ADMIN:DOCS

LPIC-1: Textverarbeitung mit Shell-Kommandos

Lerne, wie du mit grep, sed, awk, sort, uniq, wc, cut und regulären Ausdrücken Textdateien in Linux effizient analysierst, filterst, sortierst und bearbeitest.

Kategorie: LPIC-1 Serie **Veröffentlicht:** 28.05.2025 **Zuletzt geprüft:** 02.11.2026

In den ersten drei Teilen unserer LPIC-1-Serie haben wir die Grundlagen der Linux-Kommandozeile, die Navigation im Dateisystem und das Bearbeiten von Dateiinhalten kennengelernt. Nun ist es an der Zeit, einen entscheidenden Schritt weiterzugehen: die mächtige Welt der Shell-basierten Textverarbeitung zu erkunden.

Die Textverarbeitung über Shell-Kommandos ist eine der charakteristischsten und mächtigsten Eigenschaften von Linux und Unix-Systemen. Während andere Betriebssysteme oft auf grafische Tools oder spezialisierte Programme angewiesen sind, bietet Linux eine Sammlung hocheffizienter Kommandozeilen-Werkzeuge, die sich nahtlos miteinander kombinieren lassen.

> **Wichtiger Hinweis:** Wie bereits in den vorherigen Artikeln betont, ersetzt diese Serie keinen offiziellen Prüfungsvorbereitungskurs für die LPIC-1-Zertifizierung. Sie dient als praxisorientierte, didaktisch aufbereitete Ergänzung für dein Selbststudium und soll dir helfen, die teils komplexen Inhalte besser zu verstehen und anzuwenden.

Warum Shell-basierte Textverarbeitung so zentral ist

1. **Rohdaten (Dateien / Streams / Standard Input)** Logdateien, Konfigurationsdaten oder Prozess-Ausgaben

2. **1. Filtern (grep, egrep, fgrep)** Extrahiert Zeilen basierend auf Mustern oder regulären Ausdrücken
3. **2. Transformieren & Extrahieren (cut, tr, sed, awk)** Spalten schneiden, Zeichen ersetzen oder Felder gezielt formatieren
4. **3. Sortieren & Aggregieren (sort, uniq, wc)** Sortiert Datenströme, dedupliziert Zeilen und zählt Häufigkeiten
5. **Strukturierte Ausgabe / Analysebericht** Standard Output, Report-Datei oder Weiterleitung in Folgebefehle

In der Linux-Administration ist Text allgegenwärtig: Konfigurationsdateien, Logdateien, Systemausgaben, Berichte und Skripte – alles basiert auf Textdaten. Die Fähigkeit, diese Textdaten effizient zu durchsuchen, zu filtern, zu transformieren und zu analysieren, unterscheidet einen kompetenten Linux-Administrator von einem Anfänger.

Die Shell-basierten Textverarbeitungstools folgen der klassischen Unix-Philosophie:

- Jedes Tool macht eine Sache, aber die macht es sehr gut
- Tools lassen sich über Pipes miteinander kombinieren
- Einfache Tools können zu komplexen Lösungen zusammengefügt werden
- Textdaten sind das universelle Interface zwischen den Tools

Diese Herangehensweise ermöglicht es, mit wenigen, gut beherrschten Befehlen komplexe Datenanalysen und `-transformationen` durchzuführen, die in anderen Umgebungen spezialisierte Software erfordern würden.

Bedeutung für die LPIC-1-Zertifizierung

Die Shell-basierte Textverarbeitung nimmt in der LPIC-1-Prüfung einen wichtigen Stellenwert ein, insbesondere in den folgenden Prüfungszielen:

103.2:

- Textströme mit Filtern verarbeiten

103.3:

- Grundlegende Dateimanipulation durchführen

103.7:

- Suchen und Extrahieren von Daten aus Dateien mit regulären Ausdrücken

Zusammen machen diese Themenbereiche etwa **20–25 %** der Punkte in der LPIC-1 Exam 101 aus, was die Wichtigkeit dieser Fähigkeiten unterstreicht.

Was dich in diesem Artikel erwartet

In den kommenden Abschnitten werden wir folgende Kernthemen behandeln:

- Grundlegende Textverarbeitung mit den "großen Drei": `grep`, `sed` und `awk`
- Textsortierfunktionen mit `sort` und `uniq`, um Daten zu ordnen und Duplikate zu verwalten
- Textdateien zählen und formatieren mit Werkzeugen wie `wc`, `fmt`, `nl` und `cut`
- Reguläre Ausdrücke als mächtiges Werkzeug für Mustersuche und `-ersetzung`

Jeder Abschnitt wird nicht nur die technischen Aspekte erklären, sondern auch praktische Anwendungsszenarien aus der realen Systemadministration aufzeigen.

Praktische Relevanz für den Administrator-Alltag

Die in diesem Artikel behandelten Befehle gehören zum täglichen Handwerkszeug jedes Linux-Administrators.

Typische Anwendungsszenarien sind:

- Analyse von Logdateien zur Fehlersuche und Systemüberwachung
- Extraktion spezifischer Informationen aus Konfigurationsdateien
- Automatisierte Berichte über Systemzustände
- Datenbereinigung und `-transformation` für weitere Verarbeitung
- Überwachung von Systemressourcen und Benutzeraktivitäten

> □□ **Hinweis:** Wie gewohnt findest du im gesamten Artikel spezielle Markierungen:
>
> - □□ **Tipps und Hinweise** für effizientere Arbeitsweisen
> - ⚠ **Warnungen und Stolperfallen**, die dir Probleme ersparen
> - □□ **Praktische Beispiele** zum direkten Nachvollziehen
> - □ **Typische Fehlerquellen** und deren Lösungen

So holst du das meiste aus diesem Artikel heraus:

Für optimales Lernen empfehle ich dir dringend, die Befehle und Techniken parallel in einer eigenen Linux-Umgebung auszuprobieren. Erstelle testweise Textdateien mit verschiedenen Inhalten, experimentiere mit den vorgestellten Befehlen und beobachte die Ergebnisse.

Besonders bei der Textverarbeitung ist praktisches Üben unerlässlich – die Kombination verschiedener Befehle über Pipes erfordert Übung und Experimentierfreude. Je mehr du mit diesen Werkzeugen arbeitest, desto intuitiver wird ihre Anwendung.

Lass uns nun eintauchen in die faszinierende Welt der Shell-basierten Textverarbeitung – ein Bereich, der deine Effizienz als Linux-Administrator revolutionieren wird und dir gleichzeitig wichtige Punkte in der LPIC-1-Prüfung sichert.

Grundlegende Textverarbeitung

Die drei Säulen der Linux-Textverarbeitung – grep, sed und awk – bilden das Herzstück der Shell-basierten Datenmanipulation. Diese Werkzeuge ergänzen sich perfekt und ermöglichen es, komplexe Textverarbeitungsaufgaben mit eleganten, effizienten Lösungen zu bewältigen. Jedes Tool hat seine spezifischen Stärken und Einsatzgebiete, die wir in diesem Abschnitt detailliert erkunden werden.

grep - Der mächtige Textsucher

Was ist grep und warum ist es unverzichtbar?

grep (Global Regular Expression Print) ist das fundamentale Suchwerkzeug in Linux-Systemen. Es durchsucht Textdateien oder Datenströme nach Zeilen, die einem bestimmten Muster entsprechen, und gibt diese aus. Der Name verrät bereits seine Herkunft: Es stammt aus dem ed-Editor-Befehl g/re/p (global/regular expression/print).

Die Stärke von grep liegt in seiner Einfachheit und Geschwindigkeit. Es ist optimiert für eine Aufgabe: das schnelle Auffinden von Textmustern in großen Datenmengen.

Grundlegende Syntax und Anwendung

Die grundlegende Syntax von grep ist intuitiv:

```
grep [Optionen] "Suchmuster" datei(en)
```

□□ Einfache Beispiele:

```
# Suche nach dem Wort "error" in einer Logdatei
grep "error" /var/log/syslog

# Suche in mehreren Dateien gleichzeitig
grep "Failed" /var/log/auth.log /var/log/secure

# Suche über Pipes (sehr häufig verwendet)
ps aux | grep apache
```

□□ Das letzte Beispiel zeigt eine der häufigsten Anwendungen von grep: die Filterung der Ausgabe anderer Befehle über Pipes.

Wichtige Optionen und ihre praktische Anwendung

grep bietet eine Vielzahl von Optionen, die seine Funktionalität erheblich erweitern:

Option	Beschreibung	Praktisches Beispiel
`-i`	Ignoriert Groß-/Kleinschreibung	`grep -i "error" logfile.txt`
`-v`	Invertiert die Suche (zeigt Zeilen OHNE das Muster)	`grep -v "^#" config.conf`
`-n`	Zeigt Zeilennummern an	`grep -n "TODO" script.sh`
`-r` oder `-R`	Rekursive Suche in Verzeichnissen	`grep -r "password" /etc/`
`-l`	Zeigt nur Dateinamen mit Treffern	`grep -l "function" *.py`
`-c`	Zählt die Anzahl der Treffer	`grep -c "GET" access.log`
`-A n`	Zeigt n Zeilen nach dem Treffer	`grep -A 3 "Exception" error.log`
`-B n`	Zeigt n Zeilen vor dem Treffer	`grep -B 2 "FATAL" system.log`
`-C n`	Zeigt n Zeilen vor und nach dem Treffer	`grep -C 5 "segfault" kernel.log`

□□ Praktische Anwendungsbeispiele für Systemadministration:

a) Fehleranalyse in Logdateien:

```
# Finde alle kritischen Fehler der letzten Stunde
grep -i "critical\|fatal\|emergency" /var/log/syslog

# Zeige Kontext um SSH-Verbindungsfehler
grep -C 3 "Failed password" /var/log/auth.log
```

b) Konfigurationsdateien analysieren:

```
# Finde alle kritischen Fehler der letzten Stunde
grep -i "critical\|fatal\|emergency" /var/log/syslog

# Zeige Kontext um SSH-Verbindungsfehler
grep -C 3 "Failed password" /var/log/auth.log
```

c) Systemüberwachung:

```
# Zähle fehlgeschlagene Login-Versuche
grep -c "Failed password" /var/log/auth.log

# Finde alle Prozesse eines bestimmten Benutzers
ps aux | grep -v grep | grep username
```

> ⚠ **Vorsicht mit der `-i` Option:** Sie verändert die Originaldatei unwiderruflich!

