grep: wyszukiwanie w treści plików

Poziom: podstawy **Dotyczy: wszystkie dystrybucje** **Czas: ~14 min**

`grep` *przeszukuje treść plików linia po linii i wypisuje te, które pasują do wzorca. To najczęściej wpisywana komenda w karierze administratora — i jedna z najczęściej używanych źle.*

Skróty użyte na tej stronie (5)

Skrót	Co znaczy
BRE	Basic Regular Expression: podstawowa, starsza odmiana wyrażeń regularnych, wymaga ukośnika wstecznego przed nawiasami
BSD	Berkeley Software Distribution: rodzina Uniksów z uczelni Berkeley, inna gałąź niż Linux, stąd czasem inna składnia tych samych poleceń
ERE	Extended Regular Expression: rozszerzona odmiana wyrażeń regularnych, bez ukośnika wstecznego przed nawiasami i plusem
PCRE	Perl Compatible Regular Expressions: odmiana wyrażeń regularnych z językiem Perl, najbogatsza w możliwości spośród trzech
POSIX	Portable Operating System Interface: wspólny zestaw zasad, dzięki któremu polecenie działa tak samo na różnych Uniksach

Fiszka

TYLKO ODCZYT

Wyszukanie fragmentu tekstu

```
grep "error" /var/log/messages
```

Wypisujemy każdą linię pliku, która zawiera podany fragment tekstu. Bez żadnych flag to najprostszy i najczęstszy przypadek użycia.

grep = global regular expression print: szukaj wzorca w tekście

TYLKO ODCZYT

Pominięcie wielkości liter: -i

```
grep -i "error" /var/log/messages
```

Dopasowujemy `error`, `Error`, `ERROR` i każdą inną kombinację wielkości liter. Sięgamy po to, gdy nie mamy pewności, jak dokładnie aplikacja pisze komunikat.

grep = global regular expression print: szukaj wzorca w tekście

TYLKO ODCZYT

Negacja dopasowania: -v

```
grep -v "debug" /var/log/messages
```

Wypisujemy linie, które **nie** pasują do wzorca — odwrotność zwykłego działania `grep`. Przydatne do odfiltrowania szumu (np. wpisów `debug`) i skupienia się na reszcie.

grep = global regular expression print: szukaj wzorca w tekście

TYLKO ODCZYT

Numerowanie dopasowanych linii: -n

```
grep -n "error" /var/log/app.log
```

Doklejamy do każdego wyniku numer linii w pliku. Niezastąpione, gdy trzeba komuś wskazać dokładne miejsce albo otworzyć plik w edytorze pod konkretnym numerem.

Przykład:

```
142:2026-09-17 08:12:03 error: connection refused
987:2026-09-17 09:03:41 error: timeout waiting for lock
```

grep = global regular expression print: szukaj wzorca w tekście

TYLKO ODCZYT

Zliczenie dopasowań: -c

```
grep -c "error" /var/log/app.log
```

Zamiast wypisywać pasujące linie, podajemy samą ich liczbę. Szybsze niż `grep ... | wc -l`, bo nie trzeba przepychać tekstu przez dodatkowy proces.

Przykład:

```
2
```

grep = global regular expression print: szukaj wzorca w tekście

TYLKO ODCZYT

Wypisanie nazw plików z dopasowaniem: -l

```
grep -l "error" /var/log/*.log
```

Zamiast treści wypisujemy same nazwy plików, w których wzorzec choć raz wystąpił. Wygodne przy przeszukiwaniu wielu plików naraz, gdy interesuje nas tylko „gdzie", nie „co dokładnie".

grep = global regular expression print: szukaj wzorca w tekście

TYLKO ODCZYT

Wypisanie nazw plików bez dopasowania: -L

```
grep -L "error" /var/log/*.log
```

Dokładna odwrotność `-l`: wypisujemy pliki, w których wzorzec **nie** wystąpił ani razu.

grep = global regular expression print: szukaj wzorca w tekście

TYLKO ODCZYT

Dopasowanie całego słowa: -w

```
grep -w "cron" /etc/services
```

Dopasowujemy `cron` tylko jako osobne słowo — nie złapiemy go w środku `crontab` czy `microcron`. Bez tej flagi `grep` dopasowuje dowolny podciąg, niezależnie od tego, co go

otacza.

grep = global regular expression print: szukaj wzorca w tekście

TYLKO ODCZYT

Dopasowanie całej linii: -x

```
grep -x "root" /etc/passwd_grupa
```

Wymagamy, żeby cała linia była dokładnie równa wzorcowi, nie tylko go zawierała. Przydaje się przy walidacji formatu pliku wiersz po wierszu.

