Problem 1: Expression Evaluator Usint normal function

Task: Write a function evaluate_expression(expr) that evaluates a mathematical expression provided as a string containing single-digit numbers and operators (+, -, *, /). The function should compute the result following the correct order of operations (PEMDAS/BODMAS).

Constraints:

- Do not use eval() or similar built-in functions to evaluate the expression.
- The expression may contain parentheses to indicate order of operations.
- Use loops or any other wat to parse the string and the best for you to best match-case statement to identify and apply operators.
- Handle division operations as floating-point division.

Example:

`evaluate_expression("3+5*2-8/4")`

Output:

10.0

Requirements:

- Implement operator precedence.
- Support for parentheses is optional but will be considered for extra credit.
- Provide error handling for invalid expressions.

```
# Path: evaluate_expression.py

def evaluate_expression(expr: str) -> float:
    """
    Evaluate a mathematical expression string following the PEMDAS/BODMAS rules.
    Handles addition, subtraction, multiplication, and division with parentheses (optional).

    Args:
        expr (str): The mathematical expression as a string.

    Returns:
        float: The result of the evaluation.
    """
    def tokenize(expression):
        """Convert the input string into a list of tokens (numbers and operators)."""
        tokens = []
        num = ""
        for char in expression:
            if char.isdigit():
                num += char  # Accumulate multi-digit numbers
            else:
                if num:
                    tokens.append(num)
                    num = ""
                if char in "+-*/()":
                    tokens.append(char)
        if num:
            tokens.append(num)  # Append the last number if any
        return tokens

    def apply_operator(op, a, b):
        """Apply an operator to two operands."""
        match op:
            case "+":
                return a + b
            case "-":
                return a - b
            case "*":
                return a * b
            case "/":
                return a / b if b != 0 else float('inf')  # Handle division by zero

    def evaluate(tokens):
        """Evaluate a tokenized expression without parentheses."""
        # Process multiplication and division first
        stack = []
        i = 0
        while i < len(tokens):
            token = tokens[i]
            if token in "*/":
                prev = stack.pop()
                next_val = float(tokens[i + 1])
                stack.append(apply_operator(token, prev, next_val))
                i += 1  # Skip next number
            elif token in "+-":
                stack.append(token)
            else:
                stack.append(float(token))
            i += 1

        # Process addition and subtraction
        result = stack[0]
        i = 1
        while i < len(stack):
            op = stack[i]
            num = stack[i + 1]
            result = apply_operator(op, result, num)
            i += 2

        return result
    def evaluate_with_parentheses(tokens):
        """Recursively evaluate expressions with parentheses."""
        stack = []
        i = 0
        while i < len(tokens):
            if tokens[i] == "(":
                # Find matching closing parenthesis
                open_brackets = 1
                j = i + 1
                while j < len(tokens) and open_brackets > 0:
                    if tokens[j] == "(":
                        open_brackets += 1
                    elif tokens[j] == ")":
                        open_brackets -= 1
                    j += 1

                # Evaluate sub-expression inside parentheses
                sub_result = evaluate_with_parentheses(tokens[i + 1:j - 1])
                stack.append(sub_result)
                i = j - 1  # Move to token after closing parenthesis
            else:
                stack.append(tokens[i])
            i += 1

        return evaluate(stack)

    try:
        tokens = tokenize(expr)
        return evaluate_with_parentheses(tokens)
    except Exception as e:
        return f"Error: Invalid Expression ({e})"

# Example
result = evaluate_expression("3+5*2-8/4")
print(result)
```

```
11.0
```

Problem 2: Recursive Fibonacci Function

The Fibonacci series is a sequence where each number is the sum of the two preceding ones.

Fibonacci Series Formula.

$$F(n) = F(n-1) + F(n-2)$$

with seed values:

$$F(0) = 0, \quad F(1) = 1$$

Task:

Write a recursive function fibonacci_sequence(n) that returns a list containing the Fibonacci sequence starting from 0, with the length of the list being n. The Fibonacci sequence is defined as:

`F(0) = 0`

`F(1) = 1`

`F(n) = F(n-1) + F(n-2) for n > 1`

Requirements:

- Implement the function using recursion.
- The function should return a list of length n, starting from 0.
- Validate that the input n is a non-negative integer.
- Optimize the function to handle larger values of n without exceeding the maximum recursion depth (consider using memoization).

Example:

`print(fibonacci_sequence(10))`

output:

`[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]`

Considerations:

1. Without optimization, recursive functions for Fibonacci numbers can be inefficient for larger n
2. Since the function returns a list, you can build the list recursively by appending the next Fibonacci number until the list reaches the desired length n.

```
def fibonacci_sequence(n):
    # Check if n is a non-negative integer
    if not isinstance(n, int) or n < 0:
        raise ValueError("Input must be a non-negative integer")

    # Memoization helper function
    memo = {0: 0, 1: 1}

    def fibonacci(n):
        if n not in memo:
            memo[n] = fibonacci(n-1) + fibonacci(n-2)
        return memo[n]

    # Generate Fibonacci sequence up to n terms
    return [fibonacci(i) for i in range(n)]

print(fibonacci_sequence(10))
```

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Problem 3: Advanced Lambda Function Problem

Task: Create a list of dictionaries representing students with keys 'name', 'age', and 'grade'. Write a program that:

- Sorts the list of students in descending order based on their grades using a lambda function.
- Filters out students who are below a certain age using a lambda function.
- Calculates the average grade of the filtered list using a lambda function combined with a higher-order function.

