Bash Grader

Balaji Karedla

April, 2024

Contents

1 Objective

The bash script developed for this project aims to manage CSV files containing student data and incorporate version control functionality similar to Git. The script supports various commands, including `combine`, `upload`, `total`, `git_init`, `git_commit`, `git_checkout`, and `update`. These commands facilitate tasks such as combining multiple CSV files, updating student marks, and reverting to previous versions of the data.

2 Basic Logic of the code

The commands of this script are

2.1 `combine`

The `combine.sh` script is a bash script designed to combine multiple CSV files into a single file. Here's a step-by-step breakdown of what the script does:

1. **Help and Usage Instructions:** The script first checks if the first argument is '-h', '–help', or 'help'. If it is, it prints the usage instructions and exits. If there are any other arguments, it prints an error message and exits.
2. **Check for Main File:** The script then checks if a file named '$MAIN' exists. If it does, it checks if the first line of the file ends with ',total'. If it does, it sets a 'total' flag to 1, otherwise it sets the 'total' flag to 0. It then deletes the '$MAIN' file.
3. **Find CSV Files:** The script uses the 'find' command to find all CSV files in the current directory, excluding the '$MAIN' file. It removes the '.csv' extension and the './' prefix from the file names and combines them into a space-separated string.
4. **Check for CSV Files:** The script checks if any CSV files were found. If not, it prints an error message.
5. **Combine CSV Files:** If CSV files were found, the script calls another script named 'add.sh' in the 'combine' directory. If the 'total' flag is set to 1, it passes the '-t' option to 'add.sh', otherwise it just passes the names of the CSV files.

The `add.sh` script is a bash script that is called by the `combine.sh` script to combine CSV files. Here's a step-by-step breakdown of what the script does:

1. **Check for Total Flag:** The script first checks if the first argument is '-t'. If it is, it sets a 'total' flag to 1 and shifts the arguments, otherwise it sets the 'total' flag to 0.
2. **Process CSV Files:** The script then loops over the remaining arguments, which should be the names of the CSV files to combine. It skips any files named '$MAIN', 'temp.csv', or 'rubrics.csv'.
3. **Check for Main File:** If a file named '$MAIN' does not exist, the script creates it and adds a header line. It then appends the contents of the current CSV file, excluding the first line and converting the first field to lowercase.
4. **Combine CSV Files:** If the '$MAIN' file does exist, the script checks if the first line ends with ',total'. If it does, it calls an AWK script named 'add_total.awk' to combine the current CSV file with the '$MAIN' file. If the first line does not end with ',total', it calls an AWK script named 'add.awk' instead.
5. **Add Total Column:** If the 'total' flag is set to 1, the script calls another script named 'total.sh' to add a total column to the '$MAIN' file.

2.2 `upload`

The `upload.sh` script is a bash script designed to upload CSV files. Here's a step-by-step breakdown of what the script does:

1. **Help and Usage Instructions:** The script first checks if there are no arguments. If there are none, it prints the usage instructions and exits. If the first argument is '-h', '–help', or 'help', it also prints the usage instructions and exits.
2. **Upload Files:** If there are arguments, the script prints a message indicating that it is uploading files. It then loops over each argument, treating each one as a file to upload.
3. **Check for File Existence:** For each file, the script first checks if the file exists. If it does not, it prints an error message.
4. **Check for File Type:** If the file exists, the script checks if it is a regular file (as opposed to a directory or other type of file). If it is not a regular file, it prints an error message.
5. **Check for CSV Format:** If the file is a regular file, the script checks if its name ends with '.csv'. If it does not, it prints an error message.
6. **Upload CSV File:** If the file is a CSV file, the script copies it to the current directory, converts the first field of each line to lowercase (excluding the first line), and replaces the original file with the modified version. It then prints a message indicating that the file was uploaded.

2.3 `total`

The `total.sh` script is a bash script designed to add a total column to the 'main.csv' file. Here's a step-by-step breakdown of what the script does:

1. **Check for Main File:** The script first checks if a file named 'main.csv' exists. If it does not, it prints an error message, combines the CSV files using the 'combine.sh' script, and then calls itself to find the total.
2. **Add Total Column:** If the 'main.csv' file exists, the script checks if the first line ends with ',total'. If it does, it uses an AWK script to add a total column to each line, excluding the first line. The total is the sum of all fields starting from the second field.
3. **Create New Main File:** If the first line of the 'main.csv' file does not end with ',total', the script uses a different AWK script to add a total column to each line, including the first line. It then replaces the original 'main.csv' file with the new file and prints a message indicating that the files were totalled.

