Assignment 1

Programming Assignment: Renaming All Files in a Folder Sequentially

Objective:
Write a program that renames all files in a specified folder by numbering them sequentially, starting from `1`. The program should keep the original file extensions intact and only change the names of the files.

Requirements:

1. **Input**:
   The program should take the path to a folder as input from the user.
2. **Output**:
   All files in the specified folder should be renamed in the format `1.ext`, `2.ext`, `3.ext`, and so on, where `.ext` is the original file extension (e.g., `.jpg`, `.txt`, `.pdf`). The numbering should start from `1` and continue for all files in the folder.
3. **File Types**:
   The program should only rename regular files and ignore directories or non-file items in the folder.
4. **File Extensions**:
   The program should retain the original file extensions (e.g., `.jpg`, `.txt`, `.png`) when renaming the files.
5. **Error Handling**:
   - The program should check if the specified folder exists and handle errors when the folder path is invalid.
   - Handle any file system-related errors, such as permission issues.
6. **Sorting**:
   The files should be renamed in the order they are listed by the operating system (e.g., alphabetically by default in most systems), or you can sort the files before renaming (e.g., alphabetically by filename).
7. **File Renaming Logic**:
   - The program should rename each file starting from `1` and incrementing by `1`.
   - For example, if there are 5 files, they should be renamed as `1.ext`, `2.ext`, `3.ext`, `4.ext`, `5.ext`.
8. **Edge Cases**:
   - The folder might be empty. In that case, no renaming should occur.
   - If the folder contains subdirectories, the program should ignore them and only rename files.

- If the folder contains files with the same name after renaming (e.g., conflicting names), the program should handle that by skipping or reporting an error.

Tasks:

1. **Input Parsing**:
   - Prompt the user to input the path to the folder containing the files to be renamed.
   - Check if the provided path is valid and points to an existing folder.
2. **File Listing**:
   - List all files in the folder (excluding subdirectories) to determine which files need to be renamed.
3. **Renaming Files**:
   - Rename the files in the folder by assigning them new names in the format `1.ext`, `2.ext`, `3.ext`, and so on.
   - Ensure that the program preserves the original file extensions.
4. **Error Handling**:
   - Handle invalid folder paths.
   - Skip subdirectories or hidden files (e.g., `.DS_Store` on macOS).
   - If renaming a file fails (e.g., due to permission issues), provide an error message for that file.
5. **Testing**:
   - Test the program with various folder contents, including multiple file types and a mix of files and subdirectories.

Example:

Given a folder with the following files:

```
my_folder/
├── image1.jpg
├── document.txt
├── report.pdf
├── notes.docx
├── music.mp3
```

After running the program, the files should be renamed as:

```
my_folder/
├── 1.jpg
├── 2.txt
├── 3.pdf
├── 4.docx
├── 5.mp3
```

Input Example:
Please enter the path to the folder: /path/to/folder

Output Example:
```
Renaming files...
File 'image1.jpg' renamed to '1.jpg'
File 'document.txt' renamed to '2.txt'
File 'report.pdf' renamed to '3.pdf'
File 'notes.docx' renamed to '4.docx'
File 'music.mp3' renamed to '5.mp3'
Renaming completed.
```

Error Handling Example:

- If the folder doesn't exist:

```
Error: The folder '/path/to/folder' does not exist.
```

If the folder is empty:
```
The folder is empty. No files to rename.
```

If there is a permission issue:
```
Error: Unable to rename file 'image1.jpg' due to permission issues.
```

Submission Requirements:

1. A single Python file (or a small set of files) containing your implementation of the program.
2. A README file explaining how to run the program, the expected input/output, and any assumptions you've made.

Bonus (Optional):

- Allow the user to specify whether to rename files in reverse order (starting with the highest number).
- Provide an option to preview the file names before renaming them.
- Support renaming files in nested subdirectories (recursive renaming).

Assignment 2

Programming Assignment: Zipping a Folder

Objective:
Write a program that takes a folder (directory) path as input and compresses all the files and subdirectories inside the folder into a `.zip` archive. The program should handle errors, such as invalid paths, and ensure that the resulting `.zip` file can be extracted to retrieve the original content of the folder.

Requirements:

1. **Input**: The user must provide the path to the folder they wish to zip.
2. **Output**: The program should create a `.zip` file in the same directory as the original folder, named after the folder (e.g., if the folder is `my_folder`, the zip file should be `my_folder.zip`).
3. **Error Handling**:
   - The program should check if the provided folder exists and if it's a valid directory.
   - If the provided folder is empty, the program should create an empty zip file without any errors.
   - The program should gracefully handle any exceptions or errors, such as permission issues or corrupted files.
4. **Compression Format**: The zip file should use standard ZIP compression (i.e., `.zip` format).
5. **Subdirectories**: The program should preserve the folder structure (subdirectories, files inside the subdirectories, etc.) in the resulting `.zip` archive.

