Note to Students

1. ***Learning is a unique experience for each person; make it meaningful and valuable for yourself.*** If you are pursuing this knowledge for long-term understanding, you should learn in the correct ways.
2. **Do not use any AI assistance**. This assignment is designed to improve human skills, not just to get answers from AI. If your goal is only to finish the work, please do not submit it; I will appreciate your honesty even more.
3. This assignment is created to help you understand how to solve problems with code, which is a fundamental and extremely important skill in the field of computer science.
4. Logical thinking: if you use AI improperly, it will not improve your skills. In fact, it may reduce your skills. You will be like someone asking a calculator just to compute $1 + 1$ (someone who knows the calculator is capable but does not know the fundamentals of calculation).
5. Logical thinking: copying your friend's work also does not help improve your skills.
6. To truly improve, you should work on your own. Ask questions or search for help in ways that enhance your understanding and coding abilities.

The question and its examples should match each other. However, if you find any mistakes or unclear points, please let me know.

Question 31: Complex Configuration File Parser

Write a program that reads and validates a configuration from either a file or direct user input. The configuration contains key-value pairs where values can be simple strings, comma-separated lists, or nested key-value pairs enclosed in braces .

Program Logic:

1. Ask the user whether to load from a 'file' or 'manual' input.
2. Based on the choice, read the configuration data. For manual input, the user can type multiple lines and enter 'DONE' on a new line to finish.
3. Parse each line into a key and value based on the first `=`.
   Ignore empty lines and lines starting with `#`.
4. Process the value: simple strings, comma-separated lists, or parse the config "`{key=val;...}`".
5. Store the parsed data in a nested dictionary.
6. **Validate** the parsed data. If any rule is violated, print a specific error message and terminate.
7. If validation is successful, print the parsed data using the specific multi-line format shown in the examples. Lists should be output as comma-separated strings.

Validation Rules:

- The `port` must be an integer between 1 and 65535.
- The `allowed_users` list must not be empty.
- The nested `database` dictionary must contain both `user` and `password` keys.
- The `timeout` value within the `database` dictionary must be a positive integer.

Examples

Note: The user's input is decorated with underline.

Assume `app.conf` contains:

```
# Server configuration
port=8080
allowed_users=alice,bob
database={user=db;password=secret;timeout=30}
```

Example 1: Valid Configuration from File

```
Load from 'file' or 'manual' input? file
Enter filename: app.conf
Configuration file is valid.
Parsed Data:
port: 8080
allowed_users: alice,bob
database:
  user: db
  password: secret
  timeout: 30
```

Example 2: Valid Configuration from Manual Input

```
Load from 'file' or 'manual' input? manual
Enter configuration (type 'DONE' to finish):
port=443
allowed_users=guest
database={user=readonly;password=p@ss;timeout=60}
DONE
Configuration file is valid.
Parsed Data:
port: 443
allowed_users: guest
database:
  user: readonly
  password: p@ss
  timeout: 60
```

Example 3: File Not Found

```
Load from 'file' or 'manual' input? file
Enter filename: config.missing
Error: File 'config.missing' not found.
```

Example 4: Invalid Port Number

```
Load from 'file' or 'manual' input? manual
Enter configuration (type 'DONE' to finish):
port=99999
allowed_users=admin
database={user=db;timeout=10}
DONE
Validation Error: Port must be an integer between 1 and 65535.
```

Example 5: Missing Required Database Key

```
Load from 'file' or 'manual' input? manual
Enter configuration (type 'DONE' to finish):
port=8080
allowed_users=alice,bob
database={user=db;timeout=25}
DONE
Validation Error: Database dictionary must contain both 'user' and '
    password' keys.
```

Example 6: Empty Allowed Users List

```
Load from 'file' or 'manual' input? manual
Enter configuration (type 'DONE' to finish):
port=8080
allowed_users=
database={user=db;password=secret;timeout=10}
DONE
Validation Error: Allowed users list must not be empty.
```

Example 7: Invalid Timeout Value

```
Load from 'file' or 'manual' input? manual
Enter configuration (type 'DONE' to finish):
port=3000
allowed_users=alice,bob,carol
database={user=db;password=secret;timeout=-5}
DONE
Validation Error: Database timeout must be a positive integer.
```

Question 32: Robust Square Root Calculator

Write a program that calculates the square root of a number provided by the user. The program must be robust and handle multiple types of invalid input within a loop using standard Python exceptions.