Erstelle immer Backups bei wichtigen Dateien:

```
# Sicherer Ansatz mit Backup
sed -i.bak 's/alt/neu/g' wichtige_datei.conf
```

Erweiterte sed-Techniken

Adressierung von Zeilen:

```
# Nur in Zeilen 10-20 ersetzen
sed '10,20s/alt/neu/g' datei.txt

# Vom ersten Vorkommen eines Musters bis Dateiende
sed '/START/,/alt/neu/g' datei.txt

# Jede zweite Zeile bearbeiten
sed '1~2s/alt/neu/g' datei.txt
```

Mehrere Befehle kombinieren:

```
# Mehrere Ersetzungen in einem Durchgang
sed -e 's/http/https/g' -e 's/80/443/g' -e '/^#/d' config.txt

# Mit Semikolon trennen
sed 's/alt/neu/g; /^#/d; s/[[:space:]]*//' datei.txt
```

> **Tipp für komplexe Operationen:** Bei sehr komplexen `sed`-Operationen kann es sinnvoller sein, ein `sed`-Skript zu erstellen.

```
# Erstelle sed-script.sed
cat > sed-script.sed << 'EOF'
s/http/https/g
```

```
s/80/443/g
/^#/d
/^$/d
EOF

# Verwende das Skript
sed -f sed-script.sed konfiguration.conf
```

awk - Einführung in die Textverarbeitung

Grundkonzept von awk als Pattern-Action-Sprache

`awk` ist mehr als nur ein Textverarbeitungswerkzeug – es ist eine vollständige Programmiersprache, die speziell für die Verarbeitung strukturierter Textdaten entwickelt wurde. Der Name stammt von den Nachnamen seiner Entwickler: Aho, Weinberger und Kernighan.

Das Grundkonzept von awk basiert auf dem Pattern-Action-Paradigma:

1. **Eingabezeile einlesen**awk liest Zeile für Zeile aus Dateien oder Datenströmen
2. **Muster / Filterbedingung prüfen**Bedingung evaluieren (z. B. regulärer Ausdruck oder $3 >= 1000)
3. **Aktion ausführen**Bei Übereinstimmung Befehle im Block `{ Aktion }` ausführen
4. **Ausgabe & nächster Datensatz**print/printf liefert Ergebnis, awk springt zur nächsten Zeile

Für jede Eingabezeile prüft `awk`, ob sie dem Muster entspricht. Wenn ja, wird die zugehörige Aktion ausgeführt.

Die grundlegende Struktur eines awk-Programms:

```
awk 'BEGIN { Initialisierung }
     Muster1 { Aktion1 }
     Muster2 { Aktion2 }
     END { Abschluss }' eingabedatei
```

Einfache awk-Befehle für Spaltenverarbeitung

`awk` teilt jede Eingabezeile automatisch in Felder auf, die über `$1`, `$2`, `$3` etc. angesprochen werden können. `$0` repräsentiert die gesamte Zeile.

□□ Grundlegende Beispiele:

```
# Erste Spalte aller Zeilen ausgeben
awk '{print $1}' datei.txt

# Erste und dritte Spalte ausgeben
awk '{print $1, $3}' datei.txt

# Zeilen mit mehr als 3 Feldern anzeigen
awk 'NF > 3 {print $0}' datei.txt

# Zeilennummer und Inhalt anzeigen
awk '{print NR ": " $0}' datei.txt
```

Wichtige eingebaute Variablen:

Variable	Beschreibung	Beispiel
`NR`	Aktuelle Zeilennummer	`awk 'NR == 5 {print}' datei.txt`
`NF`	Anzahl Felder in aktueller Zeile	`awk '{print NF}' datei.txt`

Quelle: https://admindocs.de/de/lpic-1-serie/lpic-1-textverarbeitung-mit-shell-kommandos

Variable	Beschreibung	Beispiel
`FS`	Feldtrenner (Field Separator)	`awk 'BEGIN{FS=":"} {print $1}' /etc/passwd`
`RS`	Zeilentrenner (Record Separator)	`awk 'BEGIN{RS=";"} {print}' datei.txt`
`OFS`	Ausgabe-Feldtrenner (Output Field Separator)	`awk 'BEGIN{OFS=","} {print $1,$2}' datei.txt`

Praktische Anwendungsfälle in der Systemadministration

1. Analyse der `/etc/passwd` Datei:

```
# Alle Benutzernamen anzeigen
awk -F: '{print $1}' /etc/passwd

# Benutzer mit UID größer als 1000 (normale Benutzer)
awk -F: '$3 >= 1000 {print $1 " (UID: " $3 ")"}' /etc/passwd

# Benutzer mit Bash als Standard-Shell
awk -F: '$7 == "/bin/bash" {print $1}' /etc/passwd
```

2. Log-Analyse:

```
# Größe der Logdateien summieren
ls -l /var/log/*.log | awk '{sum += $5} END {print "Gesamt: " sum " Bytes"}'

# Apache Access Log: IP-Adressen und Status-Codes
awk '{print $1, $9}' /var/log/apache2/access.log
```

Quelle: https://admindocs.de/de/lpic-1-serie/lpic-1-textverarbeitung-mit-shell-kommandos

```
# Anzahl der Requests pro IP-Adresse
awk '{count[$1]++} END {for (ip in count) print ip, count[ip]}' access.log
```

3. Systemressourcen überwachen:

```
# Speicherverbrauch der Prozesse
ps aux | awk 'NR > 1 {mem += $4} END {print "Gesamt RAM-Nutzung: " mem "%"}'

# Top 5 CPU-verbrauchende Prozesse
ps aux | awk 'NR > 1 {print $3, $11}' | sort -nr | head -5
```

4. Netzwerk-Analyse:

```
# Aktive Verbindungen pro Port
netstat -an | awk '/LISTEN/ {count[$4]++} END {for (port in count) print port,
count[port]}'

# Eindeutige IP-Adressen in einem Access-Log zählen
awk '{ips[$1]++} END {print "Eindeutige IPs:", length(ips)}' access.log
```

Erweiterte awk-Funktionen

Bedingte Verarbeitung:

```
# Zeilen mit bestimmten Bedingungen verarbeiten
awk '$3 > 100 && $4 < 50 {print "Warnung:", $0}' messdaten.txt

# If-else Konstrukte
awk '{if ($1 > 0) print "Positiv"; else print "Null oder negativ"}' zahlen.txt
```

Mathematische Operationen:

```
# Durchschnitt berechnen
awk '{sum += $1; count++} END {print "Durchschnitt:", sum/count}' zahlen.txt

# Minimum und Maximum finden
awk 'BEGIN{min=999999; max=-999999} {if($1<min) min=$1; if($1>max) max=$1} END{print
"Min:", min, "Max:", max}' zahlen.txt
```

□□ **Tipp für komplexe awk-Programme:** Bei längeren awk-Programmen ist es oft sinnvoller, sie in separate Dateien zu schreiben:

```
# Erstelle script.awk
cat > analyse.awk << 'EOF'
BEGIN {
    print "Starte Analyse..."
    FS = ":"
}
NR > 1 {
    if ($3 >= 1000) {
```

```
        users++
        print $1 " ist ein normaler Benutzer"
    }
}
END {
    print "Gefunden:", users, "normale Benutzer"
}
EOF

# Verwende das Skript
awk -f analyse.awk /etc/passwd
```

Unterschiede und Einsatzgebiete von grep, sed und awk

Um die richtige Wahl zwischen diesen drei Werkzeugen zu treffen, ist es wichtig, ihre jeweiligen Stärken zu verstehen:

Werkzeug	Hauptzweck	Ideal für	Komplexität
grep	Textsuche und `-filterung`	Finden von Zeilen mit bestimmten Mustern	Niedrig
sed	Stream-Editing	Einfache Textersetzungen und `-manipulationen`	Mittel
awk	Strukturierte Datenverarbeitung	Spaltenbasierte Operationen und Berechnungen	Hoch

Entscheidungshilfe:

- Verwende `grep`, wenn du nur nach Text suchen oder Zeilen filtern möchtest
- Verwende `sed`, wenn du Text ersetzen, löschen oder einfache Manipulationen durchführen möchtest
- Verwende `awk`, wenn du mit strukturierten Daten arbeitest oder komplexe Logik benötigst

□□ Praktisches Beispiel - Alle drei Tools in einer Pipeline:

```
# Analysiere Apache-Logs: Finde 404-Fehler, extrahiere IP und Zeit, sortiere nach
Häufigkeit
grep "404" /var/log/apache2/access.log |
awk '{print $1, $4}' |
sed 's/[//g' |
sort | uniq -c | sort -nr | head -10
```

Diese Pipeline demonstriert die Stärken aller drei Tools:

- grep filtert relevante Zeilen (404-Fehler)
- awk extrahiert spezifische Felder (IP-Adresse und Zeit)
- sed bereinigt die Formatierung
- sort und uniq zählen und sortieren die Ergebnisse

> ⚠ **Wichtiger Hinweis für die LPIC-1-Prüfung:** Du solltest nicht nur die Syntax dieser Befehle kennen, sondern auch verstehen, wann welches Tool am besten geeignet ist. Prüfungsaufgaben testen oft das Verständnis für die richtige Werkzeugwahl.