Tasks:

1. **Input Parsing**:
   - The user should input the full path to the folder that they want to zip. If no path is provided, the program should ask the user to enter it.
2. **Directory Traversal**:
   - The program should traverse the specified folder and its contents, including all subdirectories and files.
   - Use recursion or a suitable traversal method to visit all files and subdirectories.
3. **Zipping**:
   - The program should create a `.zip` file with the same name as the folder and store it in the same location as the original folder.
   - Use a built-in library (e.g., `zipfile` in Python) to compress the folder and its contents.
4. **Error Handling**:

    - Ensure that the program handles invalid or non-existent folder paths.
    - Handle any other common errors that could occur during the zipping process, such as permission errors, file conflicts, etc.
5. **Testing**:
    - Ensure the program works for both small and large directories, with multiple levels of subdirectories.
    - Verify that the resulting `.zip` file can be opened and extracted correctly.

Example:

For the folder structure:

```
my_folder/
├── file1.txt
├── file2.txt
└── subfolder/
    ├── file3.txt
    └── file4.txt
```

After running your program, the output should be:
my_folder.zip

Which, when extracted, will recreate the original folder structure:

```
my_folder/
├── file1.txt
├── file2.txt
└── subfolder/
    ├── file3.txt
    └── file4.txt
```

Submission Requirements:

1. A single file (or a small set of files) containing your implementation of the program.
2. A README file explaining how to use your program, the expected input/output, and any assumptions you've made in your design.

Bonus (Optional):

- Allow the user to specify a custom name for the output `.zip` file.
- Add a feature to password-protect the `.zip` file, if desired.
- Provide a progress indicator or logging output for large directories.

Assignment 3

Programming Assignment: Creating a Collage from 4 Images

Objective:
Write a program that takes four images as input and arranges them into a 2x2 collage. The collage should be saved as a new image file. The user will specify the four images to be used, and the program will handle resizing, arranging, and saving the final collage.

Requirements:

1. **Input**: The program should allow the user to specify the paths to four image files. These images will be combined into a 2x2 grid.
2. **Output**: The program should generate a new image file with the collage arrangement, and the final image should be named `collage.jpg` (or any other image format like PNG or TIFF, based on your preference).
3. **Image Handling**:
   - Each image should be resized to fit into the grid, ensuring all four images have the same dimensions.
   - The final image should be the correct size to hold all four images in a 2x2 arrangement.
4. **Error Handling**:
   - The program should check if the provided files are valid images.
   - Handle any errors related to file loading, such as invalid paths or unsupported formats.
5. **Image Arrangement**:
   - The four input images should be arranged in a 2x2 grid:

     | Image 1 | Image 2 |

     | Image 3 | Image 4 |

6. **Saving the Collage**:
   - After the collage is created, it should be saved as a single image file (e.g., `collage.jpg`).
   - The program should be able to save the collage in different formats (JPEG, PNG, etc.) depending on the user's preference.

Tasks:

1. **Input Parsing**:
   - The user should input the paths to four images they want to combine into a collage. The program should check if the input paths are valid and the files are in supported image formats.
2. **Image Loading**:
   - Load the four images from the provided file paths.
   - Verify that the images are not corrupted and are of a format that can be processed (JPEG, PNG, BMP, etc.).
3. **Resizing Images**:
   - Resize each image so that all four images have the same dimensions (width and height).
   - You may decide on a fixed size for each image, or you can resize them to the size of the smallest image.
4. **Collage Creation**:
   - Arrange the four images into a 2x2 grid (Image 1 in the top-left, Image 2 in the top-right, Image 3 in the bottom-left, and Image 4 in the bottom-right).
   - The final collage should have a consistent size based on the images' dimensions.
5. **Saving the Collage**:
   - After arranging the images, save the result as a new image file, such as `collage.jpg`, `collage.png`, etc.
   - The user should be able to specify the desired output format (e.g., `.jpg`, `.png`).
6. **Error Handling**:
   - If the user provides invalid paths, non-image files, or unsupported formats, the program should provide a clear error message and ask for valid inputs.
7. **Testing**:
   - Test the program with a variety of images to ensure that the collage is created correctly.
   - Handle edge cases, such as providing images with different dimensions or invalid file types.

Example:

For the following four images:

- `image1.jpg`
- `image2.jpg`
- `image3.jpg`
- `image4.jpg`

The program should create a collage with the images arranged like this:

| Image 1 | Image 2 |

| Image 3 | Image 4 |

Input:

Please enter the path for Image 1: image1.jpg

Please enter the path for Image 2: image2.jpg

Please enter the path for Image 3: image3.jpg

Please enter the path for Image 4: image4.jpg

Please specify the output file format (jpg, png): jpg

Output:

A file named `collage.jpg` containing the 2x2 grid of images.

Submission Requirements:

1. A single file (or set of files) containing your implementation of the program.
2. A README file explaining how to use your program, including instructions for input and output formats.