Bash Automation Project

Kavya Gupta

June 14, 2023

1 Bash Script Documentation

1.1 Introduction

This documentation provides an overview of a Bash script that performs file operations, such as sorting and moving files from a source directory to a destination directory. The script supports various flags for customizing the behavior and includes functions for handling recurring filenames and printing colored messages.

1.2 Basic Requirements

The script includes two basic flags for customizing the behavior:

- **-s (style) flag:** This flag allows the user to choose the sorting style. It supports two options:
  - **ext:** When sorting by extension, files without any extension are moved to the "No_Extension" folder, while files with a specific extension are moved to the "Extension_.¡ext¿" folder.
  - **date:** When sorting by date, files are moved to folders named after their modification date, in the format "YYYY-MM-DD".
- **-d (delete) flag:** This flag enables the deletion of files from the original source directory after they have been moved to the destination directory. Use this flag with caution as the deleted files cannot be recovered.

These basic flags provide flexibility and control over the file sorting and deletion process, allowing users to customize the behavior according to their specific requirements.

1.3 Efficient Renaming System

The script incorporates an efficient renaming system that ensures unique filenames for the moved files. The renaming system follows the following rules:

- When a file is copied, it is initially started with its original name `a.py`.
- If there is a file with the same name in the destination directory, it renames the new file as `a_1.py`.
- If `a_1.py` already exists, the script appends a timestamp to the filename in the format `a_1_<timestamp>.py`. The timestamp represents the date and time of the file movement.

```
newfilepath=$desdir"/"$dir"/"$recur_name"_"$j"."$ext
s=$desdir"/"$dir"/"$recur_name"_"$j
if [ -e "$newfilepath" ]; then
  newfilepath=$desdir"/"$dir"/"$recur_name"_"$j"_"$(date +"%Y-%m-%d_%T")"."$ext
```

- If a file with the timestamp-appended name exists as well, the script continues to append the timestamp until it finds a unique filename.

```
while [[ -e "$newfilepath" && $ext == "" ]]; do
  newfilepath=$newfilepath"_"$(date +"%Y-%m-%d_%T")
done
```

- Renaming of Hidden files and No Extension Files done separately.
- "recurring_filename()" function helps in achieving this job.

This renaming system ensures that there is no overlap or duplication of filenames in the destination directory. By incorporating timestamps, it provides a systematic approach to handle recurring filenames and maintains a chronological order of file movements.

The efficient renaming system enhances the script's reliability and prevents any potential data loss or conflicts that may arise due to file name collisions.

2 Error Handling

Here I discuss the error handling mechanisms implemented in the script to ensure proper usage and prevent any potential issues. The script includes the following error checks:

- Ensuring Valid Flag : The script gives error message for flags other than -s, -e, -i, -l, -d

```
if [[ "$srcdir" == "-l" || "$srcdir" == "-d" || "$srcdir" == "-e" ||
"$srcdir" == "-s" || "$srcdir" == "-i" || "$srcdir" == "" ]]; then
print_message "${RED}" " Error: Provide a valid source directory argument."
exit 1
fi

if [[ "$desdir" == "-l" || "$desdir" == "-d" || "$desdir" == "-e" ||
```

```
"$desdir" == "-s" || "$desdir" == "-i" || "$desdir" == "" ]]; then
print_message "${RED}" " Error: Provide a valid destination directory argument."
exit 1
fi
```

- Ensuring Single Flag: The script ensures that only a single instance of each flag is entered, avoiding any errors in the program.

```
if [[ ${args[i]} == "-d" ]]; then
  if [ "$delete_files" = false ]; then
    delete_files=true
else
  print_message "${RED}" "Enter -d only 1 time. Entering more than one -d
  is wrong syntax."
  exit 1
fi
```

- Empty Arguments: The script checks for empty arguments and displays an error message if any argument is missing or empty, very important for -e, -i and -l flags.
- Directory Creation: If the destination directory is not present, the script creates it automatically to ensure smooth execution.

```
if [[ ! -d "$desdir" ]]; then
  print_message "${RED}" " Error: Destination directory '$desdir' does not exist."
  print_message "${GREEN}" "Don't worry creating one..."
  mkdir -p $desdir
fi
```

The script provides friendly error messages to guide the user in resolving any issues and ensuring a seamless experience.

