Foundations of Data Science

Name: Krishna GSVV
Roll no. AV.EN.U4CSE22016

Lab 1 - (Basic problems - list, dictionary, tuple, set)

In [2]:
```
my_set = {1, 2, 3}
my_set.add(4)
print("Set after adding element:", my_set)
```

```
Set after adding element: {1, 2, 3, 4}
```

This code creates a set `my_set` containing the elements `{1, 2, 3}`, adds the element `4` to the set using the `add()` method, and then prints the updated set.

In [10]:
```
my_list = [10, 20, 30, 40]
length = len(my_list)
print("Length of the list:", length)
```

```
Length of the list: 4
```

This code initializes a list `my_list` with elements `[10, 20, 30, 40]`, calculates its length using the `len()` function, and prints the length of the list.

In [11]:
```
my_tuple = (5, 10, 15, 20)
element = my_tuple[2]
print("Element at index 2:", element)
```

```
Element at index 2: 15
```

This code creates a tuple `my_tuple` with elements `(5, 10, 15, 20)`, retrieves the element at index `2` (which is `15`), and prints it.

In [12]:
```
my_dict = {'name': 'Alice', 'age': 25, 'city': 'New York'}
key = 'age'
if key in my_dict:
    print(f"'{key}' exists in the dictionary.")
else:
    print(f"'{key}' does not exist in the dictionary.")
```

```
'age' exists in the dictionary.
```

This code creates a dictionary `my_dict` with keys `'name'`, `'age'`, and `'city'`. It checks if the key `'age'` exists in the dictionary and prints a message indicating whether the key is present or not.

In [13]:
```
my_set = {1, 2, 3, 4}
my_set.remove(3)
print("Set after removing element:", my_set)
```

```
Set after removing element: {1, 2, 4}
```

This code creates a set `my_set` with elements `{1, 2, 3, 4}`, removes the element `3` using the `remove()` method, and then prints the updated set.

In [14]:
```
my_list = [1, 2, 2, 3, 3, 3, 4]
count = my_list.count(3)
print("Occurrences of 3:", count)
```

```
Occurrences of 3: 3
```

This code initializes a list `my_list` with elements `[1, 2, 2, 3, 3, 3, 4]`, counts the occurrences of the element `3` using the `count()` method, and then prints the result (which is `3`).

In [15]:
```
tuple1 = (1, 2, 3)
tuple2 = (4, 5, 6)
concatenated_tuple = tuple1 + tuple2
print("Concatenated tuple:", concatenated_tuple)
```

```
Concatenated tuple: (1, 2, 3, 4, 5, 6)
```

This code creates two tuples `tuple1` and `tuple2`, concatenates them using the `+` operator, and then prints the resulting concatenated tuple `(1, 2, 3, 4, 5, 6)`.

In [1]:
```
list1 = [1, 2, 3, 4, 5]
list2 = [4, 5, 6, 7, 8]
common_elements = set(list1).intersection(set(list2))
print("Common elements:", common_elements)
```

```
Common elements: {4, 5}
```

This code creates two lists, `list1` and `list2`, converts them to sets, finds the common elements between the two sets using the `intersection()` method, and then prints the common elements `{4, 5}`.

In [2]:
```
numbers = [1, 2, 2, 3, 4, 4, 5]
unique_numbers = list(set(numbers))
print("Unique numbers:", unique_numbers)
```

```
Unique numbers: [1, 2, 3, 4, 5]
```

This code converts the `numbers` list to a set (removing duplicates), then converts it back to a list to get the unique numbers, and prints the result `[1, 2, 3, 4, 5]`.

In [3]:
```
items = ['apple', 'banana', 'apple', 'orange', 'banana', 'apple']
frequency = {}
for item in items:
    frequency[item] = frequency.get(item, 0) + 1
print("Frequency of elements:", frequency)
```

```
Frequency of elements: {'apple': 3, 'banana': 2, 'orange': 1}
```

This code iterates over the `items` list, counts the occurrences of each element using a dictionary `frequency`, and then prints the frequency of each element: `{'apple': 3, 'banana': 2, 'orange': 1}`.

In [6]:
```
set1 = {1, 2, 3, 4}
set2 = {3, 4, 5, 6}
sym_diff = set1.symmetric_difference(set2)
print("Symmetric difference:", sym_diff)
```

```
Symmetric difference: {1, 2, 5, 6}
```

This code creates two sets, `set1` and `set2`, and finds their symmetric difference (elements that are in either of the sets, but not in both) using the `symmetric_difference()` method. It then prints the result `{1, 2, 5, 6}`.

In [7]:
```
list_of_tuples = [('a', 1), ('b', 2), ('c', 3)]
dictionary = dict(list_of_tuples)
print("Dictionary:", dictionary)
```

```
Dictionary: {'a': 1, 'b': 2, 'c': 3}
```

This code converts a list of tuples `list_of_tuples` into a dictionary using the `dict()` function, where each tuple's first element becomes the key and the second element becomes the value. It then prints the resulting dictionary `{'a': 1, 'b': 2, 'c': 3}`.

In [8]:
```
dict1 = {'a': 1, 'b': 2}
dict2 = {'b': 3, 'c': 4}
merged_dict = {**dict1, **dict2}
print("Merged dictionary:", merged_dict)
```

```
Merged dictionary: {'a': 1, 'b': 3, 'c': 4}
```

This code merges `dict1` and `dict2` using the unpacking operator `**`, with values from `dict2` overwriting any matching keys from `dict1`. It then prints the resulting merged dictionary `{'a': 1, 'b': 3, 'c': 4}`.

In [16]:
```
set1 = {1, 2, 3}
set2 = {1, 2, 3, 4, 5}
is_subset = set1.issubset(set2)
print("Is set1 a subset of set2?", is_subset)
```

```
Is set1 a subset of set2? True
```

This code checks if `set1` is a subset of `set2` using the `issubset()` method. It then prints the result, which will be `True` because all elements of `set1` are present in `set2`.

In [ ]:
```

```

Foundations Of Data Science

Name: Krishna GSVV
Roll no. AV.EN.U4CSE22016

Lab 2 (Codes using Numpy [Basic])

In [2]:
```
import numpy as np
a=np.array([1,2,3])
b=np.array([4,5,6])
print(b)
```

```
[4 5 6]
```

Creating two NumPy arrays 'a' and 'b', and printing array 'b'

In [3]:
```
result = a + b
print(result)
```

```
[5 7 9]
```

Adding two arrays 'a' and 'b' element-wise and printing the result

In [4]:
```
mean = np.mean(a)
print(mean)
```

```
2.0
```

Calculating and printing the mean (average) of array 'a'

In [5]:
```
median=np.median(a)
print(median)
```

```
2.0
```

Calculating and printing the median of array 'a'

In [6]:
```
sum1=np.sum(a)
print(sum1)
```

```
6
```

Calculating and printing the sum of elements in array 'a'

In [7]:
```
diff1=np.diff(a)
print(diff1)
```

```
[1 1]
```

Calculating and printing the difference between consecutive elements of array 'a'

In [8]:
```
pro1=np.prod(a)
print(pro1)
```

```
6
```

Calculating and printing the product of all elements in array 'a'

In [9]:
```
bool=np.all(a)
print(bool)
```

```
True
```

Checking if all elements in array 'a' are non-zero (True or False)

In [10]:
```
sqrt_result = np.sqrt(a)
print(sqrt_result)
```

```
[1.         1.41421356 1.73205081]
```

Calculating and printing the square root of each element in array 'a'

In [11]:
```
count_nonzero = np.count_nonzero(a)
print(count_nonzero)
```

```
3
```

Counting and printing the number of non-zero elements in array 'a'

In [12]:
```
bool1=np.any(a)
print(bool1)
```

```
True
```

Checking if any element in array 'a' is non-zero (True or False)

In [13]:
```
variance = np.var(a)
print(variance)
```

```
0.6666666666666666
```

Calculating and printing the variance of array 'a'

In [14]:
```
std_deviation = np.std(a)
print(std_deviation)
```

```
0.816496580927726
```

Calculating and printing the standard deviation of array 'a'

In [15]:
```
sort_arr=np.cumprod(a)
print(a)
```

```
[1 2 3]
```

Calculating and printing the cumulative product of elements in array 'a'

In [16]:
```
arr = np.array([" Hello ", "WORLD", "1234", "Python3", " NumPy "])
lower_case = np.char.lower(arr)
print(lower_case)
```

```
[' hello ' 'world' '1234' 'python3' ' numpy ']
```

Converting all string elements in array 'arr' to lowercase

In [17]:
```
upper_case = np.char.upper(arr)
print(upper_case)
```

```
[' HELLO ' 'WORLD' '1234' 'PYTHON3' ' NUMPY ']
```

Converting all string elements in array 'arr' to uppercase

In [18]:
```
stripped = np.char.strip(arr)
print(stripped)
```

```
['Hello' 'WORLD' '1234' 'Python3' 'NumPy']
```

Stripping leading and trailing whitespace from each string in array 'arr'

In [19]:
```
is_alpha = np.char.isalpha(arr)
print(is_alpha)
```

```
[False  True False False False]
```

Checking if each string in array 'arr' contains only alphabetic characters

In [20]:
```
is_numeric = np.char.isnumeric(arr)
print(is_numeric)
```

```
[False False  True False False]
```

Checking if each string in array 'arr' contains only numeric characters

In [21]:
```
arr1 = np.array(["Hello World", "Python Programming", "Numpy is great", "Python is fun"])
```

In [22]:
```
count_occurrences = np.char.count(arr1, "Python")
print(count_occurrences)
```

```
[0 1 0 1]
```

Counting and printing the occurrences of the substring "Python" in each element of array 'arr1'

In [23]:
```
find_substring = np.char.find(arr1, "Python")
print(find_substring)
```

```
[-1  0 -1  0]
```

Finding and printing the index of the first occurrence of the substring "Python" in each element of array 'arr1'

In [24]:
```
rfind_substring = np.char.rfind(arr1, "Python")
print(rfind_substring)
```

```
[-1  0 -1  0]
```

Finding and printing the index of the last occurrence of the substring "Python" in each element of array 'arr1'

In [25]:
```
starts_with = np.char.startswith(arr1, "Python")
print(starts_with)
```

```
[False  True False  True]
```

Checking and printing if each element of array 'arr1' starts with the substring "Python"

In [24]:
```
from datascience import Table
from datascience import make_array
import numpy as np

table = Table().with_columns('no', make_array(8, 34, 5))

print(table)
```

```
no
8
34
5
```

In [25]:
```
Table().with_columns(
'Number of petals', make_array(8, 34, 5),
'Name', make_array('lotus', 'sunflower', 'rose')
)
```

Out[25]:

Number of petals	Name
8	lotus
34	sunflower
5	rose

Creating a Table object and adding a column 'no' with values 8, 34, and 5, and Name then printing the table

In [26]:
```
table = Table.read_table('sample.csv')
print(table)
```

```
Longitude | Latitude | City        | Direction | Survivors
32        | 54.8     | Smolensk    | Advance   | 145000
33.2      | 54.9     | Dorogobouge | Advance   | 140000
34.4      | 55.5     | Chjat       | Advance   | 127100
37.6      | 55.8     | Moscou      | Advance   | 100000
34.3      | 55.2     | Wixma       | Retreat   | 55000
32        | 54.6     | Smolensk    | Retreat   | 24000
30.4      | 54.4     | Orscha      | Retreat   | 20000
26.8      | 54.3     | Moiodexno   | Retreat   | 12000
```

Reading and printing the CSV file 'sample.csv' as a Table object

In [27]:
```
table.num_columns
```

Out[27]: 5

Getting and displaying the number of columns of the table

In [28]:
```
table.num_rows
```

Out[28]: 8

Getting and displaying the number of rows of the table

In [29]:
```
table.labels
```

Out[29]:
```
('Longitude', 'Latitude', 'City', 'Direction', 'Survivors')
```

Getting and displaying the labels of the table

In [30]:
```
table.relabeled('City', 'City Name')
```

Out[30]:

Longitude	Latitude	City Name	Direction	Survivors
32	54.8	Smolensk	Advance	145000
33.2	54.9	Dorogobouge	Advance	140000
34.4	55.5	Chjat	Advance	127100
37.6	55.8	Moscou	Advance	100000
34.3	55.2	Wixma	Retreat	55000
32	54.6	Smolensk	Retreat	24000
30.4	54.4	Orscha	Retreat	20000
26.8	54.3	Moiodexno	Retreat	12000

Renaming the column 'City' to 'City Name' in the table and printing the result

In [31]:
```
print(table)
```

```
Longitude | Latitude | City        | Direction | Survivors
32        | 54.8     | Smolensk    | Advance   | 145000
33.2      | 54.9     | Dorogobouge | Advance   | 140000
34.4      | 55.5     | Chjat       | Advance   | 127100
37.6      | 55.8     | Moscou      | Advance   | 100000
34.3      | 55.2     | Wixma       | Retreat   | 55000
32        | 54.6     | Smolensk    | Retreat   | 24000
30.4      | 54.4     | Orscha      | Retreat   | 20000
26.8      | 54.3     | Moiodexno   | Retreat   | 12000
```

printing the result

In [32]:
```
table=table.relabeled('City','City Name')
```

Reapplying the renaming of column 'City' to 'City Name' and printing the updated table

In [33]:
```
print(table)
```

```
Longitude | Latitude | City Name   | Direction | Survivors
32        | 54.8     | Smolensk    | Advance   | 145000
33.2      | 54.9     | Dorogobouge | Advance   | 140000
34.4      | 55.5     | Chjat       | Advance   | 127100
37.6      | 55.8     | Moscou      | Advance   | 100000
34.3      | 55.2     | Wixma       | Retreat   | 55000
32        | 54.6     | Smolensk    | Retreat   | 24000
30.4      | 54.4     | Orscha      | Retreat   | 20000
26.8      | 54.3     | Moiodexno   | Retreat   | 12000
```

printing the result

In [34]:
```
table.column('Survivors')
```

Out[34]:
```
array([145000, 140000, 127100, 100000,  55000,  24000,  20000,  12000])
```

Accessing and printing specific columns by name or index, and retrieving the first item of the column

In [35]:
```
table.column(4)
```

Out[35]:
```
array([145000, 140000, 127100, 100000,  55000,  24000,  20000,  12000])
```

Accessing and printing specific columns by name or index, and retrieving the first item of the column

In [36]:
```
table.column(4).item(0)
```

Out[36]:
```
145000
```

Accessing and printing specific columns by name or index, and retrieving the first item of the column

In [37]:
```
table.select('Longitude',      'Latitude')
```

Out[37]:

Longitude	Latitude
32	54.8
33.2	54.9
34.4	55.5
37.6	55.8
34.3	55.2
32	54.6
30.4	54.4
26.8	54.3

Accessing and printing specific columns by name or index, and retrieving the first item of the column

In [38]:
```
table.select(0,1)
```

Out[38]:

Longitude	Latitude
32	54.8
33.2	54.9
34.4	55.5
37.6	55.8
34.3	55.2
32	54.6
30.4	54.4
26.8	54.3

Selecting and printing the 'Longitude' and 'Latitude' columns or columns by index 0 and 1

In [39]:
```
table.drop('Longitude', 'Latitude',     'Direction')
```

Out[39]:

City Name	Survivors
Smolensk	145000
Dorogobouge	140000
Chjat	127100
Moscou	100000
Wixma	55000
Smolensk	24000
Orscha	20000
Moiodexno	12000

Dropping the specified columns ('Longitude', 'Latitude', and 'Direction') from the table and printing the result

In [40]:
```
print(table)
```

```
Longitude | Latitude | City Name    | Direction | Survivors
32        | 54.8     | Smolensk     | Advance   | 145000
33.2      | 54.9     | Dorogobouge  | Advance   | 140000
34.4      | 55.5     | Chjat        | Advance   | 127100
37.6      | 55.8     | Moscou       | Advance   | 100000
34.3      | 55.2     | Wixma        | Retreat   | 55000
32        | 54.6     | Smolensk     | Retreat   | 24000
30.4      | 54.4     | Orscha       | Retreat   | 20000
26.8      | 54.3     | Moiodexno    | Retreat   | 12000
```

printing the result

In [41]:
```
table.show(3)
```

Longitude	Latitude	City Name	Direction	Survivors
32	54.8	Smolensk	Advance	145000
33.2	54.9	Dorogobouge	Advance	140000
34.4	55.5	Chjat	Advance	127100

... (5 rows omitted)

Displaying the first 3 rows of the table

In [42]:
```
table.sort('Longitude')
```

Out[42]:

Longitude	Latitude	City Name	Direction	Survivors
26.8	54.3	Moiodexno	Retreat	12000
30.4	54.4	Orscha	Retreat	20000
32	54.8	Smolensk	Advance	145000
32	54.6	Smolensk	Retreat	24000
33.2	54.9	Dorogobouge	Advance	140000
34.3	55.2	Wixma	Retreat	55000
34.4	55.5	Chjat	Advance	127100
37.6	55.8	Moscou	Advance	100000

Sorting the table based on the 'Longitude' column

In [43]:
```
table.sort('Longitude', descending=True)
```

Out[43]:

Longitude	Latitude	City Name	Direction	Survivors
37.6	55.8	Moscou	Advance	100000
34.4	55.5	Chjat	Advance	127100
34.3	55.2	Wixma	Retreat	55000
33.2	54.9	Dorogobouge	Advance	140000
32	54.8	Smolensk	Advance	145000
32	54.6	Smolensk	Retreat	24000
30.4	54.4	Orscha	Retreat	20000
26.8	54.3	Moiodexno	Retreat	12000

Sorting the table based on the 'Longitude' column in descending order

In [44]:
```
from datascience import Table
from datascience import make_array
table = Table.read_table('sample.csv')
print(table)
```

```
Longitude | Latitude | City        | Direction | Survivors
32        | 54.8     | Smolensk    | Advance   | 145000
33.2      | 54.9     | Dorogobouge | Advance   | 140000
34.4      | 55.5     | Chjat       | Advance   | 127100
37.6      | 55.8     | Moscou      | Advance   | 100000
34.3      | 55.2     | Wixma       | Retreat   | 55000
32        | 54.6     | Smolensk    | Retreat   | 24000
30.4      | 54.4     | Orscha      | Retreat   | 20000
26.8      | 54.3     | Moiodexno   | Retreat   | 12000
```

In [45]:
```
table.take(0)
```

Out[45]:

Longitude	Latitude	City	Direction	Survivors
32	54.8	Smolensk	Advance	145000

Taking and printing the row at index 0

In [46]:
```
table.take(np.arange(3, 6))
```

Out[46]:

Longitude	Latitude	City	Direction	Survivors
37.6	55.8	Moscou	Advance	100000
34.3	55.2	Wixma	Retreat	55000
32	54.6	Smolensk	Retreat	24000

Taking and printing rows from index 3 to 5 (inclusive)

In [47]:
```
from datascience import *
table.where('Latitude', are.above(10))
```

Out[47]:

Longitude	Latitude	City	Direction	Survivors
32	54.8	Smolensk	Advance	145000
33.2	54.9	Dorogobouge	Advance	140000
34.4	55.5	Chjat	Advance	127100
37.6	55.8	Moscou	Advance	100000
34.3	55.2	Wixma	Retreat	55000
32	54.6	Smolensk	Retreat	24000
30.4	54.4	Orscha	Retreat	20000
26.8	54.3	Moiodexno	Retreat	12000

Filtering and printing rows where the 'Latitude' column has values greater than 10

In [48]:
```
table.where('Latitude',are.equal_to(54.6))
```

Out[48]:

Longitude	Latitude	City	Direction	Survivors
32	54.6	Smolensk	Retreat	24000

Filtering and printing rows where the 'Latitude' column equals 54.6

In [49]:
```
table.where('Latitude',54.6)
```

Out[49]:

Longitude	Latitude	City	Direction	Survivors
32	54.6	Smolensk	Retreat	24000

Filtering and printing rows where the 'Latitude' column equals 54.6 (direct comparison)

In [50]:
```
table.where('Latitude', are.between(54, 54.4))
```

Out[50]:

Longitude	Latitude	City	Direction	Survivors
26.8	54.3	Moiodexno	Retreat	12000

Filtering and printing rows where the 'Latitude' column has values between 54 and 54.4

In [51]:
```
table.where('Direction',        are.containing('R'))
```

Out[51]:

Longitude	Latitude	City	Direction	Survivors
34.3	55.2	Wixma	Retreat	55000
32	54.6	Smolensk	Retreat	24000
30.4	54.4	Orscha	Retreat	20000
26.8	54.3	Moiodexno	Retreat	12000

Filtering and printing rows where the 'Direction' column contains the letter 'R'

In [52]:
```
table.where('Latitude', are.above_or_equal_to(55))
```

Out[52]:

Longitude	Latitude	City	Direction	Survivors
34.4	55.5	Chjat	Advance	127100
37.6	55.8	Moscou	Advance	100000
34.3	55.2	Wixma	Retreat	55000

Filtering and printing rows where the 'Latitude' column has values greater than or equal to 55

Foundations Of Data Science

Name: Krishna GSVV
Roll no. AV.EN.U4CSE22016

Lab 3 (Pandas, Merging and Concatenating Tables)

In [4]:
```
import pandas as pd

ser = pd.Series()
print(ser)

data = ['g', 'e', 'e', 'k', 's']
ser = pd.Series(data)
print(ser)
```

```
Series([], dtype: object)
0    g
1    e
2    e
3    k
4    s
dtype: object
```

Creating a Pandas Series from a list of strings and printing it

In [5]:
```
df=pd.DataFrame()
print(df)
```

```
Empty DataFrame
Columns: []
Index: []
```

Creating an empty Pandas DataFrame and printing it

In [6]:
```
import pandas as pd
```

```
data = [['g', 1], ['e', 2], ['e', 3], ['k', 4], ['s', 5]]
df = pd.DataFrame(data, columns=['Letter', 'Number'])
print(df)
```

```
  Letter  Number
0      g       1
1      e       2
2      e       3
3      k       4
4      s       5
```

Creating a Pandas DataFrame from a list of lists, with specified column names, and printing it

In [7]:
```
s1=['amrita','vishwa','vidya','peetham']
df=pd.DataFrame(s1)
print(df)
```

```
         0
0   amrita
1   vishwa
2    vidya
3  peetham
```

Creating a Pandas DataFrame from a list of strings and printing it

In [8]:
```
data={'Name':['Tom','nick','krish'],'Age':[21,19,10]}
df=pd.DataFrame(data)
print(df)
```

```
    Name  Age
0    Tom   21
1   nick   19
2  krish   10
```

Creating a Pandas DataFrame from a dictionary with 'Name' and 'Age' keys, and printing it

In [12]:
```
data={'Name':['Tom','nick','krish'],'Age':[21,19,10]}
df=pd.DataFrame(data)
print(df)
```

```
    Name  Age
0    Tom   21
1   nick   19
2  krish   10
```

Repeating the same DataFrame creation and printing it

In [15]:
```
data={'Name':['jai','princi','Gaurav','Anuj'],'Age':[27,24,22,32],'Address':['Delhi','Kanpur'
df=pd.DataFrame(data)
print(df)
```

```
     Name  Age    Address Qualification
0     jai   27      Delhi           MSC
1  princi   24     Kanpur            MA
2  Gaurav   22  Allahabad           MCA
3    Anuj   32    Kannauj           phd
```

Creating a Pandas DataFrame with multiple columns and printing it

In [22]:
```
print(df[['Name','Age']])
```

```
     Name  Age
0     jai   27
1  princi   24
2  Gaurav   22
3    Anuj   32
```

Selecting and printing specific columns ('Name' and 'Age') from the DataFrame

In [32]:
```
df = pd.read_csv('nba.CSV',index_col="Name")
print(df)
```

```
                        Team  Number Position   Age Height  Weight  \
Name
Avery Bradley  Boston Celtics     0.0       PG  25.0    6-2   180.0
Jae Crowder    Boston Celtics    99.0       SF  25.0    6-6   235.0
John Holland   Boston Celtics    30.0       SG  27.0    6-5   205.0
R.J. Hunter    Boston Celtics    28.0       SG  22.0    6-5   185.0
Jonas Jerebko  Boston Celtics     8.0       PF  29.0   6-10   231.0
...                      ...     ...      ...   ...    ...     ...
Shelvin Mack        Utah Jazz     8.0       PG  26.0    6-3   203.0
Raul Neto           Utah Jazz    25.0       PG  24.0    6-1   179.0
Tibor Pleiss        Utah Jazz    21.0        C  26.0    7-3   256.0
Jeff Withey         Utah Jazz    24.0        C  26.0    7-0   231.0
NaN                      NaN     NaN      NaN   NaN    NaN     NaN

                       College     Salary
Name
Avery Bradley            Texas  7730337.0
Jae Crowder          Marquette  6796117.0
John Holland   Boston University        NaN
R.J. Hunter      Georgia State  1148640.0
Jonas Jerebko              NaN  5000000.0
...                        ...        ...
Shelvin Mack            Butler  2433333.0
Raul Neto                  NaN   900000.0
Tibor Pleiss               NaN  2900000.0
Jeff Withey             Kansas   947276.0
NaN                        NaN        NaN

[458 rows x 8 columns]
```

Reading a CSV file ('nba.CSV') into a Pandas DataFrame, using 'Name' as the index column, and printing it

In [25]:
```
print(df.head(5))
```

```
            Name            Team  Number Position   Age Height  Weight  \
0  Avery Bradley  Boston Celtics     0.0       PG  25.0    6-2   180.0
1    Jae Crowder  Boston Celtics    99.0       SF  25.0    6-6   235.0
2   John Holland  Boston Celtics    30.0       SG  27.0    6-5   205.0
3    R.J. Hunter  Boston Celtics    28.0       SG  22.0    6-5   185.0
4  Jonas Jerebko  Boston Celtics     8.0       PF  29.0   6-10   231.0

             College     Salary
0              Texas  7730337.0
1          Marquette  6796117.0
2  Boston University        NaN
3      Georgia State  1148640.0
4                NaN  5000000.0
```

Printing the first 5 rows of the DataFrame

In [26]:
```
print(df.tail(5))
```

```
             Name       Team  Number Position   Age Height  Weight College  \
453  Shelvin Mack  Utah Jazz     8.0       PG  26.0    6-3   203.0  Butler
454     Raul Neto  Utah Jazz    25.0       PG  24.0    6-1   179.0     NaN
455  Tibor Pleiss  Utah Jazz    21.0        C  26.0    7-3   256.0     NaN
456   Jeff Withey  Utah Jazz    24.0        C  26.0    7-0   231.0  Kansas
457           NaN        NaN     NaN      NaN   NaN    NaN     NaN     NaN

        Salary
453  2433333.0
454   900000.0
455  2900000.0
456   947276.0
457        NaN
```

Printing the last 5 rows of the DataFrame

In [27]:
```
print(df.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 458 entries, 0 to 457
Data columns (total 9 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   Name      457 non-null    object
 1   Team      457 non-null    object
 2   Number    457 non-null    float64
 3   Position  457 non-null    object
 4   Age       457 non-null    float64
 5   Height    457 non-null    object
 6   Weight    457 non-null    float64
 7   College   373 non-null    object
 8   Salary    446 non-null    float64
dtypes: float64(4), object(5)
memory usage: 32.3+ KB
None
```

Displaying information about the DataFrame, including column names, data types, and non-null counts

In [28]:
```
print(df.describe())
```

```
           Number         Age      Weight        Salary
count  457.000000  457.000000  457.000000  4.460000e+02
mean    17.678337   26.938731  221.522976  4.842684e+06
std     15.966090    4.404016   26.368343  5.229238e+06
min      0.000000   19.000000  161.000000  3.088800e+04
25%      5.000000   24.000000  200.000000  1.044792e+06
50%     13.000000   26.000000  220.000000  2.839073e+06
75%     25.000000   30.000000  240.000000  6.500000e+06
max     99.000000   40.000000  307.000000  2.500000e+07
```

Generating summary statistics of the numerical columns in the DataFrame

In [29]:
```
print(df.shape)
```

```
(458, 9)
```

Printing the shape (number of rows and columns) of the DataFrame

In [31]:
```
print(df.iloc[0])
```

```
Name         Avery Bradley
Team        Boston Celtics
Number                 0.0
Position                PG
Age                   25.0
Height                 6-2
Weight               180.0
College              Texas
Salary           7730337.0
Name: 0, dtype: object
```

Accessing and printing the first row of the DataFrame using integer-location based indexing

In [33]:
```
print(df.loc['Avery Bradley'])
```

```
Team        Boston Celtics
Number                 0.0
Position                PG
Age                   25.0
Height                 6-2
Weight               180.0
College              Texas
Salary           7730337.0
Name: Avery Bradley, dtype: object
```

Accessing and printing the row corresponding to the index label 'Avery Bradley'

In [34]:
```
df.sort_values(by='Age')
```

Out[34]:

Name	Team	Number	Position	Age	Height	Weight	College	Salary
Devin Booker	Phoenix Suns	1.0	SG	19.0	6-6	206.0	Kentucky	2127840.0
Rashad Vaughn	Milwaukee Bucks	20.0	SG	19.0	6-6	202.0	UNLV	1733040.0
Christian Wood	Philadelphia 76ers	35.0	PF	20.0	6-11	220.0	UNLV	525093.0
Kristaps Porzingis	New York Knicks	6.0	PF	20.0	7-3	240.0	NaN	4131720.0
Bruno Caboclo	Toronto Raptors	20.0	SF	20.0	6-9	205.0	NaN	1524000.0
...	...	...	...	...	...	...	...	...
Vince Carter	Memphis Grizzlies	15.0	SG	39.0	6-6	220.0	North Carolina	4088019.0
Tim Duncan	San Antonio Spurs	21.0	C	40.0	6-11	250.0	Wake Forest	5250000.0
Andre Miller	San Antonio Spurs	24.0	PG	40.0	6-3	200.0	Utah	250750.0
Kevin Garnett	Minnesota Timberwolves	21.0	PF	40.0	6-11	240.0	NaN	8500000.0
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

458 rows × 8 columns

Sorting the DataFrame by the 'Age' column in ascending order (by default)

In [35]:
```
df.sort_values(by='Age',ascending=False)
```

Out[35]:

Name	Team	Number	Position	Age	Height	Weight	College	Salary
Kevin Garnett	Minnesota Timberwolves	21.0	PF	40.0	6-11	240.0	NaN	8500000.0
Andre Miller	San Antonio Spurs	24.0	PG	40.0	6-3	200.0	Utah	250750.0
Tim Duncan	San Antonio Spurs	21.0	C	40.0	6-11	250.0	Wake Forest	5250000.0
Vince Carter	Memphis Grizzlies	15.0	SG	39.0	6-6	220.0	North Carolina	4088019.0
Pablo Prigioni	Los Angeles Clippers	9.0	PG	39.0	6-3	185.0	NaN	947726.0
...	...	...	...	...	...	...	...	...
Trey Lyles	Utah Jazz	41.0	PF	20.0	6-10	234.0	Kentucky	2239800.0
Emmanuel Mudiay	Denver Nuggets	0.0	PG	20.0	6-5	200.0	NaN	3102240.0
Rashad Vaughn	Milwaukee Bucks	20.0	SG	19.0	6-6	202.0	UNLV	1733040.0
Devin Booker	Phoenix Suns	1.0	SG	19.0	6-6	206.0	Kentucky	2127840.0
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

458 rows × 8 columns

Sorting the DataFrame by the 'Age' column in descending order

In [36]:
```
df.sort_values(by=['Age','Weight'])
```

Out[36]:

Name	Team	Number	Position	Age	Height	Weight	College	Salary
Rashad Vaughn	Milwaukee Bucks	20.0	SG	19.0	6-6	202.0	UNLV	1733040.0
Devin Booker	Phoenix Suns	1.0	SG	19.0	6-6	206.0	Kentucky	2127840.0
Dante Exum	Utah Jazz	11.0	PG	20.0	6-6	190.0	NaN	3777720.0
D'Angelo Russell	Los Angeles Lakers	1.0	PG	20.0	6-5	195.0	Ohio State	5103120.0
Tyus Jones	Minnesota Timberwolves	1.0	PG	20.0	6-2	195.0	Duke	1282080.0
...	...	...	...	...	...	...	...	...
Vince Carter	Memphis Grizzlies	15.0	SG	39.0	6-6	220.0	North Carolina	4088019.0
Andre Miller	San Antonio Spurs	24.0	PG	40.0	6-3	200.0	Utah	250750.0
Kevin Garnett	Minnesota Timberwolves	21.0	PF	40.0	6-11	240.0	NaN	8500000.0
Tim Duncan	San Antonio Spurs	21.0	C	40.0	6-11	250.0	Wake Forest	5250000.0
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

458 rows × 8 columns

Sorting the DataFrame first by 'Age', then by 'Weight' (ascending by default)

In [37]:
```
df.sort_values(by=['Age','Weight'],na_position='first')
```

Out[37]:

Name	Team	Number	Position	Age	Height	Weight	College	Salary
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
Rashad Vaughn	Milwaukee Bucks	20.0	SG	19.0	6-6	202.0	UNLV	1733040.0
Devin Booker	Phoenix Suns	1.0	SG	19.0	6-6	206.0	Kentucky	2127840.0
Dante Exum	Utah Jazz	11.0	PG	20.0	6-6	190.0	NaN	3777720.0
D'Angelo Russell	Los Angeles Lakers	1.0	PG	20.0	6-5	195.0	Ohio State	5103120.0
...	...	...	...	...	...	...	...	...
Pablo Prigioni	Los Angeles Clippers	9.0	PG	39.0	6-3	185.0	NaN	947726.0
Vince Carter	Memphis Grizzlies	15.0	SG	39.0	6-6	220.0	North Carolina	4088019.0
Andre Miller	San Antonio Spurs	24.0	PG	40.0	6-3	200.0	Utah	250750.0
Kevin Garnett	Minnesota Timberwolves	21.0	PF	40.0	6-11	240.0	NaN	8500000.0
Tim Duncan	San Antonio Spurs	21.0	C	40.0	6-11	250.0	Wake Forest	5250000.0

458 rows × 8 columns

Sorting the DataFrame by 'Age' and 'Weight' while placing missing values at the top ('first')

Foundations Of Data Science

Name: Krishna GSVV
Roll no. AV.EN.U4CSE22016

Lab 4 (Handling Missing Values)

In [16]:
```
import pandas as pd
data={'A':[1,2,None,4,5],'B':[None,2,3,4,5],'C':[1,None,3,4,None]}
df=pd.DataFrame(data)
print(df)
```

```
     A    B    C
0  1.0  NaN  1.0
1  2.0  2.0  NaN
2  NaN  3.0  3.0
3  4.0  4.0  4.0
4  5.0  5.0  NaN
```

Creating a DataFrame with missing values (None) and printing it

In [2]:
```
df.isnull()
```

Out[2]:

	A	B	C
0	False	True	False
1	False	False	True
2	True	False	False
3	False	False	False
4	False	False	True

Checking for missing (null) values in the DataFrame, returns a DataFrame of boolean values

In [3]:
```
df.notnull()
```

Out[3]:

	A	B	C
0	True	False	True
1	True	True	False
2	False	True	True
3	True	True	True
4	True	True	False

Checking for non-null values in the DataFrame, returns a DataFrame of boolean values

In [3]:
```
df['A'].isnull()
```

Out[3]:
```
0    False
1    False
2     True
3    False
4    False
Name: A, dtype: bool
```

Checking for null values in column 'A' and returning a boolean Series

In [4]:
```
df.isnull().sum()
```

Out[4]:
```
A    1
B    1
C    2
dtype: int64
```

Counting the number of missing values in each column

In [5]:
```
df.dropna(axis=0)
```

Out[5]:

	A	B	C
3	4.0	4.0	4.0

Dropping rows (axis=0) with any missing values from the DataFrame

In [6]:
```
df.dropna(axis=1)
```

Out[6]:

0
1
2
3
4

Dropping columns (axis=1) with any missing values from the DataFrame

In [7]:
```
df2 = pd.DataFrame([[1, 2, 3],
                    [1, 2, 3],
                    [1, 2, 3]])
df2.dropna(axis=1)
```

Out[7]:

	0	1	2
0	1	2	3
1	1	2	3
2	1	2	3

Creating a DataFrame and dropping columns with missing values (though no missing values in df2)

In [8]:
```
df.fillna(1.5)
```

Out[8]:

	A	B	C
0	1.0	1.5	1.0
1	2.0	2.0	1.5
2	1.5	3.0	3.0
3	4.0	4.0	4.0
4	5.0	5.0	1.5

Filling missing values in the DataFrame with a constant value (1.5)

In [9]:
```
df.apply(lambda col:col.fillna(col.mean(), axis = 0))
```

Out[9]:

	A	B	C
0	1.0	3.5	1.000000
1	2.0	2.0	2.666667
2	3.0	3.0	3.000000
3	4.0	4.0	4.000000
4	5.0	5.0	2.666667

Filling missing values in each column with the mean of that column

In [10]:
```
df
```

Out[10]:

	A	B	C
0	1.0	NaN	1.0
1	2.0	2.0	NaN
2	NaN	3.0	3.0
3	4.0	4.0	4.0
4	5.0	5.0	NaN

In [11]:
```
df.ffill()
```

Out[11]:

	A	B	C
0	1.0	NaN	1.0
1	2.0	2.0	1.0
2	2.0	3.0	3.0
3	4.0	4.0	4.0
4	5.0	5.0	4.0

Forward filling missing values (propagating the previous valid value)

In [12]:
```
df.bfill()
```

Out[12]:

	A	B	C
0	1.0	2.0	1.0
1	2.0	2.0	3.0
2	4.0	3.0	3.0
3	4.0	4.0	4.0
4	5.0	5.0	NaN

Backward filling missing values (propagating the next valid value)

In [13]:
```
df.bfill().ffill()
```

Out[13]:

	A	B	C
0	1.0	2.0	1.0
1	2.0	2.0	3.0
2	4.0	3.0	3.0
3	4.0	4.0	4.0
4	5.0	5.0	4.0

Performing both backward and forward filling on the DataFrame

In [18]:
```
from sklearn.impute import SimpleImputer
imp = SimpleImputer(strategy='constant')
df_imputed = pd.DataFrame(imp.fit_transform(df), columns = df.columns)
df_imputed
```

Out[18]:

	A	B	C
0	1.0	0.0	1.0
1	2.0	2.0	0.0
2	0.0	3.0	3.0
3	4.0	4.0	4.0
4	5.0	5.0	0.0

Using SimpleImputer from sklearn to fill missing values with a constant value, then converting it back to a DataFrame

In [19]:
```
imp = SimpleImputer(strategy='mean')
df_imputed = pd.DataFrame(imp.fit_transform(df), columns = df.columns)
df_imputed
```

Out[19]:

	A	B	C
0	1.0	3.5	1.000000
1	2.0	2.0	2.666667
2	3.0	3.0	3.000000
3	4.0	4.0	4.000000
4	5.0	5.0	2.666667

Using SimpleImputer from sklearn to fill missing values with the mean value of each column

In [20]:
```
imp = SimpleImputer(strategy='median')
df_imputed = pd.DataFrame(imp.fit_transform(df), columns = df.columns)
df_imputed
```

Out[20]:

	A	B	C
0	1.0	3.5	1.0
1	2.0	2.0	3.0
2	3.0	3.0	3.0
3	4.0	4.0	4.0
4	5.0	5.0	3.0

Using SimpleImputer from sklearn to fill missing values with the median value of each column

Foundations of Data Science

Name: Krishna GSVV
Roll no. AV.EN.U4CSE22016

Lab 5- (Pearson correlation coefficient & PCA)

Pearson correlation coefficient

Using numpy library

In [ ]:
```
import numpy as np

A = np.array([12,16,20,24,28,32,36])
B = np.array([6,9,12,15,18,21,24])

mean_A = np.mean(A)
mean_B = np.mean(B)
print("Mean of A:",mean_A)
print("Mean of B:",mean_B)

sum1 = np.sum(A-mean_A)
sum2 = np.sum(B-mean_B)
print("A[i]-mean(A):",sum1)
print("B[i]-mean(B):",sum2)

sum3 = np.sum((A-mean_A)*(B-mean_B))
print("(A[i]-mean(A)(B[i]-mean(B)):",sum3)

std_A = np.std(A)
std_B = np.std(B)
print("Standard Deviation of A:",std_A)
print("Standard Deviation of B:",std_B)
n = len(A)

r = (sum3)/(n*std_A*std_B)
print("Correlation Coefficient:",r)
```

```
Mean of A: 24.0
Mean of B: 15.0
A[i]-mean(A): 0.0
B[i]-mean(B): 0.0
(A[i]-mean(A)(B[i]-mean(B)): 336.0
Standard Deviation of A: 8.0
Standard Deviation of B: 6.0
Correlation Coefficient: 1.0
```

This section calculates the Pearson correlation coefficient between two arrays, A and B, using the NumPy library. It first calculates the mean and standard deviation of each array, then uses these values to calculate the correlation coefficient.

Using Pandas library

In [ ]:
```
import pandas as pd

data = {'x': [12,16,20,24,28,32,36], 'y': [6,9,12,15,18,21,24]}
df = pd.DataFrame(data)

correlation = df['x'].corr(df['y'])

print("Correlation Coefficient:",correlation)
```

```
Correlation Coefficient: 1.0
```

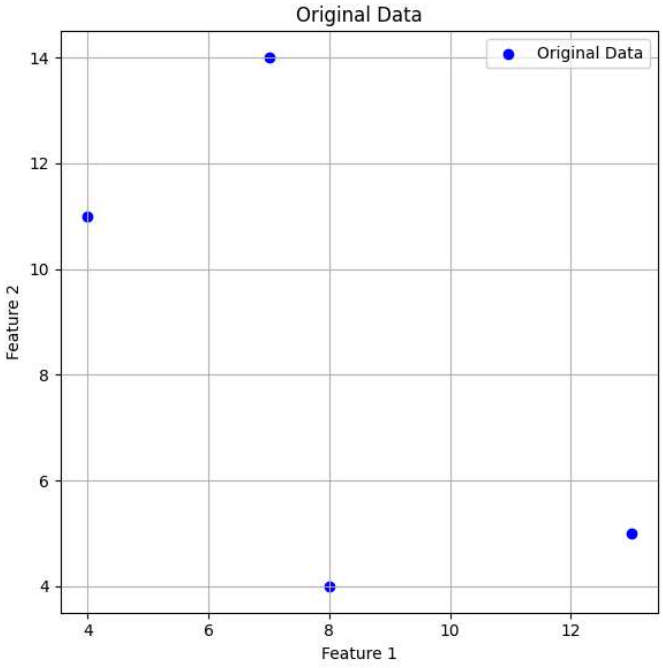
This section calculates the Pearson correlation coefficient between two columns of a Pandas DataFrame, x and y, using the corr() method. This is a more concise way to calculate the correlation coefficient than using NumPy directly.

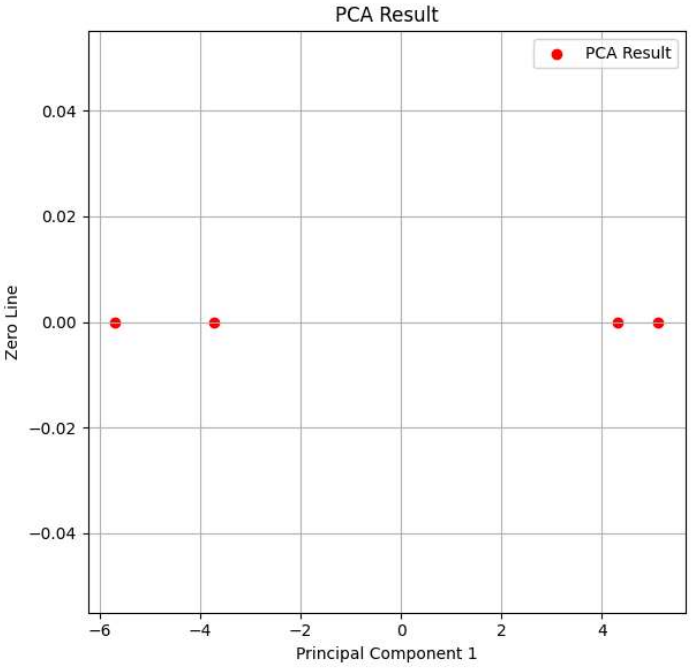
Principle Component Analysis

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt

def pca(X, num_components):
```

```
    mean = np.mean(X, axis=0)
    X_centered = X - mean
    print("Mean of each feature:\n", mean)
    print("Centered Data:\n", X_centered)

    covariance_matrix = np.cov(X_centered, rowvar=False)
    print("Covariance Matrix:\n", covariance_matrix)

    eigenvalues, eigenvectors = np.linalg.eigh(covariance_matrix)
    print("Eigenvalues:\n", eigenvalues)
    print("Eigenvectors:\n", eigenvectors)

    sorted_indices = np.argsort(eigenvalues)[::-1]
    eigenvalues = eigenvalues[sorted_indices]
    eigenvectors = eigenvectors[:, sorted_indices]
    print("Sorted Eigenvalues:\n", eigenvalues)
    print("Sorted Eigenvectors:\n", eigenvectors)

    eigenvectors = eigenvectors[:, :num_components]
    print(f"Top {num_components} Principal Components:\n", eigenvectors)

    X_reduced = np.dot(X_centered, eigenvectors)
    print("Reduced Data:\n", X_reduced)

    return X_reduced


X = np.array([
    [4, 8, 13, 7],
    [11, 4, 5, 14] ]).T

print("Original Data (Samples as rows):\n", X)

plt.figure(figsize=(12, 6))

plt.subplot(1, 2, 1)
plt.scatter(X[:, 0], X[:, 1], color='blue', label='Original Data', marker='o')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.title('Original Data')
plt.legend()
plt.grid(True)

X_reduced = pca(X, num_components=1)

plt.subplot(1, 2, 2)
plt.scatter(X_reduced, np.zeros_like(X_reduced), color='red', label='PCA Result', marker='o')
plt.xlabel('Principal Component 1')
plt.ylabel('Zero Line')
plt.title('PCA Result')
plt.legend()
plt.grid(True)

plt.tight_layout()
plt.show()

print("Original Data (Samples as rows):\n", X)
print("Reduced Data:\n", X_reduced)
```

```
Original Data (Samples as rows):
 [[ 4 11]
 [ 8  4]
 [13  5]
 [ 7 14]]
Mean of each feature:
 [8.  8.5]
Centered Data:
 [[-4.   2.5]
 [ 0.  -4.5]
 [ 5.  -3.5]
 [-1.   5.5]]
Covariance Matrix:
 [[ 14. -11.]
 [-11.  23.]]
Eigenvalues:
 [ 6.61513568 30.38486432]
Eigenvectors:
 [[-0.83025082 -0.55738997]
 [-0.55738997  0.83025082]]
Sorted Eigenvalues:
 [30.38486432  6.61513568]
Sorted Eigenvectors:
 [[-0.55738997 -0.83025082]
 [ 0.83025082 -0.55738997]]
Top 1 Principal Components:
 [[-0.55738997]
 [ 0.83025082]]
Reduced Data:
 [[ 4.30518692]
 [-3.73612869]
 [-5.69282771]
 [ 5.12376947]]
```






```
Original Data (Samples as rows):
 [[ 4 11]
 [ 8  4]
 [13  5]
 [ 7 14]]
Reduced Data:
 [[ 4.30518692]
 [-3.73612869]
 [-5.69282771]
 [ 5.12376947]]
```

This section implements PCA to reduce the dimensionality of a dataset. It first centers the data by subtracting the mean of each feature. Then, it calculates the covariance matrix of the centered data and performs eigendecomposition to obtain the eigenvalues and eigenvectors. Finally, it selects the top num_components eigenvectors (principal components) and projects the data onto these components to obtain a lower-dimensional representation.

Foundations Of Data Science

Name: Krishna GSVV
Roll no. AV.EN.U4CSE22016

Lab 6 (Visualizing Data - 1)

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

Imports necessary libraries: numpy, pandas, matplotlib.pyplot, and seaborn.

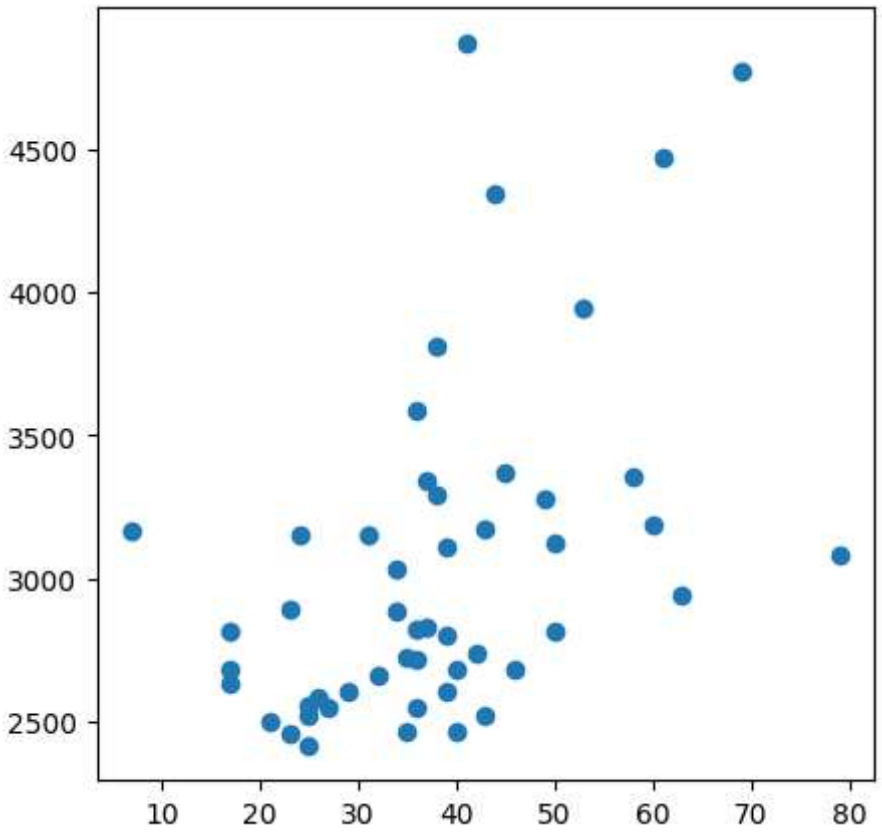
In [2]:
```
df = pd.read_excel("actors.xlsx")
print(df.columns)
```