Program Logic: The program should run in a `while` loop that continues until a valid calculation is performed. Inside the loop, you must:

1. Prompt the user to enter a number.
2. Use a `try...except` block to handle the input conversion.
3. If the input is not a valid number (e.g., "abc" or empty), a `ValueError` will occur. Catch this exception and print an appropriate error message.
4. If the input is a valid number, check if it is negative. If it is, print a specific error message and continue the loop.
5. If the input is a valid, non-negative number, calculate its square root, print the result, and then `break` the loop to terminate the program.

Requirements

- The program must not use custom exceptions. It should handle errors by checking for negative values and catching the built-in `ValueError`.
- The program must loop until it successfully calculates and prints a square root.
- Use a `try...except` block to validate that the input can be converted to a number.

Examples

Note: The user's input is decorated with underline.

Example 1: Correct on First Try

```
Enter a non-negative number: 25
The square root of 25.0 is 5.0.
```

Example 2: Handling Non-Numeric and Negative Input

```
Enter a non-negative number: text
Error: Invalid input. Please enter a number.
Enter a non-negative number: -9
Error: Cannot calculate the square root of a negative number.
Enter a non-negative number: 16
The square root of 16.0 is 4.0.
```

Example 3: Handling Floating-Point Input

```
Enter a non-negative number: 7.5
The square root of 7.5 is 2.7386127875258306.
```

Example 4: Handling Zero as Input

```
Enter a non-negative number: 0
The square root of 0.0 is 0.0.
```

Example 5: Handling Empty Input

```
Enter a non-negative number:
Error: Invalid input. Please enter a number.
Enter a non-negative number: 144
The square root of 144.0 is 12.0.
```

Question 33: Sequential Expression Calculator

Write a text based calculator that evaluates mathematical expressions sequentially from left to right. The program must handle basic arithmetic operators and gracefully manage invalid input and mathematical errors.

Requirements

- The program loop until the user inputs `EXIT`. Then, the program outputs "`Goodbye!`" and exits.
- Supported operators are: `+`, `-`, `*`, `/`.
- All calculations are performed using floating-point arithmetic.
- All numbers are converted to floats for computation. If the calculation is successful, the final result is printed as a float with one decimal place.
- Must correctly handle `ValueError`, `ZeroDivisionError`, and invalid expression formats.
- **Handle Errors:**
  - If the expression contains a non-numeric value (e.g., "five"), catch the `ValueError` and print a specific error (see from the example).
  - If the user attempts to divide by zero, catch the `ZeroDivisionError` and print a specific error (see from the example).
  - If the expression is malformed (Invalid Expression Format) (e.g., ends with an operator like `5 *`, or has consecutive operators like `5 + * 3`), print a specific error (see from the example).
  - If multiple errors exist in an expression, the program should report only the first one encountered when evaluating from left to right.

Important Calculation Rule: This calculator **does not** follow the standard order of operations. It must process the expression strictly from **left to right**.

Hint:

1. The program must run in a loop that continuously prompts the user for an expression.
2. The loop terminates only when the user types the exact string `EXIT`.
3. The valid expression format must be a number followed by an operator, and so on, always ending with a number (e.g., `number op number op ... number`). There cannot be consecutive numbers or operators. An expression can also be a single number.
4. For each input, use the `.split()` method to break the expression into a list of numbers and operators.
5. Use a `try...except` block to manage the calculation and potential errors.

Examples

Note: The user's input is decorated with underline.

Example 1: Valid Expressions

```
Enter expression: 10 * 3 + 5
Result: 35.0
Enter expression: 100 / 5 - 10
Result: 10.0
Enter expression: 42
Result: 42.0
Enter expression: EXIT
Goodbye!
```

Example 2: Handling ZeroDivisionError

```
Enter expression: 8 / 4 / 2
Result: 1.0
Enter expression: 8 / 0 + 5
Error: Division by zero is not allowed.
Enter expression: EXIT
Goodbye!
```

Example 3: Handling ValueError

```
Enter expression: 5 * seven
Error: Invalid number 'seven' in expression.
Enter expression: 10 + 2.5
Result: 12.5
Enter expression: EXIT
Goodbye!
```