3 Customization

This section highlights the customization options available in the script, particularly focused on the `-e` (exclude) and `-i` (include) flags:

- Friendly real-time input messages :

```
flag_e_display=true
if [[ $flag_e == true ]]; then #Asks the user whether he wants to see exclude output
#messages of -e flag
  echo ""
  echo "Seems you have used my -e custom flag..."
  print_message "${YELLOW}" "Do you want to see the excluded files as per the -e flag?
  (yes/no)"
```

```
  read ans
  echo ""
  if [[ $ans == "no" ]]; then flag_e_display=false; fi
fi

zip_unzip=false
print_message "${YELLOW}" "Do you want to unzip the zipped files (yes/no) ?"
read ans_zip
if [[ $ans_zip != "no" ]]; then zip_unzip=true; fi
echo ""
```

- Different colours for different lines : Using different colours for different lines, helps the user in knowing the flow of the program

```
# Define color codes
GREEN='\033[0;32m'
YELLOW='\033[0;33m'
BLUE='\033[0;34m'
RED='\033[0;31m'
NC='\033[0m' # No Color

# Function to print colored messages
print_message() {
  local color=$1
  local message=$2
  echo -e "${color}${message}${NC}"
}
```

- Handling filenames with spaces and hidden files with format .abcd or abcd. .new.sh etc.

```
if [[ $(echo $filename | grep -c '^\.') > 0 ]]; then #that means it is hidden file
  extdir="Hidden_Files"
elif [[ $(echo $filename | grep -c '\.$') > 0 ]]; then
  extdir="No_Extension"
elif [ $(echo $filename | grep -c "\.") -ne 0 ]; then # -n ensures that the string
#returned is non-empty
  extdir="Extension_.$ext" # extdir is the extension directory
else
  extdir="No_Extension"
  ext=""
fi
```

  This function handles these files efficiently.

- `-e` (exclude) Flag: This flag allows the user to exclude certain extensions from the file processing. The extensions to be excluded are provided as an argument, separated by commas. If the extension is specified as `noext`,

the script excludes files with no extension. Arguments are taken using `IFS=',' read -ra array <<< <variable>` from the very next element of the argument list.

```
elif [[ ${args[i]} == "-e" ]]; then
  if [ $flag_e = true ]; then print_message "${RED}" "Enter -e only 1 time.
  Entering more than one -e is wrong syntax."; exit 1; fi
  flag_e=true
  e_names=${args[i+1]}
  if [[ $e_names == "-d" || $e_names == "-s" || $e_names == "-e" ||
  $e_names == "-l" || $e_names == "-i" || $e_names == "" ]]; then
    print_message "${RED}" "Put valid extensions for -e flag."
    exit 1
  fi
  IFS=',' read -ra array <<< "$e_names"
  ((i=i+1))
```

No extension files are separated differently.

```
if [[ $(echo $filename | grep -c '^\.') == 0 ]]; then
  if [[ $(echo $filename | grep -c '\.$') > 0 || $(echo $filename | grep -c '\.')
  == 0 ]]; then #No Extension Files
    ext2="noext"
    ext=""
    name=$filename
    if [[ $(echo $filename | grep -c '\.$') > 0 ]]; then
      name=$(echo $filename | sed 's/\([^\.]*\)\./\1/')
    fi
  fi
fi

if [ $flag_e = true ]; then
  for element in ${array[@]}; do
    if [[ $ext == $element || $ext2 == $element ]]; then
      flag_search_e=true
    fi
  done
fi
```

- `-i` (include) Flag: This flag allows the user to include only specific extensions for processing. The extensions to be included are provided as an argument, separated by commas. If the extension is specified as `noext`, the script includes only files with no extension.

  By utilizing the `-e` and `-i` flags, users can customize the file processing behavior according to their requirements. These customization options enhance the flexibility and functionality of the script.

One of the notable features of the script is its ability to handle file renaming. It follows a sequential naming scheme to avoid any overlapping of file names. If a file with the same name already exists, the script appends a timestamp to ensure unique file naming.

- `-l` (log_file) Flag: Creates a log file with the name given by the user having the statements of all files transfer with its source and destination with the timestamp. Even the deleted files messages are shown, and file rename statements are shown.

```
if [[ "$log_file" == true ]]; then
  echo -e "${GREEN}Log file created and saved as $log_name${NC}"
  mv log.txt "$log_name"
else
  rm log.txt
fi
```

Some output messages are :-

```
    echo "$i already exists, so renamed to $newname and stored in $extdir directory"
    >> log.txt
```

- `-s size` : Sorts files by their size, uses `stat -c %s $i` to find size and arranges them in "0-10kBs", "10-20kBs", "20-30kBs" etc.

```
  elif [[ "$flag" == "size" ]]; then
    file_size_kB=$(stat -c "%s" "$i")
    floor=$(awk -v num="$file_size_kB" 'BEGIN { printf "%.0f", num }')
    if [[ $floor < 10 ]]; then extdir="0-10kBs"
    elif [[ $floor < 20 ]]; then extdir="10-20kBs"
    elif [[ $floor < 30 ]]; then extdir="20-30kBs"
    elif [[ $floor < 40 ]]; then extdir="30-40kBs"
    elif [[ $floor < 50 ]]; then extdir="40-50kBs"
    else extdir="Gt50kBs"
    fi
```