```
Index([                  1,             'Actor',       'Total Gross',
       'Number of Movies', 'Average per Movie',          '#1 Movie',
                 'Gross'],
      dtype='object')
```

Reads data from an Excel file named "actors.xlsx" into a pandas DataFrame called df

In [3]:
```
plt.figure(figsize=(5, 5))
plt.scatter(df['Number of Movies'],df['Total Gross'])
```

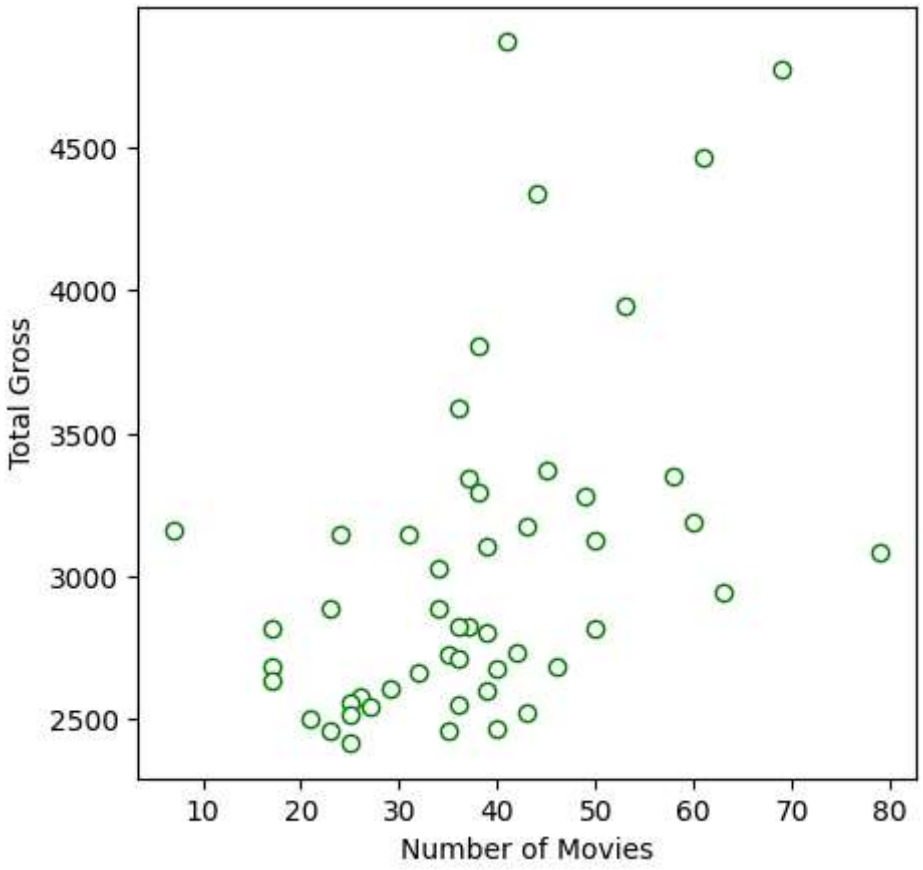
Out[3]:
```
<matplotlib.collections.PathCollection at 0x19e0e14ba70>
```



Creates a scatter plot of "Number of Movies" vs. "Total Gross" using the df DataFrame.

In [4]:
```
x = df['Number of Movies']
y = df['Total Gross']
plt.figure(figsize=(5, 5))
plt.scatter(x, y, c='white', edgecolor='green', marker='o')
plt.xlabel("Number of Movies")
plt.ylabel("Total Gross")
```

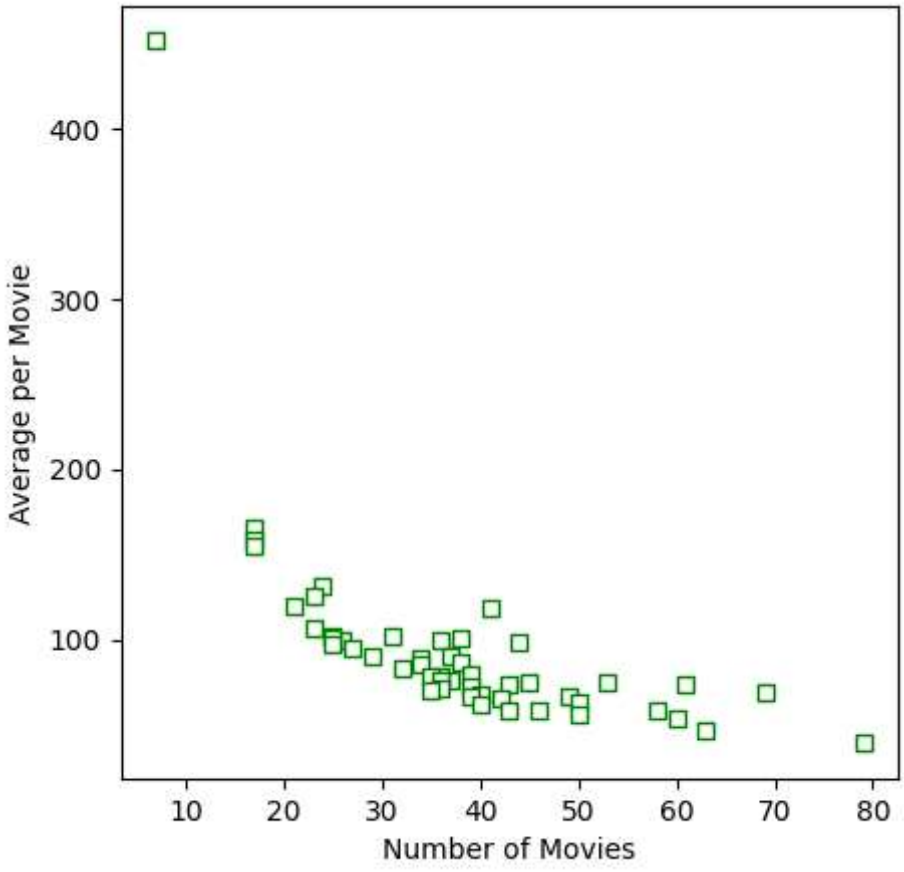
Out[4]:
```
Text(0, 0.5, 'Total Gross')
```



Creates a customized scatter plot of "Number of Movies" vs. "Total Gross", with green circles as markers.

In [5]:
```
x = df['Number of Movies']
y = df['Average per Movie']
plt.figure(figsize=(5, 5))
plt.scatter(x, y, c='white', edgecolor='green', marker='s')
plt.xlabel("Number of Movies")
plt.ylabel("Average per Movie")
```

Out[5]:
```
Text(0, 0.5, 'Average per Movie')
```



Creates a customized scatter plot of "Number of Movies" vs. "Average per Movie", with green squares as markers.

In [6]:
```
outlier = df[df['Number of Movies'] > 10]
outlier
```

Out[6]:

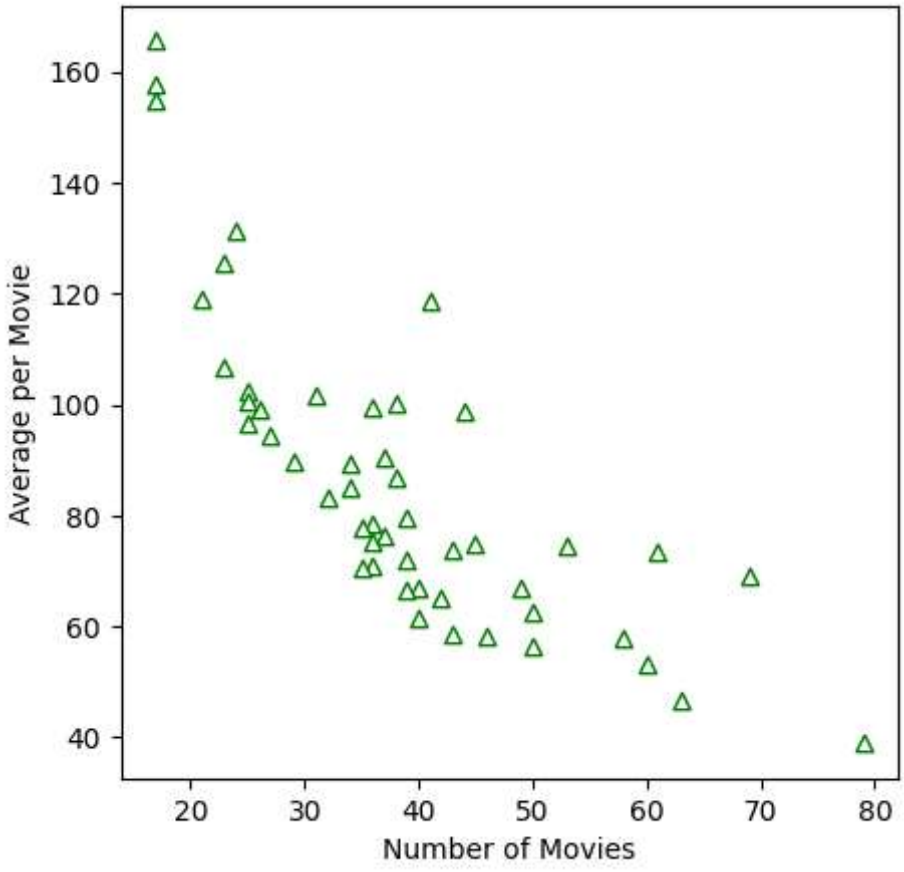
	1	Actor	Total Gross	Number of Movies	Average per Movie	#1 Movie	Gross
0	2	Harrison Ford	4871.7	41	118.8	Star Wars: The Force Awakens	936.7
1	3	Samuel L. Jackson	4772.8	69	69.2	The Avengers	623.4
2	4	Morgan Freeman	4468.3	61	73.3	The Dark Knight	534.9
3	5	Tom Hanks	4340.8	44	98.7	Toy Story 3	415.0
4	6	Robert Downey, Jr.	3947.3	53	74.5	The Avengers	623.4
5	7	Eddie Murphy	3810.4	38	100.3	Shrek 2	441.2
6	8	Tom Cruise	3587.2	36	99.6	War of the Worlds	234.3
7	9	Johnny Depp	3368.6	45	74.9	Dead Man's Chest	423.3
8	10	Michael Caine	3351.5	58	57.8	The Dark Knight	534.9
9	11	Scarlett Johansson	3341.2	37	90.3	The Avengers	623.4
10	12	Gary Oldman	3294.0	38	86.7	The Dark Knight	534.9
11	13	Robin Williams	3279.3	49	66.9	Night at the Museum	250.9
12	14	Bruce Willis	3189.4	60	53.2	Sixth Sense	293.5
13	15	Stellan Skarsgard	3175.0	43	73.8	The Avengers	623.4
15	17	Ian McKellen	3150.4	31	101.6	Return of the King	377.8
16	18	Will Smith	3149.1	24	131.2	Independence Day	306.2
17	19	Stanley Tucci	3123.9	50	62.5	Catching Fire	424.7
18	20	Matt Damon	3107.3	39	79.7	The Martian	228.4
19	21	Robert DeNiro	3081.3	79	39.0	Meet the Fockers	279.3
20	22	Cameron Diaz	3031.7	34	89.2	Shrek 2	441.2
21	23	Liam Neeson	2942.7	63	46.7	The Phantom Menace	474.5
22	24	Andy Serkis	2890.6	23	125.7	Star Wars: The Force Awakens	936.7
23	25	Don Cheadle	2885.4	34	84.9	Avengers: Age of Ultron	459.0
24	26	Ben Stiller	2827.0	37	76.4	Meet the Fockers	279.3
25	27	Helena Bonham Carter	2822.0	36	78.4	Harry Potter / Deathly Hallows (P2)	381.0
26	28	Orlando Bloom	2815.8	17	165.6	Dead Man's Chest	423.3
27	29	Woody Harrelson	2815.8	50	56.3	Catching Fire	424.7
28	30	Cate Blanchett	2802.6	39	71.9	Return of the King	377.8
29	31	Julia Roberts	2735.3	42	65.1	Ocean's Eleven	183.4
30	32	Elizabeth Banks	2726.3	35	77.9	Catching Fire	424.7
31	33	Ralph Fiennes	2715.3	36	75.4	Harry Potter / Deathly Hallows (P2)	381.0

	1	Actor	Total Gross	Number of Movies	Average per Movie	#1 Movie	Gross
32	34	Emma Watson	2681.9	17	157.8	Harry Potter / Deathly Hallows (P2)	381.0
33	35	Tommy Lee Jones	2681.3	46	58.3	Men in Black	250.7
34	36	Brad Pitt	2680.9	40	67.0	World War Z	202.4
35	37	Adam Sandler	2661.0	32	83.2	Hotel Transylvania 2	169.7
36	38	Daniel Radcliffe	2634.4	17	155.0	Harry Potter / Deathly Hallows (P2)	381.0
37	39	Jonah Hill	2605.1	29	89.8	The LEGO Movie	257.8
38	40	Owen Wilson	2602.3	39	66.7	Night at the Museum	250.9
39	41	Idris Elba	2580.6	26	99.3	Avengers: Age of Ultron	459.0
40	42	Bradley Cooper	2557.7	25	102.3	American Sniper	350.1
41	43	Mark Wahlberg	2549.8	36	70.8	Transformers 4	245.4
42	44	Jim Carrey	2545.2	27	94.3	The Grinch	260.0
43	45	Dustin Hoffman	2522.1	43	58.7	Meet the Fockers	279.3
44	46	Leonardo DiCaprio	2518.3	25	100.7	Titanic	658.7
45	47	Jeremy Renner	2500.3	21	119.1	The Avengers	623.4
46	48	Philip Seymour Hoffman	2463.7	40	61.6	Catching Fire	424.7
47	49	Sandra Bullock	2462.6	35	70.4	Minions	336.0
48	50	Chris Evans	2457.8	23	106.9	The Avengers	623.4
49	51	Anne Hathaway	2416.5	25	96.7	The Dark Knight Rises	448.1

Filters the df DataFrame to select rows where "Number of Movies" is greater than 10 and stores it in outlier.

In [7]:
```
x = outlier['Number of Movies']
y = outlier['Average per Movie']
plt.figure(figsize=(5, 5))
plt.scatter(x, y, c='white', edgecolor='green', marker='^')
plt.xlabel("Number of Movies")
plt.ylabel("Average per Movie")
```

Out[7]:
```
Text(0, 0.5, 'Average per Movie')
```

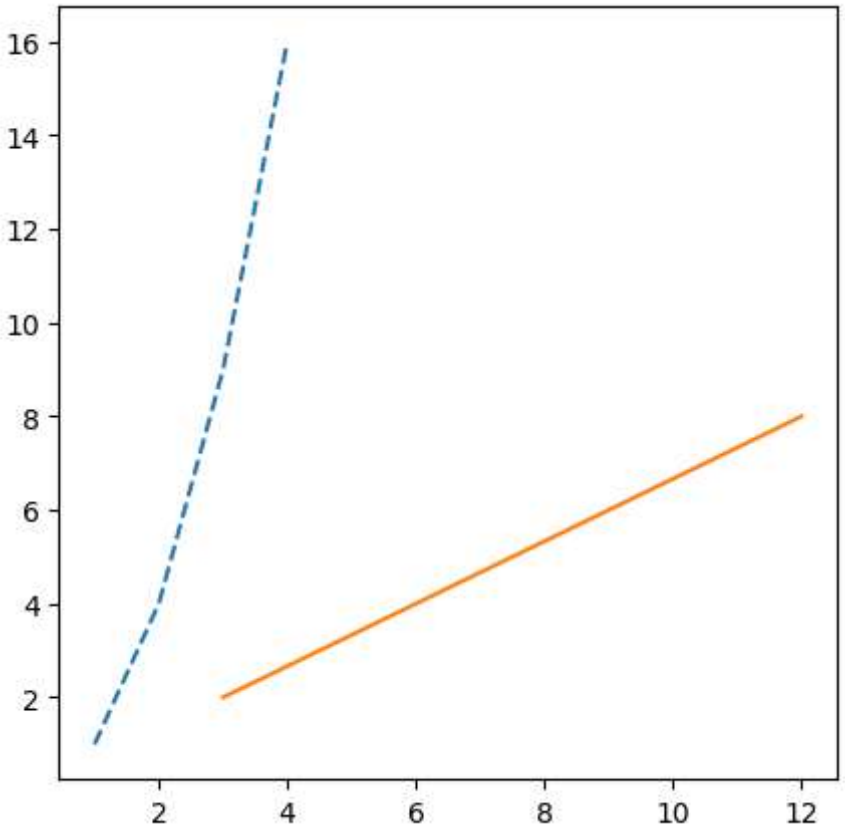


Creates a scatter plot of "Number of Movies" vs. "Average per Movie" for the outlier DataFrame, with green triangles as markers.

In [8]:
```
#Line Plot
```

In [9]:
```
x = np.array([1,2,3,4])
y = x ** 2
w = x * 3
z = x * 2
plt.figure(figsize=(5, 5))
plt.plot(x,y,'--')
plt.plot(w,z)
```

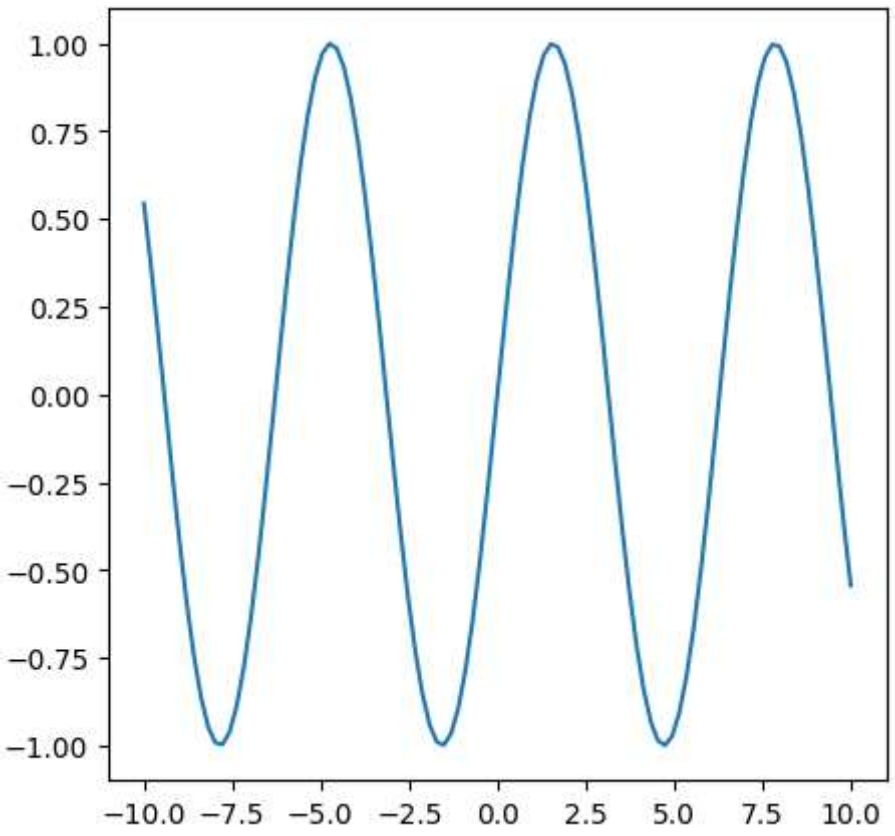
Out[9]:
```
[<matplotlib.lines.Line2D at 0x19e107e3230>]
```



Creates a line plot using numpy arrays, plotting x vs. y with a dashed line and w vs. z with a solid line.

In [10]:
```
x = np.linspace(-10,10,100)
y = np.sin(x)
plt.figure(figsize=(5, 5))
plt.plot(x,y)
```

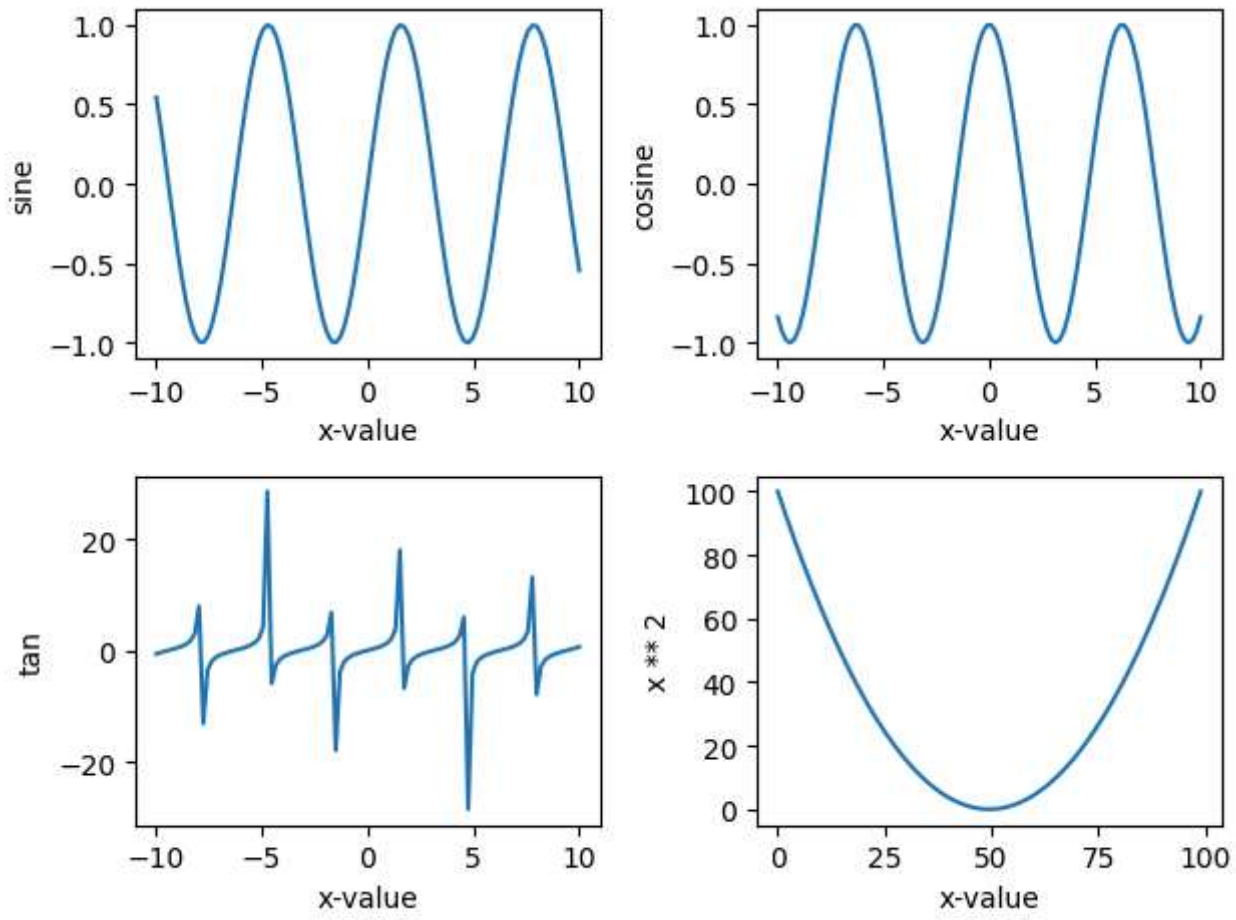
Out[10]:
```
[<matplotlib.lines.Line2D at 0x19e108472f0>]
```



Creates a line plot of the sine function using numpy's linspace and sin

In [11]:
```
x=np.linspace(-10,10,100)
plt.subplot(2,2,1)
```

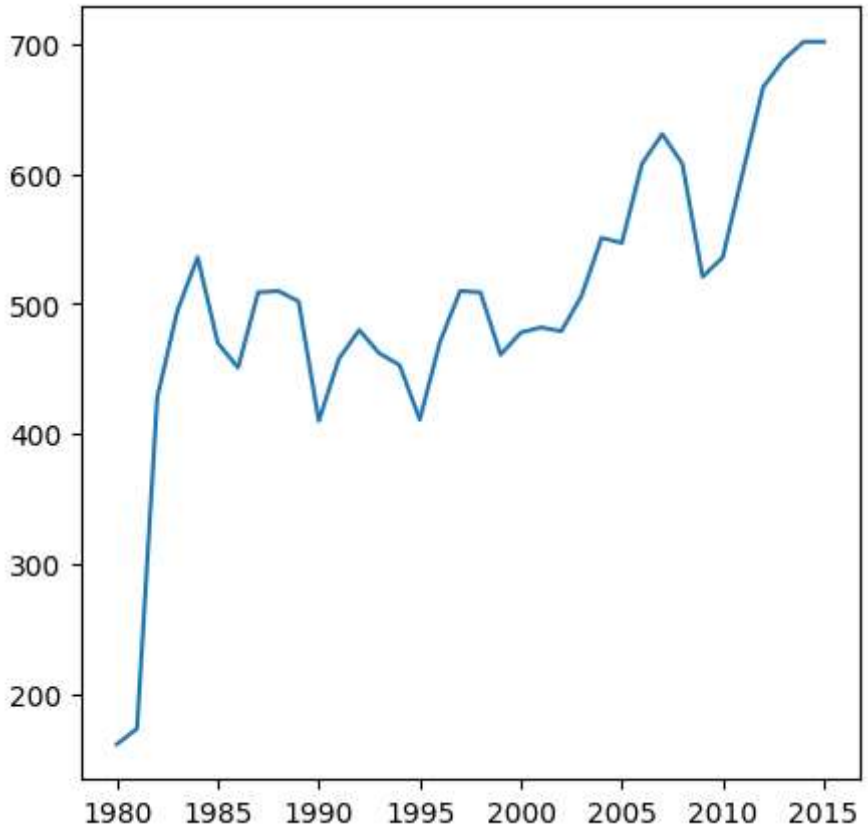
```
plt.xlabel('x-value')
plt.ylabel('sine')
plt.plot(x,np.sin(x))

plt.subplot(2,2,2)
plt.xlabel('x-value')
plt.ylabel('cosine')
plt.plot(x,np.cos(x))

plt.subplot(2,2,3)
plt.xlabel('x-value')
plt.ylabel('tan')
plt.plot(x,np.tan(x))

plt.subplot(2,2,4)
plt.xlabel('x-value')
plt.ylabel('x ** 2')
plt.plot(x ** 2)
plt.tight_layout()
plt.show()
```



Creates a 2x2 subplot grid and plots sine, cosine, tangent, and x squared functions in each subplot.

In [12]:
```
df = pd.read_csv('movies_by_year.csv')
plt.figure(figsize=(5, 5))
plt.plot(df['Year'], df['Number of Movies'])
```

Out[12]:
```
[<matplotlib.lines.Line2D at 0x19e106c35f0>]
```



Reads data from a CSV file named "movies_by_year.csv" and creates a line plot of "Year" vs. "Number of Movies".

In [13]:
```
century_21st = df[df['Year'] >= 2000]
century_21st
```

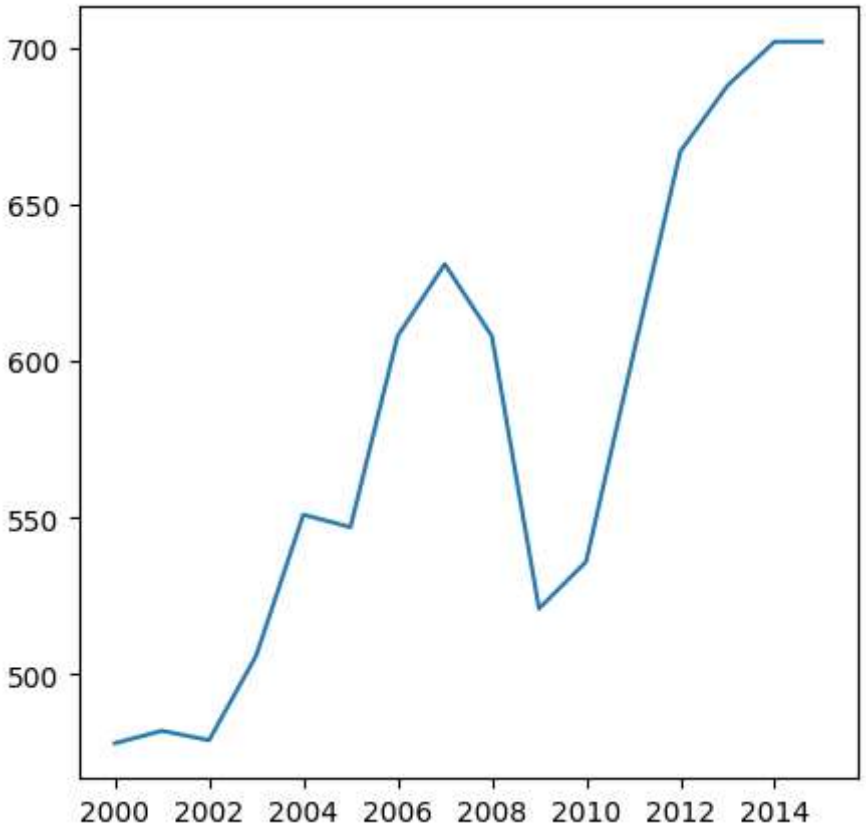
Out[13]:

	1	Year	Total Gross	Number of Movies	#1 Movie
0	2	2015	11128.5	702	Star Wars: The Force Awakens
1	3	2014	10360.8	702	American Sniper
2	4	2013	10923.6	688	Catching Fire
3	5	2012	10837.4	667	The Avengers
4	6	2011	10174.3	602	Harry Potter / Deathly Hallows (P2)
5	7	2010	10565.6	536	Toy Story 3
6	8	2009	10595.5	521	Avatar
7	9	2008	9630.7	608	The Dark Knight
8	10	2007	9663.8	631	Spider-Man 3
9	11	2006	9209.5	608	Dead Man's Chest
10	12	2005	8840.5	547	Revenge of the Sith
11	13	2004	9380.5	551	Shrek 2
12	14	2003	9239.7	506	Return of the King
13	15	2002	9155.0	479	Spider-Man
14	16	2001	8412.5	482	Harry Potter / Sorcerer's Stone
15	17	2000	7661.0	478	The Grinch

Filters the df DataFrame for years greater than or equal to 2000 and stores it in century_21st.

In [14]:
```
plt.figure(figsize=(5, 5))
plt.plot(century_21st['Year'], century_21st['Number of Movies'])
```

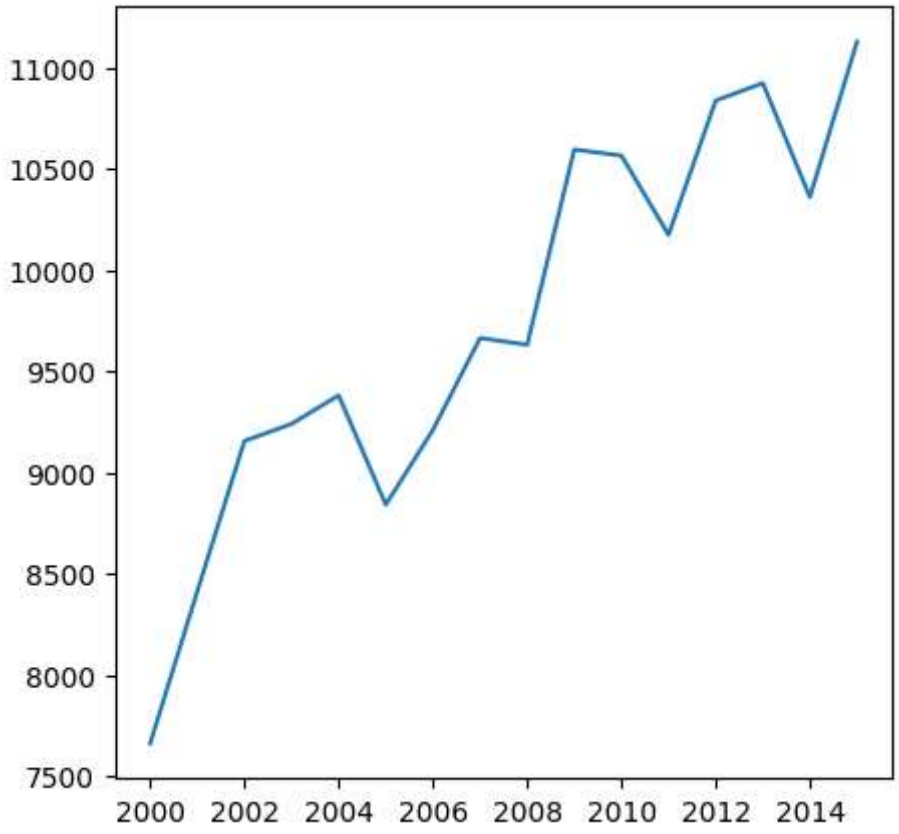
Out[14]:
```
[<matplotlib.lines.Line2D at 0x19e10b9f920>]
```



Creates a line plot of "Year" vs. "Number of Movies" for the century_21st DataFrame.

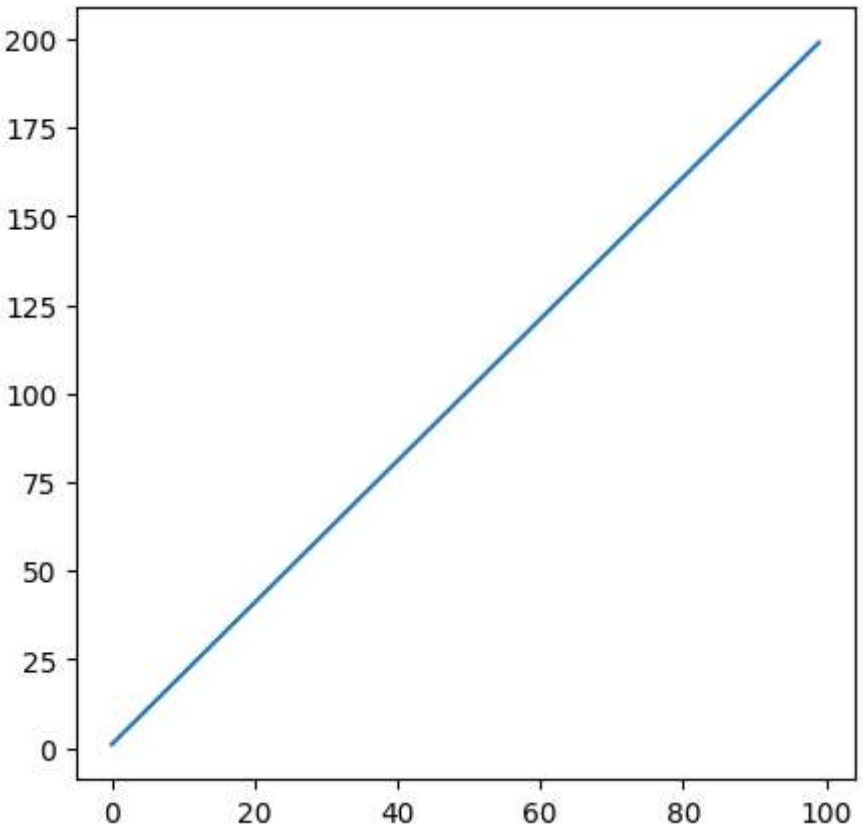
In [15]:
```
plt.figure(figsize=(5, 5))
plt.plot(century_21st['Year'], century_21st['Total Gross'])
```

Out[15]:
```
[<matplotlib.lines.Line2D at 0x19e10b90ce0>]
```



Creates a line plot of "Year" vs. "Total Gross" for the century_21st DataFrame.

In [16]:
```
x = np.arange(100)
y = (2 * x) + 1
plt.figure(figsize=(5, 5))
plt.plot(x,y)
plt.show()
```



Creates a line plot of a linear equation (y = 2x + 1).

Bar Graph

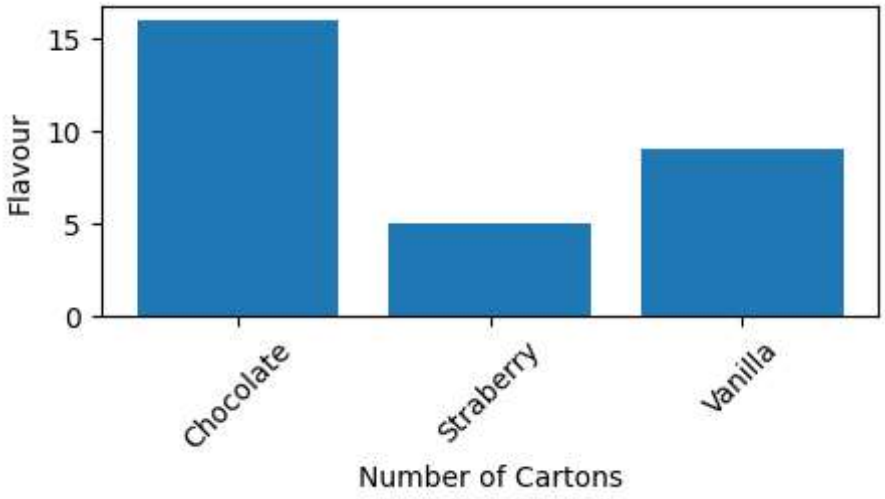
In [17]:
```
data = {'Flavour':['Chocolate','Straberry','Vanilla'],'Number of Cartons':[16,5,9]}
table = pd.DataFrame(data)
table
```

Out[17]:

	Flavour	Number of Cartons
0	Chocolate	16
1	Straberry	5
2	Vanilla	9

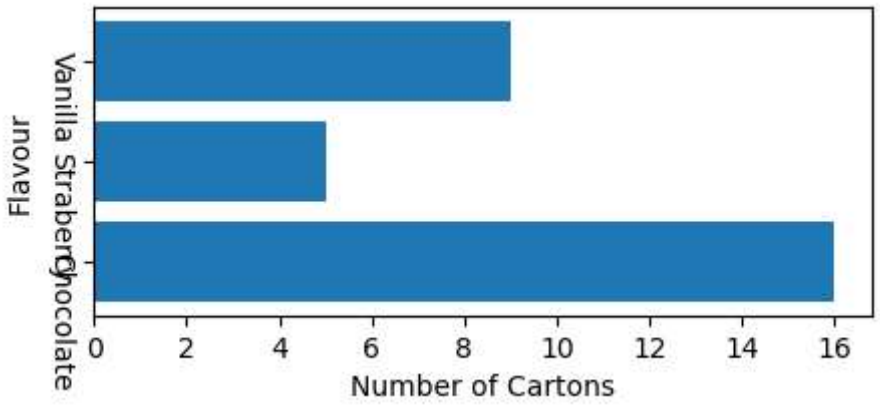
Creates a pandas DataFrame table with data about ice cream flavors and their quantities.

In [18]:
```
plt.figure(figsize=(5, 2))
plt.bar(table['Flavour'],table['Number of Cartons'])
plt.xticks(rotation=45)
plt.xlabel('Number of Cartons')
plt.ylabel('Flavour')
plt.show()
```



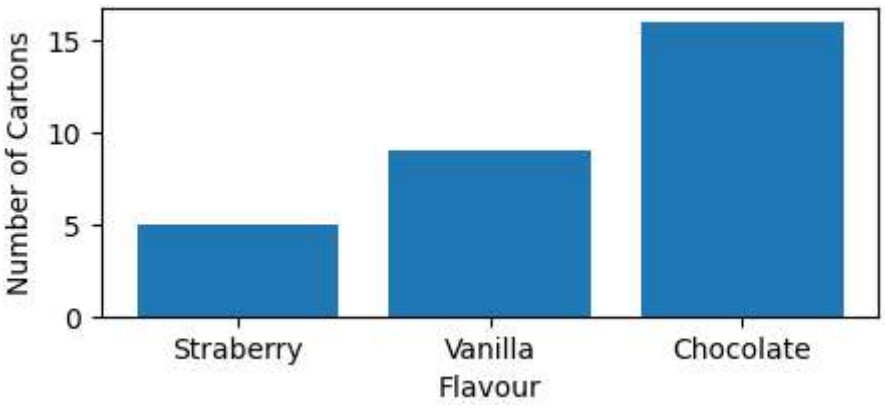
Creates a vertical bar graph of "Flavour" vs. "Number of Cartons" with rotated x-axis labels.

In [19]:
```
plt.figure(figsize=(5, 2))
plt.barh(table['Flavour'],table['Number of Cartons'])
plt.yticks(rotation=270)
plt.xlabel('Number of Cartons')
plt.ylabel('Flavour')
plt.show()
```



Creates a horizontal bar graph of "Flavour" vs. "Number of Cartons" with rotated y-axis labels.

In [20]:
```
sorted = table.sort_values('Number of Cartons')
plt.figure(figsize=(5, 2))
plt.bar(sorted['Flavour'], sorted['Number of Cartons'])
plt.xlabel('Flavour')
plt.ylabel('Number of Cartons')
plt.show()
```



Sorts the table DataFrame by "Number of Cartons" and creates a vertical bar graph using the sorted data.

Foundations Of Data Science

P NITHIN [AV.EN.U4CSE22128]

Lab 8 (Visualizing Data - 2)

In [3]:
```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
```

Imports necessary libraries: pandas, seaborn, and matplotlib.pyplot.

In [4]:
```
df = pd.read_csv('top_movies.csv')
studios = df[['Title','Studio']]
studios_group = studios.groupby('Studio').count()
studios_group.rename(columns = {'1':'Count'}, inplace = True)
print(studios_group)
```

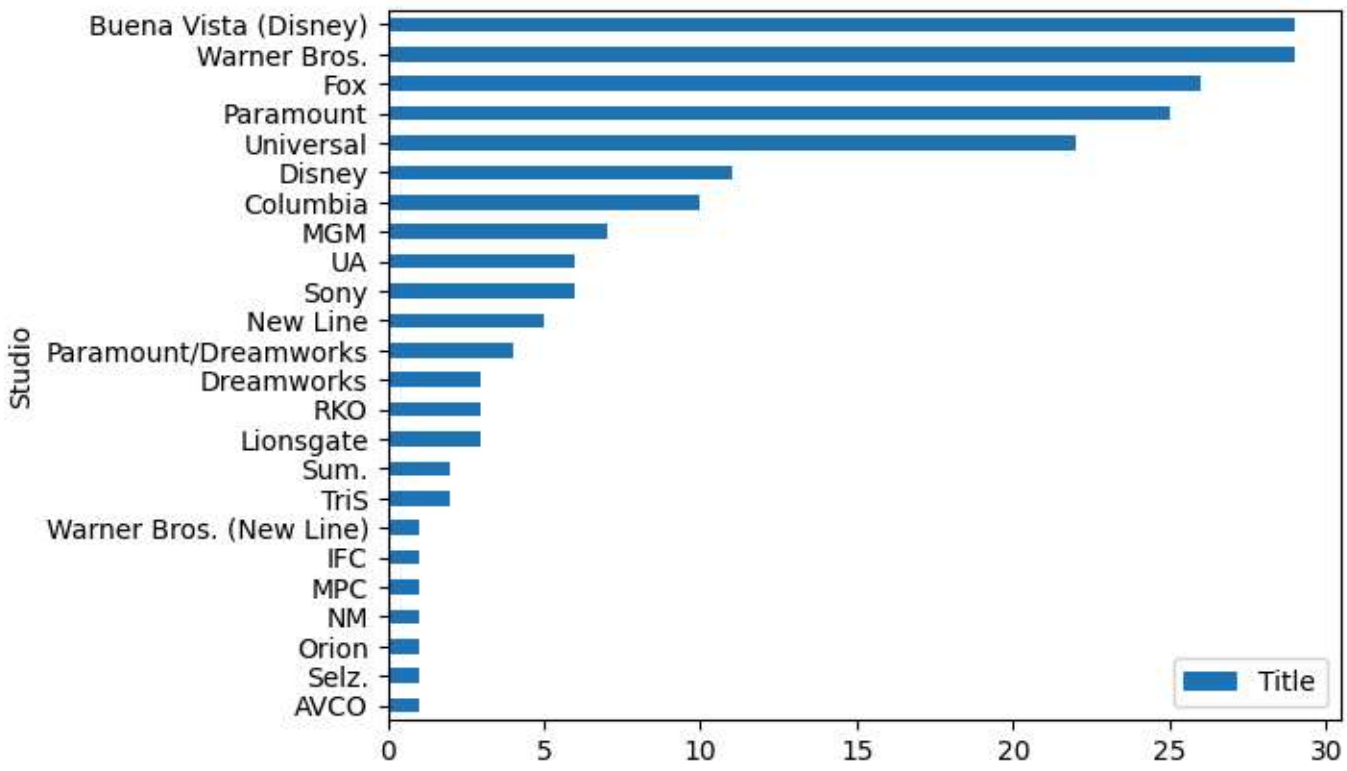
```
                        Title
Studio
AVCO                        1
Buena Vista (Disney)       29
Columbia                   10
Disney                     11
Dreamworks                  3
Fox                        26
IFC                         1
Lionsgate                   3
MGM                         7
MPC                         1
NM                          1
New Line                    5
Orion                       1
Paramount                  25
Paramount/Dreamworks        4
RKO                         3
Selz.                       1
Sony                        6
Sum.                        2
TriS                        2
UA                          6
Universal                  22
Warner Bros.               29
Warner Bros. (New Line)     1
```

Reads data from 'top_movies.csv'. Groups movies by studio and counts the number of movies for each studio. Prints the result.

In [5]:
```
studios_group_sorted = studios_group.sort_values('Title',ascending = True)
print(studios_group_sorted)
studios_group_sorted.plot(kind='barh')
```

Studio	Title
AVCO	1
Selz.	1
Orion	1
NM	1
MPC	1
IFC	1
Warner Bros. (New Line)	1
TriS	2
Sum.	2
Lionsgate	3
RKO	3
Dreamworks	3
Paramount/Dreamworks	4
New Line	5
Sony	6
UA	6
MGM	7
Columbia	10
Disney	11
Universal	22
Paramount	25
Fox	26
Warner Bros.	29
Buena Vista (Disney)	29

Out[5]: <Axes: ylabel='Studio'>



Sorts the studio group data and prints the sorted data. Creates a horizontal bar plot of the number of movies per studio.

In [6]:
```
df=pd.read_csv('top_movies.csv')
df
```

Out[6]:

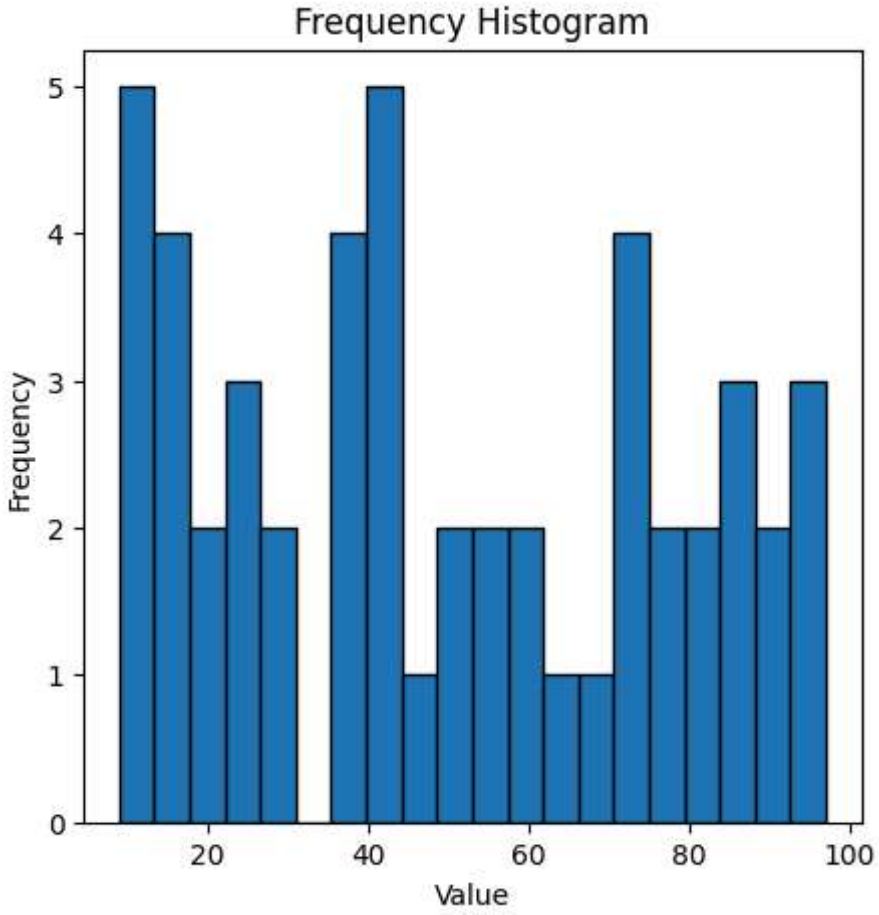
	1	Title	Studio	Gross	Gross (Adjusted)	Year
0	2	Star Wars: The Force Awakens	Buena Vista (Disney)	906723418	906723400	2015
1	3	Avatar	Fox	760507625	846120800	2009
2	4	Titanic	Paramount	658672302	1178627900	1997
3	5	Jurassic World	Universal	652270625	687728000	2015
4	6	Marvel's The Avengers	Buena Vista (Disney)	623357910	668866600	2012
...	...	...	...	...	...	...
195	197	The Caine Mutiny	Columbia	21750000	386173500	1954
196	198	The Bells of St. Mary's	RKO	21333333	545882400	1945
197	199	Duel in the Sun	Selz.	20408163	443877500	1946
198	200	Sergeant York	Warner Bros.	16361885	418671800	1941
199	201	The Four Horsemen of the Apocalypse	MPC	9183673	399489800	1921

200 rows × 6 columns

Reads data from 'top_movies.csv'. Prints the dataframe.

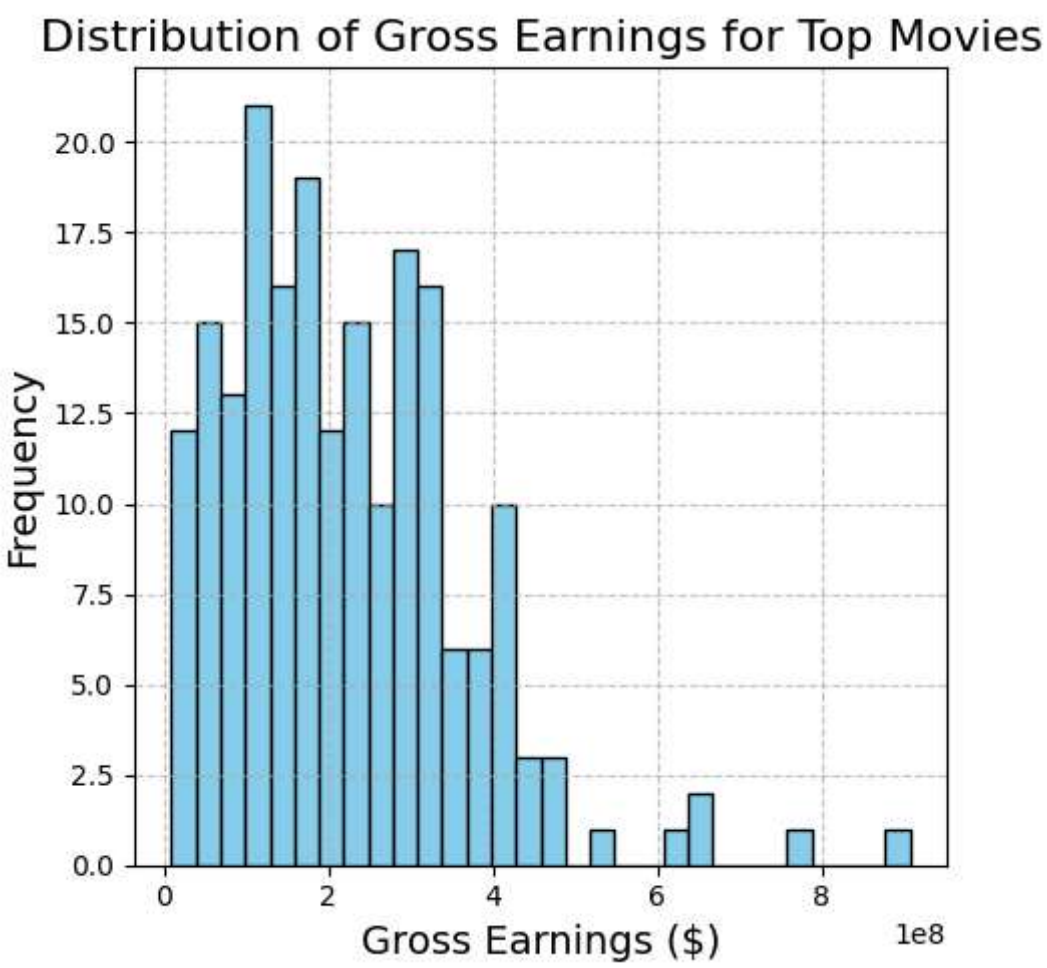
In [7]:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

data = np.random.randint(1, 101, size=50)
datal = pd.DataFrame(data, columns=['Value'])

plt.figure(figsize=(5, 5))
plt.hist(datal['Value'], bins=20, edgecolor='black')

plt.title('Frequency Histogram')
plt.xlabel('Value')
plt.ylabel('Frequency')

plt.show()
```



Generates random data and creates a dataframe. Creates a frequency histogram of the generated data.

