Experiment - 1.1

AIM:

Write a program that asks the user for a weight in kilograms and converts it to pounds. There are 2.2 pounds in a kilogram.

Description:

The purpose of this program is to convert a weight value from kilograms to pounds. The user is asked to input the weight in kilograms. The program uses the standard conversion factor, multiplies the entered value by 2.2 to obtain the equivalent weight in pounds, and then displays the result.

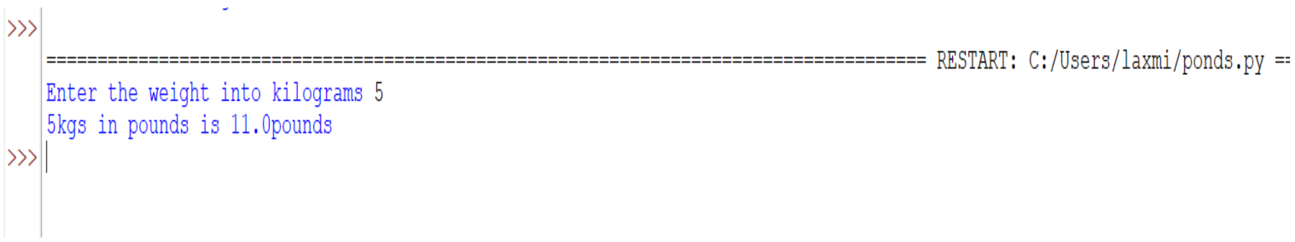
Procedure / Algorithm:

1. Start the program.
2. Prompt the user to input a weight in kilograms.
3. Read the user's input and store it as a floating-point number.
4. Multiply the input value by 2.2 to convert it to pounds.
5. Display the calculated weight in pounds.
6. End the program.

Source Code:

```
val=int(input("Enter the weight into kilograms"))

result=val*2.2

print(f"{val}kgs in pounds is {result}pounds")
```

Input and Output:



```
>>>
    ==================================================================================== RESTART: C:/Users/laxmi/ponds.py ==
    Enter the weight into kilograms 5
    5kgs in pounds is 11.0pounds
>>>
```

Result: The program correctly converts kilograms to pounds.

Experiment - 1.2

AIM:

Write a program that uses a for loop to print the numbers 8, 11, 14, 17, 20, . . . , 83, 86, 89.

Description:

This program prints a sequence of numbers starting at 8 and increasing by 3 each time, up to 89. Such a sequence, where the difference between consecutive terms is constant, is called an arithmetic progression. The sequence generated will be: 8, 11, 14, ..., 86, 89.

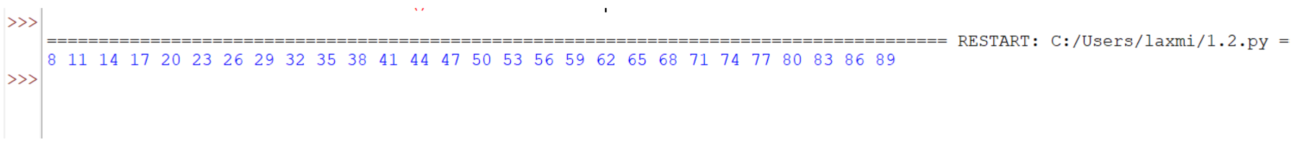
Procedure / Algorithm:

1. **Initialize** the starting number (first term) as 8.
2. **Set** the common difference as 3.
3. **Start** a loop from 8, incrementing by 3 each time.
4. **Continue** looping and printing each number until the value exceeds 89.

Source Code :

```
for n in range(8,90,3):
   print(n,end=" ")
```

Input and Output:



```
>>>
    ======================================================================================= RESTART: C:/Users/laxmi/1.2.py =
    8 11 14 17 20 23 26 29 32 35 38 41 44 47 50 53 56 59 62 65 68 71 74 77 80 83 86 89
>>>
```

Result : Numbers are correctly printed from 8 to 89 with step 3.

Experiment - 1.3

AIM:

Split a string into array of characters in Python.

Description:

In Python, strings are sequences of characters. To manipulate or analyze individual characters, it's often useful to convert the string into an array (or list) of its characters. This process is called splitting a string into a list of characters. This can be done easily using built-in Python features like list() or list comprehensions.

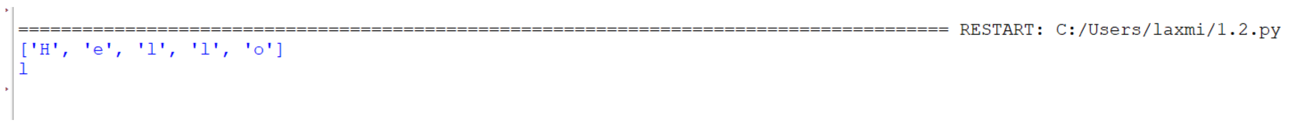
Procedure / Algorithm:

1. Start
2. Accept input from the user and store it in a variable, say input_string
3. Convert the string into a list of characters using list(input_string)
4. Display the resulting list
5. Stop

Source Code :

```
str = "Hello"
arr = list(str)
print(arr)
print(arr[3])
```

Input and Output:



```
===================================================================================== RESTART: C:/Users/laxmi/1.2.py
['H', 'e', 'l', 'l', 'o']
l
```

Result : The string is successfully split into character.

Experiment - 1.4

AIM:

write a Python program to get the largest number from a list.

Description:

In Python, lists can store multiple values including integers. To determine the largest number from a list, we can use built-in functions like max() or traverse the list manually to compare each element. This helps in understanding how to handle lists and use control flow structures effectively.

