Tutorial: Removing Duplicate Lines in a File with a Bash Script

In this tutorial, I'll walk you through a simple yet powerful bash script that checks for duplicate lines in a file, removes them, and returns a clean, sorted version of the file. This is useful when you're working with lists, logs, or any other files where duplicates need to be eliminated for clarity or processing efficiency.

Let's break it down, so you can understand the why behind each part of this script.

Step 1: Check if the Input File is Provided

The first thing we need to do is check whether the script is receiving an argument (i.e., the file name) when it's run:

```
if [ -z "$1" ]; then
    echo "Usage: $0 <input_file>"
    exit 1
fi
```

- `"$1"`: Refers to the first argument passed to the script (the file name).
- `-z "$1"`: Checks if the argument is empty (i.e., no input file was provided).
- `exit 1`: If no file is provided, we exit the script with an error code `1`, indicating that something went wrong.

Why check the input?

You always want to validate input when writing scripts. This makes the script foolproof, ensuring that it won't run if the required file isn't passed in. The user gets clear instructions about how to use the script (`Usage: $0 <input_file>`).

Step 2: Verify if the Input File Exists

Next, we check whether the provided file exists:

```
input_file="$1"

if [ ! -e "$input_file" ]; then
    echo "Error: File '$input_file' not found."
    exit 1
fi
```

- `"$input_file"`: Stores the file name passed as an argument.
- `-e "$input_file"`: Checks if the file exists in the directory. If not, the script displays an error and exits.

Why this check?

It’s common practice to ensure the file you’re working with actually exists. If the user inputs the wrong file name or the file has been moved, this check prevents the script from proceeding with an invalid or non-existent file, saving you from errors down the road.

Step 3: Create a Temporary File

We use a temporary file to store the sorted and duplicate-free content. This avoids accidentally modifying the original file before everything is confirmed.

```
temp_file=$(mktemp)
```

- `mktemp`: Creates a unique, temporary file name and stores it in `temp_file`.

Why use a temporary file?

Modifying the original file directly can be risky. If anything goes wrong during the operation, you could lose data. By working with a temporary file, we ensure we don’t touch the original until we’re confident the operation is successful.

Step 4: Sort and Remove Duplicate Lines

Now we perform the actual sorting and duplicate removal:

```
sort -u "$input_file" > "$temp_file"
```

- `sort -u`: Sorts the file (`-u` stands for unique, meaning it removes duplicate lines while sorting).
- `"$input_file"`: The file you passed as input.
- `> "$temp_file"`: Redirects the output (the sorted and unique lines) to the temporary file.

Why sort and remove duplicates?

Sorting helps you keep the file organized, and removing duplicates is crucial for situations where redundancy is unwanted. Sorting by itself doesn’t remove duplicates, but the `-u` flag ensures that only unique lines remain, which is exactly what we need.

Step 5: Overwrite the Original File

Once the duplicates have been removed and the file is sorted, we overwrite the original file with the contents of the temporary file:

```
mv "$temp_file" "$input_file"
```

- `mv "$temp_file" "$input_file"`: This moves the temporary file back to the original file location, effectively replacing it with the cleaned-up content.

Why overwrite the original?

We've processed the file and removed the duplicates. Now, we want to save the changes, so we replace the original file with the sorted, duplicate-free version. Using `mv` is efficient because it renames the temporary file to the original file name in a single step.

Step 6: Provide User Feedback

Finally, we let the user know that the operation was successful:

```
echo "Duplicate lines removed from $input_file."
```

Why include feedback?

It's always a good idea to provide feedback in scripts, especially for users who may not understand what's happening behind the scenes. This simple message lets them know the script has completed its task successfully.

Final Thoughts

This script does exactly what it's supposed to: remove duplicate lines from a file while keeping things safe with temporary files. It's a great example of using basic Linux utilities (`sort`, `mv`, `mktemp`) to solve real-world problems efficiently.

Here's the full script again for reference:

```
#!/bin/bash

# Check if the input file is provided
if [ -z "$1" ]; then
    echo "Usage: $0 <input_file>"
    exit 1
fi

input_file="$1"

# Check if the input file exists
if [ ! -e "$input_file" ]; then
    echo "Error: File '$input_file' not found."
    exit 1
fi

# Create a temporary file for the sorted and unique lines
temp_file=$(mktemp)

# Sort the input file and remove duplicate lines
sort -u "$input_file" > "$temp_file"
```

```
# Overwrite the original file with the sorted and unique lines
mv "$temp_file" "$input_file"

echo "Duplicate lines removed from $input_file."
```

With this script in your toolkit, you can automate the process of cleaning up text files, saving time and reducing errors when managing lists or datasets.

That’s it! This is an essential script for anyone working with large text files or needing to regularly clean up duplicate lines.