**Lab 1: Basic Try and Except Block**

**Objective:** Write a Python program that takes two numbers as input and performs division. Use a try and except block to handle division by zero.

**Task:**

1. Accept two numbers from the user.
2. Attempt to divide the first number by the second.
3. Catch the ZeroDivisionError exception if the denominator is zero and display a proper message.

**Lab 2: Handling Multiple Exceptions**

**Objective:** Write a program that takes input from the user and converts it to an integer. Handle ValueError and ZeroDivisionError separately.

**Task:**

1. Ask the user to enter two numbers.
2. Convert them to integers and divide them.
3. Handle cases where input is not a number (ValueError).
4. Handle division by zero (ZeroDivisionError).

**Lab 3: Using Finally in Exception Handling**

**Objective:** Implement a program where the finally block is always executed, whether an exception occurs or not.

**Task:**

1. Write a function that opens a file and reads its content.
2. Handle FileNotFoundError properly.
3. Ensure the file is closed properly using finally.

**Lab 4: Raising Exceptions Manually**

**Objective:** Use the raise keyword to generate custom exceptions in a function.

**Task:**

1. Write a function that accepts age as input.
2. If age is below 18, raise a ValueError stating "Age must be 18 or above".
3. Otherwise, print "Eligible".

**Lab 5: Defining a Custom Exception Class**

**Objective:** Define a custom exception and raise it when necessary.

**Task:**

1. Create a class NegativeNumberError inheriting from Exception.
2. Write a function that checks if a number is positive.
3. If negative, raise NegativeNumberError with a meaningful message.

**Lab 6: Using Custom Exception Handlers**

**Objective:** Define and use custom exceptions to validate input.

**Task:**

1. Create a class InvalidUsernameError inheriting from Exception.
2. Write a function to check if a username contains special characters.
3. Raise InvalidUsernameError if it does.

**Lab 7: The Exception Hierarchy**

**Objective:** Understand Python’s built-in exception hierarchy.

**Task:**

1. Write a program that demonstrates handling exceptions in a hierarchy.
2. Catch IndexError, LookupError, and Exception in a structured way.

**Lab 8: Chaining Except Blocks**

**Objective:** Write a program that demonstrates handling multiple exceptions in a chain.

**Task:**

1. Write a function that reads from a file and divides two numbers.
2. Use multiple except blocks to handle FileNotFoundError, ZeroDivisionError, and ValueError.

**Lab 9: Handling Multiple Exceptions in One Except Block**

**Objective:** Use a single except block to catch multiple exceptions.

**Task:**

1. Modify the previous exercise.
2. Catch FileNotFoundError and ZeroDivisionError using a tuple in a single except block.

**Lab 10: Nested Try and Except Blocks**

**Objective:** Implement nested try and except blocks.

**Task:**

1. Write a program where a function calls another function inside a try block.
2. Handle errors in both inner and outer try-except blocks.

**Lab 11: Using Else Block in Try-Except**

**Objective:** Implement an else block with a try-except statement.

**Task:**

1. Write a program that attempts to open a file.
2. If successful, read and print its contents.
3. Use else to print a success message.

**Lab 12: Logging Exceptions**

**Objective:** Use the logging module to log exceptions.

**Task:**

1. Modify a previous program to log exceptions instead of just printing them.
2. Use different log levels (ERROR, WARNING).

**Lab 13: Interactive Exception Handling in Python Shell**

**Objective:** Run exception-handling code interactively in the Python shell.

**Task:**

1. Open Python shell and run different exception-handling scenarios.
2. Experiment with different types of exceptions.

**Lab 14: Running Python Scripts with Command-Line Arguments**

**Objective:** Use sys.argv to read command-line arguments.

**Task:**

1. Write a Python script that accepts two numbers as command-line arguments.
2. Convert them to integers and perform division.
3. Handle errors using try-except.

**Lab 15: Handling Insufficient Command-Line Arguments**

**Objective:** Check if the required number of arguments are passed.

**Task:**

1. Modify the previous script.
2. If fewer arguments are passed, print an error message.

**Lab 16: Using argparse for Command-Line Parsing**

**Objective:** Use argparse for better command-line argument handling.

**Task:**

1. Modify the command-line script to use argparse.
2. Define required and optional arguments.
3. Provide help messages.

**Lab 17: Using argparse with Default Values**

**Objective:** Use default values in argparse.

**Task:**

1. Modify the argparse script.
2. Set default values for missing arguments.

**Lab 18: Handling Different Argument Types in argparse**

**Objective:** Use different argument types (int, float, str) in argparse.

**Task:**

1. Modify the script to accept multiple types of input.
2. Convert and validate them accordingly.

**Lab 19: Adding Choices in argparse**

**Objective:** Restrict input to specific values using choices in argparse.

**Task:**

1. Modify the script to accept an operation (add, subtract, multiply, divide).
2. Use choices to restrict input.

**Lab 20: Combining Exception Handling with argparse**

**Objective:** Use argparse along with exception handling.

**Task:**

1. Modify the script to perform arithmetic operations.
2. Handle incorrect inputs using try-except.