

```
Probability of outcome [2, 5, 3] : 0.08504999999999999
```

59. poisson()

Library: scipy.stats

Use: Models the Poisson distribution, which represents the probability of a given number of events occurring in a fixed interval of time or space, given a constant mean rate.

Explanation pmf(k, mu): Probability mass function for exactly k events. cdf(k, mu): Cumulative distribution function for k or fewer events. Suitable for modeling rare events (e.g., the number of calls received at a call center in an hour). Key Features:

Requires the parameter mu (mean rate of occurrence). Non-negative integers for the number of events.

In [94]:
```
from scipy.stats import poisson

# Define the mean rate of events (lambda)
mu = 3  # Average number of events

# Probability of observing exactly 5 events
probability = poisson.pmf(5, mu)
print("P(5 events):", probability)

# Cumulative probability of observing 5 or fewer events
cumulative_probability = poisson.cdf(5, mu)
print("P(<=5 events):", cumulative_probability)
```

```
P(5 events): 0.10081881344492458
P(<=5 events): 0.9160820579686965
```

60. expon()

Library: scipy.stats

Use: Models the exponential distribution, which represents the time between events in a Poisson process. It’s widely used for modeling waiting times.

Explanation pdf(x, scale): Computes the probability density at a specific point. cdf(x, scale): Computes the cumulative probability up to a specific point. The scale parameter is the reciprocal of the rate (λ) in a Poisson process. Key Features:

Models the time between independent events occurring at a constant average rate. Used in reliability analysis, queuing theory, and survival analysis.

In [95]:
```
from scipy.stats import expon

# Define the scale parameter (mean waiting time)
scale = 2  # Average waiting time

# Probability density function (PDF) for a specific time
time = 3
pdf_value = expon.pdf(time, scale=scale)
print("PDF at time", time, ":", pdf_value)

# Cumulative distribution function (CDF) for a specific time
cdf_value = expon.cdf(time, scale=scale)
print("CDF at time", time, ":", cdf_value)
```

```
PDF at time 3 : 0.11156508007421491
CDF at time 3 : 0.7768698398515702
```

61. norm()

Library: scipy.stats

Use: Represents the normal (Gaussian) distribution, widely used in statistics to model data distributions. It allows calculation of probabilities, density values, and sampling.

Explanation pdf(x, loc, scale): Computes the probability density at a specific value x. cdf(x, loc, scale): Computes the cumulative probability up to x. rvs(loc, scale, size): Generates random samples. Key Features:

Fully specified by the mean (loc) and standard deviation (scale). Essential for many statistical methods and hypothesis testing.

In [96]:
```
from scipy.stats import norm

# Define the mean and standard deviation
mean = 0
std_dev = 1

# Probability density function (PDF) at a specific point
x = 1
pdf_value = norm.pdf(x, loc=mean, scale=std_dev)
print("PDF at x =", x, ":", pdf_value)
```

```
# Cumulative distribution function (CDF) at a specific point
cdf_value = norm.cdf(x, loc=mean, scale=std_dev)
print("CDF at x =", x, ":", cdf_value)

# Generate random samples
samples = norm.rvs(loc=mean, scale=std_dev, size=5)
print("Random samples:", samples)
```

```
PDF at x = 1 : 0.24197072451914337
CDF at x = 1 : 0.8413447460685429
Random samples: [ 1.00058232 -0.67062021  1.3924653  -0.25004651  0.2886936
3]
```

62. beta()

Library: scipy.stats

Use: Represents the beta distribution, which is a continuous probability distribution often used to model proportions or probabilities.

Explanation pdf(x, a, b): Computes the probability density at a specific value x. cdf(x, a, b): Computes the cumulative probability up to x. rvs(a, b, size): Generates random samples from the distribution. Key Features:

Parameters a and b (shape parameters) determine the shape of the distribution. Used in Bayesian statistics and modeling probabilities constrained between 0 and 1.

In [97]:
```
from scipy.stats import beta

# Define shape parameters (alpha, beta)
a, b = 2, 5

# Probability density function (PDF) at a specific point
x = 0.5
pdf_value = beta.pdf(x, a, b)
print("PDF at x =", x, ":", pdf_value)

# Cumulative distribution function (CDF) at a specific point
cdf_value = beta.cdf(x, a, b)
print("CDF at x =", x, ":", cdf_value)

# Generate random samples
samples = beta.rvs(a, b, size=5)
print("Random samples:", samples)
```

```
PDF at x = 0.5 : 0.9374999999999999
CDF at x = 0.5 : 0.890625
Random samples: [0.53844229 0.09297362 0.41099299 0.15775619 0.34407353]
```

63. binom()

Library: scipy.stats

Use: Represents the binomial distribution, which models the number of successes in a fixed number of independent trials, each with the same probability of success.

Explanation pmf(k, n, p): Probability of exactly k successes. cdf(k, n, p): Probability of k or fewer successes. rvs(n, p, size): Generates random samples from the binomial distribution. Key Features:

Used for modeling binary outcomes (e.g., success/failure). Parameters: n: Number of trials. p: Probability of success in each trial.

In [98]:
```python
from scipy.stats import binom

# Define parameters
n = 10  # Number of trials
p = 0.5  # Probability of success

# Probability mass function (PMF) for a specific number of successes
k = 5
pmf_value = binom.pmf(k, n, p)
print("P(X =", k, "):", pmf_value)

# Cumulative distribution function (CDF) for k successes or fewer
cdf_value = binom.cdf(k, n, p)
print("P(X <= ", k, "):", cdf_value)

# Generate random samples
samples = binom.rvs(n, p, size=5)
print("Random samples:", samples)
```

```
P(X = 5 ): 0.2460937500000002
P(X <=  5 ): 0.623046875
Random samples: [5 4 6 5 4]
```

64. uniform()

Library: scipy.stats

Use: Represents the uniform distribution, where all outcomes in a given interval have an equal probability. It is commonly used for generating random numbers in a range.

Explanation pdf(x, loc, scale): Probability density for a given value x. cdf(x, loc, scale): Cumulative probability up to x. rvs(loc, scale, size): Generates random samples within the interval. Key Features:

Useful for modeling random uniform distributions (e.g., rolling a fair die, sampling within a fixed interval). Parameters: loc: Starting value of the interval. scale: Width of the interval (end = loc + scale).

In [99]:
```python
from scipy.stats import uniform

# Define the range (loc: start, scale: length of interval)
start = 10
length = 20

# Probability density function (PDF) at a specific point
x = 15
pdf_value = uniform.pdf(x, loc=start, scale=length)
print("PDF at x =", x, ":", pdf_value)

# Cumulative distribution function (CDF) at a specific point
cdf_value = uniform.cdf(x, loc=start, scale=length)
print("CDF at x =", x, ":", cdf_value)

# Generate random samples
samples = uniform.rvs(loc=start, scale=length, size=5)
print("Random samples:", samples)
```

```
PDF at x = 15 : 0.05
CDF at x = 15 : 0.25
Random samples: [14.35785109 29.76071932 19.08003242 23.76548472 12.8110593
5]
```

65. gamma()

Library: scipy.stats

Use: Represents the gamma distribution, which is a continuous distribution widely used in queuing theory, reliability analysis, and Bayesian statistics. It is often used to model waiting times and is parameterized by a shape parameter and a scale parameter.

Explanation pdf(x, a, scale): Probability density at a specific value x. cdf(x, a, scale): Cumulative probability up to x. rvs(a, scale, size): Generates random samples from the distribution. Key Features:

Parameters: a: Shape parameter. scale: Scale parameter (inverse of rate). Often used in modeling lifetimes or waiting times between events.

In [100…
```python
from scipy.stats import gamma

# Define the shape (a) and scale (scale)
shape = 2
scale = 3

# Probability density function (PDF) at a specific point
x = 5
pdf_value = gamma.pdf(x, a=shape, scale=scale)
print("PDF at x =", x, ":", pdf_value)
```

```
# Cumulative distribution function (CDF) at a specific point
cdf_value = gamma.cdf(x, a=shape, scale=scale)
print("CDF at x =", x, ":", cdf_value)

# Generate random samples
samples = gamma.rvs(a=shape, scale=scale, size=5)
print("Random samples:", samples)
```

```
PDF at x = 5 : 0.10493089046531212
CDF at x = 5 : 0.4963317257665017
Random samples: [11.32501979  5.02130274  3.3746319   1.83884852  3.7255417
3]
```

66. geom()

Library: scipy.stats

Use: Represents the geometric distribution, which models the number of trials required to get the first success in a sequence of Bernoulli trials.

Explanation pmf(k, p): Probability of requiring exactly k trials to achieve the first success. cdf(k, p): Cumulative probability of achieving the first success within k trials. rvs(p, size): Generates random samples from the geometric distribution. Key Features:

Suitable for modeling the number of trials required for a success. Parameters: p: Probability of success in each trial.

In [101…

```
from scipy.stats import geom

# Define the probability of success
p = 0.5  # Probability of success

# Probability mass function (PMF) for a specific number of trials
k = 3
pmf_value = geom.pmf(k, p)
print("P(X =", k, "):", pmf_value)

# Cumulative distribution function (CDF) for k or fewer trials
cdf_value = geom.cdf(k, p)
print("P(X <= ", k, "):", cdf_value)

# Generate random samples
samples = geom.rvs(p, size=5)
print("Random samples:", samples)
```

```
P(X = 3 ): 0.125
P(X <=  3 ): 0.875
Random samples: [1 1 4 1 5]
```

67. weibull_min()

Library: scipy.stats

Use: Represents the Weibull distribution, commonly used in reliability analysis, life data analysis, and modeling failure times.

Explanation pdf(x, c, scale): Probability density at a specific value x. cdf(x, c, scale): Cumulative probability up to x. rvs(c, scale, size): Generates random samples from the Weibull distribution. Key Features:

Parameters: c: Shape parameter (controls the shape of the distribution). scale: Scale parameter (scales the distribution). Widely used for failure and reliability analysis.

In [102…

```python
from scipy.stats import weibull_min

# Define the shape parameter (c) and scale parameter
shape = 2  # Shape parameter
scale = 1  # Scale parameter

# Probability density function (PDF) at a specific point
x = 0.5
pdf_value = weibull_min.pdf(x, c=shape, scale=scale)
print("PDF at x =", x, ":", pdf_value)

# Cumulative distribution function (CDF) at a specific point
cdf_value = weibull_min.cdf(x, c=shape, scale=scale)
print("CDF at x =", x, ":", cdf_value)

# Generate random samples
samples = weibull_min.rvs(c=shape, scale=scale, size=5)
print("Random samples:", samples)
```

```
PDF at x = 0.5 : 0.7788007830714049
CDF at x = 0.5 : 0.22119921692859515
Random samples: [0.6138641  1.48083326 0.96108018 1.32392354 2.03371325]
```

68. pareto()

Library: scipy.stats

Use: Represents the Pareto distribution, which is commonly used in economics and social sciences to model wealth distribution, or in reliability analysis to model failure rates.

Explanation pdf(x, b, scale): Probability density at a specific value x. cdf(x, b, scale): Cumulative probability up to x. rvs(b, scale, size): Generates random samples from the Pareto distribution. Key Features:

Parameters: b: Shape parameter (alpha) determines the distribution's heaviness of the tail. scale: Scale parameter sets the minimum value for x. Models distributions with

heavy tails, such as wealth, file sizes, or failure rates.

In [103…

```
from scipy.stats import pareto

# Define the shape parameter (b) and scale parameter
shape = 2  # Shape parameter (alpha)
scale = 1  # Scale parameter

# Probability density function (PDF) at a specific point
x = 1.5
pdf_value = pareto.pdf(x, b=shape, scale=scale)
print("PDF at x =", x, ":", pdf_value)

# Cumulative distribution function (CDF) at a specific point
cdf_value = pareto.cdf(x, b=shape, scale=scale)
print("CDF at x =", x, ":", cdf_value)

# Generate random samples
samples = pareto.rvs(b=shape, scale=scale, size=5)
print("Random samples:", samples)
```

```
PDF at x = 1.5 : 0.5925925925925926
CDF at x = 1.5 : 0.5555555555555556
Random samples: [1.18545122 5.07251542 1.2798254  1.27568445 1.23127605]
```

69. logistic()

Library: scipy.stats

Use: Represents the logistic distribution, commonly used in logistic regression and machine learning for classification problems. It resembles the normal distribution but has heavier tails.

Explanation pdf(x, loc, scale): Probability density at a specific value x. cdf(x, loc, scale): Cumulative probability up to x. rvs(loc, scale, size): Generates random samples from the logistic distribution. Key Features:

Parameters: loc: Location parameter (mean of the distribution). scale: Scale parameter (spread of the distribution). Heavier tails than the normal distribution, making it useful for modeling outlier-prone datasets

In [104…

```
from scipy.stats import logistic

# Define the location (mean) and scale (spread)
loc = 0  # Mean
scale = 1  # Scale parameter

# Probability density function (PDF) at a specific point
x = 0.5
```

```
pdf_value = logistic.pdf(x, loc=loc, scale=scale)
print("PDF at x =", x, ":", pdf_value)

# Cumulative distribution function (CDF) at a specific point
cdf_value = logistic.cdf(x, loc=loc, scale=scale)
print("CDF at x =", x, ":", cdf_value)

# Generate random samples
samples = logistic.rvs(loc=loc, scale=scale, size=5)
print("Random samples:", samples)
```

```
PDF at x = 0.5 : 0.2350037122015945
CDF at x = 0.5 : 0.6224593312018546
Random samples: [ 0.16601065 -1.7024539   0.21567813  0.16720774  1.1631345
5]
```

70. laplace()

Library: scipy.stats

Use: Represents the Laplace (double exponential) distribution, often used in machine learning and statistics for robust modeling due to its sharper peak and heavier tails compared to the normal distribution.

Explanation pdf(x, loc, scale): Probability density at a specific value x. cdf(x, loc, scale): Cumulative probability up to x. rvs(loc, scale, size): Generates random samples from the Laplace distribution. Key Features:

Parameters: loc: Location parameter (mean of the distribution). scale: Scale parameter (spread of the distribution). The Laplace distribution is symmetric, with a sharper peak at the mean and heavier tails than the normal distribution.

In [105…

```
from scipy.stats import laplace

# Define the location (mean) and scale (spread)
loc = 0  # Mean
scale = 1  # Scale parameter

# Probability density function (PDF) at a specific point
x = 0.5
pdf_value = laplace.pdf(x, loc=loc, scale=scale)
print("PDF at x =", x, ":", pdf_value)

# Cumulative distribution function (CDF) at a specific point
cdf_value = laplace.cdf(x, loc=loc, scale=scale)
print("CDF at x =", cdf_value)

# Generate random samples
samples = laplace.rvs(loc=loc, scale=scale, size=5)
print("Random samples:", samples)
```

```
PDF at x = 0.5 : 0.3032653298563167
CDF at x = 0.6967346701436833
Random samples: [ 1.10051387 -0.12683081 -0.50307821 -0.65648783 -0.9435394
5]
```

71. t()

Library: scipy.stats

Use: Represents the Student's t-distribution, widely used in statistics for small sample hypothesis testing and confidence interval estimation.

Explanation pdf(x, df): Probability density at a specific value x. cdf(x, df): Cumulative probability up to x. rvs(df, size): Generates random samples from the t-distribution. Key Features:

Parameter: df: Degrees of freedom, which controls the shape of the distribution. The t-distribution is symmetric and has heavier tails than the normal distribution, making it suitable for small sample sizes.

