TASK-1

```
class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author

    def summary(self):
        return f"{self.title} by {self.author}"
```

TASK-2

```
# Sample data: a list of dictionaries (user records)
users = [
    {"name": "Alice", "age": 30, "city": "New York"},
    {"name": "Bob", "age": 24, "city": "Los Angeles"},
    {"name": "Charlie", "age": 35, "city": "Chicago"},
    {"name": "David", "age": 24, "city": "Houston"}
]

print("Original users:")
for user in users:
    print(user)

# Sort the list of dictionaries by the 'age' key
# Using a lambda function as the key for sorting
sorted_users = sorted(users, key=lambda user: user["age"])

print("\nSorted users by age:")
for user in sorted_users:
    print(user)

# To sort by multiple keys (e.g., age then name for tie-breaking)
sorted_users_multi_key = sorted(users, key=lambda user: (user["age"], user["name"]))

print("\nSorted users by age then name:")
for user in sorted_users_multi_key:
    print(user)
```

```
Original users:
{'name': 'Alice', 'age': 30, 'city': 'New York'}
{'name': 'Bob', 'age': 24, 'city': 'Los Angeles'}
{'name': 'Charlie', 'age': 35, 'city': 'Chicago'}
{'name': 'David', 'age': 24, 'city': 'Houston'}

Sorted users by age:
{'name': 'Bob', 'age': 24, 'city': 'Los Angeles'}
{'name': 'David', 'age': 24, 'city': 'Houston'}
{'name': 'Alice', 'age': 30, 'city': 'New York'}
{'name': 'Charlie', 'age': 35, 'city': 'Chicago'}

Sorted users by age then name:
{'name': 'Bob', 'age': 24, 'city': 'Los Angeles'}
{'name': 'David', 'age': 24, 'city': 'Houston'}
{'name': 'Alice', 'age': 30, 'city': 'New York'}
{'name': 'Charlie', 'age': 35, 'city': 'Chicago'}
```

TASK-3

```
def add(x, y):
    """Adds two numbers and returns the result."""
    return x + y

def subtract(x, y):
    """Subtracts two numbers and returns the result."""
    return x - y

def multiply(x, y):
    """Multiplies two numbers and returns the result."""
    return x * y

def divide(x, y):
    """Divides two numbers and returns the result.
    Handles division by zero by returning an error message."""
    if y == 0:
        return "Error! Division by zero."
    return x / y
```

```
print("Select operation:")
print("1. Add")
print("2. Subtract")
print("3. Multiply")
print("4. Divide")

while True:
    choice = input("Enter choice(1/2/3/4): ")

    if choice in ('1', '2', '3', '4'):
        try:
            num1 = float(input("Enter first number: "))
            num2 = float(input("Enter second number: "))
        except ValueError:
            print("Invalid input. Please enter numbers only!")
            continue

        if choice == '1':
            print(num1, "+", num2, "=", add(num1, num2))
        elif choice == '2':
            print(num1, "-", num2, "=", subtract(num1, num2))
        elif choice == '3':
            print(num1, "*", num2, "=", multiply(num1, num2))
        elif choice == '4':
            result = divide(num1, num2)
            print(num1, "/", num2, "=", result)

        # Check if user wants another calculation
        next_calculation = input("Let's do next calculation? (yes/no): ")
        if next_calculation.lower() == "no":
            break
    else:
        print("Invalid Input")
```

```
Select operation:
1. Add
2. Subtract
3. Multiply
4. Divide
Enter choice(1/2/3/4): 1
Enter first number: 2
Enter second number: 3
2.0 + 3.0 = 5.0
Let's do next calculation? (yes/no): 4
Enter choice(1/2/3/4): 4
Enter first number: 23
Enter second number: 32
23.0 / 32.0 = 0.71875
Let's do next calculation? (yes/no): yes
Enter choice(1/2/3/4): 1
Enter first number: 2
Enter second number: 3
2.0 + 3.0 = 5.0
Let's do next calculation? (yes/no): No
```