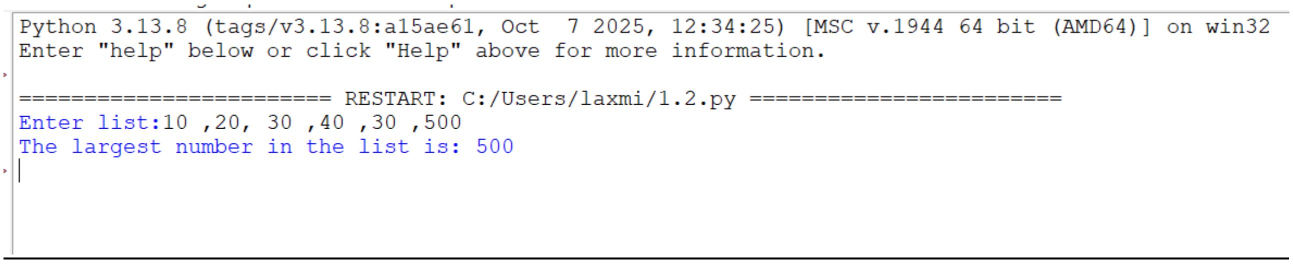
Procedure / Algorithm:

1. Input a list of numbers from the user.
2. Convert the input string into a list of integers.
3. Use the built-in max() function to find the largest number.
4. Alternatively, iterate through the list using a loop to manually find the maximum.
5. Display the largest number.

Source Code :

```
numbers = eval(input("Enter list:"))
largest = max(numbers)
print("The largest number in the list is:", largest)
```

Input and Output:



```
Python 3.13.8 (tags/v3.13.8:a15ae61, Oct  7 2025, 12:34:25) [MSC v.1944 64 bit (AMD64)] on win32
Enter "help" below or click "Help" above for more information.

======================== RESTART: C:/Users/laxmi/1.2.py ========================
Enter list:10 ,20, 30 ,40 ,30 ,500
The largest number in the list is: 500
```

Result : The program correctly identifies the largest number.

\

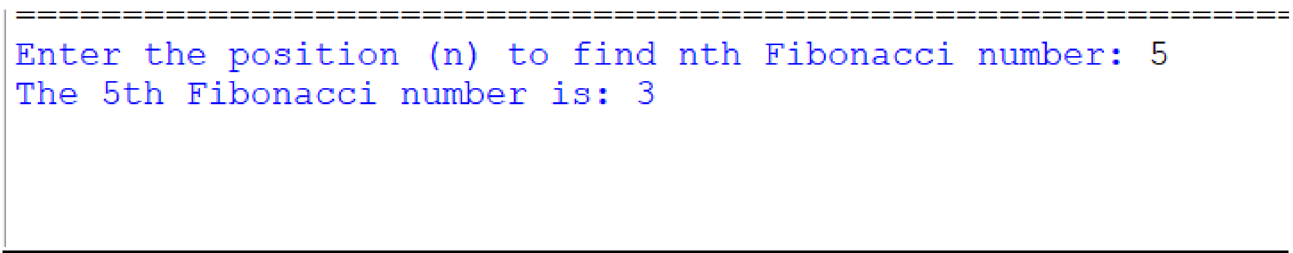
Experiment - 1.5

AIM:
write a Python program to get the largest number from a list.

Description:
The **Fibonacci sequence** is a series of numbers where each number is the sum of the two preceding ones, starting from 0 and 1.
That is:
$F(0) = 0, F(1) = 1$
$F(n) = F(n-1) + F(n-2)$ for $n \geq 2$
In this program, a **function** is defined to compute the **fibonacci** using **iteration**. Functions in Python allow us to organize code for reusability and clarity.

Procedure / Algorithm:
1. Define a function called fibonacci(n) that calculates the nth Fibonacci number.
2. Use a recursive or iterative approach inside the function.
3. Take input n from the user.
4. Call the function with n and display the result.

Source Code :
```
def fibonacci(n):
    if n <= 0:
        return "Invalid input"
    elif n == 1 or n == 2:
        return n-1
    else:
        a, b = 0, 1
        for _ in range(n - 2):
            a, b = b, a + b
        return b

n = int(input("Enter the position (n) to find nth Fibonacci number: "))
print(f"The {n}th Fibonacci number is:", fibonacci(n))
```
Input and Output:



```
==============================================================
Enter the position (n) to find nth Fibonacci number: 5
The 5th Fibonacci number is: 3
```

Result : The program correctly calculates the nth Fibonacci number

Experiment – 2.1

Aim:

Write a Python program that defines a Car class with attributes like make, model, and year, and methods like start() to start the car and stop() to stop it.

Description:

This program demonstrates the use of a class in Python by defining a Car class. The class contains data attributes to represent the make, model, and year of a car. It also includes methods to simulate starting and stopping the car, which print relevant messages.

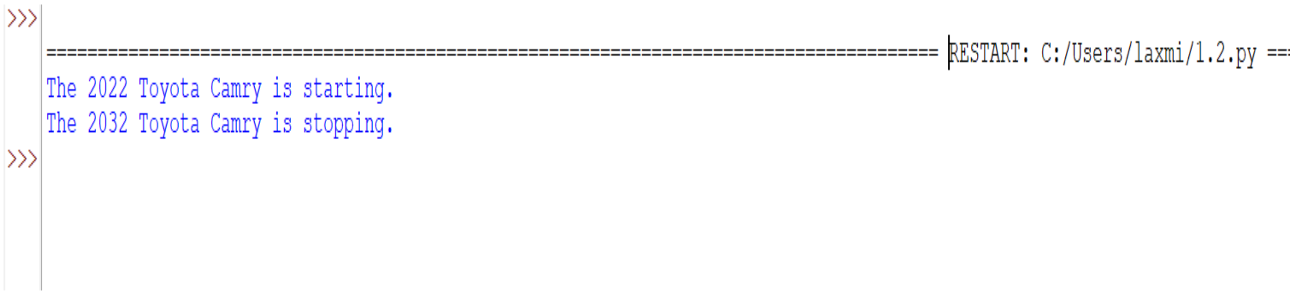
Procedure / Algorithm :

1. Start the program.
2. Define a class named Car with the attributes make, model, and year.
3. Define a method start() that prints "Car started."
4. Define a method stop() that prints "Car stopped."
5. Create an instance of the Car class and call its methods.
6. End the program.

