# Data Science Assignment 1: Ritika Thakur (2022408) | Swarnima Prasad (2022525)

In [32]:
```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np
```

## Loading the data

In [33]:
```
df = pd.read_csv('AutoMPG.csv')
```

In [34]:
```
print(df.head())

print("\nData Information:")
print(df.info())

print("\nSummary Statistics:")
print(df.describe())

print("\nMissing Values:")
print(df.isnull().sum())
```

```
    mpg  cylinders  displacement horsepower  weight  acceleration  model year  \
0  18.0          8         307.0        130  3504.0          12.0        70.0
1  15.0          8         350.0        165  3693.0          11.5        70.0
2  18.0          8         318.0        150  3436.0          11.0        70.0
3  16.0          8         304.0        150  3433.0          12.0        70.0
4  17.0          8         302.0        140  3449.0          10.5        70.0

   origin
0     1.0
1     1.0
2     1.0
3     1.0
4     1.0

Data Information:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 399 entries, 0 to 398
Data columns (total 8 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   mpg           398 non-null    float64
 1   cylinders     399 non-null    int64
 2   displacement  398 non-null    float64
 3   horsepower    398 non-null    object
 4   weight        398 non-null    float64
 5   acceleration  398 non-null    float64
 6   model year    398 non-null    float64
 7   origin        398 non-null    float64
dtypes: float64(6), int64(1), object(1)
memory usage: 25.1+ KB
None

Summary Statistics:
              mpg   cylinders  displacement       weight  acceleration  \
count  398.000000  399.000000    398.000000   398.000000    398.000000
mean    23.514573    5.461153    193.425879  2960.047739     15.568090
std      7.815984    1.703638    104.269838   882.516758      2.757689
min      9.000000    3.000000     68.000000 -2065.000000      8.000000
25%     17.500000    4.000000    104.250000  2223.750000     13.825000
50%     23.000000    4.000000    148.500000  2803.500000     15.500000
75%     29.000000    8.000000    262.000000  3608.000000     17.175000
max     46.600000    8.000000    455.000000  5140.000000     24.800000

         model year      origin
count    398.000000  398.000000
mean     327.062814    1.572864
std     5008.688771    0.802055
min       70.000000    1.000000
25%       73.000000    1.000000
50%       76.000000    1.000000
75%       79.000000    2.000000
max    99999.000000    3.000000

Missing Values:
mpg             1
cylinders       0
displacement    1
horsepower      1
weight          1
acceleration    1
```

```
model year      1
origin          1
dtype: int64
```

Dropping the redundant last row

In [35]:
```
# Drop the last row using the index of the last row
df.drop(df.index[-1], inplace=True)
```

### Handling Missing Values

In [36]:
```
# Replace '?' with NaN
df['horsepower'].replace('?', pd.NA, inplace=True)

# Drop rows where 'horsepower' is missing
df.dropna(subset=['horsepower'], inplace=True)

# Convert 'horsepower' to numeric after dropping rows
df['horsepower'] = pd.to_numeric(df['horsepower'])

# Verify the changes
print(df['horsepower'].head())
```

```
0    130
1    165
2    150
3    150
4    140
Name: horsepower, dtype: int64
```

```
C:\Users\Ritika\AppData\Local\Temp\ipykernel_52084\2527508700.py:2: FutureWarnin
g: A value is trying to be set on a copy of a DataFrame or Series through chained
assignment using an inplace method.
The behavior will change in pandas 3.0. This inplace method will never work becau
se the intermediate object on which we are setting values always behaves as a cop
y.

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.meth
od({col: value}, inplace=True)' or df[col] = df[col].method(value) instead, to pe
rform the operation inplace on the original object.


  df['horsepower'].replace('?', pd.NA, inplace=True)
```

In [37]:
```
# Verify the changes
print(df.tail())
print("\nMissing Values:")
print(df.isnull().sum())
```

```
      mpg  cylinders  displacement  horsepower  weight  acceleration  \
393  27.0          4         140.0          86  2790.0          15.6
394  44.0          4          97.0          52  2130.0          24.6
395  32.0          4         135.0          84  2295.0          11.6
396  28.0          4         120.0          79  2625.0          18.6
397  31.0          4         119.0          82  2720.0          19.4

     model year  origin
393        82.0     1.0
394        82.0     2.0
395        82.0     1.0
396        82.0     1.0
397        82.0     1.0

Missing Values:
mpg             0
cylinders       0
displacement    0
horsepower      0
weight          0
acceleration    0
model year      0
origin          0
dtype: int64
```

In [38]:
```
print(df.info())
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 392 entries, 0 to 397
Data columns (total 8 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   mpg           392 non-null    float64
 1   cylinders     392 non-null    int64
 2   displacement  392 non-null    float64
 3   horsepower    392 non-null    int64
 4   weight        392 non-null    float64
 5   acceleration  392 non-null    float64
 6   model year    392 non-null    float64
 7   origin        392 non-null    float64
dtypes: float64(6), int64(2)
memory usage: 27.6 KB
None
```

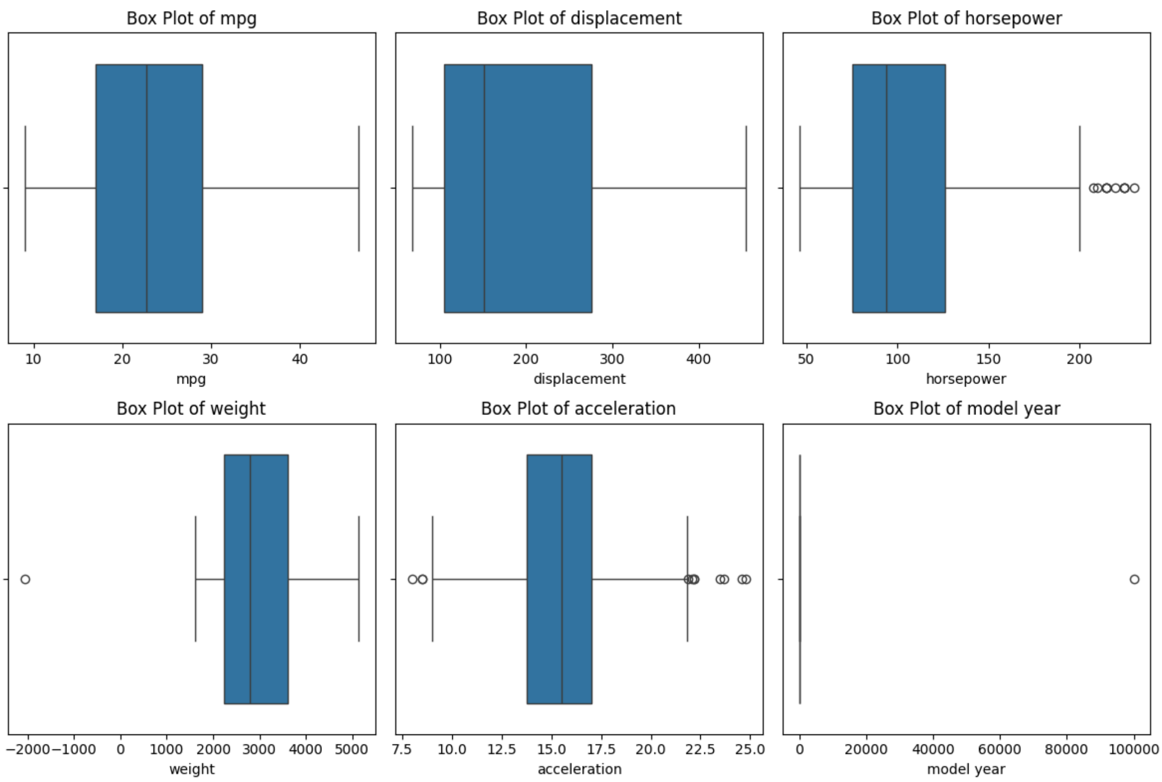
In [39]:
```
# Total number of rows before handling '?'
initial_row_count = pd.read_csv('AutoMPG.csv').shape[0]

# Total number of rows after handling '?'
final_row_count = df.shape[0]

print(f"Initial number of rows: {initial_row_count}")
print(f"Final number of rows: {final_row_count}")
```

```
Initial number of rows: 399
Final number of rows: 392
```

## Trying outlier detection Techniques

In [40]:
```
numerical_columns = ['mpg', 'displacement', 'horsepower', 'weight', 'acceleratic
```

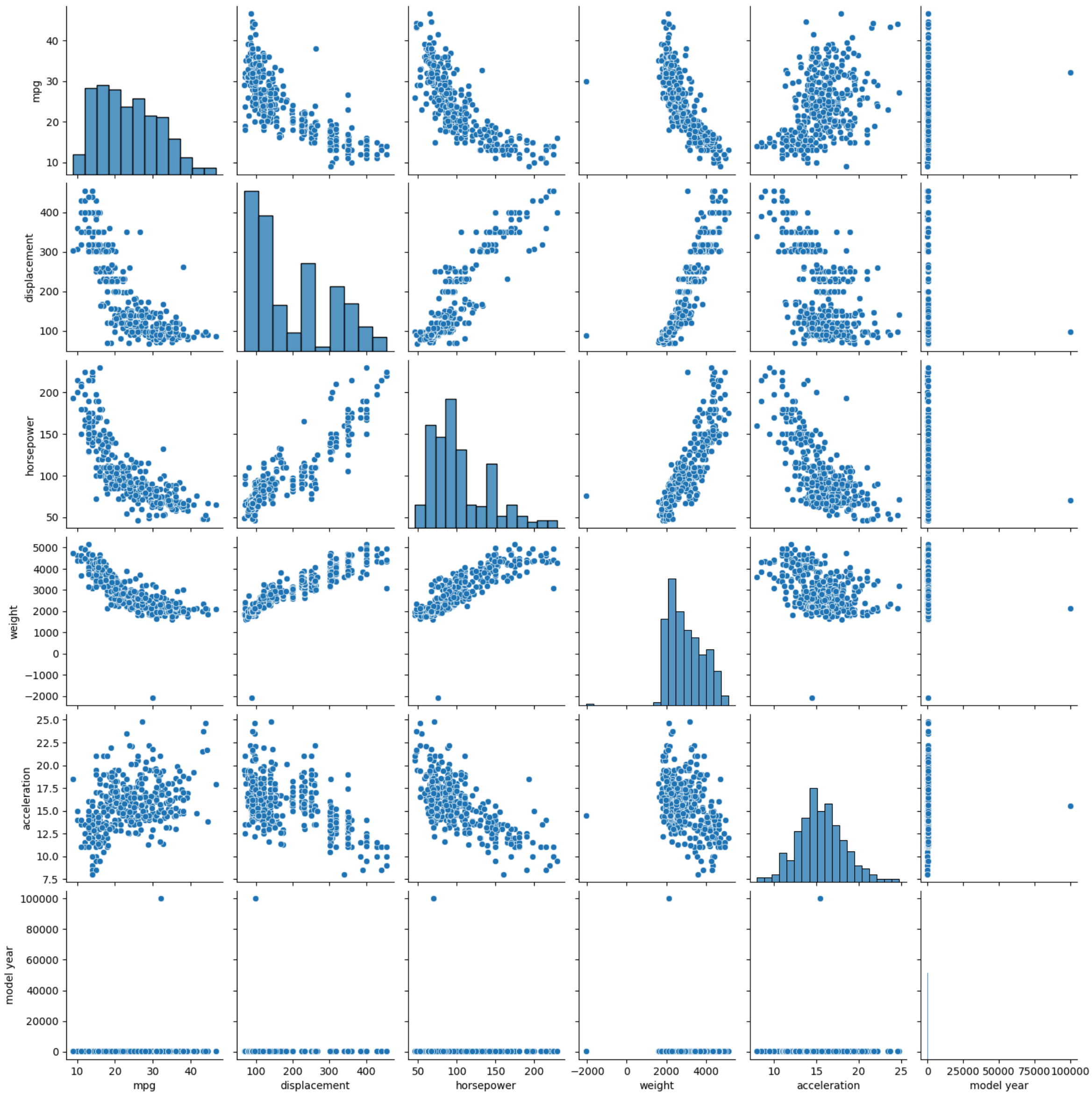
```
plt.figure(figsize=(12, 8))
for i, col in enumerate(numerical_columns, 1):
    plt.subplot(2, 3, i)
    sns.boxplot(x=df[col])
    plt.title(f'Box Plot of {col}')
plt.tight_layout()
plt.show()
```




Horsepower and acceleration are the two features with most outliers.