Task-4

```
def is_armstrong_initial(number):
    """Checks if a given number is an Armstrong number."""
    # Convert the number to a string to determine the number of digits
    num_str = str(number)
    num_digits = len(num_str)

    sum_of_powers = 0
    # Iterate through each digit of the number
    for digit_char in num_str:
        digit = int(digit_char)
        sum_of_powers += digit ** num_digits

    # Compare the sum of powers with the original number
    return sum_of_powers == number

# Example usage to test the function
print(f"Is 153 an Armstrong number? {is_armstrong_initial(153)}") # Expected: True
print(f"Is 9 an Armstrong number? {is_armstrong_initial(9)}")     # Expected: True
print(f"Is 10 an Armstrong number? {is_armstrong_initial(10)}")    # Expected: False
print(f"Is 371 an Armstrong number? {is_armstrong_initial(371)}") # Expected: True
print(f"Is 1634 an Armstrong number? {is_armstrong_initial(1634)}") # Expected: True
print(f"Is 123 an Armstrong number? {is_armstrong_initial(123)}") # Expected: False
```

```
Is 153 an Armstrong number? True
Is 9 an Armstrong number? True
Is 10 an Armstrong number? False
```

```
Is 371 an Armstrong number? True
Is 1634 an Armstrong number? True
Is 123 an Armstrong number? False
```

```
Start coding or generate with AI.
```

```
def is_armstrong_optimized(number):
    """Checks if a given number is an Armstrong number using an optimized approach."""
    original_number = number

    # Step 2: Count the number of digits using a while loop and integer division
    num_digits = 0
    temp_count_number = number
    if number == 0:
        num_digits = 1
    else:
        while temp_count_number > 0:
            temp_count_number //= 10
            num_digits += 1

    # Step 3: Initialize sum_of_powers and temp_number
    sum_of_powers = 0
    temp_number = original_number

    # Step 4-7: Extract digits, calculate powers, and sum them
    while temp_number > 0:
        digit = temp_number % 10  # Extract last digit
        sum_of_powers += digit ** num_digits # Add digit raised to num_digits
        temp_number //= 10 # Remove last digit

    # Step 8: Compare sum_of_powers with the original number
    return sum_of_powers == original_number

# Step 9: Test the optimized function
print(f"Is 153 an Armstrong number? {is_armstrong_optimized(153)}") # Expected: True
print(f"Is 9 an Armstrong number? {is_armstrong_optimized(9)}")     # Expected: True
print(f"Is 10 an Armstrong number? {is_armstrong_optimized(10)}")    # Expected: False
print(f"Is 371 an Armstrong number? {is_armstrong_optimized(371)}") # Expected: True
print(f"Is 1634 an Armstrong number? {is_armstrong_optimized(1634)}") # Expected: True
print(f"Is 123 an Armstrong number? {is_armstrong_optimized(123)}") # Expected: False
```

```
Is 153 an Armstrong number? True
Is 9 an Armstrong number? True
Is 10 an Armstrong number? False
Is 371 an Armstrong number? True
Is 1634 an Armstrong number? True
Is 123 an Armstrong number? False
```

Summarize Improvements

Subtask:

Provide a summary comparing the two versions of the Armstrong number program, highlighting the specific improvements made in the optimized version and their impact on performance or clarity.

Comparison and Optimization Summary

The two functions, `is_armstrong_initial` and `is_armstrong_optimized`, both check if a given number is an Armstrong number, but they differ significantly in their approach to digit counting and extraction, leading to improvements in the optimized version.

`is_armstrong_initial` (Initial Version):

1. **Digit Counting:** Converts the `number` to a string (`num_str = str(number)`) and then uses `len(num_str)` to get the number of digits.
2. **Digit Extraction:** Iterates through the characters of the string (`for digit_char in num_str:`), converts each character back to an integer (`digit = int(digit_char)`), and then raises it to the power of `num_digits`.

`is_armstrong_optimized` (Optimized Version):

1. **Digit Counting:** Uses a `while` loop with integer division (`temp_count_number //= 10`) to count digits arithmetically, avoiding string conversion.
2. **Digit Extraction:** Uses a `while` loop with the modulo operator (`digit = temp_number % 10`) to extract the last digit and integer division (`temp_number //= 10`) to remove it, all through arithmetic operations.

Specific Optimizations and Their Impact:

- **Avoidance of String Conversions:** The most significant optimization in `is_armstrong_optimized` is the complete removal of string conversions. In `is_armstrong_initial`, converting the number to a string and then back to individual digits involves overhead related to string manipulation (memory allocation for the string, character-by-character processing, and type conversions).
- **Arithmetic Operations:** The optimized version relies purely on arithmetic operations (integer division and modulo) for both counting digits and extracting them. These operations are generally much faster and more memory-efficient than string-based operations in Python for numerical tasks.

Impact on Performance and Clarity:

- **Performance:** The optimized version is expected to be faster, especially for very large numbers. String operations can become a bottleneck as the number of digits increases. Arithmetic operations are native to the CPU and are typically highly optimized. This reduces the computational cost and memory footprint.
- **Clarity:** While the initial version might appear slightly simpler for someone less familiar with arithmetic digit extraction, the optimized version demonstrates a more fundamental understanding of number properties and often leads to cleaner, more Pythonic code when dealing with numerical manipulations. For experienced programmers, arithmetic methods are often considered more elegant and efficient.

In summary, the `is_armstrong_optimized` function provides a more efficient solution by replacing string-based digit processing with arithmetic operations. This change directly translates to improved performance due to reduced overhead from type conversions and memory management, making it the preferred approach for checking Armstrong numbers, particularly when dealing with potentially large inputs.

Final Task

Subtask:

Conclude the task by presenting both versions of the Armstrong number program and the summary of improvements.

Summary:

Data Analysis Key Findings

- **Initial Program Correctness:** The `is_armstrong_initial` function successfully identified Armstrong numbers (153, 9, 371, 1634) and non-Armstrong numbers (10, 123) by converting the number to a string to count digits and then summing the powers of individual digits.
- **Optimized Program Correctness:** The `is_armstrong_optimized` function also correctly validated the same set of Armstrong and non-Armstrong numbers, confirming its functional accuracy.
- **Key Optimization:** The optimized version significantly improved efficiency by avoiding string conversions. Instead of converting the number to a string (`str(number)`) to count digits and extract them, it used arithmetic operations (integer division `//` and modulo `\%`) to achieve the same results.
- **Performance Impact:** This shift from string-based operations to pure arithmetic operations in the optimized version reduces computational overhead and memory usage, leading to faster execution, especially for larger numbers. Arithmetic operations are inherently more efficient for numerical tasks in Python.

Insights or Next Steps

- The optimized approach using arithmetic operations is generally preferred for numerical problems involving digit manipulation due to its superior performance and memory efficiency compared to string-based methods.
- For future work, consider conducting performance benchmarks (e.g., using Python's `timeit` module) on both functions with a wider range of input numbers, including very large ones, to quantitatively measure the performance gains of the optimized version.