2.4 `git`

2.4.1 `git_init`

The `init.sh` script is a bash script designed to initialize a new git repository. Here's a step-by-step breakdown of what the script does:

1. **Check for Arguments:** The script first checks if there is exactly one argument. If not, it prints a usage message and exits.
2. **Check for Directory Existence:** If there is one argument, the script checks if a directory with the given name exists. If it does not, it checks if the parent directory exists and is a directory. If the parent directory does not exist or is a file, it prints an error message and exits.

3. **Create New Directory:** If the parent directory exists and is a directory, the script creates a new directory with the given name, as well as a 'commits' subdirectory and a '.git_log' file. It also creates a 'last_commit' subdirectory and initializes the '.git_log' file with an empty string.
4. **Clear Existing Directory:** If a directory with the given name already exists, the script removes all its contents and then creates a 'commits' subdirectory, a '.git_log' file, and a 'last_commit' subdirectory. It also initializes the '.git_log' file with an empty string.
5. **Check for .mygit Directory:** The script then checks if a '.mygit' directory or file already exists. If it does, it prints an error message and exits.
6. **Create .mygit Directory:** If a '.mygit' directory or file does not already exist, the script creates a '.mygit' directory and a 'dest', 'tracking_files', and 'HEAD' file within it.
7. **Find CSV Files:** The script then finds all CSV files in the current directory, excluding the '$MAIN' file, removes the '.csv' extension and the './' prefix from the file names, and writes them to the 'tracking_files' file.
8. **Write Destination Directory:** Finally, the script writes the name of the destination directory to the 'dest' file.

2.4.2 `git_config`

The excerpt from the `config.sh` script is a bash script designed to configure user information in a git repository. Here's a step-by-step breakdown of what the script does:

1. **Check for Arguments:** The script first checks if there are exactly two arguments. If not, it prints a usage message and exits.
2. **Check for Git Repository:** The script then checks if the current directory is a git repository by looking for a '.mygit' directory. If not, it prints an error message and exits.
3. **Check for Destination:** The script checks if a destination has been set by looking for a '.mygit/dest' file. If not, it prints an error message and exits.
4. **Check for Valid Argument:** The script checks if the first argument is either "user.name" or "user.email". If not, it prints an error message and exits.
5. **Set User Name:** If the first argument is "user.name", the script writes the second argument to the '.mygit/user_name' file and prints a success message.
6. **Set User Email:** If the first argument is "user.email", the script writes the second argument to the '.mygit/user_email' file and prints a success message.

2.4.3 `git_commit`

The excerpt from the `commit.sh` script is a bash script designed to commit changes to a git repository. Here's a step-by-step breakdown of what the script does:

1. **Check for User Email:** The script first checks if a user email has been set. If not, it prints an error message and exits.
2. **Commit Message:** If the first argument is '-m', the script sets the commit message to the second argument. It also generates a timestamp and a random hash for the commit.
3. **Update Git Log:** The script then checks if the '.git_log' file contains any commits. If it does not, it adds a new line with the commit information. If it does, it inserts a new line with the commit information at the beginning of the file.

4. **Find Changed Files:** The script then finds all CSV files in the current directory that are different from the last commit. It adds these files to an array.
5. **Update HEAD:** The script then updates the 'HEAD' file with the hash of the new commit.
6. **Create Commit Directory:** If the number of commits is a multiple of a certain value (MOD), the script creates a new directory for the commit and copies all CSV files to it. If not, it creates a new directory for the commit and copies only the changed files to it. For files that exist in the last commit, it generates a patch file instead of copying the file.