□ Typische Fehlerquellen:

- Feldtrenner nicht beachten: awk verwendet standardmäßig Leerzeichen als Trenner
- Vergessen der Anführungszeichen: Besonders bei awk-Programmen mit Leerzeichen
- Verwechslung von Zeilen- und Feldnummern: NR vs. NF in awk
- Regex-Syntax verwechseln: sed und awk haben leicht unterschiedliche Regex-Dialekte

Mit diesem grundlegenden Verständnis von `grep`, `sed` und `awk` hast du die wichtigsten Werkzeuge der Linux-Textverarbeitung kennengelernt. Diese drei Befehle bilden das Fundament für nahezu alle textbasierten Operationen in Linux-Systemen und sind essentiell für die LPIC-1-Zertifizierung.

Quelle: https://admindocs.de/de/lpic-1-serie/lpic-1-textverarbeitung-mit-shell-kommandos

Textsortierfunktionen

Die Fähigkeit, Textdaten zu sortieren und Duplikate zu verwalten, ist ein fundamentaler Baustein der Linux-Textverarbeitung. Die Befehle `sort` und `uniq` arbeiten Hand in Hand und ermöglichen es, aus chaotischen Datenmengen strukturierte, bereinigte Informationen zu extrahieren. Diese Werkzeuge sind besonders wertvoll für die Analyse von Logdateien, die Aufbereitung von Systemdaten und die Erstellung aussagekräftiger Berichte.

sort - Texte systematisch ordnen

Was ist sort und warum ist es unverzichtbar?

Der Befehl `sort` ordnet Textzeilen nach verschiedenen Kriterien und ist eines der am häufigsten verwendeten Werkzeuge in der Linux-Textverarbeitung. Seine Stärke liegt nicht nur in der einfachen alphabetischen Sortierung, sondern in der Vielfalt der Sortieroptionen und der nahtlosen Integration in komplexe Verarbeitungspipelines.

> □□ **sort folgt dem Unix-Prinzip der Spezialisierung:** Es macht eine Sache – das Sortieren – aber diese außergewöhnlich gut und flexibel.

Grundlegende Sortierung von Textdateien

Die einfachste Verwendung von `sort` ist die lexikographische (alphabetische) Sortierung:

```
sort datei.txt
```

Standardmäßig sortiert `sort` zeilenweise nach dem gesamten Zeileninhalt, beginnend mit dem ersten Zeichen jeder Zeile.

□□ Praktisches Beispiel:

Erstelle eine unsortierte Liste:

```
cat > namen.txt << EOF
Müller
Schmidt
Bauer
Abel
Zimmermann
EOF
```

Sortiere die Liste:

```
sort namen.txt
Abel
Bauer
Müller
Schmidt
Zimmermann
```

> ⚠ **Wichtiger Hinweis zur Lokalisierung:** Die Sortierreihenfolge wird durch die Systemlokalisierung (LOCALE) beeinflusst. Deutsche Umlaute werden je nach Einstellung unterschiedlich behandelt.

```
# Aktuelle Lokalisierung anzeigen
locale | grep LC_COLLATE

# Sortierung mit C-Locale (ASCII-Reihenfolge)
LC_COLLATE=C sort namen.txt

# Sortierung mit deutscher Lokalisierung
LC_COLLATE=de_DE.UTF-8 sort namen.txt
```

Wichtige Sortieroptionen und ihre praktische Anwendung

`sort` bietet eine Vielzahl von Optionen für verschiedene Sortieranforderungen:

Option	Beschreibung	Praktisches Beispiel
`-n`	Numerische Sortierung	`sort -n zahlen.txt`
`-r`	Umgekehrte Reihenfolge	`sort -r datei.txt`
`-k`	Nach bestimmter Spalte sortieren	`sort -k2 tabelle.txt`
`-t`	Feldtrenner definieren	`sort -t: -k3 /etc/passwd`
`-u`	Entfernt Duplikate während der Sortierung	`sort -u liste.txt`
`-f`	Ignoriert Groß-/Kleinschreibung	`sort -f gemischte_liste.txt`
`-M`	Sortiert nach Monatsnamen	`sort -M monate.txt`
`-h`	"Human-readable" Größen (1K, 2M, 3G)	`sort -h dateigrößen.txt`
`-c`	Überprüft, ob bereits sortiert	`sort -c datei.txt`
`-o`	Ausgabe in Datei	`sort -o sortiert.txt eingabe.txt`

Sortierung nach verschiedenen Kriterien

1. Numerische Sortierung:

```
# Erstelle eine Liste mit Zahlen
cat > zahlen.txt << EOF
100
20
3
1000
5
EOF

# Alphabetische Sortierung (Standard) - oft nicht gewünscht!
sort zahlen.txt
100
1000
20
3
5

# Numerische Sortierung - das gewünschte Ergebnis
sort -n zahlen.txt
3
5
20
100
1000
```

2. Umgekehrte Sortierung:

```
# Größte Zahlen zuerst
sort -nr zahlen.txt
1000
100
20
5
```

```
3
```

3. Sortierung nach Spalten:

Die Option `-k` ermöglicht die Sortierung nach bestimmten Feldern oder Spalten:

```
# Beispiel: Sortierung der /etc/passwd nach UID (3. Feld)
sort -t: -k3 -n /etc/passwd | head -5
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
```

Hier bedeutet:

- `-t` : setzt den Doppelpunkt als Feldtrenner
- `-k3` sortiert nach dem 3. Feld (UID)
- `-n` führt numerische Sortierung durch

4. Mehrkriterien-Sortierung:

```
# Erstelle eine Tabelle mit Mitarbeiterdaten
cat > mitarbeiter.txt << EOF
Müller 1500 IT
Schmidt 2000 IT
Bauer 1500 HR
Abel 2500 IT
```

```
Zimmermann 1500 Sales
EOF
```

Sortiere erst nach Gehalt (2. Spalte), dann nach Name (1. Spalte):

```
sort -k2 -n -k1 mitarbeiter.txt
Bauer 1500 HR
Müller 1500 IT
Zimmermann 1500 Sales
Schmidt 2000 IT
Abel 2500 IT
```

Praktische Anwendungsbeispiele für Log-Analyse

1. Apache Access Log nach IP-Adressen sortieren:

```
# Extrahiere und sortiere IP-Adressen
awk '{print $1}' /var/log/apache2/access.log | sort | head -10
```

2. Systemlast-Analyse:

```
# Sortiere Prozesse nach CPU-Verbrauch ps aux | sort -k3 -nr | head -10
```

```
# Sortiere Prozesse nach CPU-Verbrauch
ps aux | sort -k3 -nr | head -10
```

3. Speicherverbrauch analysieren:

```
# Sortiere Verzeichnisse nach Größe
du -h /var/log/* | sort -hr | head -10
```

4. Netzwerkverbindungen analysieren:

```
# Sortiere aktive Verbindungen nach Port
netstat -an | grep ESTABLISHED | sort -k4
```

> **Tipp für komplexe Sortierungen:** Bei mehreren Sortierkriterien werden die `-k` Optionen in der Reihenfolge ihrer Priorität angegeben. Das erste `-k` hat die höchste Priorität.

Erweiterte sort-Funktionen

Sortierung mit benutzerdefinierten Feldbereichen:

```
# Sortiere nur nach einem Teil eines Feldes
sort -k1.2,1.4 datei.txt  # Sortiert nach Zeichen 2-4 des ersten Feldes
```

Sortierung von IP-Adressen:

```
# Korrekte Sortierung von IPv4-Adressen
sort -t. -k1,1n -k2,2n -k3,3n -k4,4n ip_adressen.tx
```

Überprüfung der Sortierung:

```
# Prüfe, ob eine Datei bereits sortiert ist
sort -c datei.txt
sort: datei.txt:3: ungeordnet: Abel

# Stille Überprüfung (nur Exit-Code)
sort -C datei.txt && echo "Sortiert" || echo "Nicht sortiert"
```

uniq - Duplikate erkennen und verwalten

Was ist uniq und wie arbeitet es mit sort zusammen?

Der Befehl `uniq` entfernt oder identifiziert aufeinanderfolgende identische Zeilen in einer Datei. Ein wichtiger Punkt: `uniq` erkennt nur aufeinanderfolgende Duplikate! Deshalb wird es fast immer in Kombination mit `sort` verwendet, um alle Duplikate in einer Datei zu finden.