In [41]:
```
# Scatter plots to check relationships and outliers
plt.figure(figsize=(12, 8))
sns.pairplot(df[numerical_columns])
plt.show()
```

```
<Figure size 1200x800 with 0 Axes>
```




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

# Calculate Z-scores for numerical columns
z_scores = df[numerical_columns].apply(zscore)

outliers = (z_scores.abs() > 3).sum()
print("Number of outliers in each column based on Z-score:")
print(outliers)
```

```
Number of outliers in each column based on Z-score:
mpg             0
displacement    0
horsepower      5
weight          1
acceleration    2
model year      1
dtype: int64
```

Outliers with Z-score > 3 or < -3

In [43]:
```
# Function to identify outliers using IQR
def detect_outliers_iqr(df, col):
    Q1 = df[col].quantile(0.25)
    Q3 = df[col].quantile(0.75)
```

```
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR
    return df[(df[col] < lower_bound) | (df[col] > upper_bound)]

# Check outliers for each numerical column
for col in numerical_columns:
    outliers_iqr = detect_outliers_iqr(df, col)
    print(f'Number of outliers in {col} based on IQR: {len(outliers_iqr)}')
```

```
Number of outliers in mpg based on IQR: 0
Number of outliers in displacement based on IQR: 0
Number of outliers in horsepower based on IQR: 10
Number of outliers in weight based on IQR: 1
Number of outliers in acceleration based on IQR: 11
Number of outliers in model year based on IQR: 1
```

## Removing Outliers

In [44]:

```
# Function to calculate outlier bounds
def calculate_outlier_bounds(column):
    Q1 = df[column].quantile(0.25)
    Q3 = df[column].quantile(0.75)
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR
    return lower_bound, upper_bound

# Calculate outlier bounds for each numerical column
initial_shape = df.shape
outlier_bounds = {col: calculate_outlier_bounds(col) for col in numerical_column
print(outlier_bounds)
```

```
{'mpg': (np.float64(-1.0), np.float64(47.0)), 'displacement': (np.float64(-151.12
5), np.float64(531.875)), 'horsepower': (np.float64(-1.5), np.float64(202.5)), 'w
eight': (np.float64(141.0), np.float64(5699.0)), 'acceleration': (np.float64(8.90
0000000000002), np.float64(21.899999999999995)), 'model year': (np.float64(64.0),
np.float64(88.0))}
```

In [45]:

```
# Function to remove outliers based on bounds
def remove_outliers(df, column, lower_bound, upper_bound):
    return df[(df[column] >= lower_bound) & (df[column] <= upper_bound)]

# Apply outlier removal for each numerical column
for col in numerical_columns:
    lower_bound, upper_bound = outlier_bounds[col]
    df = remove_outliers(df, col, lower_bound, upper_bound)
```

In [46]:

```
print(f"Shape before removing outliers: {initial_shape}")
print(f"Shape after removing outliers: {df.shape}")
```

```
Shape before removing outliers: (392, 8)
Shape after removing outliers: (370, 8)
```

In [47]:

```
# Save DataFrame to a CSV file
df.to_csv('Cleaned_AutoMPG.csv', index=False)
```

In [48]:

```
df = pd.read_csv('Cleaned_AutoMPG.csv')
```

In [49]:
```
print("\nCleaned Data Summary:")
print(df.describe())
```

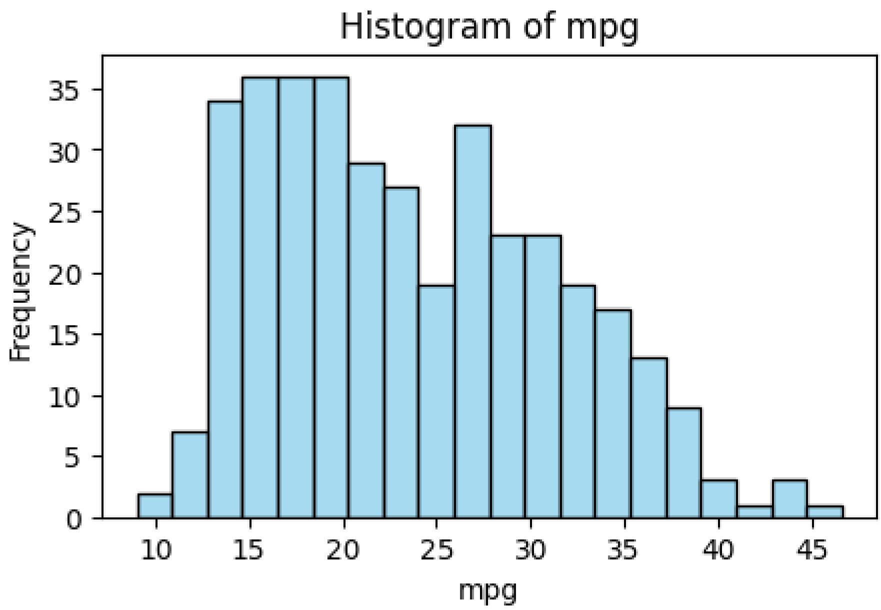
```
Cleaned Data Summary:
              mpg   cylinders  displacement  horsepower       weight  \
count  370.000000  370.000000    370.000000  370.000000   370.000000
mean    23.612703    5.418919    189.458108  102.035135  2948.648649
std      7.633255    1.674022     98.730198   33.684675   829.145929
min      9.000000    3.000000     68.000000   46.000000  1613.000000
25%     17.600000    4.000000    105.000000   76.000000  2223.750000
50%     23.000000    4.000000    146.000000   92.500000  2781.500000
75%     29.000000    6.000000    258.000000  120.000000  3556.000000
max     46.600000    8.000000    429.000000  200.000000  5140.000000

       acceleration  model year      origin
count    370.000000  370.000000  370.000000
mean      15.549189   76.113514    1.594595
std        2.408187    3.622884    0.818468
min        9.500000   70.000000    1.000000
25%       14.000000   73.000000    1.000000
50%       15.500000   76.000000    1.000000
75%       17.000000   79.000000    2.000000
max       21.800000   82.000000    3.000000
```

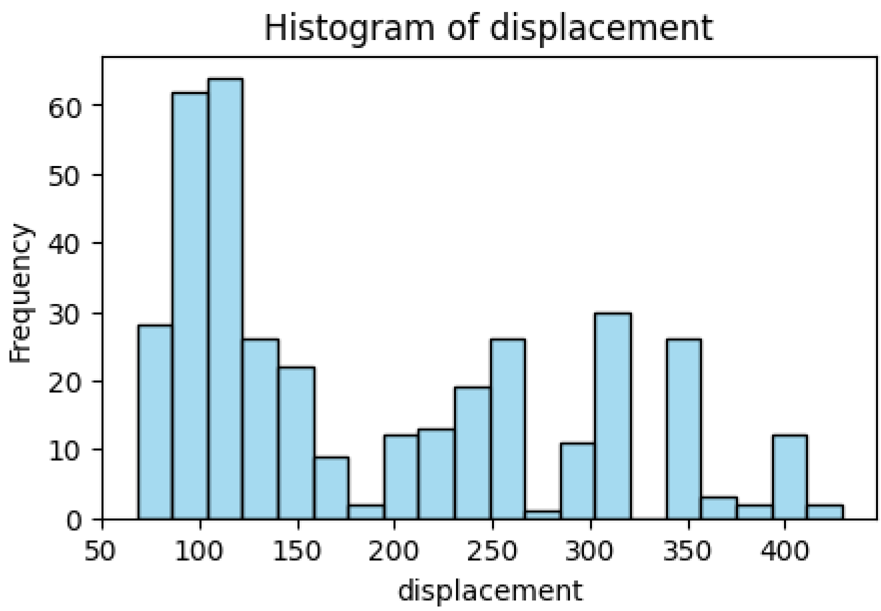
```
              mpg   cylinders  displacement  horsepower       weight  \
count  370.000000  370.000000    370.000000  370.000000   370.000000
mean    23.612703    5.418919    189.458108  102.035135  2948.648649
std      7.633255    1.674022     98.730198   33.684675   829.145929
min      9.000000    3.000000     68.000000   46.000000  1613.000000
25%     17.600000    4.000000    105.000000   76.000000  2223.750000
50%     23.000000    4.000000    146.000000   92.500000  2781.500000
75%     29.000000    6.000000    258.000000  120.000000  3556.000000
max     46.600000    8.000000    429.000000  200.000000  5140.000000

       acceleration  model year      origin
count    370.000000  370.000000  370.000000
mean      15.549189   76.113514    1.594595
std        2.408187    3.622884    0.818468
min        9.500000   70.000000    1.000000
25%       14.000000   73.000000    1.000000
50%       15.500000   76.000000    1.000000
75%       17.000000   79.000000    2.000000
max       21.800000   82.000000    3.000000
```

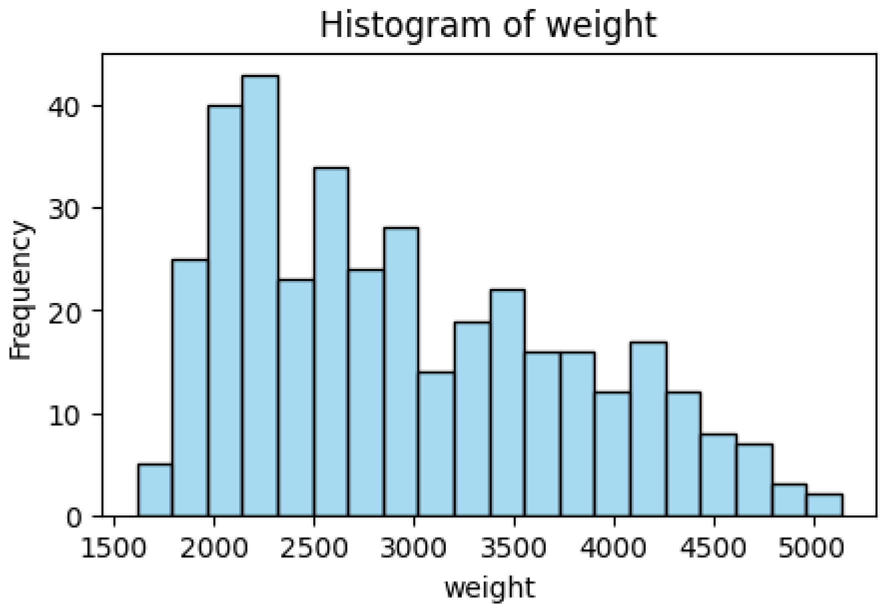

## Plotting histograms to assess the distribution of the data

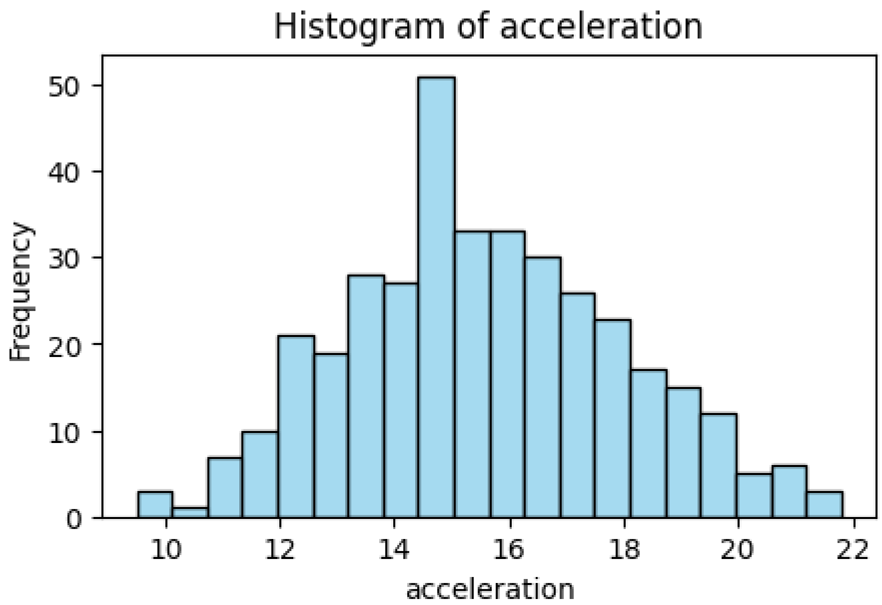
In [50]:
```
def plot_histogram(attribute):
    if attribute in df.columns:
        plt.figure(figsize=(5, 3))
        sns.histplot(df[attribute], kde=False, bins=20, color="skyblue")
        plt.title(f'Histogram of {attribute}')
        plt.xlabel(attribute)
        plt.ylabel('Frequency')
        plt.show()
    else:
        print(f"{attribute} not found in dataset.")

