Tab 1

# **Advanced Regular Expressions (Regex) with Detailed Explanations**

Regular expressions (regex) are used to search, match, and manipulate text. They are supported in grep, sed, awk, find, and many programming languages.

We'll cover **basic**, **intermediate**, and **advanced** regex concepts with **clear explanations and examples**.

## **1. Basic Regex Elements**

Regex patterns are made up of **literal characters** and **special characters**.

### **1️⃣ Literal Characters**

A literal character matches itself exactly.

🔹 **Example:**

grep "apple" fruits.txt

✅ Finds lines containing **"apple"** exactly (but not "Apple" or "apples").

## **2. Metacharacters and Special Characters**

These characters have special meanings in regex.

### **2️⃣ The . (Dot) - Match Any Single Character**

The dot (.) matches **any single character** except a newline.

🔹 **Example:**

grep "c.t" words.txt

✅ Matches: cat, cot, cut, c9t  
 ❌ Doesn't match: ct, cart (because . matches **exactly one** character)

### **3️⃣ The \* (Asterisk) - Match Zero or More Occurrences**

\* means the **preceding character or group** can appear **zero or more times**.

🔹 **Example:**

grep "go\*d" words.txt

✅ Matches: gd, god, good, goooood  
 ❌ Doesn't match: godd (because d must come after o)

🔹 **Example with a character set:**

grep "[0-9]\*" numbers.txt

✅ Matches lines that contain **zero or more** digits.

### **4️⃣ The + (Plus) - Match One or More Occurrences**

+ is similar to \*, but **must appear at least once**.

🔹 **Example:**

grep "go+d" words.txt

✅ Matches: god, good, goooood  
 ❌ Doesn't match: gd (because o must appear **at least once**)

### **5️⃣ The ? (Question Mark) - Match Zero or One Occurrence**

? means the preceding character is **optional** (can appear 0 or 1 times).

🔹 **Example:**

grep "colou?r" colors.txt

✅ Matches: color, colour  
 ❌ Doesn't match: colouur (because ? allows only 0 or 1 occurrence)

### **6️⃣ Character Sets [ ] - Match Any One Character Inside**

A character set [ ] lets you match **any one** of the listed characters.

🔹 **Example:**

grep "[aeiou]" words.txt

✅ Matches: Any line containing a **vowel** (a, e, i, o, u).

🔹 **Example (Match lowercase letters a to z):**

grep "[a-z]" words.txt

✅ Matches: Any **lowercase** letter.

🔹 **Example (Match digits 0 to 9):**

grep "[0-9]" numbers.txt

✅ Matches: Any **single** digit.

### **7️⃣ Negated Character Sets [^ ] - Match Anything Except Given Characters**

🔹 **Example:**

grep "[^0-9]" numbers.txt

✅ Matches: Any character **except** digits.

🔹 **Example (Exclude vowels):**

grep "[^aeiou]" words.txt

✅ Matches: Any character **except vowels**.

### **8️⃣ The ^ (Caret) - Match Start of a Line**

🔹 **Example:**

grep "^ERROR" logfile.txt

✅ Matches: Lines that **start** with "ERROR".

🔹 **Example (Match lines starting with a number):**

grep "^[0-9]" data.txt

✅ Matches: Lines where the **first** character is a digit.

### **9️⃣ The $ (Dollar Sign) - Match End of a Line**

🔹 **Example:**

grep "success$" logfile.txt

✅ Matches: Lines that **end** with "success".

🔹 **Example (Find lines ending in a number):**

grep "[0-9]$" data.txt

✅ Matches: Lines where the **last** character is a digit.

### **🔟 {} - Match Exact, Minimum, or Maximum Occurrences**

| **Pattern** | **Meaning** |
| --- | --- |
| a{3} | Matches exactly **3** occurrences of a |
| a{2,5} | Matches **2 to 5** occurrences of a |
| a{2,} | Matches **at least 2** occurrences of a |

🔹 **Example:**

grep "a{3}" words.txt

✅ Matches: aaa but not aa or aaaaa.

🔹 **Example (Match numbers with exactly 4 digits):**

grep "[0-9]{4}" numbers.txt

✅ Matches: 1234, 5678  
 ❌ Doesn't match: 12, 12345

### **🔟 () - Grouping Expressions**

Parentheses () group parts of a pattern.