Die typische Kombination ist:

```
sort datei.txt | uniq
```

Entfernen von doppelten Zeilen

Grundlegende Duplikatentfernung:

```
# Erstelle eine Datei mit Duplikaten
cat > duplikate.txt << EOF
Apfel
Banane
Apfel
Orange
Banane
Apfel
EOF
```

Nur aufeinanderfolgende Duplikate entfernen (nicht das gewünschte Ergebnis):

```
uniq duplikate.txt
Apfel
Banane
Apfel
Orange
Banane
```

```
Apfel
```

Korrekte Methode: Erst sortieren, dann Duplikate entfernen

```
sort duplikate.txt | uniq
Apfel
Banane
Orange
```

□□ Praktisches Beispiel für Systemadministration:

```
# Eindeutige IP-Adressen aus einem Access-Log extrahieren
awk '{print $1}' /var/log/apache2/access.log | sort | uniq > eindeutige_ips.txt

# Anzahl eindeutiger Besucher zählen
awk '{print $1}' /var/log/apache2/access.log | sort | uniq | wc -l
```

Wichtige uniq-Optionen

Option	Beschreibung	Praktisches Beispiel
`-c`	Zählt Vorkommen jeder Zeile	`sort datei.txt | uniq -c`
`-d`	Zeigt nur doppelte Zeilen	`sort datei.txt | uniq -d`
`-u`	Zeigt nur eindeutige Zeilen	`sort datei.txt | uniq -u`

Option	Beschreibung	Praktisches Beispiel
`-i`	Ignoriert Groß-/Kleinschreibung	`sort datei.txt | uniq -i`
`-f n`	Ignoriert erste n Felder	`sort datei.txt | uniq -f 1`
`-s n`	Ignoriert erste n Zeichen	`sort datei.txt | uniq -s 3`

1. Häufigkeiten zählen mit `-c`:

```
cat > browser.txt << EOF
Firefox
Chrome
Firefox
Safari
Chrome
Firefox
Edge
EOF
```

```
sort browser.txt | uniq -c
```

```
1 Chrome
1 Edge
3 Firefox
1 Safari
```

```
# Sortiert nach Häufigkeit (häufigste zuerst)
sort browser.txt | uniq -c | sort -nr
3 Firefox
1 Safari
1 Edge
1 Chrome
```

2. Nur Duplikate anzeigen mit `-d`:

```
sort browser.txt | uniq -d
Firefox
```

3. Nur eindeutige Zeilen anzeigen mit `-u`:

```
sort browser.txt | uniq -u
Chrome
Edge
Safari
```

Kombination mit sort für effektive Datenanalyse

Die Kombination von `sort` und `uniq` ist ein mächtiges Werkzeug für die Datenanalyse:

1. Top-10-Analyse von Logdateien:

```
# Häufigste IP-Adressen in Apache-Logs
awk '{print $1}' /var/log/apache2/access.log |
     sort | uniq -c | sort -nr | head -10

# Häufigste HTTP-Status-Codes
awk '{print $9}' /var/log/apache2/access.log |
    sort | uniq -c | sort -nr
```

2. Systembenutzer-Analyse:

```
# Häufigste Login-Shells
awk -F: '{print $7}' /etc/passwd | sort | uniq -c | sort -nr
    15 /bin/bash
     8 /usr/sbin/nologin
     3 /bin/false
     2 /bin/sync
     1 /usr/bin/git-shell
```

3. Netzwerk-Traffic-Analyse:

```
# Häufigste Ziel-Ports in netstat-Ausgabe
netstat -an | awk '{print $4}' | grep -E ':[0-9]+$' |
```

```
sed 's/.*://' | sort -n | uniq -c | sort -nr | head -10
```

Anwendungsbeispiele für Systemüberwachung

1. Überwachung fehlgeschlagener Login-Versuche:

```
# Häufigste fehlgeschlagene Login-Versuche nach IP
grep "Failed password" /var/log/auth.log |
awk '{print $11}' | sort | uniq -c | sort -nr | head -10
```

2. Analyse von Systemfehlern:

```
# Häufigste Fehlermeldungen im Syslog
grep -i error /var/log/syslog |
awk '{for(i=6;i<=NF;i++) printf "%s ", $i; print ""}' |
sort | uniq -c | sort -nr | head -5
```

3. Überwachung der Speichernutzung:

```
# Verzeichnisse mit den meisten Dateien
find /var -type f 2>/dev/null |
dirname | sort | uniq -c | sort -nr | head -10
```

```

```

4. Analyse von Konfigurationsdateien:

```
# Häufigste Konfigurationsparameter in Apache
grep -v "^#|^$" /etc/apache2/apache2.conf |
awk '{print $1}' | sort | uniq -c | sort -nr
```

> **Tipp für die Praxis:** Die Kombination sort | uniq `-c` | sort `-nr` ist so häufig, dass viele Administratoren sie als Alias definieren.

```
# In ~/.bashrc
alias frequency='sort | uniq -c | sort -nr'

# Verwendung:
awk '{print $1}' access.log | frequency | head -10
```

> ⚠ **Wichtige Überlegung zur Performance:** Bei sehr großen Dateien kann die Kombination von `sort` und `uniq` speicherintensiv sein. Für extrem große Datenmengen gibt es spezialisierte Tools oder man kann die Daten in kleinere Chunks aufteilen.

Typische Fehlerquellen:

Vergessen zu sortieren vor uniq:

```
# FALSCH - findet nicht alle Duplikate:
uniq datei.txt

# RICHTIG:
sort datei.txt | uniq
```

Verwechslung von `-d` und `-u`:

```
# -d zeigt nur Duplikate
sort datei.txt | uniq -d

# -u zeigt nur eindeutige (nicht-duplizierte) Zeilen
sort datei.txt | uniq -u
```

Nicht bedenken, dass uniq case-sensitive ist:

```
# Ohne -i werden "Test" und "test" als unterschiedlich behandelt
echo -e "TestntestnTest" | sort | uniq
test
Test

# Mit -i werden sie als identisch behandelt
echo -e "TestntestnTest" | sort | uniq -i
Test
```

```

```

Erweiterte Kombinationen von sort und uniq

Die wahre Stärke von `sort` und `uniq` zeigt sich in komplexen Datenverarbeitungspipelines:

Komplexe Log-Analyse:

```
# Analysiere Apache-Logs: Finde die Top-10 User-Agents
awk -F'"' '{print $6}' /var/log/apache2/access.log |
  sort | uniq -c | sort -nr | head -10

# Analysiere Fehlerverteilung nach Stunden
grep "$(date +%d/%b/%Y)" /var/log/apache2/access.log |
  awk '{print substr($4,14,2)}' | sort | uniq -c | sort -k2n
```

Systemressourcen-Überwachung:

```
# Finde Prozesse, die am häufigsten gestartet werden
ps aux | awk '{print $11}' | sort | uniq -c | sort -nr | head -10

# Analysiere Speicherverbrauch nach Prozessname
ps aux | awk '{print $11, $4}' | sort |
  awk '{proc[$1]+=$2} END {for(p in proc) print proc[p], p}' | sort -nr
```

Netzwerk-Sicherheitsanalyse:

```
# Finde verdächtige Verbindungsversuche
grep "Failed password" /var/log/auth.log |
  awk '{print $11}' | sort | uniq -c |
  awk '$1 > 10 {print $1, $2}' | sort -nr
```

Die Befehle `sort` und `uniq` sind fundamentale Werkzeuge für jeden Linux-Administrator. Ihre Beherrschung ermöglicht es, aus chaotischen Datenmengen strukturierte Informationen zu extrahieren und ist essentiell für die LPIC-1-Zertifizierung. Die Kombination beider Befehle in Pipelines mit anderen Tools wie `awk`, `grep` und `cut` eröffnet nahezu unbegrenzte Möglichkeiten für die Datenanalyse und Systemüberwachung.

Textdateien zählen und formatieren

Die Analyse und Formatierung von Textdateien gehört zu den täglichen Aufgaben eines Linux-Administrators. Ob es darum geht, die Größe von Logdateien zu überwachen, Berichte zu erstellen oder Daten für die weitere Verarbeitung aufzubereiten – die in diesem Abschnitt behandelten Befehle `wc`, `fmt`, `nl` und `cut` sind unverzichtbare Werkzeuge für diese Aufgaben. Sie ergänzen die bereits behandelten Textverarbeitungstools perfekt und ermöglichen präzise Statistiken sowie professionelle Formatierungen.

wc - Wörter, Zeilen und Zeichen zählen

Was ist wc und warum ist es fundamental wichtig?

Der Befehl `wc` (word count) ist eines der grundlegendsten und am häufigsten verwendeten Werkzeuge zur Textanalyse in Linux-Systemen. Er zählt Zeilen, Wörter, Zeichen und Bytes in Textdateien oder Datenströmen und liefert damit essenzielle Statistiken für die Systemadministration.

Die Stärke von `wc` liegt in seiner Einfachheit und Vielseitigkeit: Es kann sowohl für schnelle Überprüfungen als auch für komplexe Datenanalysen in Pipelines verwendet werden.

Quelle: https://admindocs.de/de/lpic-1-serie/lpic-1-textverarbeitung-mit-shell-kommandos

Grundfunktionen von wc (`-l` , `-w` , `-c` , `-m`)

Die grundlegende Syntax von `wc` ist intuitiv:

```
wc [Optionen] datei(en)
```

Ohne Optionen gibt `wc` drei Werte aus: Anzahl der Zeilen, Wörter und Bytes:

```
wc beispiel.txt 15 42 256 beispiel.txt
```

```
wc beispiel.txt
    15    42   256 beispiel.txt
```

Diese Ausgabe bedeutet: 15 Zeilen, 42 Wörter, 256 Bytes.

Die wichtigsten wc-Optionen:

Option	Beschreibung	Praktisches Beispiel
`-l`	Zählt nur Zeilen (lines)	`wc -l /var/log/syslog`
`-w`	Zählt nur Wörter (words)	`wc -w dokument.txt`

Option	Beschreibung	Praktisches Beispiel
`-c`	Zählt nur Bytes	`wc -c binary_file.dat`
`-m`	Zählt nur Zeichen (characters)	`wc -m unicode_text.txt`
`-L`	Zeigt die Länge der längsten Zeile	`wc -L konfiguration.conf`

⚠ **Wichtiger Unterschied zwischen `-c` und `-m`:** In UTF-8-kodierten Dateien können Zeichen mehrere Bytes umfassen. Während `-c` die Bytes zählt, zählt `-m` die tatsächlichen Zeichen:

```
echo "Café" | wc -c
5
echo "Café" | wc -m
4
```

Das "é" benötigt in UTF-8 zwei Bytes, wird aber als ein Zeichen gezählt.