# Plot histograms for continuous attributes
plot_histogram('mpg')
plot_histogram('displacement')
```

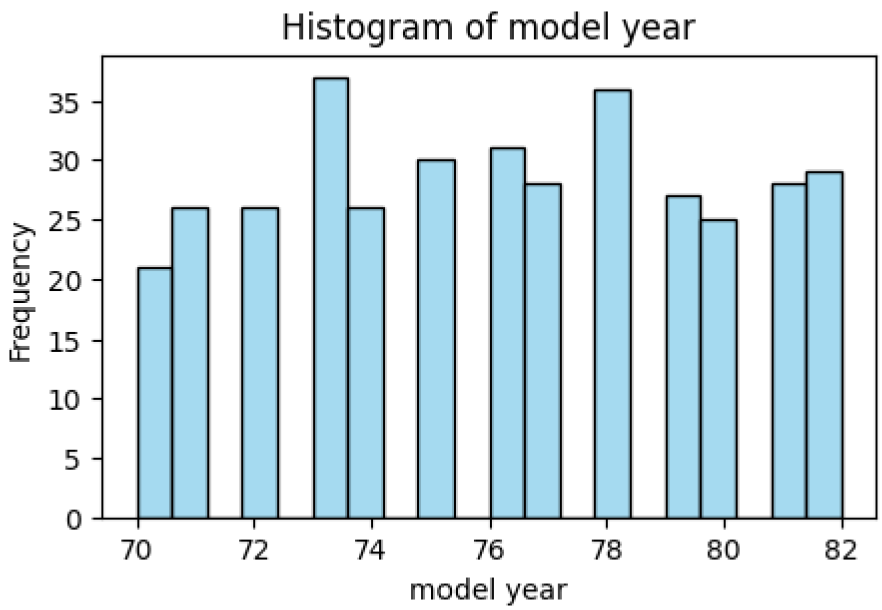
```
plot_histogram('weight')
plot_histogram('acceleration')
plot_histogram('model year')
plot_histogram('horsepower')
plot_histogram('cylinders')
plot_histogram('origin')
```

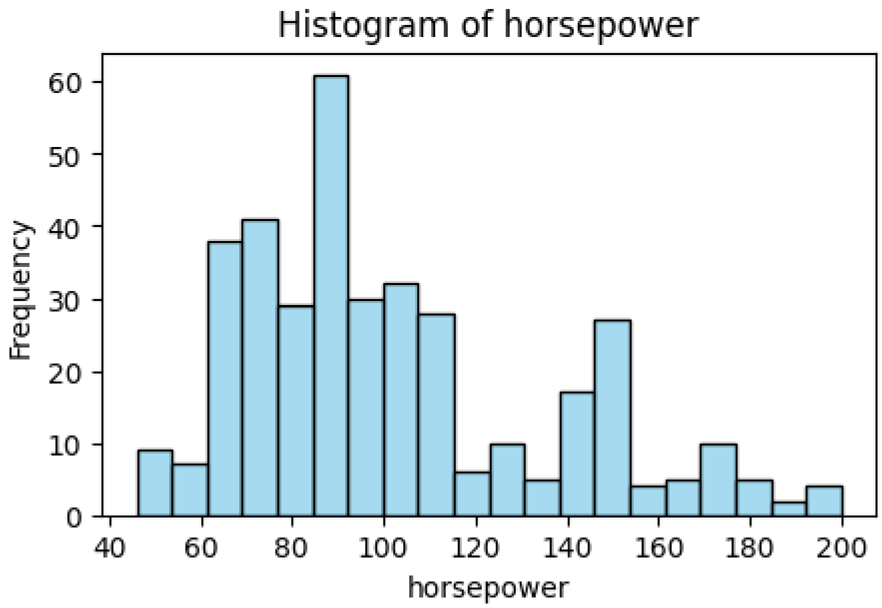


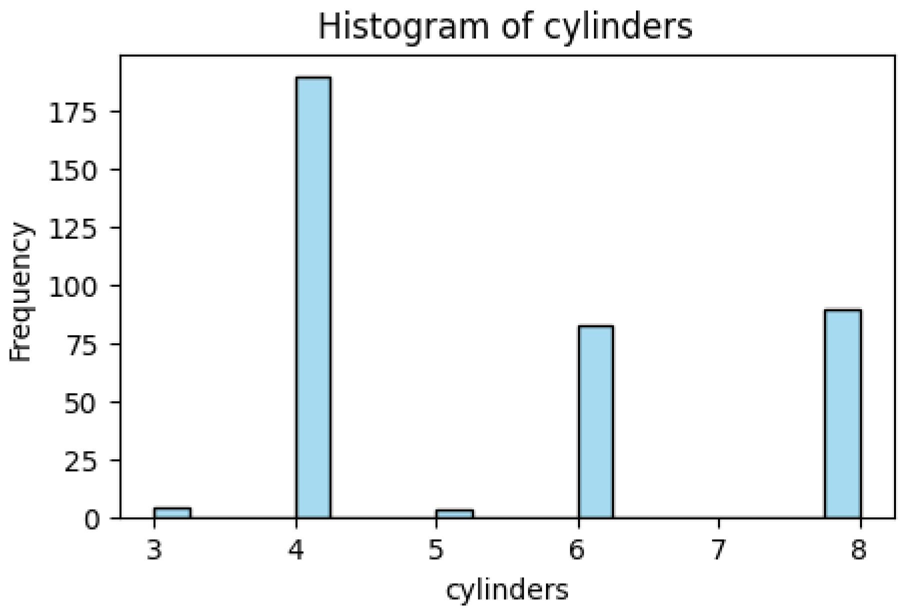



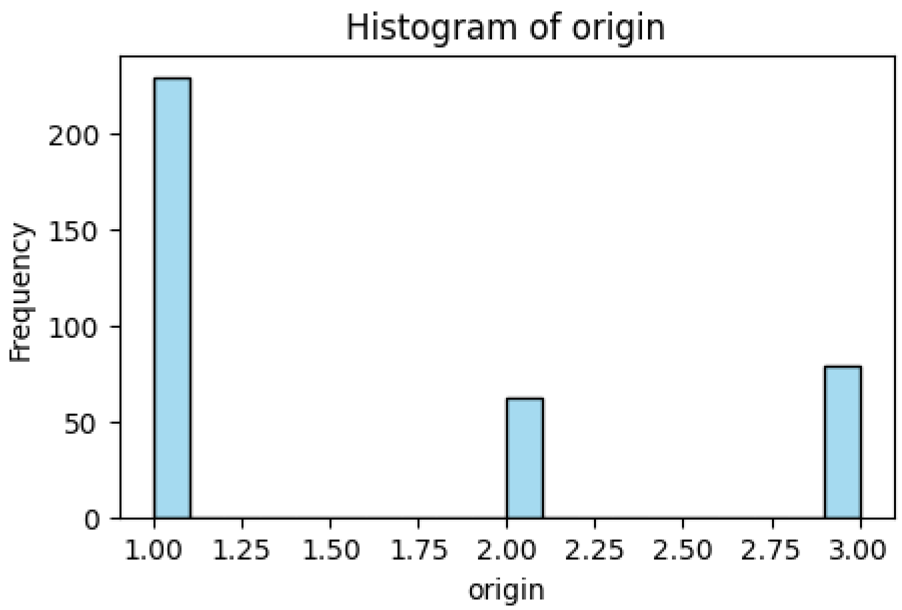

Histogram of weight
Frequency
weight
0
10
20
30
40
1500
2000
2500
3000
3500
4000
4500
5000


Histogram of acceleration
Frequency
acceleration
0
10
20
30
40
50
10
12
14
16
18
20
22


Histogram of model year
Frequency
model year
0
5
10
15
20
25
30
35
70
72
74
76
78
80
82




In [51]:
```
features = df.select_dtypes(include=[np.number])
print(features)
```

```
      mpg  cylinders  displacement  horsepower  weight  acceleration  \
0    18.0          8         307.0         130  3504.0          12.0
1    15.0          8         350.0         165  3693.0          11.5
2    18.0          8         318.0         150  3436.0          11.0
3    16.0          8         304.0         150  3433.0          12.0
4    17.0          8         302.0         140  3449.0          10.5
..    ...        ...           ...         ...     ...           ...
365  27.0          4         151.0          90  2950.0          17.3
366  27.0          4         140.0          86  2790.0          15.6
367  32.0          4         135.0          84  2295.0          11.6
368  28.0          4         120.0          79  2625.0          18.6
369  31.0          4         119.0          82  2720.0          19.4

     model year  origin
0          70.0     1.0
1          70.0     1.0
2          70.0     1.0
3          70.0     1.0
4          70.0     1.0
..          ...     ...
365        82.0     1.0
366        82.0     1.0
367        82.0     1.0
368        82.0     1.0
369        82.0     1.0

[370 rows x 8 columns]
```

# Calculating Mean , Variance and Squared Deviation on attributes

## Mean of each feature

$$\mu = \frac{\sum_{i=1}^{n} x_i}{n}$$

In [52]:
```python
def calculate_mean(df):
    num_rows = len(df)
    mean_vector = df.sum() / num_rows
    return mean_vector

mean_vector = calculate_mean(df)
print("Mean of each feature:\n", mean_vector)
```

```
Mean of each feature:
 mpg               23.612703
cylinders          5.418919
displacement     189.458108
horsepower       102.035135
weight          2948.648649
acceleration      15.549189
model year        76.113514
origin             1.594595
dtype: float64
```

## Variance of each feature

$$\sigma^2 = \frac{\sum_{i=1}^{n}(x_i - \mu)^2}{n}$$

In [53]:
```
def calculate_variance(df, mean_vector):
    num_rows = len(df)
    variance = ((df - mean_vector) ** 2).sum() / num_rows  # Manual calculation
    return variance

variance_vector = calculate_variance(df, mean_vector)
print("Variance of each feature:\n", variance_vector)
```

```
Variance of each feature:
 mpg                58.109109
cylinders           2.794777
displacement     9721.307029
horsepower       1131.590657
weight         685624.908985
acceleration        5.783689
model year         13.089817
origin              0.668079
dtype: float64
```

### Total Variance

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^T(x_i - \mu)$$

In [54]:
```
total_variance = 0
for i in range(len(df)):
    total_variance += np.dot((df.iloc[i] - mean_vector).T, (df.iloc[i] - mean_ve
total_variance = total_variance / len(df)
print("\nTotal Variance  (σ²):")
print(total_variance)
```

```
Total Variance  (σ²):
696558.2521418552
```

## Normalizing Data

Standard deviation:

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \mu)^2}{n}}$$

Standardization:

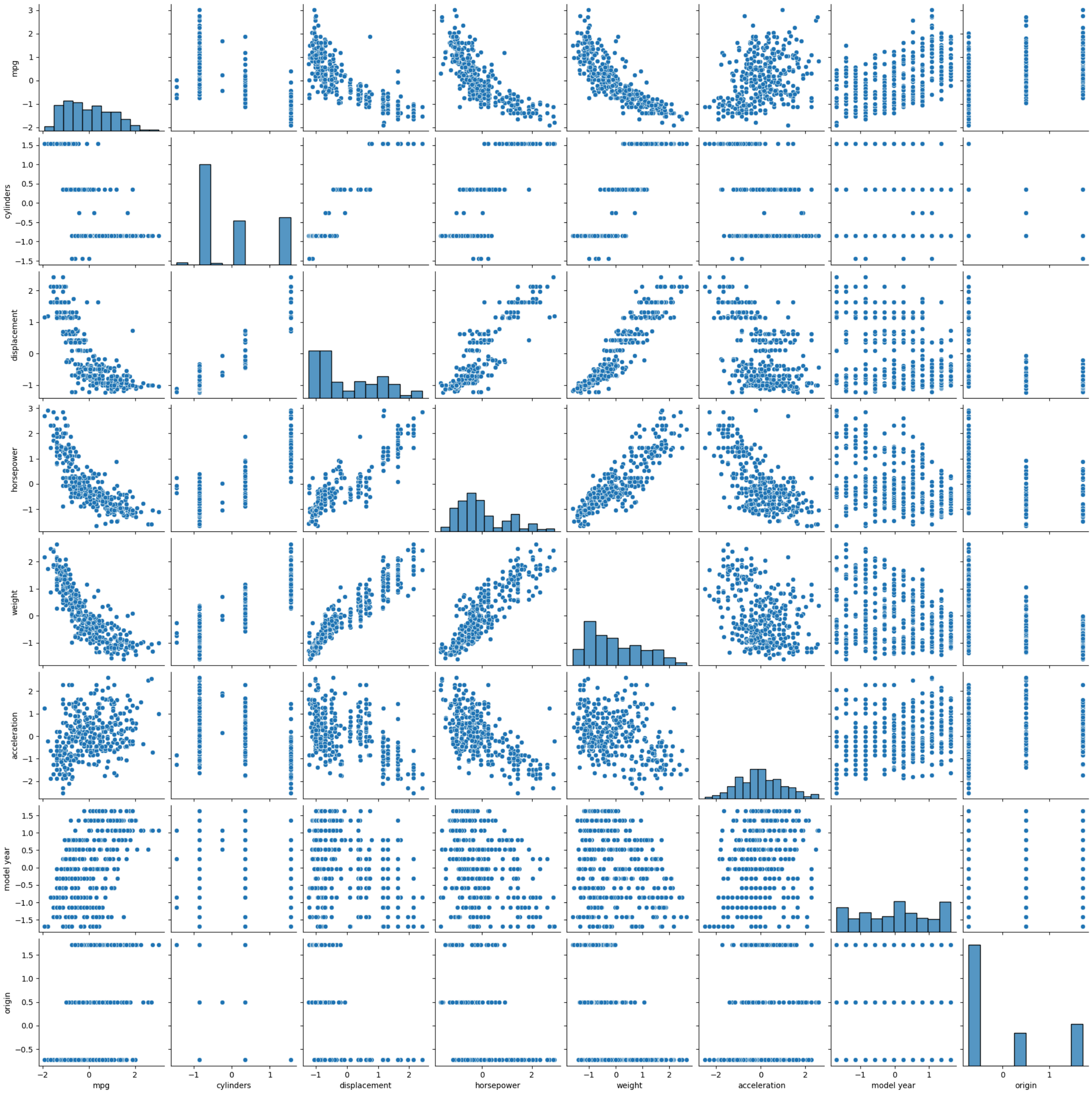
$$x_{new} = \frac{x - \mu}{\sigma}$$

In [55]:
```
def standardize_data(df):
    mean_vector = df.mean()  # Compute the mean for each column
    print(mean_vector)
    std_vector =   (calculate_variance(df, mean_vector))**0.5 # Compute the stan
    print(std_vector)
    standardized_df = (df - mean_vector) / std_vector  # Perform standardization
    return standardized_df, mean_vector, std_vector
```

