**Translating Uppercase and Lowercase with tr**

**Explanation of Commands and Options:**

1. **tr**: This is used for **translating** or **deleting characters** in a stream. Here are some key options:
   * **tr "[a-z]" "[A-Z]"**: Converts lowercase letters to uppercase.
   * **tr -d 'a'**: Deletes all occurrences of the character a.
   * **tr -s ' '**: Squeezes consecutive spaces into a single space.
2. **cut**: This command is used to **extract specific portions** of text from each line of a file:
   * **cut -c 1-5**: Extracts the characters from position 1 to 5 of each line.
   * **cut -f**: Can also be used to cut out specific fields (for example, CSVs or tab-delimited data).

| **Command** | **Purpose** |
| --- | --- |
| **tr "[a-z]" "[A-Z]"** | Converts lowercase letters to uppercase |
| **tr "[A-Z]" "[a-z]"** | Converts uppercase letters to lowercase |
| **tr -d 'a'** | Removes the character a from the text |
| **tr -s ' '** | Squeezes consecutive spaces into a single space |
| **cut -c 1-5** | Extracts the first 5 characters of each line |

$ cat student | cut -c 1,3,5

* **cut -c 1,3,5**: This extracts characters from **positions 1, 3, and 5** of each line in the file.
* If the student file contains:

This extracts just the **1st, 3rd, and 5th characters** from each line.

**2. Cutting Data by Delimiter with cut -d**

The **cut -d** option allows you to specify a delimiter (such as a comma, semicolon, or colon) to separate fields. You can then extract specific fields based on that delimiter.

$ cat student | cut -d':' -f1

* **cut -d':' -f1**: This splits the line by the colon : delimiter and extracts **field 1**.
* If the student file contains:

$ cat student | cut -d';' -f2

* **cut -d';' -f2**: This splits the line by the semicolon ; delimiter and extracts **field 2**.
* If the file student contains:

**3. Cutting Multiple Fields with cut -d**

You can also select **multiple fields** by specifying a comma-separated list after the -f flag.

$ cat student | cut -d';' -f1,3

* **cut -d';' -f1,3**: This splits the line by the semicolon ; delimiter and extracts **fields 1 and 3**.
* If the file student contains:

**Key Options of cut:**

| **Command** | **Description** |
| --- | --- |
| **cut -c 1,3,5** | Extracts characters from positions 1, 3, and 5 of each line. |
| **cut -d':' -f1** | Splits data by : and extracts **field 1**. |
| **cut -d';' -f2** | Splits data by ; and extracts **field 2**. |
| **cut -d';' -f1,3** | Splits data by ; and extracts **fields 1 and 3**. |

**Head & Tail Commands in Linux**

The **head** and **tail** commands are commonly used to display the beginning or end of a file. Let's break them down based on your examples.

**1. head Command:**

The head command displays the **first few lines** of a file by default. You can specify how many lines you want to display using the -n option.

$ head -3 student

* **head -3**: This shows the first 3 lines of the student file.

**Default Behavior:**

* If you use **head** without specifying a number, it will display the first 10 lines by default.

**2. tail Command:**

The **tail** command displays the **last few lines** of a file by default. Like head, you can use the -n option to specify the number of lines you want to view.

$ tail -3 student

* **tail -3**: This shows the last 3 lines of the student file.

$ tail +3 student

* **tail +3**: This will display the content starting from line 3 onward (it shows everything from line 3 to the end of the file).

**3. Combining tail and head:**

You can combine multiple commands with pipes to filter or extract specific data. For instance:

$ tail +3 student | head -1 | cut -d';' -f1

* **tail +3 student**: Starts displaying from line 3 of the student file.
* **head -1**: Only shows the first line from the tail output.
* **cut -d';' -f1**: Cuts the first field (separated by ;) of that line.

This will extract the first part of the 3rd line onward from the file, depending on your file's contents.

**Saving Extracted Data:**

You can redirect the output of commands to create new files and save the extracted data.

$ tail +3 student | grep '[A-Z]' > newstudent

* **tail +3 student**: Starts displaying from line 3 onward.
* **grep '[A-Z]'**: Filters lines containing uppercase letters.
* **> newstudent**: Saves the filtered data into a new file called newstudent.

This command will create a new file newstudent and store the filtered data in it.

**4. File Comparison:**

**comm Command:**

The comm command compares **two sorted files** line by line and gives a 3-column output:

* **Column 1**: Lines only in the first file.
* **Column 2**: Lines only in the second file.
* **Column 3**: Lines common to both files.

$ comm student newstudent

* **comm student newstudent**: Compares the student and newstudent files and outputs the differences and similarities between them.

**cmp Command:**

The cmp command compares two files byte by byte. If the files are identical, it produces no output. If there is a difference, it reports the first byte where the files differ.

$ cmp student newstudent

* **cmp student newstudent**: Compares the student and newstudent files byte by byte.

If the files are the same, there will be no output. If they differ, the command will display the first difference.