Source Code:

```
class Car:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year
    def start(self):
        print(f"The {self.year} {self.make} {self.model} is starting.")
    def stop(self):
        print(f"The {self.year} {self.make} {self.model} is stopping.")
# Example usage
car1 = Car("Toyota", "Camry", 2022)
car2=Car("Toyota", "Camry", 2032)
car1.start()
car2.stop()
```

Input and Output:



```
>>>
    ============================================================================================ RESTART: C:/Users/laxmi/1.2.py ==
    The 2022 Toyota Camry is starting.
    The 2032 Toyota Camry is stopping.
>>>
```

Result : Class and method execution demonstrated correctly**.**

Experiment – 2.2

Aim:

Write a Python program that demonstrates inheritance by creating a base class Animal and derived classes like Dog, Cat, etc., each with their specific behaviors.

Description:

This program demonstrates the concept of inheritance in Python. A base class Animal is created with a generic behavior. Two child classes Dog and Cat inherit from Animal and add their own specific behaviors.

Procedure / Algorithm:

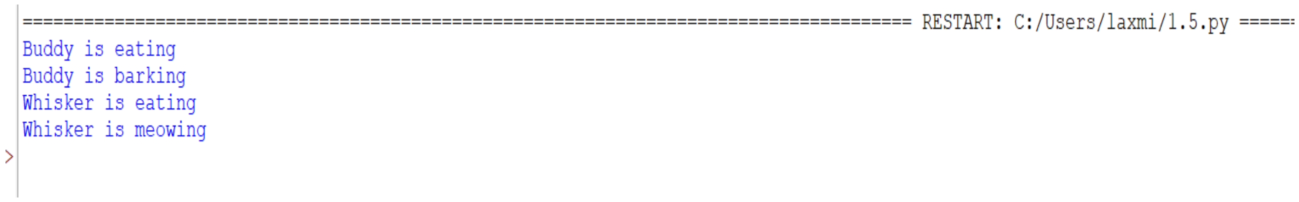
1. Start the program.
2. Define a base class Animal with a generic method.
3. Define subclasses Dog and Cat that inherit from Animal.
4. Add specific methods for each subclass.
5. Create objects of each class and call their respective methods.
6. End the program.

Source Code:

```
class Animal:
    def __init__(self, name):
        self.name = name
    def eat(self):
        print(f"{self.name} is eating")
class Dog(Animal):
    def bark(self):
        print(f"{self.name} is barking")
class Cat(Animal):
    def meow(self):
        print(f"{self.name} is meowing")
```

```
dog = Dog("Buddy")

cat = Cat("Whisker")

dog.eat()

dog.bark()

cat.eat()

cat.meow()
```

Input and Output:



```
======================================================================================== RESTART: C:/Users/laxmi/1.5.py =====
Buddy is eating
Buddy is barking
Whisker is eating
Whisker is meowing
>
```

Result : Inheritance and method overriding demonstrated.

Experiment – 2.3

Aim:

Define a base class called Animal with a method make_sound(). Implement derived classes like Dog, Cat, and Bird that override the make_sound() method to produce different sounds. Demonstrate polymorphism by calling the method on objects of different classes.

Description:

This program demonstrates polymorphism in object-oriented programming. A method make_sound() is defined in the base class and is overridden in each derived class. The overridden method is invoked using a single interface, showcasing polymorphic behavior.

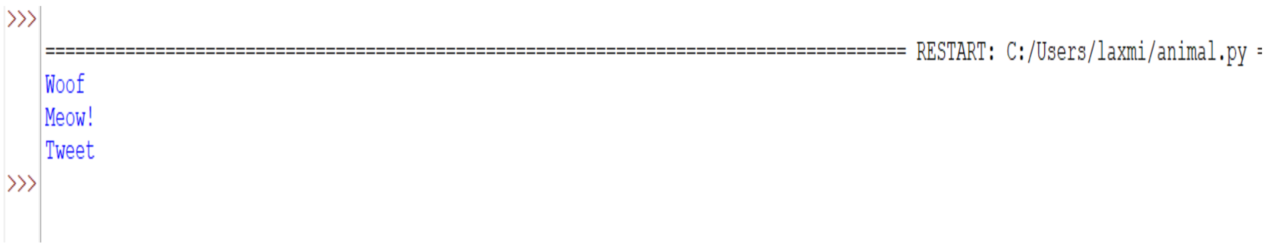
Procedure / Algorithm:

1. Start the program.
2. Define a base class Animal with a method make_sound().
3. Override this method in derived classes Dog, Cat, and Bird.
4. Create instances of each derived class.
5. Use a loop or function to call make_sound() on each object.
6. End the program.

Source Code:

```
class Animal:
    def make_sound(self):
        print("Some generic animal sound")
class Dog(Animal):
    def make_sound(self):
        print("Woof")
class Cat(Animal):
    def make_sound(self):
        print("Meow!")
class Bird(Animal):
    def make_sound(self):
        print("Tweet")
animals = [Dog(), Cat(), Bird()]
for animal in animals:
    animal.make_sound()
```

Input and Output:



```
>>> 
    ========================================================================================= RESTART: C:/Users/laxmi/animal.py =
    Woof
    Meow!
    Tweet
>>> 
```

Result : Polymorphism demonstrated correctly.

Experiment – 2.4

Aim:

Write a Python program that demonstrates error handling using the try-except block to handle division by zero.

Description:

This program asks the user to input two numbers and attempts to divide the first number by the second. If the second number is zero, the program handles the error gracefully using a try-except block and displays an appropriate message instead of crashing.

