Veeam Technical Task

By Łukasz Klępka

Contest:

1.Test Task

Please implement a program that synchronizes two folders: source and replica. The program should maintain a full, identical copy of source folder at replica folder. Solve the test task by writing a program in Python.

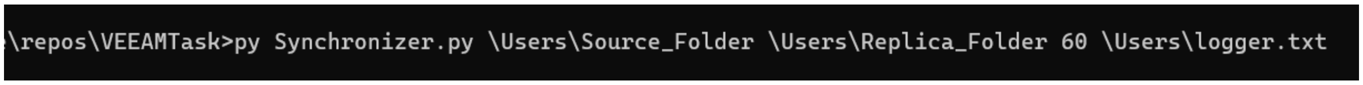
- Synchronization must be one-way: after the synchronization content of the replica folder should be modified to exactly match content of the source folder;
- Synchronization should be performed periodically;
- File creation/copying/removal operations should be logged to a file and to the console output;
- Folder paths, synchronization interval and log file path should be provided using the command line arguments;
- It is undesirable to use third-party libraries that implement folder synchronization;
- It is allowed (and recommended) to use external libraries implementing other well-known algorithms. For example, there is no point in implementing yet another function that calculates MD5 if you need it for the task – it is perfectly acceptable to use a third-party (or built-in) library.

2.Program description

Program is saved as a Synchronizer.py, to use it. To use it, you need to insert all inputs which are necessary for the proper operation of the program. These inputs are:

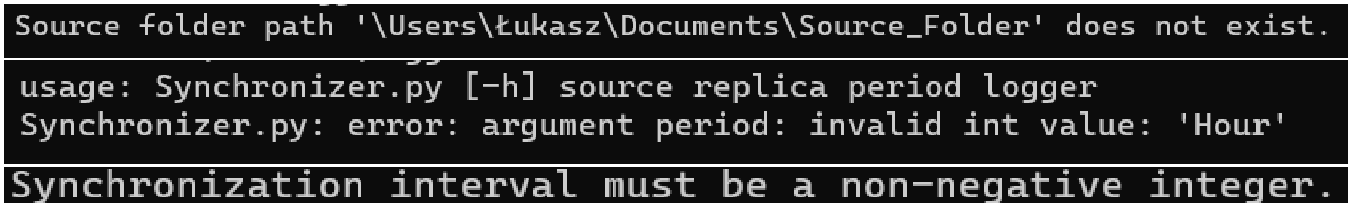
- Source folder path
- Replica folder path
- Time between synchronizations (must be integer, time in seconds)
- Logger file location

For example:



```
\repos\VEEAMTask>py Synchronizer.py \Users\Source_Folder \Users\Replica_Folder 60 \Users\logger.txt
```

If incorrect input data is provided, the program does not run and returns the reason why in case of **source folder path** and **time between synchronizations**.

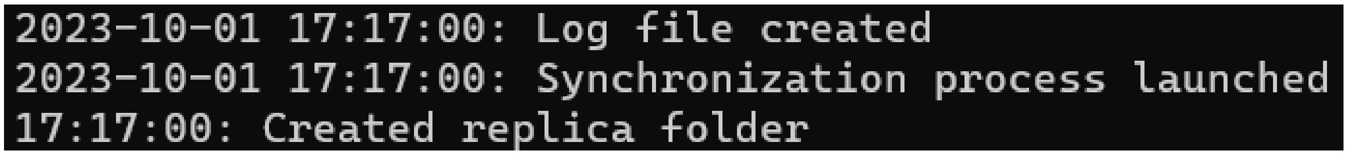


```
Source folder path '\Users\Łukasz\Documents\Source_Folder' does not exist.
```

```
usage: Synchronizer.py [-h] source replica period logger
Synchronizer.py: error: argument period: invalid int value: 'Hour'
```

```
Synchronization interval must be a non-negative integer.
```

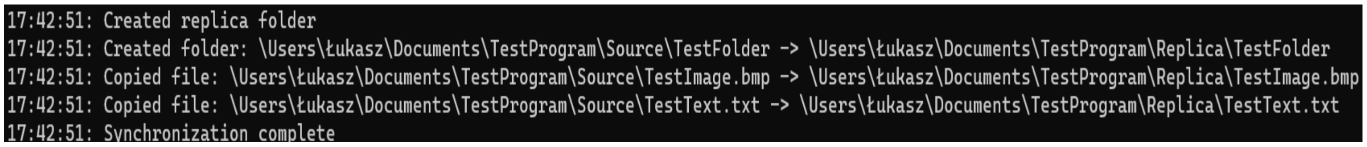
For inputs: **replica folder path**, **logger file location**. The program creates them, if they are not occurred. In the case of logger file, it can be created only one time, when program is started, replica folder can be created during each synchronization period. Information about those processes will be available in the console window and logger file.