```python
# Normalize the cleaned dataset
standardized_df, mean_vector, std_vector = standardize_data(df)
print("Standardized data:\n", standardized_df)
```

```
mpg              23.612703
cylinders         5.418919
displacement    189.458108
horsepower      102.035135
weight         2948.648649
acceleration     15.549189
model year       76.113514
origin            1.594595
dtype: float64
mpg             7.622933
cylinders       1.671759
displacement   98.596689
horsepower     33.639124
weight        828.024703
acceleration    2.404930
model year      3.617985
origin          0.817361
dtype: float64
Standardized data:
           mpg  cylinders  displacement  horsepower    weight  acceleration  \
0   -0.736292   1.543932      1.192148    0.831320  0.670694     -1.475797
1   -1.129841   1.543932      1.628269    1.871775  0.898948     -1.683704
2   -0.736292   1.543932      1.303714    1.425865  0.588571     -1.891610
3   -0.998658   1.543932      1.161721    1.425865  0.584948     -1.475797
4   -0.867475   1.543932      1.141437    1.128593  0.604271     -2.099516
..        ...        ...           ...         ...       ...           ...
365  0.444356  -0.848758     -0.390055   -0.357772  0.001632      0.728009
366  0.444356  -0.848758     -0.501620   -0.476681 -0.191599      0.021128
367  1.100272  -0.848758     -0.552332   -0.536136 -0.789407     -1.642122
368  0.575539  -0.848758     -0.704467   -0.684772 -0.390868      1.268565
369  0.969089  -0.848758     -0.714609   -0.595590 -0.276137      1.601215

     model year    origin
0     -1.689756 -0.727457
1     -1.689756 -0.727457
2     -1.689756 -0.727457
3     -1.689756 -0.727457
4     -1.689756 -0.727457
..          ...       ...
365    1.627007 -0.727457
366    1.627007 -0.727457
367    1.627007 -0.727457
368    1.627007 -0.727457
369    1.627007 -0.727457

[370 rows x 8 columns]
```

In [56]:
```python
sns.pairplot(standardized_df)
plt.show()
```




In [57]:
```
mean_scaled = standardized_df.mean()
print(mean_scaled)
variance_scaled=calculate_variance(standardized_df, mean_scaled)
scaled_std_vector = (variance_scaled)**0.5
print(variance_scaled)
```

```
mpg            -3.840772e-16
cylinders      -1.728347e-16
displacement   -1.344270e-16
horsepower     -5.761157e-17
weight         -1.536309e-16
acceleration    9.601929e-17
model year      9.217852e-16
origin          1.152231e-16
dtype: float64
mpg             1.0
cylinders       1.0
displacement    1.0
horsepower      1.0
weight          1.0
acceleration    1.0
model year      1.0
origin          1.0
dtype: float64
```

## Total Variance After normalizing

In [58]:
```
total_variance = 0
for i in range(len(standardized_df)):
    total_variance += np.dot((standardized_df.iloc[i] - mean_scaled).T, (standar
total_variance = total_variance / len(standardized_df)
print("\nTotal Variance  (σ²):")
print(total_variance)
```

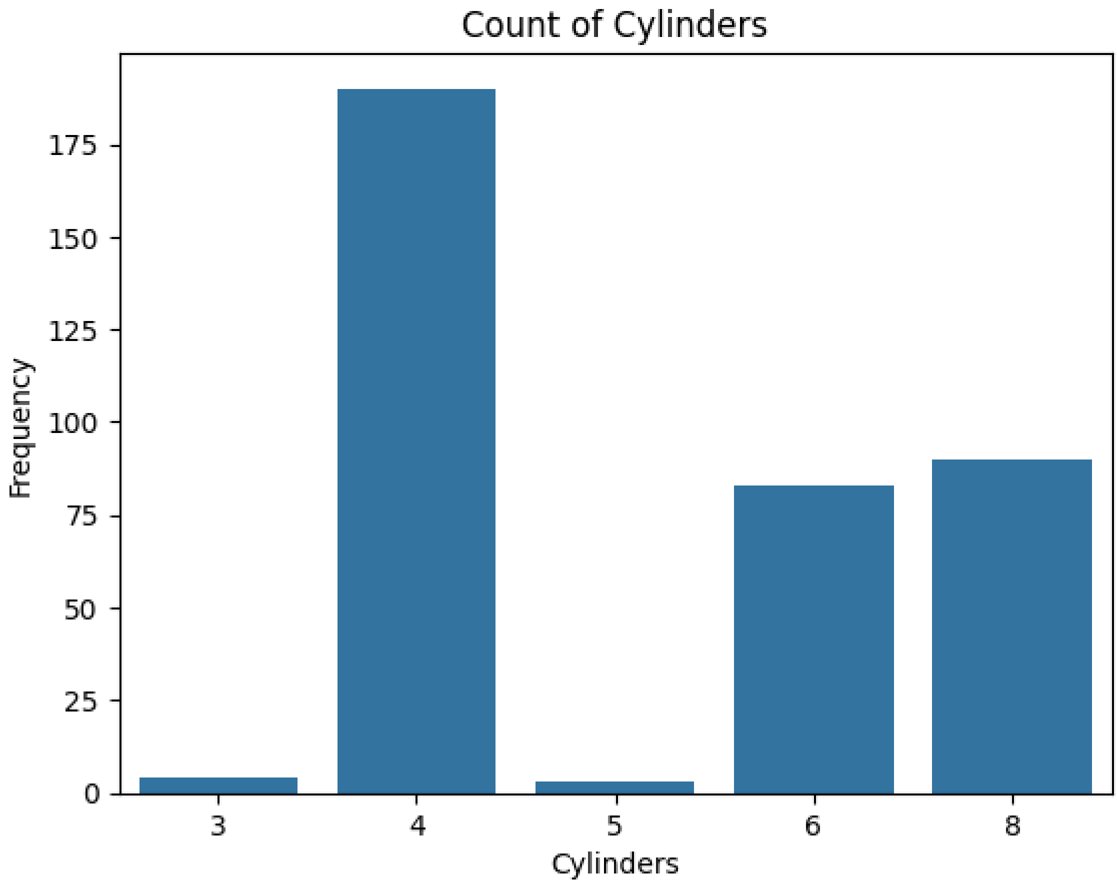
```
Total Variance  (σ²):
8.000000000000004

8.000000000000004
```

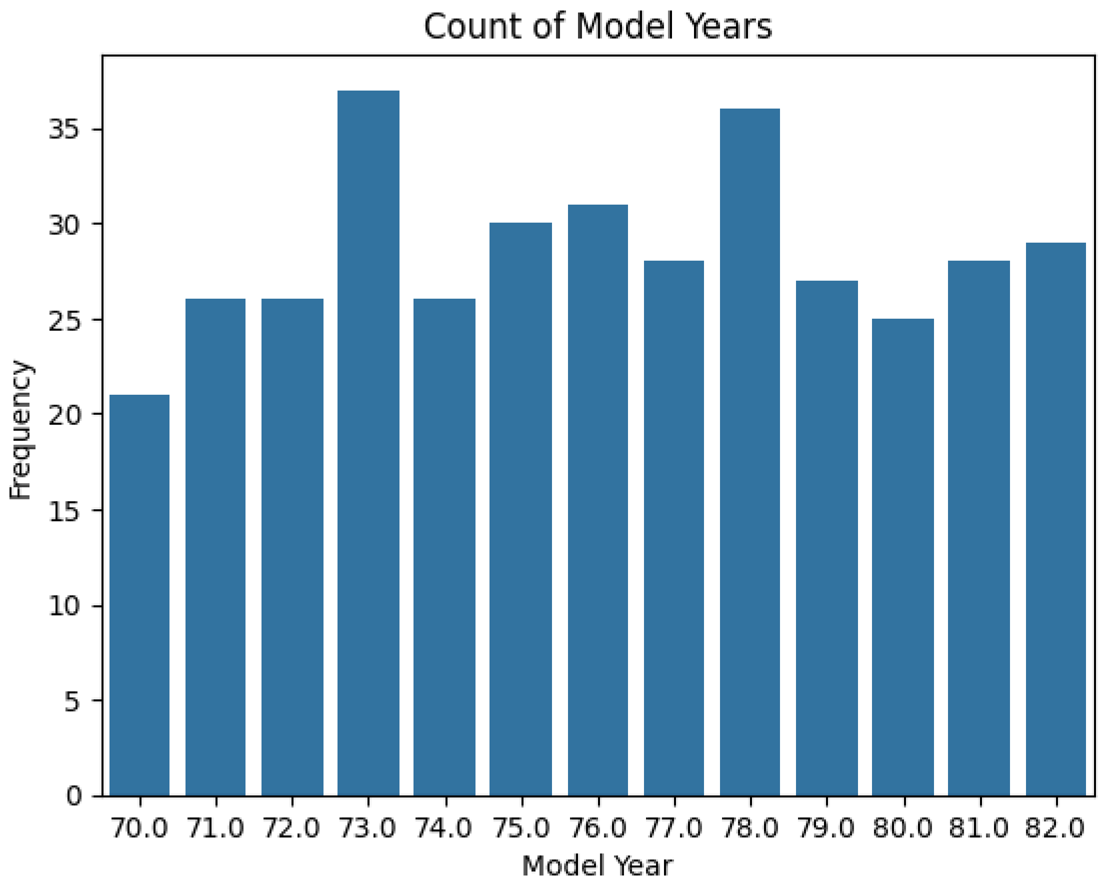
Convert 'model year' and 'cylinders' to categorical data types

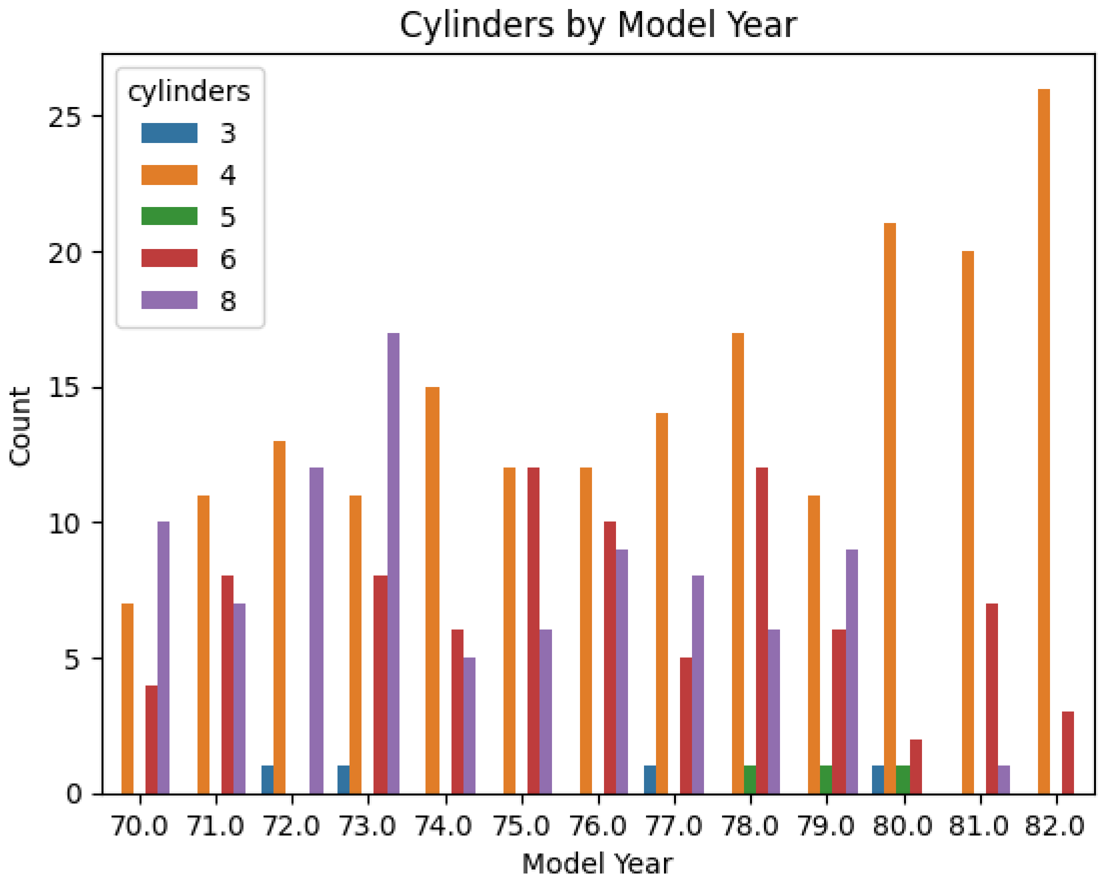
In [59]:
```
df['model year'] = df['model year'].astype('category')
df['cylinders'] = df['cylinders'].astype('category')

print(df.dtypes)
```

```
mpg              float64
cylinders       category
displacement     float64
horsepower         int64
weight           float64
acceleration     float64
model year      category
origin           float64
dtype: object
```

In [60]:
```
sns.countplot(data=df, x='cylinders')
plt.title("Count of Cylinders")
plt.xlabel("Cylinders")
plt.ylabel("Frequency")
plt.show()

sns.countplot(data=df, x='model year')
plt.title("Count of Model Years")
plt.xlabel("Model Year")
plt.ylabel("Frequency")
plt.show()

sns.countplot(data=df, x='model year', hue='cylinders')
plt.title("Cylinders by Model Year")
plt.xlabel("Model Year")
plt.ylabel("Count")
plt.show()
```


Count of Cylinders
Frequency
0
25
50
75
100
125
150
175
3
4
5
6
8
Cylinders


Count of Model Years
Frequency
0
5
10
15
20
25
30
35
70.0
71.0
72.0
73.0
74.0
75.0
76.0
77.0
78.0
79.0
80.0
81.0
82.0
Model Year




This visualization shows how the cylinders and model year values are categorical. Since both variables are categorical, the Chi-Square Test is appropriate for testing the independence of these variables.

## Chi-Square Test

Null Hypothesis ($H_0$): There is no significant association between the number of cylinders and the model year of the cars in the dataset. In other words, the variables model year and cylinders are independent.

Alternative Hypothesis ($H_1$): There is a significant association between the number of cylinders and the model year of the cars. In other words, the variables model year and cylinders are not independent.

Chi-Square:

$$\chi^2 = \sum \frac{(O_i - E_i)^2}{E_i}$$

In [61]:

