Python Programs

Experiment - 1.1

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

Description:
This program prompts the user to input a weight value in kilograms. After receiving the input, it multiplies the value by 2.2 to convert it into pounds, since 1 kilogram is equal to 2.2 pounds. Finally, the program displays the calculated weight in pounds to the user.

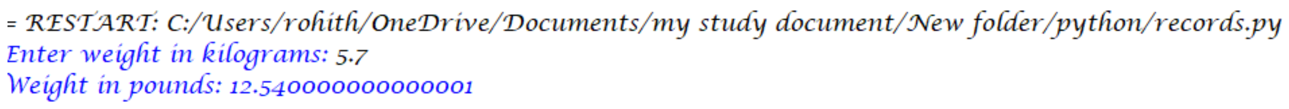
Procedure / Algorithm

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.

Program:

```
# Ask the user to enter weight in kilograms
kg = float(input("Enter weight in kilograms: "))

# Convert kilograms to pounds
pounds = kg * 2.2

# Display the result
print("Weight in pounds:", pounds)
```

Output:



```
= RESTART: C:/Users/rohith/OneDrive/Documents/my study document/New folder/python/records.py
Enter weight in kilograms: 5.7
Weight in pounds: 12.540000000000001
```

Result:

Hence the program of converting the kilograms to pounds has been executed and it’s output is verified successfully.

Experiment - 1.2

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

Description:
The program is intended to print a sequence of numbers that follow an arithmetic pattern. It begins with a specific starting value and increases by a fixed amount in each step. The sequence continues until it reaches or slightly before a defined upper limit, ensuring the final number does not exceed the specified maximum.
A for loop is used to implement this functionality. The loop initializes with the starting value, increments the value by a constant difference in each iteration, and runs until the value exceeds the ending condition. In each iteration, the current value is printed.
This approach efficiently generates a numeric sequence with consistent intervals using a structured looping mechanism. The sequence generated will be: 8, 11, 14, ..., 86, 89.