🔹 **Example:**

grep -E "(error|fail|warning)" logfile.txt

✅ Matches: error, fail, or warning (same as grep -E "error|fail|warning").

🔹 **Example (Match repeated groups):**

grep -E "(ab){2}" words.txt

✅ Matches: abab but not ab.

### **🔟 | (Pipe) - OR Operator**

The pipe | means **"or"**.

🔹 **Example:**

grep -E "cat|dog|bird" pets.txt

✅ Matches: **any** of cat, dog, or bird.

### **🔟 \ (Escape Special Characters)**

If you want to search for special characters (. \* + ? ^ $ [] { } | ()), **escape them** with \.

🔹 **Example (Find $100 exactly):**

grep "\$100" prices.txt

🔹 **Example (Find . exactly, not as "any character"):**

grep "\." filenames.txt

# **🔹 Advanced Examples with Explanations**

### **1️⃣ Find all valid email addresses**

grep -E "[a-zA-Z0-9.\_%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}" file.txt

✅ Explanation:

* [a-zA-Z0-9.\_%+-]+ → Matches the username (letters, numbers, .\_%+-).
* @ → The **@** symbol is required.
* [a-zA-Z0-9.-]+ → Matches the domain name (gmail, yahoo, etc.).
* \. → Escaped . ensures a literal dot before the domain.
* [a-zA-Z]{2,} → Matches top-level domains (com, org, edu).

### **2️⃣ Find all IP addresses**

grep -E "([0-9]{1,3}\.){3}[0-9]{1,3}" file.txt

✅ Explanation:

* ([0-9]{1,3}\.){3} → Matches 3 sets of numbers (0-999) followed by ..
* [0-9]{1,3} → Matches the last set of numbers.

## **Conclusion**

Tab 2

# **Advanced Guide to grep with Detailed Explanations and Examples**

The grep command is a **powerful text-search tool** in Linux that allows you to find patterns in files using **basic, extended, and Perl-compatible regular expressions**.

We'll cover **basic to advanced options** with **clear explanations and examples**.

## **1. Basic Usage of grep**

### **1️⃣ Searching for a Simple String**

grep "error" logfile.txt

✅ Finds lines that contain the word "error" in logfile.txt.

## **2. Common grep Options**

grep has various options that enhance its functionality.

### **2️⃣ Case-Insensitive Search (-i)**

grep -i "error" logfile.txt

✅ Matches error, Error, ERROR, etc.

### **3️⃣ Show Line Numbers (-n)**

grep -n "error" logfile.txt

✅ Displays **line numbers** where "error" appears.

### **4️⃣ Display Only Matching Text (-o)**

grep -o "error" logfile.txt

✅ Extracts **only** the matched word (error), not the entire line.

### **5️⃣ Count Occurrences (-c)**

grep -c "error" logfile.txt

✅ Shows the **count** of lines containing "error".

### **6️⃣ Invert Match (-v - Show Lines That Do NOT Match)**

grep -v "error" logfile.txt

✅ Displays **all lines** **except** those containing "error".

### **7️⃣ Match Whole Words (-w)**

grep -w "is" textfile.txt

✅ Matches is but not this or his.

## **3. Advanced Searching with grep**

### **8️⃣ Matching Start of Line (^)**

grep "^ERROR" logfile.txt

✅ Finds lines **starting** with "ERROR".

### **9️⃣ Matching End of Line ($)**

grep "success$" logfile.txt

✅ Finds lines **ending** with "success".

### **🔟 Recursive Search in Directories (-r or -R)**

grep -r "TODO" /home/user/projects

✅ Searches for "TODO" **inside all files** within /home/user/projects.

### **🔟 Searching Multiple Files**

grep "error" file1.txt file2.txt

✅ Searches for "error" in file1.txt and file2.txt.

### **🔟 Ignore Binary Files (--binary-files=text)**

grep --binary-files=text "error" \*

✅ Ensures grep searches inside **text files only**, ignoring binary files.

## **4. Using Extended Regular Expressions (grep -E)**

grep -E (or egrep) supports **extended** regex patterns.

### **1️⃣ Using | (OR Operator)**

grep -E "error|fail|warning" logfile.txt

✅ Matches "error", "fail", or "warning".

### **2️⃣ Using + (Match One or More)**

grep -E "go+d" words.txt

✅ Matches: god, good, goooood  
 ❌ Doesn't match: gd (because o must appear at least **once**).