In [8]:
```
import pandas as pd
import matplotlib.pyplot as plt

top_movies = pd.read_csv('top_movies.csv')
num_bins = int(np.sqrt(len(top_movies)))
plt.figure(figsize=(5, 5))
plt.hist(top_movies['Gross'], bins=30, edgecolor='black', color='skyblue')
plt.grid(True, linestyle='--', alpha=0.7)
plt.title('Distribution of Gross Earnings for Top Movies', fontsize=16)
plt.xlabel('Gross Earnings ($)', fontsize=14)
plt.ylabel('Frequency', fontsize=14)

plt.tight_layout()
plt.show()
```



Reads data from 'top_movies.csv'. Creates a histogram to visualize the distribution of gross earnings.

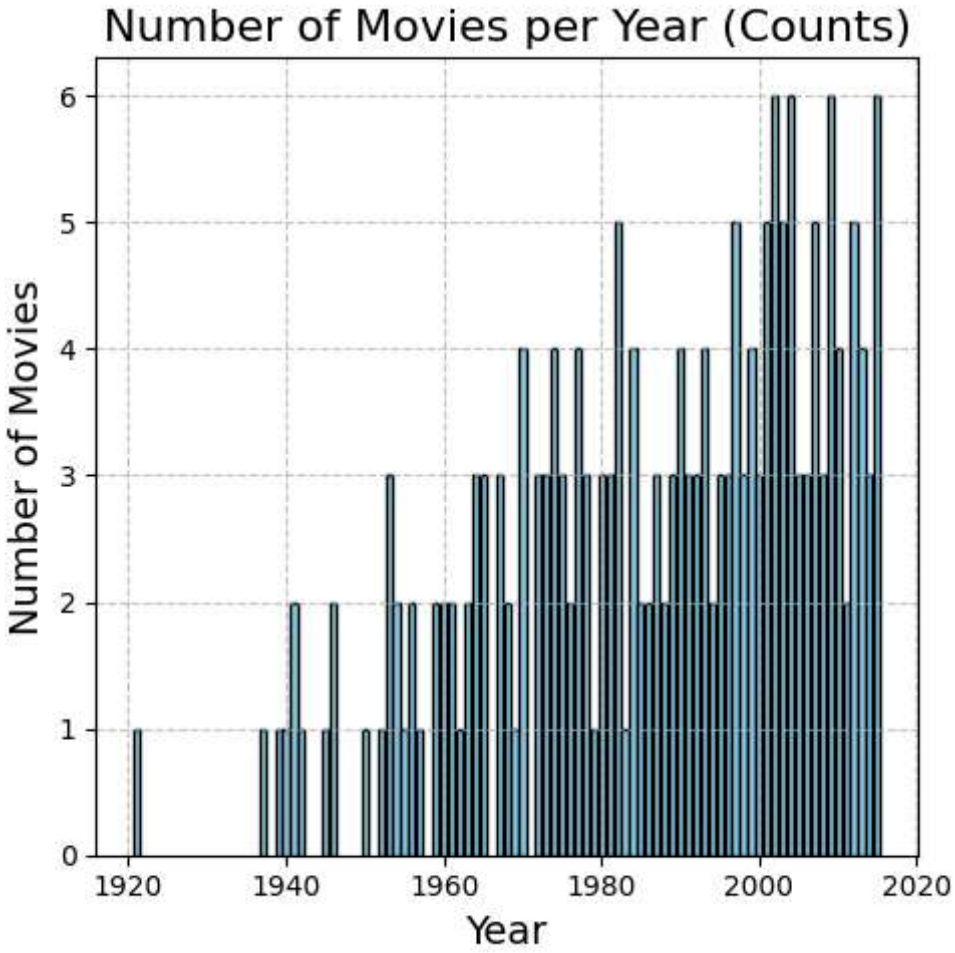
In [9]:
```
top_movies = pd.read_csv('top_movies.csv')

bins = np.arange(300, 2001, 100)
top_movies['Gross Bin'] = pd.cut(top_movies['Gross'], bins=bins)
bin_counts = top_movies['Gross Bin'].value_counts().sort_index()
print(bin_counts)
```

```
Gross Bin
(300, 400]      0
(400, 500]      0
(500, 600]      0
(600, 700]      0
(700, 800]      0
(800, 900]      0
(900, 1000]     0
(1000, 1100]    0
(1100, 1200]    0
(1200, 1300]    0
(1300, 1400]    0
(1400, 1500]    0
(1500, 1600]    0
(1600, 1700]    0
(1700, 1800]    0
(1800, 1900]    0
(1900, 2000]    0
Name: count, dtype: int64
```

Reads data from 'top_movies.csv'. Bins the 'Gross' column and counts the number of movies in each bin. Prints the bin counts.

In [10]:
```
import pandas as pd
import matplotlib.pyplot as plt
```

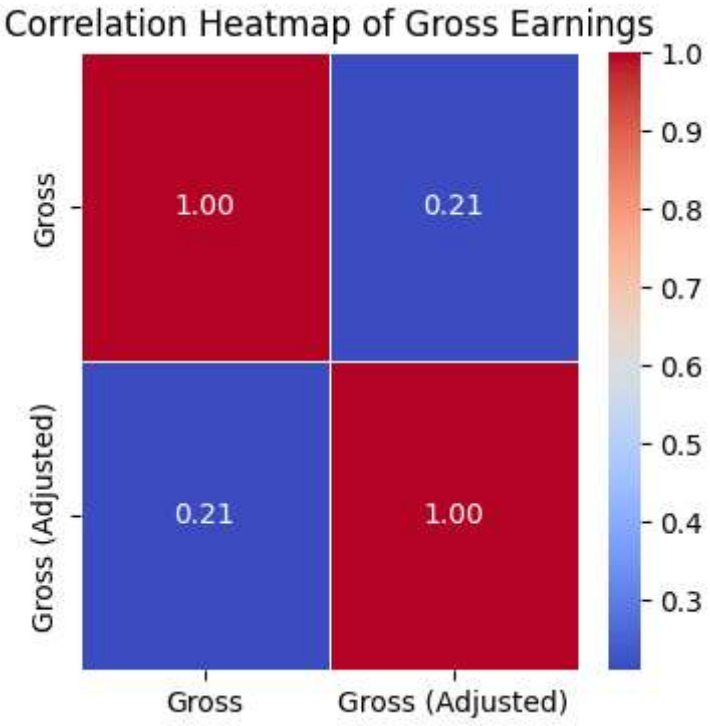
```
top_movies = pd.read_csv('top_movies.csv')

year_counts = top_movies['Year'].value_counts().sort_index().reset_index()
year_counts.columns = ['Year', 'Count']

total_count = year_counts['Count'].sum()
year_counts['Percent'] = (year_counts['Count'] / total_count) * 100
year_counts['Height'] = year_counts['Percent'] / 100

plt.figure(figsize=(5, 5))
plt.bar(year_counts['Year'], year_counts['Count'], color='skyblue', edgecolor='black')

plt.title('Number of Movies per Year (Counts)', fontsize=16)
plt.xlabel('Year', fontsize=14)
plt.ylabel('Number of Movies', fontsize=14)

plt.grid(True, linestyle='--', alpha=0.7)
plt.tight_layout()
plt.show()

print(year_counts[['Year', 'Height']])
```




```
    Year  Height
0   1921   0.005
1   1937   0.005
2   1939   0.005
3   1940   0.005
4   1941   0.010
..   ...     ...
65  2011   0.010
66  2012   0.025
67  2013   0.020
68  2014   0.015
69  2015   0.030

[70 rows x 2 columns]
```

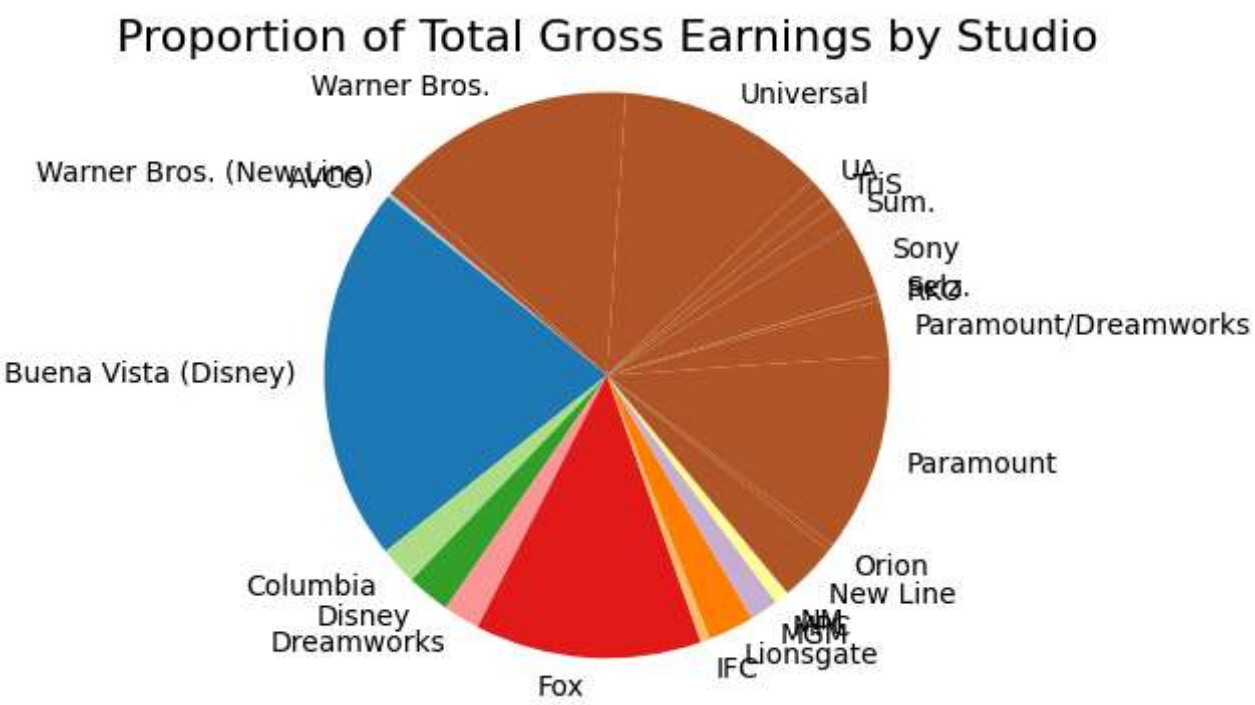
Reads data from 'top_movies.csv'. Calculates the number of movies per year and the percentage. Creates a bar chart of the number of movies per year. Prints a table with year and height (percentage/100).

In [17]:
```
top_movies = pd.read_csv('top_movies.csv')
corr_matrix = top_movies[['Gross', 'Gross (Adjusted)']].corr()
plt.figure(figsize=(4, 4))
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', fmt='.2f', linewidths=0.5)
plt.title('Correlation Heatmap of Gross Earnings')
plt.show()
```



Reads data from 'top_movies.csv'. Creates a correlation heatmap of 'Gross' and 'Gross (Adjusted)'.

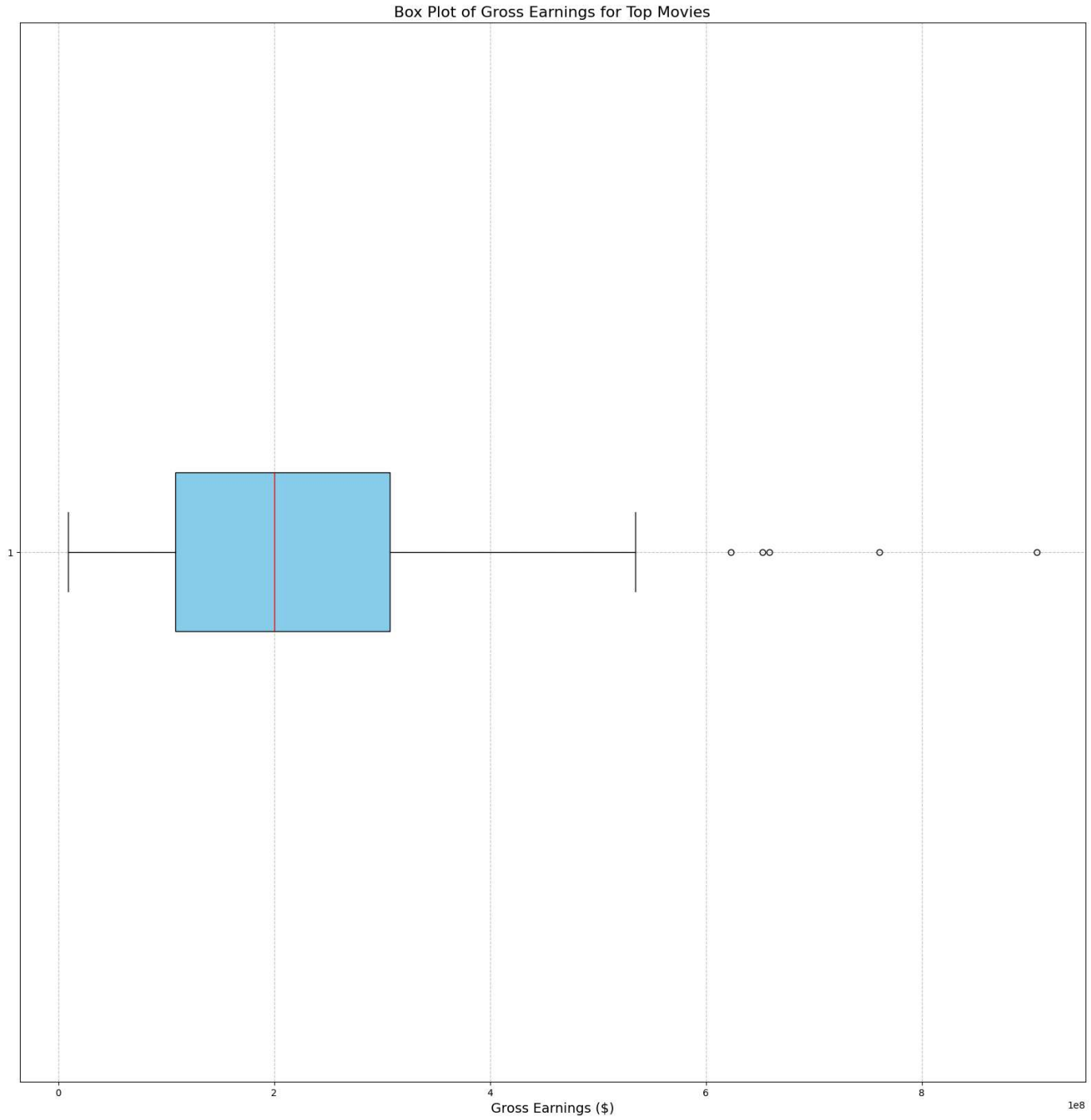
In [21]:
```
studio_gross = top_movies.groupby('Studio')['Gross'].sum()
plt.figure(figsize=(4, 4))
plt.pie(studio_gross, labels=studio_gross.index, startangle=140, colors=plt.cm.Paired(range(l
plt.title('Proportion of Total Gross Earnings by Studio', fontsize=16)
plt.axis('equal')
plt.axis('equal')
plt.xticks(rotate=45)
plt.show()
```



Groups the data by 'Studio' and sums the 'Gross' for each studio. Creates a pie chart to show the proportion of total gross earnings by studio.

Creates a pie chart showing the distribution of total gross by year.

In [20]:
```
plt.figure(figsize=(20,20))
plt.boxplot(top_movies['Gross'], vert=False, patch_artist=True,
            boxprops=dict(facecolor='skyblue', color='black'),
            whiskerprops=dict(color='black'),
            medianprops=dict(color='red'))
plt.title('Box Plot of Gross Earnings for Top Movies', fontsize=16)
plt.xlabel('Gross Earnings ($)', fontsize=14)
plt.grid(True, linestyle='--', alpha=0.7)
plt.show()
```



Creates a box plot of the 'Gross' earnings for the top movies.

Foundation Of Data Science

Name: Krishna GSVV

Roll no. AV.EN.U4CSE22016

Lab 8 (Sampling and Distribution)

In [28]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import random
```

Imports necessary libraries like pandas, numpy, matplotlib, seaborn, and random.

In [64]:
```
d1 = pd.read_csv('uk-500.csv')
d1.head(2)
```

Out[64]:

	first_name	last_name	company_name	address	city	county	postal	phone1	phone
0	Aleshia	Tomkiewicz	Alan D Rosenburg Cpa Pc	14 Taylor St	St. Stephens Ward	Kent	CT2 7PP	01835-703597	01944 36996
1	Evan	Zigomalas	Cap Gemini America	5 Binney St	Abbey Ward	Buckinghamshire	HP11 2AX	01937-864715	01714 73766

Reads the 'uk-500.csv' file into a pandas DataFrame named 'd1' and displays the first few rows

In [30]:
```
d1.sample(n=3)
```

Out[30]:

	first_name	last_name	company_name	address	city	county	postal	phone1	pho
76	Brynn	Elkan	A O Hardee & Son Inc	67 Pulford St	Prittlewell Ward	Essex	SS2 6NL	01388-416867	01 625
303	Wilda	Brigham	Morelli Hoskins Ford	501 Sandon Terrace #200	Little Clacton	Essex	CO16 9LZ	01950-109108	01 561
122	Ressie	Bontemps	Typesetters	42 Eastwood St	Walton Ward	Cambridgeshire	PE4 6HB	01996-854517	01 608

Randomly samples 3 rows from 'd1' without replacement

In [65]:
```
d1.sample(frac=0.1).head(2)
```

Out[65]:

	first_name	last_name	company_name	address	city	county	postal	phone1	phone2
435	Dudley	Dibartolo	Mcknight, H Vincent Jr	91 Ludlow St	Woodhouse Ward	Greater London	N12 0EF	01997-409950	01722-935404
75	Billy	Venus	Tipton & Hurst	61 Miriam St	Abbey Road Ward	Greater London	NW8 9BD	01537-356648	01703-435212

Randomly samples 10% of the rows from 'd1' without replacement.

In [32]:
```
d1.sample(n=3, replace=True)
```

Out[32]:

	first_name	last_name	company_name	address	city	county	postal	phone1	phone2
431	Talia	Marthe	Kenney Mfg Co	5 Minerva St	Westbury	Wiltshire	BA13 3QR	01711-170147	01607-480159
482	Jamika	Conoly	Keller, Raymond C	19 Soho St	Whitley Ward	Berkshire	RG1 9XH	01726-595316	01597-168997
209	Alyce	Flamino	Sadowski, Jeffrey A	4 Mather St	Holbrook Ward	West Midlands	CV6 4BN	01361-927368	01678-781372

Randomly samples 3 rows from 'd1' with replacement.

In [33]:
```
d1.sample(n=3, random_state=1)
```

Out[33]:

	first_name	last_name	company_name	address	city	county	postal	phone1	ph
304	Remedios	Arlinghaus	Miller, Martin M Esq	9 Duckenfield St	Aldbrough	E Riding of Yorkshire	HU11 4QA	01536-498792	01 43
340	Shannon	Kobayashi	Hungs	1111 Nesfield St	Tonge with the Haulgh Ward	Greater Manchester	BL2 2SU	01620-435994	01 12
47	Luisa	Devereux	Cash 4 Checks	3 North View #35	Burmantofts and Richmond Hill	West Yorkshire	LS9 7JH	01607-269930	01 80

Randomly samples 3 rows from 'd1' without replacement, using a specific random state for reproducibility.

In [66]:
```
d1.sample(frac=0.10, replace=True, random_state=5).head(2)
```

Out[66]:

	first_name	last_name	company_name	address	city	county	postal	phone1	phone2
355	Margery	Rohrs	C G Mccullough Insurance Agcy	40 Peters Lane	Long Preston	North Yorkshire	BD23 4NF	01634-340524	01933-273913
206	Ashanti	Donn	Denticator	3409 Benns Gardens	Hollington Ward	East Sussex	TN38 9NE	01352-599278	01721-697548

Randomly samples 10% of the rows from 'd1' with replacement, using a specific random state for reproducibility.

In [35]:
```
d1.sample(n=5, replace=True, random_state=42)
```

Out[35]:

	first_name	last_name	company_name	address	city	county	postal	phone1
102	Graham	Stanwick	Tiburon Pen Chmbr Commrce Inc	73 Hawkstone St	Renfrew South & Gallowhill War	Dunbartonshire	G52 4YG	01860-191930
435	Dudley	Dibartolo	Mcknight, H Vincent Jr	91 Ludlow St	Woodhouse Ward	Greater London	N12 0EF	01997-409950
348	Jacob	Kippel	Acc Automation Inc	4 Monmouth Rd	Llwyn-y-Pia Community	Rhondda Cynon Taff	CF40 2JJ	01550-463222
270	Charlie	Isita	Sotorrio, Rene A Esq	39 Hooton Place	Neilston, Uplawmoor and Newton	East Renfrewshire	G78 3AB	01295-844061
106	Shawana	Cantua	Best Western Riverview Inn	33 Vipond St	Woodhall Farm Ward	Hertfordshire	HP2 7JP	01413-348876

Randomly samples 5 rows from 'd1' with replacement, using a specific random state for reproducibility.

In [36]:
```
d2 = pd.read_csv('StudentsPerformance.csv')
d2.head()
```

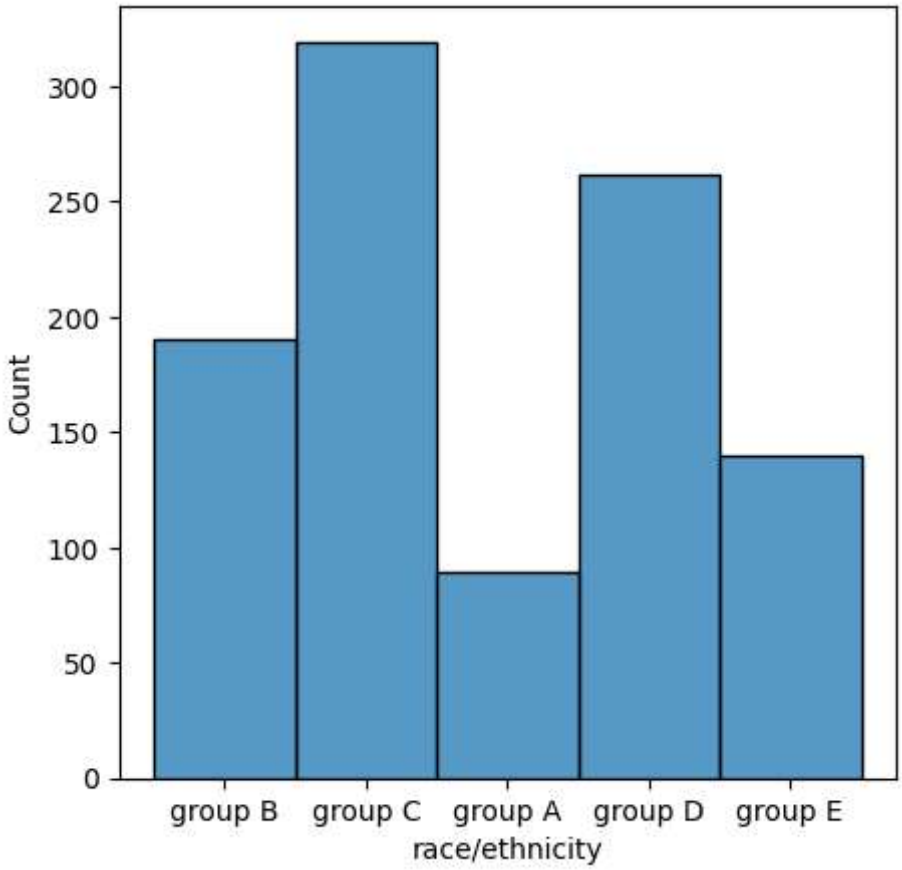
Out[36]:

	gender	race/ethnicity	parental level of education	lunch	test preparation course	math score	reading score	writing score
0	female	group B	bachelor's degree	standard	none	72	72	74
1	female	group C	some college	standard	completed	69	90	88
2	female	group B	master's degree	standard	none	90	95	93
3	male	group A	associate's degree	free/reduced	none	47	57	44
4	male	group C	some college	standard	none	76	78	75

Reads the 'StudentsPerformance.csv' file into a pandas DataFrame named 'd2' and displays the first few rows.

In [58]:
```
plt.figure(figsize=(5, 5))
sns.histplot(d2['race/ethnicity'])
```

Out[58]:
```
<Axes: xlabel='race/ethnicity', ylabel='Count'>
```



Creates a histogram of the 'race/ethnicity' column in 'd2' using seaborn.

In [67]:
```
d2.groupby('race/ethnicity', group_keys=False).apply(lambda x:x.sample(min(len(x),3))).head(2
```

```
<ipython-input-67-be7bfb1670f8>:1: DeprecationWarning: DataFrameGroupBy.apply operated on the
grouping columns. This behavior is deprecated, and in a future version of pandas the grouping
columns will be excluded from the operation. Either pass `include_groups=False` to exclude the
groupings or explicitly select the grouping columns after groupby to silence this warning.
  d2.groupby('race/ethnicity', group_keys=False).apply(lambda x:x.sample(min(len(x),3))).head
(2)
```

Out[67]:

	gender	race/ethnicity	parental level of education	lunch	test preparation course	math score	reading score	writing score
384	female	group A	some high school	free/reduced	none	38	43	43
994	male	group A	high school	standard	none	63	63	62

Groups 'd2' by 'race/ethnicity' and samples a maximum of 3 rows from each group.

In [39]:
```
d2.groupby('gender', group_keys=False).apply(lambda x:x.sample(min(len(x),3)))
```

```
<ipython-input-39-592fe43aea01>:1: DeprecationWarning: DataFrameGroupBy.apply operated on the
grouping columns. This behavior is deprecated, and in a future version of pandas the grouping
columns will be excluded from the operation. Either pass `include_groups=False` to exclude the
groupings or explicitly select the grouping columns after groupby to silence this warning.
  d2.groupby('gender', group_keys=False).apply(lambda x:x.sample(min(len(x),3)))
```

Out[39]:

	gender	race/ethnicity	parental level of education	lunch	test preparation course	math score	reading score	writing score
388	female	group D	high school	standard	none	62	64	64
55	female	group C	high school	free/reduced	none	33	41	43
824	female	group C	some high school	free/reduced	none	48	58	52
495	male	group D	high school	standard	completed	68	64	66
297	male	group E	associate's degree	standard	completed	71	74	68
605	male	group C	some high school	standard	none	75	72	62

Groups 'd2' by 'race/ethnicity' and samples a maximum of 3 rows from each group.

In [68]:
```
step_size = 6
d2.iloc[::step_size].head(2)
```

Out[68]:

	gender	race/ethnicity	parental level of education	lunch	test preparation course	math score	reading score	writing score
0	female	group B	bachelor's degree	standard	none	72	72	74
6	female	group B	some college	standard	completed	88	95	92

Selects every 6th row from 'd2' using slicing.

In [41]:
```
reading_range = np.amax(d2['reading score'])-np.amin(d2['reading score'])
reading_range
```

Out[41]:
```
83
```

Calculates the class interval for 'reading score'.

In [42]:
```
num_of_classes=6
```

In [43]:
```
class_interval = reading_range/num_of_classes
class_interval
```

Out[43]:
```
13.833333333333334
```

Calculates the mean of 'math score' in 'd2'.

In [69]:
```
d2['reading score'].value_counts().head(2)
```

Out[69]:

reading score	count
72	34
74	33

dtype: int64

In [70]:
```
d2['reading score'].value_counts().cumsum().head(2)
```

Out[70]:

reading score	count
72	34
74	67

dtype: int64

In [46]:
```
d2['math score'].mean()
```

Out[46]: 66.089

Calculates the mean of 'math score' in 'd2'.

In [47]:
```
d2.mean(axis=0, skipna=True, numeric_only=True)
```

Out[47]:

	0
math score	66.089
reading score	69.169
writing score	68.054

dtype: float64

Calculates the mean of all numeric columns in 'd2'.

In [48]:
```
d2['reading score'].median()
```

Out[48]: 70.0

Calculates the median of 'reading score' in 'd2'

In [49]:
```
d2['reading score'].mode()
```

Out[49]:

	reading score
0	72

dtype: int64

Calculates the mode of 'reading score' in 'd2'

In [50]:
```
d2.quantile(.2, axis = 0,numeric_only=True)
```

Out[50]:

	0.2
math score	53.0
reading score	57.0
writing score	54.0

dtype: float64

Calculates the 20th percentile of all numeric columns in 'd2'

In [51]:
```
d2.quantile([.1, .3, .63, .95], axis = 0,numeric_only=True)
```

Out[51]:

	math score	reading score	writing score
0.10	47.00	51.00	48.0
0.30	59.00	62.00	60.0
0.63	71.00	74.37	74.0
0.95	90.05	92.00	92.0

Calculates the 10th, 30th, 63rd, and 95th percentiles of all numeric columns in 'd2'.

In [52]:
```
d2.quantile([.25, .50, .75], axis = 0,numeric_only=True)
```

Out[52]:

	math score	reading score	writing score
0.25	57.0	59.0	57.75
0.50	66.0	70.0	69.00
0.75	77.0	79.0	79.00

Calculates the 25th, 50th, and 75th percentiles (quartiles) of all numeric columns in 'd2'.

In [71]:
```
d3 = pd.read_csv('nba.csv')
d3.head(2)
```

Out[71]:

	Name	Team	Number	Position	Age	Height	Weight	College	Salary
0	Avery Bradley	Boston Celtics	0	PG	25	06-Feb	180	Texas	7730337.0
1	Jae Crowder	Boston Celtics	99	SF	25	06-Jun	235	Marquette	6796117.0

Reads the 'nba.csv' file into a pandas DataFrame named 'd3' and displays the first few rows.

In [54]:
```
d3.skew(axis=0, skipna=True, numeric_only=True)
```

Out[54]:

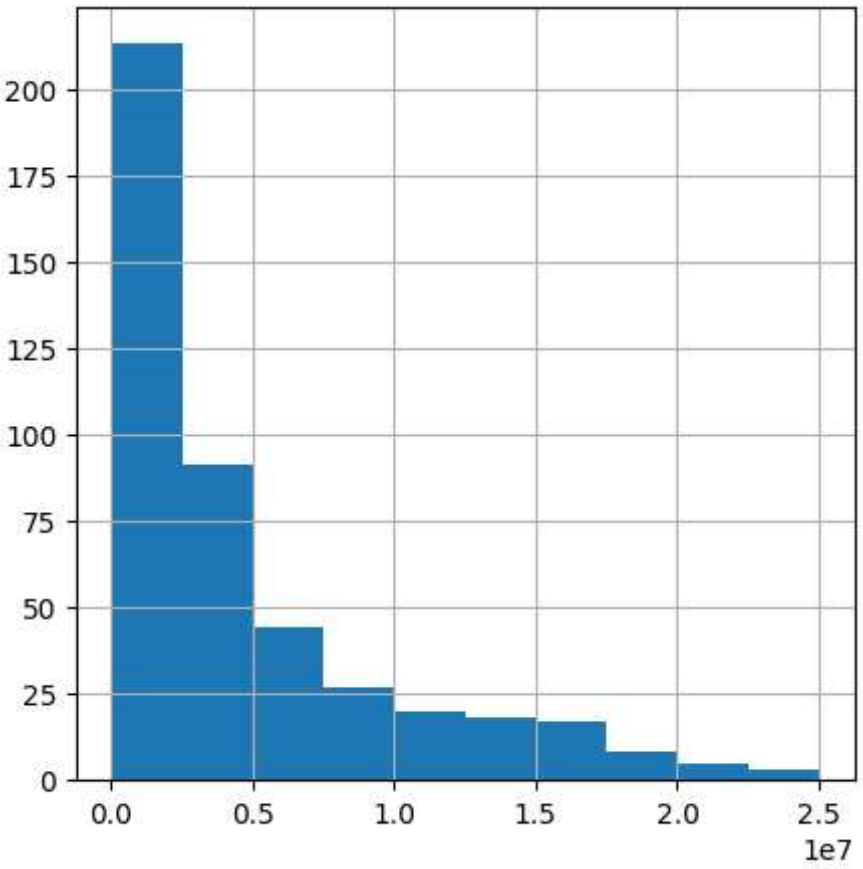
	0
Number	1.668386
Age	0.626349
Weight	0.113788
Salary	1.576321

dtype: float64

Calculates the skewness of all numeric columns in 'd3'.

In [55]:
```
plt.figure(figsize=(5, 5))
d3['Salary'].sort_values().hist().plot(use_index=False)
```

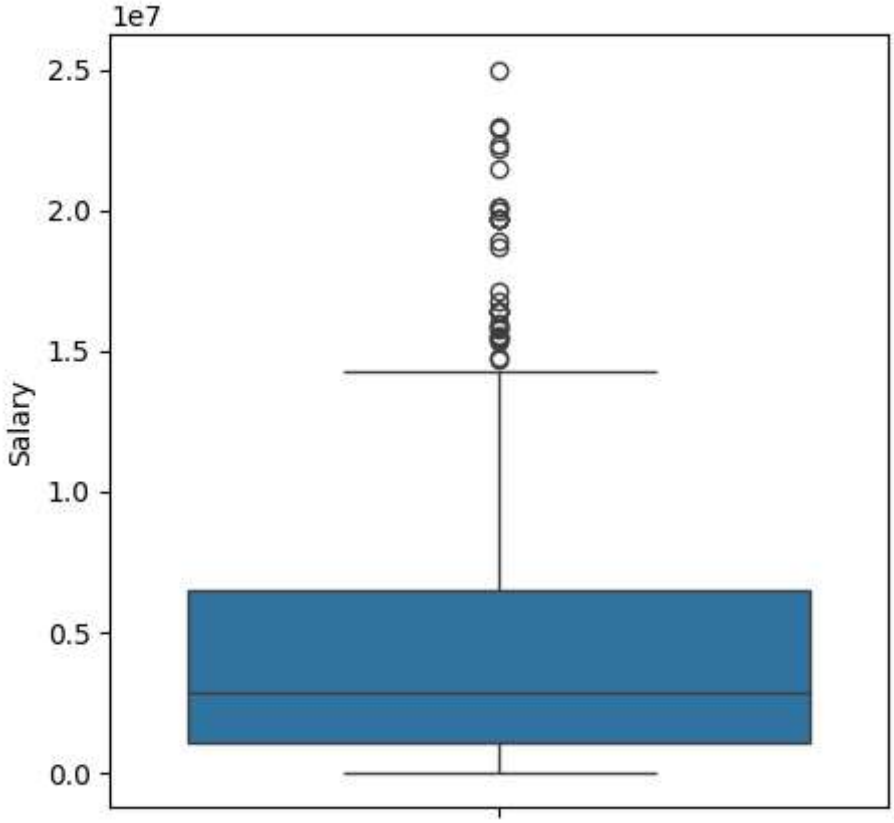
Out[55]: []



Creates a histogram of the 'Salary' column in 'd3' after sorting the values

In [59]:
```
plt.figure(figsize=(5, 5))
sns.boxplot(d3['Salary'])
```

Out[59]: <Axes: ylabel='Salary'>



Creates a box plot of the 'Salary' column in 'd3' using seaborn.

In [57]:
```
d3.kurtosis(numeric_only=True)
```

Out[57]:

	0
Number	4.364464
Age	-0.051119
Weight	-0.552546
Salary	1.839177

dtype: float64

Calculates the kurtosis of all numeric columns in 'd3'.

Foundations of Data Science

Name: Krishna GSVV
Roll no. AV.EN.U4CSE22016

Lab 9 - (Probability Distributions)

In [3]:
```
import numpy as np
```

Imports the numpy library and assigns it the alias np.

Binomial

Suppose there are twelve multiple choice questions in an English class quiz. Each question has five possible answers, and only one of them is correct. Find the probability of having (a) Exactly 4 ans correct

(b) four or less correct answers if a student attempts to answer every question at random. Solution Since only one out of five possible answers is correct, the probability of answering a question correctly by random is 1/5=0.2. We can find the probability of having exactly 4 correct answers by random attempts as follows. (4, size=12, prob=0.2)

In [ ]:
```
sum(np.random.binomial(12, 0.2, 20000) == 4)/20000. # Exactly four answers correct
```

Out[ ]:
```
0.134
```

Calculates the probability of getting exactly 4 questions correct out of 12 using a simulation with 20000 trials. It uses the binomial function from numpy.random and calculates the proportion of trials where the result is 4.

In [ ]:
```
sum(np.random.binomial(12, 0.2, 20000) <= 4)/20000. # four or less correct
```

Out[ ]:
```
0.92845
```

Calculates the probability of getting 4 or less questions correct using a similar simulation approach.

A real world example. A company drills 9 wild-cat oil exploration wells, each with an estimated probability of success of 0.1. All nine wells fail. What is the probability of that happening? Let's do 20,000 trials of the model, and count the number that generate zero positive results.

In [ ]:
```
sum(np.random.binomial(9, 0.1, 20000) == 0)/20000.
```

Out[ ]:
```
0.3879
```

Calculates the probability of all 9 wells failing using a simulation with 20000 trials.

Poisson

If there are twelve cars crossing a bridge per minute on average, find the probability of having seventeen or more cars crossing the bridge in a particular minute.

In [ ]:
```
sum(np.random.poisson(12, 10000) >=17)/10000
```

Out[ ]:
```
0.1036
```

Calculates the probability of having 17 or more cars crossing the bridge using a simulation with 10000 trials.

Uniform Distribution

In [ ]:
```
sum(np.random.uniform(0,20,1000) >5)/1000
```

Out[ ]:
```
0.743
```

In [ ]:
```
sum(np.random.uniform(0,20,1000) <10)/1000
```

Out[ ]:
```
0.492
```

In [ ]:
```
(sum(np.random.uniform(0,20,1000) >5)/1000) - (sum(np.random.uniform(0,20,1000) <10)/1000)
```

Out[ ]:
```
0.22999999999999998
```

In [ ]:
```
s = np.random.uniform(-1,0,1000)
#s
```

In [ ]:
```
np.all(s >= 0)
```

Out[ ]:
```
True
```

In [ ]:
```
np.all(s < 20)
```

Out[ ]:
```
True
```

The above cells explore the Uniform distribution, generating random numbers between 0 and 20, and calculating probabilities based on conditions.

Normal Distribution

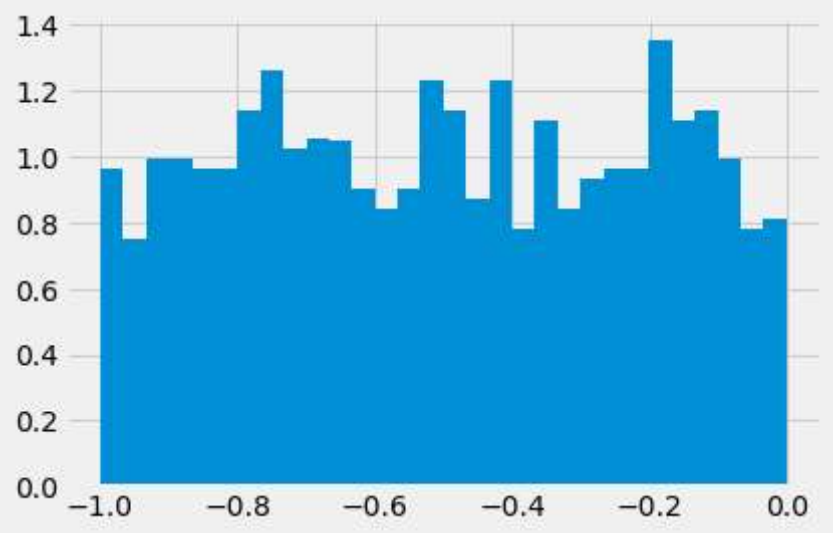
In [ ]:
```
mu, sigma = 10, 2 # mean and standard deviation
sum(np.random.normal(mu, sigma, 1000) >13)/1000
```

Out[ ]:
```
0.072
```

Calculates the probability of a value being greater than 13 in a normal distribution with mean 10 and standard deviation 2 using a simulation.

In [ ]:
```
 s = np.random.normal(mu, sigma, 1000)
```

In [ ]:
```
import matplotlib.pyplot as plt
%matplotlib inline
plt.style.use('fivethirtyeight')
count, bins, ignored = plt.hist(s, 30, density=True)
plt.plot(bins, 1/(sigma * np.sqrt(2 * np.pi)) *
np.exp( - (bins - mu)**2 / (2 * sigma**2) ),
linewidth=2, color='r')
plt.show()
```



These cells generate random numbers from the normal distribution, plot a histogram, and overlay the theoretical normal distribution curve.

Exponential

lamda is 1.4 min time between detections in a Geiger counter. Probability of detecting within 0.6 mins

In [4]:
```
sum(np.random.exponential(scale=1/4, size=1000) < 0.5) / 1000
```

Out[4]:
```
0.857
```

In [ ]:
```
1/1.4
```

Out[ ]:
```
0.7142857142857143
```

In [ ]:
```
#np.random.exponential(scale=1.4, size=1000)
```

In [ ]:
```
sum(np.random.exponential(scale=1.4, size=10000) <0.5)/10000
```

Out[ ]:
```
0.2956
```

In [ ]:
```
import math
x = -(0.5/1.4)
1-math.exp( x )
```

Out[ ]:
```
0.30032746262486965
```

In [ ]:
```
sum(np.random.exponential(scale=12, size=10000) <8)/10000
```

Out[ ]:
```
0.4877
```

In [ ]:
```
import math
x = -(8/12)
1-math.exp( x )
```

Out[ ]:
```
0.486582880967408
```

The above cells explore the Exponential distribution, calculating probabilities and comparing them with theoretical values.

Foundations of Data Science

Name: Krishna GSVV
Roll no. AV.EN.U4CSE22016

Lab 10 (Hypothesis Testing)

Accessing Models

In data science, a "model" is a set of assumptions about data. Often, models include assumptions about chance processes used to generate data.

Sometimes, data scientists have to decide whether or not their models are good. In this section we will discuss two examples of making such decisions. In later sections we will use the methods developed here as the building blocks of a general framework for testing hypotheses.

In 1965, the U.S. Supreme Court case *Swain v. Alabama* addressed whether an all-white jury in Talladega County, Alabama, was formed randomly or was the result of racial exclusion. Robert Swain, a Black man convicted of raping a white woman, appealed his death sentence, pointing to the absence of Black jurors on his trial jury, despite 26% of the eligible jurors in the county being Black. The Supreme Court, however, ruled against him, stating that the racial disparity on the jury panel could have been a result of random selection rather than intentional exclusion. To test if this could indeed be attributed to chance, a simulation model can be applied to evaluate the likelihood of obtaining a similar outcome in a random selection. By simulating the selection of 100 jurors from a population where 26% are Black, we can compare the actual jury composition with the outcomes predicted by the random selection model. If the number of Black jurors in the simulated samples aligns poorly with the observed panel, this would suggest that the jury selection was likely not random, supporting Swain's claim.

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

U.S. Supreme Court, 1965: Swain vs. Alabama

The sample_proportions function will take two arguments:

the sample size

the distribution of the categories in the population, as a list or array of proportions that add up to 1

As the function uses a numpy 'method' an array will be returned containing the distribution of the categories in a random sample of the given size taken from the population. This is an array consisting of the sample proportions in all the different categories.

To see how to use this, remember that according to our model, the panel is selected at random from a population of men among whom 26% were black and 74% were not. Thus the distribution of the two categories can be represented as the list [0.26, 0.74], which we have assigned to the name eligible_population. Now let's sample at random 100 times from this distribution, and see what proportions of the two categories we get in our sample.

In [ ]:
```
def sample_proportions(sample_size, probabilities):
    return np.random.multinomial(sample_size, probabilities) / sample_size
```

In [ ]:
```
eligible_population = [0.26, 0.74]
sample_proportions(100, eligible_population)
```

Out[ ]:
```
array([0.29, 0.71])
```

In [ ]:
```
(100 * sample_proportions(100, eligible_population)).item(0)
```

Out[ ]:
```
27.0
```

Running the Simulation

To get a sense of the variability without running the cell over and over, let's generate 10,000 simulated values of the count.

The code follows the same steps that we have used in every simulation. First, we define a function to simulate one value of the count, using the code we wrote above.

In [ ]:
```
def one_simulated_count():
    return (100 * sample_proportions(100, eligible_population)).item(0)
```

In [ ]:
```
counts = np.array([])

repetitions = 10000
for i in np.arange(repetitions):
    counts = np.append(counts, one_simulated_count())
```

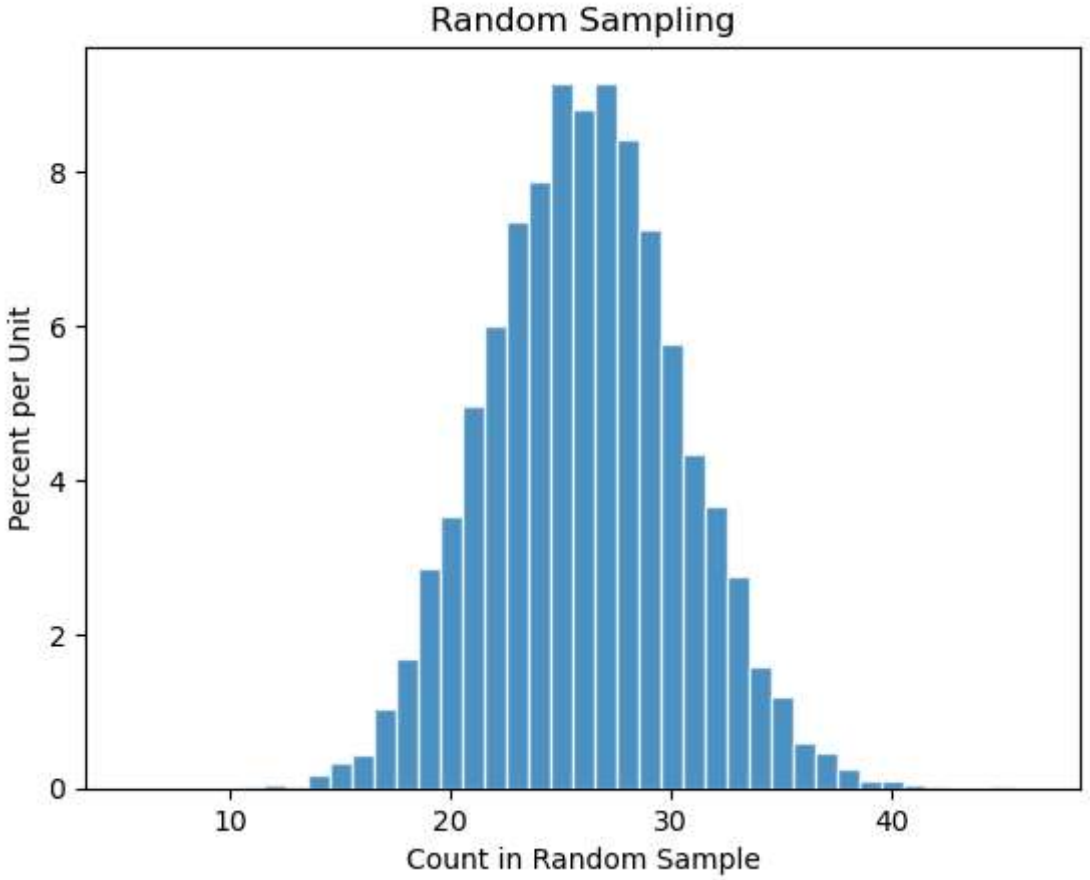
The Prediction

To interpret the results of our simulation, we start as usual by visualizing the results by an empirical histogram.

In [ ]:
```
def hist_with_bins(df, bins, plot_title, x_axis_label, y_axis_label):
    source = df
    fig, ax1 = plt.subplots()
    ax1.hist(source, bins=bins, density=True, alpha=0.8, ec='white')
    y_vals = ax1.get_yticks()
    y_label = y_axis_label
    x_label = x_axis_label
    ax1.set_yticklabels(['{:g}'.format(x * 100) for x in y_vals])
    plt.ylabel(y_label)
    plt.xlabel(x_label)
    plt.title('%s' % plot_title)
    plt.show()
```

In [ ]:
```
random_sample_counts = pd.DataFrame({'Count in a Random Sample':counts})
hist_with_bins(random_sample_counts, np.arange(5.5, 46.6, 1),
               'Random Sampling', 'Count in Random Sample', 'Percent per Unit')
```

```
C:\Users\91868\AppData\Local\Temp\ipykernel_14316\2998315744.py:8: UserWarning: set_ticklabels
() should only be used with a fixed number of ticks, i.e. after set_ticks() or using a FixedLo
cator.
  ax1.set_yticklabels(['{:g}'.format(x * 100) for x in y_vals])
```

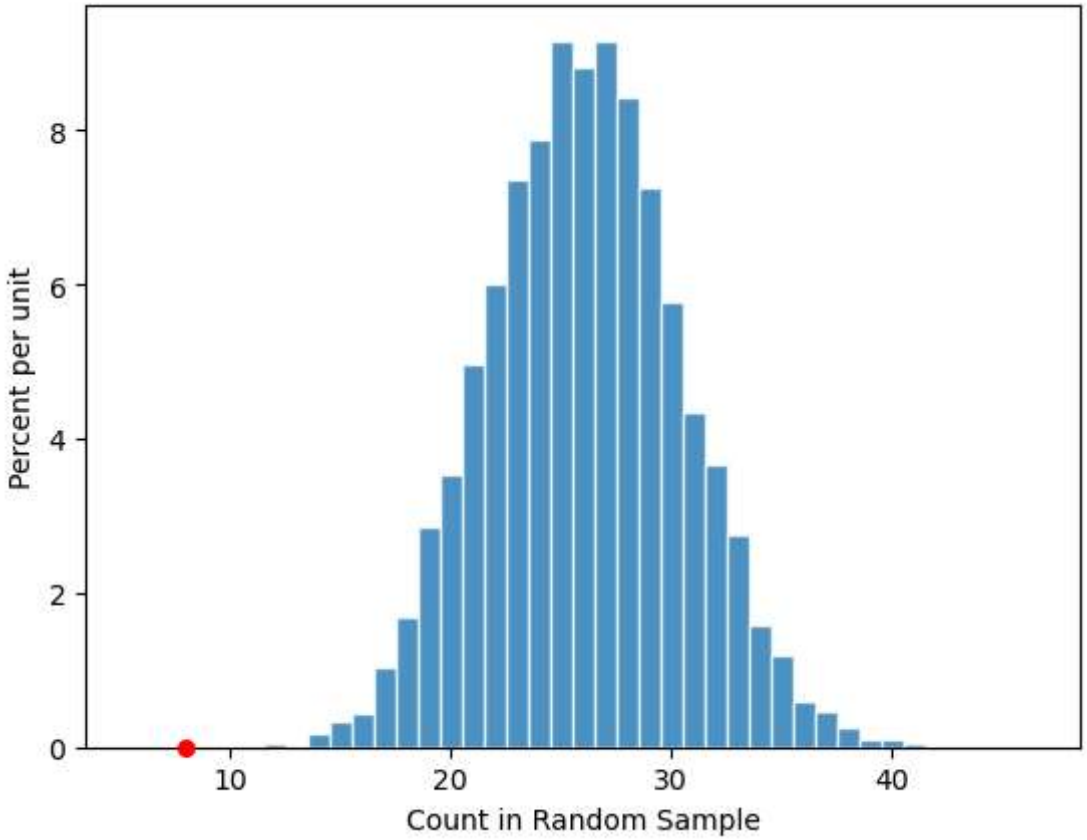


Comparing the Prediction and the Data

Though the simulated counts are quite varied, very few of them came out to be eight or less. The value eight is far out in the left hand tail of the histogram. It's the red dot on the horizontal axis of the histogram.

In [ ]:
```
bins = np.arange(5.5, 46.6, 1)
unit = ''
fig, ax = plt.subplots()
ax.hist(random_sample_counts, bins=bins, density=True, alpha=0.8, ec='white', zorder=5)
ax.scatter(8,0,color='red', s=30, zorder=10).set_clip_on(False)
y_vals = ax.get_yticks()
y_label = 'Percent per ' + (unit if unit else 'unit')
x_label = 'Count in Random Sample'
ax.set_yticklabels(['{:g}'.format(x * 100) for x in y_vals])
plt.ylabel(y_label)
plt.xlabel(x_label)
plt.title('');
plt.show()
```