**Summary of Key Commands:**

| **Command** | **Description** |
| --- | --- |
| **head -3 student** | Displays the first 3 lines of the student file. |
| **tail -3 student** | Displays the last 3 lines of the student file. |
| **tail +3 student** | Displays lines starting from line 3 onward. |
| \*\*`tail +3 student | head -1`\*\* |
| **comm student newstudent** | Compares two files and shows their differences and similarities. |
| **cmp student newstudent** | Compares two files byte by byte. If no differences, it gives no output. |

**diff Command:**

The **diff** command is used to compare two files line by line. It displays the differences between the files.

$ diff student studentaply

* **diff** compares the student file with the studentaply file.
* This output shows that the first line differs in case (capitalization).

**2. uniq Command:**

The **uniq** command is used to filter out repeated lines from a sorted file. It shows only the unique lines.

$ cat student | uniq

* This will display only the **unique** lines from the student file.
* If there are duplicate lines, they will be removed, and only one instance will be shown.

$ cat student | sort | uniq -d

* **sort**: Sorts the lines.
* **uniq -d**: Displays only the lines that are repeated.

**Example 3: Count occurrences of duplicate lines:**

$ cat student | sort | uniq -c

* **-c**: Prefixes each line with the number of occurrences.

**Example 4: Show lines that are not repeated:**

$ cat student | sort | uniq -u

* **-u**: Shows only lines that are **unique** (i.e., not repeated).

**3. split Command:**

The **split** command is used to split a file into multiple smaller files based on line numbers or byte size.

$ split -l 3 student stdlist

* **-l 3**: This option splits the student file into smaller files with 3 lines each.
* **stdlist**: This is the prefix for the output files. The files will be named stdlistaa, stdlistab, etc., depending on the number of lines in the original file.
* **Output**: If the student file has 6 lines, two files (stdlistaa and stdlistab) will be created with 3 lines each.

**Summary of Commands:**

| **Command** | **Description** |
| --- | --- |
| **diff student studentaply** | Compares two files (student and studentaply) and shows their differences. |
| \*\*`cat student | uniq`\*\* |
| \*\*`cat student | sort |
| \*\*`cat student | sort |
| \*\*`cat student | sort |
| **split -l 3 student stdlist** | Splits the student file into smaller files, each containing 3 lines. |

**rm Command:**

The **rm** (remove) command is used to delete files or directories.

$ rm "data"

* **rm "data"**: This will delete the file named **data** in the current directory.
* If you want to delete a directory, use **rm -r** for recursive removal.

**2. whoami Command:**

The **whoami** command is used to display the username of the current user.

$ whoami

* **whoami**: This will display the username, in this case, **aps**.

**3. Scheduling a Task with at:**

The **at** command is used to schedule tasks to run at a specified time.

$ at 10:26 AM

* **at 10:26 AM**: This schedules a task to run at **10:26 AM**. After running the command, you will enter a new prompt where you can specify the task to run at that time.

$ echo "Hello, this is a scheduled task!" > scheduled\_task.txt

* This would run the echo command at the scheduled time and create the file scheduled\_task.txt with the message.

**4. sleep Command:**

The **sleep** command is used to pause the execution of the script or command for a specified amount of time.

$ sleep 30

* **sleep 30**: Pauses the terminal or script for 30 seconds.
* It can also accept time in minutes (m), hours (h), and days (d).

$ sleep 30s # 30 seconds

$ sleep 2m # 2 minutes

$ sleep 1h # 1 hour

$ sleep 1d # 1 day

**5. echo Command:**

The **echo** command is used to display a message or output to the terminal.

$ echo "Time over!"

* **echo "Time over!"**: This will print **"Time over!"** to the terminal.

**6. Process Management with ps and kill:**

* **ps**: The **ps** (process status) command is used to display a list of currently running processes.

$ ps

* This shows the list of processes running in the background with their process IDs (PID).
* **kill**: The **kill** command is used to terminate a process based on its process ID (PID).

$ kill 12195

* **kill 12195**: This terminates the process with PID **12195**.

**Summary of Commands:**

| **Command** | **Description** |
| --- | --- |
| **rm "data"** | Deletes the file named **data** in the current directory. |
| **whoami** | Displays the current username. |
| **at 10:26 AM** | Schedules a task to be run at 10:26 AM. After entering the task, press **Ctrl+D** to schedule it. |
| **sleep 30** | Pauses the terminal or script for 30 seconds. |
| **echo "Time over!"** | Displays **"Time over!"** in the terminal. |
| **ps** | Displays the list of currently running processes. |
| **kill 12195** | Terminates the process with PID **12195**. |

**nohup Command:**

The **nohup** command stands for "no hangup" and is used to run a command in the background, ensuring that the command keeps running even if the user logs out or the terminal is closed.