Requirements:

- Use **multiple lambda functions** for sorting, filtering, and calculation.
- Prompt the user to enter the minimum age for filtering.
- Display the sorted list, the filtered list, and the average grade.

Example:

```
students = [
    {'name': 'Alice', 'age': 20, 'grade': 88},
    {'name': 'Bob', 'age': 19, 'grade': 75},
    {'name': 'Charlie', 'age': 22, 'grade': 93},
    {'name': 'David', 'age': 21, 'grade': 85}
]
```

If the user enters `age_threshold = 20`, the program should:

- Use a lambda function to sort students by grade in descending order.

- Use a lambda function to filter out students younger than 20.
- Use a lambda function to calculate the average grade of the filtered students.

Sorted Students by Grade: [{'name': 'Charlie', 'age': 22, 'grade': 93}, {'name': 'Alice', 'age': 20, 'grade': 88}, {'name': 'David', 'age': 21, 'grade': 85}, {'name': 'Bob', 'age': 19, 'grade': 75}]

Filtered Students (Age >= 20): [{'name': 'Charlie', 'age': 22, 'grade': 93}, {'name': 'Alice', 'age': 20, 'grade': 88}, {'name': 'David', 'age': 21, 'grade': 85}]

Average Grade of Filtered Students: 88.66666666666667

Hints:

- You should build at least three lambda functions:
  - One for sorting.
  - One for filtering.
  - One for calculating the average grade (possibly within a higher-order function like reduce or using a combination of map and lambda).
- Feel free to use additional lambda functions if needed.

```python
from functools import reduce

# Sample list of students
students = [
    {'name': 'Alice', 'age': 20, 'grade': 88},
    {'name': 'Bob', 'age': 19, 'grade': 75},
    {'name': 'Charlie', 'age': 22, 'grade': 93},
    {'name': 'David', 'age': 21, 'grade': 85}
]

# Prompt user to enter minimum age threshold
age_threshold = int(input("Enter the minimum age for filtering: "))

# 1. Sort the students by grade in descending order using a lambda function
sorted_students = sorted(students, key=lambda x: x['grade'], reverse=True)

# 2. Filter out students who are younger than specified age using a lambda function
filtered_students = list(filter(lambda x: x['age'] >= age_threshold, students))

# 3. Calculate average grade of the filtered list using a lambda function and reduce
average_grade = reduce(lambda acc, student: acc + student['grade'], filtered_students, 0) / len(filtered_students) if filtered_students else 0

print("Sorted Students by Grade:", sorted_students)
print(f"Filtered Students (Age >= {age_threshold}):", filtered_students)
print(f"Average Grade of Filtered Students: {average_grade}")
```

```
Enter the minimum age for filtering: 21
Sorted Students by Grade: [{'name': 'Charlie', 'age': 22, 'grade': 93}, {'name': 'Alice', 'age': 20, 'grade': 88}, {'name': 'David', 'age': 21, 'grade': 85}, {'name': 'Bob',
Filtered Students (Age >= 21): [{'name': 'Charlie', 'age': 22, 'grade': 93}, {'name': 'David', 'age': 21, 'grade': 85}]
```

```
Average Grade of Filtered Students: 89.0
```

Problem 4: Input Validation with Decorators (Multiple Inputs)

Task:

Write a decorator validate_inputs that can be applied to any function to validate its input arguments. Specifically, create a function calculate_power(base, exponent) that calculates the power of a number (base raised to the exponent). Use the decorator to ensure that:

- The input base is a real number (int or float).
- The input exponent is an integer.
- The input exponent is non-negative.
- If any input is invalid, the decorator should raise a ValueError with an appropriate error message.

Requirements:

Implement the validate_inputs decorator to validate multiple inputs. Apply the decorator to the calculate_power function. Do not modify the original calculate_power function's signature. Handle exceptions gracefully in your program.

Example:

```
@validate_inputs
def calculate_power(base, exponent):
    return base ** exponent
```

Usage:

```
try:
    print(calculate_power(2, 3))    # Should output 8
    print(calculate_power(5, -2))   # Should raise ValueError
    print(calculate_power('a', 2))  # Should raise ValueError
except ValueError as e:
    print("Error:", e)
```

```
# Define the decorator to validate inputs
def validate_inputs(func):
    def wrapper(base, exponent):
        # int or float
        if not isinstance(base, (int, float)):
            raise ValueError("The base must be a real number (int or float).")

        # if integer
        if not isinstance(exponent, int):
            raise ValueError("The exponent must be an integer.")

        # if non-negative
        if exponent < 0:
            raise ValueError("The exponent must be non-negative.")

        # Call original function if all validations pass
        return func(base, exponent)

    return wrapper

# Apply decorator to calculate_power function
@validate_inputs
def calculate_power(base, exponent):
    return base ** exponent

# exception handling
try:
    print(calculate_power(2, 3))  # 8
    print(calculate_power(5, -2))  # ValueError
    print(calculate_power('a', 2))  # ValueError
except ValueError as e:
    print("Error:", e)
```

```
8
Error: The exponent must be non-negative.
```