```
C:\Users\91868\AppData\Local\Temp\ipykernel_14316\3139847714.py:9: UserWarning: set_ticklabels
() should only be used with a fixed number of ticks, i.e. after set_ticks() or using a FixedLo
cator.
  ax.set_yticklabels(['{:g}'.format(x * 100) for x in y_vals])
```



Mendel's Pea Flowers

The method of assessing models through simulation is general and can apply in various contexts, including jury selection and genetics. For example, in *Swain v. Alabama*, a simulation showed that selecting 100 jurors at random from a population with 26% Black members is unlikely to yield only 8 Black jurors, suggesting the jury was not randomly selected. Similarly, Gregor Mendel, in his experiments with pea plants, hypothesized a model where purple-flowering plants would occur with a probability of 75%. To test this model, we can simulate plant samples and measure the distance between the observed proportion of purple flowers and the expected 75%. Large deviations from 75% would indicate the model may not fit the observed data. This approach, which involves comparing actual data with the

results of model-based simulations, allows us to test the validity of assumptions across various scientific and legal settings.

In [ ]:
```
def distance_from_75(p):
    return abs(100*p - 75)
```

In [ ]:
```
model_proportions = [0.75, 0.25]
```

In [ ]:
```
proportion_purple_in_sample = sample_proportions(929, model_proportions).item(0)
distance_from_75(proportion_purple_in_sample)
```

Out[ ]:
```
2.28740581270182
```

Running the Simulation

To get a sense of how variable the distance could be, we have to simulate it many more times.

We will generate 10,000 values of the distance. As before, we will first use the code we developed above to define a function that returns one simulated value Mendel's hypothesis.

In [ ]:
```
def one_simulated_distance():
    proportion_purple_in_sample = sample_proportions(929, model_proportions).item(0)
    return distance_from_75(proportion_purple_in_sample)
```

Next, we will use a for loop to create 10,000 such simulated distances.

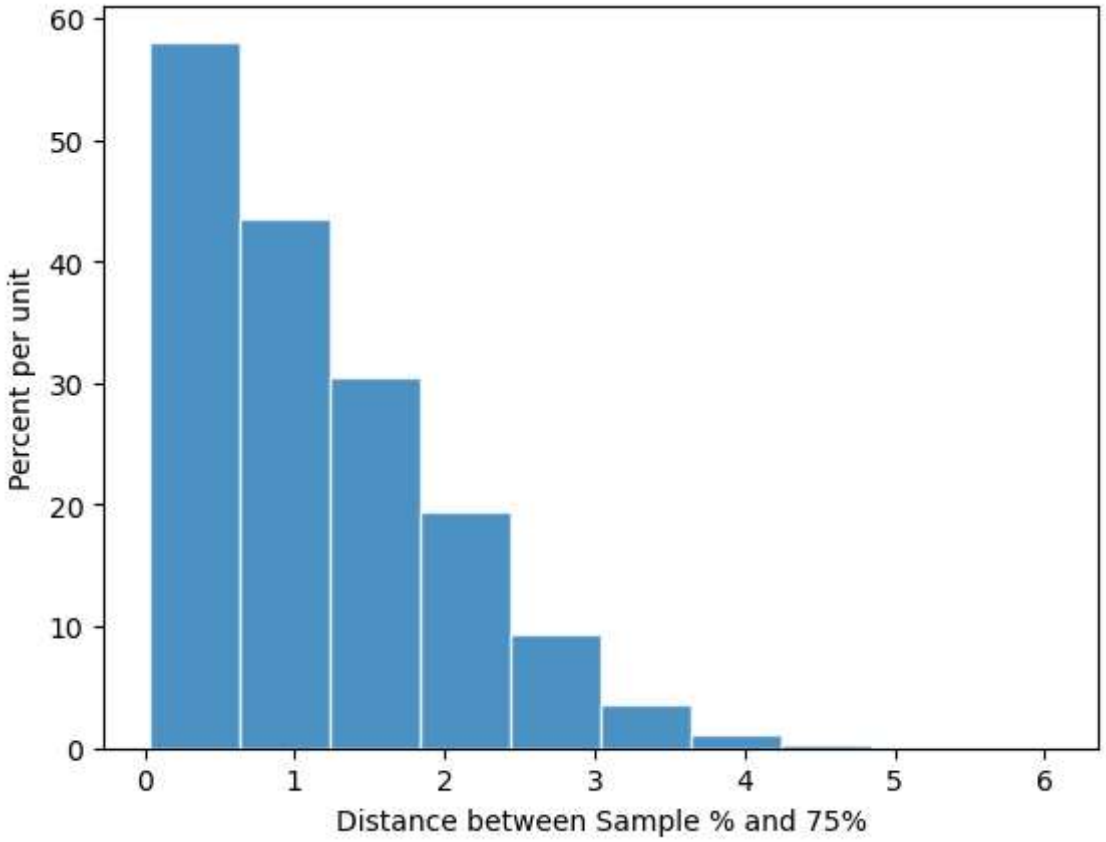
In [ ]:
```
distances = np.array([])

repetitions = 10000
for i in np.arange(repetitions):
    distances = np.append(distances, one_simulated_distance())
```

The Prediction

The empirical histogram of the simulated values shows the distribution of the distance as predicted by Mendel's model.

In [ ]:
```
sample_distance = pd.DataFrame({'Distance between Sample % and 75%':distances})
unit = ''
fig, ax = plt.subplots()
ax.hist(sample_distance, density=True, alpha=0.8, ec='white', zorder=5)
y_vals = ax.get_yticks()
y_label = 'Percent per ' + (unit if unit else 'unit')
x_label = 'Distance between Sample % and 75%'
ax.set_yticklabels(['{:g}'.format(x * 100) for x in y_vals])
plt.ylabel(y_label)
plt.xlabel(x_label)
plt.title('');
plt.show()
```

```
C:\Users\91868\AppData\Local\Temp\ipykernel_14316\1117388378.py:8: UserWarning: set_ticklabels
() should only be used with a fixed number of ticks, i.e. after set_ticks() or using a FixedLo
cator.
  ax.set_yticklabels(['{:g}'.format(x * 100) for x in y_vals])
```



Look on the horizontal axis to see the typical values of the distance, as predicted by the model. They are rather small. For example, a high proportion of the distances are in the range 0 to 1, meaning that for a high proportion of the samples, the percent of purple-flowering plants is within 1% of 75%, that is, the sample percent is in the range 74% to 76%.

Comparing the Prediction and the Data

To assess the model, we have to compare this prediction with the data. Mendel recorded the number of purple and white flowering plants. Among the 929 plants that he grew, 705 were purple flowering. That's just about 75.89%.

In [ ]:
```
705 / 929
```

Out[ ]:
```
0.7588805166846071
```

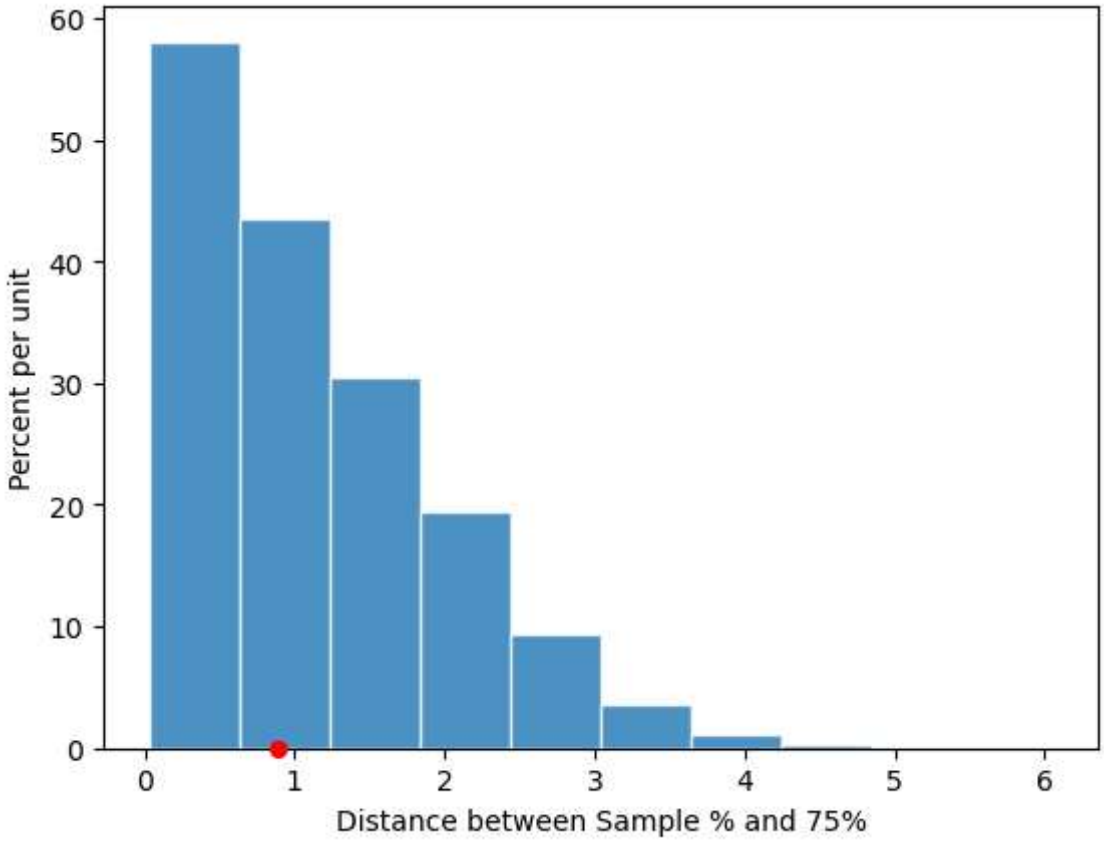
In [ ]:
```
observed_statistic = distance_from_75(705/929)
observed_statistic
```

Out[ ]:
```
0.8880516684607045
```

The cell below redraws the histogram with the observed value plotted on the horizontal axis.

In [ ]:
```
distance_0_75 = pd.DataFrame({'Distance between Sample % and 75%':distances})
unit = ''
fig, ax = plt.subplots()
ax.hist(distance_0_75, density=True, alpha=0.8, ec='white', zorder=5)
ax.scatter(observed_statistic, 0, color='red', s=30, zorder=10).set_clip_on(False)
y_vals = ax.get_yticks()
y_label = 'Percent per ' + (unit if unit else 'unit')
x_label = 'Distance between Sample % and 75%'
ax.set_yticklabels(['{:g}'.format(x * 100) for x in y_vals])
plt.ylabel(y_label)
plt.xlabel(x_label)
plt.title('');
plt.show()
```

```
C:\Users\91868\AppData\Local\Temp\ipykernel_14316\3732953736.py:9: UserWarning: set_ticklabels
() should only be used with a fixed number of ticks, i.e. after set_ticks() or using a FixedLo
cator.
  ax.set_yticklabels(['{:g}'.format(x * 100) for x in y_vals])
```



2. Multiple Categories

We have developed a way of assessing models about chance processes that generate data in two categories. The method extends to models involving data in multiple categories. The process of assessment is the same as before, the only difference being that we have to come up with a new statistic to simulate.

Let's do this in an example that addresses the same kind of question that was raised in the case of Robert Swain's jury panel. This time, the data are more recent.

Jury Selection in Alameda County

In 2010, the ACLU of Northern California released a report on jury selection in Alameda County, focusing on the ethnic composition of jury panels and highlighting potential disparities. According to California law, jury panels are meant to be random samples that represent the community's demographic makeup. However, in an analysis of 11 felony trial jury panels from 2009 to 2010, the ACLU found discrepancies in representation among ethnic groups. They reviewed data from 1,453 individuals who reported for jury service and compared the ethnic distribution in this group with that of all eligible jurors in Alameda County. The report questioned whether these jury panels truly represented a cross-section of the community and proposed reforms to ensure greater diversity among prospective jurors.

In [ ]:
```
jury = pd.DataFrame(
    {'Ethnicity':np.array(['Asian', 'Black', 'Latino', 'White', 'Other']),
    'Eligible':np.array([0.15, 0.18, 0.12, 0.54, 0.01]),
    'Panels':np.array([0.26, 0.08, 0.08, 0.54, 0.04])}
)
```

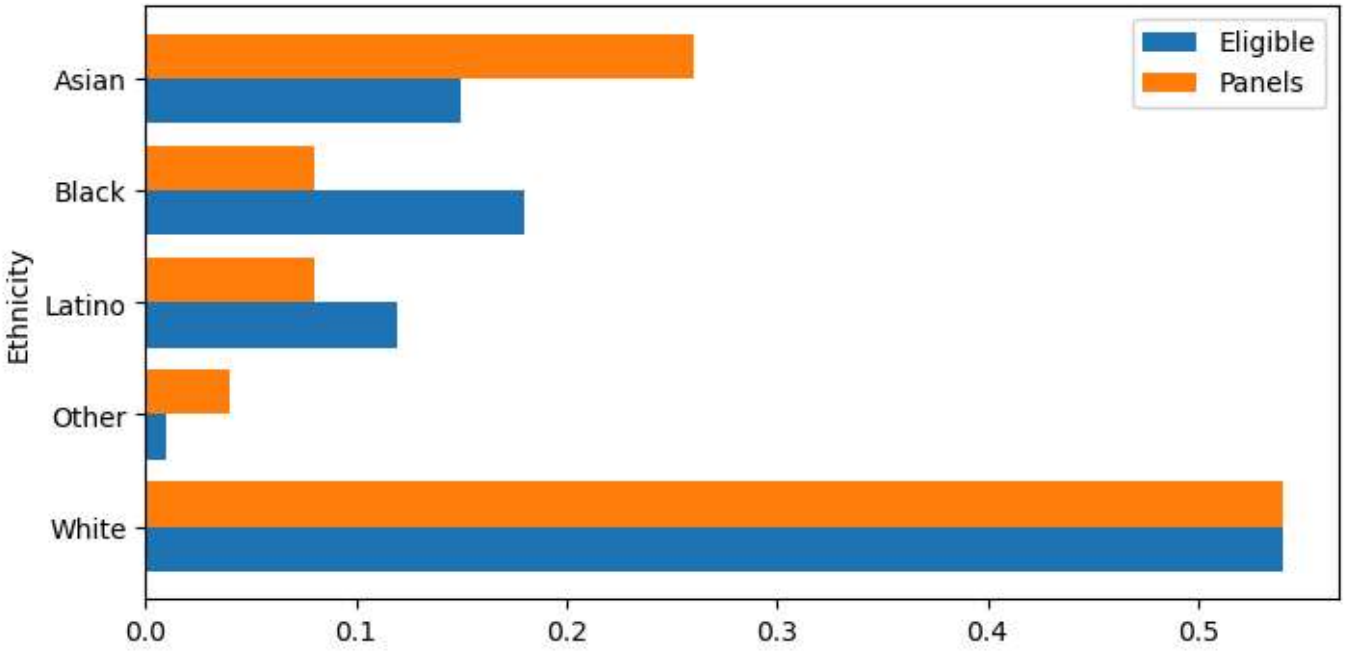
```
jury
```

Out[ ]:

	Ethnicity	Eligible	Panels
0	Asian	0.15	0.26
1	Black	0.18	0.08
2	Latino	0.12	0.08
3	White	0.54	0.54
4	Other	0.01	0.04

Some ethnicities are overrepresented and some are underrepresented on the jury panels in the study. A bar chart is helpful for visualizing the differences.

In [ ]:
```
jury = jury.sort_values(by=['Ethnicity'], ascending=False)
jury.plot.barh('Ethnicity', width=0.8, figsize=(8,4))
plt.show()
```



Comparison with Panels Selected at Random

What if we select a random sample of 1,453 people from the population of eligible jurors? Will the distribution of their ethnicities look like the distribution of the panels above?

We can answer these questions by using sample_proportions and augmenting the jury table with a column of the proportions in our sample.

Technical note. Random samples of prospective jurors would be selected without replacement. However, when the size of a sample is small relative to the size of the population, sampling without replacement resembles sampling with replacement; the proportions in the population don't change much between draws. The population of eligible jurors in Alameda County is over a million, and compared to that, a sample size of about 1500 is quite small. We will therefore sample with replacement.

In the cell below, we sample at random 1453 times from the distribution of eligible jurors, and display the distribution of the random sample along with the distributions of the eligible jurors and the panel in the data.

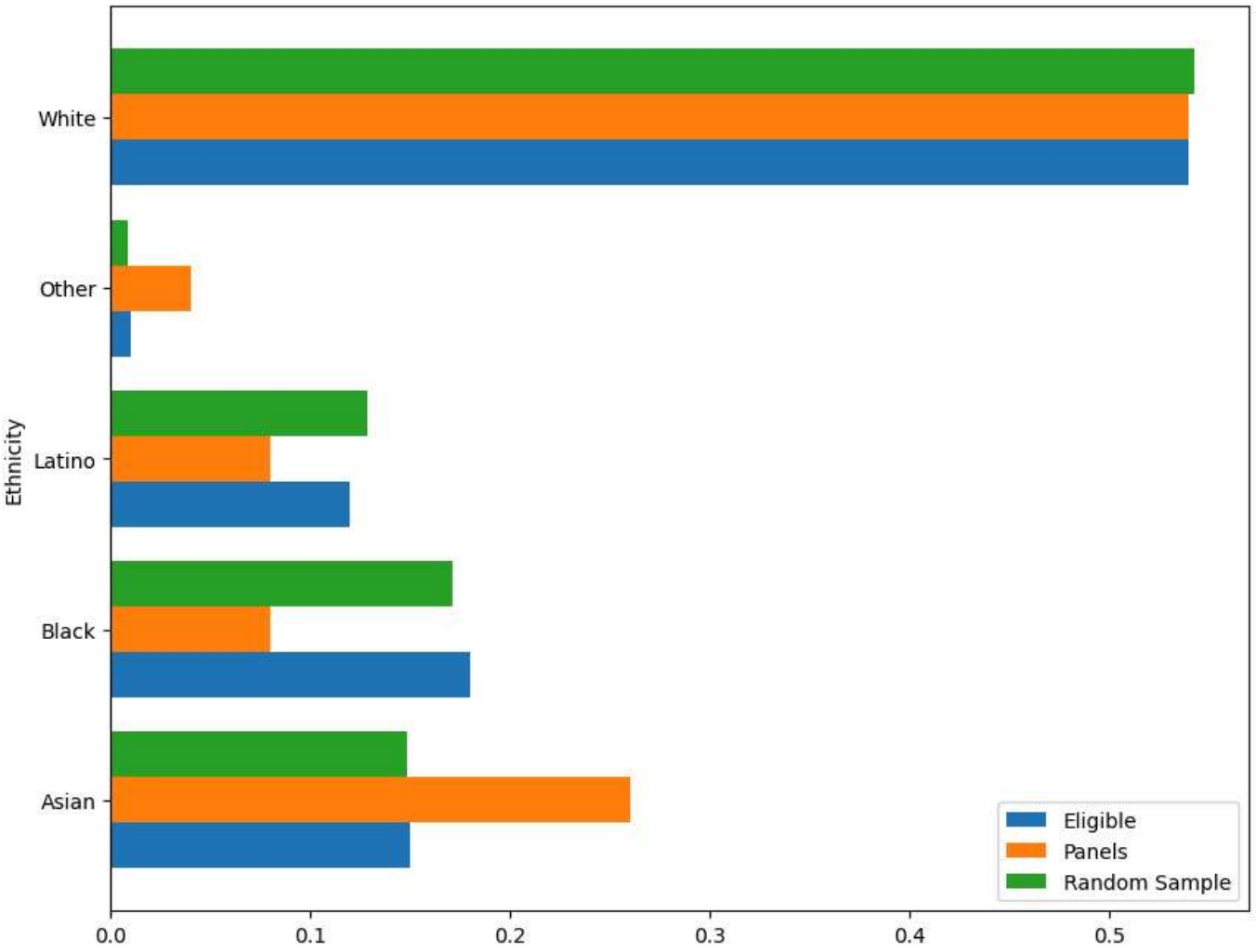
In [ ]:
```
def sample_proportions(sample_size, probabilities):
    return np.random.multinomial(sample_size, probabilities) / sample_size
```

In [ ]:
```
jury1 = jury.copy()

eligible_population = jury1['Eligible']
sample_distribution = sample_proportions(1453, eligible_population)

panels_and_sample = jury1
panels_and_sample['Random Sample'] = sample_distribution
panels_and_sample = panels_and_sample.sort_values(['Ethnicity'], ascending=True)
panels_and_sample
```

Out[ ]:

	Ethnicity	Eligible	Panels	Random Sample
0	Asian	0.15	0.26	0.148658
1	Black	0.18	0.08	0.171370
2	Latino	0.12	0.08	0.128699
4	Other	0.01	0.04	0.008259
3	White	0.54	0.54	0.543014

In [ ]:
```
panels_and_sample.plot.barh('Ethnicity', width=0.8, figsize=(10,8))
plt.show()
```



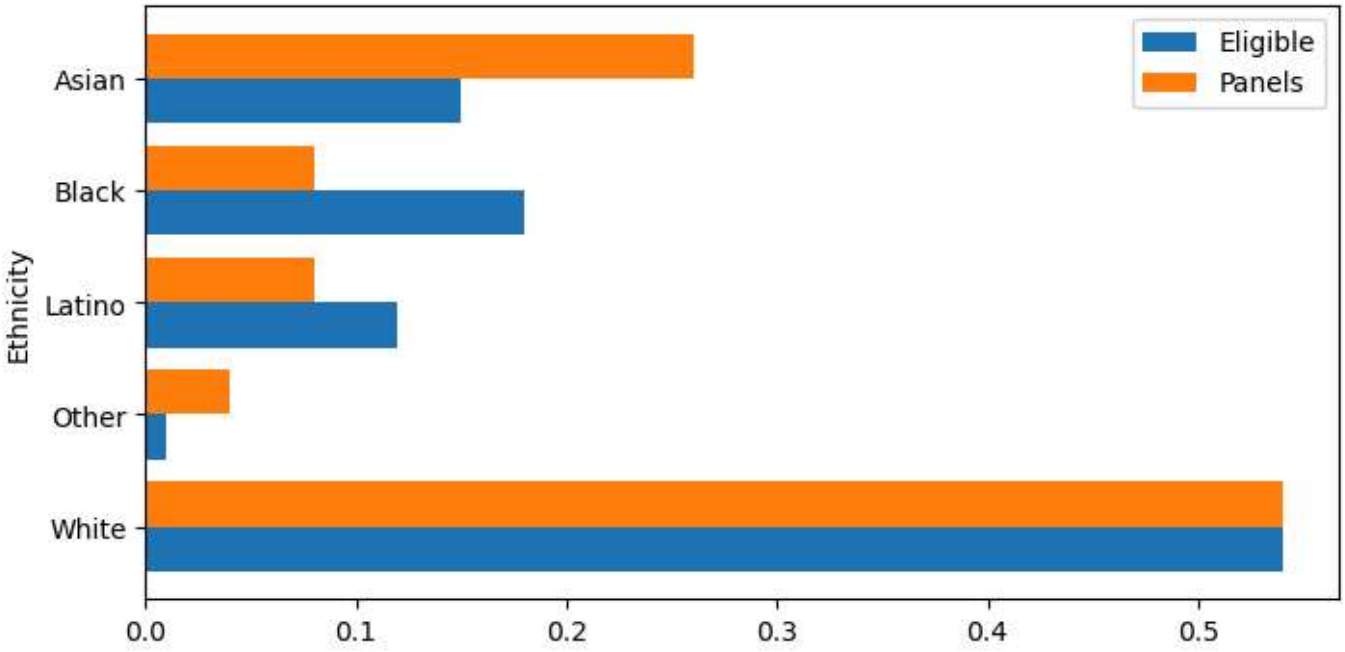
The bar chart shows that the distribution of the random sample resembles the eligible population but the distribution of the panels does not.

To assess whether this observation is particular to one random sample or more general, we can simulate multiple panels under the model of random selection and see what the simulations predict. But we won't be able to look at thousands of bar charts like the one above. We need a statistic that will help us assess whether or not the model or random selection is supported by the data.

A New Statistic: The Distance between Two Distributions

We know how to measure how different two numbers are – if the numbers are x and y , the distance between them is $|x-y|$. Now we have to quantify the distance between two distributions. For example, we have to measure the distance between the blue and gold distributions below.

In [ ]:
```
jury.plot.barh('Ethnicity', width=0.8, figsize=(8,4))
plt.show()
```



For this we will compute a quantity called the total variation distance between two distributions. The calculation is as an extension of the calculation of the distance between two numbers.

To compute the total variation distance, we first take the difference between the two proportions in each category.

In [ ]:
```
jury_with_diffs = jury.sort_values(by=['Ethnicity']).copy()
jury_with_diffs['Difference'] = jury['Panels'] - jury['Eligible']
jury_with_diffs
```

Out[ ]:

	Ethnicity	Eligible	Panels	Difference
0	Asian	0.15	0.26	0.11
1	Black	0.18	0.08	-0.10
2	Latino	0.12	0.08	-0.04
4	Other	0.01	0.04	0.03
3	White	0.54	0.54	0.00

Take a look at the column Difference and notice that the sum of its entries is 0: the positive entries add up to 0.14, exactly canceling the total of the negative entries which is -0.14.

This is numerical evidence of the fact that in the bar chart, the red bars exceed the blue bars by exactly as much as the blue bars exceed the red. The proportions in each of the two columns Panels and Eligible add up to 1, and so the give-and-take between their entries must add up to 0.

To avoid the cancellation, we drop the negative signs and then add all the entries. But this gives us two times the total of the positive entries (equivalently, two times the total of the negative entries, with the sign removed). So we divide the sum by 2.

In [ ]:
```
jury_with_diffs['Absolute Difference'] =  np.abs(jury_with_diffs['Difference'])
jury_with_diffs
```

Out[ ]:

	Ethnicity	Eligible	Panels	Difference	Absolute Difference
0	Asian	0.15	0.26	0.11	0.11
1	Black	0.18	0.08	-0.10	0.10
2	Latino	0.12	0.08	-0.04	0.04
4	Other	0.01	0.04	0.03	0.03
3	White	0.54	0.54	0.00	0.00

In [ ]:
```
jury_with_diffs['Absolute Difference'].sum() / 2
```

Out[ ]:
```
0.14
```

This quantity 0.14 is the total variation distance (TVD) between the distribution of ethnicities in the eligible juror population and the distribution in the panels.

We could have obtained the same result by just adding the positive differences. But our method of including all the absolute differences eliminates the need to keep track of which differences are positive and which are not.

Simulating One Value of the Statistic

We will use the total variation distance between distributions as the statistic to simulate. It will help us decide whether the model of random selection is good, because large values of the distance will be evidence against the model.

Keep in mind that the observed value of our statistic is 0.14, calculated above.

Since we are going to be computing total variation distance repeatedly, we will write a function to compute it.

The function total_variation_distance returns the TVD between distributions in two arrays.

In [ ]:
```
def total_variation_distance(distribution_1, distribution_2):
    return sum(np.abs(distribution_1 - distribution_2)) / 2
```

This function will help us calculate our statistic in each repetition of the simulation. But first, let's check that it gives the right answer when we use it to compute the distance between the blue (eligible) and gold (panels) distributions above.

In [ ]:
```
total_variation_distance(jury['Panels'], jury['Eligible'])
```

Out[ ]:
```
0.14
```

This agrees with the value that we computed directly without using the function.

In the cell below we use the function to compute the TVD between the distributions of the eligible jurors and one random sample. This is the code for simulating one value of our statistic. Recall that eligible_population is the array containing the distribution of the eligible jurors.

In [ ]:
```
sample_distribution = sample_proportions(1453, eligible_population)
total_variation_distance(sample_distribution, eligible_population)
```

Out[ ]:
```
0.02282174810736412
```

Notice that the distance is quite a bit smaller than 0.14, the distance between the distribution of the panels and the eligible jurors.

Predicting the Statistic Under the Model of Random Selection

The total variation distance between the distributions of the random sample and the eligible jurors is the statistic that we are using to measure the distance between the two distributions. By repeating the process of sampling, we can see how much the statistic varies across different random samples.

The code below simulates the statistic based on a large number of replications of the random sampling process, following our usual sequence of steps for simulation. We first define a function that returns one simulated value of the total variation distance under the hypothesis of random selection. Then we use our function in a for loop to create an array tvds consisting of 5,000 such distances.

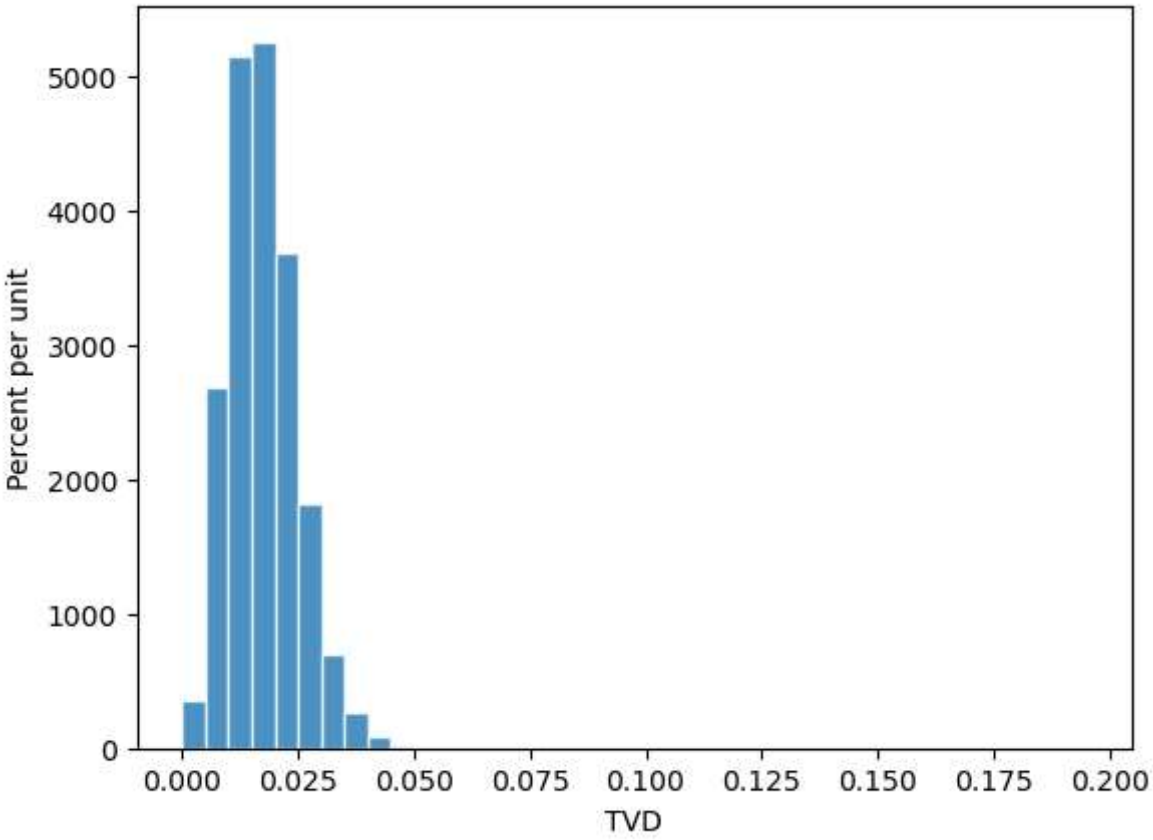
In [ ]:
```
def one_simulated_tvd():
    sample_distribution = sample_proportions(1453, eligible_population)
    return total_variation_distance(sample_distribution, eligible_population)
```

In [ ]:
```
tvds = np.array([])

repetitions = 5000
for i in np.arange(repetitions):
    tvds = np.append(tvds, one_simulated_tvd())
```

The empirical histogram of the simulated distances shows that drawing 1453 jurors at random from the pool of eligible candidates results in a distribution that rarely deviates from the eligible jurors' race distribution by more than about 0.05.

In [ ]:
```
TVD = pd.DataFrame({'TVD':tvds})
unit = ''
fig, ax1 = plt.subplots()
ax1.hist(TVD, bins=np.arange(0, 0.2, 0.005), density=True, alpha=0.8, ec='white')
y_vals = ax1.get_yticks()
y_label = 'Percent per ' + (unit if unit else 'unit')
x_label = 'TVD'
ax1.set_yticklabels(['{:g}'.format(x * 100) for x in y_vals])
plt.ylabel(y_label)
plt.xlabel(x_label)
plt.show()
```

```
C:\Users\91868\AppData\Local\Temp\ipykernel_14316\1302995910.py:8: UserWarning: set_ticklabels
() should only be used with a fixed number of ticks, i.e. after set_ticks() or using a FixedLo
cator.
  ax1.set_yticklabels(['{:g}'.format(x * 100) for x in y_vals])
```



3. Decisions and Uncertainty

Statistical hypothesis testing helps make decisions when data could be explained by chance or a meaningful effect. Outcomes are not always clear-cut, so statistical conventions and judgment are used to determine whether observed data align with the null hypothesis. Visualizations and test statistics provide evidence, but some decisions still require careful interpretation.

Step 1: The Hypotheses Formulate two hypotheses: a null hypothesis, which assumes the data is due to random chance, and an alternative hypothesis, which suggests that other factors affect the data. The null hypothesis provides a model to simulate data.

Step 2: The Test Statistic Select a statistic that will indicate which hypothesis is more likely. For example, the distance between the observed data and an expected value can serve as a test statistic; large distances suggest the data may not fit the null hypothesis.

Step 3: Distribution of the Test Statistic Under the Null Hypothesis Simulate the test statistic repeatedly under the assumptions of the null hypothesis to understand its possible values. The resulting distribution gives context to the observed value and helps determine the likelihood of observing such data by chance.

Step 4: Conclusion of the Test Compare the observed test statistic to the simulated distribution. If it's consistent, the data supports the null hypothesis. If not, the null hypothesis is rejected in favor of the alternative, suggesting other factors influenced the data.

Here is an example where the decision requires judgment.

The GSI's Defense

In a Berkeley Statistics class of 350 students, Section 3 students noticed their midterm scores were lower on average than the rest of the class, leading them to question their GSI's teaching. However, the GSI

suggested this could simply be due to random variation. To investigate, we set up hypotheses: the **null hypothesis** assumes Section 3's average score could occur by chance if a random group of students were selected, while the **alternative hypothesis** suggests Section 3's average is unusually low. Testing this hypothesis involves comparing Section 3's average to simulated averages from random samples.

The table scores contains the section number and midterm score for each student in the class. The midterm scores were integers in the range 0 through 25; 0 means that the student didn't take the test.

In [ ]:
```
scores = pd.read_csv('scores_by_section.csv')
scores
```

Out[ ]:

	Section	Midterm
0	1	22
1	2	12
2	2	23
3	2	14
4	1	20
...	...	...
354	5	24
355	2	16
356	2	17
357	12	16
358	10	14

359 rows × 2 columns

To find the average or mean score in each section, we will use groupby.

In [ ]:
```
section_averages = scores.groupby(['Section']).mean()
section_averages
```

Out[ ]:

Section	Midterm
1	15.593750
2	15.125000
3	13.666667
4	14.766667
5	17.454545
6	15.031250
7	16.625000
8	16.310345
9	14.566667
10	15.235294
11	15.807692
12	15.733333

The average score of Section 3 is 13.667, which does look low compared to the other section averages. But is it lower than the average of a section of the same size selected at random from the class?

To answer this, we can select a section at random from the class and find its average. To select a section at random to we need to know how big Section 3 is, which we can by once again using group.

In [ ]:
```
scores.groupby('Section').count()
```

Out[ ]:

Section	Midterm
1	32
2	32
3	27
4	30
5	33
6	32
7	24
8	29
9	30
10	34
11	26
12	30

Section 3 had 27 students.

Now we can figure out how to create one simulated value of our test statistic, the random sample average.

First we have to select 27 scores at random without replacement. Since the data are already in a table, we will use the Table method sample.

Remember that by default, sample draws with replacement. The optional argument with_replacement = False produces a random sample drawn without replacement.

In [ ]:
```
scores_only = scores.drop(columns=['Section'])
```

In [1]:
```
random_sample = scores_only.sample(27, replace=False)
random_sample.head(2)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-1-2d150f2c0351> in <cell line: 1>()
----> 1 random_sample = scores_only.sample(27, replace=False)
      2 random_sample.head(2)

NameError: name 'scores_only' is not defined
```

In [ ]:
```
np.average(random_sample['Midterm'])
```

Out[ ]:
```
15.962962962962964
```

That's the average of 27 randomly selected scores.

The cell below collects the code necessary for generating this random average.

Now we can simulate the random sample average by repeating the calculation multple times.

In [ ]:
```
def random_sample_average():
    random_sample = scores_only.sample(27, replace=False)
    return np.average(random_sample['Midterm'])
```

In [ ]:
```
sample_averages = np.array([])

repetitions = 10000
for i in np.arange(repetitions):
    sample_averages = np.append(sample_averages, random_sample_average())

sample_averages
```

Out[ ]:
```
array([15.2962963 , 15.59259259, 16.62962963, ..., 17.37037037,
       14.03703704, 13.59259259])
```

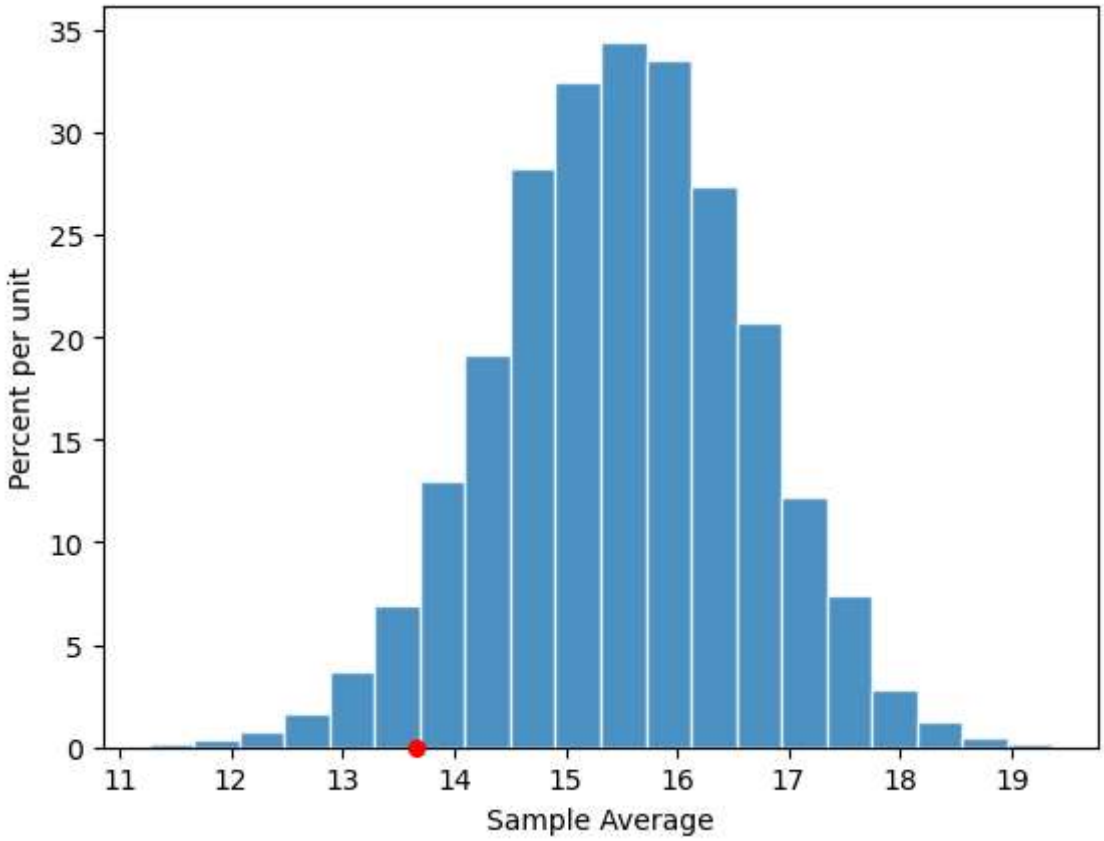
Here is the histogram of the simulated averages. It shows the distribution of what the Section 3 average might have been, if Section 3 had been selected at random from the class.

The observed Section 3 average score of 13.667 is shown as a red dot on the horizontal axis. You can ignore the last line of code; it just draws the dot.

In [ ]:
```
averages_tbl = pd.DataFrame({'Sample Average':sample_averages})
observed_statistic = 13.667
unit = ''
fig, ax = plt.subplots()
ax.hist(averages_tbl, bins=20,density=True, alpha=0.8, ec='white', zorder=5)
ax.scatter(observed_statistic, 0, color='red', s=30, zorder=10).set_clip_on(False)
y_vals = ax.get_yticks()
y_label = 'Percent per ' + (unit if unit else 'unit')
```

```
x_label = 'Sample Average'
ax.set_yticklabels(['{:g}'.format(x * 100) for x in y_vals])
plt.ylabel(y_label)
plt.xlabel(x_label)
plt.title('');
plt.show()
```

```
C:\Users\91868\AppData\Local\Temp\ipykernel_14316\3449540556.py:10: UserWarning: set_ticklabel
s() should only be used with a fixed number of ticks, i.e. after set_ticks() or using a FixedL
ocator.
  ax.set_yticklabels(['{:g}'.format(x * 100) for x in y_vals])
```



4. Error Probabilities

In the process by which we decide which of two hypotheses is better supported by our data, the final step involves a judgment about the consistency of the data and the null hypothesis. While this step results in a good decision a vast majority of the time, it can sometimes lead us astray. The reason is chance variation. For example, even when the null hypothesis is true, chance variation might cause the sample to look quite different from what the null hypothesis predicts.

Wrong Conclusions

The Chance of an Error : Suppose you want to test whether a coin is fair or not. Then the hypotheses are:

Null: The coin is fair. That is, the results are like draws made at random with replacement from Heads, Tails.

Alternative: The coin is not fair.

Suppose you are going to test this hypothesis based on 2000 tosses of the coin. You would expect a fair coin to land heads 1000 times out of 2000, so a reasonable test statistic to use is

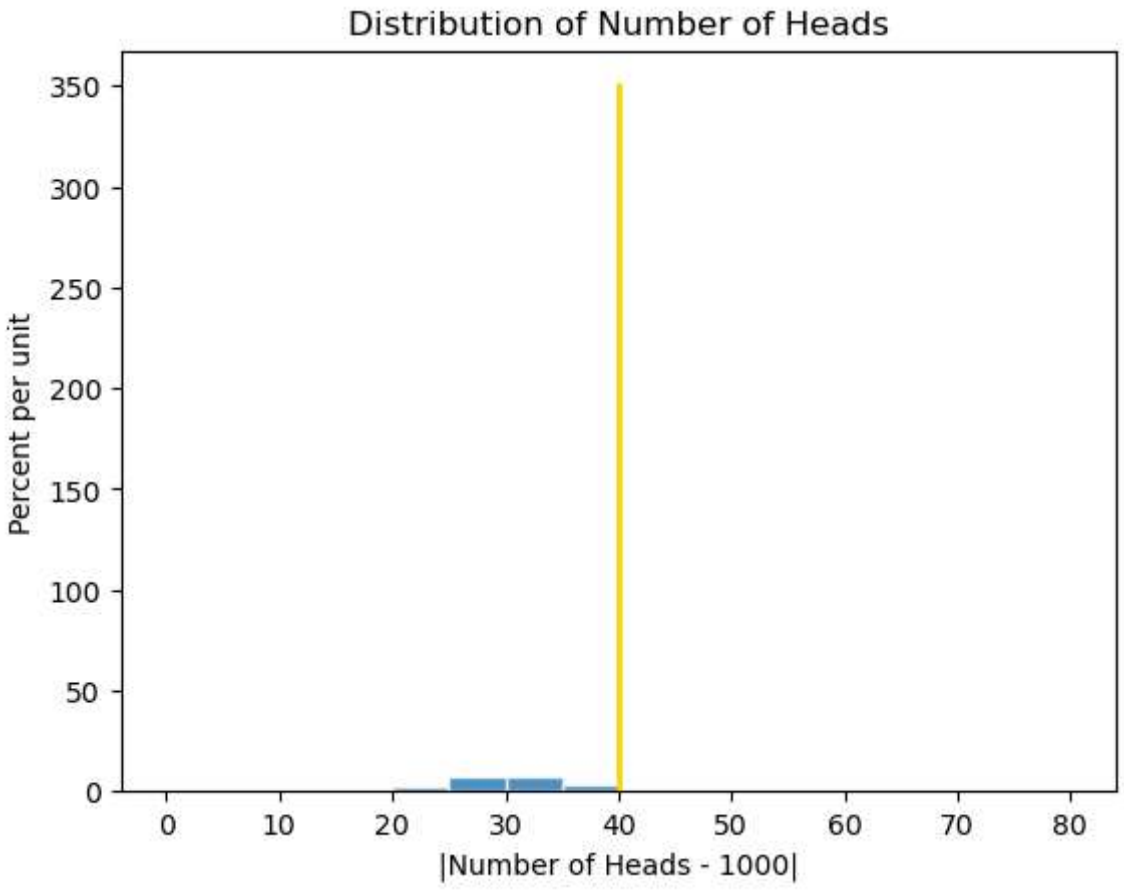
test statistic = |number of heads − 1000|

Small values of this statistic favor the null hypothesis, and large values favor the alternative.

We have simulated this statistic under the null hypothesis many times, and drawn its empirical distribution.

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

np.random.seed(0)
statistics = np.random.binomial(100, 0.3, 1000)

results = pd.DataFrame({'|Number of Heads - 1000|': statistics})
unit = ''

fig, ax = plt.subplots()
ax.hist(results['|Number of Heads - 1000|'], bins=np.arange(0, 81, 5), density=True, alpha=0.
ax.plot([40, 40], [0, 3.5], color='gold', lw=2, zorder=10)

y_vals = ax.get_yticks()
y_label = 'Percent per ' + (unit if unit else 'unit')
x_label = '|Number of Heads - 1000|'
ax.set_yticklabels(['{:g}'.format(x * 100) for x in y_vals])
plt.ylabel(y_label)
plt.xlabel(x_label)
plt.title('Distribution of Number of Heads')
plt.show()
```

```
C:\Users\91868\AppData\Local\Temp\ipykernel_14316\2073813211.py:18: UserWarning: set_ticklabel
s() should only be used with a fixed number of ticks, i.e. after set_ticks() or using a FixedL
ocator.
  ax.set_yticklabels(['{:g}'.format(x * 100) for x in y_vals])
```



The area to the right of 45 (where the gold line is) is about 5%. Large values of the test statistic favor the alternative. So if the test statistic comes out to be 45 or more, the test will conclude that the coin is

unfair.

However, as the figure shows, a fair coin can produce test statistics with values 45 or more. In fact it does so with chance about 5%.

So if the coin is fair and our test uses a 5% cutoff for deciding whether it is fair or not, then there is about a 5% chance that the test will wrongly conclude that the coin is unfair.

Foundations Of Data Science

Name: Krishna GSVV
Roll no. AV.EN.U4CSE22016

Lab 11 (Comparing Two Samples)

A/B Testing

In modern data analytics, deciding whether two numerical samples come from the same underlying distribution is called A/B testing. The name refers to the labels of the two samples, A and B.

We will develop the method in the context of an example. The data come from a sample of newborns in a large hospital system. We will treat it as if it were a simple random sample though the sampling was done in multiple stages. Stat Labs by Deborah Nolan and Terry Speed has details about a larger dataset from which this set is drawn.

Smokers and Nonsmokers

The table births contains the following variables for 1,174 mother-baby pairs: the baby's birth weight in ounces, the number of gestational days, the mother's age in completed years, the mother's height in inches, pregnancy weight in pounds, and whether or not the mother smoked during pregnancy.

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

In [ ]:
```
births = pd.read_csv('baby.csv')
births.head()
```

Out[ ]:

	Birth Weight	Gestational Days	Maternal Age	Maternal Height	Maternal Pregnancy Weight	Maternal Smoker
0	120	284	27	62	100	False
1	113	282	33	64	135	False
2	128	279	28	64	115	True
3	108	282	23	67	125	True
4	136	286	25	62	93	False

One of the aims of the study was to see whether maternal smoking was associated with birth weight. Let's see what we can say about the two variables. We'll start by selecting just Birth Weight and Maternal

Smoker. There are 715 non-smokers among the women in the sample, and 459 smokers.

In [ ]:
```
smoking_and_birthweight = births[['Maternal Smoker', 'Birth Weight']]
```

In [ ]:
```
smoking_birthweight1 = smoking_and_birthweight.groupby(["Maternal Smoker"]).agg(
    count=pd.NamedAgg(column="Maternal Smoker", aggfunc="count")
)
smoking_birthweight1.reset_index()
```

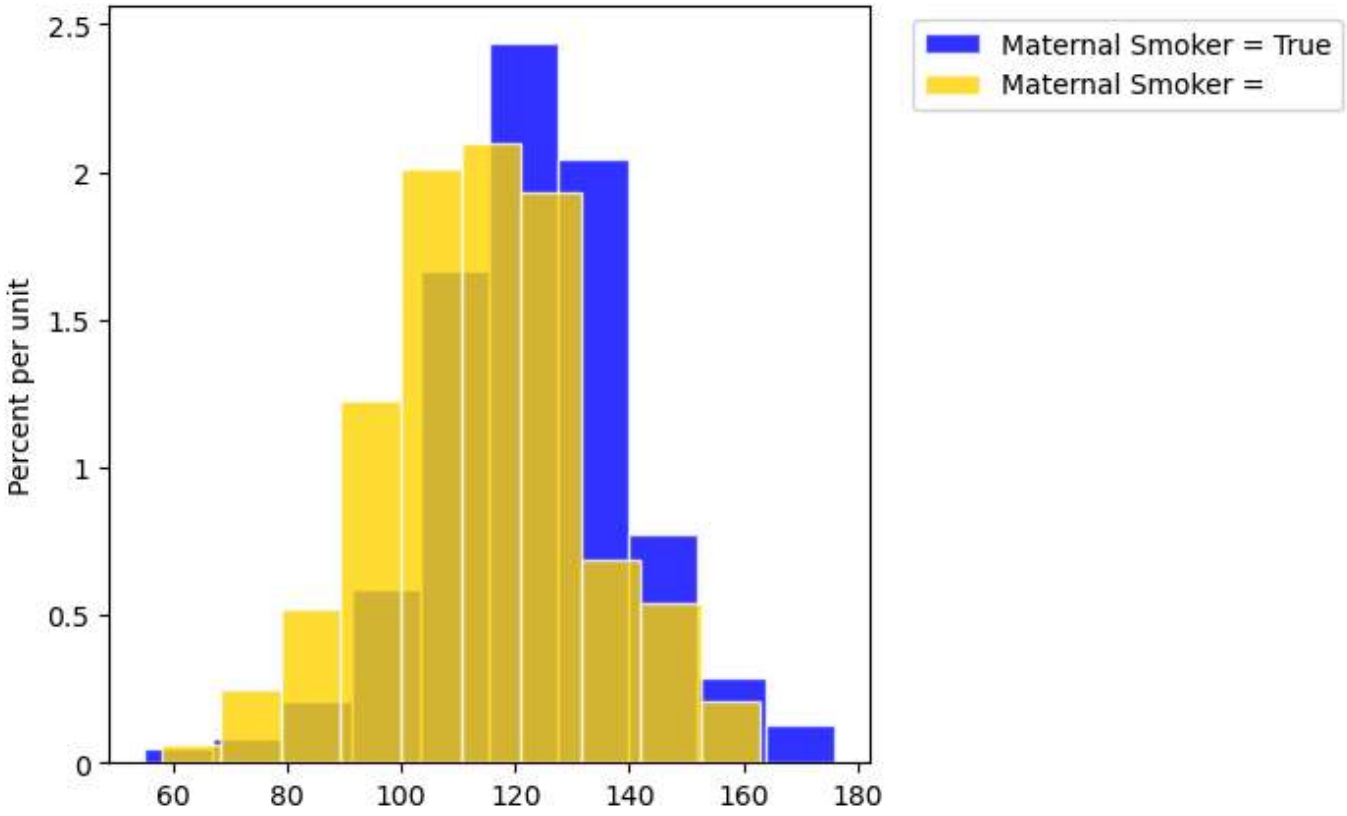
Out[ ]:

	Maternal Smoker	count
0	False	715
1	True	459

In [ ]:
```
smoker = births[births['Maternal Smoker'] == False]
non_smoker = births[births['Maternal Smoker'] == True]
```

Let's look at the distribution of the birth weights of the babies of the non-smoking mothers compared to those of the smoking mothers. To generate two overlaid histograms, we will use hist with the optional group argument which is a column label or index. The rows of the table are first grouped by this column and then a histogram is drawn for each one.