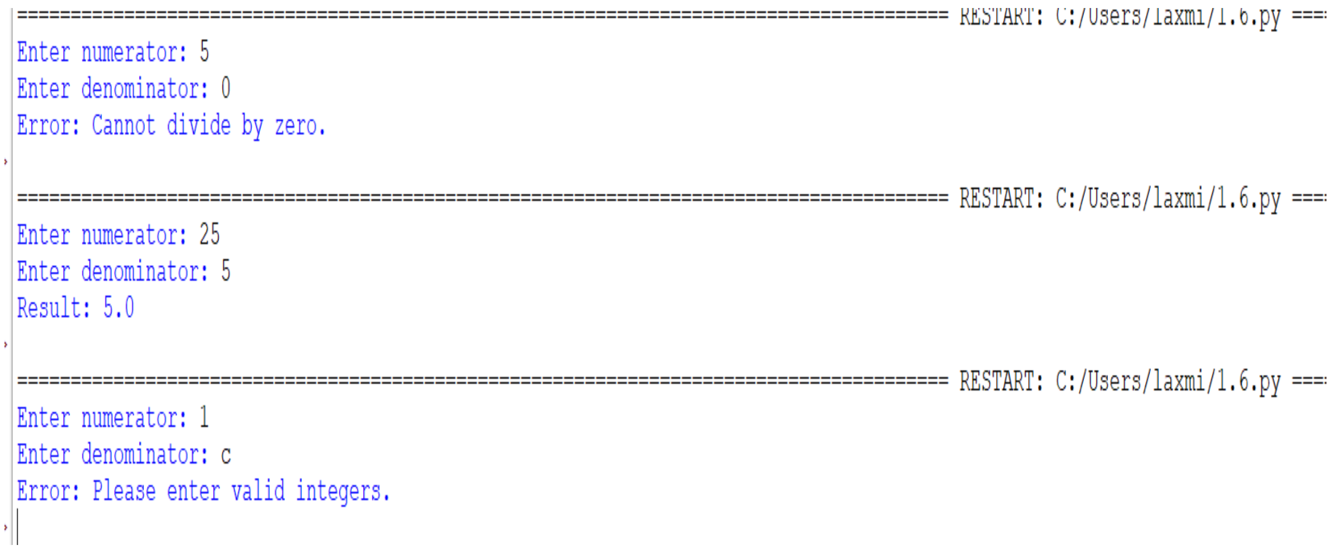
Procedure / Algorithm:

1. Start the program.
2. Prompt the user to enter two numbers.
3. Use a try block to perform division.
4. Use an except block to catch division by zero.
5. Display appropriate success or error messages.
6. End the program.

Source Code:

```
try:
    numerator = int(input("Enter numerator: "))
    denominator = int(input("Enter denominator: "))
    result = numerator / denominator
    print(f"Result: {result}")
except ZeroDivisionError:
    print("Error: Cannot divide by zero.")
except ValueError:
    print("Error: Please enter valid integers.")
```

Input and Output :



```
================================================================================ RESTART: C:/Users/laxmi/1.6.py ===
Enter numerator: 5
Enter denominator: 0
Error: Cannot divide by zero.

================================================================================ RESTART: C:/Users/laxmi/1.6.py ===
Enter numerator: 25
Enter denominator: 5
Result: 5.0

================================================================================ RESTART: C:/Users/laxmi/1.6.py ===
Enter numerator: 1
Enter denominator: c
Error: Please enter valid integers.
```

Result : Program handles division by zero successfully.

Experiment – 3.1

Aim:

Write a NumPy program using methods — info, add, array, all, greater, greater_equal, less, less_equal, equal, allclose, zeros, ones, linspace, tolist.

a. To get help on the add function

b. To test whether none of the elements of a given array is zero.

c. To create an element-wise comparison (greater, greater_equal, less and less_equal,

equal, equal within a tolerance) of two given arrays.

Description:

This Python program demonstrates several basic operations and functions of the NumPy library. It includes getting help and information on functions, testing array elements, performing element-wise comparisons between arrays, and creating arrays using built-in NumPy functions like zeros(), ones(), and linspace(). It also shows how to convert a NumPy array into a Python list using tolist().

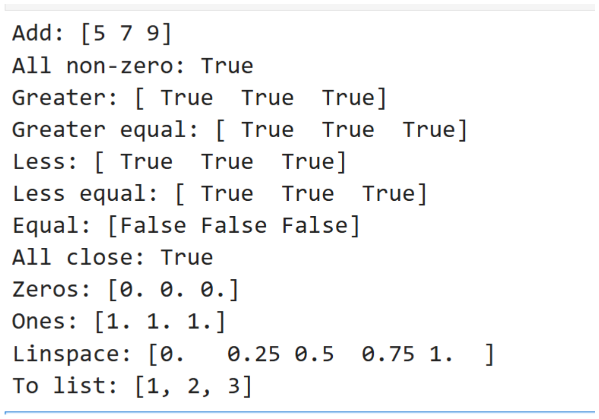
Procedure / Algorithm :

1. Import NumPy using import numpy as np to access its mathematical and array-handling functions.
2. Use help(np.add) and np.info(np.add) to get documentation about the add() function.
3. Create an array and use np.all() to test whether all elements are non-zero.
4. Define two arrays and perform element-wise comparisons using functions like np.greater(), np.greater_equal(), np.less(), np.less_equal(), np.equal(), and np.allclose().
5. Create arrays filled with zeros and ones using np.zeros() and np.ones().
6. Generate evenly spaced numbers using np.linspace().
7. Convert a NumPy array to a regular Python list using tolist() and display all results.

Source Code:

```
import numpy as np

a=np.array([1,2,3])

b=np.array([4,5,6])

print("Add:",np.add(a,b))

print("All non-zero:",np.all(a))

print("Greater:",np.greater(b,a))
print("Greater equal:",np.greater_equal(b,a))

print("Less:",np.less(a,b))

print("Less equal:",np.less_equal(a,b))

print("Equal:",np.equal(a,b))

print("All close:",np.allclose(a,[1,2,3]))

print("Zeros:",np.zeros(3))

print("Ones:",np.ones(3))

print("Linspace:",np.linspace(0,1,5))

print("To list:",a.tolist())
```

Input and Output:



```
Add: [5 7 9]
All non-zero: True
Greater: [ True  True  True]
Greater equal: [ True  True  True]
Less: [ True  True  True]
Less equal: [ True  True  True]
Equal: [False False False]
All close: True
Zeros: [0. 0. 0.]
Ones: [1. 1. 1.]
Linspace: [0.   0.25 0.5  0.75 1.  ]
To list: [1, 2, 3]
```

Result : Demonstrated all listed NumPy methods successfully.

Experiment – 3.2

Aim :

Write a NumPy program using NumPy methods — max, min, argmax, argmin, repr,

count, bincount, unique.

a. To extract all numbers from a given array which are less and greater than a specified number.

b. To find the indices of the maximum and minimum numbers along the given axis of an array.

Description:

This program demonstrates the use of various NumPy methods to perform essential operations on arrays. It shows how to determine the maximum and minimum values in an array, identify their respective indices, find unique elements, and count the frequency of each element.

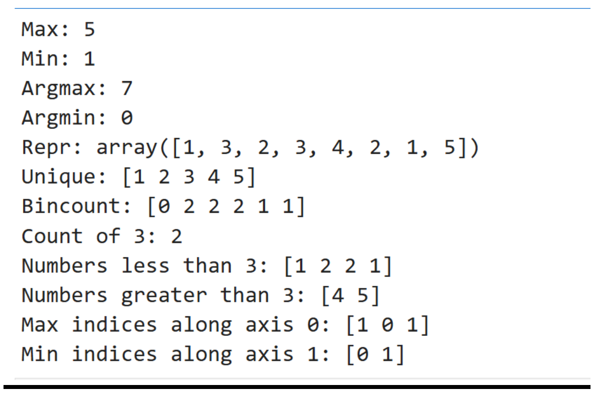
Procedure / Algorithm :

1. Start the program.
2. Import numpy as np.
3. Create a NumPy array.
4. Use np.max(), np.min() to find maximum and minimum.
5. Use np.argmax(), np.argmin() to find indices of max and min.
6. Use np.unique() to get unique elements.
7. Use np.bincount() to count occurrences of integers.
8. Use np.count_nonzero() to count how many times a specific value appears
9. Use regular filtering to extract values:
10. Less than a specified number using arr[arr < value]
11. Greater than a specified number using arr[arr > value]
12. Display results.
13. End the program

Source Code:

```
import numpy as np

arr = np.array([1, 3, 2, 3, 4, 2, 1, 5])

print("Max:", np.max(arr))

print("Min:", np.min(arr))
print("Argmax:", np.argmax(arr))
print("Argmin:", np.argmin(arr))
print("Repr:", repr(arr))
print("Unique:", np.unique(arr))
print("Bincount:", np.bincount(arr))
# Count example (count of a specific value, e.g., 3)
count_3 = np.count_nonzero(arr == 3)
print("Count of 3:", count_3)
specified_num = 3
less_than = arr[arr < specified_num]
greater_than = arr[arr > specified_num]
print(f"Numbers less than {specified_num}:", less_than)
print(f"Numbers greater than {specified_num}:", greater_than)
arr_2d = np.array([[1, 5, 2], [8, 3, 4]])
print("Max indices along axis 0:", np.argmax(arr_2d, axis=0))
print("Min indices along axis 1:", np.argmin(arr_2d, axis=1))
```

Input and Output:



```
Max: 5
Min: 1
Argmax: 7
Argmin: 0
Repr: array([1, 3, 2, 3, 4, 2, 1, 5])
Unique: [1 2 3 4 5]
Bincount: [0 2 2 2 1 1]
Count of 3: 2
Numbers less than 3: [1 2 2 1]
Numbers greater than 3: [4 5]
Max indices along axis 0: [1 0 1]
Min indices along axis 1: [0 1]
```

Result: Program demonstrates array statistics, counting, and unique elements using NumPy.

Experiment 4.1

4.1) (i)

Aim:

Write a Pandas program to create and display a one-dimensional array-like object containing an array of data using Pandas module.

Description :

Pandas provides a powerful data structure called **Series**, which is a one-dimensional labeled array capable of holding any data type (integers, strings, floats, etc.). This program demonstrates how to create a Series from a simple Python list and display its contents.

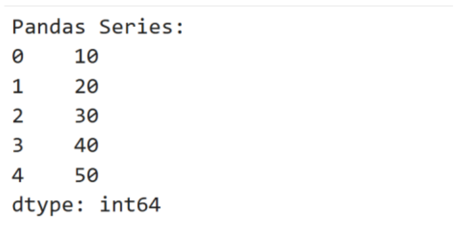
Procedure / Algorithm :

1. Start the program.
2. Import Pandas as pd.
3. Define a list of data.
4. Use pd.Series() to create a Series.
5. Print the Series.
6. End the program.

Source Code :

```
import pandas as pd

# Creating a Pandas Series

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

series = pd.Series(data)

print("Pandas Series:")

print(series)
```

Input and Output:



```
Pandas Series:
0    10
1    20
2    30
3    40
4    50
dtype: int64
```

Result: A one-dimensional Pandas Series is created and displayed correctly

4.1) (ii)

Aim:

Write a Pandas program to convert a Panda module Series to Python list and it's type.

Description :

Pandas Series is a one-dimensional labeled array. While it behaves similarly to a list, it is a Pandas object. Sometimes, we may need to convert it into a native Python list for compatibility with other Python functions or libraries. This program demonstrates how to perform that conversion and verify the result using the type() function.

Procedure / Algorithm :

1. Start the program.
2. Create a Pandas Series.
3. Convert the Series to a list using .tolist().
4. Print the list and its type.
5. End the program.

Source Code :

```
# Convert Series to Python list

list_data = series.tolist()

print("Series as Python List:", list_data)

print("Type of list_data:", type(list_data))
```

Input and Output:

```
Series as Python List: [10, 20, 30, 40, 50]
Type of list_data: <class 'list'>
```

Result : Series is successfully converted into a Python list.

Experiment- 4.2

4.2) (i)

Aim:

Write a Pandas program to create and display a DataFrame from a specified dictionary data which has the index labels.

Description :

A **DataFrame** is a two-dimensional labeled data structure in Pandas, similar to a table in a database or an Excel spreadsheet. It can be created from various data sources including dictionaries. This program demonstrates how to construct a DataFrame from a dictionary where each key represents a column and each value is a list of column entries. Custom index labels are also assigned to the rows.

Procedure / Algorithm :

1. Start the program.
2. Import Pandas as pd and NumPy as np.
3. Define dictionary exam_data and list labels.
4. Use pd.DataFrame(data, index=labels) to create DataFrame.
5. Print the DataFrame.
6. End the program.

Source Code :

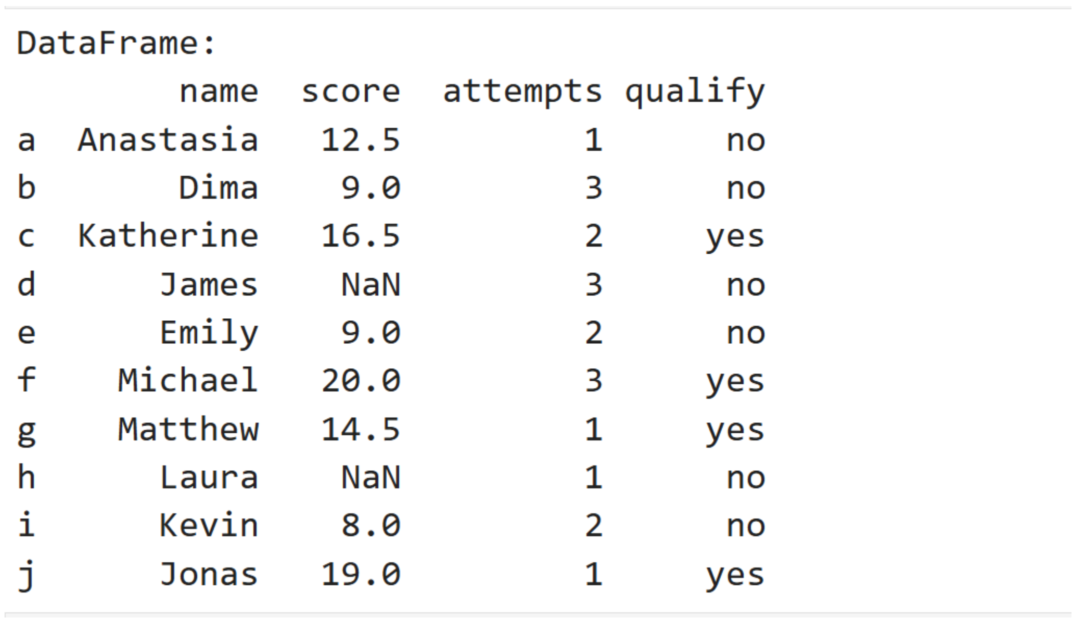
```
import pandas as pd
import numpy as np
exam_data = {
   'name': ['Anastasia', 'Dima', 'Katherine', 'James', 'Emily',
            'Michael', 'Matthew', 'Laura', 'Kevin', 'Jonas'],
   'score': [12.5, 9, 16.5, np.nan, 9, 20, 14.5, np.nan, 8, 19],
   'attempts': [1, 3, 2, 3, 2, 3, 1, 1, 2, 1],
   'qualify': ['yes', 'no', 'yes', 'no', 'no',
              'yes', 'yes', 'no', 'no', 'yes']}
labels = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']
# Create DataFrame with index labels
```

df = pd.DataFrame(exam_data, index=labels)

print("DataFrame:")

print(df)

Input and Output:



```
DataFrame:
        name  score  attempts qualify
a  Anastasia   12.5         1      no
b       Dima    9.0         3      no
c  Katherine   16.5         2     yes
d      James    NaN         3      no
e      Emily    9.0         2      no
f    Michael   20.0         3     yes
g    Matthew   14.5         1     yes
h      Laura    NaN         1      no
i      Kevin    8.0         2      no
j      Jonas   19.0         1     yes
```

Result : DataFrame is created with specified dicƟonary data and index labels.

4.2) (ii)

Aim:

Write a Pandas program to change the name 'James' to 'Suresh' in name column of the DataFrame.

Description :

Pandas allows for efficient data manipulation using its powerful DataFrame structure. This program demonstrates how to **replace specific values** in a column using the .replace() method. We target the 'name' column and change all occurrences of 'James' to 'Suresh'.

Procedure / Algorithm :

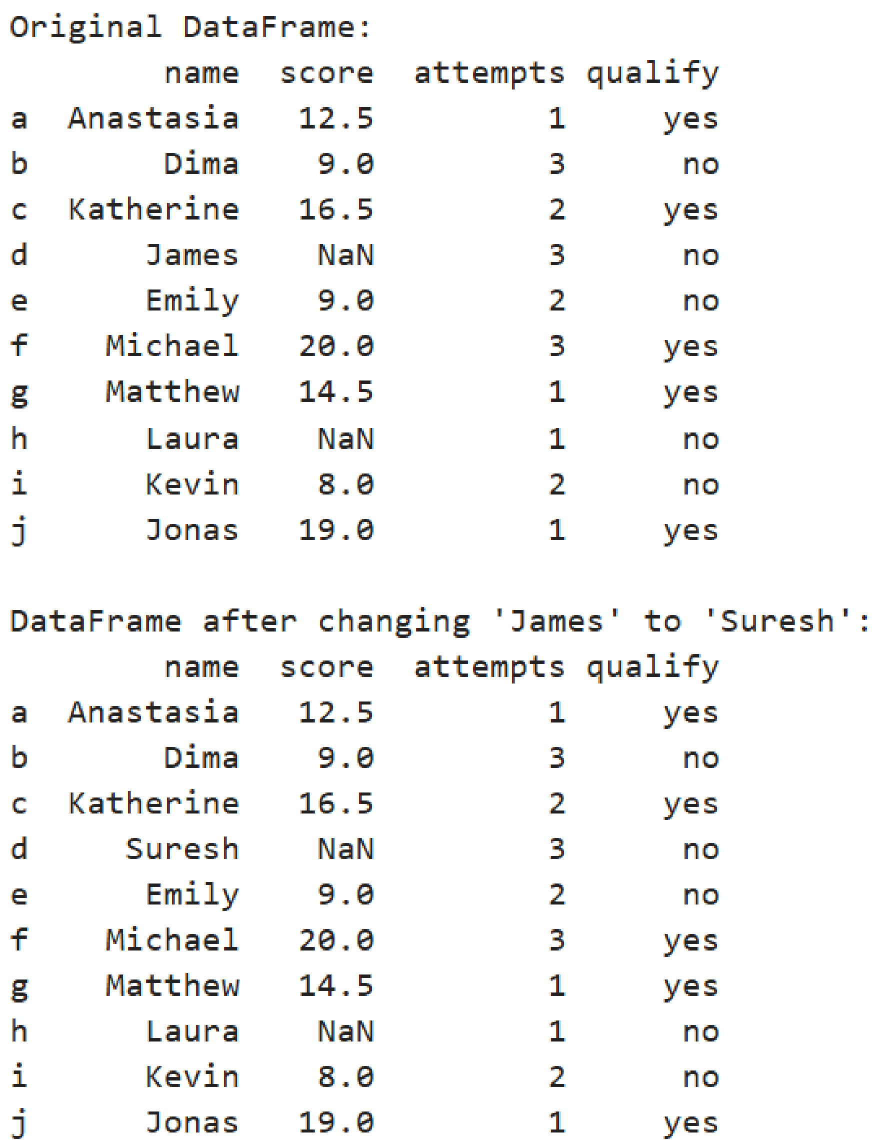
1. Start program.
2. Create DataFrame as before.
3. Use df['name'].replace('James','Suresh', inplace=True) to update value.
4. Print updated DataFrame.
5. End program.

Source Code:

```
import pandas as pd
import numpy as np
# Sample dictionary data and labels
exam_data = {
  'name': ['Anastasia', 'Dima', 'Katherine', 'James', 'Emily', 'Michael',
           'Matthew', 'Laura', 'Kevin', 'Jonas'],
  'score': [12.5, 9, 16.5, np.nan, 9, 20, 14.5, np.nan, 8, 19],
  'attempts': [1, 3, 2, 3, 2, 3, 1, 1, 2, 1],
  'qualify': ['yes', 'no', 'yes', 'no', 'no', 'yes', 'yes', 'no', 'no', 'yes']
}
labels = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']
# Create DataFrame
df = pd.DataFrame(exam_data, index=labels)
# Display original DataFrame
```

```
print("Original DataFrame:")
print(df)
# Change 'James' to 'Suresh' in the 'name' column
df.loc[df['name'] == 'James', 'name'] = 'Suresh'
# Display updated DataFrame
print("\nDataFrame after changing 'James' to 'Suresh':")
print(df)
```

Input and Output:



```
Original DataFrame:
        name  score  attempts qualify
a  Anastasia   12.5         1     yes
b       Dima    9.0         3      no
c  Katherine   16.5         2     yes
d      James    NaN         3      no
e      Emily    9.0         2      no
f    Michael   20.0         3     yes
g    Matthew   14.5         1     yes
h      Laura    NaN         1      no
i      Kevin    8.0         2      no
j      Jonas   19.0         1     yes

DataFrame after changing 'James' to 'Suresh':
        name  score  attempts qualify
a  Anastasia   12.5         1     yes
b       Dima    9.0         3      no
c  Katherine   16.5         2     yes
d     Suresh    NaN         3      no
e      Emily    9.0         2      no
f    Michael   20.0         3     yes
g    Matthew   14.5         1     yes
h      Laura    NaN         1      no
i      Kevin    8.0         2      no
j      Jonas   19.0         1     yes
```

Result : Name 'James' is successfully updated to 'Suresh'.

4.2(iii)

Aim:

Write a Pandas program to insert a new column in existing DataFrame.

Description :

Pandas allows dynamic modification of DataFrames, including adding new columns. This program demonstrates how to insert a new column (e.g., 'salary') into an existing DataFrame using direct assignment. The new column can be derived from a list, Series, or computed values.

Procedure / Algorithm :