2.4.4 `git_checkout`

The excerpt from the `checkout.sh` script is a bash script designed to checkout a specific commit in a git repository. Here's a step-by-step breakdown of what the script does:

1. **Check for Arguments:** The script first checks if there are any arguments. If not, it prints a usage message and exits.
2. **Check for Git Repository:** The script then checks if the current directory is a git repository. If not, it prints an error message and exits.
3. **Check for Destination:** The script checks if a destination has been set. If not, it prints an error message and exits.
4. **Check for Commits:** The script checks if there are any commits. If not, it prints an error message and exits.
5. **Get Commit Hash:** If the first argument is "LAST", the script gets the hash of the last commit. Otherwise, it uses the first argument as the commit hash.
6. **Check for Commit:** The script checks if the commit exists. If not, it prints an error message and exits.
7. **Check for Ambiguous Commit Hash:** The script checks if the commit hash is ambiguous. If it is, it prints an error message and exits.
8. **Find Changed Files:** The script then finds all CSV files in the current directory that are different from the last commit. It adds these files to an array.

2.4.5 `git_log`

The excerpt from the `log.sh` script is a bash script designed to display the commit log of a git repository. Here's a step-by-step breakdown of what the script does:

1. **Check for Git Repository:** The script first checks if the current directory is a git repository by looking for a '.mygit' directory. If not, it prints an error message and exits.
2. **Check for Destination:** The script checks if a destination has been set by looking for a '.mygit/dest' file. If not, it prints an error message and exits.
3. **Check for Help Argument:** The script checks if the first argument is '-h', '–help', or 'help'. If it is, it prints the usage message and exits.
4. **Check for Oneline Argument:** The script checks if the first argument is '–oneline'. If it is, it reads each line from the '.git_log' file, splits the line into an array, and prints the commit hash and message. If the commit hash matches the 'HEAD', it also prints '(HEAD)'.
5. **Check for Invalid Arguments:** The script checks if there is exactly one argument. If there is, it prints the usage message and exits.
6. **Check for Commits:** The script checks if there are any commits by looking for a '.git_log' file. If not, it prints an error message and exits.

2.5 `update`

The excerpt from the `update.sh` script is a bash script designed to update a student's marks in a quiz. Here's a step-by-step breakdown of what the script does:

1. **Check for Arguments:** The script first checks if there is exactly one argument. If not, it prints a usage message and exits.
2. **Check for Quiz File:** The script then checks if a file exists for the specified quiz. If not, it prints an error message and exits.
3. **Check for Quiz in Main File:** The script checks if the quiz exists in the main file. If not, it prints an error message and exits.
4. **Find Field Number:** The script finds the field number of the quiz in the main file using the 'awk' command.
5. **Enter Student Information:** The script enters a loop where it prompts the user to enter the student's name, roll number, and marks. The roll number is converted to lowercase.
6. **Check for Valid Marks:** The script checks if the marks are a valid number. If not, it prints an error message and continues the loop.
7. **Check for Student in Main File:** The script checks if the student exists in the main file. If the student exists and the total field is present, it checks if the student exists in the quiz file.
8. **Update Quiz File:** If the student does not exist in the quiz file, the script appends the student's information to the file. If the student does exist, it updates the student's marks.
9. **Update Main File:** The script then updates the student's marks in the main file using the 'awk' command.

3 Utilities

- **bash** - For running shell commands.
- **sed** - For stream editing.
- **awk** - For pattern scanning and processing.
- **grep** - For searching text.
- **python3** - For running python scripts.

The following python libraries are used:

- **argparse** - For parsing command line arguments.
- **os** - For interacting with the operating system.
- **subprocess** - For running shell commands.
- **sys** - For system specific parameters.

4 Customizations

The customizations made to the script are as follows:

4.1 `train`

The excerpt from the `train.sh` script is a bash script designed to animate a train moving across the terminal. Here's a step-by-step breakdown of what the script does:

1. **Read Frames:** The script first reads lines from a file named "train.txt" and stores them in an array. Each line represents a frame of the animation.
2. **Calculate Terminal Width:** The script then calculates the width of the terminal using the 'tput cols' command.
3. **Animate Train:** The script defines a function to animate the train. This function loops over a range from the terminal width to the negative length of the train. For each iteration, it clears the screen and prints each frame of the train, adjusting the position and length of the frame based on the current iteration.
4. **Run Animation:** Finally, the script runs the animation function.