### **3️⃣ Using {} (Match Exact, Min, Max Occurrences)**

grep -E "[0-9]{4,}" numbers.txt

✅ Finds numbers with **at least 4 digits**.

## **5. Complex grep Commands**

### **1️⃣ Find All Valid Email Addresses**

grep -E "[a-zA-Z0-9.\_%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}" file.txt

✅ Matches valid emails like user@example.com.

### **2️⃣ Find All IP Addresses**

grep -E "([0-9]{1,3}\.){3}[0-9]{1,3}" file.txt

✅ Matches valid IP addresses like 192.168.1.1.

## **6. Combining grep with Other Commands**

### **1️⃣ Find Running Processes Containing "firefox"**

ps aux | grep "firefox"

✅ Lists processes related to firefox.

### **2️⃣ Filter Log Files for Errors and Save to a File**

grep "ERROR" /var/log/syslog > errors.log

✅ Extracts lines containing "ERROR" from /var/log/syslog and saves them to errors.log.

## **7. Using grep with find**

To search for files containing a specific word:

find /home/user/docs -type f -exec grep "important" {} +

✅ Searches for "important" in all files inside /home/user/docs.

# **Advanced Guide to find with Detailed Explanations and Examples**

The find command in Linux is a **powerful tool** for searching files and directories based on **name, type, size, date, permissions, and other attributes**. It can also execute commands on matched files.

We'll cover **basic to advanced options** with **clear explanations and practical examples**.

## **1. Basic Syntax of find**

find [starting\_directory] [options] [expression]

* **starting\_directory** → Where to begin searching (/, ., ~/Documents, etc.).
* **options** → Filters like -name, -size, -type, etc.
* **expression** → Actions like -delete, -exec, etc.

# **🔹 2. Basic find Commands**

### **1️⃣ Find Files by Name (-name)**

find /home/user -name "notes.txt"

✅ Searches for notes.txt inside /home/user.

🔹 **Case-Insensitive Search (-iname)**

find /home/user -iname "notes.txt"

✅ Finds notes.txt, NOTES.txt, NoTeS.TxT, etc.

### **2️⃣ Find Directories by Name (-type d)**

find /home/user -type d -name "projects"

✅ Finds **directories** named "projects".

### **3️⃣ Find Files by Extension**

find /home/user -type f -name "\*.pdf"

✅ Finds all **PDF files** inside /home/user.

### **4️⃣ Find Files by Size (-size)**

| **Pattern** | **Meaning** |
| --- | --- |
| +1M | Files **larger** than 1MB |
| -100k | Files **smaller** than 100KB |
| 10M | Files **exactly** 10MB |

🔹 **Example:**

find /var/log -type f -size +10M

✅ Finds all **files larger than 10MB** inside /var/log.

### **5️⃣ Find Empty Files (-empty)**

find /home/user -type f -empty

✅ Finds **empty files**.

🔹 **Find Empty Directories:**

find /home/user -type d -empty

✅ Finds **empty directories**.

# **🔹 3. Searching by Date & Time**

### **6️⃣ Find Files Modified in the Last X Days (-mtime)**

| **Pattern** | **Meaning** |
| --- | --- |
| -3 | Files modified **in the last 3 days** |
| +7 | Files modified **more than 7 days ago** |

🔹 **Example:**

find /home/user -type f -mtime -7

✅ Finds files **modified in the last 7 days**.

### **7️⃣ Find Files Accessed in the Last X Days (-atime)**

find /home/user -type f -atime -1

✅ Finds files **accessed in the last 1 day**.

### **8️⃣ Find Files Changed in the Last X Minutes (-mmin)**

find /home/user -type f -mmin -30

✅ Finds files **modified in the last 30 minutes**.

# **🔹 4. Searching by Permissions & Ownership**

### **9️⃣ Find Files by Permissions (-perm)**

| **Pattern** | **Meaning** |
| --- | --- |
| 644 | Files with **644** permissions |
| -u=w | Files where the **owner can write** (w) |

🔹 **Example (Find files with 777 permissions):**

find /home/user -type f -perm 777

✅ Finds files that are **world-readable, writable, and executable**.

🔹 **Example (Find files where the owner has execute permissions):**

find /home/user -type f -perm -u=x

✅ Finds files where the **owner can execute** (x).