□□ Praktische Anwendungsbeispiele für Systemstatistiken:

Logdatei-Überwachung:

```
# Anzahl der Zeilen in einer Logdatei
wc -l /var/log/apache2/access.log
45823 /var/log/apache2/access.log

# Überwachung des Logwachstums
watch "wc -l /var/log/syslog"
```

Systembenutzer zählen:

```
# Anzahl der Systembenutzer
wc -l /etc/passwd
42 /etc/passwd

# Anzahl der normalen Benutzer (UID >= 1000)
awk -F: '$3 >= 1000 {print}' /etc/passwd | wc -l
```

Konfigurationsdateien analysieren:

```
# Aktive Konfigurationszeilen (ohne Kommentare und leere Zeilen)
grep -v "^s*#|^s*$" /etc/apache2/apache2.conf | wc -l

# Durchschnittliche Zeilenlänge in einer Konfigurationsdatei
awk '{total += length($0); count++} END {print total/count}' /etc/ssh/sshd_config
```

Kombination mit anderen Befehlen für komplexe Analysen

Die wahre Stärke von `wc` zeigt sich in der Kombination mit anderen Befehlen über Pipes:

Prozessanalyse:

```
# Anzahl laufender Prozesse
ps aux | wc -l

# Anzahl Apache-Prozesse
ps aux | grep apache | grep -v grep | wc -l
```

```
# Speicherverbrauch aller Prozesse (vereinfacht)
ps aux | awk 'NR>1 {sum+=$4} END {print sum "%"}'
```

Netzwerk-Monitoring:

```
# Anzahl aktiver Verbindungen
netstat -an | grep ESTABLISHED | wc -l

# Anzahl lauschender Ports
netstat -ln | grep LISTEN | wc -l
```

Dateisystem-Analyse:

```
# Anzahl Dateien in einem Verzeichnis (rekursiv)
find /var/log -type f | wc -l

# Anzahl Verzeichnisse
find /etc -type d | wc -l

# Größte Dateien nach Zeilenzahl
find /var/log -name "*.log" -exec wc -l {} ; | sort -nr | head -5
```

Log-Analyse:

```
# Fehlerrate in Logdateien
grep -c "ERROR" /var/log/application.log

# Verhältnis von Erfolg zu Fehler in Apache-Logs
awk '$9 ~ /^2/ {success++} $9 ~ /^[^45]/ {error++} END {print "Erfolg:", success,
"Fehler:", error}' /var/log/apache2/access.log
```

Tipp für Monitoring-Skripte: `wc` eignet sich hervorragend für einfache Monitoring-Skripte:

```
#!/bin/bash
# Einfaches Log-Monitoring

LOGFILE="/var/log/application.log"
THRESHOLD=1000

# Aktuelle Zeilenzahl
CURRENT_LINES=$(wc -l < "$LOGFILE")

if [ "$CURRENT_LINES" -gt "$THRESHOLD" ]; then
    echo "WARNUNG: Logdatei hat $CURRENT_LINES Zeilen (Schwellwert: $THRESHOLD)"
    # Hier könnte eine Benachrichtigung oder Rotation ausgelöst werden
fi
```

fmt - Textformatierung und Zeilenumbruch

Was ist fmt und wann wird es benötigt?

Der Befehl `fmt` (format) ist ein Textformatierer, der lange Zeilen umbrechen und Text in einheitliche Absätze umwandeln kann. Er ist besonders nützlich für die Aufbereitung von Dokumentation, E-Mails oder anderen Texten, die eine konsistente Formatierung benötigen.

`fmt` folgt intelligenten Regeln beim Umbrechen: Es versucht, Wörter nicht zu trennen und berücksichtigt Satzzeichen und Leerzeichen.

Automatische Textformatierung

Die grundlegende Verwendung von `fmt` ist einfach:

```
fmt datei.txt
```

Standardmäßig formatiert `fmt` Text auf eine Zeilenlänge von 75 Zeichen.

□□ Praktisches Beispiel:

```
# Erstelle eine Datei mit langen Zeilen
cat > langer_text.txt << 'EOF'
Dies ist ein sehr langer Text, der über mehrere Zeilen gehen sollte, aber in einer
einzigen langen Zeile geschrieben wurde, was die Lesbarkeit erheblich beeinträchtigt.
EOF
```

Formatiere den Text:

```
fmt langer_text.txt
Dies ist ein sehr langer Text, der über mehrere Zeilen gehen sollte,
aber in einer einzigen langen Zeile geschrieben wurde, was die
Lesbarkeit erheblich beeinträchtigt.
```

Wichtige Optionen für Zeilenlänge und Einrückung

Option	Beschreibung	Beispiel
`-w n`	Setzt maximale Zeilenlänge auf n Zeichen	`fmt -w 60 datei.txt`
`-s`	Teilt nur lange Zeilen, fügt kurze nicht zusammen	`fmt -s datei.txt`
`-u`	Einheitliche Leerzeichen (ein Leerzeichen zwischen Wörtern)	`fmt -u datei.txt`
`-p PREFIX`	Formatiert nur Zeilen mit dem angegebenen Präfix	`fmt -p "> " email.txt`
`-t`	Behält Einrückung der ersten Zeile bei	`fmt -t liste.txt`

□□ Praktische Anwendungsfälle:

E-Mail-Formatierung:

```
# Formatiere eine E-Mail mit zitierten Zeilen
fmt -p "> " -w 72 email.txt
```

Code-Kommentare formatieren:

```
# Formatiere Kommentare in einer Konfigurationsdatei
fmt -p "# " -w 80 config.conf
```

```

```

Dokumentation aufbereiten:

```
# Formatiere README-Dateien
fmt -w 80 README.txt > README_formatted.txt
```

Anwendungsfälle für Dokumentation und E-Mails

Automatische Dokumentationsformatierung:

```
#!/bin/bash
# Skript zur Formatierung von Markdown-Dateien

for file in *.md; do
    # Formatiere nur Textabsätze, nicht Code-Blöcke
    sed '/``````/!s/.*/fmt -w 80 <<< "&"/e' "$file" > "${file}.formatted"
done
```

E-Mail-Template-Erstellung:

```
# Erstelle formatierte E-Mail-Templates
cat > email_template.txt << 'EOF'
```

```
Sehr geehrte Damen und Herren, hiermit möchten wir Sie über wichtige
Systemwartungsarbeiten informieren, die am kommenden Wochenende durchgeführt werden.
EOF

fmt -w 72 email_template.txt
```

> **Tipp für die Praxis:** fmt kann in Vim/Emacs integriert werden, um Text direkt im Editor zu formatieren. In Vim: :%!fmt `-w` 80

nl - Zeilennummerierung

Was ist nl und wofür wird es verwendet?

Der Befehl `nl` (number lines) fügt Zeilennummern zu Textdateien hinzu. Im Gegensatz zu einfachen Lösungen wie `cat -n` bietet `nl` erweiterte Optionen für die Nummerierung und kann verschiedene Nummerierungsstile anwenden.

Verschiedene Nummerierungsoptionen

Die grundlegende Syntax:

```
nl datei.txt
```

Wichtige nl-Optionen:

Option	Beschreibung	Beispiel
`-b a`	Nummeriert alle Zeilen	`nl -b a datei.txt`

Option	Beschreibung	Beispiel
`-b t`	Nummeriert nur nicht-leere Zeilen (Standard)	`nl -b t datei.txt`
`-b n`	Keine Nummerierung	`nl -b n datei.txt`
`-n ln`	Linksbündige Nummerierung	`nl -n ln datei.txt`
`-n rn`	Rechtsbündige Nummerierung	`nl -n rn datei.txt`
`-n rz`	Rechtsbündig mit führenden Nullen	`nl -n rz datei.txt`
`-w n`	Breite des Nummerierungsfeldes	`nl -w 4 datei.txt`
`-s STRING`	Trennzeichen zwischen Nummer und Text	`nl -s ": " datei.txt`
`-v n`	Startnummer	`nl -v 100 datei.txt`
`-i n`	Inkrement (Schrittweite)	`nl -i 5 datei.txt`

□□ Praktische Beispiele:

a) Code-Dateien nummerieren:

Python

```
# Python-Skript mit Zeilennummern
nl -b a -w 3 -s " | " script.py
  1 | #!/usr/bin/env python3
  2 | import sys
  3 |
  4 | def main():
  5 |     print("Hello World")
```

b) Konfigurationsdateien für Debugging:

```
# Apache-Konfiguration mit Zeilennummern für Fehlersuche
nl -b t -w 4 -s ": " /etc/apache2/apache2.conf
```

Praktische Anwendung für Code und Konfigurationsdateien

Fehleranalyse in Konfigurationsdateien:

```
# Zeige Konfigurationsdatei mit Zeilennummern für Fehlermeldungen
nl -b a -w 3 /etc/ssh/sshd_config | grep -A 2 -B 2 "Port"
```

Log-Analyse mit Zeilennummern:

```
# Füge Zeilennummern zu Logdateien für bessere Referenzierung hinzu
nl -b a /var/log/apache2/error.log | tail -20
```

Dokumentation erstellen:

```
# Erstelle nummerierte Checklisten
```

```
nl -b a -s ". " checkliste.txt
```

cut - Spalten und Felder extrahieren

Was ist cut und warum ist es unverzichtbar?

Der Befehl `cut` extrahiert bestimmte Spalten oder Felder aus strukturierten Textdateien. Er ist besonders wertvoll für die Arbeit mit CSV-Dateien, Logdateien oder anderen spaltenbasierten Daten und ermöglicht es, gezielt bestimmte Informationen zu extrahieren.