$ nohup sleep 30 && echo "closing"

* **nohup sleep 30 && echo "closing"**: This command will run the **sleep** command for 30 seconds in the background, and once it finishes, it will print **"closing"** to the terminal. The **nohup** ensures the command continues to run even if the session is closed.

**2. awk Command:**

The **awk** command is a powerful text-processing tool. It can be used for searching, manipulating, and reporting data in files. It's especially useful for column-based operations.

$ awk '{print $1, $3}' student

* **awk '{print $1, $3}' student**: This command will print the **1st** and **3rd** columns of the **student** file. In this case, the output would show the first and third elements of each line (like names and subjects/marks).

$ awk '$2 > 90 {print $1, $2, $3}' student

* **awk '$2 > 90 {print $1, $2, $3}' student**: This command filters and prints lines where the **2nd** column (marks) is greater than 90. It will display the name, marks, and subject or degree.

**3. Working with Field Separators in awk:**

In many cases, data in a file may be separated by different delimiters, such as spaces, commas, or semicolons. The **-F** option in **awk** allows you to specify a custom field separator.

$ awk -F ';' '{print $1, $2, $3}' student

* **awk -F ';' '{print $1, $2, $3}' student**: This command tells **awk** to use the semicolon **(;)** as the field separator and then print the first, second, and third columns from the **student** file.

**4. NR (Number of Records) in awk:**

The **NR** variable in **awk** keeps track of the number of records (lines) processed so far. You can use it to count the number of lines or entries.

$ awk '{print "Total students:", NR}' student

* **awk '{print "Total students:", NR}' student**: This will print the total number of students (lines in the **student** file). The **NR** variable counts each line processed.

| **Command** | **Description** |
| --- | --- |
| **nohup** | Runs a command in the background and ensures it continues even after logout. |
| **awk** | Text-processing command to manipulate or extract specific data from files. |
| **awk -F** | Specifies the field separator (e.g., semicolon **;**) in awk. |
| **NR** | A built-in **awk** variable to track the number of processed records (lines). |
| **awk '$2 > 90 {print $1, $2, $3}'** | Filters data based on conditions (e.g., marks greater than 90). |

**awk 'END {print "Total Students", NR}'**

This **awk** command is used to print the total number of lines (students) in the file at the **end** of the processing.

$ awk 'END {print "Total Students", NR}' student

* **NR** is a built-in variable in **awk** that tracks the number of records (lines) processed. The END block ensures the command runs after processing all the lines in the file.

Total Students: 6

**2. Matching a Pattern with awk**

You can use **awk** to search for specific patterns (like a student name) in a file.

$ awk '/Sandy/' student

* This command will search for any lines that contain the pattern "Sandy" in the **student** file.

Sandy 93 BE

**3. Inverse Matching with awk**

If you want to find lines that do not match a specific pattern, you can use the ! operator in **awk**.

$ awk '!/Sandy/' student

* The **!** operator before the pattern **/Sandy/** inverts the match, meaning it will print all lines that **do not** contain "Sandy."

**4. Converting Text to Uppercase with awk**

You can convert text to uppercase using **awk**'s **toupper()** function.

$ awk '{print toupper($1)}' student

* **toupper($1)** converts the first column (name) of the **student** file to uppercase.

**5. Converting Text to Lowercase with awk**

Similarly, you can convert text to lowercase using **tolower()**.

$ awk -F ';' '{print tolower($1)}' student

* **-F ';'** sets the field separator as a semicolon, and **tolower($1)** converts the first field (name) to lowercase.

**6. Text Substitution with awk**

**awk** allows you to perform text substitutions using the **sub()** or **gsub()** functions.

$ awk '{gsub("BE", "Bachelor of Engineering"); print $0}' student

* **gsub("BE", "Bachelor of Engineering")** globally replaces every instance of **BE** with **Bachelor of Engineering**.

**7. awk Example with Data Substitution:**

You can substitute a part of a string with a different value, as shown here:

$ awk '{gsub("Engine", "Engineering"); print $0}' student

* **gsub("Engine", "Engineering")** replaces "Engine" with "Engineering" in all the data.

**Summary of Key Commands:**

| **Command** | **Description** |
| --- | --- |
| **awk 'END {print "Total Students", NR}'** | Prints the total number of students (lines) at the end of file processing. |
| **awk '/Sandy/'** | Matches and prints lines containing "Sandy." |
| **awk '!/Sandy/'** | Prints lines that do not contain "Sandy." |
| **awk '{print toupper($1)}'** | Converts the first column to uppercase. |
| **awk '{print tolower($1)}'** | Converts the first column to lowercase. |
| **awk '{gsub("BE", "Bachelor of Engineering"); print $0}'** | Replaces "BE" with "Bachelor of Engineering" in each line. |
| **awk '{gsub("Engine", "Engineering"); print $0}'** | Replaces "Engine" with "Engineering" in the file data. |

**Additional Notes:**

* **gsub()**: Used for **global** substitution (replaces all occurrences).
* **sub()**: Used for **single** substitution (replaces the first occurrence).