### Problem 1: Robust Calculator

🔹 **Problem Statement:**  
You are building a calculator that takes two numbers and an operator (+, -, \*, /, %, \*\*).

* Handle invalid operations.
* Handle division by zero.
* Handle incorrect input types.

**Stubbed Code:**

def calculator(a, b, operator):  
 """  
 Performs a calculation based on the given operator.  
   
 :param a: First number (int/float)  
 :param b: Second number (int/float)  
 :param operator: String representing an operation (+, -, \*, /, %, \*\*)  
 :return: Computed result or error message  
 """  
 try:  
 # TODO: Implement operation handling and raise exceptions for invalid cases  
 pass   
 except ZeroDivisionError:  
 pass # TODO: Handle division by zero  
 except ValueError:  
 pass # TODO: Handle invalid numbers  
 except TypeError:  
 pass # TODO: Handle non-numeric input  
 except Exception as e:  
 pass # TODO: Handle any unexpected exceptions  
  
# Example Usage:  
print(calculator(10, 0, "/")) # Should return: "Error: Division by zero"  
print(calculator(10, "five", "+")) # Should return: "Error: Invalid input type"  
print(calculator(10, 5, "$")) # Should return: "Error: Unsupported operator"

### Problem 2: Nested Exception Handling – Banking System

🔹 **Problem Statement:**  
Simulate a **banking system** where users can **withdraw money** from their account.

* Raise ValueError if the amount is negative.
* Raise InsufficientFundsError if the withdrawal amount is greater than the balance.
* Handle unexpected exceptions gracefully.

**Stubbed Code:**

class InsufficientFundsError(Exception):  
 """Custom exception for insufficient balance"""  
 pass  
  
class BankAccount:  
 def \_\_init\_\_(self, balance):  
 self.balance = balance  
  
 def withdraw(self, amount):  
 """  
 Withdraws money from the account.  
   
 :param amount: Amount to withdraw  
 :return: Remaining balance or error message  
 """  
 try:  
 # TODO: Implement withdrawal logic  
 pass   
 except ValueError:  
 pass # TODO: Handle negative withdrawal amounts  
 except InsufficientFundsError:  
 pass # TODO: Handle insufficient funds  
 except Exception as e:  
 pass # TODO: Handle unexpected errors  
  
# Example Usage:  
account = BankAccount(100)  
print(account.withdraw(150)) # Should raise InsufficientFundsError  
print(account.withdraw(-10)) # Should raise ValueError

### Problem 3: Data Processing with Exception Handling

🔹 **Problem Statement:**  
Write a function that processes a list of numbers and performs **division, type conversion, and list indexing**.

* Handle ZeroDivisionError when dividing by zero.
* Handle ValueError for invalid conversions.
* Handle IndexError if an invalid index is accessed.

**Stubbed Code:**

def process\_data(data, index):  
 """  
 Processes data with error handling.  
   
 :param data: List of numbers (strings that should be converted to int)  
 :param index: Index to divide with  
 :return: Processed result or error message  
 """  
 try:  
 # TODO: Convert elements to integers  
 # TODO: Perform division  
 pass   
 except ZeroDivisionError:  
 pass # TODO: Handle division by zero  
 except ValueError:  
 pass # TODO: Handle invalid conversions  
 except IndexError:  
 pass # TODO: Handle out-of-bounds index  
 except Exception as e:  
 pass # TODO: Handle unexpected errors  
  
# Example Usage:  
data\_list = ["10", "20", "0", "40"]  
print(process\_data(data\_list, 2)) # Should handle division by zero  
print(process\_data(["10", "abc", "30"], 1)) # Should handle ValueError  
print(process\_data([10, 20], 5)) # Should handle IndexError