Extraktion bestimmter Spalten aus Textdateien

Die grundlegende Syntax von `cut`:

```
cut [Optionen] datei
```

Wichtige cut-Optionen:

Option	Beschreibung	Beispiel
`-d CHAR`	Setzt Feldtrenner (delimiter)	`cut -d: -f1 /etc/passwd`
`-f LIST`	Wählt Felder aus	`cut -f1,3,5 tabelle.txt`
`-c LIST`	Wählt Zeichen-Positionen aus	`cut -c1-10 datei.txt`
`-b LIST`	Wählt Byte-Positionen aus	`cut -b1-5 datei.txt`
`--complement`	Invertiert die Auswahl	`cut -d: -f1 --complement /etc/passwd`

Option	Beschreibung	Beispiel
`--output-delimiter=STRING`	Setzt Ausgabe-Trennzeichen	`cut -d: -f1,3 --output-delimiter=" " /etc/passwd`

Wichtige Optionen (`-d`, `-f`, `-c`)

Feldbasierte Extraktion mit `-d` und `-f`:

```
# Extrahiere Benutzernamen aus /etc/passwd
cut -d: -f1 /etc/passwd | head -5
root
daemon
bin
sys
sync

# Extrahiere Benutzername und UID
cut -d: -f1,3 /etc/passwd | head -5
root:0
daemon:1
bin:2
sys:3
sync:4
```

Zeichenbasierte Extraktion mit `-c`:

```
# Extrahiere die ersten 10 Zeichen jeder Zeile
cut -c1-10 /var/log/syslog | head -5

# Extrahiere Zeichen 1-3 und 8-12
cut -c1-3,8-12 datei.txt
```

Bereiche und Listen:

```
# Verschiedene Bereichsnotationen
cut -d: -f1-3 /etc/passwd    # Felder 1 bis 3
cut -d: -f1,3,5- /etc/passwd # Felder 1, 3 und ab 5
cut -d: -f-3 /etc/passwd     # Felder vom Anfang bis 3
```

Praktische Anwendungen für CSV-Dateien und strukturierte Logs

CSV-Datenanalyse:

```
# Beispiel-CSV erstellen
cat > verkaufsdaten.csv << 'EOF'
Datum,Produkt,Menge,Preis
2023-05-01,Laptop,5,999.99
2023-05-01,Maus,20,29.99
2023-05-02,Tastatur,15,79.99
EOF

# Extrahiere nur Produkt und Preis
cut -d, -f2,4 verkaufsdaten.csv
Produkt,Preis
Laptop,999.99
Maus,29.99
Tastatur,79.99

# Berechne Gesamtumsatz (vereinfacht)
cut -d, -f3,4 verkaufsdaten.csv | tail -n +2 | awk -F, '{sum += $1 * $2} END {print
```

```
    "Gesamtumsatz:", sum}'
```

Apache-Log-Analyse:

```
# Extrahiere IP-Adressen und HTTP-Status-Codes
cut -d' ' -f1,9 /var/log/apache2/access.log | head -5

# Extrahiere nur die Uhrzeit aus Logeinträgen
cut -d' ' -f4 /var/log/apache2/access.log | cut -d: -f2-4 | head -5
```

Systemkonfiguration analysieren:

```
# Extrahiere alle verwendeten Shells
cut -d: -f7 /etc/passwd | sort | uniq -c

# Zeige Benutzer mit spezifischen UIDs
cut -d: -f1,3 /etc/passwd | awk -F: '$2 >= 1000 {print $1}'
```

Kombination mit anderen Befehlen für Datenextraktion

Komplexe Datenverarbeitung:

```
# Analysiere Prozess-Speicherverbrauch
ps aux | cut -c12-15,65- | sort -nr | head -10

# Extrahiere und analysiere Netzwerk-Ports
netstat -an | grep LISTEN | cut -d' ' -f4 | cut -d: -f2 | sort -n | uniq -c
```

Log-Rotation und `-Analyse`:

```
# Extrahiere Zeitstempel für Rotation
cut -d' ' -f1-3 /var/log/syslog | tail -1

# Analysiere Fehlerverteilung nach Stunden
grep "ERROR" /var/log/application.log | cut -d' ' -f2 | cut -d: -f1 | sort | uniq -c
```

Konfigurationsmanagement:

```
# Extrahiere aktive Apache-Module
apache2ctl -M | cut -d' ' -f1 | sort

# Analysiere SSH-Konfiguration
grep -v "^#|^$" /etc/ssh/sshd_config | cut -d' ' -f1 | sort | uniq -c
```

□□ **Tipp für komplexe Extraktion:** Bei komplexen Datenstrukturen kann awk flexibler sein als cut, aber cut ist oft schneller und einfacher für einfache Spaltenextraktionen.

⚠ **Wichtiger Hinweis:** cut kann nur mit konsistenten Trennzeichen arbeiten. Bei unregelmäßigen Leerzeichen ist awk oft die bessere Wahl.

- □ **Typische Fehlerquellen:

Falsche Feldnummerierung:

- cut beginnt bei 1, nicht bei 0

Inkonsistente Trennzeichen:

- Mischung aus Tabs und Leerzeichen

Vergessen des Trennzeichens:

- Ohne `-d` verwendet cut Tabs als Standard

Die Befehle `wc`, `fmt`, `nl` und `cut` erweitern dein Toolkit um wichtige Funktionen für Statistiken, Formatierung und Datenextraktion. Sie sind essentiell für die tägliche Arbeit als Linux-Administrator und wichtige Bestandteile der LPIC-1-Prüfung. Die Kombination dieser Tools mit den bereits behandelten Befehlen ermöglicht es, nahezu jede Textverarbeitungsaufgabe effizient zu lösen.

Reguläre Ausdrücke - Grundlagen

Reguläre Ausdrücke (Regular Expressions, kurz RegEx) sind eine der mächtigsten und gleichzeitig am meisten gefürchteten Technologien in der Linux-Welt. Sie ermöglichen es, komplexe Textmuster zu beschreiben und zu finden, wodurch sie zu einem unverzichtbaren Werkzeug für jeden Linux-Administrator werden. Obwohl ihre Syntax zunächst kryptisch erscheinen mag, eröffnen reguläre Ausdrücke nach dem Erlernen der Grundlagen völlig neue

Dimensionen der Textverarbeitung.

Konzept und Bedeutung für die Textverarbeitung

Reguläre Ausdrücke sind eine formale Sprache zur Beschreibung von Zeichenkettenmustern[^2]. Sie dienen dazu, Texte nach bestimmten Mustern zu durchsuchen, zu analysieren und zu manipulieren, wobei sie weit über einfache Textsuche hinausgehen. Während eine normale Suche nur nach exakten Übereinstimmungen sucht, können reguläre Ausdrücke flexible Muster definieren, die verschiedene Variationen eines Suchbegriffs erfassen.

□□ Praktisches Beispiel:

Anstatt einzeln nach "`Error`", "`ERROR`", "`error`" und "`Err`" zu suchen, kann ein regulärer Ausdruck alle diese Varianten mit einem einzigen Muster erfassen:

```
grep -E u0022[Ee]rr(or)?u0022 /var/log/syslog
```

Die Stärke regulärer Ausdrücke liegt in ihrer Fähigkeit, abstrakte Muster zu beschreiben:

- Alle E-Mail-Adressen in einem Text finden
- IP-Adressen in Logdateien identifizieren
- Datumswerte in verschiedenen Formaten erkennen
- Konfigurationsparameter mit bestimmten Strukturen extrahieren

Unterschiede zwischen verschiedenen Regex-Varianten (BRE, ERE)

Linux verwendet hauptsächlich zwei Varianten regulärer Ausdrücke, die sich in ihrer Syntax und ihrem Funktionsumfang unterscheiden:

Basic Regular Expressions (BRE):

- Standardformat in traditionellen Unix-Tools wie `grep` und `sed`

- Konservativere Syntax mit weniger Metazeichen
- Bestimmte Zeichen müssen "escaped" werden, um ihre spezielle Bedeutung zu erhalten

Extended Regular Expressions (ERE):

- Erweiterte Syntax mit mehr Funktionen
- Verfügbar über grep `-E`, egrep, awk und moderne Tools
- Intuitivere Syntax für komplexe Ausdrücke

Feature	BRE (Basic)	ERE (Extended)
Gruppierung	`$pattern$`	`(pattern)`
Oder-Operator	Nicht verfügbar	`pattern1|pattern2`
Plus-Quantifizierer	Nicht verfügbar	`pattern+`
Fragezeichen-Quantifizierer	Nicht verfügbar	`pattern?`
Geschweifte Klammern	`{n,m}`	`{n,m}`

□□ Vergleichsbeispiel:

```
# BRE-Syntax (traditionelles grep)
grep 'colou?r' datei.txt          # Funktioniert nicht in BRE
grep 'colou*r' datei.txt           # Sucht nach "colo" + beliebig viele "u" + "r"
```

Wo werden reguläre Ausdrücke in Linux verwendet?

Reguläre Ausdrücke sind in nahezu allen wichtigen Linux-Tools integriert:

Kommandozeilen-Tools:

- `grep`, `egrep`, `fgrep` - Textsuche

- `sed` - Stream-Editing und Textersetzung
- `awk` - Strukturierte Textverarbeitung
- `find` - Dateisuche mit Namensmustern
- `less`, `more` - Suche in Pagern

Editoren:

- `vim`, `emacs` - Suchen und Ersetzen
- `nano` - Grundlegende Regex-Unterstützung

Programmiersprachen und Skripte:

- Bash, Python, Perl, PHP - Eingebaute Regex-Unterstützung
- Shell-Skripte für Systemadministration

Systemkonfiguration:

- Apache, Nginx - URL-Rewriting und Konfiguration
- Logrotate - Dateinamensmuster
- Firewall-Regeln - Musterbasierte Filterung

> ⚠ **Wichtiger Hinweis:** Nicht alle Tools verwenden dieselbe Regex-Syntax. Es ist wichtig zu wissen, welche Variante ein bestimmtes Tool unterstützt.