### **🔟 Find Files by Owner (-user)**

find /var/www -type f -user apache

✅ Finds all files **owned by user apache**.

### **🔟 Find Files by Group (-group)**

find /home/shared -type f -group developers

✅ Finds all files belonging to **group developers**.

# **🔹 5. Executing Commands on Found Files**

### **1️⃣ Delete All Matched Files (-delete)**

find /tmp -type f -name "\*.tmp" -delete

✅ Deletes all **temporary (.tmp) files** in /tmp.

🔴 **⚠ WARNING:** -delete is **dangerous**! Always test with -print first.

### **2️⃣ Execute a Command on Found Files (-exec)**

find /home/user -type f -name "\*.log" -exec rm {} \;

✅ Deletes all **.log** files inside /home/user.

🔹 **Find and Change Permissions (chmod)**

find /var/www -type f -name "\*.php" -exec chmod 644 {} \;

✅ Sets permissions **644** on all **PHP files**.

### **3️⃣ Using -exec with Multiple Commands (\;)**

find /home/user -type f -name "\*.txt" -exec echo "Processing: {}" \; -exec cat {} \;

✅ Prints "Processing: filename" before displaying the file's content.

### **4️⃣ Using -exec with + (Faster Execution)**

find /home/user -type f -name "\*.log" -exec rm {} +

✅ Deletes **all .log files at once**, reducing command execution overhead.

# **🔹 6. Using find with grep**

### **1️⃣ Find Files Containing a Specific Word**

find /var/log -type f -name "\*.log" -exec grep "error" {} +

✅ Searches for "error" inside all **.log** files.

# **🔹 7. Finding and Sorting Results**

### **1️⃣ Sort Files by Size**

find /home/user -type f -exec ls -lh {} + | sort -k5 -rh

✅ Finds all files and **sorts them by size** (largest first).

# **🔹 8. Find Files Modified Between Two Dates**

find /home/user -type f -newermt "2024-01-01" ! -newermt "2024-02-01"

✅ Finds files modified **between Jan 1, 2024, and Feb 1, 2024**.

# **🔹 9. Combining Multiple Conditions**

### **1️⃣ Find Files Matching Multiple Conditions**

find /home/user -type f -name "\*.txt" -size +10M -mtime -7

✅ Finds **.txt files** **larger than 10MB** and **modified in the last 7 days**.

## **Conclusion**

The find command is **one of the most powerful** Linux tools for searching files based on **name, size, time, permissions, ownership, and content**. Combined with -exec, -delete, and grep, it becomes an essential tool for **file management and automation**.

Would you like additional examples or explanations? 😊

Tab 3

I'll break down **cut**, **sort**, **uniq**, and **wc** in detail, covering their syntax, options, and use cases with in-depth explanations of their operations.

# **🔹 1. The cut Command**

The cut command is used to **extract specific parts of each line** from a file or standard input. It is commonly used to **select columns, extract fields, or split data**.

## **1️⃣ Basic Syntax**

cut -d 'delimiter' -f field\_number file

* **-d 'delimiter'** → Specifies the **delimiter** (default is **Tab**).
* **-f field\_number** → Extracts **specific field(s)**.
* **file** → Input file.

## **2️⃣ Extracting Columns (Fields)**

### **Example: Extract the 2nd Column from a CSV File**

cut -d ',' -f 2 data.csv

✅ Extracts the **2nd field** from data.csv, assuming **comma-separated values (CSV)**.

🔹 **How It Works?**

| **Name** | **Age** | **Country** |
| --- | --- | --- |
| John | 25 | USA |
| Alice | 30 | UK |

💡 Output:

25

30

## **3️⃣ Extracting Multiple Fields**

cut -d ',' -f 1,3 data.csv

✅ Extracts **1st and 3rd** fields.

💡 Output:

John,USA

Alice,UK

## **4️⃣ Cutting by Character Positions (-c)**

cut -c 1-5 names.txt

✅ Extracts **characters 1 to 5** from each line.

💡 Input:

Jonathan

Michael

💡 Output:

Jonat

Micha

## **5️⃣ Cutting Without a Delimiter (Fixed Width)**

cut -c 3- names.txt

✅ Extracts **everything from the 3rd character onwards**.

# **🔹 2. The sort Command**

The sort command arranges **lines of text files** in **ascending** or **descending** order.

## **1️⃣ Basic Syntax**

sort [options] file

* By default, sort sorts **alphabetically**.

## **2️⃣ Sorting a File Alphabetically**

sort names.txt

✅ Sorts **names in ascending order**.

💡 Input:

banana

apple

cherry

💡 Output:

apple

banana

cherry

## **3️⃣ Sorting in Reverse Order (-r)**

sort -r names.txt

✅ Sorts in **descending** order.

💡 Output:

cherry

banana

apple

## **4️⃣ Sorting Numerically (-n)**

sort -n numbers.txt

✅ Sorts **numbers** correctly.

💡 Input:

10

2

30

5

💡 Output:

2

5

10

30

## **5️⃣ Sorting by Column (-k)**

sort -t ',' -k2 -n data.csv

✅ Sorts **CSV** data **by the 2nd column numerically**.

💡 Input (data.csv):

Alice,30

John,25

Bob,40

💡 Output:

John,25

Alice,30

Bob,40

## **6️⃣ Sorting Unique Values (-u)**

sort -u names.txt

✅ Removes **duplicate entries**.

## **7️⃣ Case-Insensitive Sorting (-f)**

sort -f names.txt

✅ Ignores case when sorting.

💡 Input:

apple

Banana

cherry

💡 Output:

apple

Banana

cherry

# **🔹 3. The uniq Command**

The uniq command **removes duplicate adjacent lines**.

## **1️⃣ Basic Syntax**

uniq [options] file

* **uniq only works if duplicates are consecutive.**
* Always **sort** before using uniq.

## **2️⃣ Removing Duplicate Lines**

sort names.txt | uniq

✅ Sorts and removes duplicate lines.

💡 Input:

apple

apple

banana

banana

cherry

💡 Output:

apple

banana

cherry

## **3️⃣ Counting Duplicate Occurrences (-c)**

sort names.txt | uniq -c

✅ Counts **repeated lines**.

💡 Output:

2 apple

2 banana

1 cherry

## **4️⃣ Display Only Duplicates (-d)**

sort names.txt | uniq -d

✅ Shows **only repeated lines**.

💡 Output:

apple

banana

## **5️⃣ Display Only Unique (Non-Repeated) Lines (-u)**

sort names.txt | uniq -u

✅ Shows **only unique lines** (appearing once).

💡 Output:

cherry

# **🔹 4. The wc Command**

The wc (Word Count) command **counts the number of lines, words, and characters** in a file.

## **1️⃣ Basic Syntax**

wc [options] file

## **2️⃣ Counting Lines (-l)**

wc -l data.txt

✅ Counts the **number of lines** in data.txt.

💡 Output:

5 data.txt

(Means 5 lines)

## **3️⃣ Counting Words (-w)**

wc -w data.txt

✅ Counts **words**.

## **4️⃣ Counting Characters (-c)**

wc -c data.txt

✅ Counts **characters**.

## **5️⃣ Counting Bytes (-m)**

wc -m data.txt

✅ Counts **characters, including special ones** (like Unicode).

# **🔹 5. Combining Commands**

## **1️⃣ Find the Most Frequent Word in a File**

cat file.txt | tr ' ' '\n' | sort | uniq -c | sort -nr | head -10

✅ **Breakdown:**

1. cat file.txt → Read the file.
2. tr ' ' '\n' → Convert spaces to new lines (one word per line).
3. sort → Sort words alphabetically.
4. uniq -c → Count occurrences.
5. sort -nr → Sort counts in descending order.
6. head -10 → Show **top 10 words**.

## **Conclusion**

* **cut** → Extracts specific **columns/characters**.
* **sort** → Sorts data **alphabetically, numerically, or by column**.
* **uniq** → Removes **duplicate** adjacent lines.
* **wc** → Counts **lines, words, characters**.

Would you like more examples or deeper breakdowns? 😊

I'll now break down these options in **depth** with detailed explanations and **step-by-step illustrations**.

# **🔹 1. --complement Option**

The **--complement** option is used with the cut command to **invert the selection**—instead of extracting the specified fields, it **excludes them**, displaying everything **except** what is selected.

## **🔸 1.1 Basic Syntax**

cut --complement -f <fields> -d '<delimiter>' <file>

* **-f <fields>** → Specifies which fields (columns) to extract or exclude.
* **-d '<delimiter>'** → Defines the field separator (default is **TAB**).
* **--complement** → **Inverts** the selection (removes the specified fields instead of keeping them).