4.2 Command Autocomplete

The excerpt from the `completion.sh` script is a bash script designed to provide command-line auto-completion for a script named 'submission.sh'. Here's a step-by-step breakdown of what the script does:

1. **Define Variables:** The script first defines several local variables, including 'cur' and 'prev' to hold the current and previous words on the command line, and 'opts' to hold the list of possible completions.
2. **Check Previous Word:** The script then checks the previous word on the command line. If the previous word is "bash", it suggests "submission.sh" as a completion. If the previous word is "submission.sh" or "./submission.sh", it suggests a list of sub-commands as completions.
3. **Check for Quiz Argument:** If the previous word is "-q", "–quiz", or "update", the script suggests the names of CSV files in the current directory as completions, excluding certain files.
4. **Check for Stats Argument:** If the previous word is "stats", the script suggests "-b", "-q", and "-w" as completions.
5. **Check for Git Config Argument:** If the previous word is "git_config", the script suggests "user.name" and "user.email" as completions.
6. **Check for Git Commit Argument:** If the previous word is "git_commit", the script suggests "-m" as a completion.
7. **Check for Git Checkout Argument:** If the previous word is "git_checkout", the script suggests the commit hashes from the '.git_log' file as completions, if the file exists.
8. **Default Completion:** If none of the above conditions are met, the script suggests the contents of the 'opts' variable as completions.

4.3 Statistics

The `statistics.py` script is a Python script designed to analyze the marks of students. Here's a step-by-step breakdown of what the script does:

1. **Import Libraries:** The script first imports necessary libraries, including 'sys', 'numpy', 'pandas', 'os', 'matplotlib.pyplot', and 'argparse'.
2. **Argument Parsing:** The script sets up an argument parser to handle command-line arguments. It accepts three optional arguments: '-b' or '–bins' to specify the number of bins for the histogram, '-q' or '–quiz' to specify the name of a specific quiz, and '-w' or '–webpage' to generate a webpage with the statistics.
3. **Data Loading:** The script checks if the 'main.csv' file exists. If not, it prints a message and exits. If the file does exist, it reads the file into a pandas DataFrame. It replaces any 'a' values in the DataFrame with 0. If a 'total' column does not exist, it calculates the total marks for each student and adds it as a new column.
4. **Histogram Generation:** If the '–webpage' argument is provided, the script opens the 'statistics-template.html' file and reads its contents. It then generates a histogram of the total marks. If the '–bins' argument is provided, it uses the specified number of bins; otherwise, it automatically determines a good number of bins. It saves the histogram as a PNG file.
5. **Scatter Plot Generation:** The script generates a scatter plot of the sorted total marks. It saves the scatter plot as a PNG file.
6. **HTML Generation:** The script replaces placeholders in the HTML template with various statistics and the file names of the histogram and scatter plot. It writes the modified HTML to a new file and opens the file in the default web browser.
7. **Quiz-Specific Statistics:** If the '–quiz' argument is provided, the script generates similar statistics and visualizations for the marks of the specified quiz.

4.4 `absolute_grade`

The `absolute_grade.py` script is a Python script designed to analyze the marks of students and assign grades based on a rubric. Here's a step-by-step breakdown of what the script does:

1. **Import Libraries:** The script first imports necessary libraries, including 'sys', 'os', 'numpy', 'pandas', 'matplotlib.pyplot', and 'argparse'.
2. **Argument Parsing:** The script sets up an argument parser to handle command-line arguments. It accepts one optional argument: '-e' or '–edit' to edit the rubrics.
3. **Rubric Editing:** If the '–edit' argument is provided, the script prompts the user to include each grade in the rubrics and to enter the corresponding marks. The rubrics are then written to the 'rubrics.csv' file.
4. **Data Loading:** If the 'rubrics.csv' file exists and the '–edit' argument is not provided, the script reads the 'main.csv' file into a pandas DataFrame. It replaces any 'a' values in the DataFrame with 0.
5. **Total Marks Calculation:** If a 'total' column does not exist in the DataFrame, the script calculates the total marks for each student by summing the marks in each row (ignoring the first two columns) and adds it as a new column.

6. **Rubric Data Loading:** The script reads the ‘rubrics.csv‘ file into another DataFrame and reverses the order of the rows.
7. **Grade Assignment:** The script creates a list of grade labels from the ’grades’ column of the rubric DataFrame and inserts ’FR’ at the beginning of the list. It also creates a list of bin edges for the ‘pd.cut()‘ function, which includes negative infinity, the marks from the rubric DataFrame, and positive infinity. The ‘pd.cut()‘ function is then used to assign a grade to each total mark in the main DataFrame. The resulting grade is stored in a new column named ’grade’.
8. **Data Saving:** The script writes the main DataFrame, which now includes the ’grade’ column, to a new CSV file named ’graded.csv’.
9. **Grade Counts:** The script counts the number of occurrences of each grade in the ’grade’ column of the main DataFrame and prints the counts.

4.5 diff-patch

The version control system implemented in the script uses the ‘diff‘ and ‘patch‘ commands to generate and apply patches. Here’s a step-by-step breakdown of how the ‘diff-patch‘ system works:

1. **File Saving** The VCS takes a snap of all the csv files in the directory every 10 commits and stores them in the remote repository.
2. **Patch Generation** The VCS generates a patch file for each file that existed in the last commit but has been modified in the current commit. If a file is newly added, the entire file is included in the patch. The patch files are named as ‘file.patch‘, and the files added are not renamed.
3. **Patch Application** When checking out a commit, the VCS checks for the closest MOD 10 commit and applies the patches to reach the desired commit.

4.6 `student`

The `student` command is a custom command that can be used to view the marks of a specific student. Here’s a step-by-step breakdown of what the command does:

1. **Argument Parsing:** The script sets up an argument parser to handle command-line arguments. It accepts one required argument: either the roll number or the name of the student.
2. **Data Loading:** The script reads the ‘main.csv‘ file into a pandas DataFrame. It replaces any ’a’ values in the DataFrame with 0.
3. **Student Information:** The script checks if the argument is a roll number or a name. If it is a roll number, it converts it to lowercase. It then searches for the student in the DataFrame and prints their marks.
4. **Error Handling:** If the student is not found, the script prints an error message.
5. **Grade Assignment:** The script assigns a grade to the student based on the rubric and prints the grade.
6. **Plot Generation:** The script generates a bar plot of the student’s marks and saves it as a PNG file.

7. **Webpage Generation:** The script generates an HTML file with the student's information and the bar plot. It opens the file in the default web browser.
8. **PDF Generation:** The script generates a student's grade card in PDF format by editing the tex file and compiling it.

5 Usage of the commands

The following commands can be used to run the scripts:

- The whole project only runs in the same directory as the script.
- The autocompletion of the script needs the command `source completion.sh` to be run in the terminal.
- The `train` command can be run using the command `bash submission.sh train`. This was a fun addition to the script. The train moves across the terminal which looks the `sl` command.
- Add the files to the working directory using the command `bash submission.sh upload <files>`
- Combine the files using the command `bash submission.sh combine`.
- Update the marks of the students using the command `bash submission.sh update <quiz>`.
- Total the marks of the students using the command `bash submission.sh total`. Make sure to run the `combine` command before running this command.
- Initialize the git repository using the command `bash submission.sh git_init <directory>`.
- Configure the user information using the command `bash submission.sh git_config <user.name/user.email> <value>`.
- Commit the changes using the command `bash submission.sh git_commit -m <message>`.
- Checkout a specific commit using the command `bash submission.sh git_checkout <commit>`.
- View the commit log using the command `bash submission.sh git_log`.
- View the statistics of the students using the command `python3 statistics.py`. It has flags like `-b`, `-q`, and `-w`. The `-w` flag generates a webpage with the statistics. The `-q` flag generates statistics for a specific quiz with the quiz name as the argument. The `-b` flag specifies the number of bins for the histogram.
- Assign grades to the students using the command `python3 absolute_grade.py`. It has a flag `-e` to edit the rubrics. The rubrics are stored in the `rubrics.csv` file. The grades are assigned based on the rubrics. The grades are stored in the `graded.csv` file.
- View the marks of a specific student using the command `bash submission.sh student <roll number/name>`. It generates a webpage with the student's information and a bar plot of the student's marks. It has flags like `-r`, `-n`, `-w` and `-p`. The `-r` flag specifies the roll number of the student. The `-n` flag specifies the name of the student. The `-w` flag generates a webpage with the student's information and the bar plot. The `-p` flag generates a PDF of the student's grade card.