```
contingency_table = pd.crosstab(df['model year'], df['cylinders'])
print("Contingency Table:")
print(contingency_table)

total = contingency_table.to_numpy().sum()

expected_table = np.outer(contingency_table.sum(axis=1), contingency_table.sum(a
expected_table = pd.DataFrame(expected_table, index=contingency_table.index, col
```

```python
print("\nExpected Frequencies:")
print(expected_table)

chi2_stat = ((contingency_table - expected_table) ** 2 / expected_table).to_nump

dof = (contingency_table.shape[0] - 1) * (contingency_table.shape[1] - 1)

print(f"\nChi-Square Statistic: {chi2_stat}")
print(f"Degrees of Freedom: {dof}")
```

```
Contingency Table:
cylinders   3   4  5   6   8
model year
70.0        0   7  0   4  10
71.0        0  11  0   8   7
72.0        1  13  0   0  12
73.0        1  11  0   8  17
74.0        0  15  0   6   5
75.0        0  12  0  12   6
76.0        0  12  0  10   9
77.0        1  14  0   5   8
78.0        0  17  1  12   6
79.0        0  11  1   6   9
80.0        1  21  1   2   0
81.0        0  20  0   7   1
82.0        0  26  0   3   0

Expected Frequencies:
cylinders          3          4         5         6         8
model year
70.0        0.227027  10.783784  0.170270  4.710811  5.108108
71.0        0.281081  13.351351  0.210811  5.832432  6.324324
72.0        0.281081  13.351351  0.210811  5.832432  6.324324
73.0        0.400000  19.000000  0.300000  8.300000  9.000000
74.0        0.281081  13.351351  0.210811  5.832432  6.324324
75.0        0.324324  15.405405  0.243243  6.729730  7.297297
76.0        0.335135  15.918919  0.251351  6.954054  7.540541
77.0        0.302703  14.378378  0.227027  6.281081  6.810811
78.0        0.389189  18.486486  0.291892  8.075676  8.756757
79.0        0.291892  13.864865  0.218919  6.056757  6.567568
80.0        0.270270  12.837838  0.202703  5.608108  6.081081
81.0        0.302703  14.378378  0.227027  6.281081  6.810811
82.0        0.313514  14.891892  0.235135  6.505405  7.054054

Chi-Square Statistic: 98.92845355132319
Degrees of Freedom: 48
```

### Conclusion

Since 98.93 > 65.171, you reject the null hypothesis and conclude that there is a significant association between model year and cylinders. (From the Chi-Square Test table)

## References


- https://www.scribbr.com/statistics/chi-square-test-of-independence/
- https://www.simplilearn.com/tutorials/statistics-tutorial/hypothesis-testing-in-statistics#:~:text=Choose%20a%20statistical%20test%20based,%2Dtailed%20or%20two

- https://www.medcalc.org/manual/statistical-tables.php

In [49]:
```
import numpy as np
import matplotlib.pyplot as plt
```

# Creating population of 1,00,000 points uniformly distributed between 0.01 and 1000

In [50]:
```
# creating data consisting of 100000 points uniformly distribtuied between 0.01
data = np.zeros(100000)
for i in range(100000):
    data[i] = 0.01 * (i + 1)

print(data)
```

```
[1.0000e-02 2.0000e-02 3.0000e-02 ... 9.9998e+02 9.9999e+02 1.0000e+03]
```

## Mean and true variance

Mean:

$\mu = \frac{1}{n} \sum_{i=1}^{n} x_i$

True Variance:

$\sigma^2 = \frac{1}{n} \sum_{i=1}^{n} (x_i - \mu)^2$

In [51]:
```
# mean
data_sum = np.sum(data)
data_mean = data_sum / 100000

print("mean:", data_mean)

# true variance
data_variance = np.sum((data - data_mean)**2) / 100000

print("true variance:", data_variance)
```

```
mean: 500.005
true variance: 83333.33332500001
```

## Computing s1_squared, s2_squared and s3_squared for a sample of 50 points with replacement

s1_squared:

$s1^2 = \frac{1}{n-1} \sum_{i=1}^{n} (x_i - \bar{x})^2$

s2_squared:

$s2^2 = \frac{1}{n} \sum_{i=1}^{n} (x_i - \bar{x})^2$

s3_squared:

$s3^2 = \frac{n}{n-1} s2^2$

In [52]:
```
def compute_sample_variance(sample):
    sample_mean = np.sum(sample) / 50

    s1_sqd = sum((xi - sample_mean) ** 2 for xi in sample) / (51)
    s2_sqd = sum((xi - sample_mean) ** 2 for xi in sample) / 50
    s3_sqd = sum((xi - sample_mean) ** 2 for xi in sample) / (49)

    return s1_sqd, s2_sqd, s3_sqd
```

Average_s1_squared:

$\frac{1}{m}\sum_{i=1}^{m} s1^2$

Average_s2_squared:

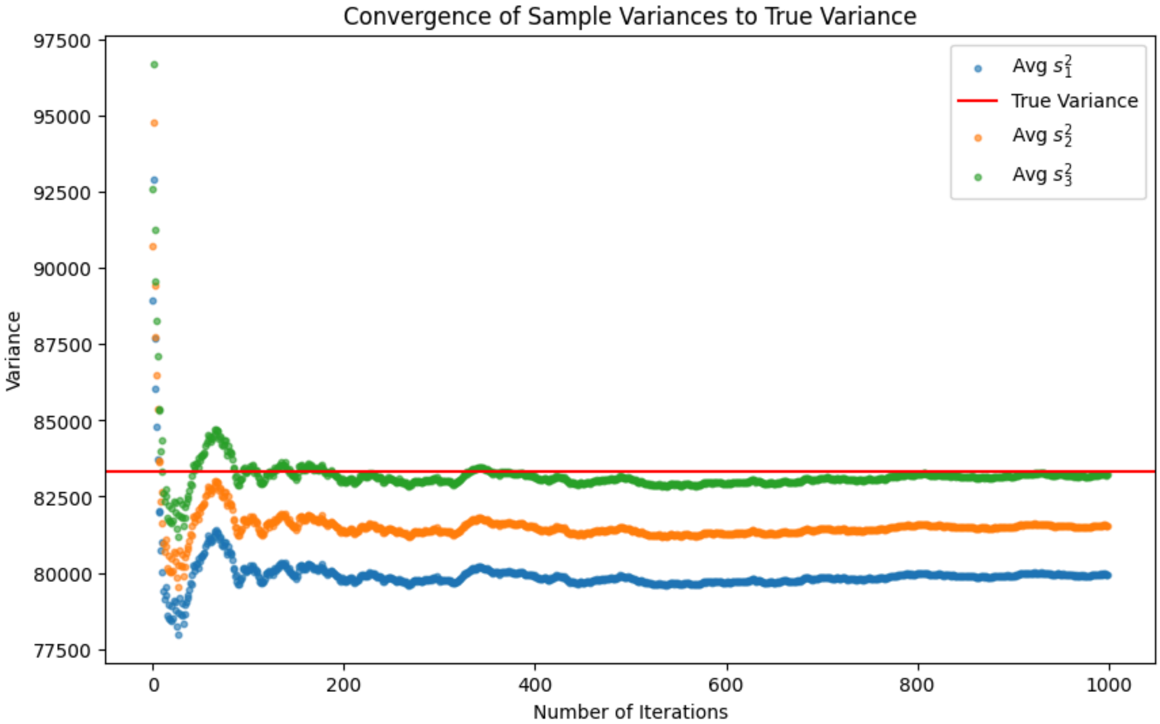
$\frac{1}{m}\sum_{i=1}^{m} s2^2$

Average_s3_squared:

$\frac{1}{m}\sum_{i=1}^{m} s3^2$

In [53]:
```
s1 = []
s2 = []
s3 = []

avg_s1 = []
avg_s2 = []
avg_s3 = []

itr = 1000

for _ in range(itr):
    sample = np.random.choice(data, 50, replace=True)
    s1_sqd, s2_sqd, s3_sqd = compute_sample_variance(sample)
    s1.append(s1_sqd)
    s2.append(s2_sqd)
    s3.append(s3_sqd)

    avg_s1.append(np.mean(s1))
    avg_s2.append(np.mean(s2))
    avg_s3.append(np.mean(s3))


# print("s1:", s1)
# print("s2:", s2)
# print("s3:", s3)

# print("avg_s1:", avg_s1)
# print("avg_s2:", avg_s2)
# print("avg_s3:", avg_s3)

plt.figure(figsize=(10, 6))

plt.scatter(range(itr), avg_s1, label = r'Avg $s_1^2$', s=10, alpha=0.6)
plt.axhline(y=data_variance, color='r', linestyle='-', label = 'True Variance')

plt.scatter(range(itr), avg_s2, label = r'Avg $s_2^2$', s=10, alpha=0.6)
# plt.axhline(y=data_variance, color='r', linestyle='-', label = 'True Variance'
```

```
plt.scatter(range(itr), avg_s3, label = r'Avg $s_3^2$', s=10, alpha=0.6)
# plt.axhline(y=data_variance, color='r', linestyle='-', label = 'True Variance'

plt.title('Convergence of Sample Variances to True Variance')
plt.xlabel('Number of Iterations')
plt.ylabel('Variance')
plt.legend()