In [ ]:
```
import warnings
warnings.filterwarnings("ignore")

unit = ''
fig, ax = plt.subplots(figsize=(5,5))
ax.hist(smoker['Birth Weight'], density=True,
        label='Maternal Smoker = True',
        color='blue',
        alpha=0.8,
        ec='white',
        zorder=5)
ax.hist(non_smoker['Birth Weight'], density=True,
        label='Maternal Smoker = ',
        color='gold',
        alpha=0.8,
        ec='white',
        zorder=10)
y_vals = ax.get_yticks()
y_label = 'Percent per ' + (unit if unit else 'unit')
x_label = ''
ax.set_yticklabels(['{:g}'.format(x * 100) for x in y_vals])
plt.ylabel(y_label)
plt.xlabel(x_label)
plt.title('')
ax.legend(bbox_to_anchor=(1.04,1), loc="upper left")
plt.show()
```



The distribution of the weights of the babies born to mothers who smoked appears to be based slightly to the left of the distribution corresponding to non-smoking mothers. The weights of the babies of the mothers who smoked seem lower on average than the weights of the babies of the non-smokers.

The Hypotheses We can try to answer this question by a test of hypotheses. The chance model that we will test says that there is no underlying difference in the popuations; the distributions in the samples are different just due to chance.

Formally, this is the null hypothesis. We are going to have to figure out how to simulate a useful statistic under this hypothesis. But as a start, let's just state the two natural hypotheses.

Null hypothesis: In the population, the distribution of birth weights of babies is the same for mothers who don't smoke as for mothers who do. The difference in the sample is due to chance.

Alternative hypothesis: In the population, the babies of the mothers who smoke have a lower birth weight, on average, than the babies of the non-smokers.

Test Statistic : The alternative hypothesis compares the average birth weights of the two groups and says that the average for the mothers who smoke is smaller. Therefore it is reasonable for us to use the difference between the two group means as our statistic.

We will do the subtraction in the order "average weight of the smoking group − average weight of the non-smoking group". Small values (that is, large negative values) of this statistic will favor the alternative hypothesis.

The observed value of the test statistic is about −9.27 ounces.

In [ ]:
```
means_table = smoking_and_birthweight.groupby('Maternal Smoker').mean()
means_table = means_table.reset_index()
means_table
```

Out[ ]:

	Maternal Smoker	Birth Weight
0	False	123.085315
1	True	113.819172

In [ ]:
```
means = means_table['Birth Weight']
observed_difference = means[1] - means[0]
observed_difference
```

Out[ ]:
```
-9.266142572024918
```

We are going compute such differences repeatedly in our simulations below, so we will define a function to do the job. The function takes three arguments: the name of the table of data the label of the column that contains the numerical variable whose average is of interest the label of the column that contains the Boolean variable for grouping It returns the difference between the means of the True group and the False group.

In [ ]:
```
def difference_of_means(table, label, group_label):
    reduced = table[[label, group_label]]
    means_table = reduced.groupby(group_label).mean()
    means = means_table[label]
    return means[1] - means[0]
```

In [ ]:
```
difference_of_means(births, 'Birth Weight', 'Maternal Smoker')
```

Out[ ]:
```
-9.266142572024918
```

That's the same as the value of observed_difference calculated earlier.

Predicting the Statistic Under the Null Hypothesis

In [ ]:
```
smoking_and_birthweight.head()
```

Out[ ]:

	Maternal Smoker	Birth Weight
0	False	120
1	False	113
2	True	128
3	True	108
4	False	136

There are 1,174 rows in the table. To shuffle all the labels, we will draw a random sample of 1,174 rows without replacement. Then the sample will include all the rows of the table, in random order.

We can use the Table method sample with the optional with_replacement=False argument. We don't have to specify a sample size, because by default, sample draws as many times as there are rows in the table.

In [ ]:
```
smoking_and_birthweight2 = smoking_and_birthweight.copy()
shuffled_labels = smoking_and_birthweight2.sample(len(smoking_and_birthweight2), replace = Fa
shuffled_labels = np.array(shuffled_labels['Maternal Smoker'])
smoking_and_birthweight2['Shuffled Label'] = shuffled_labels
original_and_shuffled = smoking_and_birthweight2
```

In [ ]:
```
original_and_shuffled.head()
```

Out[ ]:

	Maternal Smoker	Birth Weight	Shuffled Label
0	False	120	False
1	False	113	False
2	True	128	False
3	True	108	False
4	False	136	True

Each baby's mother now has a random smoker/non-smoker label in the column Shuffled Label, while her original label is in Maternal Smoker. If the null hypothesis is true, all the random re-arrangements of the labels should be equally likely.

Let's see how different the average weights are in the two randomly labeled groups.

In [ ]:
```
shuffled_only = original_and_shuffled.drop(columns=['Maternal Smoker'])
shuffled_group_means = shuffled_only.groupby('Shuffled Label').mean()
shuffled = shuffled_group_means.reset_index()
shuffled.head()
```

Out[ ]:

	Shuffled Label	Birth Weight
0	False	119.587413
1	True	119.267974

The averages of the two randomly selected groups are quite a bit closer than the averages of the two original groups. We can use our function difference_of_means to find the two differences.

In [ ]:
```
import warnings
warnings.filterwarnings("ignore")

difference_of_means(original_and_shuffled, 'Birth Weight', 'Shuffled Label')
```

Out[ ]:
```
-0.3194387312034337
```

In [ ]:
```
import warnings
warnings.filterwarnings("ignore")

difference_of_means(original_and_shuffled, 'Birth Weight', 'Maternal Smoker')
```

Out[ ]:
```
-9.266142572024918
```

In [ ]:
```
def one_simulated_difference(table, label, group_label):
    births1 = table.copy()
    shuffled_labels = births1.sample(len(births1), replace = False)
    shuffled_labels = np.array(shuffled_labels[group_label])
    births1['Shuffled Label'] = shuffled_labels
    original_and_shuffled = births1
    shuffled_only = original_and_shuffled.drop(columns=['Maternal Smoker'])
    shuffled_group_means = shuffled_only.groupby('Shuffled Label').mean()
    table1 = shuffled_group_means.reset_index()
    return difference_of_means(table1, label, 'Shuffled Label')
```

In [ ]:
```
import warnings
warnings.filterwarnings("ignore")
```

```
one_simulated_difference(births, 'Birth Weight', 'Maternal Smoker')
```

Out[ ]:
```
1.2008927891280905
```

Permutation Test

Tests based on random permutations of the data are called permutation tests. We are performing one in this example. In the cell below, we will simulate our test statistic – the difference between the averages of the two groups – many times and collect the differences in an array.

In [ ]:
```
import warnings
warnings.filterwarnings("ignore")

differences = []

repetitions = 5000
for i in range(repetitions):
    new_difference = one_simulated_difference(births, 'Birth Weight', 'Maternal Smoker')
    differences.append(new_difference)

differences = np.array(differences)
differences
```

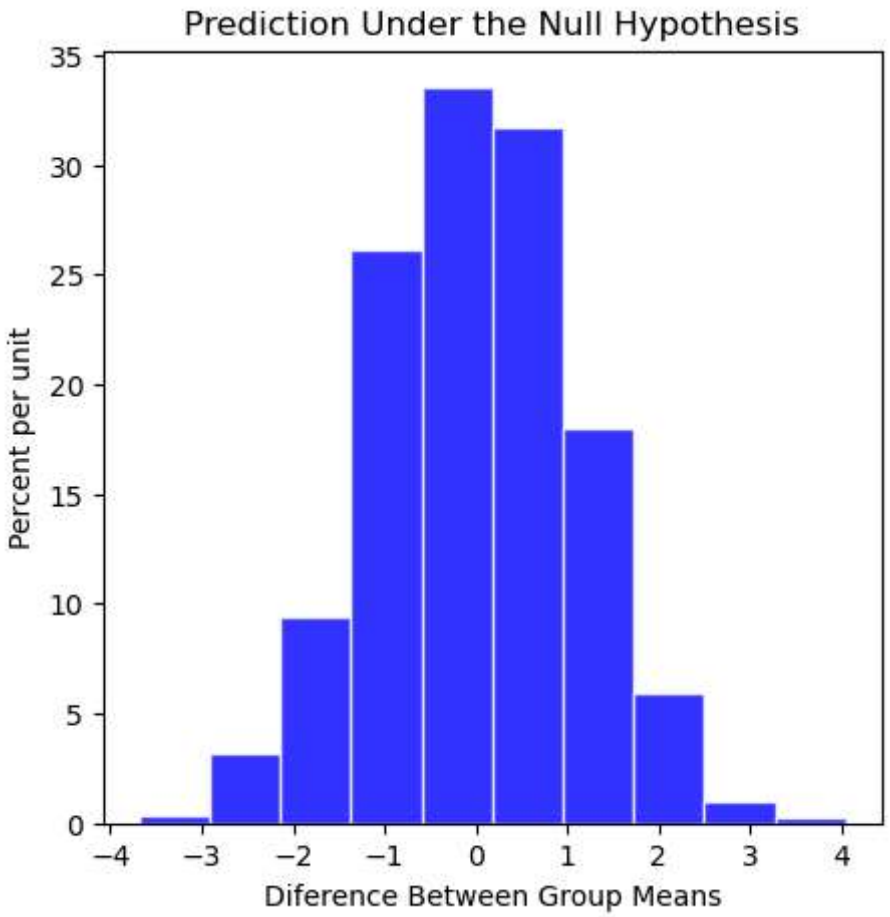
Out[ ]:
```
array([ 0.31015738,  1.2724378 ,  0.57487393, ..., -0.65212304,
       -2.85570943,  0.19568536])
```

Conclusion of the Test

The histogram below shows the distribution of these 5,000 values. It is the empirical distribution of the test statistic simulated under the null hypothesis. This is a prediction about the test statistic, based on the null hypothesis.

In [ ]:
```
import warnings
warnings.filterwarnings("ignore")

test_conclusion = pd.DataFrame({'Difference Between Group Means':differences})
print('Observed Difference:', observed_difference)
unit = ''
fig, ax = plt.subplots(figsize=(5,5))
ax.hist(test_conclusion, density=True, color='blue', alpha=0.8, ec='white')
y_vals = ax.get_yticks()
y_label = 'Percent per ' + (unit if unit else 'unit')
x_label = 'Diference Between Group Means'
ax.set_yticklabels(['{:g}'.format(x * 100) for x in y_vals])
plt.ylabel(y_label)
plt.xlabel(x_label)
plt.title('Prediction Under the Null Hypothesis');
plt.show()
```

```
Observed Difference: -9.266142572024918
```



In [ ]:
```
empirical_P = np.count_nonzero(differences <= observed_difference) / repetitions
empirical_P
```

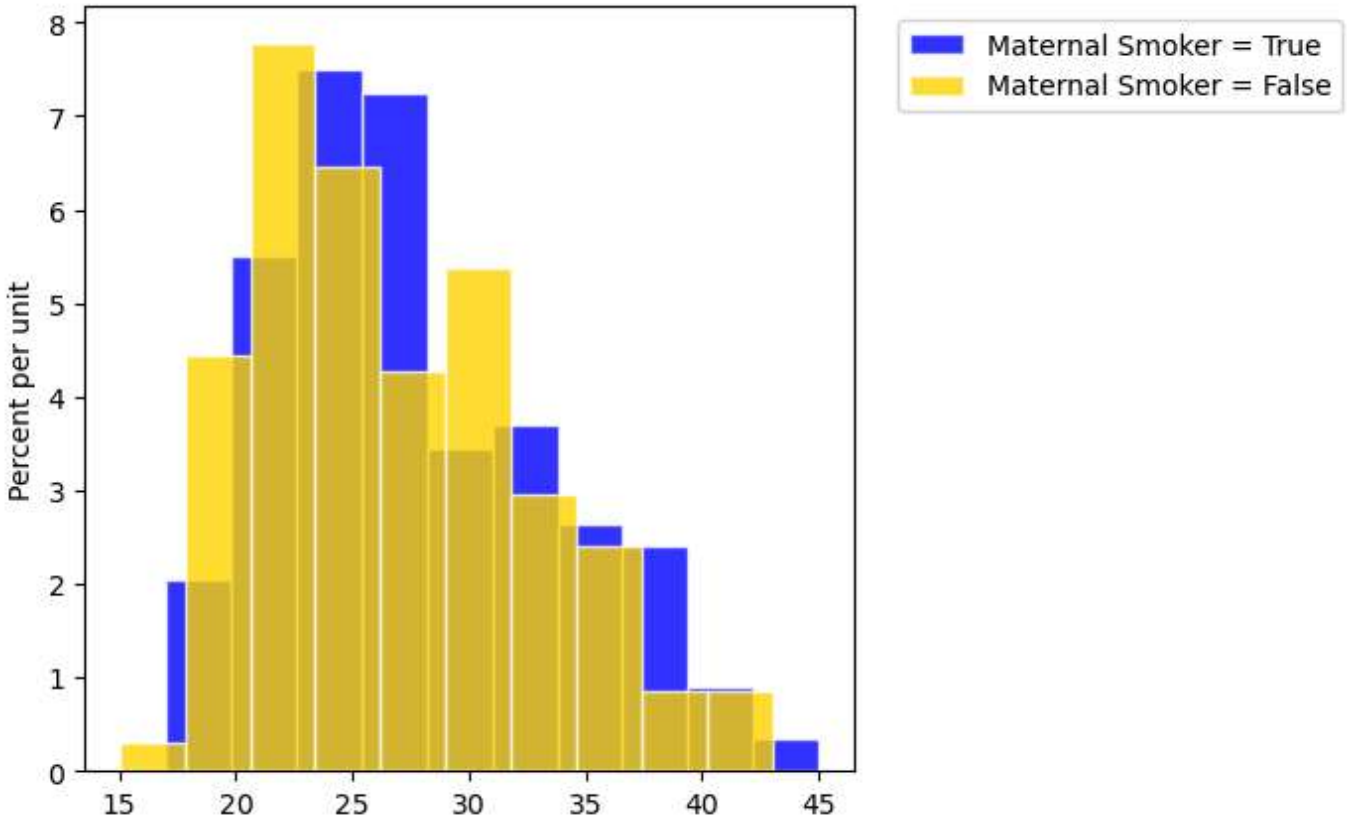
Out[ ]:
```
0.0
```

The empirical P-value is 0, meaning that none of the 5,000 permuted samples resulted in a difference of -9.27 or lower. This is only an approximation. The exact chance of getting a difference in that range is not 0 but it is vanishingly small.

Another Permutation Test

We can use the same method to compare other attributes of the smokers and the non-smokers, such as their ages. Histograms of the ages of the two groups show that in the sample, the mothers who smoked tended to be younger.

In [ ]:
```
import warnings
warnings.filterwarnings("ignore")

smoking_and_age = births[['Maternal Smoker', 'Maternal Age']]
unit = ''
fig, ax = plt.subplots(figsize=(5,5))
ax.hist(smoker['Maternal Age'], density=True, label='Maternal Smoker = True', color='blue', a
ax.hist(non_smoker['Maternal Age'], density=True, label='Maternal Smoker = False', color='gol
y_vals = ax.get_yticks()
y_label = 'Percent per ' + (unit if unit else 'unit')
x_label = ''
ax.set_yticklabels(['{:g}'.format(x * 100) for x in y_vals])
plt.ylabel(y_label)
plt.xlabel(x_label)
plt.title('')
ax.legend(bbox_to_anchor=(1.04,1), loc="upper left")
plt.show()
```



The observed difference between the average ages is about −0.8 years.

In [ ]:
```
import warnings
warnings.filterwarnings("ignore")

observed_age_difference = difference_of_means(births, 'Maternal Age', 'Maternal Smoker')
observed_age_difference
```

Out[ ]:
```
-0.8076725017901509
```

Remember that the difference is calculated as the mean age of the smokers minus the mean age of the non-smokers. The negative sign shows that the smokers are younger on average.

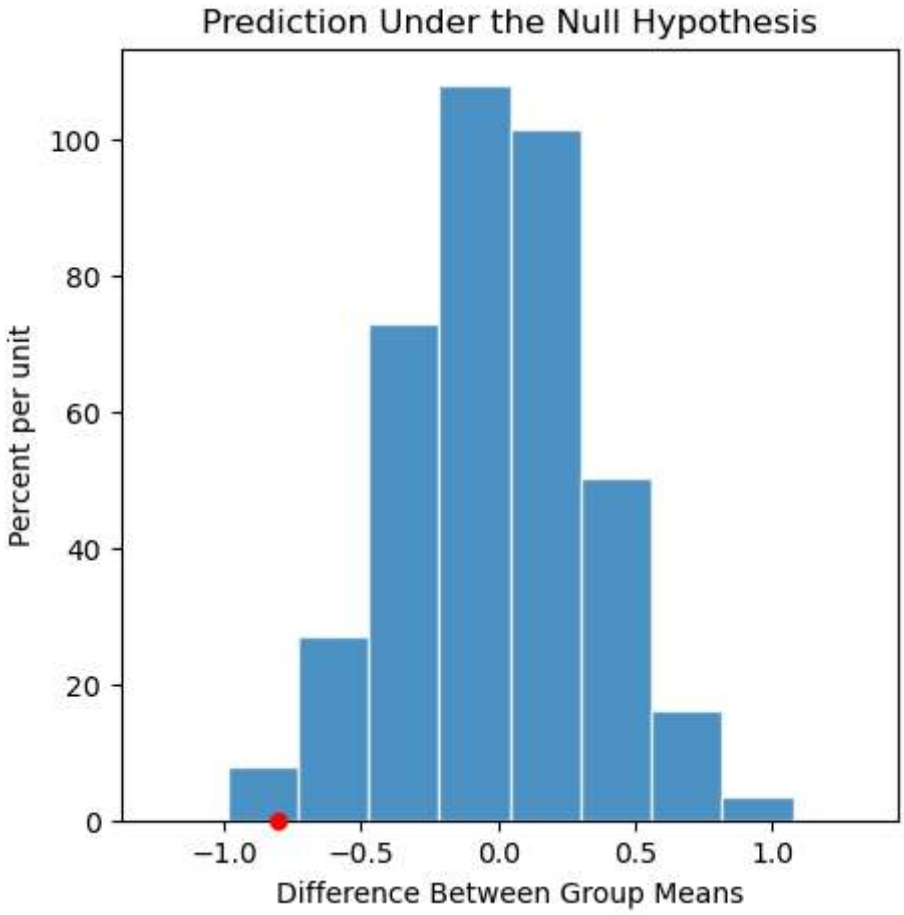
In [ ]:
```
import warnings
warnings.filterwarnings("ignore")

age_differences = np.array([])

repetitions = 5000
for i in np.arange(repetitions):
    new_difference = one_simulated_difference(births, 'Maternal Age', 'Maternal Smoker')
    age_differences = np.append(age_differences, new_difference)
```

The observed difference is in the tail of the empirical distribution of the differences simulated under the null hypothesis.

In [ ]:
```
import warnings
warnings.filterwarnings("ignore")

mean_differences = pd.DataFrame({'Difference Between Group Means':age_differences})
unit = ''
print('Observed Difference:', observed_age_difference)
fig, ax = plt.subplots(figsize=(5,5))
ax.hist(mean_differences, density=True, alpha=0.8, ec='white', zorder=5)
ax.scatter(observed_age_difference, 0, color='red', s=30, zorder=10).set_clip_on(False)
y_vals = ax.get_yticks()
y_label = 'Percent per ' + (unit if unit else 'unit')
```

```
x_label = 'Difference Between Group Means'
ax.set_yticklabels(['{:g}'.format(x * 100) for x in y_vals])
plt.ylabel(y_label)
plt.xlabel(x_label)
plt.title('Prediction Under the Null Hypothesis');
plt.show()
```

```
Observed Difference: -0.8076725017901509
```



The empirical P-value of the test is the proportion of simulated differences that were equal to or less than the observed difference. This is because low values of the difference favor the alternative hypothesis that the smokers were younger on average.

empirical_P = np.count_nonzero(age_differences <= observed_age_difference) / 5000 empirical_P

The empirical P-value is around 1% and therefore the result is statistically significant. The test supports the hypothesis that the smokers were younger on average.

2.Deflategate

Here are the data. Each row corresponds to one football. Pressure is measured in psi. The Patriots ball that had been intercepted by the Colts was not inspected at half-time. Nor were most of the Colts' balls – the officials simply ran out of time and had to relinquish the balls for the start of second half play.

In [ ]:
```
football = pd.read_csv('deflategate.csv')
football.head()
```

Out[ ]:

	Team	Blakeman	Prioleau
0	Patriots	11.50	11.80
1	Patriots	10.85	11.20
2	Patriots	11.15	11.50
3	Patriots	10.70	11.00
4	Patriots	11.10	11.45

For each of the 15 balls that were inspected, the two officials got different results. It is not uncommon that repeated measurements on the same object yield different results, especially when the measurements are performed by different people. So we will assign to each the ball the average of the two measurements made on that ball.

In [ ]:
```
football_1 = football.copy()
football_1['Combined'] = (football_1['Blakeman']+football_1['Prioleau'])/2
football_combined = football_1.drop(columns=['Blakeman','Prioleau'])
football_combined.head()
```

Out[ ]:

	Team	Combined
0	Patriots	11.650
1	Patriots	11.025
2	Patriots	11.325
3	Patriots	10.850
4	Patriots	11.275

In [ ]:
```
np.ones(11)
```

Out[ ]:
```
array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
```

In [ ]:
```
patriots_start = 12.5 * np.ones(11)
colts_start = 13 * np.ones(4)
start = np.append(patriots_start, colts_start)
start
```

Out[ ]:
```
array([12.5, 12.5, 12.5, 12.5, 12.5, 12.5, 12.5, 12.5, 12.5, 12.5, 12.5,
       13. , 13. , 13. , 13. ])
```

The drop in pressure for each football is the difference between the starting pressure and the combined pressure measurement.

In [ ]:
```
drop = start - football_combined['Combined']
football_combined['Pressure Drop'] = drop
football_combined.head()
```

Out[ ]:

	Team	Combined	Pressure Drop
0	Patriots	11.650	0.850
1	Patriots	11.025	1.475
2	Patriots	11.325	1.175
3	Patriots	10.850	1.650
4	Patriots	11.275	1.225

It looks as though the Patriots' drops were larger than the Colts'. Let's look at the average drop in each of the two groups. We no longer need the combined scores.

In [ ]:
```
football_combined1 = football_combined.copy()
football_combined_average = football_combined1.drop(columns=['Combined'])
football_combined_average = football_combined_average.groupby(by=['Team']).mean()
football_combined_average = football_combined_average.reset_index()
football_combined_average = football_combined_average.rename(columns={'Pressure Drop':'Pressu
football_combined_average
```

Out[ ]:

	Team	Pressure Drop average
0	Colts	0.468750
1	Patriots	1.202273

The average drop for the Patriots was about 1.2 psi compared to about 0.47 psi for the Colts.

The Hypotheses How does chance come in here? Nothing was being selected at random. But we can make a chance model by hypothesizing that the 11 Patriots' drops look like a random sample of 11 out of all the 15 drops, with the Colts' drops being the remaining four. That's a completely specified chance model under which we can simulate data. So it's the null hypothesis. For the alternative, we can take the position that the Patriots' drops are too large, on average, to resemble a random sample drawn from all the drops.

Test Statistic A natural statistic is the difference between the two average drops, which we will compute as "average drop for Patriots - average drop for Colts". Large values of this statistic will favor the alternative hypothesis.

In [ ]:
```
observed_means = football_combined_average['Pressure Drop average']
observed_difference = observed_means.iloc[1] - observed_means.iloc[0]
observed_difference
```

Out[ ]:
```
0.733522727272728
```

This positive difference reflects the fact that the average drop in pressure of the Patriots' footballs was greater than that of the Colts.

In [ ]:
```
def difference_of_means(table, label, group_label):
    reduced = table[[label, group_label]]
    means_table = reduced.groupby(group_label).mean()
    means = means_table[label]
    return means[1] - means[0]
```

We have defined this function in an earlier section. The definition is repeated here for ease of reference.

In [ ]:
```
import warnings
warnings.filterwarnings("ignore")

difference_of_means(football_combined, 'Pressure Drop', 'Team')
```

Out[ ]:
```
0.733522727272728
```

Predicting the Statistic Under the Null Hypothesis

If the null hypothesis were true, then it shouldn't matter which footballs are labeled Patriots and which are labeled Colts. The distributions of the two sets of drops would be the same. We can simulate this by randomly shuffling the team labels.

In [ ]:
```
shuffled_labels3 = football_combined.copy()
shuffled_labels4 = shuffled_labels3
shuffled_labels5 = shuffled_labels4.sample(len(shuffled_labels3), replace = False)
shuffled_labels6 = np.array(shuffled_labels5['Team'])
shuffled_labels3['Shuffled Label'] = shuffled_labels6
original_and_shuffled = shuffled_labels3.drop(columns=['Combined'])
original_and_shuffled.head()
```

Out[ ]:

	Team	Pressure Drop	Shuffled Label
0	Patriots	0.850	Patriots
1	Patriots	1.475	Patriots
2	Patriots	1.175	Patriots
3	Patriots	1.650	Colts
4	Patriots	1.225	Patriots

In [ ]:
```
import warnings
warnings.filterwarnings("ignore")

difference_of_means(original_and_shuffled, 'Pressure Drop', 'Shuffled Label')
```

Out[ ]:
```
-0.451136363636364
```

In [ ]:
```
import warnings
warnings.filterwarnings("ignore")

difference_of_means(original_and_shuffled, 'Pressure Drop', 'Team')
```

Out[ ]:
```
0.733522727272728
```

The two teams' average drop values are closer when the team labels are randomly assigned to the footballs than they were for the two groups actually used in the game.

Permutation Test

It's time for a step that is now familiar. We will do repeated simulations of the test statistic under the null hypothesis, by repeatedly permuting the footballs and assigning random sets to the two teams. Once again, we will use the function one_simulated_difference defined in an earlier section as follows.

In [ ]:
```
def one_simulated_difference(table, label, group_label):
    table_copy = table.copy()
    shuffled_labels = table_copy.sample(len(table_copy), replace = False)
    shuffled_labels = np.array(shuffled_labels[group_label])
    table_copy['Shuffled Label'] = shuffled_labels
```

```
    original_and_shuffled = table_copy.drop(columns=['Combined'])
    shuffled_group_means = original_and_shuffled.groupby('Shuffled Label').mean()
    table1 = shuffled_group_means.reset_index()
    return difference_of_means(table1, label, 'Shuffled Label')
```

In [ ]:
```
non_numeric = football_combined[pd.to_numeric(football_combined['Pressure Drop'], errors='coe
print("\nRows with non-numeric values in 'Pressure Drop':")
print(non_numeric[['Team', 'Pressure Drop']])
football_combined['Pressure Drop'] = pd.to_numeric(football_combined['Pressure Drop'], errors
football_combined['Pressure Drop'].fillna(football_combined['Pressure Drop'].mean(), inplace=
print("\nData types after cleaning:")
print(football_combined.dtypes)
football_combined_average = football_combined.groupby(by=['Team']).mean().reset_index()
football_combined_average = football_combined_average.rename(columns={'Pressure Drop': 'Press
print("\nFootball Combined Average DataFrame:")
print(football_combined_average)
repetitions = 1000
differences = np.empty(repetitions)
```

```
Rows with non-numeric values in 'Pressure Drop':
Empty DataFrame
Columns: [Team, Pressure Drop]
Index: []

Data types after cleaning:
Team              object
Combined         float64
Pressure Drop    float64
dtype: object

Football Combined Average DataFrame:
       Team   Combined  Pressure Drop average
0     Colts  12.531250               0.468750
1  Patriots  11.297727               1.202273
```

In [ ]:
```
empirical_P = np.count_nonzero(differences >= observed_difference) / 10000
empirical_P
```

Out[ ]:
```
0.0558
```

That's a pretty small P-value. To visualize this, here is the empirical distribution of the test statistic under the null hypothesis, with the observed statistic marked on the horizontal axis.

if len(differences) > 0: test_conclusion = pd.DataFrame({'Difference Between Group Averages':differences}) print('Empirical P-value:', empirical_P) unit = '' fig, ax = plt.subplots(figsize=(2,2)) ax.hist(test_conclusion, density=True, color='blue', alpha=0.8, ec='white') y_vals = ax.get_yticks() y_label = 'Percent per ' + (unit if unit else 'unit') x_label = 'Diference Between Group Averages' ax.set_yticklabels(['{:g}'.format(x * 100) for x in y_vals]) plt.ylabel(y_label) plt.xlabel(x_label) plt.title('Prediction Under the Null Hypothesis'); plt.show() else: print("No values in 'differences' array to plot.")

3.Causality

Our methods for comparing two samples have a powerful use in the analysis of randomized controlled experiments. Since the treatment and control groups are assigned randomly in such experiements, differences in their outcomes can be compared to what would happen just due to chance if the treatment had no effect at all. If the observed differences are more marked than what we would predict as purely due to chance, we will have evidence of causation. Because of the unbiased assignment of

individuals to the treatment and control groups, differences in the outcomes of the two groups can be ascribed to the treatment.

The key to the analysis of randomized controlled experiments is understanding exactly how chance enters the picture. This helps us set up clear null and alternative hypotheses. Once that's done, we can simply use the methods of the previous sections to complete the analysis.

Let's see how to do this in an example.

In [ ]:
```
bta = pd.read_csv('bta.csv')
bta.tail(2)
```

Out[ ]:

	Group	Result
29	Treatment	0.0
30	Treatment	0.0

Remember that counting is the same as adding zeros and ones. The sum of 1's in the control group is the number of control group patients who had pain relief. So the average of the number of 1's is the proportion of control group patients who had pain relief.

In [ ]:
```
bta_group = bta.groupby('Group').mean()
bta_group.reset_index()
```

Out[ ]:

	Group	Result
0	Control	0.125
1	Treatment	0.600

The table observed_outcomes collects the information about every patient's potential outcomes, leaving the unobserved half of each "ticket" blank. (It's just another way of thinking about the bta table, carrying the same information.)

In [ ]:
```
observed_outcomes = pd.read_csv("observed_outcomes.csv")
observed_outcomes.head(2)
```

Out[ ]:

	Group	Outcome if assigned treatment	Outcome if assigned control
0	Control	Unknown	1
1	Control	Unknown	1

The Hypotheses

This hypothesis test evaluates whether a treatment (botulinum toxin A) has an effect compared to a control (saline). Under the **null hypothesis**, there's no difference between the treatment and control outcomes, meaning any observed differences are due to chance. The **alternative hypothesis** suggests the treatment has a distinct effect. To test this, we simulate by randomly permuting the outcomes between treatment and control groups and comparing the resulting distributions, just as in A/B testing.

The Test Statistic : If the two group proportions are very different from each other, we will lean towards the alternative hypothesis that the two underlying distributions are different. So our test statistic will be the distance between the two group proportions, that is, the absolute value of the difference between them.

Large values of the test statistic will favor the alternative hypothesis over the null.

Since the two group proportions were 0.6 and 0.125, the observed value of the test statistic is $|0.6-0.125| = 0.475$.

In [ ]:
```
bta_group = bta.groupby('Group').mean()
bta_group = bta_group.reset_index()
bta_group
```

Out[ ]:

	Group	Result
0	Control	0.125
1	Treatment	0.600

In [ ]:
```
observed_proportions = bta_group['Result']
observed_distance = abs(observed_proportions.iloc[0] - observed_proportions.iloc[1])
observed_distance
```

Out[ ]:
```
0.475
```

As we have done before, we will define a function that takes the following arguments: the name of the table of data the column label of the numerical variable the column label of the group labels and returns the absolute difference between the two group proportions.

In [ ]:
```
def distance(table, label, group_label):
    reduced = table[[label, group_label]]
    proportions = reduced.groupby(group_label).mean()
    distance = proportions[label]
    return abs(distance.iloc[1] - distance.iloc[0])
```

In [ ]:
```
distance(bta_group, 'Result', 'Group')
```

Out[ ]:
```
0.475
```

Predicting the Statistic Under the Null Hypothesis : We can simulate results under the null hypothesis, to see how our test statistic should come out if the null hypothesis is true.

Generating One Value of the Statistic : The simulation follows exactly the same process we used in the previous section. We start by randomly permuting the all group labels and then attaching the shuffled labels to the 0/1 results.

In [ ]:
```
shuffled_labels3 = bta.copy()
shuffled_labels4 = shuffled_labels3
shuffled_labels5 = shuffled_labels4.sample(len(shuffled_labels3), replace = False)
shuffled_labels6 = np.array(shuffled_labels5['Group'])
shuffled_labels3['Shuffled Label'] = shuffled_labels6
bta_with_shuffled_labels = shuffled_labels3
bta_with_shuffled_labels.head(2)
```

Out[ ]:

	Group	Result	Shuffled Label
0	Control	1.0	Treatment
1	Control	1.0	Treatment

We can now find the distance between the two proportions after the group labels have been shuffled.

In [ ]:
```
distance(bta_with_shuffled_labels, 'Result', 'Shuffled Label')
```

Out[ ]: 0.041666666666666685

This is quite different from the distance between the two original proportions.

In [ ]:
```
distance(bta_with_shuffled_labels, 'Result', 'Group')
```

Out[ ]: 0.475

Permutation Test : If we shuffled the labels again, how different would the new distance be? To answer this, we will define a function that simulates one simulated value of the distance under the hypothesis of random draws from the same underlying distribution. And then we will collect 20,000 such simulated values in an array.

You can see that we are doing exactly what we did in our previous examples of the permutation test.

In [ ]:
```
def one_simulated_distance(table, label, group_label):
    table_copy = table.copy()

    shuffled_labels = table_copy[group_label].sample(frac=1).reset_index(drop=True)
    table_copy['Shuffled Label'] = shuffled_labels

    table_copy[label] = pd.to_numeric(table_copy[label], errors='coerce')

    table_copy = table_copy.dropna(subset=[label])
    shuffled_group_means = table_copy.groupby('Shuffled Label')[label].mean()
    table1 = shuffled_group_means.reset_index()

    return distance(table1, label, 'Shuffled Label')

distances = np.array([])
repetitions = 20000
for i in np.arange(repetitions):
    new_distance = one_simulated_distance(bta, 'Result', 'Group')
    distances = np.append(distances, new_distance)
```

Conclusion of the Test : The array distances contains 20,000 values of our test statistic simulated under the null hypothesis.

To find the P-value of the test, remember that large values of the test statistic favor the alternative hypothesis. So the empirical P-value is the proportion of simulated statistics that were equal to or larger than the observed statistic.

In [ ]:
```
empirical_P = np.count_nonzero(distances >= observed_distance) / repetitions
empirical_P
```

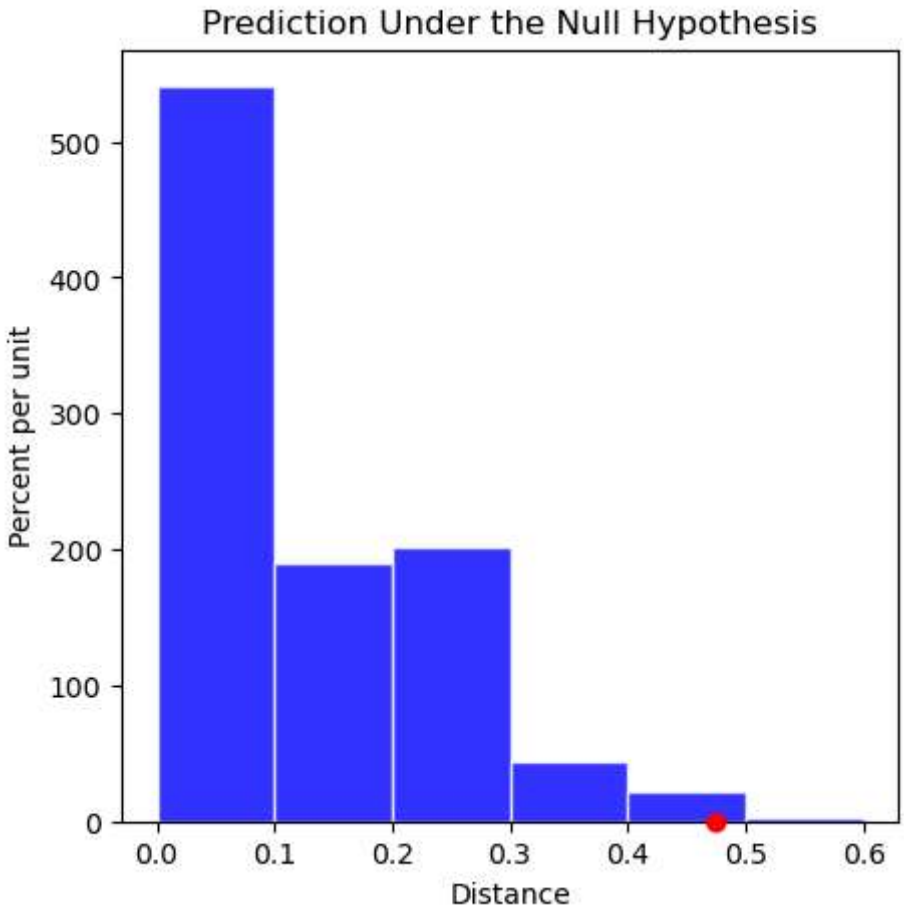
Out[ ]: 0.01045

his is a small P-value. The observed statistic, shown as the red dot below, is in the tail of the empirical histogram of the test statistic generated under the null hypothesis. The result is statistically significant. The test favors the alternative hypothesis over the null. The evidence supports the hypothesis that the treatment is doing something.

In [ ]:
```
import warnings
warnings.filterwarnings("ignore")

test_conclusion = pd.DataFrame({'Distance':distances})
print('Observed Distance', observed_distance)
print('Empirical P-value:', round(empirical_P, 4) *100, '%')
unit = ''
fig, ax = plt.subplots(figsize=(5,5))
```

```
ax.hist(test_conclusion, bins = np.arange(0, 0.7, 0.1), density=True, color='blue', alpha=0.8
ax.scatter(observed_distance, 0, color='red', s=40, zorder=10).set_clip_on(False)
y_vals = ax.get_yticks()
y_label = 'Percent per ' + (unit if unit else 'unit')
x_label = 'Distance'
ax.set_yticklabels(['{:g}'.format(x * 100) for x in y_vals])
plt.ylabel(y_label)
plt.xlabel(x_label)
plt.title('Prediction Under the Null Hypothesis');
plt.show()
```

```
Observed Distance 0.475
Empirical P-value: 1.04 %
```



Foundations Of DataScience

Name: Krishna GSVV

Roll no. AV.EN.U4CSE22016

Lab 14 Estimating the unknown Parameter (Percentile Method, Bootstrap Method, Confidence Intervals)

In [5]:
```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
```

Imports necessary libraries like numpy, pandas, seaborn, and matplotlib.pyplot for numerical computations, data manipulation, visualization, and plotting, respectively.

In [6]:
```
sizes = np.array([12, 17, 6, 9, 7])
np.sort(sizes)
```

Out[6]:
```
array([ 6,  7,  9, 12, 17])
```

Creates a NumPy array named sizes with the given values and sorts it in ascending order.

In [7]:
```
np.percentile(sizes, 70, interpolation='nearest')
```

Out[7]: 12

Calculates the 70th percentile of the sizes array using the nearest-rank method for interpolation.

In [8]:
```
scores_and_sections = pd.read_csv('scores_by_section.csv')
scores_and_sections.head(10)
```

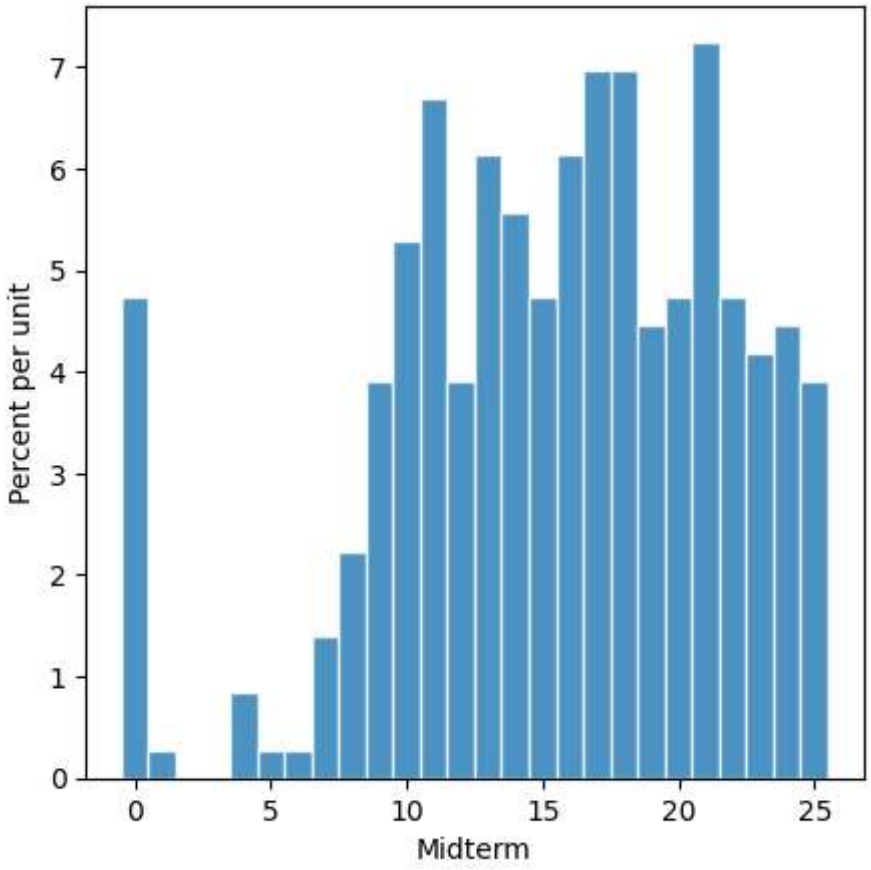
Out[8]:

	Section	Midterm
0	1	22
1	2	12
2	2	23
3	2	14
4	1	20
5	3	25
6	4	19
7	1	24
8	5	8
9	6	14

Reads data from a CSV file named 'scores_by_section.csv' into a Pandas DataFrame called scores_and_sections and displays the first 10 rows.

In [9]:
```
unit = ''
fig, ax = plt.subplots(figsize=(5,5))
ax.hist(scores_and_sections['Midterm'], bins = np.arange(-0.5, 25.6, 1), density=True, alpha=
y_vals = ax.get_yticks()
y_label = 'Percent per ' + (unit if unit else 'unit')
x_label = 'Midterm'
ax.set_yticklabels(['{:g}'.format(x * 100) for x in y_vals])
plt.ylabel(y_label)
plt.xlabel(x_label)
plt.title('');
plt.show()
```

```
<ipython-input-9-f665afe7712c>:7: UserWarning: set_ticklabels() should only be used with a fix
ed number of ticks, i.e. after set_ticks() or using a FixedLocator.
  ax.set_yticklabels(['{:g}'.format(x * 100) for x in y_vals])
```



Creates a histogram of the 'Midterm' scores from the scores_and_sections DataFrame, setting bin edges and formatting axes labels.

In [10]:
```
scores = scores_and_sections.iloc[:,1]
scores
```

Out[10]:

	Midterm
0	22
1	12
2	23
3	14
4	20
...	...
354	24
355	16
356	17
357	16
358	14

359 rows × 1 columns

dtype: int64

Extracts the 'Midterm' scores from the DataFrame into a Pandas Series named scores.

In [11]:
```
np.percentile(scores , 85, interpolation='nearest')
```

Out[11]: 22

Computes the 85th percentile of the scores Series using the nearest-rank method.

In [12]:
```
sorted_scores = np.sort(scores_and_sections.iloc[:,1])
sorted_scores
```

Out[12]:
```
array([ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
        1,  4,  4,  4,  5,  6,  7,  7,  7,  7,  7,  8,  8,  8,  8,  8,  8,
        8,  8,  9,  9,  9,  9,  9,  9,  9,  9,  9,  9,  9,  9,  9,  9, 10,
       10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10,
       10, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11,
       11, 11, 11, 11, 11, 11, 11, 11, 12, 12, 12, 12, 12, 12, 12, 12, 12,
       12, 12, 12, 12, 12, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13,
       13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 14, 14, 14, 14, 14, 14, 14,
       14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 15, 15, 15, 15, 15,
       15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 16, 16, 16, 16, 16,
       16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 17,
       17, 17, 17, 17, 17, 17, 17, 17, 17, 17, 17, 17, 17, 17, 17, 17, 17,
       17, 17, 17, 17, 17, 17, 17, 17, 17, 18, 18, 18, 18, 18, 18, 18, 18,
       18, 18, 18, 18, 18, 18, 18, 18, 18, 18, 18, 18, 18, 18, 18, 18, 18,
       19, 19, 19, 19, 19, 19, 19, 19, 19, 19, 19, 19, 19, 19, 19, 19, 20,
       20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 21,
       21, 21, 21, 21, 21, 21, 21, 21, 21, 21, 21, 21, 21, 21, 21, 21, 21,
       21, 21, 21, 21, 21, 21, 21, 21, 22, 22, 22, 22, 22, 22, 22, 22, 22,
       22, 22, 22, 22, 22, 22, 22, 22, 23, 23, 23, 23, 23, 23, 23, 23, 23,
       23, 23, 23, 23, 23, 23, 24, 24, 24, 24, 24, 24, 24, 24, 24, 24, 24,
       24, 24, 24, 24, 24, 25, 25, 25, 25, 25, 25, 25, 25, 25, 25, 25, 25,
       25, 25])
```

Sorts the 'Midterm' scores and stores them in a NumPy array named sorted_scores.

In [13]:
```
0.85 * 359
```

Out[13]: 305.15

Calculates the index corresponding to the 85th percentile in the sorted scores array (which has 359 elements).

In [14]:
```
sorted_scores.item(305)
```

Out[14]: 22

Accesses the element at the calculated index (305) in the sorted_scores array, which represents the 85th percentile value.

In [15]:
```
np.percentile(scores, 25, interpolation='nearest')
```

Out[15]: 11

In [16]:
```
np.percentile(scores, 50, interpolation='nearest')
```

Out[16]: 16

In [17]:
```
np.percentile(scores, 75, interpolation='nearest')
```

Out[17]: 20

Calculate and display the 25th, 50th (median), and 75th percentiles of the scores Series, respectively.

2. The Bootstrap

In [18]:
```
sf2015 = pd.read_csv('san_francisco_2015.csv')
sf2015.head()
```

Out[18]:

	Unnamed: 0	Year Type	Year	Organization Group Code	Organization Group	Department Code	Department	Union Code	
0	0	Calendar	2015	2	Public Works, Transportation & Commerce	WTR	PUC Water Department	21.0	Prof Engi Miscella L
1	1	Calendar	2015	2	Public Works, Transportation & Commerce	DPW	General Services Agency - Public Works	12.0	Linoleu S Worker
2	2	Calendar	2015	4	Community Health	DPH	Public Health	790.0	Miscella Loc
3	3	Calendar	2015	4	Community Health	DPH	Public Health	351.0	Mu Ex Assoc Miscell
4	4	Calendar	2015	2	Public Works, Transportation & Commerce	MTA	Municipal Transportation Agency	790.0	Miscella Loc

5 rows × 23 columns

Reads data from a CSV file named 'san_francisco_2015.csv' into a Pandas DataFrame called sf2015.

In [19]:
```
sf2015.head()
```

Out[19]:

	Unnamed: 0	Year Type	Year	Organization Group Code	Organization Group	Department Code	Department	Union Code	
0	0	Calendar	2015	2	Public Works, Transportation & Commerce	WTR	PUC Water Department	21.0	Prof Engi Miscella L
1	1	Calendar	2015	2	Public Works, Transportation & Commerce	DPW	General Services Agency - Public Works	12.0	Linoleu S Worker
2	2	Calendar	2015	4	Community Health	DPH	Public Health	790.0	Miscella Loc
3	3	Calendar	2015	4	Community Health	DPH	Public Health	351.0	Mu Ex Assoc Miscell
4	4	Calendar	2015	2	Public Works, Transportation & Commerce	MTA	Municipal Transportation Agency	790.0	Miscella Loc

5 rows × 23 columns

Displays the first 10 rows of the sf2015 DataFrame.

In [20]:
```
sf2015[sf2015['Job'] == 'Mayor']
```

Out[20]:

	Unnamed: 0	Year Type	Year	Organization Group Code	Organization Group	Department Code	Department	Union Code	Uni
3335	3335	Calendar	2015	6	General Administration & Finance	MYR	Mayor	556.0	Elect Offici

1 rows × 23 columns

Filters the sf2015 DataFrame to display rows where the 'Job' column is 'Mayor'.

In [21]:
```
sf2015.sort_values(by=['Total Compensation']).head()
```

Out[21]:

	Unnamed: 0	Year Type	Year	Organization Group Code	Organization Group	Department Code	Department	Union Code	
27308	27308	Calendar	2015	1	Public Protection	FIR	Fire Department	798.0	Fi Mis
15746	15746	Calendar	2015	4	Community Health	DPH	Public Health	790.0	Mis
24576	24576	Calendar	2015	1	Public Protection	JUV	Juvenile Probation	790.0	Mis
42982	42982	Calendar	2015	6	General Administration & Finance	CPC	City Planning	21.0	Mis
23310	23310	Calendar	2015	6	General Administration & Finance	CPC	City Planning	21.0	Mis

5 rows × 23 columns

Sorts the sf2015 DataFrame by 'Total Compensation' in ascending order.

