Group 1: With Statement (Context Manager) and Iterators

Practice 1: With Statement (Context Manager)

Objective:

- Create a context manager to manage resources safely using the `with` statement.

Detailed Explanation:

- **Task**: Implement a context manager `FileReader` that reads a file and automatically closes it when done. This ensures that file resources are properly managed, even if exceptions occur during file operations.

Practice 2: Custom File Reader Iterator

Objective:

- Create a custom iterator to iterate over a sequence of items.

Detailed Explanation:

- **Task**: Design an iterator `FileReaderIterator` to sequentially read lines from a text file. This iterator allows efficient handling of large files and supports iteration protocols in Python, such as `__iter__` and `__next__`.

Group 2: Typing and Global, Local, Nonlocal Variables

Practice 4: Scope Demonstration with Nested Functions

Objective:

- Explore variable scopes in Python with nested functions.

Detailed Explanation:

- **Task**: Define nested functions that access variables in different scopes:
  - **Global variables**: Variables defined at the module level.
  - **Local variables**: Variables defined inside functions.
  - **Nonlocal variables**: Variables in enclosing functions (used within nested functions).

Practice 5: Variable Scopes in Classes

Objective:

- Understand variable scopes within class methods and attributes.

Detailed Explanation:

- **Task**: Create a class `VariableScopeDemo` with methods that access class-level and instance-level variables. Demonstrate how Python handles variable scopes within classes, including accessing and modifying variables at different levels of class hierarchy.

Group 4: With Statement (Context Manager)

Practice 7: Custom Context Manager

Objective:

- Create a custom context manager using the `with` statement.

Detailed Explanation:

- **Task**: Define a context manager class `Timer` that measures the time taken by a block of code to execute. Implement `__enter__` and `__exit__` methods to start and stop the timer, respectively.

Group 6: argparse

Practice 1: Basic Argument Parsing

Objective:

- Learn to parse command-line arguments using `argparse`.

Detailed Explanation:

- **Task**: Create a script `basic_parser.py` that accepts two command-line arguments: a filename and a verbosity level.
  - Use `argparse.ArgumentParser` to define and parse these arguments.
  - Print the parsed arguments to demonstrate successful parsing.

Practice 2: Required and Optional Arguments

Objective:

- Understand how to define required and optional command-line arguments.

Detailed Explanation:

- **Task**: Modify `basic_parser.py` to include a required argument for the filename and an optional argument for the verbosity level (default value should be `1`).
  - Use `add_argument` with `required=True` for the filename.
  - Provide a default value for the verbosity argument and test the script with and without specifying the verbosity level.

Practice 3: Argument Types and Choices

Objective:

- Learn to specify argument types and limit choices for command-line arguments.

Detailed Explanation:

- **Task**: Extend `basic_parser.py` to include an argument for the operation type (e.g., `read`, `write`, `append`), ensuring the argument can only take one of these specified choices.
  - Use `type` to enforce argument type and `choices` to restrict possible values.
  - Test the script with valid and invalid operation types to see the effect.

Practice 4: Parsing Multiple Arguments

Objective:

- Handle multiple command-line arguments and perform actions based on them.

Detailed Explanation:

- **Task**: Create a script `multi_parser.py` that accepts a list of filenames and a flag for recursive processing.
  - Use `nargs='+'` to accept multiple filenames.

- Use `action='store_true'` for the recursive flag.
- Print the list of filenames and whether the recursive flag is set.

Practice 5: Subparsers for Command Groups

Objective:

- Use subparsers to handle different sets of arguments for different commands.

Detailed Explanation:

- **Task**: Create a script `command_parser.py` with two subcommands: `add` and `remove`.
  - The `add` command should accept a filename and a description.
  - The `remove` command should accept a filename.
  - Use `argparse.ArgumentParser.add_subparsers` to set up the subcommands and define their arguments.
  - Test the script with both subcommands to ensure correct behavior.

Group 5: Iterators

Practice 8: Implementing a Range-like Iterator

Objective:

- Create an iterator that mimics the behavior of Python's built-in `range` function.

Detailed Explanation:

- **Task**: Define a class `CustomRange` that implements `__iter__` and `__next__` methods to generate a sequence of numbers, similar to the `range` function. **Practice 9: Reversible Iterator**

Objective:

- Implement an iterator that can traverse a sequence in both forward and reverse directions.

Detailed Explanation:

- **Task**: Define a class `ReversibleIterator` that iterates over a sequence in both directions. Implement methods to switch between forward and reverse traversal.