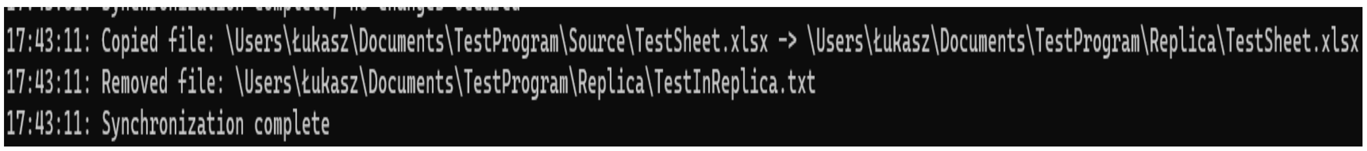
```
2023-10-01 17:17:00: Log file created
2023-10-01 17:17:00: Synchronization process launched
17:17:00: Created replica folder
```

All logger entries have the time when they are detected. For creating logger file, starting and ending synchronization process, logs have also a date. The shape of logs can be redesigned, this shape of presenting was most beneficial in my opinion.

The program synchronizes files at periods specified by the user. For each turn of the loop, changes taking place in the replica folder are written to the console and saved in the logger. When all of the changes are described, the program returns the information that synchronization was completed.

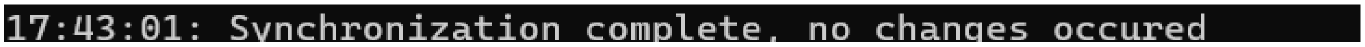


```
17:42:51: Created replica folder
17:42:51: Created folder: \Users\Łukasz\Documents\TestProgram\Source\TestFolder -> \Users\Łukasz\Documents\TestProgram\Replica\TestFolder
17:42:51: Copied file: \Users\Łukasz\Documents\TestProgram\Source\TestImage.bmp -> \Users\Łukasz\Documents\TestProgram\Replica\TestImage.bmp
17:42:51: Copied file: \Users\Łukasz\Documents\TestProgram\Source\TestText.txt -> \Users\Łukasz\Documents\TestProgram\Replica\TestText.txt
17:42:51: Synchronization complete
```



```
17:43:11: Copied file: \Users\Łukasz\Documents\TestProgram\Source\TestSheet.xlsx -> \Users\Łukasz\Documents\TestProgram\Replica\TestSheet.xlsx
17:43:11: Removed file: \Users\Łukasz\Documents\TestProgram\Replica\TestInReplica.txt
17:43:11: Synchronization complete
```

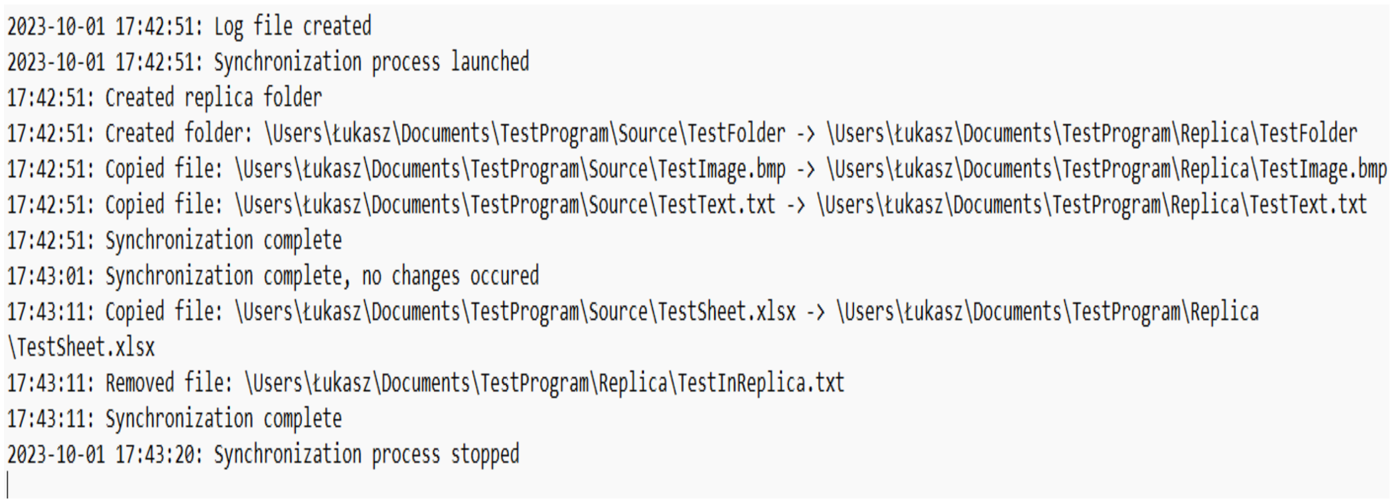
In case, when changes are not detected in turn of the loop. Program returns an information that synchronization was copmleted without changes.



```
17:43:01: Synchronization complete, no changes occured
```