In [22]:
```
sf2015 = sf2015[sf2015['Salaries'] > 10000]
```

Filters the sf2015 DataFrame to keep only rows where 'Salaries' are greater than 10000.

In [23]:
```
len(sf2015)
```

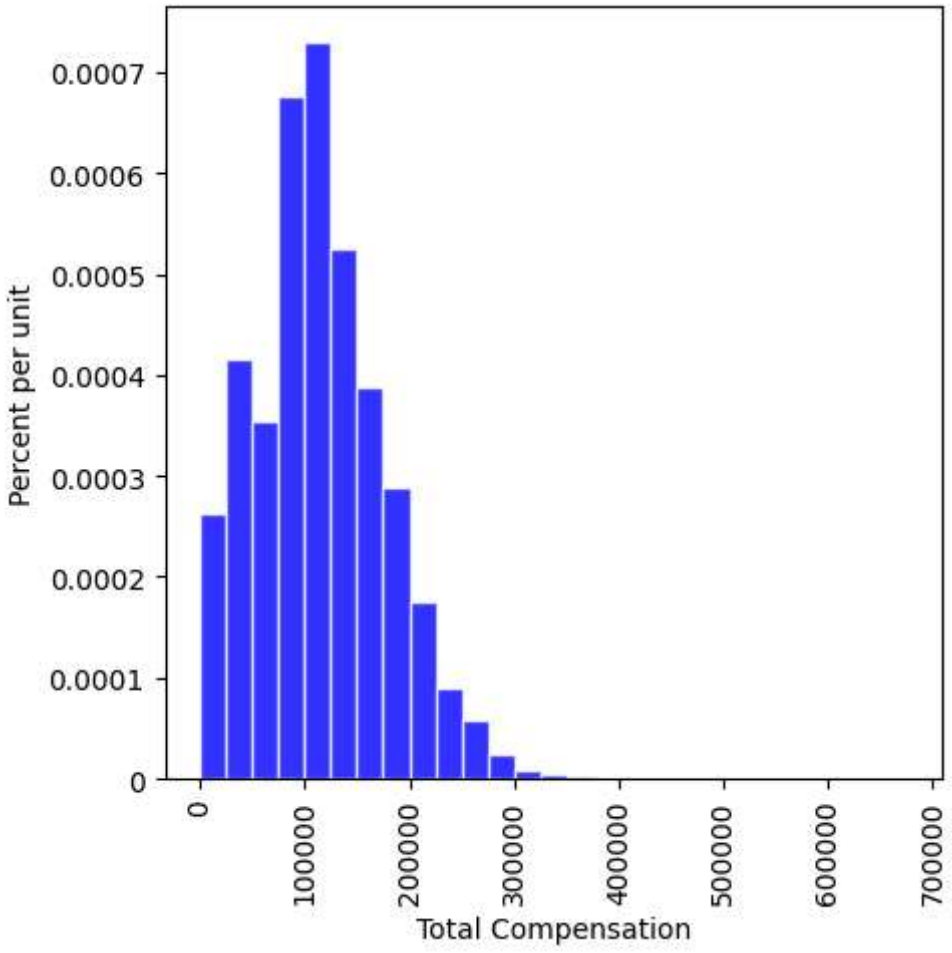
Out[23]: 36569

Displays the number of rows in the filtered sf2015 DataFrame.

Population and Parameter

In [24]:
```
sf_bins = np.arange(0, 700000, 25000)
unit = ''
fig, ax = plt.subplots(figsize=(5,5))
ax.hist(sf2015['Total Compensation'], bins = sf_bins, density=True, color='blue', alpha=0.8,
y_vals = ax.get_yticks()
y_label = 'Percent per ' + (unit if unit else 'unit')
x_label = 'Total Compensation'
ax.set_yticklabels(['{:g}'.format(x * 100) for x in y_vals])
plt.ylabel(y_label)
plt.xlabel(x_label)
plt.xticks(rotation=90)
plt.title('');
plt.show()
```

```
<ipython-input-24-c6cc109eb3c3>:8: UserWarning: set_ticklabels() should only be used with a fi
xed number of ticks, i.e. after set_ticks() or using a FixedLocator.
  ax.set_yticklabels(['{:g}'.format(x * 100) for x in y_vals])
```



Creates a histogram of 'Total Compensation' from the sf2015 DataFrame, with specified bin edges and formatting.

In [25]:
```
sf2015.sort_values(by=['Total Compensation'], ascending=False).head(2)
```

Out[25]:

	Unnamed: 0	Year Type	Year	Organization Group Code	Organization Group	Department Code	Department	Union Code	
19177	19177	Calendar	2015	6	General Administration & Finance	RET	Retirement System	351.0	As Mis
13194	13194	Calendar	2015	6	General Administration & Finance	ADM	General Services Agency - City Admin	164.0	and Mis

2 rows × 23 columns

Sorts the sf2015 DataFrame by 'Total Compensation' in descending order and displays the top 2 rows.

In [26]:
```
pop_median = np.percentile(sf2015['Total Compensation'], 50)
pop_median
```

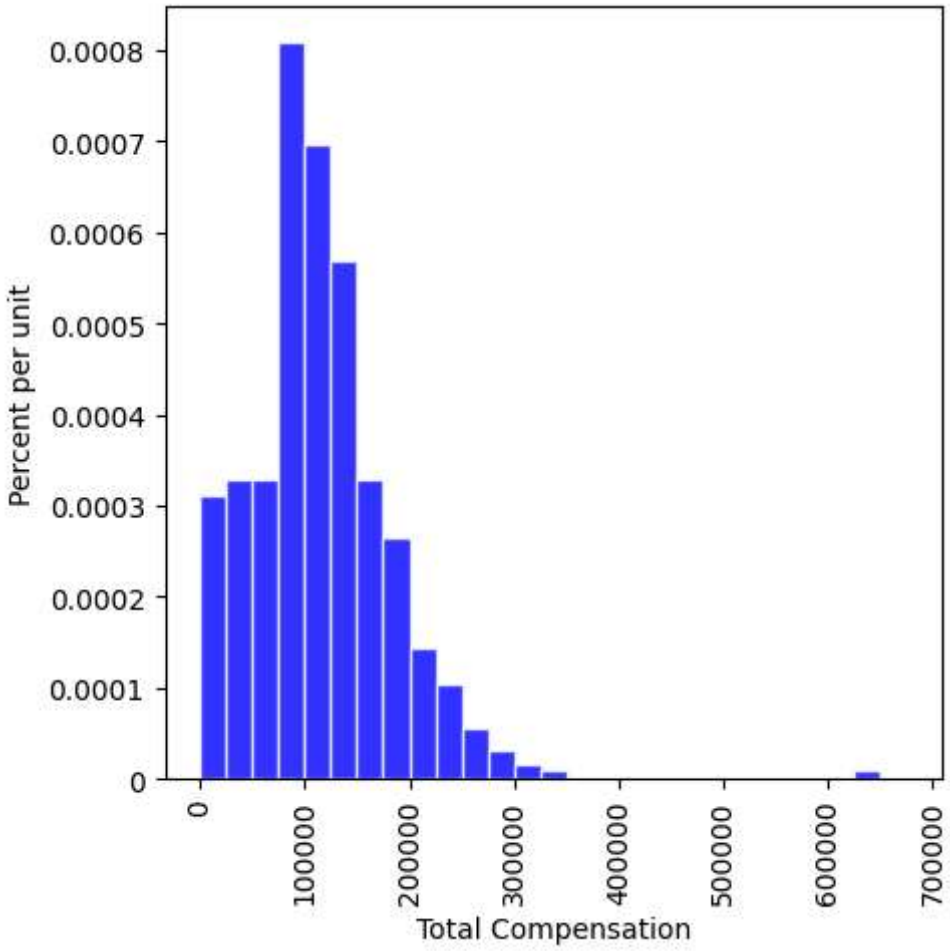
Out[26]:
```
110305.79
```

Calculates the median 'Total Compensation' from the sf2015 DataFrame and assigns it to pop_median.

A Random Sample and an Estimate

In [27]:
```
our_sample = sf2015.sample(500, replace=False)
unit = ''
fig, ax = plt.subplots(figsize=(5,5))
ax.hist(our_sample['Total Compensation'], bins = sf_bins, density=True, color='blue', alpha=0
y_vals = ax.get_yticks()
y_label = 'Percent per ' + (unit if unit else 'unit')
x_label = 'Total Compensation'
ax.set_yticklabels(['{:g}'.format(x * 100) for x in y_vals])
plt.ylabel(y_label)
plt.xlabel(x_label)
plt.xticks(rotation=90)
plt.title('');
plt.show()
```

```
<ipython-input-27-79e5953d04b4>:8: UserWarning: set_ticklabels() should only be used with a fi
xed number of ticks, i.e. after set_ticks() or using a FixedLocator.
  ax.set_yticklabels(['{:g}'.format(x * 100) for x in y_vals])
```



Takes a random sample of 500 rows from the sf2015 DataFrame without replacement and assigns it to our_sample. It then creates a histogram of 'Total Compensation' for this sample.

In [28]:
```
est_median = np.percentile(our_sample['Total Compensation'], 50)
est_median
```

Out[28]:
```
109736.235
```

Calculates the median 'Total Compensation' from the our_sample DataFrame and assigns it to est_median.

The Bootstrap: Resampling from the Sample

In [29]:
```
resample_1 = our_sample.sample(len(our_sample), replace=True)
resample_1.head()
```

Out[29]:

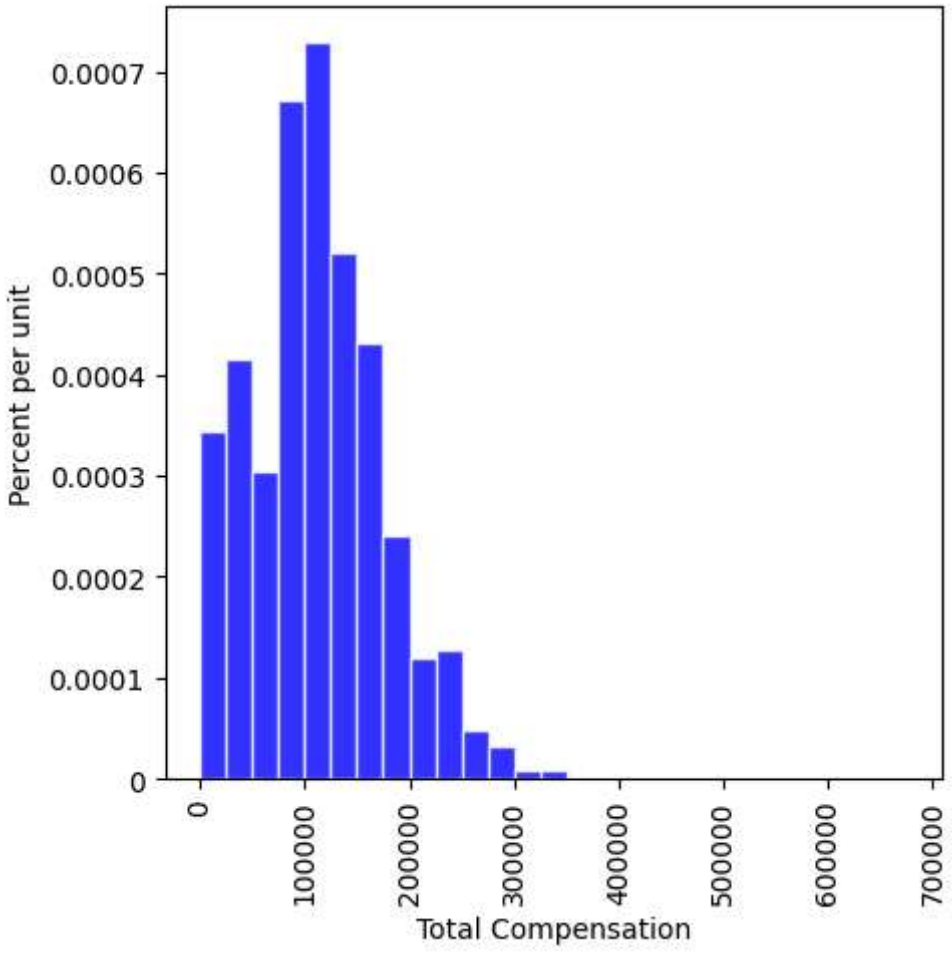
	Unnamed: 0	Year Type	Year	Organization Group Code	Organization Group	Department Code	Department	Union Code	
7737	7737	Calendar	2015	1	Public Protection	POL	Police	911.0	
16207	16207	Calendar	2015	4	Community Health	DPH	Public Health	790.0	Mi
12391	12391	Calendar	2015	2	Public Works, Transportation & Commerce	MTA	Municipal Transportation Agency	253.0	
7245	7245	Calendar	2015	4	Community Health	DPH	Public Health	790.0	Mi
20827	20827	Calendar	2015	2	Public Works, Transportation & Commerce	DPW	General Services Agency - Public Works	38.0	Pl

5 rows × 23 columns

Creates a bootstrap resample from our_sample with replacement and assigns it to resample_1.

In [30]:
```
unit = ''
fig, ax = plt.subplots(figsize=(5,5))
ax.hist(resample_1['Total Compensation'], bins = sf_bins, density=True, color='blue', alpha=0
y_vals = ax.get_yticks()
y_label = 'Percent per ' + (unit if unit else 'unit')
x_label = 'Total Compensation'
ax.set_yticklabels(['{:g}'.format(x * 100) for x in y_vals])
plt.ylabel(y_label)
plt.xlabel(x_label)
plt.xticks(rotation=90)
plt.title('');
plt.show()
```

```
<ipython-input-30-2adf367ed7cc>:7: UserWarning: set_ticklabels() should only be used with a fi
xed number of ticks, i.e. after set_ticks() or using a FixedLocator.
  ax.set_yticklabels(['{:g}'.format(x * 100) for x in y_vals])
```



Creates a histogram of 'Total Compensation' for the resample_1 DataFrame.

In [31]:
```
resampled_median_1 = np.percentile(resample_1['Total Compensation'], 50, interpolation='neare
resampled_median_1
```

Out[31]:
```
111456.02
```

Calculates the median 'Total Compensation' from the resample_1 DataFrame and assigns it to resampled_median_1.

In [32]:
```
resample_2 = our_sample.sample(len(our_sample), replace=True)
resampled_median_2 = np.percentile(resample_2['Total Compensation'], 50, interpolation='neare
resampled_median_2
```

Out[32]:
```
102017.31
```

Creates another bootstrap resample (resample_2) and calculates its median (resampled_median_2).

Bootstrap Empirical Distribution of the Sample Median

In [33]:
```
def bootstrap_median(original_sample, label, replications):
    """Returns an array of bootstrapped sample medians:
    original_sample: table containing the original sample
    label: label of column containing the variable
    replications: number of bootstrap samples
    """
    just_one_column = original_sample[label]
    medians = np.array([])
    for i in np.arange(replications):
        bootstrap_sample = just_one_column.sample(len(just_one_column), replace=True)
        resampled_median = np.percentile(bootstrap_sample, 50)
        medians = np.append(medians, resampled_median)
```

```
    return medians
```

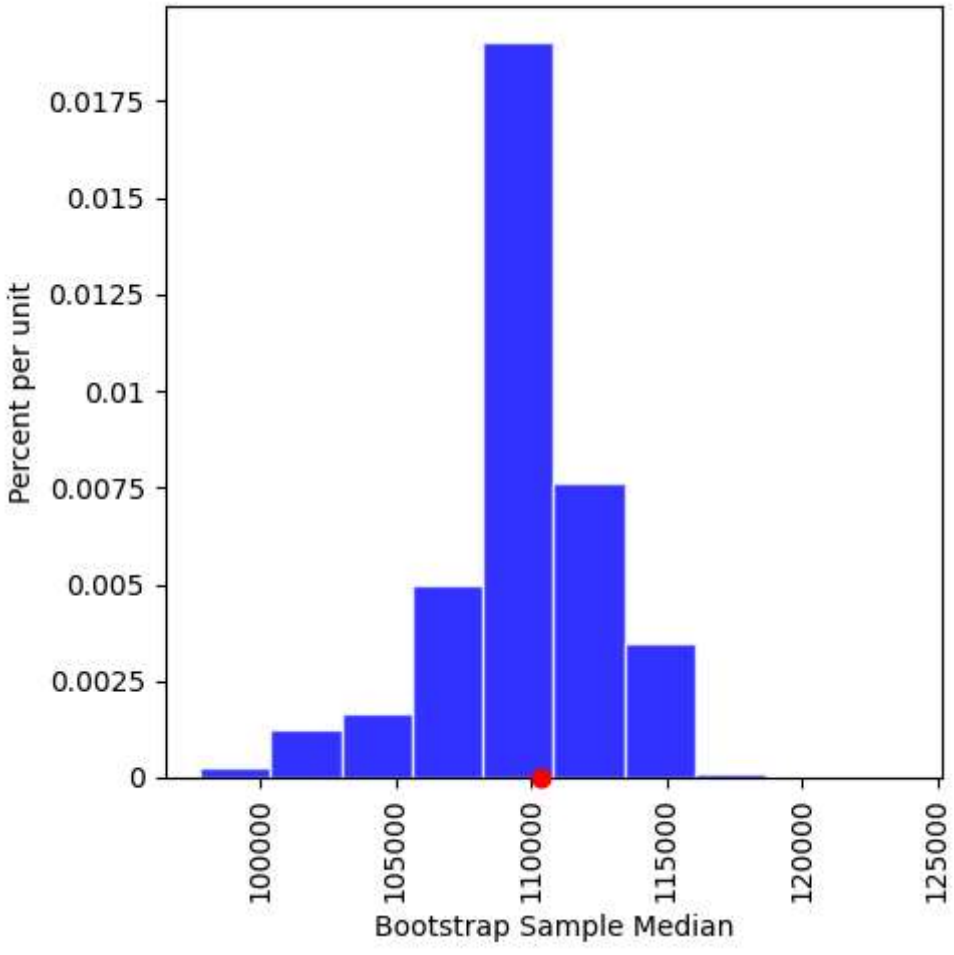
Defines a function bootstrap_median that takes an original sample, a column label, and the number of replications as input. It generates bootstrap samples, calculates the median of each sample, and returns an array of these medians.

In [34]:
```
bstrap_medians = bootstrap_median(our_sample, 'Total Compensation', 5000)
```

Calls the bootstrap_median function with the our_sample DataFrame, 'Total Compensation' column, and 5000 replications to generate an array of bootstrapped medians.

In [35]:
```
resampled_medians = pd.DataFrame({'Bootstrap Sample Median':bstrap_medians})
unit = ''
fig, ax = plt.subplots(figsize=(5,5))
ax.hist(resampled_medians, density=True, color='blue', alpha=0.8, ec='white')
ax.scatter(pop_median, 0, color='red', s=40, zorder=10).set_clip_on(False)
y_vals = ax.get_yticks()
y_label = 'Percent per ' + (unit if unit else 'unit')
x_label = 'Bootstrap Sample Median'
ax.set_yticklabels(['{:g}'.format(x * 100) for x in y_vals])
plt.ylabel(y_label)
plt.xlabel(x_label)
plt.xticks(rotation=90)
plt.title('');
plt.show()
```

```
<ipython-input-35-614f9327ad10>:9: UserWarning: set_ticklabels() should only be used with a fi
xed number of ticks, i.e. after set_ticks() or using a FixedLocator.
  ax.set_yticklabels(['{:g}'.format(x * 100) for x in y_vals])
```



Creates a histogram of the bootstrapped medians, highlighting the population median with a red dot.

Do the Estimates Capture the Parameter?

In [36]:
```
left = np.percentile(bstrap_medians, 2.5, interpolation='nearest')
left
```

Out[36]:
```
102017.31
```

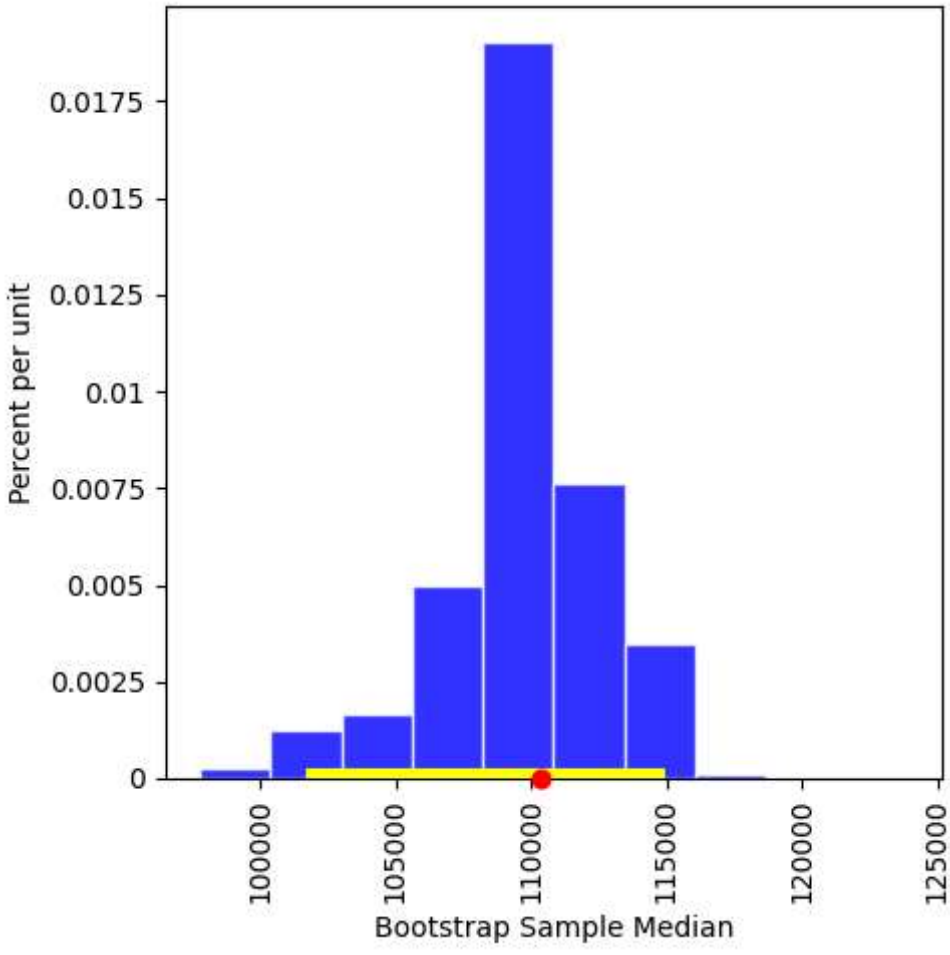
In [37]:
```
right = np.percentile(bstrap_medians, 97.5, interpolation='nearest')
right
```

Out[37]:
```
114478.75
```

Calculates the 2.5th and 97.5th percentiles of the bootstrapped medians to define the lower and upper bounds of a 95% confidence interval.

In [38]:
```
resampled_medians = pd.DataFrame({'Bootstrap Sample Median':bstrap_medians})
unit = ''
fig, ax = plt.subplots(figsize=(5,5))
ax.hist(resampled_medians, density=True, color='blue', alpha=0.8, ec='white', zorder=5)
ax.plot(np.array([left, right]), np.array([0,0]), color='yellow', lw=8, zorder=10)
ax.scatter(pop_median, 0, color='red', s=40, zorder=15).set_clip_on(False)
y_vals = ax.get_yticks()
y_label = 'Percent per ' + (unit if unit else 'unit')
x_label = 'Bootstrap Sample Median'
ax.set_yticklabels(['{:g}'.format(x * 100) for x in y_vals])
plt.ylabel(y_label)
plt.xlabel(x_label)
plt.xticks(rotation=90)
plt.title('');
plt.show()
```

```
<ipython-input-38-746303b4b36a>:10: UserWarning: set_ticklabels() should only be used with a f
ixed number of ticks, i.e. after set_ticks() or using a FixedLocator.
  ax.set_yticklabels(['{:g}'.format(x * 100) for x in y_vals])
```



Creates a histogram of the bootstrapped medians, highlighting the 95% confidence interval with a yellow line and the population median with a red dot.

In [39]:
```
total_comps = sf2015[['Total Compensation']]
total_comps
```

Out[39]:

	Total Compensation
0	117766.86
1	41209.83
2	110561.13
3	38624.97
6	260280.95
...	...
42983	61349.71
42984	132788.81
42986	73295.59
42987	19973.37
42988	55812.90

36569 rows × 1 columns

Selects the 'Total Compensation' column from the sf2015 DataFrame and assigns it to total_comps.

In [40]:
```
# THE BIG SIMULATION: This one takes several minutes.

# Generate 100 intervals, in the table intervals

left_ends = np.array([])
right_ends = np.array([])

total_comps = sf2015[['Total Compensation']]

for i in np.arange(100):
    first_sample = total_comps.sample(500, replace=False)
    medians = bootstrap_median(first_sample, 'Total Compensation', 5000)
    left_ends = np.append(left_ends, np.percentile(medians, 2.5))
    right_ends = np.append(right_ends, np.percentile(medians, 97.5))

intervals = pd.DataFrame(
    {'Left':left_ends,
    'Right':right_ends}
)
```

Performs a simulation to generate 100 confidence intervals for the median 'Total Compensation'. It repeatedly samples from the data, calculates bootstrap medians, and stores the interval endpoints. The results are stored in a DataFrame called intervals.

In [41]:
```
intervals
```

Out[41]:

	Left	Right
0	102092.135000	113449.505000
1	109523.980000	117805.415625
2	101441.385000	114467.380000
3	101932.340000	113369.965000
4	103916.760000	117738.480000
...	...	...
95	105452.350000	118188.205000
96	102277.002000	118400.900000
97	105169.657625	116552.360000
98	103816.550000	114313.905000
99	109710.220000	119122.590000

100 rows × 2 columns

Prints the intervals DataFrame.

In [42]:
```
pop_median
```

Out[42]:
```
110305.79
```

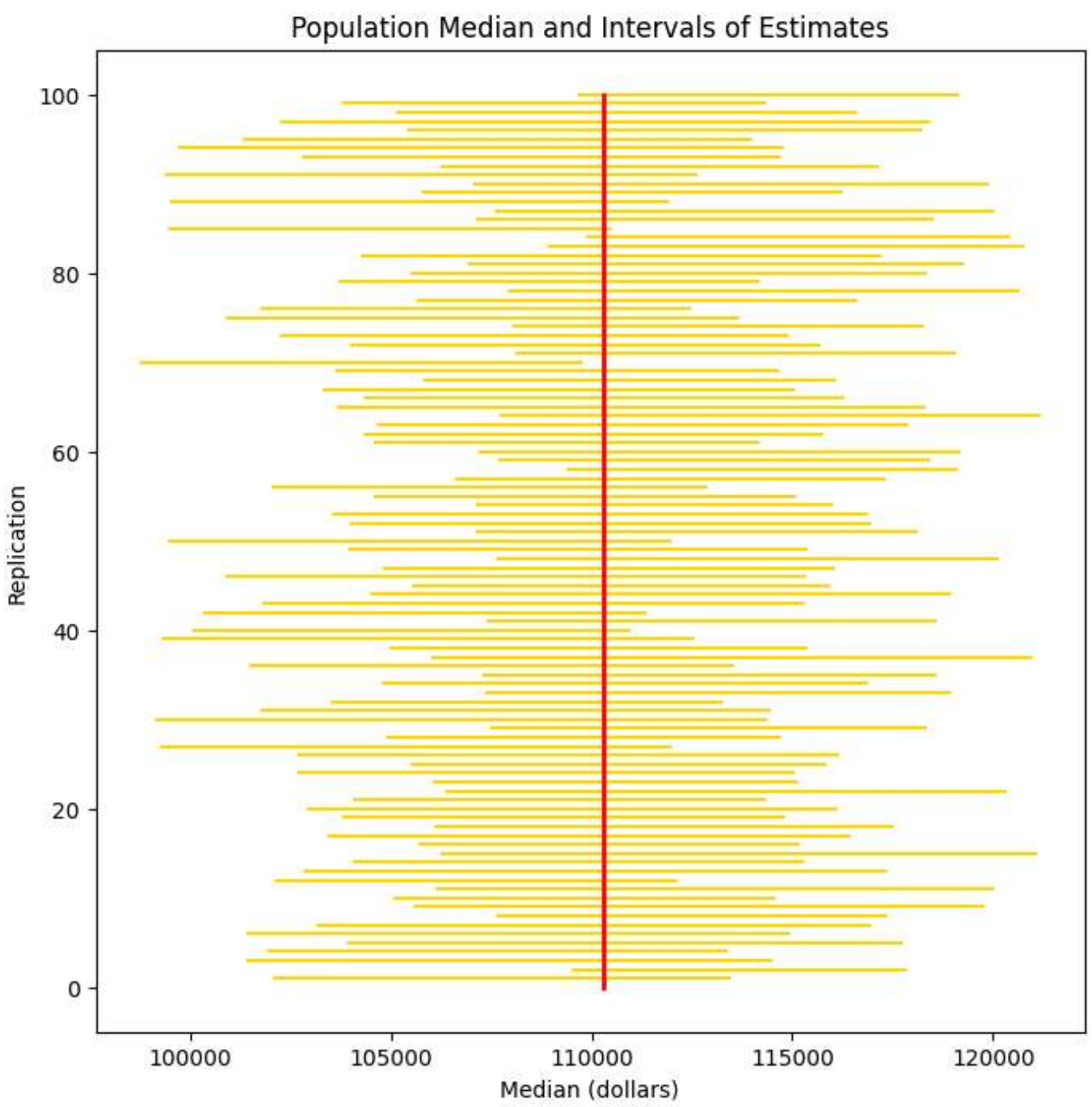
Prints the population median (pop_median).

In [43]:
```
len(
    intervals[
        (intervals['Left'] < pop_median) &
              (intervals['Right'] > pop_median)
    ]
)
```

Out[43]:
```
99
```

: Calculates the number of intervals that contain the population median.

In [44]:
```
replication_number = np.arange(1, 101)

replication_number = replication_number.astype(str)

intervals2 = pd.DataFrame(np.array([left_ends, right_ends]), columns=[replication_number])

intervals2

plt.figure(figsize=(8,8))
for i in np.arange(100):
    ends = intervals2.iloc[:,i]
    plt.plot(ends, np.array([i+1, i+1]), color='gold')
plt.plot(np.array([pop_median, pop_median]), np.array([0, 100]), color='red', lw=2)
plt.xlabel('Median (dollars)')
plt.ylabel('Replication')
plt.title('Population Median and Intervals of Estimates');
```



Visualizes the 100 confidence intervals as horizontal lines, with the population median as a vertical red line.

3. Confidence Intervals

In [45]:
```
baby = pd.read_csv('baby.csv')
baby.head()
```

Out[45]:

	Birth Weight	Gestational Days	Maternal Age	Maternal Height	Maternal Pregnancy Weight	Maternal Smoker
0	120	284	27	62	100	False
1	113	282	33	64	135	False
2	128	279	28	64	115	True
3	108	282	23	67	125	True
4	136	286	25	62	93	False

Reads data from 'baby.csv' into a DataFrame called baby.

In [46]:
```
ratios = baby[['Birth Weight', 'Gestational Days']]
ratios['Ratio BW/GD'] = baby['Birth Weight']/baby['Gestational Days']
```

```
<ipython-input-46-6f55d08f06bf>:2: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  ratios['Ratio BW/GD'] = baby['Birth Weight']/baby['Gestational Days']
```

Creates a new DataFrame ratios with 'Birth Weight' and 'Gestational Days', and adds a new column 'Ratio BW/GD' calculated by dividing 'Birth Weight' by 'Gestational Days'.

In [47]:
```
ratios
```

Out[47]:

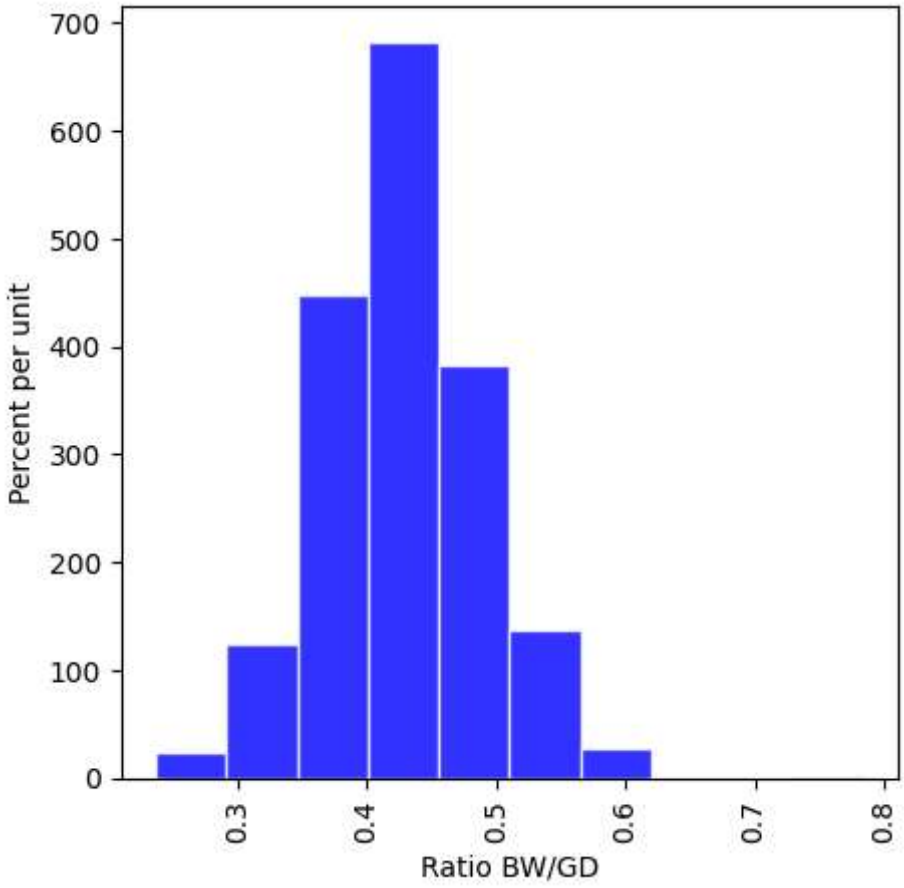
	Birth Weight	Gestational Days	Ratio BW/GD
0	120	284	0.422535
1	113	282	0.400709
2	128	279	0.458781
3	108	282	0.382979
4	136	286	0.475524
...	...	...	...
1169	113	275	0.410909
1170	128	265	0.483019
1171	130	291	0.446735
1172	125	281	0.444840
1173	117	297	0.393939

1174 rows × 3 columns

Displays the ratios DataFrame.

In [48]:
```
unit = ''
fig, ax = plt.subplots(figsize=(5,5))
ax.hist(ratios['Ratio BW/GD'], density=True, color='blue', alpha=0.8, ec='white')
y_vals = ax.get_yticks()
y_label = 'Percent per ' + (unit if unit else 'unit')
x_label = 'Ratio BW/GD'
ax.set_yticklabels(['{:g}'.format(x * 100) for x in y_vals])
plt.ylabel(y_label)
plt.xlabel(x_label)
plt.xticks(rotation=90)
plt.title('');
plt.show()
```

```
<ipython-input-48-e1f41ed0b246>:7: UserWarning: set_ticklabels() should only be used with a fixed number of ticks, i.e. after set_ticks() or using a FixedLocator.
  ax.set_yticklabels(['{:g}'.format(x * 100) for x in y_vals])
```



Creates a histogram of the 'Ratio BW/GD' column.

In [49]:
```
ratios_1 = ratios.sort_values(by=['Ratio BW/GD'], ascending=False)
ratios_1.iloc[[0]]
```

Out[49]:

	Birth Weight	Gestational Days	Ratio BW/GD
238	116	148	0.783784

Sorts the ratios DataFrame by 'Ratio BW/GD' in descending order and displays the first row (highest ratio).

In [50]:
```
np.median(ratios.iloc[:,2])
```

Out[50]:
```
0.42907801418439717
```

Calculates and displays the median of the 'Ratio BW/GD' column.

In [51]:
```
def bootstrap_median(original_sample, label, replications):

    """Returns an array of bootstrapped sample medians:
    original_sample: table containing the original sample
    label: label of column containing the variable
    replications: number of bootstrap samples
    """
    just_one_column = original_sample[[label]]
    medians = np.array([])
    for i in np.arange(replications):
        bootstrap_sample = just_one_column.sample(len(just_one_column), replace=True)
        resampled_median = np.percentile(bootstrap_sample, 50)
        medians = np.append(medians, resampled_median)

    return medians
```

Redefines the bootstrap_median function. (This is redundant as it's already defined in Cell 37).

In [52]:
```
# Generate the medians from 5000 bootstrap samples
bstrap_medians = bootstrap_median(ratios, 'Ratio BW/GD', 5000)
```

Generates bootstrapped medians for 'Ratio BW/GD' and stores them in bstrap_medians.

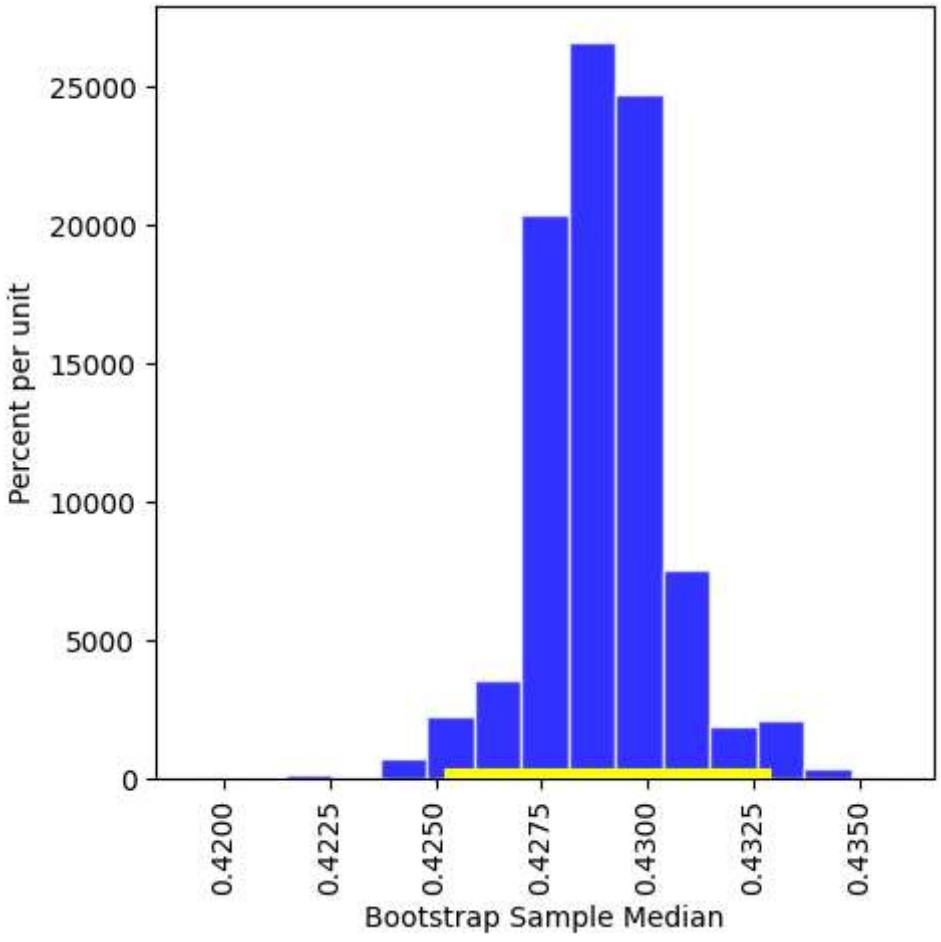
In [53]:
```
# Get the endpoints of the 95% confidence interval
left = np.percentile(bstrap_medians, 2.5, interpolation='nearest')
right = np.percentile(bstrap_medians, 97.5, interpolation='nearest')

np.array([left, right])
```

Out[53]:
```
array([0.42545455, 0.43262411])
```

Calculates the 95% confidence interval for the median 'Ratio BW/GD' using the bootstrapped medians.

In [54]:
```
resampled_medians = pd.DataFrame({'Bootstrap Sample Median':bstrap_medians})
unit = ''
fig, ax = plt.subplots(figsize=(5,5))
ax.hist(resampled_medians, bins=15, density=True, color='blue', alpha=0.8, ec='white', zorder
ax.plot(np.array([left, right]), np.array([0,0]), color='yellow', lw=8, zorder=10)
y_vals = ax.get_yticks()
y_label = 'Percent per ' + (unit if unit else 'unit')
x_label = 'Bootstrap Sample Median'
ax.set_yticklabels(['{:g}'.format(x * 100) for x in y_vals])
plt.ylabel(y_label)
plt.xlabel(x_label)
plt.xticks(rotation=90)
plt.title('');
plt.show()
```

```
<ipython-input-54-b0d5464f15ef>:9: UserWarning: set_ticklabels() should only be used with a fi
xed number of ticks, i.e. after set_ticks() or using a FixedLocator.
  ax.set_yticklabels(['{:g}'.format(x * 100) for x in y_vals])
```

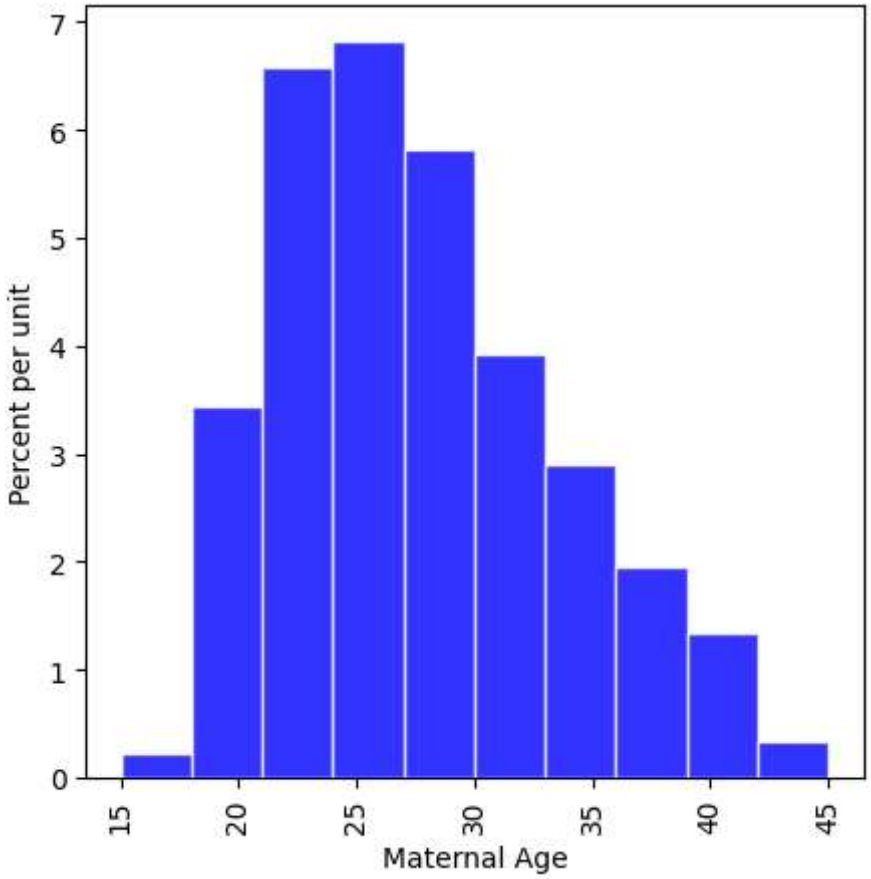


Creates a histogram of the bootstrapped medians, highlighting the 95% confidence interval.

Confidence Interval for a Population Mean: Bootstrap Percentile Method

In [55]:
```
unit = ''
fig, ax = plt.subplots(figsize=(5,5))
ax.hist(baby['Maternal Age'], density=True, color='blue', alpha=0.8, ec='white', zorder=5)
y_vals = ax.get_yticks()
y_label = 'Percent per ' + (unit if unit else 'unit')
x_label = 'Maternal Age'
ax.set_yticklabels(['{:g}'.format(x * 100) for x in y_vals])
plt.ylabel(y_label)
plt.xlabel(x_label)
plt.xticks(rotation=90)
plt.title('');
plt.show()
```

```
<ipython-input-55-79fb60fadbc9>:7: UserWarning: set_ticklabels() should only be used with a fi
xed number of ticks, i.e. after set_ticks() or using a FixedLocator.
  ax.set_yticklabels(['{:g}'.format(x * 100) for x in y_vals])
```



Creates a histogram of 'Maternal Age' from the baby DataFrame.

In [56]:
```
np.mean(baby['Maternal Age'])
```

Out[56]:
```
27.228279386712096
```

Calculates and displays the mean of 'Maternal Age'.

In [57]:
```
def bootstrap_mean(original_sample, label, replications):

    """Returns an array of bootstrapped sample means:
    original_sample: table containing the original sample
    label: label of column containing the variable
    replications: number of bootstrap samples
    """
```

```
    just_one_column = original_sample[[label]]
    means = np.array([])
    for i in np.arange(replications):
        bootstrap_sample = just_one_column.sample(len(just_one_column), replace=True)
        resampled_mean = np.mean(bootstrap_sample.iloc[:,0])
        means = np.append(means, resampled_mean)

    return means
```

Defines a function bootstrap_mean to generate bootstrapped sample means.

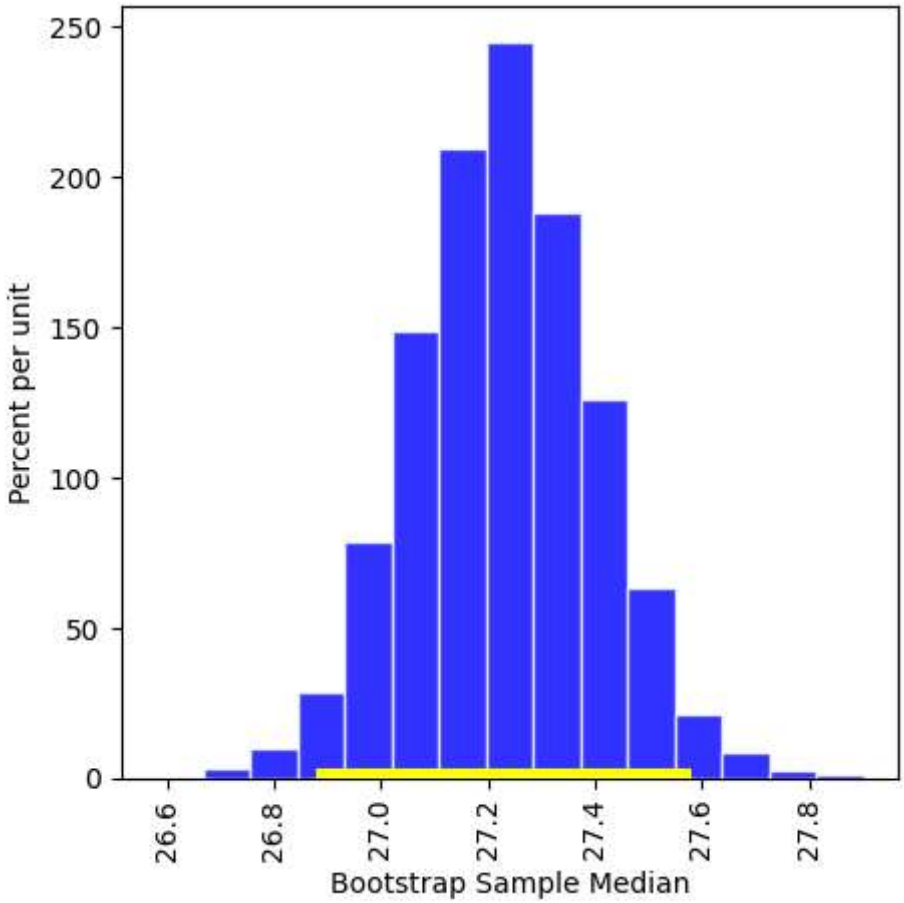
In [58]:
```
# Generate the means from 5000 bootstrap samples
bstrap_means = bootstrap_mean(baby, 'Maternal Age', 5000)

# Get the endpoints of the 95% confidence interval
left = np.percentile(bstrap_means, 2.5, interpolation='nearest')
right = np.percentile(bstrap_means, 97.5, interpolation='nearest')

np.array([left, right])
```

Out[58]:
```
array([26.89778535, 27.55792164])
```

Generates bootstrapped means for 'Maternal Age' and calculates the 95% confidence interval.

In [59]:
```
resampled_means = pd.DataFrame({'Bootstrap Sample Mean':bstrap_means})
unit = ''
fig, ax = plt.subplots(figsize=(5,5))
ax.hist(resampled_means, bins=15, density=True, color='blue', alpha=0.8, ec='white', zorder=5
ax.plot(np.array([left, right]), np.array([0,0]), color='yellow', lw=8, zorder=10)
y_vals = ax.get_yticks()
y_label = 'Percent per ' + (unit if unit else 'unit')
x_label = 'Bootstrap Sample Median'
ax.set_yticklabels(['{:g}'.format(x * 100) for x in y_vals])
plt.ylabel(y_label)
plt.xlabel(x_label)
plt.xticks(rotation=90)
plt.title('');
plt.show()
```

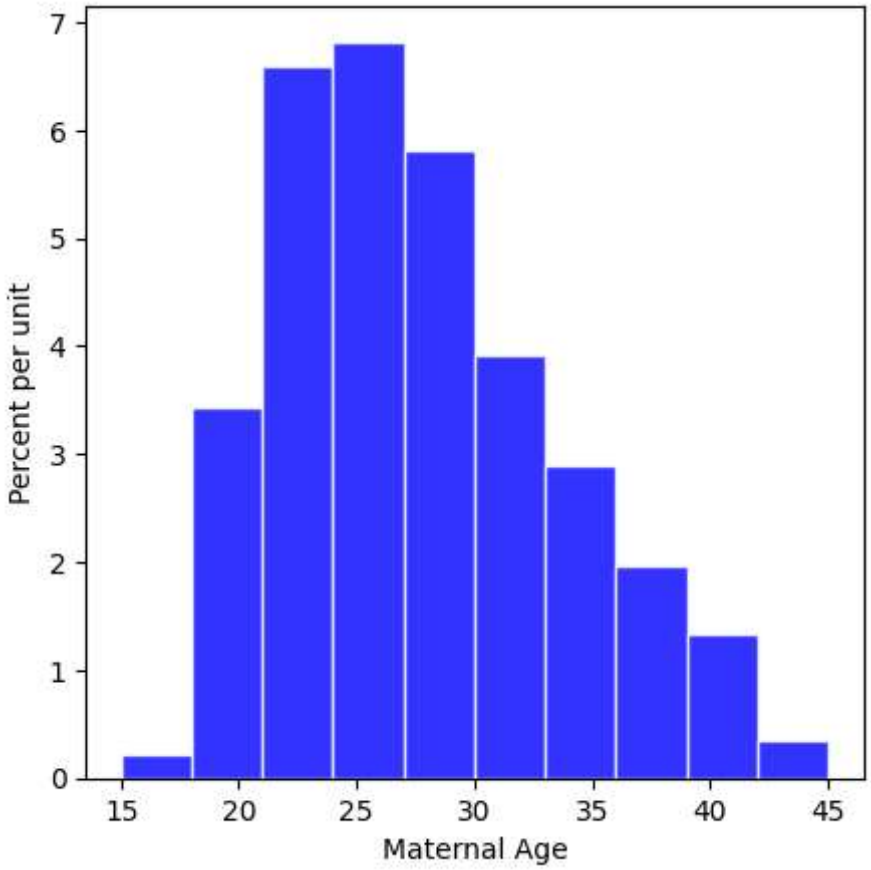
```
<ipython-input-59-274529df0967>:9: UserWarning: set_ticklabels() should only be used with a fi
xed number of ticks, i.e. after set_ticks() or using a FixedLocator.
  ax.set_yticklabels(['{:g}'.format(x * 100) for x in y_vals])
```



Creates a histogram of the bootstrapped means, highlighting the 95% confidence interval.