plt.show()
```




## Inferences

We notice that r'Avg $s_3^2$' reaches the true variance more quickly and frequently compared to the rest. This is because:

- the formula for $s_1^2$ uses n + 1 in the denominator, which tends to underestimate the variance. This makes $s_1^2$ a biased estimator that is slightly biased downwards.
- the formula for $s_2^2$ uses n in the denominator, which also results in a biased estimator but less than $s_1^2$.
- the formula for $s_3^2$ uses n - 1 in the denominator, which is the unbiased sample variance estimator. This formula compensates for the fact that the sample mean (used in calculating the variance) is based on the same data, thus giving a better approximation of the population variance.

$s_3^2$ is called an unbiased estimator because it is corrected for small sample sizes by dividing by n - 1. In statistics, dividing by n - 1 is known as Bessel's correction, which accounts for the fact that the sample mean is less variable than the true population mean. This correction results in a more accurate estimate of the population variance when sampling randomly. Therefore, it tends to converge to the true variance more quickly and frequently compared to the other two estimators.

As more samples are taken, the law of large numbers ensures that all three sample variances will eventually converge to the true variance. However, for small sample sizes, $s_3^2$ is preferred due to its unbiased nature and better approximation of the population variance.

## References


- https://en.wikipedia.org/wiki/Bessel%27s_correction
- https://en.wikipedia.org/wiki/Variance#Sample_variance


In [ ]:

# DSC Assignment 1 Question-3

September 20, 2024

## 1 Part (a)

**(a)** Let the die be unbiased with $k$ faces. We want to find the expected number of rolls until the number $\lfloor \sqrt{k} \rfloor$ appears on the upward face.

Since the die is unbiased, the probability of rolling any particular number is $\frac{1}{k}$.

Let $X$ be the random variable representing the number of rolls needed to obtain $\lfloor \sqrt{k} \rfloor$ on the upward face. The number of rolls follows a geometric distribution with success probability $p = \frac{1}{k}$.

The expected value for the geometric distribution is:

$$E(X) = \frac{1}{p} = \frac{1}{\frac{1}{k}} = k$$

Thus, the expected number of rolls to see the number $\lfloor \sqrt{k} \rfloor$ is $k$.

# Part (B): Coupon Collector Problem

## General Formulation

In the general case of the **Coupon Collector Problem**, we have $K$ distinct items (coupons, grades, etc.). The goal is to collect all $K$ distinct items, and we are interested in the expected number of trials (or rolls, or papers) required to collect all of them at least once.

Let:

- $Y_i$ denote the number of trials until the $i$th new distinct item is obtained.
- $X_i = Y_{i+1} - Y_i$ represent the number of trials between obtaining the $i$th new distinct item and the $(i+1)$th new distinct item.

We are interested in calculating $\mathbb{E}[Y_K]$, the expected number of trials to collect all $K$ items.

## Expectation of $X_i$

Each $X_i$ is a geometric random variable, and the probability of success (obtaining a new distinct item) decreases as we collect more distinct items. Specifically:

$$\mathbb{P}(\text{new distinct item on the } i\text{th trial}) = \frac{K-i}{K}$$

Therefore, the expected number of trials to get the next distinct item is:

$$\mathbb{E}[X_i] = \frac{K}{K-i}$$

for $i = 0, 1, 2, \ldots, K-1$.

## Total Expected Number of Trials

To find the total expected number of trials to collect all $K$ distinct items, we sum the expectations of $X_i$ over all possible $i$. Thus, we have:

$$\mathbb{E}[Y_K] = \sum_{i=0}^{K-1} \mathbb{E}[X_i] = \sum_{i=0}^{K-1} \frac{K}{K-i}$$

This simplifies to:

$$\mathbb{E}[Y_K] = K \cdot \sum_{i=1}^{K} \frac{1}{i}$$

## Harmonic Numbers

The sum $\sum_{i=1}^{K} \frac{1}{i}$ is known as the $K$th *harmonic number*, denoted by $H_K$. Therefore, the expected number of trials to collect all $K$ distinct items can also be written as:

$$\mathbb{E}[Y_K] = K \cdot H_K$$

For large values of $K$, the harmonic number $H_K$ can be approximated as:

$$H_K \approx \ln(K) + \gamma$$

where $\gamma \approx 0.5772$ is the Euler-Mascheroni constant.

Thus, for large $K$, the expected number of trials is approximately:

$$\mathbb{E}[Y_K] \approx K \cdot (\ln(K) + \gamma)$$

## Conclusion

In the general case of the coupon collector problem, the expected number of trials to collect all $K$ distinct items grows logarithmically with $K$, following the harmonic number approximation. This result is significant in fields such as probability theory and combinatorics, where such "collection problems" frequently arise.

# Part (c)

## 1 Unequal probability while rolling dice

Let's break down the steps to calculate the expected number of rolls using inclusion-exclusion.

### 1.1 Problem Statement

We have a $k$-faced die, where each face $i$ has a probability $p_i$ of landing on that face. The goal is to compute the expected number of rolls until **all faces** have been rolled at least once.

### 1.2 Inclusion-Exclusion Formula

The inclusion-exclusion formula for this type of problem is:

$$E = \sum_i \frac{1}{p_i} - \sum_{i,j} \frac{1}{p_i + p_j} + \sum_{i,j,k} \frac{1}{p_i + p_j + p_k} - \cdots + (-1)^{n-1} \frac{1}{p_1 + p_2 + \cdots + p_n}$$

- **First term**: The expected number of rolls to see any single face (e.g., $P(i)$) is the inverse of the probability of seeing that face: $\frac{1}{p_i}$.
- **Second term**: The expected number of rolls to see two faces (e.g., face $i$ and face $j$) is $\frac{1}{p_i+p_j}$, but we need to subtract this because we've double-counted rolls that show both faces.
- **Third term**: We add back the cases where three faces have been counted together (e.g., $\frac{1}{p_i+p_j+p_k}$), and so on.
- This alternating sum accounts for all possible overlaps between the faces.

### 1.3 Probabilities for the Geometric Die

A geometric die is a die where the probabilities decrease geometrically. For example, for a 3-sided geometric die:

$$P(1) = \frac{1}{4}, \quad P(2) = \frac{1}{2}, \quad P(3) = \frac{1}{4}$$

More generally, for a $k$-sided geometric die, the probability of face $i$ is $P(i) = \frac{1}{2^{i-1}}$. So the probability for the first face is $\frac{1}{1}$, the second face is $\frac{1}{2}$, the third is $\frac{1}{4}$, and so on.

### 1.4 Apply the Formula

For a 3-sided die with probabilities $\frac{1}{4}, \frac{1}{2}, \frac{1}{4}$, the expected number of rolls $E$ is computed as:

$$E = \left(\frac{1}{P(1)} + \frac{1}{P(2)} + \frac{1}{P(3)}\right) - \left(\frac{1}{P(1)+P(2)} + \frac{1}{P(1)+P(3)} + \frac{1}{P(2)+P(3)}\right) + \frac{1}{P(1)+P(2)+P(3)}$$

Genefralized formula:

$$\mathbb{E} = \sum_{\mu \neq 0} \frac{(-1)^{|\mu|-1}}{\mu \cdot \mathbf{p}},$$

Substituting the values:

$$E = (4 + 2 + 4) - \left(\frac{4}{3} + 2 + \frac{4}{3}\right) + 1$$

Simplifying:

$$E = 10 - 4.67 + 1 = 6.33$$

So the expected number of rolls to see all three faces at least once is approximately 6.33 rolls.

In [3]:
```
import numpy as np
import matplotlib.pyplot as plt
from itertools import combinations
```

## Inclusion-Exclusion Principle

$$E[\text{rolls}] = \sum_{r=1}^{n} (-1)^{r-1} \sum_{S \subseteq \text{probs}, |S|=r} \frac{1}{\sum_{i \in S} p_i}$$

In [4]:
```
def expected_rolls(probs):
    n = len(probs)
    expected_value = 0

    for r in range(1, n+1):
        for subset in combinations(probs, r):
            prob_sum = sum(subset)
            expected_value += (-1)**(r - 1) * 1 / prob_sum

    return expected_value
```

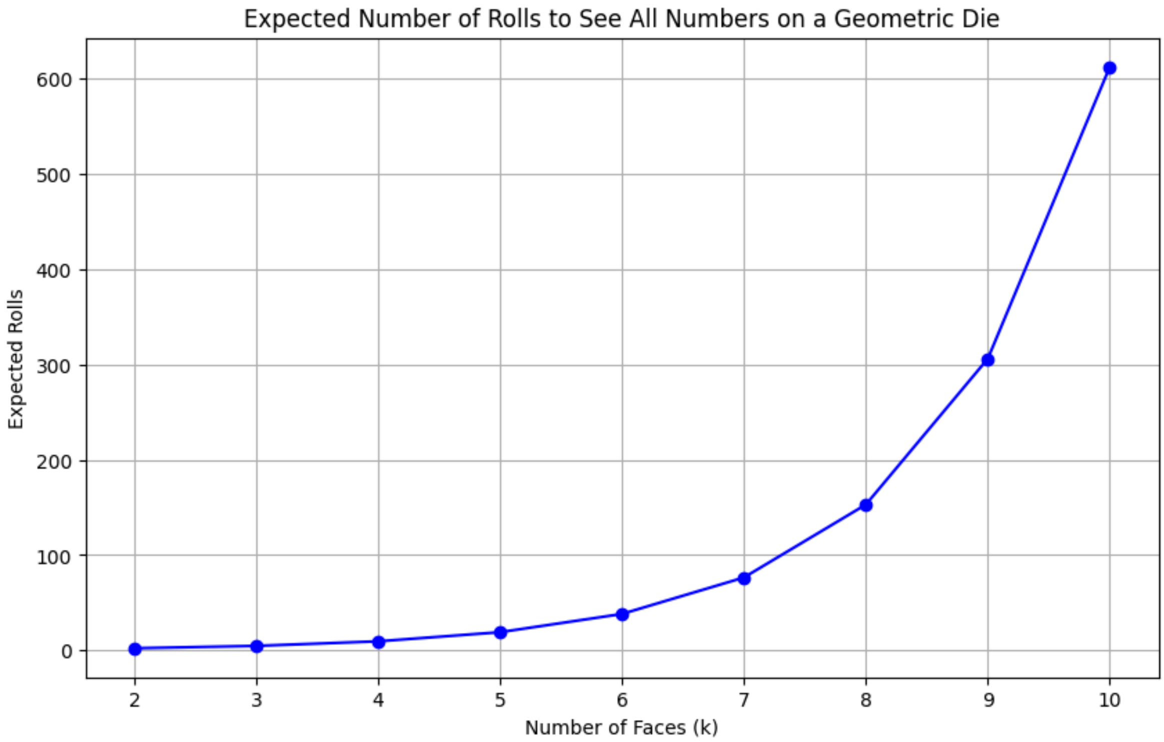
## Calculating probabilities for a k-faced geometric die

$$P(i) = \frac{1}{2^{(i-1)}} \quad \text{for } i = 1, 2, \ldots, k$$

In [5]:
```
def geometric_probabilities(k):
    return [1/(2**(i-1)) for i in range(1, k+1)]
```

## Plotting the expected number of rolls for different values of k

In [6]:
```
ks = range(2, 11)
expected_values = []

for k in ks:
    probabilities = geometric_probabilities(k)
    e_value = expected_rolls(probabilities)
    expected_values.append(e_value)

plt.figure(figsize=(10, 6))
plt.plot(ks, expected_values, marker='o', linestyle='-', color='b')
plt.title('Expected Number of Rolls to See All Numbers on a Geometric Die')
plt.xlabel('Number of Faces (k)')
plt.ylabel('Expected Rolls')
plt.grid(True)
plt.show()
```




The code simulates the expected number of rolls needed to observe all faces of a geometric die, where each face has a different probability of showing up. Specifically, the die has ( k ) faces, and the probability of each face being rolled follows a geometric distribution where the probability of rolling face ( i ) is

$$\frac{1}{2^{i-1}}$$

## Inference from the Plot:

1. Increasing ( k ) Increases Expected Rolls:
   - As ( k ) (the number of faces of the die) increases, the expected number of rolls to see all faces at least once also increases because with more faces, it becomes harder to roll each unique number due to the geometric nature of the probabilities.
2. Geometric Distribution Effect:
   - The probabilities of rolling each face decrease exponentially as ( i ) increases (the first face has a probability of ( 1 ), the second face has a probability of ( \frac{1}{2} ), the third face has a probability of ( \frac{1}{4} )...). This means that the later faces become progressively harder to roll, contributing significantly to the expected number of rolls required to observe all faces.
3. Diminishing Return of Rolls:
   - The increase in expected rolls shows a slowing growth as ( k ) increases. This diminishing rate of increase can be attributed to the geometric probabilities: while adding more faces makes it harder to observe all of them, the difference between adding, say, the 9th and 10th faces (which have very low probabilities) is less significant than adding earlier faces (which have higher probabilities).

4. Inclusion-Exclusion Principle:

   - The code uses the inclusion-exclusion principle to calculate the expected number of rolls. This accounts for overlapping probabilities of combinations of faces, providing a more accurate estimate than a simple sum of probabilities would.

# Overall, difficulty of observing all numbers on a geometric die increases as the number of faces increases, with a slowing growth due to the nature of the geometric distribution.

## References


- https://www.jstor.org/stable/40378689?seq=3
- https://www.jstor.org/stable/40378689?seq=2
- https://www.youtube.com/watch?v=3mu47FWEuqA
- https://math.stackexchange.com/questions/600012/coupon-collectors-problem-with-unequal-probabilities

In [135…
```
import pandas as pd
import math
from scipy import stats
import matplotlib.pyplot as plt
```

## Loading the dataset

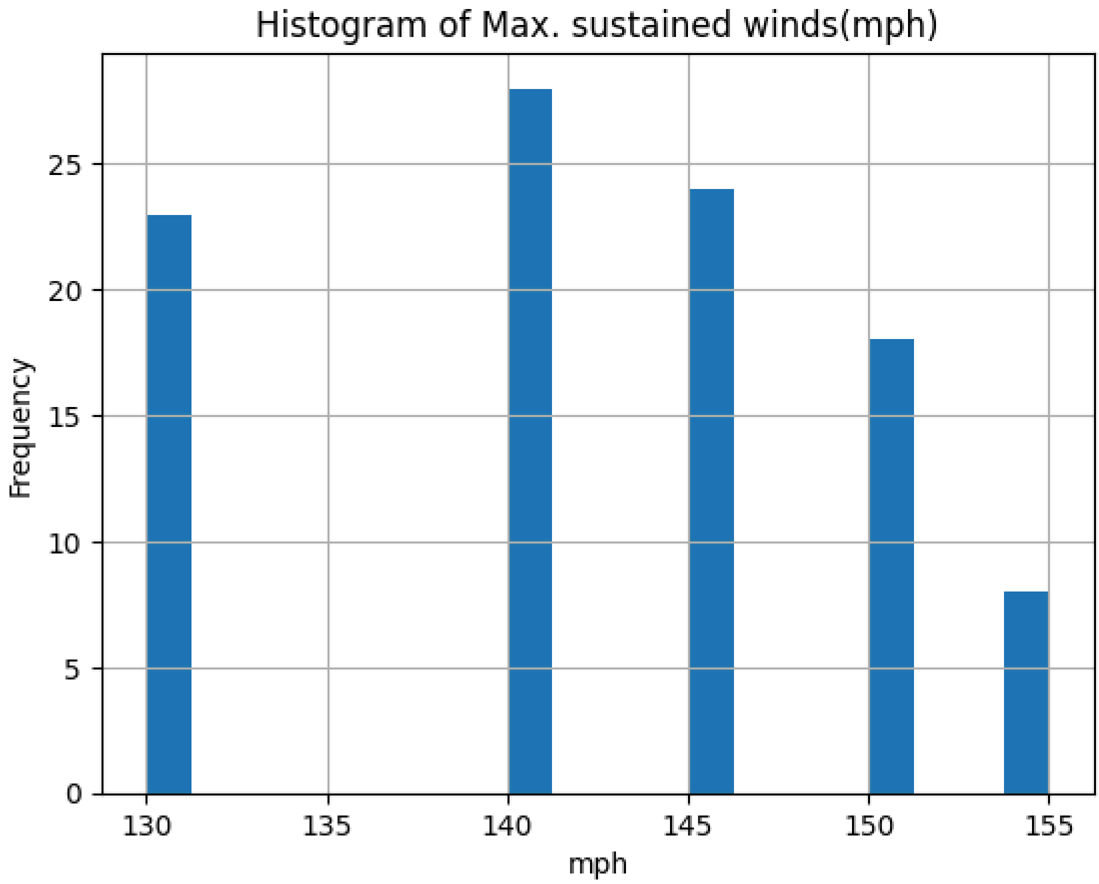
In [135…
```
data = pd.read_csv('Hurricane.csv')
```

In [135…
```
print(data.head())
```