## **🔸 1.2 Example Use Case**

Let's say we have a **CSV file** called data.csv:

#### **File (data.csv):**

ID,Name,Age,Gender,Salary

1,Alice,25,F,50000

2,Bob,30,M,60000

3,Charlie,35,M,70000

### **1️⃣ Extracting Only Certain Fields (-f)**

If we **extract** columns **1 (ID) and 3 (Age)**:

cut -d ',' -f 1,3 data.csv

✅ Output:

ID,Age

1,25

2,30

3,35

### **2️⃣ Using --complement to Remove Fields**

Instead of **keeping** fields 1 and 3, let’s **remove** them:

cut --complement -d ',' -f 1,3 data.csv

✅ Output:

Name,Gender,Salary

Alice,F,50000

Bob,M,60000

Charlie,M,70000

💡 **What happened?**

* **Without --complement**, cut keeps fields 1 and 3.
* **With --complement**, it removes fields **1 and 3**, keeping everything else.

### **3️⃣ Practical Use Case: Filtering Out Unnecessary Data**

Suppose you need to process a **large CSV** but don’t want to include **ID and Age** in your report. Instead of manually selecting all other fields, you can just **exclude** them using --complement.

# **🔹 2. --output-delimiter Option**

The **--output-delimiter** option lets you specify a **custom delimiter** instead of using the default one.

## **🔸 2.1 Basic Syntax**

cut -d '<input-delimiter>' -f <fields> --output-delimiter='<new-delimiter>' <file>

* **-d '<input-delimiter>'** → Specifies the delimiter used in the input file.
* **-f <fields>** → Specifies which fields to extract.
* **--output-delimiter='<new-delimiter>'** → Changes the delimiter **in the output**.

## **🔸 2.2 Example Use Case**

Let's take the same data.csv file:

#### **File (data.csv):**

ID,Name,Age,Gender,Salary

1,Alice,25,F,50000

2,Bob,30,M,60000

3,Charlie,35,M,70000

### **1️⃣ Extracting Fields with Default Output Delimiter**

cut -d ',' -f 1,2,5 data.csv

✅ Output:

ID,Name,Salary

1,Alice,50000

2,Bob,60000

3,Charlie,70000

* The output **preserves** the input delimiter (,).

### **2️⃣ Using --output-delimiter to Change the Delimiter**

Now, let's replace , with | (pipe symbol):

cut -d ',' -f 1,2,5 --output-delimiter='|' data.csv

✅ Output:

ID|Name|Salary

1|Alice|50000

2|Bob|60000

3|Charlie|70000

💡 **What changed?**

* Input delimiter (-d ',') was a **comma**.
* Output delimiter (--output-delimiter='|') became a **pipe (|)**.

### **3️⃣ Why Use --output-delimiter?**

* If you're **converting** between formats (e.g., CSV → TSV).
* When working with **inconsistent data formats**.
* If you need **cleaner** output for further processing.

## **🔹 Summary**

| **Option** | **Purpose** | **Example Usage** |
| --- | --- | --- |
| **--complement** | Excludes instead of selecting fields | cut --complement -d ',' -f 1,3 data.csv |
| **--output-delimiter** | Changes the delimiter in the output | `cut -d ',' -f 1,2 --output-delimiter=' |

Would you like more advanced examples or a deeper breakdown? 😊

Tab 4

I'll now break down **zip / tar**, **paste**, and **redirection (>, >>, <)** with detailed explanations, syntax, and examples.

# **🔹 1. The zip and tar Commands**

The **zip** and **tar** commands are used for file compression and archiving.

### **What’s the Difference?**

* **zip** → Compresses each file separately **(Windows-compatible)**.
* **tar** → Archives files into one package, optionally compressing it.

## **🔸 1.1 The zip Command**

The **zip** command compresses files into a .zip archive.

### **1️⃣ Basic Syntax**

zip [options] archive\_name.zip file1 file2 ...

### **2️⃣ Creating a Zip Archive**

zip my\_files.zip file1.txt file2.txt

✅ Compresses file1.txt and file2.txt into my\_files.zip.

### **3️⃣ Creating a Zip Archive Recursively (-r)**

zip -r my\_folder.zip my\_folder

✅ Compresses the entire my\_folder directory.