In [60]:
```
unit = ''
fig, ax = plt.subplots(figsize=(5,5))
ax.hist(baby['Maternal Age'], density=True, color='blue', alpha=0.8, ec='white', zorder=5)
y_vals = ax.get_yticks()
y_label = 'Percent per ' + (unit if unit else 'unit')
x_label = 'Maternal Age'
ax.set_yticklabels(['{:g}'.format(x * 100) for x in y_vals])
plt.ylabel(y_label)
plt.xlabel(x_label)
plt.title('');
plt.show()
```

```
<ipython-input-60-c5d99b9184a9>:7: UserWarning: set_ticklabels() should only be used with a fi
xed number of ticks, i.e. after set_ticks() or using a FixedLocator.
  ax.set_yticklabels(['{:g}'.format(x * 100) for x in y_vals])
```



Creates another histogram of 'Maternal Age' (redundant, similar to Cell 69).

An 80% Confidence Interval

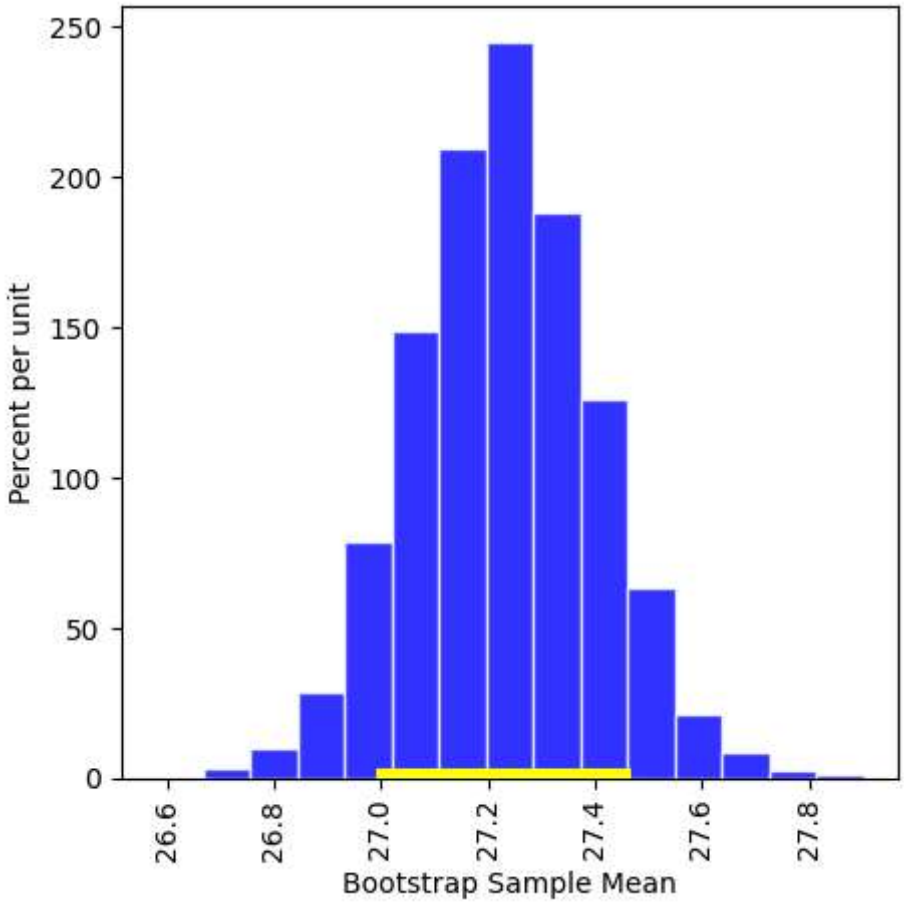
In [61]:
```
left_80 = np.percentile(bstrap_means, 10, interpolation='nearest')
right_80 = np.percentile(bstrap_means, 90, interpolation='nearest')
np.array([left_80, right_80])
```

Out[61]:
```
array([27.01107325, 27.44293015])
```

Calculates the 80% confidence interval for the mean 'Maternal Age' using bootstrapped means.

In [62]:
```
unit = ''
fig, ax = plt.subplots(figsize=(5,5))
ax.hist(resampled_means, bins=15, density=True, color='blue', alpha=0.8, ec='white', zorder=5
ax.plot(np.array([left_80, right_80]), np.array([0,0]), color='yellow', lw=8, zorder=10)
y_vals = ax.get_yticks()
y_label = 'Percent per ' + (unit if unit else 'unit')
x_label = 'Bootstrap Sample Mean'
ax.set_yticklabels(['{:g}'.format(x * 100) for x in y_vals])
plt.ylabel(y_label)
plt.xlabel(x_label)
plt.xticks(rotation=90)
plt.title('');
plt.show()
```

```
<ipython-input-62-12024dae7bf2>:8: UserWarning: set_ticklabels() should only be used with a fi
xed number of ticks, i.e. after set_ticks() or using a FixedLocator.
  ax.set_yticklabels(['{:g}'.format(x * 100) for x in y_vals])
```



Creates a histogram of the bootstrapped means, highlighting the 80% confidence interval.

Confidence Interval for a Population Proportion: Bootstrap Percentile Method

In [63]:
```
len(baby[baby['Maternal Smoker'] == True]) / len(baby)
```

Out[63]:
```
0.3909710391822828
```

Calculates and prints the proportion of 'Maternal Smoker' being True in the baby DataFrame.

In [64]:
```
smoking = baby['Maternal Smoker']
np.count_nonzero(smoking)/len(smoking)
```

Out[64]:
```
0.3909710391822828
```

Calculates and prints the proportion of 'Maternal Smoker' being True using np.count_nonzero. (Similar to Cell 79).

In [65]:
```
def bootstrap_proportion(original_sample, label, replications):

    """Returns an array of bootstrapped sample proportions:
    original_sample: table containing the original sample
    label: label of column containing the Boolean variable
    replications: number of bootstrap samples
    """

    just_one_column = original_sample[[label]]
    proportions = np.array([])
    for i in np.arange(replications):
        bootstrap_sample = just_one_column.sample(len(just_one_column), replace=True)
        resample_array = bootstrap_sample.iloc[:,0]
        resampled_proportion = np.count_nonzero(resample_array)/len(resample_array)
        proportions = np.append(proportions, resampled_proportion)
```

```
    return proportions
```

Defines a function bootstrap_proportion to generate bootstrapped sample proportions.

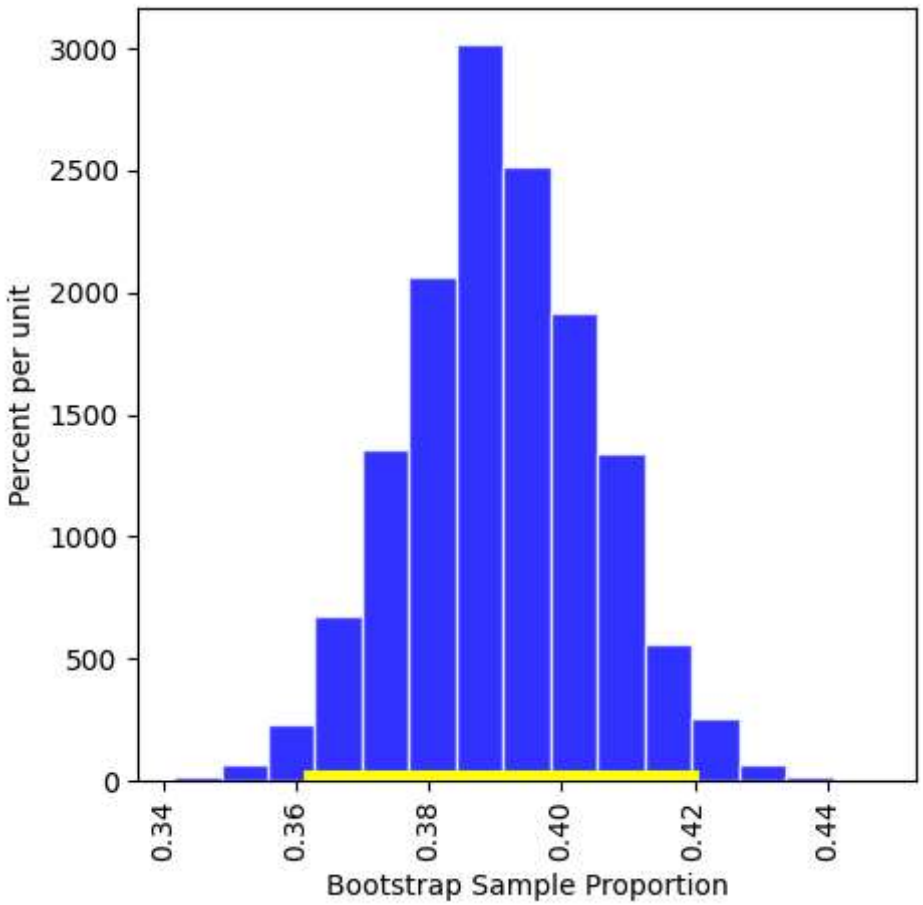
In [66]:
```
# Generate the proportions from 5000 bootstrap samples
bstrap_props = bootstrap_proportion(baby, 'Maternal Smoker', 5000)

# Get the endpoints of the 95% confidence interval
left = np.percentile(bstrap_props, 2.5, interpolation='nearest')
right = np.percentile(bstrap_props, 97.5, interpolation='nearest')

np.array([left, right])
```

Out[66]:
```
array([0.36286201, 0.41908007])
```

Generates bootstrapped proportions for 'Maternal Smoker' and calculates the 95% confidence interval.

In [67]:
```
resampled_proportions = pd.DataFrame({'Bootstrap Sample Proportion':bstrap_props})
unit = ''
fig, ax = plt.subplots(figsize=(5,5))
ax.hist(resampled_proportions, bins=15, density=True, color='blue', alpha=0.8, ec='white', zo
ax.plot(np.array([left, right]), np.array([0,0]), color='yellow', lw=8, zorder=10)
y_vals = ax.get_yticks()
y_label = 'Percent per ' + (unit if unit else 'unit')
x_label = 'Bootstrap Sample Proportion'
ax.set_yticklabels(['{:g}'.format(x * 100) for x in y_vals])
plt.ylabel(y_label)
plt.xlabel(x_label)
plt.xticks(rotation=90)
plt.title('');
plt.show()
```

```
<ipython-input-67-db9c047bb263>:9: UserWarning: set_ticklabels() should only be used with a fi
xed number of ticks, i.e. after set_ticks() or using a FixedLocator.
  ax.set_yticklabels(['{:g}'.format(x * 100) for x in y_vals])
```



Creates a histogram of the bootstrapped proportions, highlighting the 95% confidence interval.

Care in Using the Bootstrap

In [68]:
```
def bootstrap_median(original_sample, label, replications):
    """Returns an array of bootstrapped sample medians:
    original_sample: table containing the original sample
    label: label of column containing the variable
    replications: number of bootstrap samples
    """
    just_one_column = original_sample[label]
    medians = np.array([])
    for i in np.arange(replications):
        bootstrap_sample = just_one_column.sample(len(just_one_column), replace=True)
        resampled_median = np.percentile(bootstrap_sample, 50)
        medians = np.append(medians, resampled_median)

    return medians
```

In [69]:
```
def bootstrap_mean(original_sample, label, replications):

    """Returns an array of bootstrapped sample means:
    original_sample: table containing the original sample
    label: label of column containing the variable
    replications: number of bootstrap samples
    """

    just_one_column = original_sample[[label]]
    means = np.array([])
    for i in np.arange(replications):
        bootstrap_sample = just_one_column.sample(len(just_one_column), replace=True)
        resampled_mean = np.mean(bootstrap_sample.iloc[:,0])
        means = np.append(means, resampled_mean)

    return means
```

In [70]:
```
def bootstrap_proportion(original_sample, label, replications):

    """Returns an array of bootstrapped sample proportions:
    original_sample: table containing the original sample
    label: label of column containing the Boolean variable
    replications: number of bootstrap samples
    """

    just_one_column = original_sample[[label]]
    proportions = np.array([])
    for i in np.arange(replications):
        bootstrap_sample = just_one_column.sample(len(just_one_column), replace=True)
        resample_array = bootstrap_sample.iloc[:,0]
        resampled_proportion = np.count_nonzero(resample_array)/len(resample_array)
        proportions = np.append(proportions, resampled_proportion)

    return proportions
```

Redefines the bootstrap_median, bootstrap_mean, and bootstrap_proportion functions (redundant, already defined earlier).

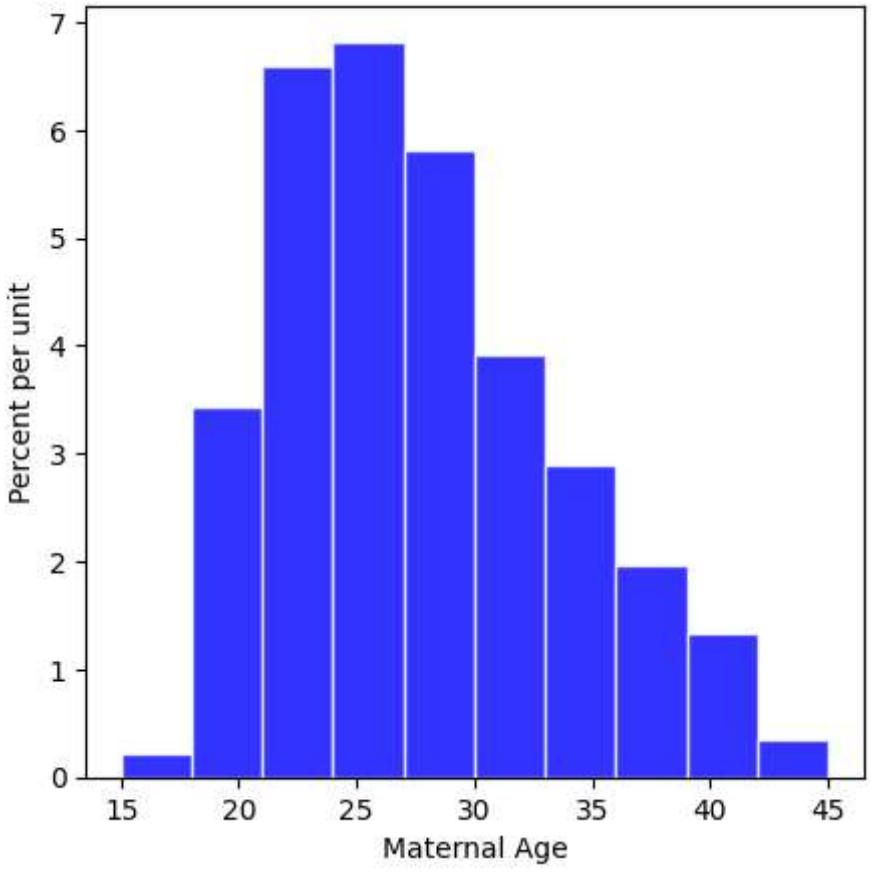
4. Using Confidence Intervals

In [71]:
```
baby = pd.read_csv('baby.csv')
```

Reads data from 'baby.csv' into the baby DataFrame (redundant, already loaded in Cell 58).

In [72]:
```
unit = ''
fig, ax = plt.subplots(figsize=(5,5))
ax.hist(baby['Maternal Age'], density=True, color='blue', alpha=0.8, ec='white', zorder=5)
y_vals = ax.get_yticks()
y_label = 'Percent per ' + (unit if unit else 'unit')
x_label = 'Maternal Age'
ax.set_yticklabels(['{:g}'.format(x * 100) for x in y_vals])
plt.ylabel(y_label)
plt.xlabel(x_label)
plt.title('');
plt.show()
```

```
<ipython-input-72-c5d99b9184a9>:7: UserWarning: set_ticklabels() should only be used with a fi
xed number of ticks, i.e. after set_ticks() or using a FixedLocator.
  ax.set_yticklabels(['{:g}'.format(x * 100) for x in y_vals])
```



Creates a histogram of 'Maternal Age' (redundant, similar to Cell 69 and 74).

Using a Confidence Interval to Test Hypotheses

In [73]:
```
hodgkins = pd.read_csv('hodgkins.csv')
hodgkins
```

Out[73]:

	Unnamed: 0	height	rad	chemo	base	month15
0	0	164	679	180	160.57	87.77
1	1	168	311	180	98.24	67.62
2	2	173	388	239	129.04	133.33
3	3	157	370	168	85.41	81.28
4	4	160	468	151	67.94	79.26
5	5	170	341	96	150.51	80.97
6	6	163	453	134	129.88	69.24
7	7	175	529	264	87.45	56.48
8	8	185	392	240	149.84	106.99
9	9	178	479	216	92.24	73.43
10	10	179	376	160	117.43	101.61
11	11	181	539	196	129.75	90.78
12	12	173	217	204	97.59	76.38
13	13	166	456	192	81.29	67.66
14	14	170	252	150	98.29	55.51
15	15	165	622	162	118.98	90.92
16	16	173	305	213	103.17	79.74
17	17	174	566	198	94.97	93.08
18	18	173	322	119	85.00	41.96
19	19	173	270	160	115.02	81.12
20	20	183	259	241	125.02	97.18
21	21	188	238	252	137.43	113.20

Reads data from 'hodgkins.csv' into a DataFrame called hodgkins.

In [74]:
```
hodgkins['drop'] = hodgkins['base'] - hodgkins['month15']
```

Creates a new column 'drop' in the hodgkins DataFrame, calculated as the difference between 'base' and 'month15'.

In [75]:
```
hodgkins
```

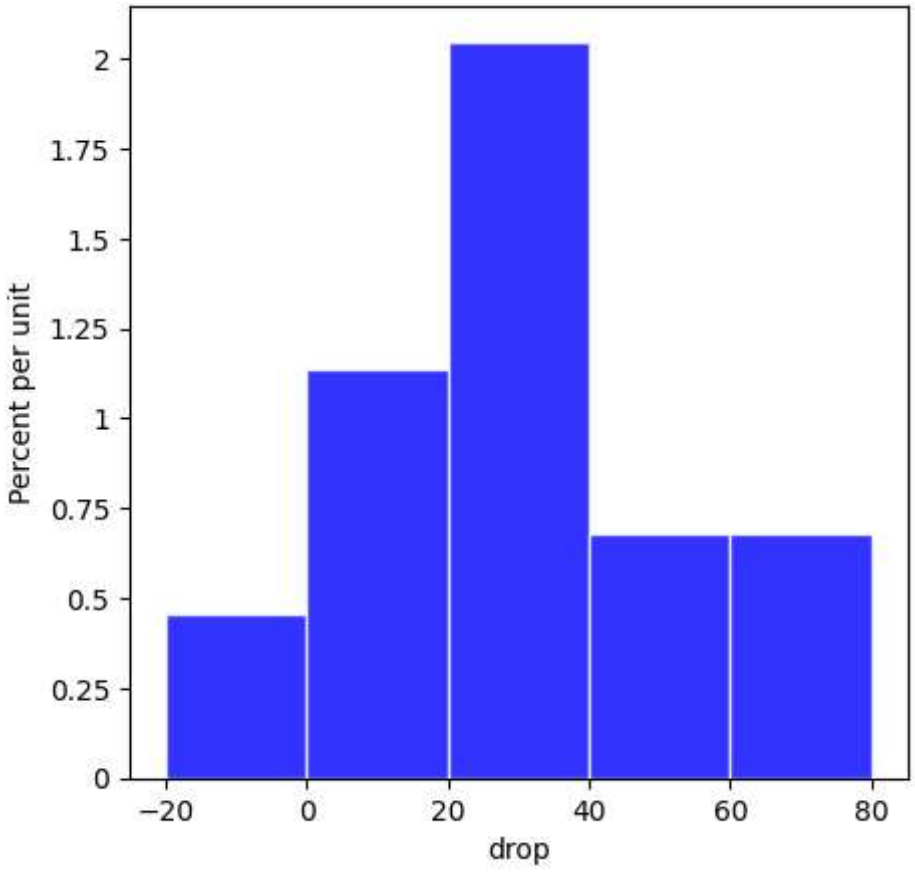
Out[75]:

	Unnamed: 0	height	rad	chemo	base	month15	drop
0	0	164	679	180	160.57	87.77	72.80
1	1	168	311	180	98.24	67.62	30.62
2	2	173	388	239	129.04	133.33	-4.29
3	3	157	370	168	85.41	81.28	4.13
4	4	160	468	151	67.94	79.26	-11.32
5	5	170	341	96	150.51	80.97	69.54
6	6	163	453	134	129.88	69.24	60.64
7	7	175	529	264	87.45	56.48	30.97
8	8	185	392	240	149.84	106.99	42.85
9	9	178	479	216	92.24	73.43	18.81
10	10	179	376	160	117.43	101.61	15.82
11	11	181	539	196	129.75	90.78	38.97
12	12	173	217	204	97.59	76.38	21.21
13	13	166	456	192	81.29	67.66	13.63
14	14	170	252	150	98.29	55.51	42.78
15	15	165	622	162	118.98	90.92	28.06
16	16	173	305	213	103.17	79.74	23.43
17	17	174	566	198	94.97	93.08	1.89
18	18	173	322	119	85.00	41.96	43.04
19	19	173	270	160	115.02	81.12	33.90
20	20	183	259	241	125.02	97.18	27.84
21	21	188	238	252	137.43	113.20	24.23

Displays the hodgkins DataFrame.

In [76]:
```
unit = ''
fig, ax = plt.subplots(figsize=(5,5))
ax.hist(hodgkins['drop'], bins=np.arange(-20, 81, 20), density=True, color='blue', alpha=0.8,
y_vals = ax.get_yticks()
y_label = 'Percent per ' + (unit if unit else 'unit')
x_label = 'drop'
ax.set_yticklabels(['{:g}'.format(x * 100) for x in y_vals])
plt.ylabel(y_label)
plt.xlabel(x_label)
plt.title('');
plt.show()
```

```
<ipython-input-76-15cdd25c2af7>:7: UserWarning: set_ticklabels() should only be used with a fi
xed number of ticks, i.e. after set_ticks() or using a FixedLocator.
  ax.set_yticklabels(['{:g}'.format(x * 100) for x in y_vals])
```



Creates a histogram of the 'drop' column.

In [77]:
```
np.mean(hodgkins['drop'])
```

Out[77]:
```
28.615909090909096
```

Calculates and prints the mean of the 'drop' column.

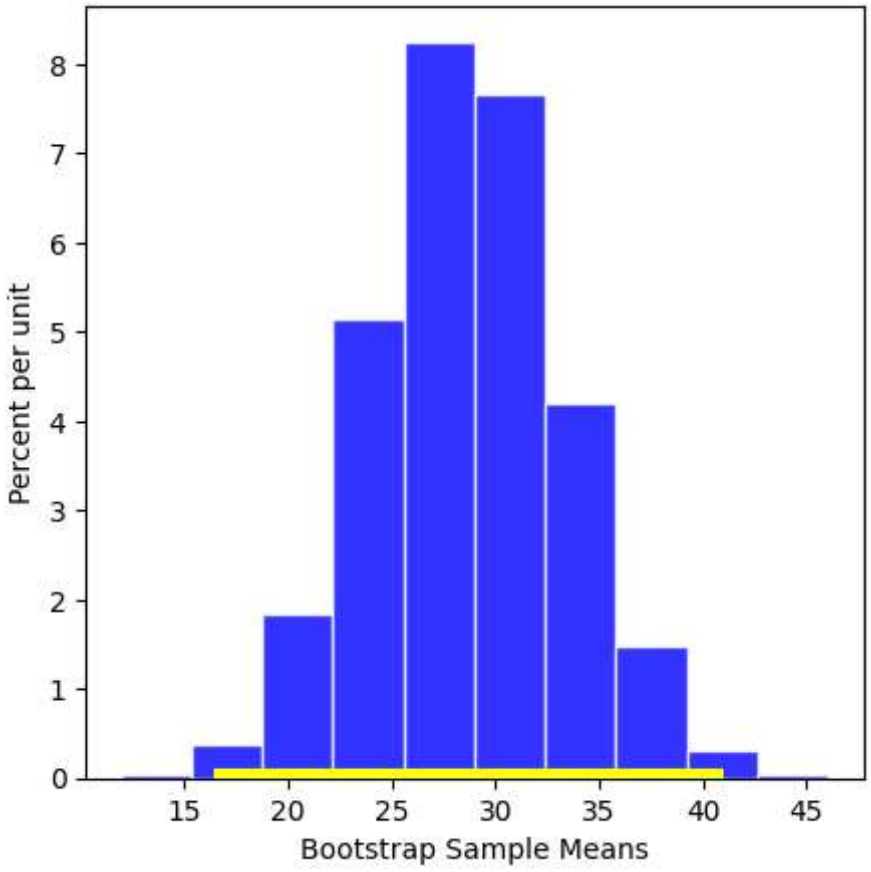
In [78]:
```
bstrap_means = bootstrap_mean(hodgkins, 'drop', 10000)

left = np.percentile(bstrap_means, 0.5, interpolation='nearest')
right = np.percentile(bstrap_means, 99.5, interpolation='nearest')

np.array([left, right])
```

Out[78]:
```
array([16.92863636, 40.38272727])
```

Generates bootstrapped means for 'drop' and calculates the 99% confidence interval.

In [79]:
```
resampled_means = pd.DataFrame({'Bootstrap Sample Mean': bstrap_means})
unit = ''
fig, ax = plt.subplots(figsize=(5,5))
ax.hist(resampled_means, density=True, color='blue', alpha=0.8, ec='white', zorder=5)
ax.plot(np.array([left, right]), np.array([0,0]), color='yellow', lw=8, zorder=10)
y_vals = ax.get_yticks()
y_label = 'Percent per ' + (unit if unit else 'unit')
x_label = 'Bootstrap Sample Means'
ax.set_yticklabels(['{:g}'.format(x * 100) for x in y_vals])
plt.ylabel(y_label)
plt.xlabel(x_label)
plt.title('');
plt.show()
```

```
<ipython-input-79-f326279c66fd>:9: UserWarning: set_ticklabels() should only be used with a fi
xed number of ticks, i.e. after set_ticks() or using a FixedLocator.
  ax.set_yticklabels(['{:g}'.format(x * 100) for x in y_vals])
```



Creates a histogram of the bootstrapped means, highlighting the 99% confidence interval.

Foundations of Data Science

Name: Krishna GSVV

Roll no. AV.EN.U4CSE22016

Lab 13 (Prediction and Classification)

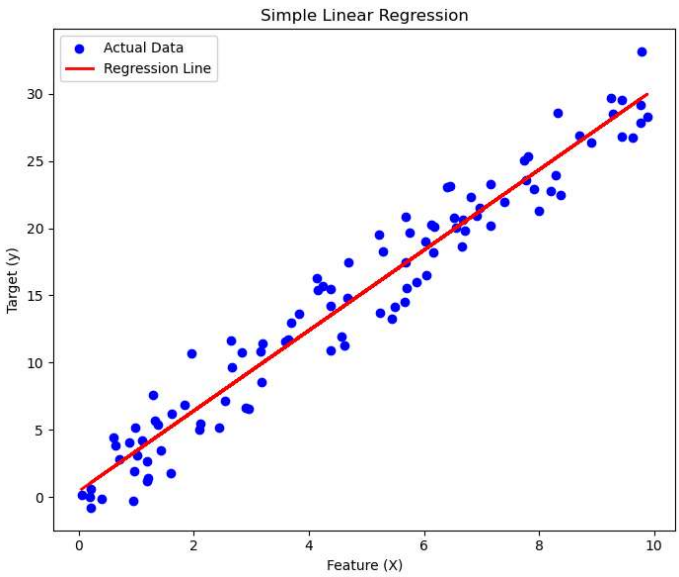
The code demonstrates the implementation and evaluation of both simple and multiple linear regression models using Python's scikit-learn library. It generates sample datasets, trains the models, makes predictions, and assesses their performance using metrics like Mean Squared Error and R2 Score. Additionally, it visualizes the results to provide a graphical representation of the relationships between features and target variables.

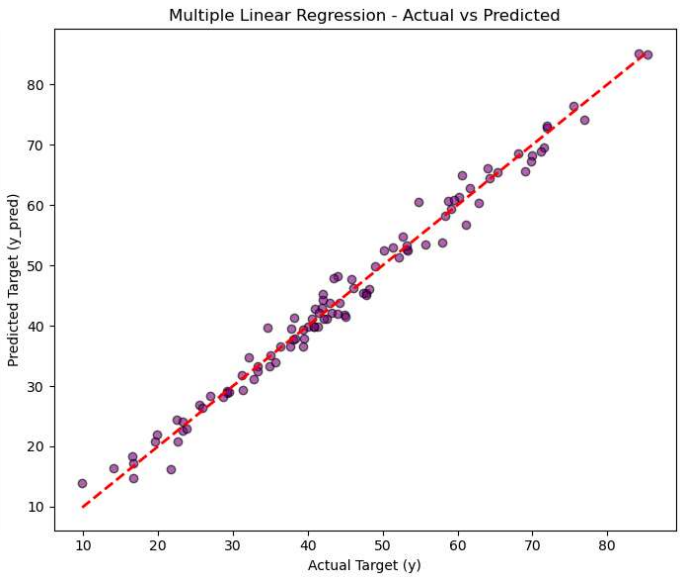
In essence, the code provides a comprehensive workflow for building and evaluating linear regression models for both single and multiple predictor variables.

In [ ]:
```
# Import necessary libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score

# Generate a sample dataset
np.random.seed(0)
X_single = np.random.rand(100, 1) * 10  # Single feature
y_single = 3 * X_single.flatten() + np.random.randn(100) * 2  # y = 3x + noise

X_multi = np.random.rand(100, 3) * 10  # Three features
y_multi = 2 * X_multi[:, 0] + 3 * X_multi[:, 1] + 4 * X_multi[:, 2] + np.random.randn(100) *
```

```python
# Simple Linear Regression
lin_reg_single = LinearRegression()
lin_reg_single.fit(X_single, y_single)
y_pred_single = lin_reg_single.predict(X_single)

# Multiple Linear Regression
lin_reg_multi = LinearRegression()
lin_reg_multi.fit(X_multi, y_multi)
y_pred_multi = lin_reg_multi.predict(X_multi)

# Evaluation metrics
print("Simple Linear Regression")
print("Coefficients:", lin_reg_single.coef_)
print("Intercept:", lin_reg_single.intercept_)
print("Mean Squared Error:", mean_squared_error(y_single, y_pred_single))
print("R2 Score:", r2_score(y_single, y_pred_single))

print("\nMultiple Linear Regression")
print("Coefficients:", lin_reg_multi.coef_)
print("Intercept:", lin_reg_multi.intercept_)
print("Mean Squared Error:", mean_squared_error(y_multi, y_pred_multi))
print("R2 Score:", r2_score(y_multi, y_pred_multi))

# Plotting Simple Linear Regression Results
plt.figure(figsize=(14, 6))

# Simple Linear Regression
plt.subplot(1, 2, 1)
plt.scatter(X_single, y_single, color="blue", label="Actual Data")
plt.plot(X_single, y_pred_single, color="red", linewidth=2, label="Regression Line")
plt.xlabel("Feature (X)")
plt.ylabel("Target (y)")
plt.title("Simple Linear Regression")
plt.legend()

# Plotting Multiple Linear Regression Results
plt.subplot(1, 2, 2)
plt.scatter(y_multi, y_pred_multi, color="purple", alpha=0.6, edgecolors="k")
plt.plot([y_multi.min(), y_multi.max()], [y_multi.min(), y_multi.max()], 'r--', linewidth=2)
plt.xlabel("Actual Target (y)")
plt.ylabel("Predicted Target (y_pred)")
plt.title("Multiple Linear Regression - Actual vs Predicted")

plt.tight_layout()
plt.show()
```

```
Simple Linear Regression
Coefficients: [2.987387]
Intercept: 0.444302154894455
Mean Squared Error: 3.9697545948985944
R2 Score: 0.9492021559001605

Multiple Linear Regression
Coefficients: [2.0346571  2.97432503 4.02284203]
Intercept: -0.12318663252225548
Mean Squared Error: 4.405434930185427
R2 Score: 0.9837952480562537
```





Classfication

In [40]:
```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import roc_curve, roc_auc_score, classification_report
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
```

In [42]:
```
data = pd.read_csv("titanic.csv")
```

Reading the data set of titanic

In [44]:
```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 12 columns):
 #   Column       Non-Null Count  Dtype  
---  ------       --------------  -----  
 0   PassengerId  891 non-null    int64  
 1   Survived     891 non-null    int64  
 2   Pclass       891 non-null    int64  
 3   Name         891 non-null    object 
 4   Sex          891 non-null    object 
 5   Age          714 non-null    float64
 6   SibSp        891 non-null    int64  
 7   Parch        891 non-null    int64  
 8   Ticket       891 non-null    object 
 9   Fare         891 non-null    float64
 10  Cabin        204 non-null    object 
 11  Embarked     889 non-null    object 
dtypes: float64(2), int64(5), object(5)
memory usage: 83.7+ KB
```

producing the dataset information data type

In [46]:
```
data.isnull().sum()
```

Out[46]:
```
PassengerId      0
Survived         0
Pclass           0
Name             0
Sex              0
Age            177
SibSp            0
Parch            0
Ticket           0
Fare             0
Cabin          687
Embarked         2
dtype: int64
```

checking the Null values of the data set

In [48]:
```
data.head()
```

Out[48]:

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Er
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	

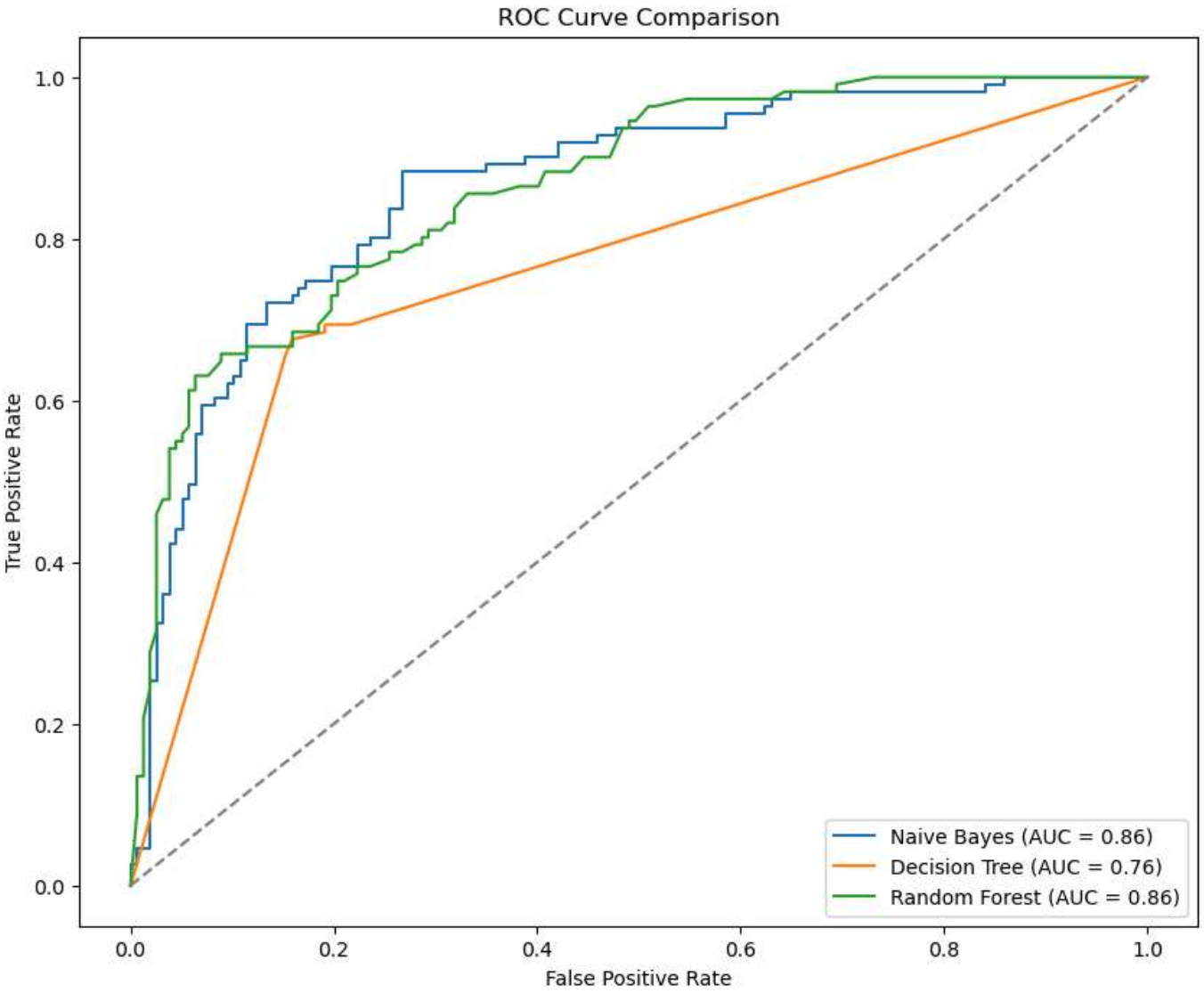
printing the top 5 data in dataset

In [ ]:
```
# Preprocess data
# Select relevant features and target
X = data[['Pclass', 'Sex', 'Age', 'Fare', 'SibSp', 'Parch']]
y = data['Survived']

# Handle missing values
X['Age'].fillna(X['Age'].median(), inplace=True)

# Encode categorical variable 'Sex'
X = pd.get_dummies(X, columns=['Sex'], drop_first=True)

# Split data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Standardize features
```

```
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Initialize models
models = {
    "Naive Bayes": GaussianNB(),
    "Decision Tree": DecisionTreeClassifier(random_state=42),
    "Random Forest": RandomForestClassifier(random_state=42)
}

# Plot ROC curves for each model
plt.figure(figsize=(10, 8))
for model_name, model in models.items():
    # Fit the model
    model.fit(X_train, y_train)

    # Predict probabilities
    y_probs = model.predict_proba(X_test)[:, 1]
    y_pred = model.predict(X_test)

    # Compute ROC curve and AUC
    fpr, tpr, thresholds = roc_curve(y_test, y_probs)
    roc_auc = roc_auc_score(y_test, y_probs)

    # Plot ROC curve
    plt.plot(fpr, tpr, label=f'{model_name} (AUC = {roc_auc:.2f})')


    print(f"Classification Report for {model_name}:\n")
    print(classification_report(y_test, y_pred))
    print("\n" + "="*60 + "\n")

# Plot settings
plt.plot([0, 1], [0, 1], color='gray', linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve Comparison')
plt.legend(loc='lower right')
plt.show()
```

```
Classification Report for Naive Bayes:

              precision    recall  f1-score   support

           0       0.82      0.83      0.82       157
           1       0.75      0.74      0.75       111

    accuracy                           0.79       268
   macro avg       0.78      0.78      0.78       268
weighted avg       0.79      0.79      0.79       268


============================================================

Classification Report for Decision Tree:

              precision    recall  f1-score   support

           0       0.79      0.84      0.81       157
           1       0.75      0.68      0.71       111

    accuracy                           0.77       268
   macro avg       0.77      0.76      0.76       268
weighted avg       0.77      0.77      0.77       268


============================================================

Classification Report for Random Forest:

              precision    recall  f1-score   support

           0       0.79      0.84      0.81       157
           1       0.75      0.68      0.72       111

    accuracy                           0.78       268
   macro avg       0.77      0.76      0.77       268
weighted avg       0.77      0.78      0.77       268


============================================================
```

```
/var/folders/w1/kjg7ry3x3tg4f6_6_w2h745c0000gp/T/ipykernel_3401/2353502941.py:7: FutureWarnin
g: A value is trying to be set on a copy of a DataFrame or Series through chained assignment u
sing an inplace method.
The behavior will change in pandas 3.0. This inplace method will never work because the interm
ediate object on which we are setting values always behaves as a copy.

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.method({col: valu
e}, inplace=True)' or df[col] = df[col].method(value) instead, to perform the operation inplac
e on the original object.


  X['Age'].fillna(X['Age'].median(), inplace=True)
/var/folders/w1/kjg7ry3x3tg4f6_6_w2h745c0000gp/T/ipykernel_3401/2353502941.py:7: SettingWithCo
pyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/
indexing.html#returning-a-view-versus-a-copy
  X['Age'].fillna(X['Age'].median(), inplace=True)
```



This code preprocesses Titanic dataset features (`X`) and target (`y`), including handling missing values, encoding categorical variables, splitting data, and standardizing features. It trains multiple models (Naive Bayes, Decision Tree, Random Forest), evaluates them using ROC curves and classification reports, and visualizes their ROC curves with corresponding AUC scores.

Name: Krishna GSVV
Roll no. AV.EN.U4CSE22016
Assignment - 1

In [44]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

Introduction to the Dataset

The dataset provided is named `iphone.csv` . It likely contains data related to iPhones, potentially including features such as model specifications, prices, release dates, sales, or user ratings. Before diving into the tasks, we will first explore and preprocess this dataset to understand its structure and clean it as necessary. Following this, we will apply different visualization and analytical techniques to uncover patterns, relationships, and insights within the data.

Task 1: Understanding and Exploring the Dataset

Explanation:

In this task, the dataset is loaded using the `pandas` library, and basic exploration functions like `head()`, `info()`, and `describe()` are used to get an overview of the dataset's structure, data types, and statistical summary. These functions provide insights into the dataset's columns, missing data, and general distributions.

Code:

In [45]:
```
import pandas as pd

# Load the dataset
df = pd.read_csv("cities_r2.csv")

# Explore the dataset
display(df.head())
print(df.info())
display(df.describe())
print(df.shape)
```

	name_of_city	state_code	state_name	dist_code	population_total	population_male	population_fe
0	Abohar	3.0	PUNJAB	9.0	145238	76840	
1	Achalpur	27.0	MAHARASHTRA	7.0	112293	58256	
2	Adilabad	28.0	ANDHRA PRADESH	1.0	117388	59232	
3	Adityapur	20.0	JHARKHAND	24.0	173988	91495	
4	Adoni	28.0	ANDHRA PRADESH	21.0	166537	82743	

5 rows × 22 columns

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 493 entries, 0 to 492
Data columns (total 22 columns):
 #   Column                          Non-Null Count  Dtype
---  ------                          --------------  -----
 0   name_of_city                    493 non-null    object
 1   state_code                      492 non-null    float64
 2   state_name                      493 non-null    object
 3   dist_code                       492 non-null    float64
 4   population_total                493 non-null    int64
 5   population_male                 493 non-null    int64
 6   population_female               493 non-null    int64
 7   0-6_population_total            493 non-null    int64
 8   0-6_population_male             493 non-null    int64
 9   0-6_population_female           492 non-null    float64
 10  literates_total                 492 non-null    float64
 11  literates_male                  493 non-null    int64
 12  literates_female                493 non-null    int64
 13  sex_ratio                       492 non-null    float64
 14  child_sex_ratio                 493 non-null    int64
 15  effective_literacy_rate_total   493 non-null    float64
 16  effective_literacy_rate_male    492 non-null    float64
 17  effective_literacy_rate_female  492 non-null    float64
 18  location                        491 non-null    object
 19  total_graduates                 493 non-null    int64
 20  male_graduates                  492 non-null    float64
 21  female_graduates                493 non-null    int64
dtypes: float64(9), int64(10), object(3)
memory usage: 84.9+ KB
None
```

	state_code	dist_code	population_total	population_male	population_female	0-6_population_total
count	492.000000	492.000000	4.930000e+02	4.930000e+02	4.930000e+02	4.930000e+02
mean	18.621951	16.768293	4.481124e+05	2.343468e+05	2.137656e+05	4.709285e+04
std	9.294862	15.578563	1.033228e+06	5.487786e+05	4.848622e+05	1.050279e+05
min	1.000000	1.000000	1.000360e+05	5.020100e+04	4.512600e+04	6.547000e+03
25%	9.000000	7.000000	1.261420e+05	6.638400e+04	6.041100e+04	1.363900e+04
50%	19.000000	12.500000	1.841330e+05	9.665500e+04	8.776800e+04	1.944000e+04
75%	27.000000	21.000000	3.490330e+05	1.750550e+05	1.700260e+05	3.794500e+04
max	35.000000	99.000000	1.247845e+07	6.736815e+06	5.741632e+06	1.209275e+06

```
(493, 22)
```

Task 2: Checking for Missing Data

Explanation:

We use the `isnull()` function to identify missing values in the dataset. This helps us decide on appropriate methods to handle missing data in the subsequent task.

Code:

Code to randomly remove data to simulate data inconsistencies

In [46]:
```
# import pandas as pd
# import numpy as np
```

```
# # Load the CSV file into a DataFrame
# df = pd.read_csv('cities_r2.csv')

# # Print the original number of non-NaN values
# print(f"Original number of non-NaN values: {df.notna().sum().sum()}")

# # Randomly select a few data points to set to NaN (e.g., 10 data points)
# num_points_to_nan = 10
# nan_indices = [(np.random.randint(0, df.shape[0]), np.random.randint(0, df.shape[1])) for _

# # Set the selected data points to NaN
# for row, col in nan_indices:
#     df.iat[row, col] = np.nan

# # Print the number of non-NaN values after setting some to NaN
# print(f"Number of non-NaN values after setting some to NaN: {df.notna().sum().sum()}")

# # Save the modified DataFrame back to a CSV file (optional)
# df.to_csv('cities_r2.csv', index=False)
```

In [47]:
```
# Check for missing data
missing_data = df.isnull().sum()

# Display missing data
print(missing_data)
```

```
name_of_city                    0
state_code                      1
state_name                      0
dist_code                       1
population_total                0
population_male                 0
population_female               0
0-6_population_total            0
0-6_population_male             0
0-6_population_female           1
literates_total                 1
literates_male                  0
literates_female                0
sex_ratio                       1
child_sex_ratio                 0
effective_literacy_rate_total   0
effective_literacy_rate_male    1
effective_literacy_rate_female  1
location                        2
total_graduates                 0
male_graduates                  1
female_graduates                0
dtype: int64
```

Task 3: Handling Missing Data and Outliers

Explanation:

Missing data is handled by filling numerical columns with their mean values. To detect and remove outliers, we use the Z-score method, which identifies values that are more than three standard deviations from the mean. The dataset is then filtered to remove these outliers.

Code:

In [48]:
```
# Fill missing values with mean for numerical columns
df.fillna(df.select_dtypes(include=['float64', 'int64']).mean(), inplace=True)
```

```
# Detect outliers using Z-score
from scipy.stats import zscore

# Calculate Z-scores and filter out data points with high Z-scores
z_scores = zscore(df.select_dtypes(include=['float64', 'int64']))
df_cleaned = df[(z_scores < 3).all(axis=1)]

print(df_cleaned.shape)
```

```
(472, 22)
```

Task 4: Converting Categorical Data to Numerical

Explanation:

Categorical columns are converted to numerical format using one-hot encoding. This process transforms categorical variables into a format suitable for machine learning algorithms by creating new binary columns for each category.

Code:

In [49]:
```
# Convert categorical columns using one-hot encoding
df_encoded = pd.get_dummies(df_cleaned, drop_first=True)

display(df_encoded.head())
```

	state_code	dist_code	population_total	population_male	population_female	0-6_population_total	6_pc
0	3.0	9.0	145238	76840	68398	15870	
1	27.0	7.0	112293	58256	54037	11810	
2	28.0	1.0	117388	59232	58156	13103	
3	20.0	24.0	173988	91495	82493	23042	
4	28.0	21.0	166537	82743	83794	18406	

5 rows × 983 columns

Task 5: Feature Scaling (if necessary)

Explanation:

Feature scaling is applied using `StandardScaler` to normalize numerical columns so that they have a mean of 0 and a standard deviation of 1. This ensures that all features are on a similar scale, which is important for many machine learning models.

Code:

In [50]:
```
from sklearn.preprocessing import StandardScaler

# Initialize the scaler
scaler = StandardScaler()

# Apply scaling to numerical columns
scaled_data = scaler.fit_transform(df_encoded)

# Convert scaled data back to DataFrame
df_scaled = pd.DataFrame(scaled_data, columns=df_encoded.columns)
```

```
print(df_scaled.head())
```

```
   state_code  dist_code  population_total  population_male  \
0   -1.690853  -0.534364         -0.489914        -0.476607
1    0.891722  -0.709835         -0.582540        -0.575696
2    0.999329  -1.236249         -0.568215        -0.570492
3    0.138471   0.781671         -0.409083        -0.398467
4    0.999329   0.518464         -0.430032        -0.445132

   population_female  0-6_population_total  0-6_population_male  \
0          -0.503920             -0.476631            -0.462113
1          -0.589193             -0.582488            -0.579721
2          -0.564735             -0.548776            -0.553025
3          -0.420226             -0.289633            -0.291849
4          -0.412501             -0.410509            -0.424494

   0-6_population_female  literates_total  literates_male  ...  \
0              -0.493650        -0.509060       -0.496396  ...
1              -0.586025        -0.548617       -0.556016  ...
2              -0.544375        -0.579424       -0.576234  ...
3              -0.287854        -0.426698       -0.409463  ...
4              -0.395206        -0.516425       -0.510599  ...

   location_34.0836708,74.7972825  location_8.1832857,77.4118996  \
0                       -0.046078                       -0.046078
1                       -0.046078                       -0.046078
2                       -0.046078                       -0.046078
3                       -0.046078                       -0.046078
4                       -0.046078                       -0.046078

   location_8.5241391,76.9366376  location_8.7139126,77.7566523  \
0                      -0.046078                      -0.046078
1                      -0.046078                      -0.046078
2                      -0.046078                      -0.046078
3                      -0.046078                      -0.046078
4                      -0.046078                      -0.046078

   location_8.7641661,78.1348361  location_8.8932118,76.6141396  \
0                      -0.046078                      -0.046078
1                      -0.046078                      -0.046078
2                      -0.046078                      -0.046078
3                      -0.046078                      -0.046078
4                      -0.046078                      -0.046078

   location_9.4653377,77.5275463  location_9.4980667,76.3388484  \
0                      -0.046078                      -0.046078
1                      -0.046078                      -0.046078
2                      -0.046078                      -0.046078
3                      -0.046078                      -0.046078
4                      -0.046078                      -0.046078

   location_9.9252007,78.1197754  location_9.9312328,76.2673041
0                      -0.046078                      -0.046078
1                      -0.046078                      -0.046078
2                      -0.046078                      -0.046078
3                      -0.046078                      -0.046078
4                      -0.046078                      -0.046078