```
                            Name  Season                Month  \
0                  Hurricane #3    1853   August, September
1  "1856 Last Island Hurricane"    1856               August
2                  Hurricane #6    1866  September, October
3                  Hurricane #7    1878  September, October
4                  Hurricane #2    1880               August

   Max. sustained winds(mph)  Minimum pressure(mbar)
0                        150                     924
1                        150                     934
2                        140                     938
3                        140                     938
4                        150                     931
```

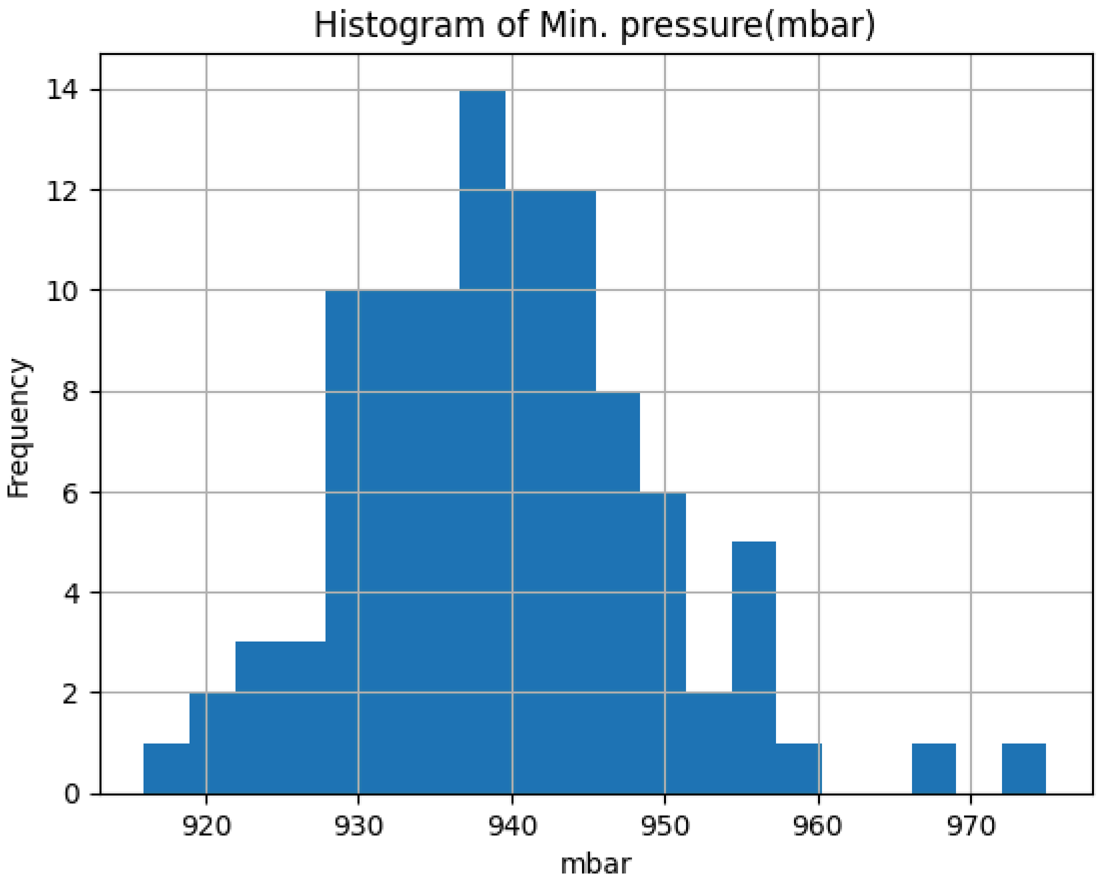
In [135…
```
data.columns = ['Name', 'Season', 'Month', 'mph', 'mbar']
```

In [135…
```
data['mph'].hist(bins=20)
plt.xlabel('mph')
plt.ylabel('Frequency')
plt.title('Histogram of Max. sustained winds(mph)')
plt.show()
```




In [136…

```
data['mbar'].hist(bins=20)
plt.xlabel('mbar')
plt.ylabel('Frequency')
plt.title('Histogram of Min. pressure(mbar)')
plt.show()
```




## Preprocessing

In [136…

```
# cleaning Month column
data['Month'] = data['Month'].str.strip()
data['Month'] = data['Month'].str.replace(',', ' ')
data['Month'] = data['Month'].str.replace('-', ' ')
data['Month'] = data['Month'].str.replace('Aug', 'August')
data['Month'] = data['Month'].str.replace(r'\d+', '', regex=True)
data['Month'] = data['Month'].str.replace('Augustust', 'August')

data['Month']
```

Out[136…

```
0        August  September
1                   August
2       September  October
3       September  October
4                   August
              ...
96                 October
97      August  September
98                 August
99      August  September
100              September
Name: Month, Length: 101, dtype: object
```

In [136…

```
# splitting one row into multiple rows if it contains multiple months
data['Month'] = data['Month'].str.split()
data = data.explode('Month')

data
```

Out[136…

| | Name | Season | Month | mph | mbar |
|---|---|---|---|---|---|
| **0** | Hurricane #3 | 1853 | August | 150 | 924 |
| **0** | Hurricane #3 | 1853 | September | 150 | 924 |
| **1** | "1856 Last Island Hurricane" | 1856 | August | 150 | 934 |
| **2** | Hurricane #6 | 1866 | September | 140 | 938 |
| **2** | Hurricane #6 | 1866 | October | 140 | 938 |
| **...** | ... | ... | ... | ... | ... |
| **97** | Hurricane Fabian | 2003 | September | 145 | 939 |
| **98** | Hurricane Charley | 2004 | August | 150 | 941 |
| **99** | Hurricane Frances | 2004 | August | 145 | 935 |
| **99** | Hurricane Frances | 2004 | September | 145 | 935 |
| **100** | Hurricane Karl | 2004 | September | 145 | 938 |

137 rows × 5 columns

# (a) With a 1% level of significance conduct t-test for correlation coefficient between “Max. sustained winds(mph)” and “Minimum pressure(mbar)”.

## Preprocessing the data

In [136…

```
# min max scaling numerical data
# mph_org = data['mph']
# mbar_org = data['mbar']
# data['mph'] = (data['mph'] - data['mph'].min()) / (data['mph'].max() - data['m
# data['mbar'] = (data['mbar'] - data['mbar'].min()) / (data['mbar'].max() - dat
```

### t-test

Null Hypothesis: There is no correlation between “Max. sustained winds(mph)” and “Minimum pressure(mbar)”
Alternate Hypothesis: There is a correlation between “Max. sustained winds(mph)” and “Minimum pressure(mbar)”

Method:\

Covariance:\

$$Cov(X,Y) = \frac{\sum_{i=1}^{n}(X_i - \bar{X})(Y_i - \bar{Y})}{n}$$

Standard Deviation:\

$$\sigma_X = \sqrt{\frac{\sum_{i=1}^{n}(X_i - \bar{X})^2}{n}}$$

Correlation Coefficient:\

$$r = \frac{Cov(X,Y)}{\sigma_X \sigma_Y}$$

t-test:\

$$t = \frac{r\sqrt{n-2}}{\sqrt{1-r^2}}$$

where n is the number of samples

In [136…

```
# Calculating mean
mean_mph = data['mph'].mean()
mean_mbar = data['mbar'].mean()

print(f"Mean of Max Sustained Winds: {mean_mph}")
print(f"Mean of Minimum Pressure: {mean_mbar}")

# Calculating covariance
covariance = sum((data['mph'] - mean_mph) * (data['mbar'] - mean_mbar)) / (len(d

print(f"Covariance: {covariance}")

# Calculating standard deviation
std_dev_mph = math.sqrt(sum((data['mph'] - mean_mph) ** 2) / (len(data) - 1))
std_dev_mbar = math.sqrt(sum((data['mbar'] - mean_mbar) ** 2) / (len(data) - 1))

print(f"Standard Deviation of Max Sustained Winds: {std_dev_mph}")
print(f"Standard Deviation of Minimum Pressure: {std_dev_mbar}")

# Calculating correlation
correlation = covariance / (std_dev_mph * std_dev_mbar)

print(f"Correlation: {correlation}")

# t-test
n = len(data)
t = correlation * math.sqrt(n - 2) / math.sqrt(1 - correlation ** 2)

print(f"t-test: {t}")

# range: mean - t * std_dev, mean + t * std_dev
t_value = stats.t.ppf(0.995, n-2)

print(f"t-value: {t_value}")

print("------------------------------------------------------------------------

if abs(t) > t_value:
    print("Reject the null hypothesis; there is a significant correlation betwee
else:
    print("Accept the null hypothesis; there is no significant correlation betwe

print("------------------------------------------------------------------------

# p-value
p = 2 * (1 - stats.t.cdf(abs(t), df=n-2))
```

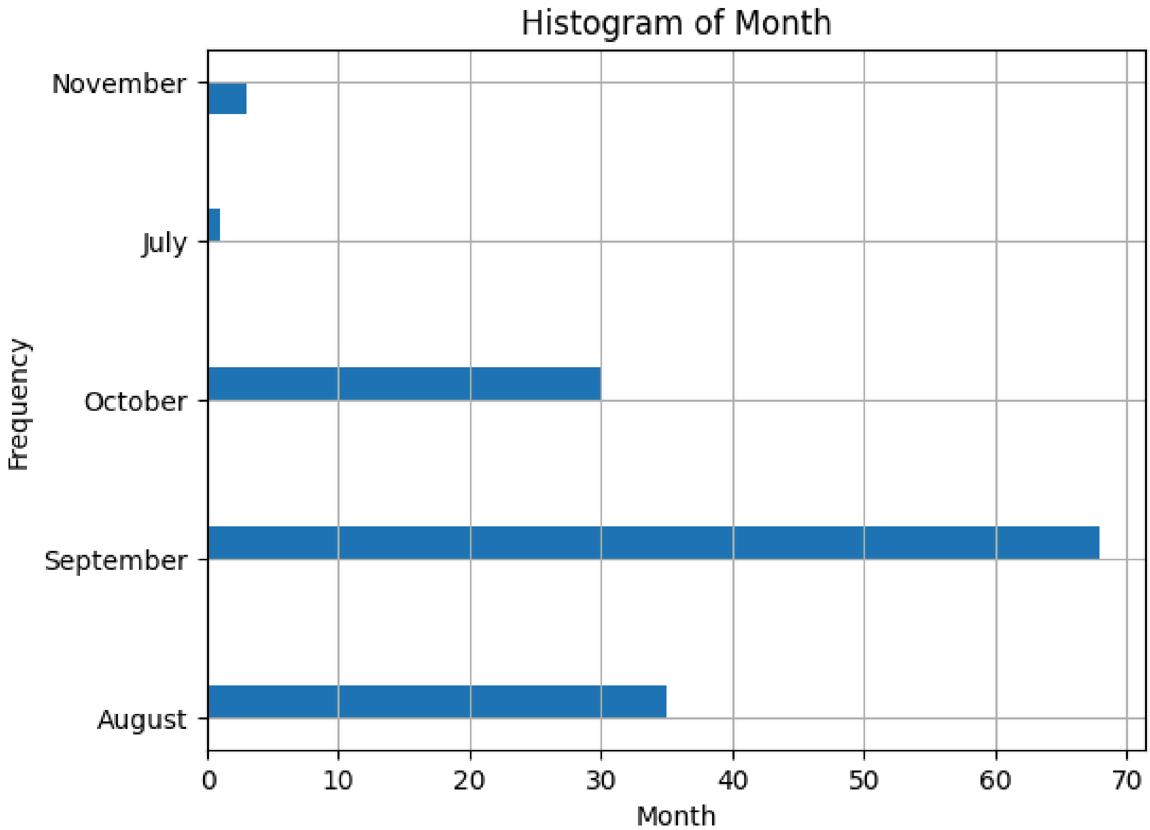
```
print("Degree of Freedom:", n-2)

print(f"p-value: {p}")

print("----------------------------------------------------------------------

# Conclusion
if p < 0.01:
    print("Reject the null hypothesis; there is a significant correlation betwee
else:
    print("Accept the null hypothesis; there is no significant correlation betwe
```