Example 4: Handling Invalid Expression Format

```
Enter expression: 10 +
Error: Invalid expression format.
Enter expression: 5 * + 3
Error: Invalid expression format.
Enter expression: EXIT
Goodbye!
```

Example 5: Full Session with Mixed Cases

```
Enter expression: 50 + 10 / 2 * 3
Result: 90.0
Enter expression: 10 / 0
Error: Division by zero is not allowed.
Enter expression: -10 + 5
Result: -5.0
Enter expression: my dog ate it
Error: Invalid number 'my' in expression.
Enter expression: 20 -
Error: Invalid expression format.
Enter expression: EXIT
Goodbye!
```

Question 34: Nested File Shape Area Reporter

Write a Python program that processes a series of data files containing shape information to generate a consolidated area report. The program begins by reading an `index.txt` file, which lists the names of the data files to be processed. It must calculate the area for various shapes, aggregate the results, and handle potential errors gracefully.

Requirements

- **File Structure:**
  - The program must first read a file named `index.txt`.
  - `index.txt` contains a list of data filenames, with one filename per line.
  - Each data file contains shape information, with one shape per line, in the format: `shape_name value1 [value2]`.
- **Supported Shapes & Area Formulas:**
  - `circle radius`: Area = $\pi \times \text{radius}^2$. Use `math.pi`.
  - `rectangle width height`: Area = width × height.
  - `square side`: Area = side^2.
- **Reporting:**
  - Process files in the order they appear in `index.txt`.
  - For each successfully opened file, print a summary report showing the total counts for each shape type, the total number of valid shapes, and the total area for that file.
  - The shape counts in the report must be displayed in the following order: `circle`, `rectangle`, `square`. If a shape type does not appear in a file, its count should be reported as `0`.
  - After processing all files, print a "Grand Total" summary report with the same information aggregated across all valid shapes from all processed files.
  - All area calculations should be done with floating-point numbers, and the final reported areas must be formatted to two decimal places.
- **Error Handling:**
  - If `index.txt` cannot be found, print `Error: index.txt not found.` and terminate.
  - If a data file listed in `index.txt` cannot be found, print `Error: <filename> not found.` and continue to the next file.
  - If a line within a data file is malformed for any reason (e.g., an unknown shape, incorrect number of parameters, or a non-numeric value), print `Error on line: <line_content>` and skip that line.

Examples

For the following examples, assume the given files are in the same directory as the script.

Example 1: index.txt Not Found

Assume `index.txt` *does not exist in the directory.* **Program Output:**

```
Error: index.txt not found.
```

Example 2: Successful Run

File Contents:

index.txt

```
data1.txt
data2.txt
```

data1.txt

```
circle 5
rectangle 10 4
square 2
```

data2.txt

```
square 8
rectangle 2.5 3
circle 1
```

Program Output:

```
Processing file: data1.txt
--- Report for data1.txt ---
Shapes:
  circle: 1
  rectangle: 1
  square: 1
Total shapes: 3
Total area: 122.54
--------------------

Processing file: data2.txt
--- Report for data2.txt ---
Shapes:
  circle: 1
  rectangle: 1
  square: 1
Total shapes: 3
Total area: 74.64
--------------------

--- Grand Total ---
Total shapes:
  circle: 2
  rectangle: 2
  square: 2
Total overall shapes: 6
Total overall area: 197.18
--------------------
```

Example 3: Handling Mixed Errors

File Contents:

index.txt

```
shapes_ok.txt
missing_file.txt
shapes_with_errors.txt
```

shapes_ok.txt

```
rectangle 10 10
circle 10
```

shapes_with_errors.txt

```
square 6
circle five
rectangle 3
triangle 5 5
square 2.5
```

Program Output:

```
Processing file: shapes_ok.txt
--- Report for shapes_ok.txt ---
Shapes:
  circle: 1
  rectangle: 1
  square: 0
Total shapes: 2
Total area: 414.16
--------------------

Error: missing_file.txt not found.

Processing file: shapes_with_errors.txt
Error on line: circle five
Error on line: rectangle 3
Error on line: triangle 5 5
--- Report for shapes_with_errors.txt ---
Shapes:
  circle: 0
  rectangle: 0
  square: 2
Total shapes: 2
Total area: 42.25
--------------------

--- Grand Total ---
Total shapes:
  circle: 1
  rectangle: 1
  square: 2
Total overall shapes: 4
Total overall area: 456.41
--------------------
```