Grundlegende Regex-Syntax

Literale Zeichen und Metazeichen

Reguläre Ausdrücke bestehen aus zwei Arten von Zeichen:

Literale Zeichen:

- Normale Buchstaben, Zahlen und die meisten Sonderzeichen
- Repräsentieren sich selbst
- Beispiel: "abc" findet exakt die Zeichenfolge "abc"

Metazeichen:

- Spezielle Zeichen mit besonderer Bedeutung
- Steuern das Verhalten des regulären Ausdrucks
- Grundlegende Metazeichen: ". ^ $ * + ? [ ] { } ( ) | "

□□ Praktische Beispiele für Metazeichen:

Der Punkt (.) - Beliebiges Zeichen:

```
echo -e "catncarncup" | grep "ca."
cat
car
```

Das Caret (^) - Zeilenanfang:

```
grep "^root" /etc/passwd
root:x:0:0:root:/root:/bin/bash
```

Das Dollar-Zeichen ($) - Zeilenende:

```
grep "bash$" /etc/passwd
root:x:0:0:root:/root:/bin/bash
username:x:1000:1000:User:/home/username:/bin/bash
```

Escaping von Metazeichen: Wenn du nach einem Metazeichen als literales Zeichen suchen möchtest, musst du es mit einem Backslash "escapen":

```
# Suche nach einem wörtlichen Punkt
grep "192.168.1.1" /etc/hosts

# Suche nach einem Dollar-Zeichen
grep "$HOME" script.sh
```

Zeichenklassen und Bereiche ([a-z], [0-9], d, w)

Zeichenklassen ermöglichen es, eine Gruppe von Zeichen zu definieren, von denen eines an einer bestimmten Position stehen kann.

Grundlegende Zeichenklassen:

```
# Einzelne Zeichen
[abc]           # Entspricht 'a', 'b' oder 'c'
[aeiou]         # Entspricht jedem Vokal

# Bereiche
[a-z]           # Jeder Kleinbuchstabe
[A-Z]           # Jeder Großbuchstabe
[0-9]           # Jede Ziffer
[a-zA-Z0-9]     # Alphanumerische Zeichen
```

Negierte Zeichenklassen:

```
[^0-9]          # Alles außer Ziffern
[^aeiou]        # Alle Konsonanten
[^a-zA-Z]       # Alle Nicht-Buchstaben
```

□□ Praktische Anwendungsbeispiele:

IP-Adressen finden:

```
grep -E "[0-9]{1,3}.[0-9]{1,3}.[0-9]{1,3}.[0-9]{1,3}" /var/log/apache2/access.log
```

E-Mail-Adressen (vereinfacht):

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

Hexadezimale Werte:

```
grep -E "[0-9a-fA-F]+" system.log
```



Vordefinierte Zeichenklassen (in erweiterten Regex-Engines):

Klasse	Bedeutung	Äquivalent
`d`	Ziffer	`[0-9]`
`w`	Wortzeichen	`[a-zA-Z0-9_]`
`s`	Leerzeichen	`[ tnrf]`
`D`	Keine Ziffer	`[^0-9]`
`W`	Kein Wortzeichen	`[^a-zA-Z0-9_]`
`S`	Kein Leerzeichen	`[^ tnrf]`

> ⚠ **Beachte:** Diese vordefinierten Klassen sind nicht in allen Linux-Tools verfügbar. grep und sed unterstützen sie standardmäßig nicht, während awk und moderne Programmiersprachen sie verwenden.

Quantifizierer (*, +, ?, {n,m})

Quantifizierer bestimmen, wie oft ein Zeichen oder eine Gruppe von Zeichen wiederholt werden kann:

Quantifizierer	Bedeutung	Beispiel
`*`	Null oder mehr	`ab*` findet "a", "ab", "abb", "abbb"
`+`	Ein oder mehr (nur ERE)	`ab+` findet "ab", "abb", "abbb"
`?`	Null oder eins (nur ERE)	`ab?` findet "a", "ab"
`{n}`	Genau n mal	`a{3}` findet "aaa"

Quantifizierer	Bedeutung	Beispiel
`{n,}`	Mindestens n mal	`a{2,}` findet "aa", "aaa", "aaaa"
`{n,m}`	Zwischen n und m mal	`a{2,4}` findet "aa", "aaa", "aaaa"

□□ Praktische Beispiele:

Logdatei-Analyse:

```
# Finde Zeilen mit mehreren aufeinanderfolgenden Leerzeichen
grep -E " {2,}" /var/log/syslog

# Finde IP-Adressen mit optionalen führenden Nullen
grep -E "[0-9]{1,3}.[0-9]{1,3}.[0-9]{1,3}.[0-9]{1,3}" access.log
```

Konfigurationsdateien validieren:

```
# Finde Zeilen mit mindestens einem Gleichheitszeichen
grep -E ".+=.+" config.conf

# Finde Kommentarzeilen (optional mit Leerzeichen vor #)
grep -E "^ *#" apache2.conf
```

Systemüberwachung:

```
# Finde Prozesse mit PID-Mustern
ps aux | grep -E "[0-9]{4,}"  # PIDs mit mindestens 4 Ziffern
```

Anker (^, $) für Zeilenanfang und `-ende`

Anker definieren Positionen im Text, nicht Zeichen:

Zeilenanfang (^):

```
# Finde Zeilen, die mit "Error" beginnen
grep "^Error" /var/log/application.log

# Finde Konfigurationszeilen, die nicht mit # beginnen
grep "^[^#]" /etc/ssh/sshd_config
```

Zeilenende ($):

```
# Finde Zeilen, die mit einem Punkt enden
grep ".$" dokument.txt

# Finde Shell-Skripte (Zeilen, die mit .sh enden)
ls -la | grep ".sh$"
```

Kombination von Ankern:

```
# Finde leere Zeilen
grep "^$" datei.txt

# Finde Zeilen, die nur aus Leerzeichen bestehen
grep "^ *$" datei.txt

# Finde Zeilen mit genau einem Wort
grep "^[a-zA-Z]*$" wortliste.txt
```

Praktische Anwendung regulärer Ausdrücke

Verwendung mit grep für erweiterte Suchen

`grep` ist das primäre Tool für die Anwendung regulärer Ausdrücke in der Textsuche:

Grundlegende grep-Regex-Optionen:

```
grep "pattern" datei          # BRE (Basic Regular Expressions)
grep -E "pattern" datei       # ERE (Extended Regular Expressions)
grep -P "pattern" datei       # PCRE (Perl Compatible, wenn verfügbar)
grep -F "pattern" datei       # Feste Strings (keine Regex)
```

□□ Erweiterte Suchbeispiele:

Log-Analyse mit komplexen Mustern:

```
# Finde alle HTTP-Fehler (4xx und 5xx Status-Codes)
grep -E " [45][0-9]{2} " /var/log/apache2/access.log

# Finde Zeitstempel in verschiedenen Formaten
grep -E "[0-9]{4}-[0-9]{2}-[0-9]{2}|[0-9]{2}/[0-9]{2}/[0-9]{4}" logfile.txt

# Finde kritische Systemereignisse
grep -E "(CRITICAL|FATAL|EMERGENCY)" /var/log/syslog
```

Netzwerk- und Sicherheitsanalyse:

```
# Finde verdächtige IP-Adressen (private Bereiche ausschließen)
grep -E "([0-9]{1,3}.){3}[0-9]{1,3}" access.log | grep -vE "^(10.|172.(1[6-9]|2[0-
9]|3[01]).|192.168.)"

# Finde fehlgeschlagene SSH-Logins mit IP-Adressen
grep -E "Failed password.*from ([0-9]{1,3}.){3}[0-9]{1,3}" /var/log/auth.log
```

Einsatz in `sed` für komplexe Ersetzungen

`sed` verwendet reguläre Ausdrücke sowohl für die Suche als auch für die Ersetzung:

Erweiterte sed-Operationen:

```
# Ersetze alle IP-Adressen durch "XXX.XXX.XXX.XXX"
sed -E 's/([0-9]{1,3}.){3}[0-9]{1,3}/XXX.XXX.XXX.XXX/g' logfile.txt
```

```
# Extrahiere Domain aus E-Mail-Adressen
sed -E 's/.*@([a-zA-Z0-9.-]+.[a-zA-Z]{2,}).*/1/' email_liste.txt

# Formatiere Datumsangaben um (YYYY-MM-DD zu DD.MM.YYYY)
sed -E 's/([0-9]{4})-([0-9]{2})-([0-9]{2})/3.2.1/g' datei.txt
```

Verwendung von Rückverweisen (Backreferences):

```
# Vertausche Vor- und Nachname
sed -E 's/([A-Za-z]+) ([A-Za-z]+)/2, 1/' namen.txt

# Füge Anführungszeichen um Wörter hinzu
sed -E 's/([a-zA-Z]+)/"1"/g' wortliste.txt
```

Typische Muster für Systemadministration

IP-Adressen und Netzwerk:

```
# IPv4-Adresse (vereinfacht)
^([0-9]{1,3}.){3}[0-9]{1,3}$

# IPv4 mit Portangabe
^([0-9]{1,3}.){3}[0-9]{1,3}:[0-9]{1,5}$

# MAC-Adresse
^([0-9a-fA-F]{2}:){5}[0-9a-fA-F]{2}$
```

E-Mail-Adressen:

```
# Einfache E-Mail-Validierung
^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+.[a-zA-Z]{2,}$

# E-Mail-Extraktion aus Text
[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+.[a-zA-Z]{2,}
```