```
Mean of Max Sustained Winds: 142.33576642335765
Mean of Minimum Pressure: 938.8029197080292
Covariance: -35.89657578359814
Standard Deviation of Max Sustained Winds: 7.766138199470568
Standard Deviation of Minimum Pressure: 9.985165037413662
Correlation: -0.46290584088602793
t-test: -6.067728544364732
t-value: 2.612737907693308
-----------------------------------------------------------------------------------
---------------------------
Reject the null hypothesis; there is a significant correlation between Max Sustai
ned Winds and Minimum Pressure
-----------------------------------------------------------------------------------
---------------------------
Degree of Freedom: 135
p-value: 1.2303904561861145e-08
-----------------------------------------------------------------------------------
---------------------------
Reject the null hypothesis; there is a significant correlation between Max Sustai
ned Winds and Minimum Pressure
```

## (b) With a 5% level of significance test if the “Max. sustained winds(mph)” of hurricane depends on the month of its occurrence.

In [136…

```
data['Month'].hist(bins=20, orientation='horizontal')
plt.xlabel('Month')
plt.ylabel('Frequency')
plt.title('Histogram of Month')
plt.show()
```

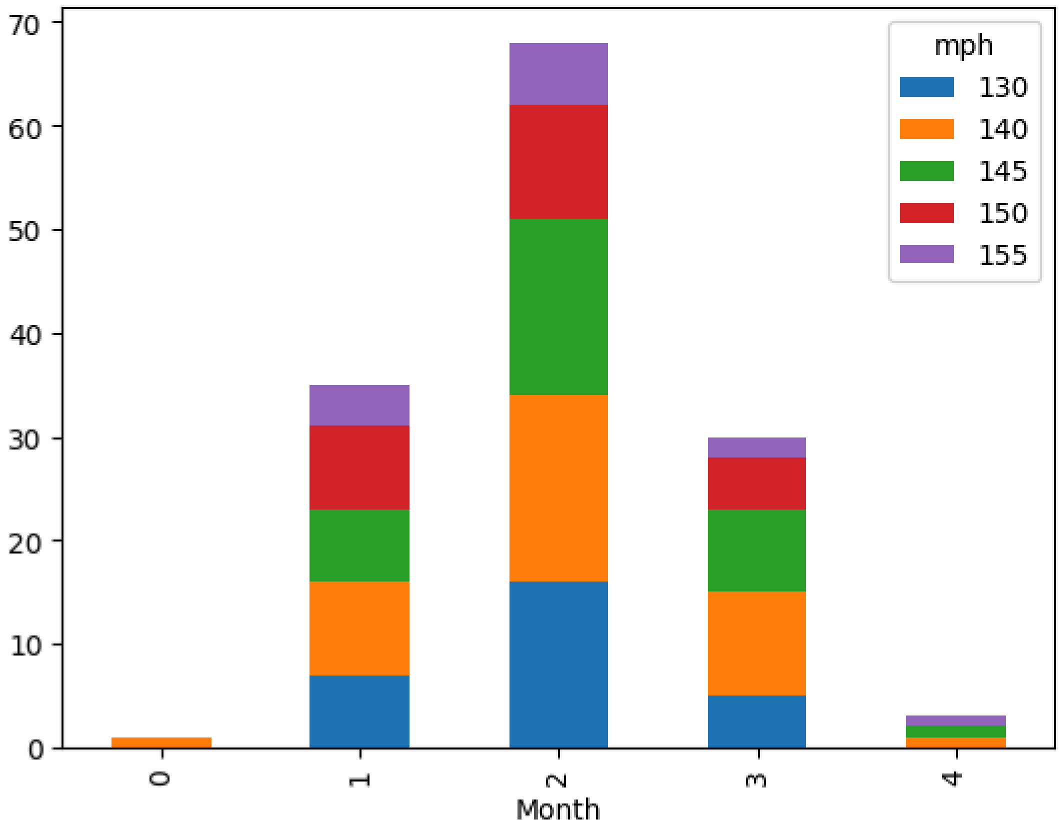



In [138…

```
contingency_table = pd.crosstab(data['Month'], data['mph'])
contingency_table.plot(kind='bar', stacked=True)
contingency_table
```

Out[138…

| mph | 130 | 140 | 145 | 150 | 155 |
|---|---|---|---|---|---|
| **Month** | | | | | |
| **0** | 0 | 1 | 0 | 0 | 0 |
| **1** | 7 | 9 | 7 | 8 | 4 |
| **2** | 16 | 18 | 17 | 11 | 6 |
| **3** | 5 | 10 | 8 | 5 | 2 |
| **4** | 0 | 1 | 1 | 0 | 1 |




### Chi Square test

Null Hypothesis: The "Max. sustained winds(mph)" of hurricane does not depend on the month of its occurrence
Alternate Hypothesis: The "Max. sustained winds(mph)" of hurricane depends on the month of its occurrence

Method:\

Chi Square test:\

$$\chi^2 = \sum_{i=1}^{n} \frac{(O_i - E_i)^2}{E_i}$$

where n is the number of categories

In [136...

```
month_total = contingency_table.sum(axis=1)
mph_total = contingency_table.sum(axis=0)
contingency_total = contingency_table.values.sum()

print("month_total:", month_total)
print("mph_total:", mph_total)
print("contingency_total:", contingency_total)
```

```
month_total: Month
August       35
July          1
November      3
October      30
September    68
dtype: int64
mph_total: mph
130    28
140    39
145    33
150    24
155    13
dtype: int64
contingency_total: 137
```

In [136…

```
expected_frequency = pd.DataFrame(index=contingency_table.index, columns=conting

for i in contingency_table.index:
    for j in contingency_table.columns:
        expected_frequency.at[i, j] = (month_total[i] * mph_total[j]) / continge

print("Expected Frequency:")
print(expected_frequency)
```

```
Expected Frequency:
mph              130        140        145        150       155
Month
August      7.153285   9.963504   8.430657   6.131387  3.321168
July         0.20438   0.284672   0.240876   0.175182  0.094891
November    0.613139   0.854015   0.722628   0.525547  0.284672
October     6.131387   8.540146   7.226277   5.255474  2.846715
September   13.89781  19.357664  16.379562  11.912409  6.452555
```

In [136…

```
# Chi-square test
chi_square = 0

for i in contingency_table.index:
    for j in contingency_table.columns:
        observed = contingency_table.at[i, j]
        expected = expected_frequency.at[i, j]
        chi_square += (observed - expected) ** 2 / expected

print(f"Chi-square: {chi_square}")
```

```
Chi-square: 7.971634778620072
```

In [137…

```
rows, cols = contingency_table.shape
df = (rows - 1) * (cols - 1)

print(f"Degrees of Freedom: {df}")
```

```
Degrees of Freedom: 16
```

In [137…

```
chi_square_critical = stats.chi2.ppf(1 - 0.025, df)
chi_square_critical_lower = stats.chi2.ppf(0.025, df)

print(f"Chi-square Critical: {chi_square_critical}")
print(f"Chi-square Critical Lower: {chi_square_critical_lower}")

if chi_square > chi_square_critical or chi_square < chi_square_critical_lower:
```

```
    print("Reject the null hypothesis; there is a significant relationship betwe
else:
    print("Accept the null hypothesis; there is no significant relationship betw
```

```
Chi-square Critical: 28.845350723404753
Chi-square Critical Lower: 6.907664353497004
Accept the null hypothesis; there is no significant relationship between Month an
d Max Sustained Winds
```

In [137…

```
# p-value
p = 1 - stats.chi2.cdf(chi_square, df=df)

print(f"p-value: {p}")

# Conclusion
if p < 0.05:
    print("Reject the null hypothesis; there is a significant association betwee
else:
    print("Accept the null hypothesis; there is no significant association betwe
```

```
p-value: 0.9497063344388978
Accept the null hypothesis; there is no significant association between Month and
Max Sustained Winds
```

## Further processing of 'Month' column

In [137…

```
# converting months into relative ordinal values
data['Month'] = pd.Categorical(data['Month'], ordered=True, categories=['July',
data['Month'] = data['Month'].cat.codes

print(data['Month'])
```

```
0      1
0      2
1      1
2      2
2      3
      ..
97     2
98     1
99     1
99     2
100    2
Name: Month, Length: 137, dtype: int8
```

### t-test

Null Hypothesis: There is no correlation between “Max. sustained winds(mph)” and “Month”

Alternate Hypothesis: There is a correlation between “Max. sustained winds(mph)” and “Month”

In [137…

```
# t-test
# calculating mean
mean_month = data['Month'].mean()

print(f"Mean of Month: {mean_month}")

# calculating covariance
```

```
covariance = sum((data['Month'] - mean_month) * (data['mph'] - mean_mph)) / (len

print(f"Covariance: {covariance}")

# calculating standard deviation
std_dev_month = math.sqrt(sum((data['Month'] - mean_month) ** 2) / (len(data) -

print(f"Standard Deviation of Month: {std_dev_month}")

# calculating correlation
correlation = covariance / (std_dev_month * std_dev_mph)

print(f"Correlation: {correlation}")

# t-test
n = len(data)
t = correlation * math.sqrt(n - 2) / math.sqrt(1 - correlation ** 2)

print(f"t-test: {t}")

# range: mean - t * std_dev, mean + t * std_dev
t_value = stats.t.ppf(0.975, n-2)

print(f"t-value: {t_value}")

if abs(t) > t_value:
    print("Reject the null hypothesis; there is a significant correlation betwee
else:
    print("Accept the null hypothesis; there is no significant correlation betwe

# p-value
p = 2 * (1 - stats.t.cdf(abs(t), df=n-2))

print("Degree of Freedom:", n-2)

print(f"p-value: {p}")

# Conclusion
if p < 0.05:
    print("Reject the null hypothesis; there is a significant correlation betwee
else:
    print("Accept the null hypothesis; there is no significant correlation betwe
```

```
Mean of Month: 1.9927007299270072
Covariance: 0.05393945899527636
Standard Deviation of Month: 0.7717088597331507
Correlation: 0.009000113479535317
t-test: 0.1045761043854583
t-value: 1.977692277222804
Accept the null hypothesis; there is no significant correlation between Month and
Max Sustained Winds
Degree of Freedom: 135
p-value: 0.9168673863071735
Accept the null hypothesis; there is no significant correlation between Month and
Max Sustained Winds
Degree of Freedom: 135
p-value: 0.9168673863071735
Accept the null hypothesis; there is no significant correlation between Month and
Max Sustained Winds
```

# With a 10% level of significance conduct test if “Max. sustained winds(mph)” follows a Poisson distribution.

Poission distribution is given by:

$$P(X = k) = \frac{e^{-\lambda}\lambda^k}{k!}$$

where k is the number of occurrences, and $\lambda$ is the average number of occurrences

## Expected and Observed frequencies

In [137…

```
# scaling_factor = 2

# # scaling mph
# data['mph'] = data['mph'] // scaling_factor
```

In [137…

```
# scaling_factor = 2

# data['mph'] = data['mph'] * scaling_factor
# data['mph'] = data['mph'].astype(int)
# data['mph']
```

In [137…

```
#min max scaling month
# data['Month'] = (data['Month'] - data['Month'].min()) / (data['Month'].max() -
```

In [137…

```
mph_count = data['mph'].value_counts()

print(mph_count)
```

```
mph
140    39
145    33
130    28
150    24
155    13
Name: count, dtype: int64
```

In [137…

```
# # how many times each wind occurred
# mph_count = mph_org.value_counts()

# print("mph_count:")
# print(mph_count)
```

In [138…

```
mean_mph_scaled = data['mph'].mean()
```

In [138…

```
# expected frequencies using Poisson distribution
expected_frequency = {}
n = len(data)

# print("Total Observations:", n)

for i in mph_count.index:
    poisson_prob = stats.poisson.pmf(i, mean_mph_scaled)
    # poisson_prob = (mean_mph_scaled ** i) * math.exp(-mean_mph_scaled) / math.
    expected_frequency[i] = poisson_prob * n + 1e-15
```

```
print("Expected Frequency:")
print(expected_frequency)
```

```
Expected Frequency:
{140: 4.52833796439542, 145: 4.425222015865378, 130: 2.7619462507085055, 150: 3.6
416302840133272, 155: 2.5380588506680306}
```

### Chi Square test

Null Hypothesis: "Max. sustained winds(mph)" follows a Poisson distribution

Alternate Hypothesis: "Max. sustained winds(mph)" does not follow a Poisson distribution

In [138…

```
chi_square = 0

for i in mph_count.index:
    observed = mph_count[i]
    expected = expected_frequency[i]
    chi_square += (observed - expected) ** 2 / expected

print(f"Chi-square: {chi_square}")
```

```
Chi-square: 834.484409173985
```

In [138…

```
df = len(mph_count) - 2

print(f"Degrees of Freedom: {df}")
```

```
Degrees of Freedom: 3
```

In [138…

```
chi_square_critical = stats.chi2.ppf(0.90, df)

print(f"Chi-square Critical: {chi_square_critical}")

if abs(chi_square) > chi_square_critical:
    print("Reject the null hypothesis; the distribution of Max Sustained Winds i
else:
    print("Accept the null hypothesis; the distribution of Max Sustained Winds i
```

```
Chi-square Critical: 6.251388631170325
Reject the null hypothesis; the distribution of Max Sustained Winds is not Poisso
n
```

## References


- https://www.medcalc.org/manual/statistical-tables.php
- https://www.statology.org/t-test-for-correlation/
- https://stats.libretexts.org/Bookshelves/Introductory_Statistics/Introductory_Statistics_1e


### Note:

High Level discussions conducted with the group consisting of: Saksham Singh and Sidhartha Garg

In [ ]: