Task 1: Handling Division by Zero

Objective: Understand how to handle division by zero errors using a `try-except` block.

Steps:

1. Create a Python program that takes two numbers as input.
2. Perform division between the two numbers.
3. Handle a `ZeroDivisionError` if the second number is zero and print a message like "Error: Division by zero is not allowed."
4. If no error occurs, print the result of the division.

Code Example:

```
try:
    num1 = float(input("Enter the first number: "))
    num2 = float(input("Enter the second number: "))
    result = num1 / num2
except ZeroDivisionError:
    print("Error: Division by zero is not allowed.")
else:
    print(f"The result is: {result}")
```

Questions:

1. What exception is raised when trying to divide a number by zero?
2. How does the program respond when an exception is raised?
3. What would happen if you removed the `except` block from the program?

Task 2: Handling Invalid User Input

Objective: Learn how to handle `ValueError` when user input cannot be converted to the expected data type.

Steps:

1. Prompt the user to enter their age.
2. Attempt to convert the input to an integer.
3. If the input is not a valid number (e.g., a string), handle the `ValueError` and print a message like "Invalid input. Please enter a valid number."
4. If the input is valid, print the user's age.

Code Example:

```
try:
    age = int(input("Please enter your age: "))
except ValueError:
    print("Invalid input. Please enter a valid number.")
else:
    print(f"Your age is {age}.")
```

Questions:

1. What is the purpose of the `int()` function in this task?
2. Why does the program raise a `ValueError` if the input is not a valid integer?
3. What changes would you make to allow the user to enter their age repeatedly until a valid input is given?

Task 3: Reading from a File with Error Handling

Objective: Learn how to handle `FileNotFoundError` when attempting to open a file that doesn't exist.

Steps:

1. Write a program that tries to open a file called "data.txt".
2. Handle the `FileNotFoundError` in case the file does not exist, and print a message like "Error: File not found."
3. If the file exists, read and print its content.
4. Ensure that the file is closed after attempting to read, whether successful or not.

Code Example:

```
try:
    with open("data.txt", "r") as file:
        content = file.read()
except FileNotFoundError:
    print("Error: File not found.")
else:
    print("File content:\n", content)
```

Questions:

1. What happens if "data.txt" is not found in the same directory as the script?
2. Why is the `with` statement used to open the file in this task?
3. What other types of errors might occur when working with files, and how can they be handled?

Task 4: Handling Multiple Exceptions

Objective: Learn how to handle multiple types of exceptions in a single `try-except` block.

Steps:

1. Write a program that prompts the user to enter a number and then tries to divide 100 by the number.
2. Handle both `ZeroDivisionError` (if the user enters zero) and `ValueError` (if the user enters a non-numeric value).
3. Print appropriate error messages for both exceptions.
4. If no exceptions are raised, print the result of the division.

Code Example:

```
try:
    num = float(input("Enter a number: "))
    result = 100 / num
except ZeroDivisionError:
    print("Error: Division by zero is not allowed.")
except ValueError:
    print("Error: Invalid input. Please enter a numeric value.")
else:
    print(f"The result of 100 divided by {num} is {result}.")
```

Questions:

1. What is the purpose of using multiple `except` blocks in this program?
2. How does the program handle non-numeric input and division by zero differently?
3. How would you modify the code to handle more exceptions, such as an `OverflowError`?

Task 5: Using Finally to Close Resources

Objective: Understand the use of the `finally` block to ensure resources (like files) are properly closed, regardless of whether an exception occurred or not.

Steps:

1. Write a program that opens a file for writing.
2. In the `try` block, attempt to write some data to the file.
3. In the `finally` block, ensure the file is closed after the operation, whether or not an exception occurs.

4. If an exception occurs while writing (e.g., the file is read-only), handle the exception and print an error message.

Code Example:

```
try:
    file = open("output.txt", "w")
    file.write("Hello, world!")
except IOError:
    print("Error: Unable to write to file.")
finally:
    file.close()
    print("File closed.")
```

Questions:

1. What would happen if the `finally` block was removed?
2. How does the `finally` block guarantee that the file is closed even if an exception occurs?
3. What other use cases can you think of where the `finally` block would be useful?

Task 6: Raising Custom Exceptions

Objective: Learn how to raise custom exceptions to control the flow of a program.

Steps:

1. Write a program that defines a custom exception class called `InvalidAgeError`.
2. Prompt the user for their age, and raise an `InvalidAgeError` if the age is negative or greater than 120.
3. Handle the `InvalidAgeError` and print a message like "Error: Invalid age entered."
4. If the input is valid, print the entered age.

Code Example:

```
class InvalidAgeError(Exception):
    pass

try:
    age = int(input("Please enter your age: "))
    if age < 0 or age > 120:
        raise InvalidAgeError("Age must be between 0 and 120.")
except InvalidAgeError as e:
    print(f"Error: {e}")
except ValueError:
    print("Error: Please enter a valid number.")
else:
    print(f"Your age is {age}.")
```

Questions:

1. What is the purpose of defining a custom exception like `InvalidAgeError`?
2. Why do we raise the custom exception when the age is outside the allowed range?
3. How can custom exceptions improve the error handling process in a program?

Task 7: Using Assertions for Validations

Objective: Learn how to use `assert` statements for debugging and validation in the program.

Steps:

1. Write a program that asks the user for their age.
2. Use an `assert` statement to check if the age is a positive number. If not, raise an `AssertionError`.
3. If the assertion fails, print an error message indicating the invalid input.
4. If the assertion passes, print the valid age.

Code Example:

```
try:
    age = int(input("Enter your age: "))
    assert age > 0, "Age must be a positive number."
except AssertionError as e:
    print(f"Error: {e}")
except ValueError:
    print("Error: Please enter a valid number.")
else:
    print(f"Your age is {age}.")
```

Questions:

1. How does the `assert` statement help in validating the input?
2. What would happen if the user enters a non-numeric value?
3. How does the program handle the `AssertionError`?

Task 8: Handling Multiple File Operations

Objective: Learn how to handle multiple file operations and ensure proper error handling and resource management.

Steps:

1. Write a program that tries to read data from a file and write it to another file.
2. If the source file does not exist, handle the `FileNotFoundError`.
3. If the program encounters any error while writing to the destination file, handle the `IOError`.
4. Ensure the files are properly closed using a `finally` block.

Code Example:

```
try:
    with open("source.txt", "r") as src_file:
        data = src_file.read()

    with open("destination.txt", "w") as dest_file:
        dest_file.write(data)
except FileNotFoundError:
    print("Error: The source file does not exist.")
except IOError:
    print("Error: There was an issue with file writing.")
finally:
    print("File operations completed.")
```

Questions:

1. What happens if the source file does not exist?
2. How are the files handled within the `with` statement in this task?
3. What other exceptions might arise in file operations, and how can you handle them?