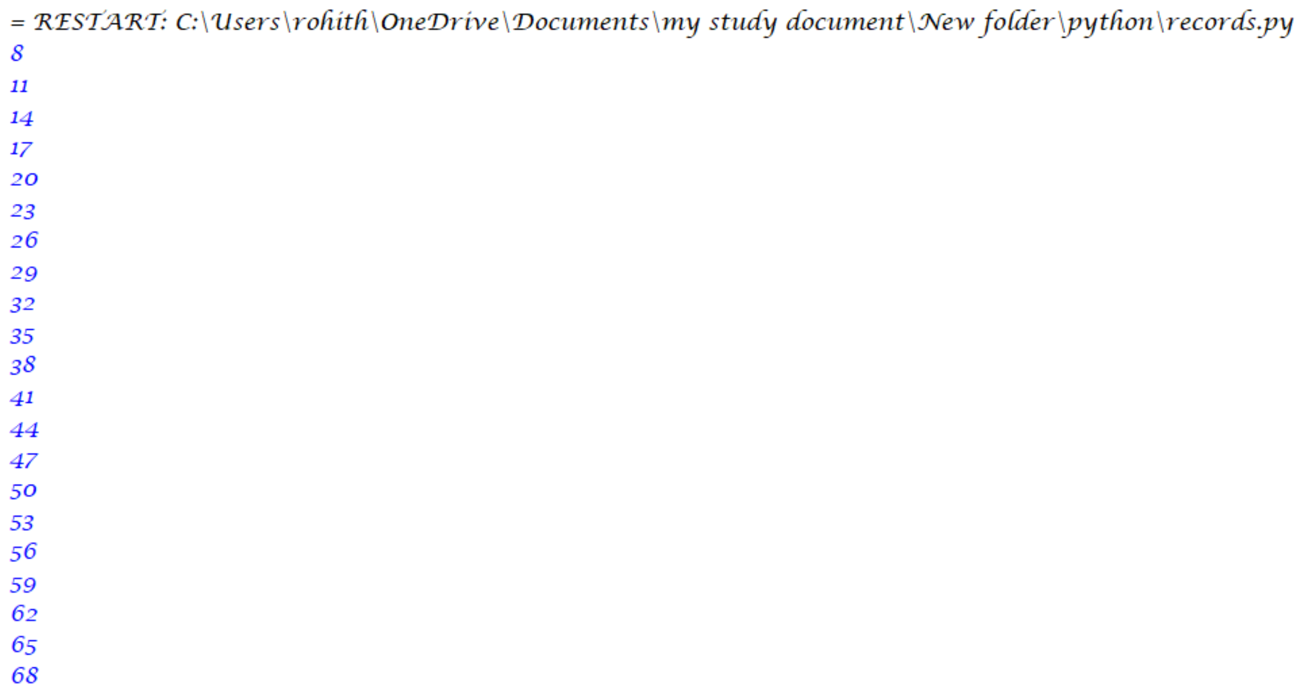
Procedure / Algorithm

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.

Program:

```
# Loop from 8 to 89, incrementing by 3 each time
for number in range(8, 90, 3):
   print(number)
```

Output:



```
= RESTART: C:\Users\rohith\OneDrive\Documents\my study document\New folder\python\records.py
8
11
14
17
20
23
26
29
32
35
38
41
44
47
50
53
56
59
62
65
68
```

Result:
Hence the program of using for loop to print pattern of numbers has been executed and it’s output is verified successfully.

Experiment - 1.3

AIM: Write a python program to 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.

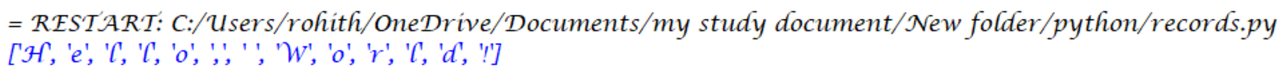
Procedure / Algorithm

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

Program:

```
# Input string
text = "Hello, World!"

# Split string into list of characters
char_list = list(text)

# Print the result
print(char_list)
```

Output:



```
= RESTART: C:/Users/rohith/OneDrive/Documents/my study document/New folder/python/records.py
['H', 'e', 'l', 'l', 'o', ',', ' ', 'W', 'o', 'r', 'l', 'd', '!']
```

Result:

Hence, the program of splitting a string into array of lists has been executed and it's output is verified successfully.

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. The program is designed to determine and display the **largest number in a list** of numeric values. It begins by defining a list that contains a collection of numbers. To find the largest value, the program uses a built-in function that compares all elements in the list and returns the one with the highest value. Finally, the result is displayed as output. This approach allows for quickly identifying the maximum value from a set of numbers using an efficient and straightforward method.

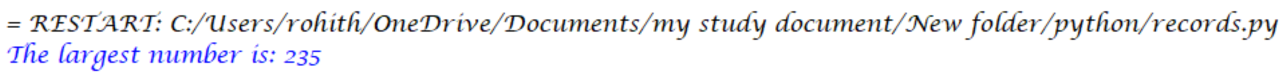
Procedure / Algorithm

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.

Program:

```
# Define a list of numbers
numbers = [13, 49, 235, 64, 90, 34]
# Find the largest number using max()
largest = max(numbers)

# Print the result
print("The largest number is:", largest)
```

Output:



```
= RESTART: C:/Users/rohith/OneDrive/Documents/my study document/New folder/python/records.py
The largest number is: 235
```

Result:

Hence the program of <u>getting the largest number from a list</u> has been executed and it's output has verified successfully.

Experiment - 1.5

AIM: Write a Python program to calculate the nth Fibonacci number using a function.

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.

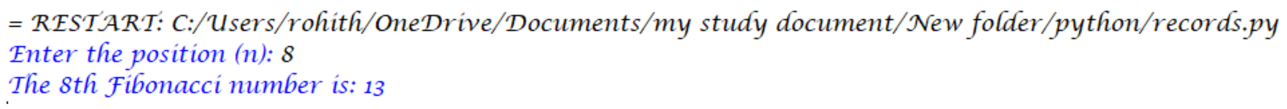
Procedure / Algorithm

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.

Program:

```
# Define a function to calculate the nth Fibonacci number
def fibonacci(n):
    if n <= 0:
        return "Invalid input"
    elif n == 1:
        return 0
    elif n == 2:
        return 1
    else:
        a, b = 0, 1
        for _ in range(2, n):
            a, b = b, a + b
        return b

# Input from the user
n = int(input("Enter the position (n): "))

# Calculate and display the nth Fibonacci number
print(f"The {n}th Fibonacci number is:", fibonacci(n))
```

Output:



```
= RESTART: C:/Users/rohith/OneDrive/Documents/my study document/New folder/python/records.py
Enter the position (n): 8
The 8th Fibonacci number is: 13
```

Result:

Hence, the program of calculating the Fibonacci number using a function has been executed and it's output has been verified successfully.

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.

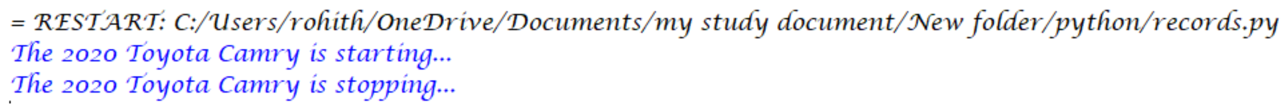
Procedure / Algorithm:

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.

Program:

```
# Define the Car class
class Car:
    def __init__(self, make, model, year):
        self.make = make      # Brand of the car
        self.model = model    # Model name
        self.year = year      # Manufacturing 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...")

# Create an object of the Car class
my_car = Car("Toyota", "Camry", 2020)

# Call methods
my_car.start()
my_car.stop()
```

<u>Output:</u>



```
= RESTART: C:/Users/rohith/OneDrive/Documents/my study document/New folder/python/records.py
The 2020 Toyota Camry is starting...
The 2020 Toyota Camry is stopping...
```

<u>Result:</u>

Hence, the program of <u>defining a car class and giving attributes and methods like starting and stopping</u> has been executed and it’s output has been verified successfully.

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:

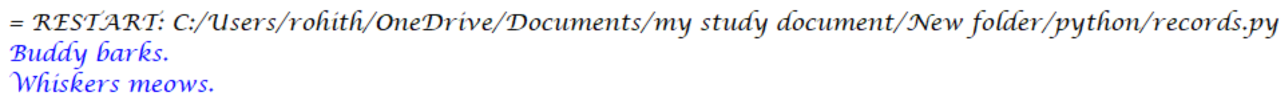
1. Start the program.
2. Define a base class Animal with a generic method.
3. Define subclasses Dog and Cat that inherit from Animal.
4. Add specific methods for each subclass.
5. Create objects of each class and call their respective methods.
6. End the program.

Program:

```
# Base class
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        print(f"{self.name} makes a sound.")

# Derived class: Dog
class Dog(Animal):
    def speak(self):
        print(f"{self.name} barks.")

# Derived class: Cat
class Cat(Animal):
    def speak(self):
        print(f"{self.name} meows.")
```

```
# Create objects of Dog and Cat
dog = Dog("Buddy")
cat = Cat("Whiskers")

# Call the speak method for each
dog.speak()
cat.speak()
```

<u>Output:</u>



```
= RESTART: C:/Users/rohith/OneDrive/Documents/my study document/New folder/python/records.py
Buddy barks.
Whiskers meows.
```

<u>Result:</u>

Hence, the program of <u>demonstrating the inheritance by creating a base class animal and derived classes with the attributes</u> has been executed and it’s output has been verified successfully.

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:
The program demonstrates the concepts of inheritance, method overriding, and polymorphism using object-oriented programming in Python.

A base class named Animal is defined with a method called make_sound(), which represents a generic behavior shared by all animals. This method is then overridden in multiple derived classes such as Dog, Cat, and Bird, where each subclass provides its own specific implementation of the make_sound() method to reflect the sound that particular animal makes.

To demonstrate polymorphism, objects of the derived classes are created and stored in a collection. A common function is then used to call the make_sound() method on each object. Although the method name is the same, the actual behavior depends on the class of the object being used. This shows how the same method can behave differently depending on the object it is called on, which is the core concept of polymorphism.