[5 rows x 983 columns]
```

Task 6: Visualizing Relationships Between Features

Explanation:

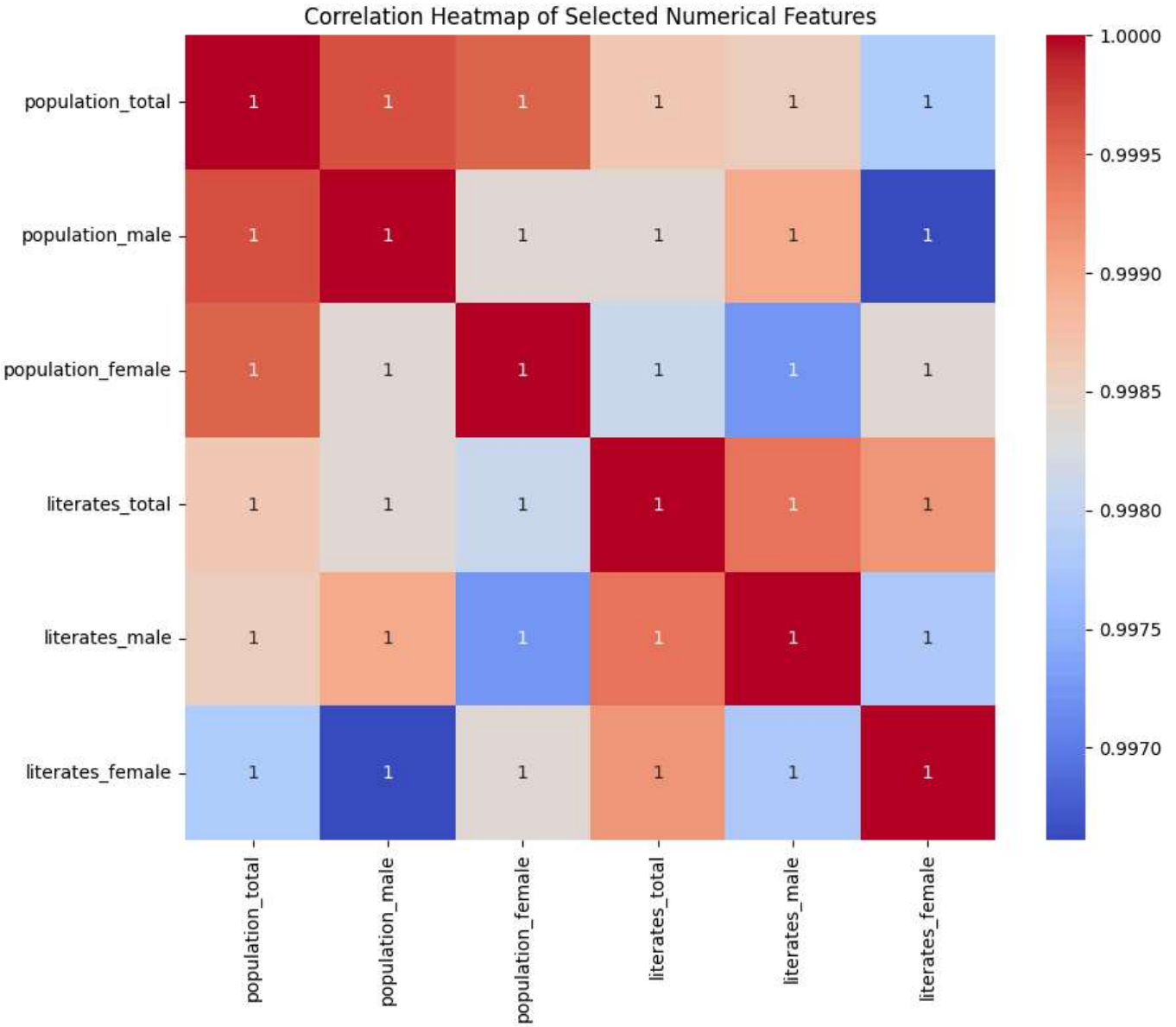
In this task, we will visualize the relationships between different features in the dataset using scatter plots, pair plots, and correlation heatmaps. These visualizations help in identifying patterns, trends, and correlations between features, providing insights into the data.

Code:

In [51]:
```
import seaborn as sns
import matplotlib.pyplot as plt


# Print the columns of the DataFrame to verify their names
print("Columns in DataFrame:", df.columns)
```

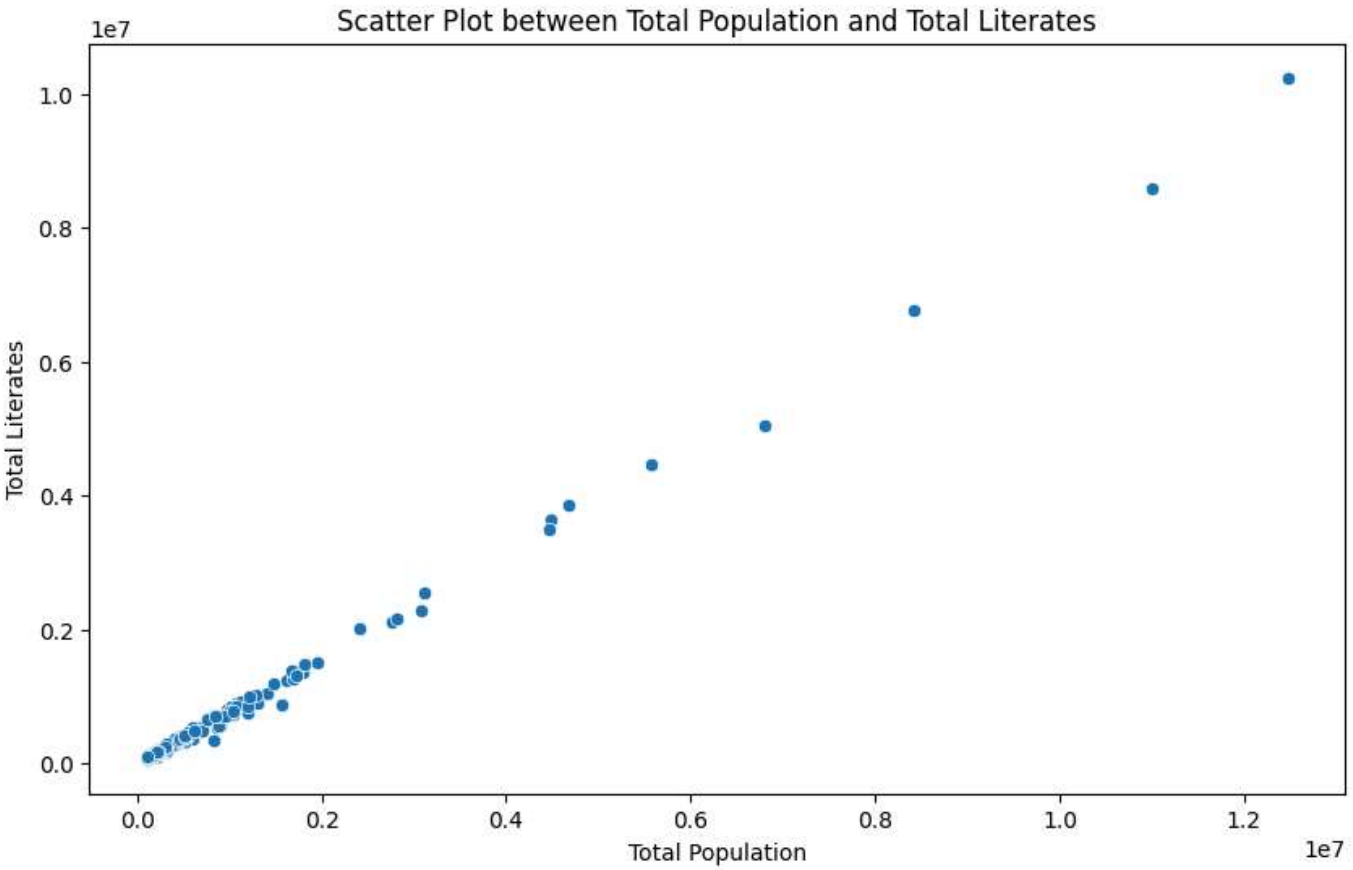
```
Columns in DataFrame: Index(['name_of_city', 'state_code', 'state_name', 'dist_code',
       'population_total', 'population_male', 'population_female',
       '0-6_population_total', '0-6_population_male', '0-6_population_female',
       'literates_total', 'literates_male', 'literates_female', 'sex_ratio',
       'child_sex_ratio', 'effective_literacy_rate_total',
       'effective_literacy_rate_male', 'effective_literacy_rate_female',
       'location', 'total_graduates', 'male_graduates', 'female_graduates'],
      dtype='object')
```

In [52]:
```
# Create a correlation heatmap
plt.figure(figsize=(10, 8))
sns.heatmap(df[['population_total', 'population_male', 'population_female', 'literates_total'
plt.title('Correlation Heatmap of Selected Numerical Features')
plt.show()
```



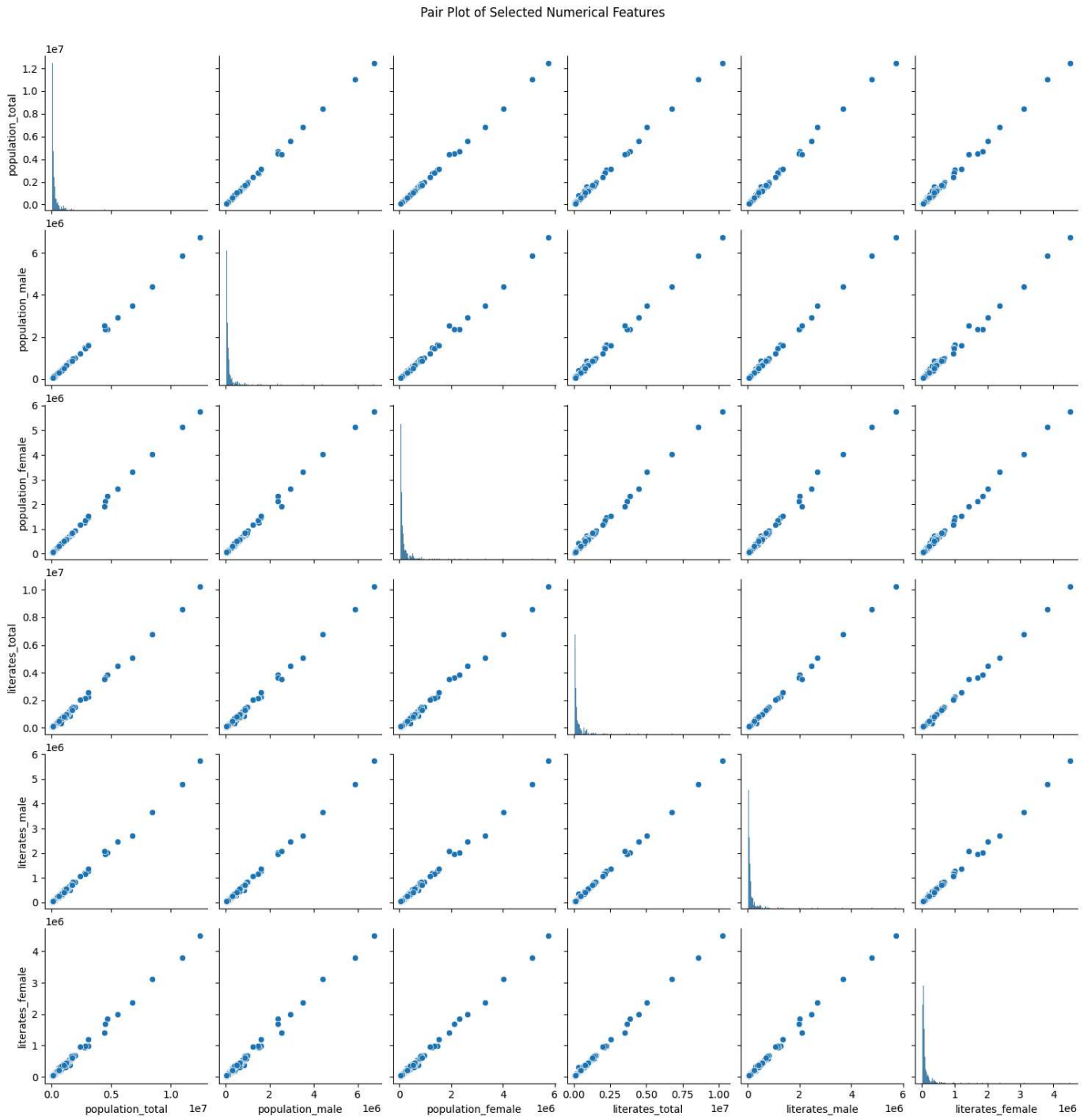
Scatter Plots

In [53]:
```
# Scatter plot for 'population_total' and 'literates_total'
plt.figure(figsize=(10, 6))
sns.scatterplot(data=df, x='population_total', y='literates_total')
plt.title('Scatter Plot between Total Population and Total Literates')
plt.xlabel('Total Population')
plt.ylabel('Total Literates')
plt.show()
```



Pair Plots

In [54]:
```
# Pair plot for selected numerical features
sns.pairplot(df[['population_total', 'population_male', 'population_female', 'literates_total
plt.suptitle('Pair Plot of Selected Numerical Features', y=1.02)
plt.show()
```



Task 7: Visualizing Distributions of Features

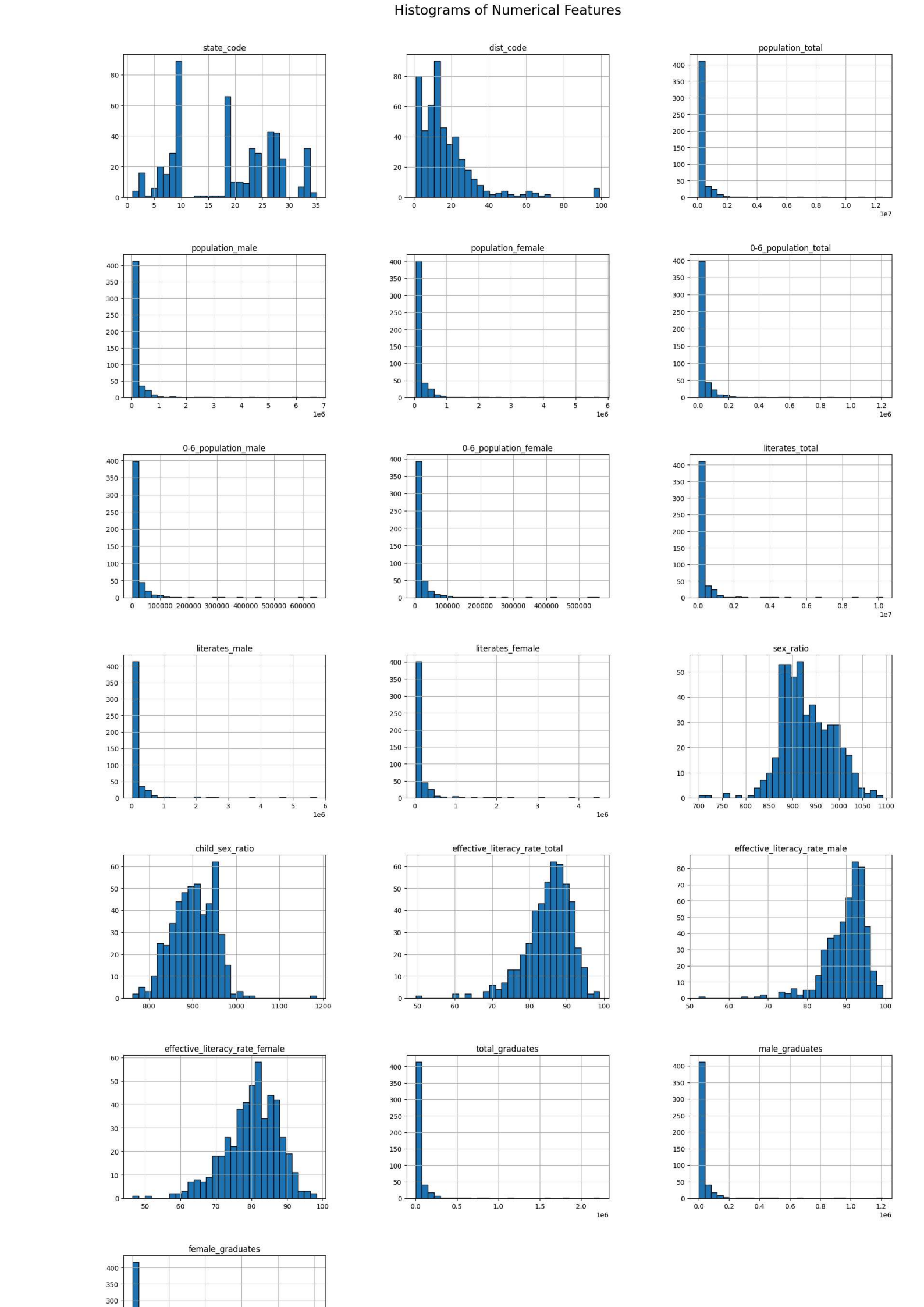
Explanation:

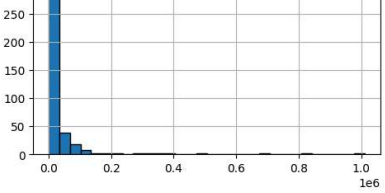
Histograms are used to visualize the distributions of individual numerical features. This allows us to observe whether the data is normally distributed, skewed, or contains outliers.

Code:

In [55]:
```
# Optimize data types
for col in df.select_dtypes(include=['float64']).columns:
    df[col] = df[col].astype('float32')

# Sample a subset of the data (e.g., 10,000 rows)
df_sampled = df.sample(n=10000, random_state=1) if len(df) > 10000 else df

# Get the list of numerical columns
numerical_cols = df_sampled.select_dtypes(include=[np.number]).columns

# Calculate the number of rows and columns needed for the subplots
num_cols = len(numerical_cols)
```

```
num_rows = int(np.ceil(num_cols / 3))  # Adjust the number of columns per row as needed

# Create subplots dynamically
fig, axes = plt.subplots(nrows=num_rows, ncols=3, figsize=(20, num_rows * 5))  # Adjust the f
axes = axes.flatten()

# Plot histograms for numerical features
for i, col in enumerate(numerical_cols):
    df_sampled[col].hist(ax=axes[i], bins=30, edgecolor='black')
    axes[i].set_title(col)

# Remove any unused subplots
for j in range(i + 1, len(axes)):
    fig.delaxes(axes[j])

# Adjust spacing between subplots
plt.subplots_adjust(left=0.1, right=0.9, top=0.9, bottom=0.1, hspace=0.4, wspace=0.4)

# Add a title to the entire figure
fig.suptitle('Histograms of Numerical Features', y=0.93, fontsize = 20)

plt.show()
```


Histograms of Numerical Features
state_code
dist_code
population_total
population_male
population_female
0-6_population_total
0-6_population_male
0-6_population_female
literates_total
literates_male
literates_female
sex_ratio
child_sex_ratio
effective_literacy_rate_total
effective_literacy_rate_male
effective_literacy_rate_female
total_graduates
male_graduates
female_graduates



Task 8: Detecting Outliers Using Visualization

Explanation:

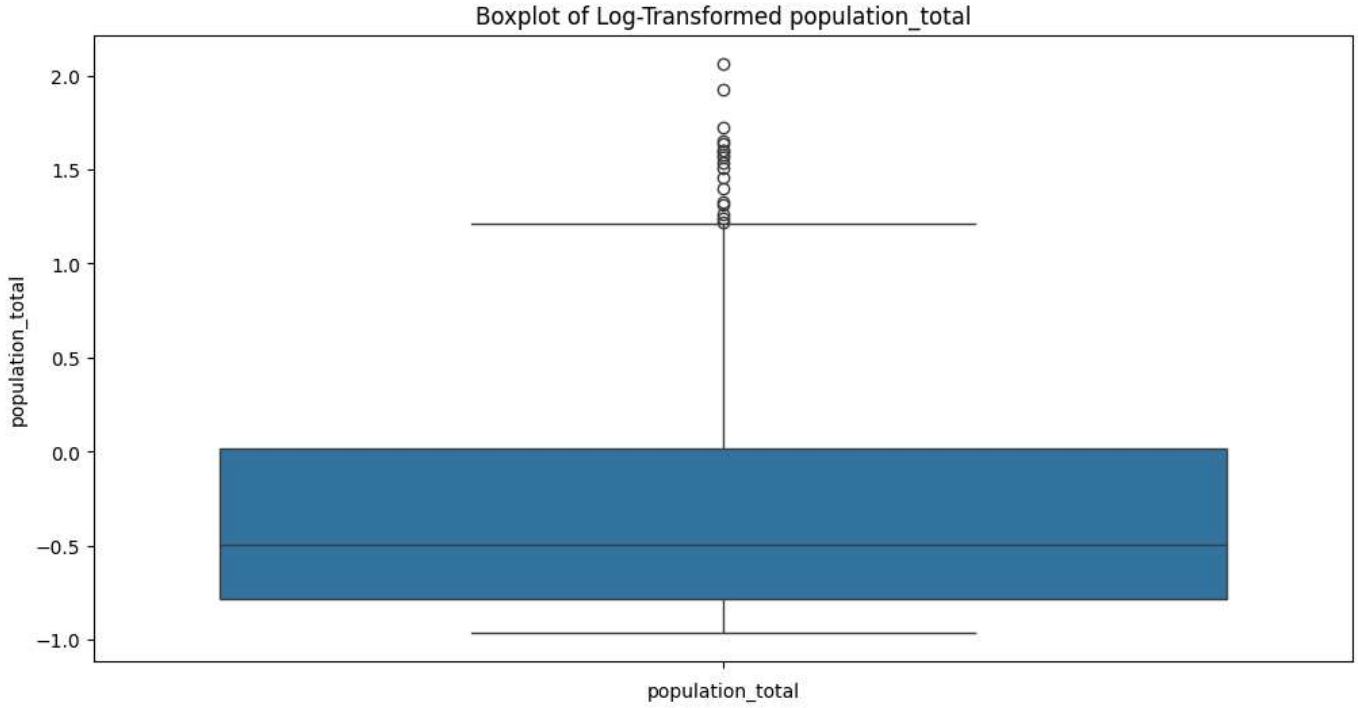
Boxplots are used to visualize the presence of outliers in the dataset. The plots help in identifying extreme values that could affect the analysis.

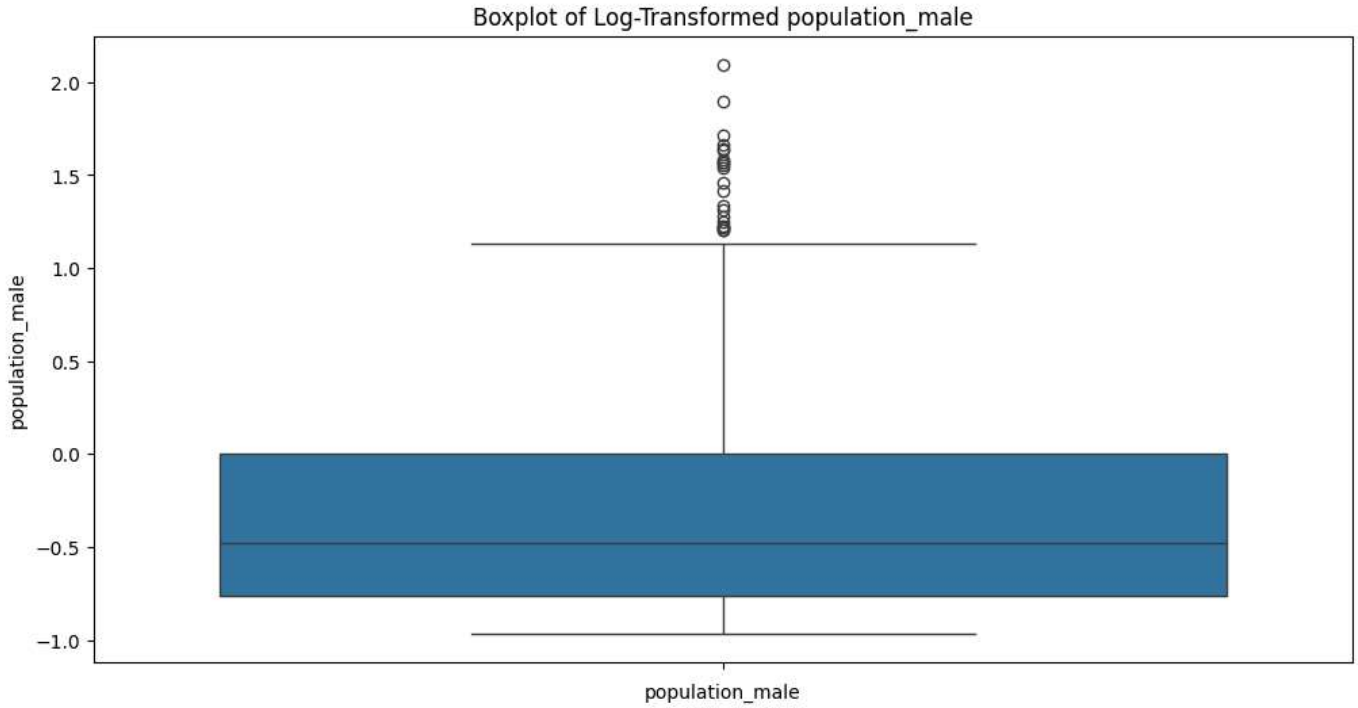
Code:

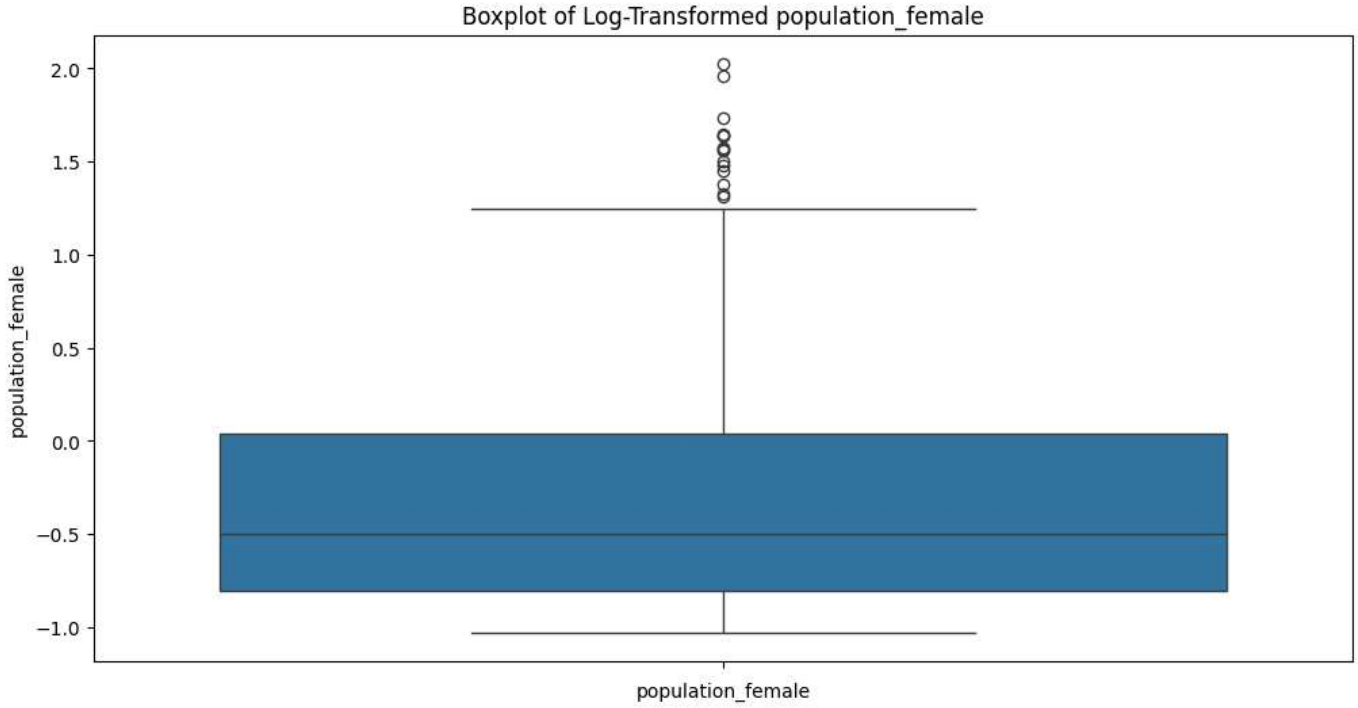
In [56]:
```
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np
```

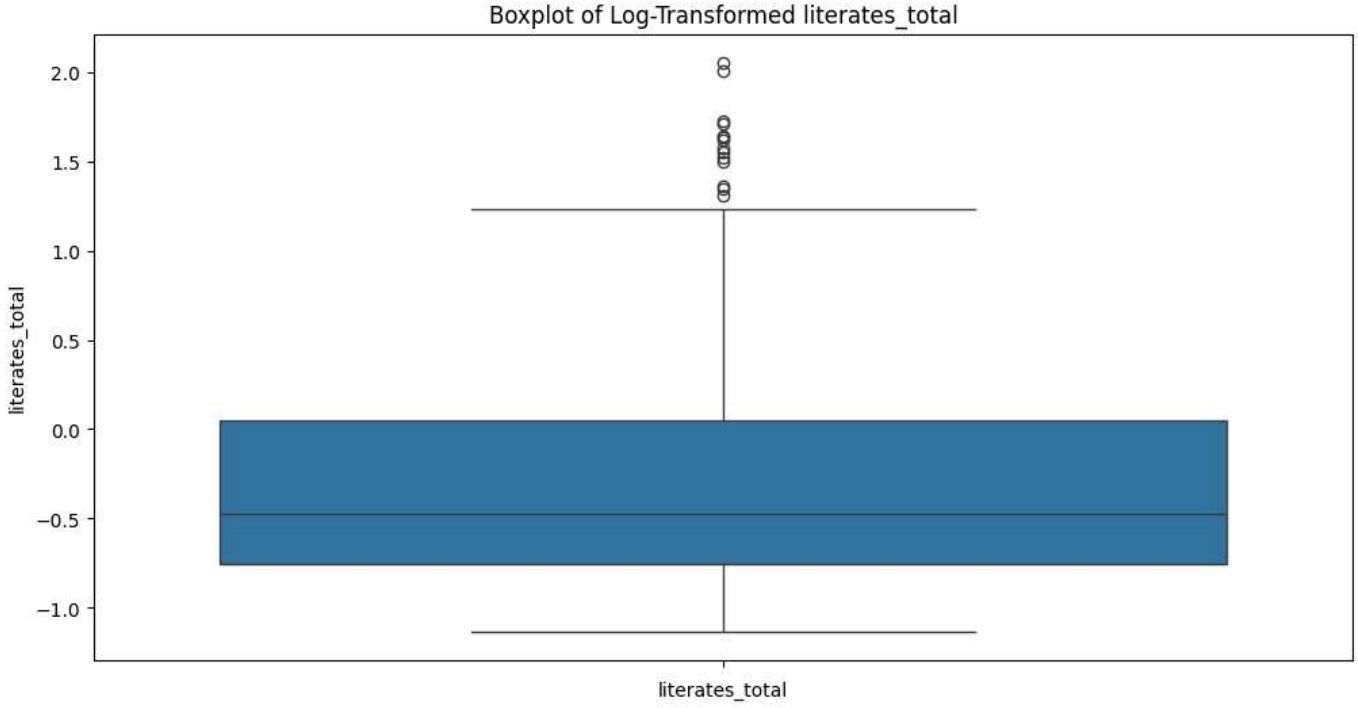
In [57]:
```
# Apply log transformation to the data
df_log_transformed = df_scaled.apply(lambda x: np.log1p(x) if np.issubdtype(x.dtype, np.numbe

# Plot boxplots for individual features after log transformation
features_to_plot = ['population_total', 'population_male', 'population_female', 'literates_to

plt.figure(figsize=(12, 8))
for feature in features_to_plot:
    plt.figure(figsize=(12, 6))
    sns.boxplot(data=df_log_transformed[feature])
    plt.title(f'Boxplot of Log-Transformed {feature}')
    plt.xlabel(feature)
    plt.show()
```

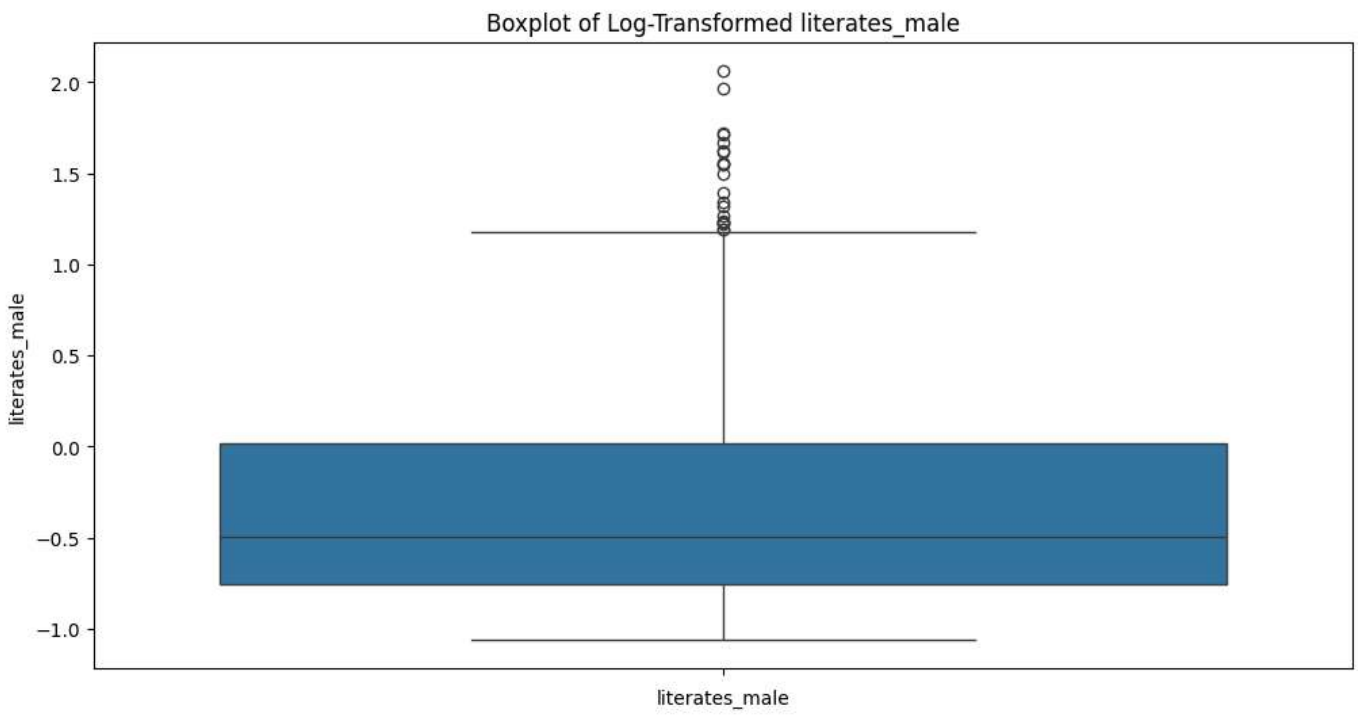
```
c:\Program Files\Python312\Lib\site-packages\pandas\core\arraylike.py:399: RuntimeWarning: inv
alid value encountered in log1p
  result = getattr(ufunc, method)(*inputs, **kwargs)
<Figure size 1200x800 with 0 Axes>
```

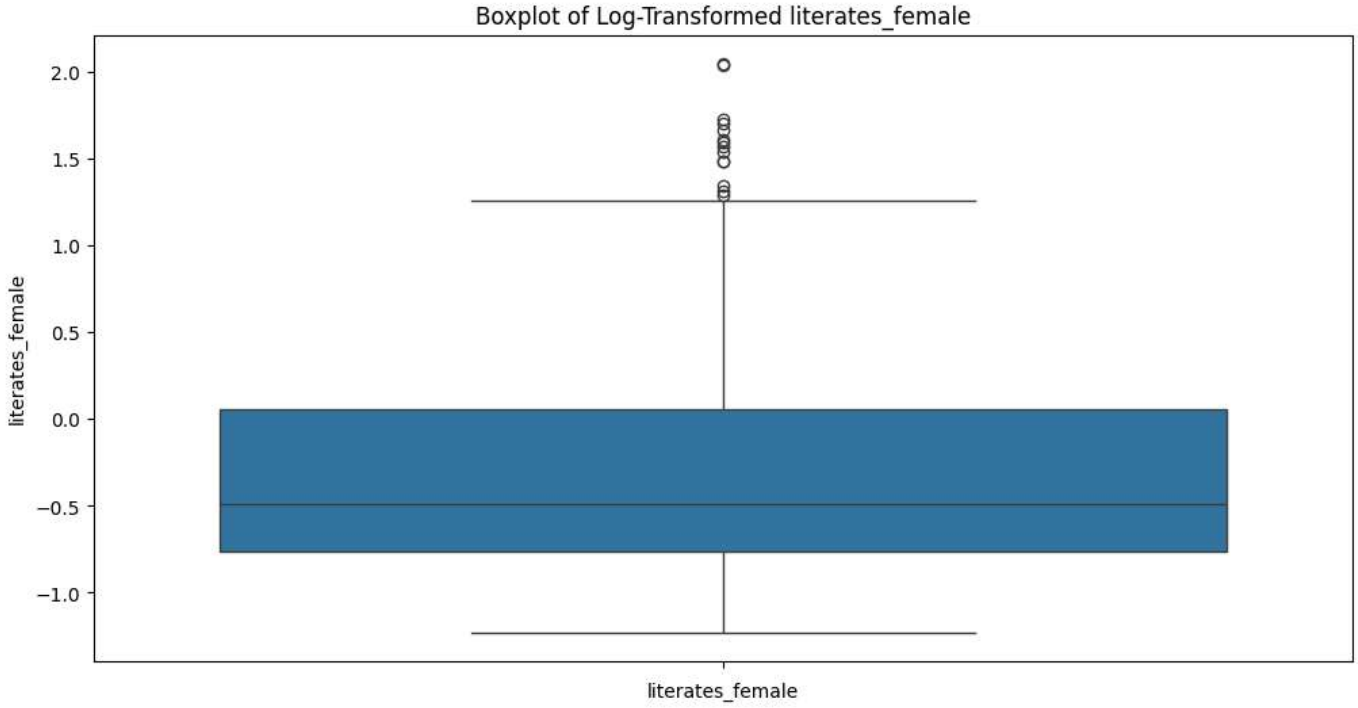



Boxplot of Log-Transformed population_male
population_male
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
population_male


Boxplot of Log-Transformed population_female
population_female
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
population_female


Boxplot of Log-Transformed literates_total
literates_total
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
literates_total





Task 9: Visualizing Trends or Time-Based Data

Explanation:

If the dataset contains time-series data, line plots are created to visualize trends over time. This helps in identifying patterns such as seasonality, growth, or decline in specific features over a period.

Code:

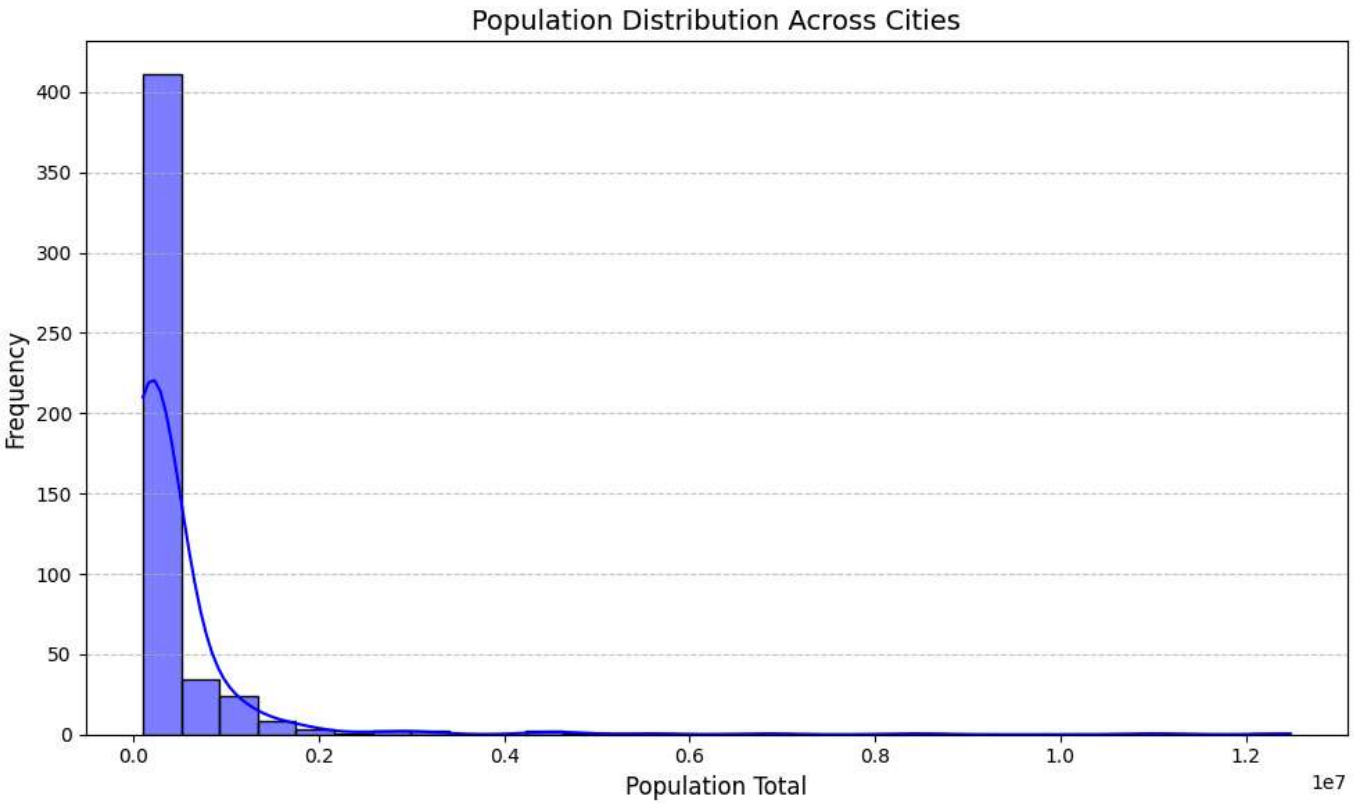
This database does not require time-based visualizations

In [58]:
```
# # If time-based data is available, plot a line graph
# if 'date_column' in df.columns:  # Replace 'date_column' with the actual time column name
#     plt.figure(figsize=(10, 6))
#     plt.plot(df['date_column'], df['target_column'])  # Replace 'target_column' with the fe
#     plt.xlabel('Date')
#     plt.ylabel('Target Feature')
#     plt.title('Trend over Time')
#     plt.show()
```

Task 10: Summary of Insights

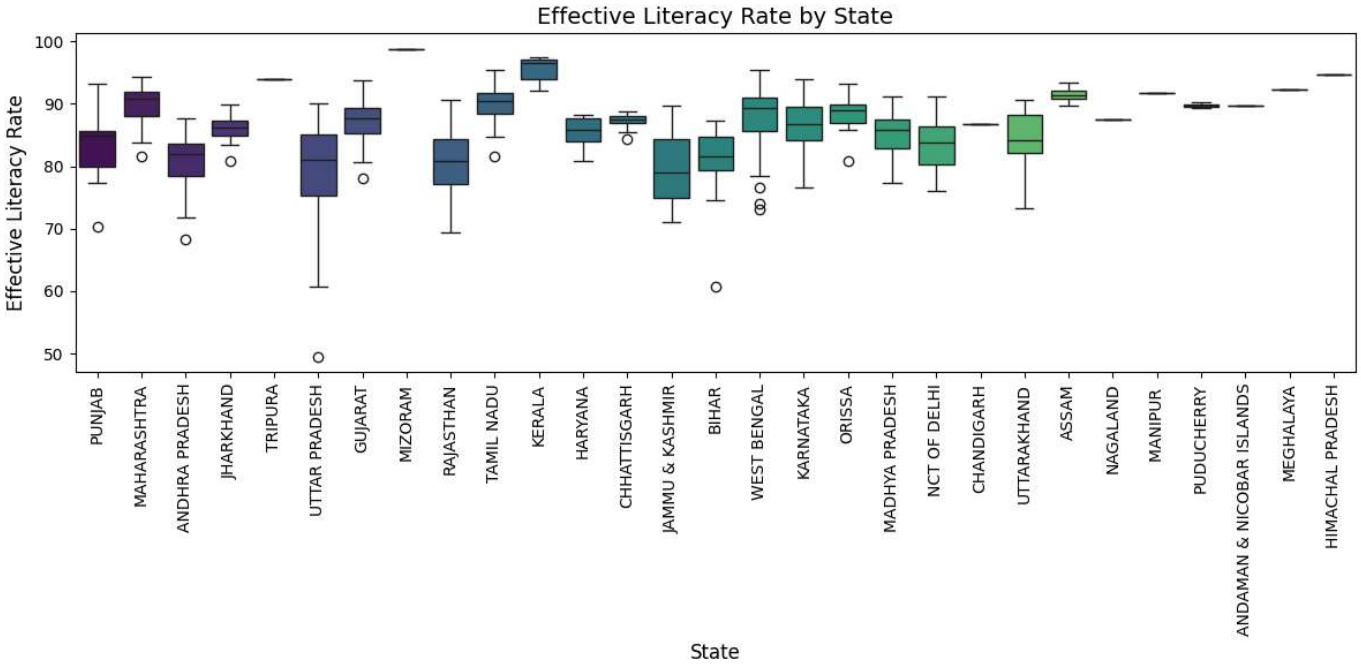
Through the preprocessing and visualization tasks, I learned the importance of understanding and cleaning the dataset before applying any analysis. Handling missing data ensures that no critical information is lost, while outlier detection helps in maintaining data integrity. The conversion of categorical data and feature scaling are crucial for making the dataset compatible with machine learning algorithms. Visualizing relationships and distributions provides a deeper understanding of how the features interact with each other. Finally, summarizing insights allows us to reflect on the key findings and patterns that can guide further analysis.

In [59]:
```
# Population Distribution Chart
plt.figure(figsize=(10, 6))
sns.histplot(df['population_total'], bins=30, kde=True, color='blue')
plt.title('Population Distribution Across Cities', fontsize=14)
plt.xlabel('Population Total', fontsize=12)
plt.ylabel('Frequency', fontsize=12)
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.tight_layout()
plt.show()
```

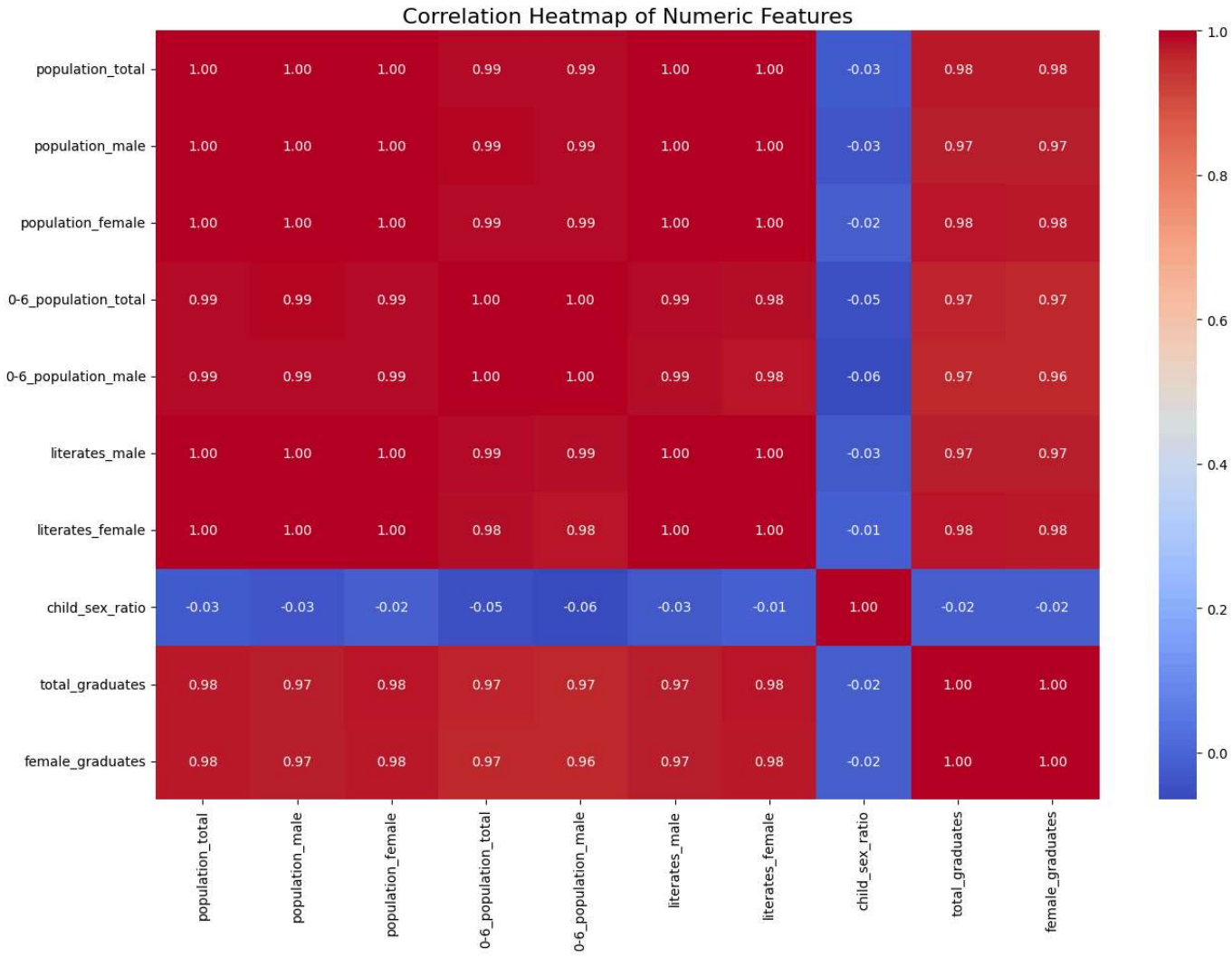


In [60]:
```
# Literacy Rate Comparison (Male vs Female by State)
plt.figure(figsize=(12, 6))
sns.boxplot(data=df, x='state_name', y='effective_literacy_rate_total', palette="viridis")
plt.xticks(rotation=90)
plt.title('Effective Literacy Rate by State', fontsize=14)
plt.xlabel('State', fontsize=12)
plt.ylabel('Effective Literacy Rate', fontsize=12)
plt.tight_layout()
plt.show()
```

```
C:\Users\krish\AppData\Local\Temp\ipykernel_32412\1281340485.py:3: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign the `x` variable to `hue` and set `legend=False` for the same effect.

  sns.boxplot(data=df, x='state_name', y='effective_literacy_rate_total', palette="viridis")
```



In [61]:
```
# Correlation Heatmap of Numeric Features
plt.figure(figsize=(14, 10))
numeric_columns = df.select_dtypes(include=['float64', 'int64']).columns
corr_matrix = df[numeric_columns].corr()
sns.heatmap(corr_matrix, annot=True, fmt='.2f', cmap='coolwarm', cbar=True)
plt.title('Correlation Heatmap of Numeric Features', fontsize=16)
plt.tight_layout()
plt.show()
```



In [ ]:
```
# Define the columns to include in the heatmap
columns_to_include = [
    "population_total",
```

```
    "population_male",
    "population_female",
    "0-6_population_total",
    "0-6_population_male",
    "0-6_population_female",
    "literates_total",
    "literates_male",
    "literates_female",
    "sex_ratio",
    "child_sex_ratio",
    "effective_literacy_rate_total",
    "effective_literacy_rate_male",
    "effective_literacy_rate_female",
    "total_graduates",
    "male_graduates",
    "female_graduates",
]

# Create a correlation matrix for the specified columns
corr_matrix = df[columns_to_include].corr()

# Plot the heatmap
plt.figure(figsize=(14, 10))
sns.heatmap(corr_matrix, annot=True, fmt=".2f", cmap="coolwarm", cbar=True)
plt.title("Correlation Heatmap of Selected Features", fontsize=16)
plt.tight_layout()
plt.show()
```