CODE 1
To implement the Gradient Descent algorithm to find the local minima of the function y=(x+3)2y = (x + 3)^2y=(x+3)2, we will follow these steps:

Steps for Gradient Descent:

1. **Define the function** y=(x+3)2y = (x + 3)^2y=(x+3)2 and its derivative.
2. **Initialize the starting point** for the search (here x=2x = 2x=2).
3. **Define the learning rate** (step size) for the gradient descent algorithm.
4. **Iterate** through the update rule: x=x−η⋅ddxf(x)x = x - \eta \cdot \frac{d}{dx} f(x)x=x−η⋅dxdf(x) where η\etaη is the learning rate, and ddxf(x)\frac{d}{dx} f(x)dxdf(x) is the derivative of the function with respect to xxx.
5. **Stop the iteration** once the change in xxx is very small (indicating that the algorithm has converged).

1. Define the function and its derivative:

The given function is:

y=(x+3)2y = (x + 3)^2y=(x+3)2

The derivative (gradient) of yyy with respect to xxx is:

ddx((x+3)2)=2(x+3)\frac{d}{dx} \left((x + 3)^2 \right) = 2(x + 3)dxd((x+3)2)=2(x+3)

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

# Define the function and its derivative
def func(x):
    return (x + 3)**2

def gradient(x):
    return 2 * (x + 3)

# Gradient Descent Algorithm
def gradient_descent(starting_point, learning_rate, iterations):
    x = starting_point
    x_history = [x]  # To store x values for plotting

    for _ in range(iterations):
        grad = gradient(x)  # Compute the gradient at current point
        x = x - learning_rate * grad  # Update x using the gradient descent formula
        x_history.append(x)
```

```
    return x, x_history  # Return the final x and the history of x values

# Parameters for Gradient Descent
starting_point = 2  # Starting point x = 2
learning_rate = 0.1  # Learning rate (step size)
iterations = 20  # Number of iterations

# Run the gradient descent algorithm
final_x, x_history = gradient_descent(starting_point, learning_rate, iterations)

# Output the final result
print(f"Final x value after {iterations} iterations: {final_x}")
print(f"Function value at final x: {func(final_x)}")

# Plot the function and the steps taken by gradient descent
x_vals = np.linspace(-10, 4, 400)  # Values for x to plot the function
y_vals = func(x_vals)

plt.figure(figsize=(8, 6))
plt.plot(x_vals, y_vals, label=r'$y = (x + 3)^2$', color='blue')
plt.scatter(x_history, [func(x) for x in x_history], color='red', marker='x', label='Steps Taken')
plt.title("Gradient Descent to Find Local Minima")
plt.xlabel("x")
plt.ylabel("y")
plt.legend()
plt.grid(True)
plt.show()
```

Explanation of the Code:

- **`func(x)`**: This is the function y=(x+3)2y = (x + 3)^2y=(x+3)2.
- **`gradient(x)`**: This function computes the gradient (derivative) of yyy with respect to xxx, which is 2(x+3)2(x + 3)2(x+3).
- **`gradient_descent()`**: This is the main function implementing the Gradient Descent algorithm. It starts at a given point (`starting_point`), updates the value of xxx using the formula x=x−η⋅ddxf(x)x = x - \eta \cdot \frac{d}{dx} f(x)x=x−η⋅dxdf(x), and iterates for the specified number of iterations.
- **`x_history`**: This list stores the xxx values at each step so we can visualize how the algorithm converges.
- **Plot**: After running the gradient descent, we plot the function and the steps taken by the algorithm using red "x" markers.

Output:

- The final xxx value after 20 iterations, which should be close to the local minimum of the function.
- A plot showing the function curve and the points where the gradient descent updates xxx.

Running the Code:

1. The initial point is x=2x = 2x=2.
2. The gradient descent updates will gradually move towards the local minima at x=−3x = -3x=−3 (since this is the point where the function y=(x+3)2y = (x + 3)^2y=(x+3)2 reaches its minimum).
3. The plot will show how the algorithm converges towards x=−3x = -3x=−3.

CODE 2

```
# Class to represent an item with value and weight
class Item:
def __init__(self, value, weight):
self.value = value
self.weight = weight
# Function to calculate the maximum value that can be carried
def fractional_knapsack(items, capacity):
# Sort items by value-to-weight ratio in descending order
items.sort(key=lambda item: item.value / item.weight, reverse=True)
total_value = 0.0 # To store the total value
for item in items:
if capacity >= item.weight:
# If the item can fit in the remaining capacity, take it all
capacity -= item.weight
total_value += item.value
else:
# Otherwise, take the fraction of the item that fits
fraction = capacity / item.weight
total_value += item.value * fraction
break # The knapsack is full
return total_value
# Driver code
if __name__ == "__main__":
# Taking the number of items as input
n = int(input("Enter the number of items: "))
# Taking item values and weights as input from the user
items = []
```

```
for i in range(n):
value = float(input(f"Enter the value of item {i + 1}: "))
weight = float(input(f"Enter the weight of item {i + 1}: "))
items.append(Item(value, weight))
# Taking the capacity of the knapsack as input
capacity = float(input("Enter the capacity of the knapsack: "))
# Calculate and print the maximum value
max_value = fractional_knapsack(items, capacity)
print(f"Maximum value we can obtain = {max_value:.2f}")
```