In [106…

```python
from scipy.stats import t

# Define degrees of freedom
df = 10  # Degrees of freedom

# Probability density function (PDF) at a specific point
x = 0.5
pdf_value = t.pdf(x, df)
print("PDF at x =", x, ":", pdf_value)

# Cumulative distribution function (CDF) at a specific point
cdf_value = t.cdf(x, df)
print("CDF at x =", x, ":", cdf_value)

# Generate random samples
samples = t.rvs(df, size=5)
print("Random samples:", samples)
```

```
PDF at x = 0.5 : 0.3396951363520778
CDF at x = 0.5 : 0.6860531971285135
Random samples: [-0.19495488  0.05666085 -1.05162663 -0.13395342  0.7669140
4]
```

72. chi2()

Library: scipy.stats

Use: Represents the Chi-Square distribution, commonly used in hypothesis testing (e.g., goodness-of-fit tests) and for constructing confidence intervals for variance.

Explanation pdf(x, df): Probability density at a specific value x. cdf(x, df): Cumulative probability up to x. rvs(df, size): Generates random samples from the Chi-Square distribution. Key Features:

Parameter: df: Degrees of freedom, which determines the shape of the distribution. The Chi-Square distribution is non-negative and right-skewed, with its skewness decreasing as degrees of freedom increase.

In [107…

```
from scipy.stats import chi2

# Define degrees of freedom
df = 5  # Degrees of freedom

# Probability density function (PDF) at a specific point
x = 2
pdf_value = chi2.pdf(x, df)
print("PDF at x =", x, ":", pdf_value)

# Cumulative distribution function (CDF) at a specific point
cdf_value = chi2.cdf(x, df)
print("CDF at x =", x, ":", cdf_value)

# Generate random samples
samples = chi2.rvs(df, size=5)
print("Random samples:", samples)
```

```
PDF at x = 2 : 0.1383691658068649
CDF at x = 2 : 0.15085496391539038
Random samples: [3.14393814 1.55804209 1.21639969 5.08199764 5.16415768]
```

73. bernoulli()

Library: scipy.stats

Use: Represents the Bernoulli distribution, which models a single trial with two outcomes: success (1) with probability

(1) with probability p, or failure (0), with probability 1 – p

Explanation pmf(k, p): Probability of observing k (either 0 or 1). rvs(p, size): Generates random samples of 0s and 1s based on the probability of success p. Key Features:

Parameter: p: Probability of success. The Bernoulli distribution is a foundational concept in probability and statistics, forming the basis of the binomial distribution.

In [108…

```
from scipy.stats import bernoulli

# Define the probability of success
p = 0.7  # Probability of success
```

```
# Probability mass function (PMF) for each outcome
pmf_success = bernoulli.pmf(1, p)  # Probability of success (1)
pmf_failure = bernoulli.pmf(0, p)  # Probability of failure (0)
print("P(Success):", pmf_success)
print("P(Failure):", pmf_failure)

# Generate random samples
samples = bernoulli.rvs(p, size=10)
print("Random samples:", samples)
```

```
P(Success): 0.7
P(Failure): 0.29999999999999993
Random samples: [1 0 1 0 1 0 0 1 1 0]
```

74. hypergeom()

Library: scipy.stats

Use: Represents the hypergeometric distribution, which models the probability of

k successes in n draws from a population of size N containing K successes, without replacement.

Explanation pmf(k, N, K, n): Probability of exactly k successes.

cdf(k, N, K, n): Cumulative probability of k or fewer successes. rvs(N, K, n, size): Generates random samples from the hypergeometric distribution. Key Features:

Parameters:

N: Population size. K: Number of successes in the population. n: Number of draws.

Used in scenarios where sampling is done without replacement (e.g., selecting defective items from a batch).

In [109…

```
from scipy.stats import hypergeom

# Define the parameters
N = 20  # Total population size
K = 7   # Total number of successes in the population
n = 5   # Number of draws

# Probability mass function (PMF) for exactly 3 successes
k = 3
pmf_value = hypergeom.pmf(k, N, K, n)
print("P(X =", k, "):", pmf_value)

# Cumulative distribution function (CDF) for 3 or fewer successes
cdf_value = hypergeom.cdf(k, N, K, n)
print("P(X <= ", k, "):", cdf_value)
```

```python
# Generate random samples
samples = hypergeom.rvs(N, K, n, size=5)
print("Random samples:", samples)
```

```
P(X = 3 ): 0.17608359133126936
P(X <=  3 ): 0.9692982456140351
Random samples: [1 1 3 2 3]
```