Also, the program has built in an option to throw exceptions when they are detected and describe it in console and logger.

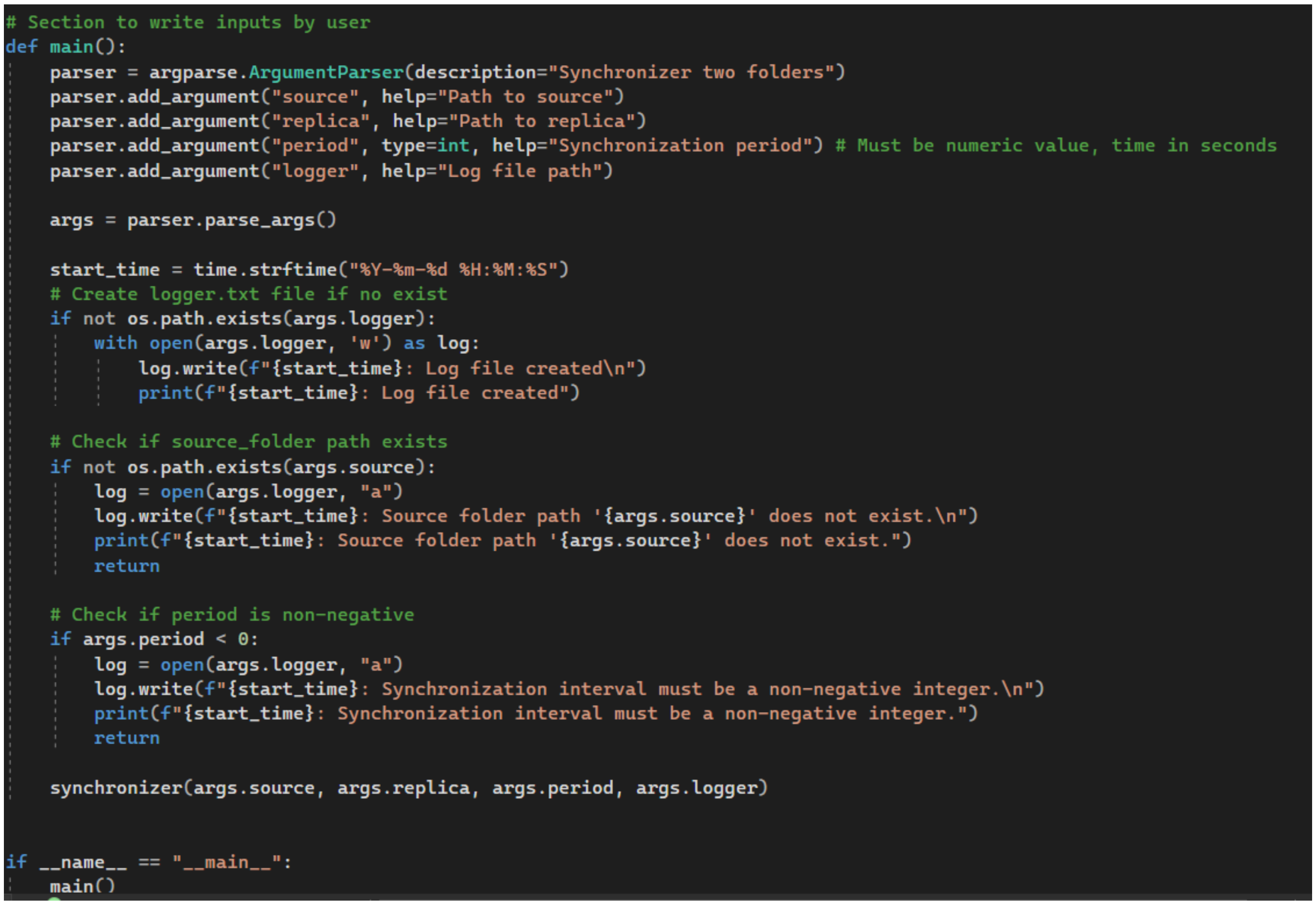
Bellow fragment of logger file:



```
2023-10-01 17:42:51: Log file created
2023-10-01 17:42:51: Synchronization process launched
17:42:51: Created replica folder
17:42:51: Created folder: \Users\Łukasz\Documents\TestProgram\Source\TestFolder -> \Users\Łukasz\Documents\TestProgram\Replica\TestFolder
17:42:51: Copied file: \Users\Łukasz\Documents\TestProgram\Source\TestImage.bmp -> \Users\Łukasz\Documents\TestProgram\Replica\TestImage.bmp
17:42:51: Copied file: \Users\Łukasz\Documents\TestProgram\Source\TestText.txt -> \Users\Łukasz\Documents\TestProgram\Replica\TestText.txt
17:42:51: Synchronization complete
17:43:01: Synchronization complete, no changes occured
17:43:11: Copied file: \Users\Łukasz\Documents\TestProgram\Source\TestSheet.xlsx -> \Users\Łukasz\Documents\TestProgram\Replica
\TestSheet.xlsx
17:43:11: Removed file: \Users\Łukasz\Documents\TestProgram\Replica\TestInReplica.txt
17:43:11: Synchronization complete
2023-10-01 17:43:20: Synchronization process stopped
```

3.Code Overview

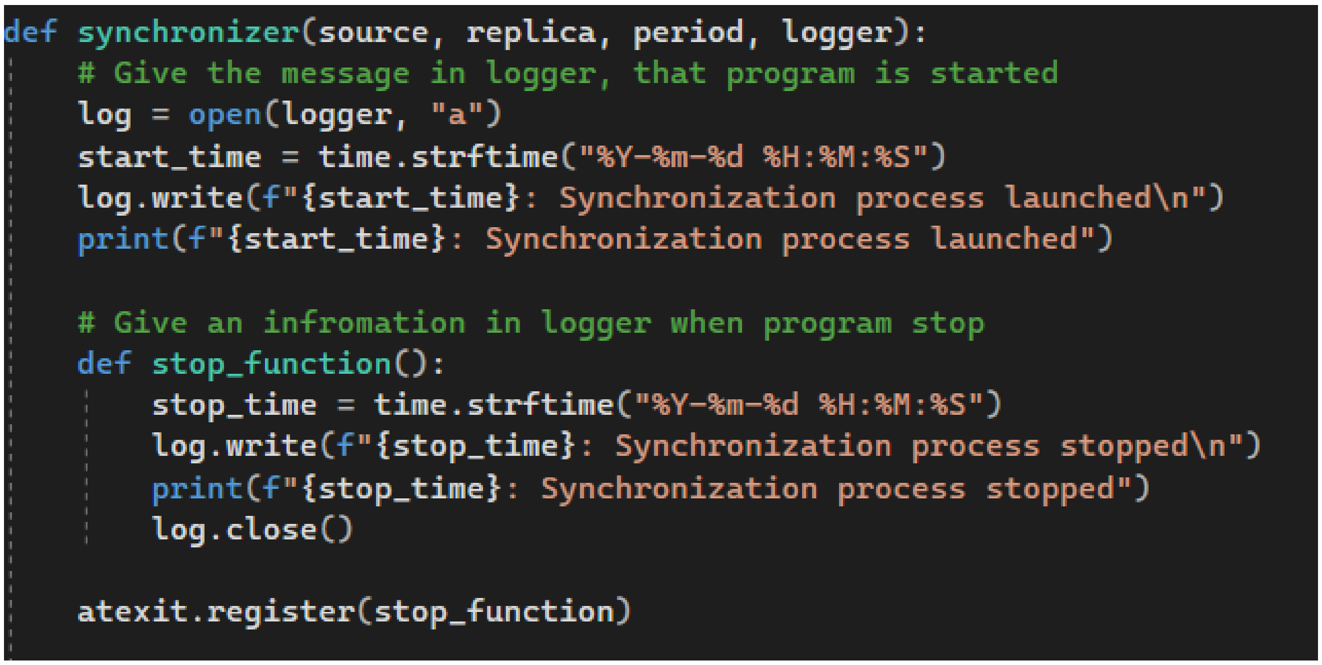
The program consists of two main parts. These are function **synchronizer**, where you can find all of logic implemented and **main** to run program and read the inputs. Firstly, I want to focus on **main** function. It starts at line 87 in **Synchronizer.py** file.



```python
# Section to write inputs by user
def main():
    parser = argparse.ArgumentParser(description="Synchronizer two folders")
    parser.add_argument("source", help="Path to source")
    parser.add_argument("replica", help="Path to replica")
    parser.add_argument("period", type=int, help="Synchronization period") # Must be numeric value, time in seconds
    parser.add_argument("logger", help="Log file path")

    args = parser.parse_args()

    start_time = time.strftime("%Y-%m-%d %H:%M:%S")
    # Create logger.txt file if no exist
    if not os.path.exists(args.logger):
        with open(args.logger, 'w') as log:
            log.write(f"{start_time}: Log file created\n")
            print(f"{start_time}: Log file created")

    # Check if source_folder path exists
    if not os.path.exists(args.source):
        log = open(args.logger, "a")
        log.write(f"{start_time}: Source folder path '{args.source}' does not exist.\n")
        print(f"{start_time}: Source folder path '{args.source}' does not exist.")
        return

    # Check if period is non-negative
    if args.period < 0:
        log = open(args.logger, "a")
        log.write(f"{start_time}: Synchronization interval must be a non-negative integer.\n")
        print(f"{start_time}: Synchronization interval must be a non-negative integer.")
        return

    synchronizer(args.source, args.replica, args.period, args.logger)


if __name__ == "__main__":
    main()
```

This function helps the user enter all the necessary inputs and invoke the synchronizer function. Really helpful is **argparse** module. Which takes care of the inserting process. When all of the inputs are inserted, the function checks critical errors, which prevents the program from being executed. Firstly, the function checks if **logger file** exists, without this file, user cannot store information about the processes. If this file does not exist, the program creates it and writes the first message. Next step is checking **source folder**, without it we cannot start process. The last option in **main** function is checking that the **period** is positive integer.

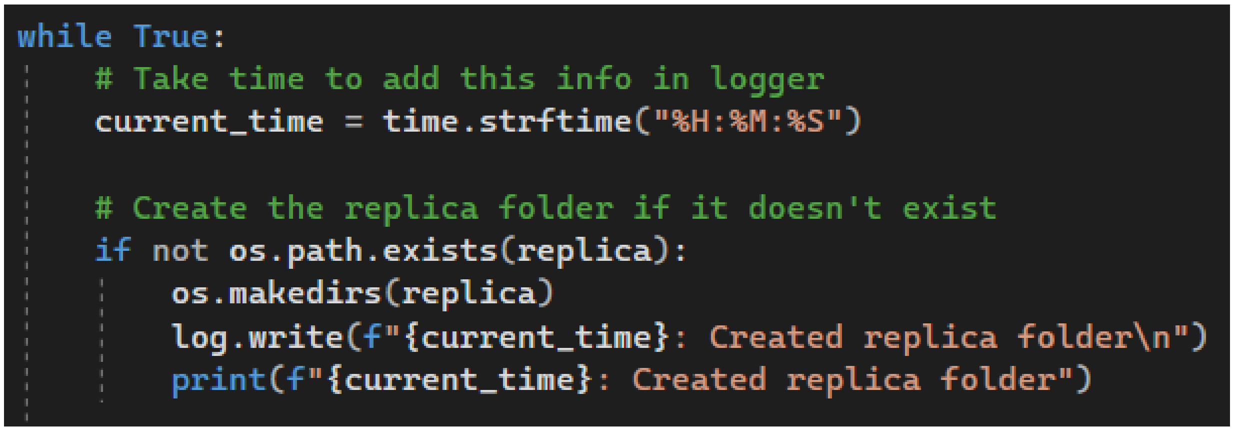
All of the program logic is implemented in **synchronizer** function. This function is separated into a few parts. At the beginning, this function returns a message about starting synchronization process in console and logger.



```python
def synchronizer(source, replica, period, logger):
    # Give the message in logger, that program is started
    log = open(logger, "a")
    start_time = time.strftime("%Y-%m-%d %H:%M:%S")
    log.write(f"{start_time}: Synchronization process launched\n")
    print(f"{start_time}: Synchronization process launched")

    # Give an infromation in logger when program stop
    def stop_function():
        stop_time = time.strftime("%Y-%m-%d %H:%M:%S")
        log.write(f"{stop_time}: Synchronization process stopped\n")
        print(f"{stop_time}: Synchronization process stopped")
        log.close()

    atexit.register(stop_function)
```

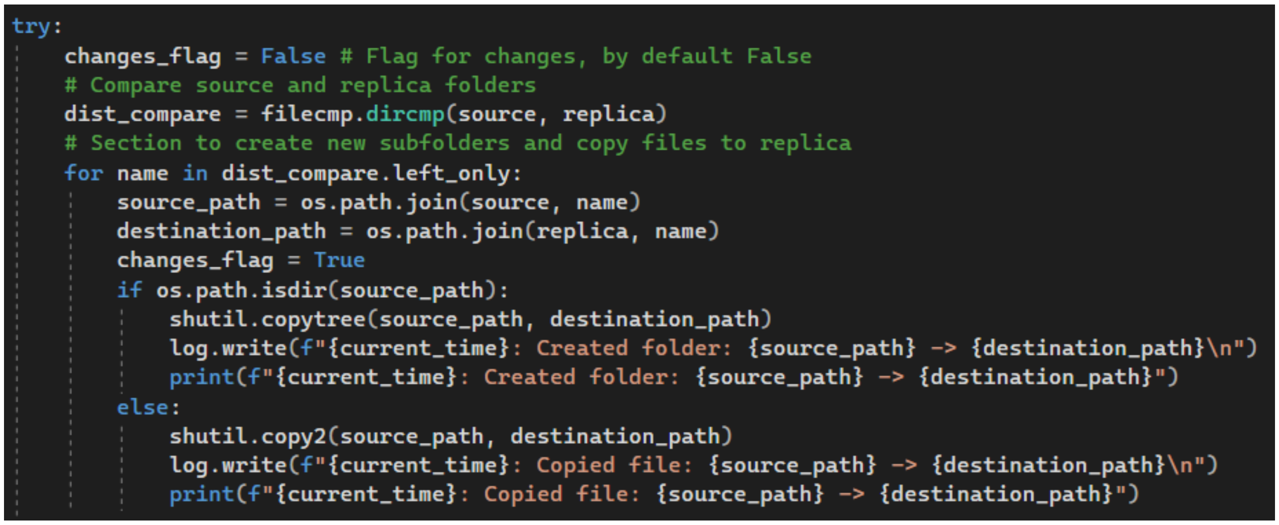
Also, in this part of code, we can find a reference to give information about the time when the process was stopped. To do this, the function uses **atexist** module. This part of code is not conducted with process but gives the user useful information.

Next part of code is taking actual **time** for every turn of the loop, as a log input, this part creates a **replica** folder in replica_path. This functionality was placed here due to the possibility of changes in replica folder during program execution.



```python
while True:
    # Take time to add this info in logger
    current_time = time.strftime("%H:%M:%S")

    # Create the replica folder if it doesn't exist
    if not os.path.exists(replica):
        os.makedirs(replica)
        log.write(f"{current_time}: Created replica folder\n")
        print(f"{current_time}: Created replica folder")
```

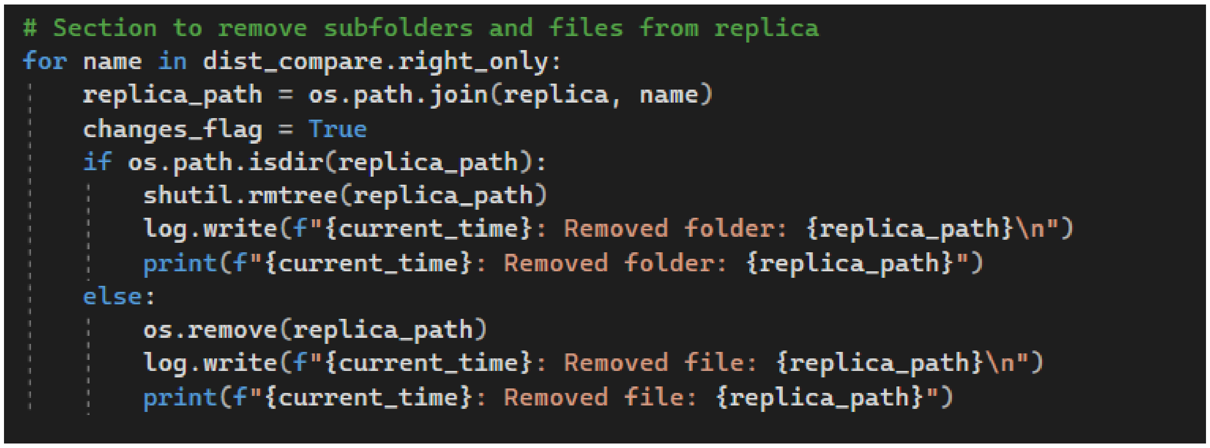
When the program checks whether the **replica** folder exists, it starts proper synchronization.



```python
try:
    changes_flag = False # Flag for changes, by default False
    # Compare source and replica folders
    dist_compare = filecmp.dircmp(source, replica)
    # Section to create new subfolders and copy files to replica
    for name in dist_compare.left_only:
        source_path = os.path.join(source, name)
        destination_path = os.path.join(replica, name)
        changes_flag = True
        if os.path.isdir(source_path):
            shutil.copytree(source_path, destination_path)
            log.write(f"{current_time}: Created folder: {source_path} -> {destination_path}\n")
            print(f"{current_time}: Created folder: {source_path} -> {destination_path}")
        else:
            shutil.copy2(source_path, destination_path)
            log.write(f"{current_time}: Copied file: {source_path} -> {destination_path}\n")
            print(f"{current_time}: Copied file: {source_path} -> {destination_path}")
```

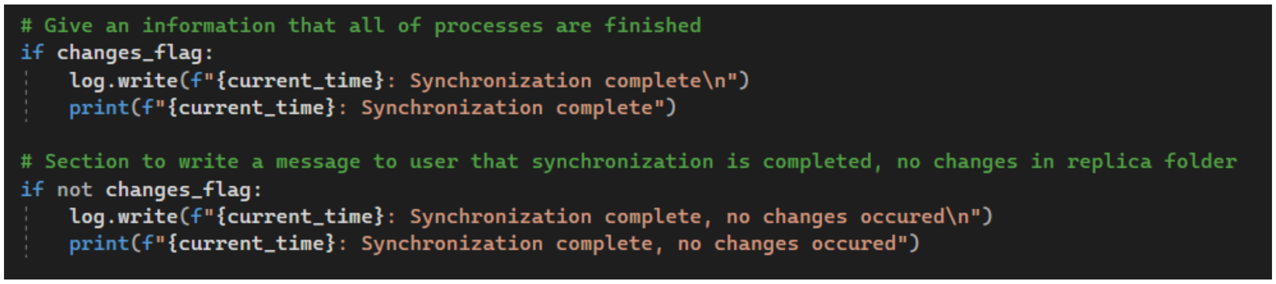
At the beginning, the program set the flag (**changes_flag**) as False, to notice that changes are occurred in this turn of the loop. After this, the program makes a comparison between **source** and **replica** folder. To do this, it uses **filecmp** module. Depending on outputs from **dist_compare**, the program uses implemented logic. If they are files which exist only in **source** folder, the program decides to create a new file in **replica** folder using **shutil.copytree** (only if the found file was folder) or copy this file by using **shutil.copy2**.

Different situation when files were detected in **replica** folder, and they are not existing in **source** folder. In this case, the program removes folders using **shutil.rmtree**, or files using **os** module.



```python
# Section to remove subfolders and files from replica
for name in dist_compare.right_only:
    replica_path = os.path.join(replica, name)
    changes_flag = True
    if os.path.isdir(replica_path):
        shutil.rmtree(replica_path)
        log.write(f"{current_time}: Removed folder: {replica_path}\n")
        print(f"{current_time}: Removed folder: {replica_path}")
    else:
        os.remove(replica_path)
        log.write(f"{current_time}: Removed file: {replica_path}\n")
        print(f"{current_time}: Removed file: {replica_path}")
```

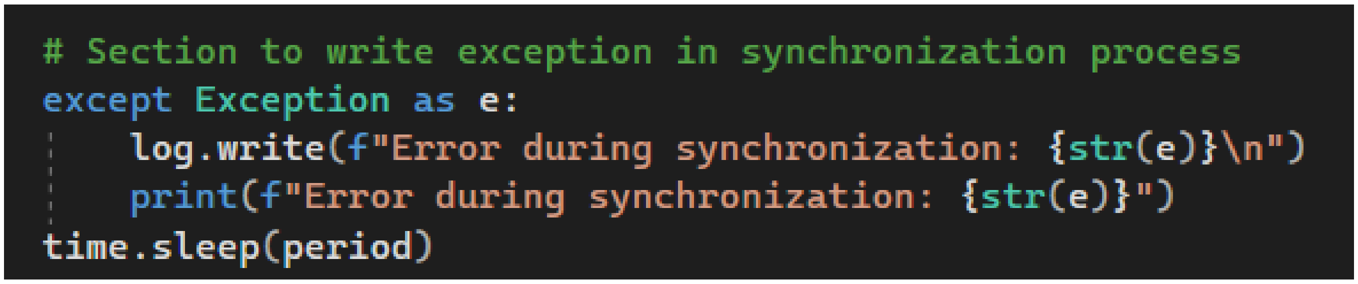
When all of the processes were finished, it was time to describe a message in console and logger that synchronization was completed.



```
# Give an information that all of processes are finished
if changes_flag:
    log.write(f"{current_time}: Synchronization complete\n")
    print(f"{current_time}: Synchronization complete")

# Section to write a message to user that synchronization is completed, no changes in replica folder
if not changes_flag:
    log.write(f"{current_time}: Synchronization complete, no changes occured\n")
    print(f"{current_time}: Synchronization complete, no changes occured")
```

Otherwise, we can have a situation that in turn of the loop no changes occurred. In that situation, the program describes the message that synchronization was completed without changes.

The last part of code is exceptions throwing. When synchronization cannot be obtained, the program describes this information with the message why.



```
# Section to write exception in synchronization process
except Exception as e:
    log.write(f"Error during synchronization: {str(e)}\n")
    print(f"Error during synchronization: {str(e)}")
time.sleep(period)
```

And of course, at the end is time function to wait the period time until next turn of loop be executed.

4.Libraries Used

- Os - operation system interface, most basic module which is necessary to run the program. Allow to make operations on the files, give an access etc.
- Filecmp – module which can compare folders. By using filecmp.dircmp it was possible to create two list containing the names of files contained in folders and further operations on them
- Shutil – allow to make operation on files and collections. This module allows the program to copy files from source to replica or create/delete folders. This module was really helpful to solve the problem with access denied to subfolders.
- Time – allow to use time-related functions. Allow take actual time to log messages or wait specified period of time before execution next turn of the loop.
- Argparse – module which can be used to write command line interface, in this case very useful. Allow to add comments to inputs and describe type for period. Add_arguments create a list of objects from command line to be used as inputs in synchronizer function. Last thing is parse_args to convert command into expected arguments.
- Atexist – this module allows to execute function when interpreter is shutting down. In case of this function, atexist call stop_function to give information when the synchronizer was stopped.

5.Problems and solutions

The main problem I encountered while writing the program was **access denied** error. This problem occurred when I tried to make an operation on a folder inside source or replica. To solve this, I used **shutil** module. At the beginning I wanted to make a solution using only **shutil.copy2** to copy files from source to replica and **os.remove** to delete it from replica folder. But this doesn't work for folders, I need to use **shutil.copytree** and **shutil.rmtree**. After implementing those into code, I was trying to leave only those two options to menage files, but that was not efficient. In final form, code has a lot of lines, but it works like I want.

The second problem I faced was to introduce logs about the end of synchronization or no changes in replica folder. To solve this problem, I decided to introduce the **changes_flag**, this variable stored the information about changes (occurred or not) and returned it at the end of the loop step. Depending on that value, if statement decide which log message return. I can also use if/else statement instead double if, but I want to make a double check solution.

The last problem I faced was deciding which inputs are really necessary. I decided to make as necessary **source** folder and **period** between synchronization. When the user defines the destination of **replica** and **logger** files, the program can create it without any problems.