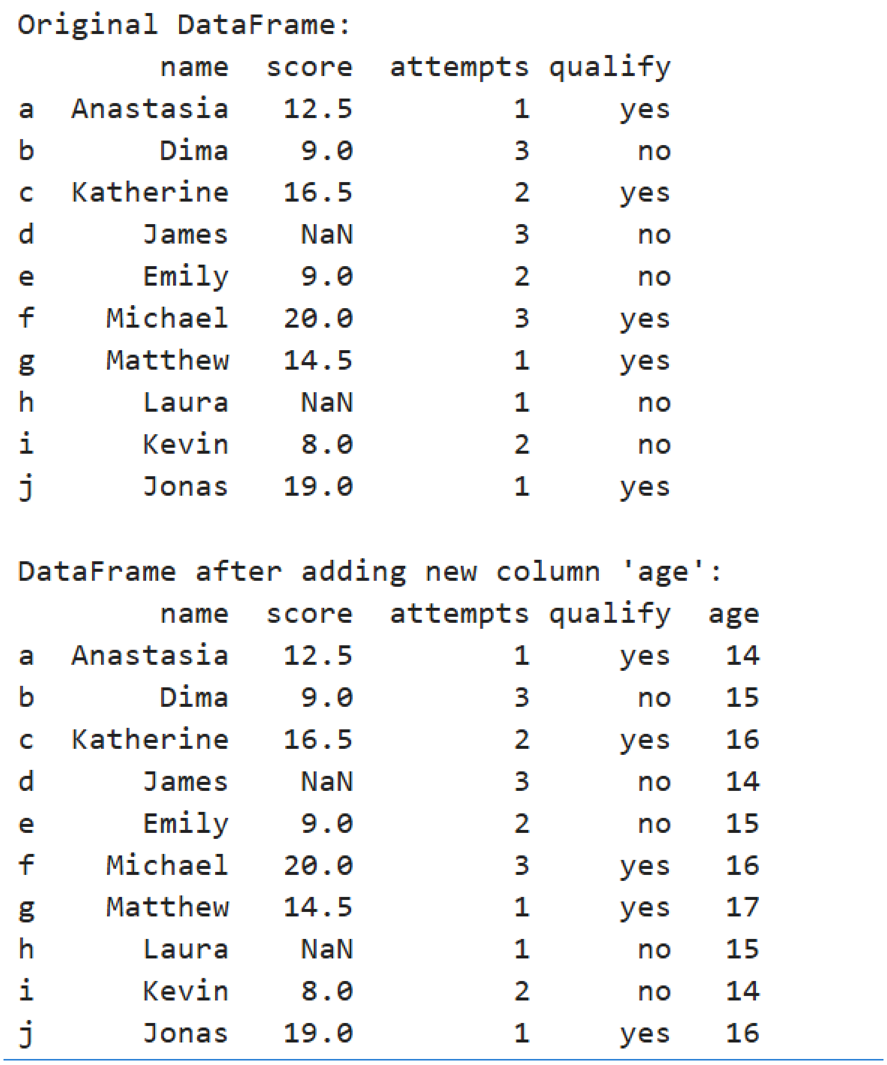
1. Start program.
2. Create DataFrame.
3. Insert new column grade using df['grade'] = [...].
4. Print updated DataFrame.
5. End program.

Source Code :

```
import pandas as pd

import numpy as np

# Sample dictionary data and labels

exam_data = {

   'name': ['Anastasia', 'Dima', 'Katherine', 'James', 'Emily', 'Michael',

            'Matthew', 'Laura', 'Kevin', 'Jonas'],

   'score': [12.5, 9, 16.5, np.nan, 9, 20, 14.5, np.nan, 8, 19],

   'attempts': [1, 3, 2, 3, 2, 3, 1, 1, 2, 1],

   'qualify': ['yes', 'no', 'yes', 'no', 'no', 'yes', 'yes', 'no', 'no', 'yes']

}

labels = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']

# Create the DataFrame

df = pd.DataFrame(exam_data, index=labels)

# Display original DataFrame
```

```
print("Original DataFrame:")
print(df)
# Insert a new column 'age' into the DataFrame
df['age'] = [14, 15, 16, 14, 15, 16, 17, 15, 14, 16]
# Display updated DataFrame
print("\nDataFrame after adding new column 'age':")
print(df)
```

Input and Output:



```
Original DataFrame:
        name  score  attempts qualify
a  Anastasia   12.5         1     yes
b       Dima    9.0         3      no
c  Katherine   16.5         2     yes
d      James    NaN         3      no
e      Emily    9.0         2      no
f    Michael   20.0         3     yes
g    Matthew   14.5         1     yes
h      Laura    NaN         1      no
i      Kevin    8.0         2      no
j      Jonas   19.0         1     yes

DataFrame after adding new column 'age':
        name  score  attempts qualify  age
a  Anastasia   12.5         1     yes   14
b       Dima    9.0         3      no   15
c  Katherine   16.5         2     yes   16
d      James    NaN         3      no   14
e      Emily    9.0         2      no   15
f    Michael   20.0         3     yes   16
g    Matthew   14.5         1     yes   17
h      Laura    NaN         1      no   15
i      Kevin    8.0         2      no   14
j      Jonas   19.0         1     yes   16
```

Result : New column is added successfully.

4.2(iv)

Aim:

Write a Pandas program to get list from DataFrame column headers.

Description :

In Pandas, each DataFrame contains a set of column labels. These labels can be accessed using the .columns attribute. To convert them into a standard Python list, we use the .tolist() method. This is useful for dynamic column operations, documentation, or validation tasks. **Procedure /**

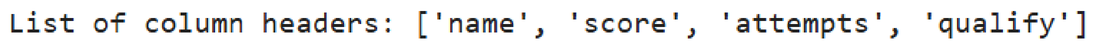
Procedure / Algorithm :

1. Start program.
2. Create DataFrame.
3. Use list(df.columns) to get column headers.
4. Print list.
5. End program.

Source Code :

```
import pandas as pd

import numpy as np

exam_data = {

   'name': ['Anastasia', 'Dima', 'Katherine', 'James', 'Emily', 'Michael',

            'Matthew', 'Laura', 'Kevin', 'Jonas'],

   'score': [12.5, 9, 16.5, np.nan, 9, 20, 14.5, np.nan, 8, 19],

   'attempts': [1, 3, 2, 3, 2, 3, 1, 1, 2, 1],

   'qualify': ['yes', 'no', 'yes', 'no', 'no', 'yes', 'yes', 'no', 'no', 'yes']

}

labels = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']

df = pd.DataFrame(exam_data, index=labels)

column_list = list(df.columns)

print("List of column headers:", column_list)
```

Input and Output:



```
List of column headers: ['name', 'score', 'attempts', 'qualify']
```

Result : Column headers successfully converted to a list.