- Unzipping `.zip` files and sorting inside them : The best part of my program is that it uses a "tree" search structure that a function `makeTree()` unzips all the possible `.zip` files into a temporary folder with a suffix `@unzipped` and does this recursively till all the zip files are not unzipped, then `main` function is ran, going through the temporary folders and then at the end, `killTree()` deletes all the temporary folders, hence restoring all the source directory to normal.

```
makeTree()
{
        while  [ `find $srcdir -name "*.zip" -type f | wc -l` -gt 0 ] ;
        do
                for i in `find $srcdir -name "*.zip" -type f`;
```

```
                    do
                 name=`basename $i`
                unzip -o -q $i -d "$i@unzipped" 2> /dev/null
                 mv $i $i"%"
                done
         done
}

killPercent()
{
        for j in `find $srcdir -type f -name "*.zip%"`;
        do
                mv $j `echo $j | sed -n 's/%$//p' `
        done
}

killTree()
{
        for k in `find $srcdir -type d -name "*.zip@unzipped"`
        do
                rm -r $k 2> /dev/null
        done
}
```

- Deleting files with same hash : After the complete `main` function, the program executes a for-loop that stores the hash of all the files using `sha256sum` keyword, and other for-loop ensures that the hash of the current program is not repetitive with the previous ones.

```
#Hash Deleting Files
echo ""
echo "Do you want to calculate the hash deletion? (yes/no)"
read answer

if [[ "$answer" != "no" ]]; then
    # Perform the hash deletion calculation
    print_message "$YELLOW" "Performing hash deletion..."
    flag_hash=false
    find $desdir -type f -printf '%p\n' | while read -r i; do
      filename=$(basename $i)
      flag_hash=false
      hash=$(sha256sum "$i" | cut -d ' ' -f 1)
      for j in $(cat hash_file); do
        if [[ $hash == $j ]]; then
          flag_hash=true
          break
```

```
            fi
          done
          if [ $flag_hash = true ]; then
            rm "$i"
          else
            echo $hash >> hash_file
          fi
        done
        # Your code for hash deletion goes here
    else
        print_message "$YELLOW" "Skipping hash deletion."
    fi
```

3.1 End Display

Displays the number of files transferred, with a menu of the files in each folder created.

```
echo ""
echo -e "${BLUE}================================================================="
echo -e "                        Some Statistics      "
echo -e "================================================================${NC}"
echo -e "Folders created: $(cat temp_new_folder.txt | wc -l)"
echo -e "Files transferred : $(cat temp.txt | wc -l)"
if [[ $answer != "no" ]]; then echo -e "Files left after hash-check : $(cat hash_file
| wc -l)"; fi
if [[ "$log_file" == true ]]; then
  echo -e "${GREEN}Log file created and saved as $log_name${NC}"
  mv log.txt "$log_name"
else
  rm log.txt
fi
echo ""
print_message "$YELLOW" "Number of files in each folder:-"
for i in `sort temp.txt | uniq`; do
  printf "%-20s | %-20s\n" $i $(find $desdir"/"$i -type f | wc -l)
done
```

3.2 Amazing ASCII Art

Welcome Message and GoodBye Message :)

```
# Welcome message
echo -e "${BLUE}================================================"
echo -e "          Welcome to the Amazing Script         "
echo -e "================================================${NC}"
echo -e "${GREEN}"
```

```
echo -e "   __    _  _______  _______  _______ "
echo -e "  |  \  | ||       ||   _   ||       |"
echo -e "  |   \ | ||    ___||  |_|  ||_     _|"
echo -e "  |    \| ||   |___ |       |  |   |  "
echo -e "  |  |\   ||    ___||       |  |   |  "
echo -e "  |  | \  ||   |___ |   _   |  |   |  "
echo -e "  |__|  \_||_______||__| |__|  |___|  "
echo -e "${NC}"
```

3.3 Conclusion

In conclusion, this Bash script provides a powerful solution for sorting and moving files with customizable options. It incorporates an efficient renaming system to ensure unique filenames in the destination directory and handles recurring filenames by appending timestamps. The script's basic flags allow users to choose the sorting style and enable file deletion, giving them control over the behavior of the script. With its functionality and customization options, this script simplifies file management tasks and enhances workflow efficiency.