This structure provides a clear and organized way to manage related classes and behaviors in an extensible and maintainable manner.

Procedure / Algorithm:

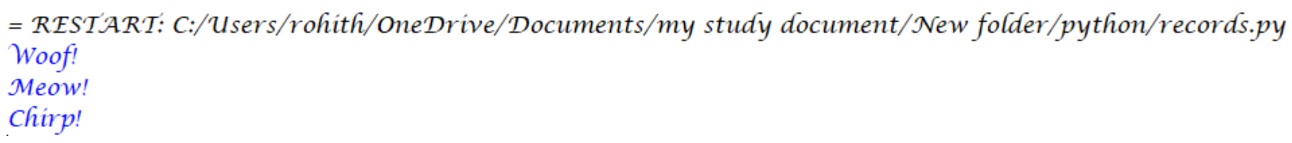
1. Start the program.
2. Define a base class Animal with a method make_sound().
3. Override this method in derived classes Dog, Cat, and Bird.
4. Create instances of each derived class.
5. Use a loop or function to call make_sound() on each object.
6. End the program.

Program:

```
# Base class
class Animal:
    def make_sound(self):
        print("Some generic animal sound.")
```

```
# Derived class: Dog
class Dog(Animal):
    def make_sound(self):
        print("Woof!")

# Derived class: Cat
class Cat(Animal):
    def make_sound(self):
        print("Meow!")

# Derived class: Bird
class Bird(Animal):
    def make_sound(self):
        print("Chirp!")

# List of animals
animals = [Dog(), Cat(), Bird()]

# Demonstrate polymorphism
for animal in animals:
    animal.make_sound()
```

<u>Output:</u>



```
= RESTART: C:/Users/rohith/OneDrive/Documents/my study document/New folder/python/records.py
Woof!
Meow!
Chirp!
```

<u>Result:</u>

Hence, the program of <u>demonstrating polymorphism by calling the methods on objects of different classes</u> has been executed and it's output has been verified successfully.

Experiment – 2.4

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

Description:

The program is designed to demonstrate error handling in Python using the try-except block. It focuses specifically on managing the case of division by zero, which is a common runtime error.

The program prompts the user to enter two numbers: a numerator and a denominator. These inputs are then used to perform a division operation. Since dividing a number by zero is mathematically undefined and causes a runtime error in Python, the operation is placed inside a try block.

If the user enters zero as the denominator, a ZeroDivisionError is raised. This error is caught by the corresponding except block, which displays a user-friendly error message instead of allowing the program to crash.

The program also includes a separate except block for ValueError to handle cases where the user enters invalid input (such as non-numeric values).

A finally block is included to execute code that should run regardless of whether an exception occurred or not. This is typically used for cleanup or confirmation messages, ensuring graceful program termination.

This structure improves the reliability and user experience of the program by preventing unexpected crashes and guiding the user through input errors.

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.

Program:

```
# Input two numbers from the user
try:
    numerator = int(input("Enter the numerator: "))
    denominator = int(input("Enter the denominator: "))

    # Attempt division
    result = numerator / denominator
    print("Result:", result)
```

```
except ZeroDivisionError:
    print("Error: Division by zero is not allowed.")

except ValueError:
    print("Error: Please enter valid integers.")

finally:
   print("Program execution completed.")
```

Output 1:

```
= RESTART: C:/Users/rohith/OneDrive/Documents/my study document/New folder/python/records.py
Enter the numerator: 25
Enter the denominator: 4
Result: 6.25
Program execution completed.
```

Output 2:

```
= RESTART: C:/Users/rohith/OneDrive/Documents/my study document/New folder/python/records.py
Enter the numerator: 1
Enter the denominator: 0
Error: Division by zero is not allowed.
Program execution completed.
```

Output 3:

```
= RESTART: C:/Users/rohith/OneDrive/Documents/my study document/New folder/python/records.py
Enter the numerator: 6
Enter the denominator: r
Error: Please enter valid integers.
Program execution completed.
```

Result:

Hence, the program of demonstrating error handling using the try-except block to handle division by zero has been executed and it's output has been verified successfully.