### **4️⃣ Extracting a Zip Archive**

unzip my\_files.zip

✅ Extracts all files from my\_files.zip.

## **🔸 1.2 The tar Command**

The **tar** command is used to **archive** multiple files into one .tar file, optionally compressing it.

### **1️⃣ Basic Syntax**

tar [options] archive\_name.tar file1 file2 ...

### **2️⃣ Creating an Archive (-cvf)**

tar -cvf my\_archive.tar file1.txt file2.txt

✅ Creates an **uncompressed** my\_archive.tar containing the files.

* **-c** → Create an archive
* **-v** → Verbose (shows progress)
* **-f** → Specify filename

### **3️⃣ Extracting an Archive (-xvf)**

tar -xvf my\_archive.tar

✅ Extracts my\_archive.tar.

### **4️⃣ Creating a Compressed Archive (-zcvf)**

tar -zcvf my\_archive.tar.gz file1.txt file2.txt

✅ Creates a **compressed** archive using **Gzip** (.tar.gz).

* **-z** → Compress with gzip

### **5️⃣ Extracting a Compressed Archive**

tar -zxvf my\_archive.tar.gz

✅ Extracts .tar.gz.

### **6️⃣ Creating a bzip2 Compressed Archive (-jcvf)**

tar -jcvf my\_archive.tar.bz2 file1.txt file2.txt

✅ Uses **bzip2** compression (.tar.bz2).

### **7️⃣ Listing Contents Without Extracting**

tar -tf my\_archive.tar

✅ Displays files inside the archive.

# **🔹 2. The paste Command**

The **paste** command joins lines from multiple files **side by side**.

## **🔸 2.1 Basic Syntax**

paste file1 file2

### **1️⃣ Merging Two Files Line by Line**

#### **File 1 (names.txt):**

Alice

Bob

Charlie

#### **File 2 (ages.txt):**

25

30

35

🔹 **Command:**

paste names.txt ages.txt

✅ Output:

Alice 25

Bob 30

Charlie 35

### **2️⃣ Using a Custom Delimiter (-d)**

paste -d ',' names.txt ages.txt

✅ Uses **comma (,)** as a separator.

💡 Output:

Alice,25

Bob,30

Charlie,35

### **3️⃣ Joining Multiple Lines of the Same File (-s)**

paste -s names.txt

✅ Converts **vertical** lines into **horizontal**.

💡 Input:

Alice

Bob

Charlie

💡 Output:

Alice Bob Charlie

# **🔹 3. Redirection (>, >>, <)**

Redirection allows **input and output manipulation** in the terminal.

## **🔸 3.1 Output Redirection (>)**

The **> operator** redirects **stdout** (standard output) to a file, **overwriting** existing content.

echo "Hello, world!" > output.txt

✅ Writes "Hello, world!" to output.txt, **overwriting** existing content.

💡 **Example File (output.txt):**

Hello, world!

## **🔸 3.2 Appending to a File (>>)**

The **>> operator** appends output **without overwriting**.

echo "New line" >> output.txt

✅ Appends "New line" at the end of output.txt.

💡 **Updated File (output.txt):**

Hello, world!

New line

## **🔸 3.3 Input Redirection (<)**

The **< operator** feeds a file as input to a command.

wc -w < file.txt

✅ Counts words in file.txt.

## **🔸 3.4 Combining Input and Output Redirection**

sort < unsorted.txt > sorted.txt

✅ Reads from unsorted.txt, sorts it, and saves to sorted.txt.

## **🔸 3.5 Using Pipes (|)**

Pipes (|) **connect** multiple commands.

cat file.txt | sort | uniq -c | sort -nr

✅ Sorts, removes duplicates, and counts word frequency.

## **🔸 3.6 Redirecting Errors (2>)**

ls non\_existent\_file 2> error.log

✅ Redirects **error messages** to error.log.

## **🔸 3.7 Redirecting Both Output and Errors**

command > output.log 2>&1

✅ Redirects **both stdout and stderr** to output.log.

## **Conclusion**

| **Command** | **Purpose** |
| --- | --- |
| **zip** | Compresses files (Windows-compatible) |
| **tar** | Archives multiple files (Linux standard) |
| **paste** | Merges files line by line |
| **>** | Overwrites output to a file |
| **>>** | Appends output to a file |
| **<** | Reads input from a file |
| **2>** | Redirects errors |

Would you like more advanced examples? 😊