Datumsformate:

```
# ISO-Datum (YYYY-MM-DD)
^[0-9]{4}-[0-9]{2}-[0-9]{2}$

# Deutsches Datum (DD.MM.YYYY)
^[0-9]{2}.[0-9]{2}.[0-9]{4}$

# Amerikanisches Datum (MM/DD/YYYY)
^[0-9]{2}/[0-9]{2}/[0-9]{4}$
```

Systemspezifische Muster:

```
# Linux-Benutzername
^[a-z_][a-z0-9_-]*$

# Absolute Pfade
^/([a-zA-Z0-9._-]+/?)*$
```

```
# Prozess-IDs
^[0-9]+$
```

□□ Praktisches Beispiel - Komplette Log-Analyse:

```
#!/bin/bash
# Umfassendes Log-Analyse-Skript mit Regex

LOGFILE="/var/log/apache2/access.log"

echo "=== Apache Log-Analyse ==="

# 1. Eindeutige IP-Adressen zählen
echo "Eindeutige Besucher:"
grep -oE "([0-9]{1,3}.){3}[0-9]{1,3}" "$LOGFILE" | sort | uniq | wc -l

# 2. Top 10 IP-Adressen
echo -e "nTop 10 IP-Adressen:"
grep -oE "([0-9]{1,3}.){3}[0-9]{1,3}" "$LOGFILE" | sort | uniq -c | sort -nr | head -10

# 3. HTTP-Fehler analysieren
echo -e "nHTTP-Fehler (4xx und 5xx):"
grep -E " [45][0-9]{2} " "$LOGFILE" | awk '{print $9}' | sort | uniq -c | sort -nr

# 4. Verdächtige User-Agents finden
echo -e "nVerdächtige User-Agents (Bot-Patterns):"
grep -iE "(bot|crawler|spider|scraper)" "$LOGFILE" | awk -F'"' '{print $6}' | sort | uniq
-c | sort -nr | head -5
```

Häufige Fallstricke und deren Vermeidung

□ Typische Regex-Fehler:

a) Vergessen des Escapings:

```
# FALSCH - Punkt wird als "beliebiges Zeichen" interpretiert
grep "192.168.1.1" /etc/hosts

# RICHTIG - Punkt wird als literaler Punkt behandelt
grep "192.168.1.1" /etc/hosts
```

b) Gierige vs. nicht-gierige Quantifizierer:

```
# Gierig - findet den längsten möglichen Match
echo '<tag>content</tag>' | sed 's/<.*>/REPLACED/'
# Ergebnis: REPLACED

# Lösung: Spezifischeres Muster verwenden
echo '<tag>content</tag>' | sed 's/<[^>]*>/REPLACED/g'
# Ergebnis: REPLACEDcontent</tag>
```

c) Verwechslung von `BRE` und `ERE`:

```
# BRE (funktioniert nicht wie erwartet)
grep 'error+' logfile.txt        # Sucht nach "error" gefolgt von "+"

# ERE (korrekt)
grep -E 'error+' logfile.txt     # Sucht nach "erro" + ein oder mehr "r"
```

d) Anker-Missverständnisse:

```
# FALSCH - ^ und $ beziehen sich auf die ganze Zeile
echo "start middle end" | grep "^middle$"  # Findet nichts

# RICHTIG - Für Teilstrings keine Anker verwenden
echo "start middle end" | grep "middle"    # Findet "middle"
```

Tipps für effektive Regex-Nutzung:

- **Teste deine Regex schrittweise:** Beginne mit einfachen Mustern und erweitere sie graduell
- **Verwende Online-Regex-Tester:** Tools wie regex101.com helfen beim interaktiven Debuggen
- **Dokumentiere komplexe Ausdrücke:** Füge in Skripten Kommentare hinzu, um die Suchlogik zu erklären
- **Bevorzuge Lesbarkeit:** Mehrere einfache Filter in einer Pipeline sind oft verständlicher und robuster als ein überladener Regex

> ⚠ **Performance-Überlegungen:** Sehr komplexe reguläre Ausdrücke können langsam sein, besonders bei großen Dateien. In solchen Fällen können mehrere einfache grep-Befehle in einer Pipeline effizienter sein.

Reguläre Ausdrücke sind ein mächtiges Werkzeug, das mit Übung und Erfahrung immer natürlicher wird. Sie sind essentiell für die LPIC-1-Zertifizierung und unverzichtbar für effiziente Systemadministration. Die Investition in das Erlernen der Grundlagen zahlt sich durch erhebliche Zeitersparnis und erweiterte Möglichkeiten bei der Textverarbeitung aus.

> **Tipp für die LPIC-1-Prüfung:** Für die LPIC-1-Prüfung ist nicht das Auswendiglernen, sondern das Verständnis und die praktische Anwendung der Textverarbeitungsbefehle entscheidend. Übe intensiv mit Tools wie `grep`, `sed`, `awk`, `sort` und `cut` an echten Beispielen und arbeite regelmäßig mit den Manpages.
>
> Viele Prüfungsfragen testen typische Aufrufoptionen, Regex-Grundlagen und Kombinationsmöglichkeiten – praktische Erfahrung hilft dir, auch Detailfragen sicher zu beantworten.

Warum ist das für die LPIC-1 wichtig?

Die sichere Beherrschung von Shell-basierten Textverarbeitungswerkzeugen ist ein zentrales Prüfungsziel der LPIC-1-Zertifizierung. Viele Aufgaben in der Prüfung – und im Alltag als Administrator – erfordern, dass du große Mengen an System- und Logdaten schnell analysieren, filtern und gezielt bearbeiten kannst. Die Fähigkeit, Werkzeuge wie `grep`, `sed`, `awk`, `sort` oder `cut` gezielt einzusetzen, bildet die Grundlage für effiziente Fehleranalyse, Systemüberwachung und Automatisierung. Ohne diese Kenntnisse ist eine nachhaltige Linux-Administration kaum möglich.

> **Hinweis:** Die Ressourcen und Prüfungsinformationen findest du in unserem Grundlagen-Modul: LPIC-1: Grundlegende Navigation und Dateisystembefehle.

Befehlsreferenz (Cheatsheet)

grep -i -v -n -E "muster" datei

$ `Durchsucht Text nach Mustern (-i ignoriert Case, -v invertiert, -n Zeilennummer, -E ERE)`

egrep "muster1|muster2" datei

$ `Erweiterte reguläre Ausdrücke (ERE), voll kompatibles Äquivalent zu grep -E`

fgrep "fester.string" datei

$ `Schnelle Suche nach festen Zeichenketten ohne Regex-Overhead, Äquivalent zu grep -F`

sed -i 's/alt/neu/g' datei

$ `Nicht-interaktiver Stream-Editor (s Ersetzen, d Löschen, p Drucken, -i In-Place)`

awk -F: '$3 >= 1000 {print $1}' datei

$ `Mustergesteuerte Programmiersprache zur spaltenbasierten Textverarbeitung (-F Trennzeichen)`

cut -d: -f1,7 /etc/passwd

$ `Schneidet Spalten (-f) mit Trennzeichen (-d) oder Zeichenbereiche (-c) aus Zeilen`

paste -d, datei1 datei2Führt Zeilen korrespondierender Dateien horizontal mit Trennzeichen zusammen

sort -n -r -k2 -t: datei

$ `Sortiert Zeilen (-n numerisch, -r umgekehrt, -k Feldnummer, -t Trennzeichen)`

uniq -c -d -u datei

$ `Filtert aufeinanderfolgende Duplikate (-c zählt Häufigkeit, -d nur Duplikate, -u nur Unikate)`

wc -l -w -c dateiZählt Zeilen (-l), Wörter (-w) und Bytes/Zeichen (-c / -m)

tr 'a-z' 'A-Z' < datei

$ `Transformiert oder löscht Zeichen (-d Löschen, -s aufeinanderfolgende komprimieren)`

fmt -w 75 datei

$ `Formatiert und bricht Absätze auf eine maximale Zeilenbreite um`

nl -ba datei

$ `Nummeriert Zeilen einer Textdatei (-ba nummeriert alle Zeilen inklusive Leerzeilen)`

pr -3 -h "Titel" datei

$ `Bereitet Textdateien für mehrspaltige Druckausgabe auf`

expand -t 4 datei

$ `Wandelt Tabulatoren in Leerzeichen um (unexpand wandelt Leerzeichen zurück in Tabs)`

Weiterführende Ressourcen

Fazit

Die Beherrschung der UNIX-Textwerkzeuge und regulärer Ausdrücke bildet das Fundament für effiziente Systemanalyse, Logfile-Auswertung und Automatisierung in Linux. Durch das nahtlose Zusammenspiel von `grep`, `sed`, `awk`, `sort`, `uniq` und `cut` über Pipes kannst du selbst komplexe Datenströme und Konfigurationen mit wenigen, präzisen Befehlszeilen ohne externe Skriptsprachen transformieren.

> **Praxis-Tipp:** Denke beim Aufbau komplexer Pipelines immer modular: Teste jeden Filterschritt einzeln an einer kleinen Stichprobe (z. B. `head -n 20`), bevor du das Ergebnis mit `sed -i` schreibst oder in Produktivdateien umleitest.

Im nächsten Teil unserer LPIC-1-Serie widmen wir uns einem der mächtigsten Konzepte von Unix-Systemen: LPIC-1: Streams, Pipes und Umleitungen – dort lernst du `stdin`, `stdout`, `stderr`, Redirections sowie Werkzeuge wie `tee` und `xargs` kennen.

Zur Kursübersicht: Alle LPIC-1-Artikel & Module

Quelle: https://admindocs.de/de/lpic-1-serie/lpic-1-textverarbeitung-mit-shell-kommandos