grep = global regular expression print: szukaj wzorca w tekście

TYLKO ODCZYT

Wypisanie tylko dopasowanego fragmentu: -o

```
grep -o -E '[0-9]{1,3}(\.[0-9]{1,3}){3}' /var/log/nginx/access.log | sort -u
```

Zmieniamy zachowanie z „wypisz całą pasującą linię" na „wypisz tylko dopasowany fragment". Bez tej flagi dostalibyśmy całe linie logu, z `-o` — same adresy, po jednym na linię, gotowe do dalszego przetwarzania.

grep = global regular expression print: szukaj wzorca w tekście

TYLKO ODCZYT

Sprawdzenie dopasowania po kodzie wyjścia: -q

```
grep -q "NazwaSerwera" /etc/hosts && echo "wpis istnieje"
```

Nic nie wypisujemy na ekran — liczy się wyłącznie to, czy `grep` zakończy się kodem 0 (dopasowanie było) czy 1 (nie było). Standard w instrukcjach `if` wewnątrz skryptów.

grep = global regular expression print: szukaj wzorca w tekście

TYLKO ODCZYT

Dołączenie kontekstu wokół dopasowania: -A / -B / -C

```
grep -A 2 "Traceback" app.log
```

`-A N` doklejamy N linii **po** dopasowaniu, `-B N` — N linii **przed**, `-C N` — po tyle samo z obu stron. Niezbędne przy analizie stack trace'ów, gdzie sama linia z komunikatem błędu nic nie mówi bez otoczenia.

Przykład:

```
2026-09-17 09:03:41 error: Traceback (most recent call last):
  File "app.py", line 88, in connect
  File "app.py", line 41, in <module>
```

grep = global regular expression print: szukaj wzorca w tekście

TYLKO ODCZYT

Przeszukanie katalogów rekurencyjnie: -r

```
grep -r "TODO" ./src
```

Przeszukujemy cały wskazany katalog i jego podkatalogi zamiast pojedynczego pliku. Wariant `-R` robi to samo, ale dodatkowo podąża za dowiązaniami symbolicznymi napotkanymi po drodze.

grep = global regular expression print: szukaj wzorca w tekście

TYLKO ODCZYT

Filtrowanie po wzorcu nazwy pliku: --include / --exclude

```
grep -r --include='*.conf' "server" /etc
grep -r --exclude='*.log' "haslo" /opt/app
```

Ograniczamy rekurencyjne przeszukiwanie do plików pasujących do wzorca (`--include`) albo pomijamy te pasujące (`--exclude`) — bez tego `grep -r` przetworzy każdy plik w drzewie, nawet binarne czy nieistotne.

Przykład:

```
/etc/nginx/nginx.conf:12:    listen 80;
/etc/nginx/conf.d/ssl.conf:3:    listen 443 ssl;
```

□□ **grep** = global regular expression print: szukaj wzorca w tekście

□□ TYLKO ODCZYT

Pominięcie całego katalogu: --exclude-dir

```
grep -r --exclude-dir='.git' --exclude-dir='node_modules' "TODO" .
```

Wykluczamy z rekurencji całe katalogi po nazwie, nie tylko pojedyncze pliki. Bez tego `grep -r` w repozytorium Gita przeczyta też historię commitów zapisaną binarnie w `.git/objects` — wolno i bezcelowo.

□□ **grep** = global regular expression print: szukaj wzorca w tekście

□□ TYLKO ODCZYT

Przełączenie na wyrażenia rozszerzone: -E (ERE)

```
grep -E "error|warning|fail" app.log
```

Przełączamy silnik dopasowania z domyślnego BRE na ERE, gdzie `+ ? | ( ) { }` są od razu metaznakami, bez potrzeby ich escapowania. Pełny opis różnic dialektów wyrażeń regularnych jest na osobnej stronie.

□□ **ERE** = Extended Regular Expression: rozszerzona odmiana wyrażeń regularnych, bez ukośnika wstecznego przed nawiasami i plusem · **BRE** = Basic Regular Expression: podstawowa, starsza odmiana wyrażeń regularnych, wymaga ukośnika wstecznego przed nawiasami ·**grep** = global regular expression print: szukaj wzorca w tekście

□□ TYLKO ODCZYT

Dopasowanie dosłownego tekstu bez regexu: -F

```
grep -F "3.14+2.71" wzory.txt
```

Traktujemy wzorzec jako zwykły ciąg znaków, nie wyrażenie regularne — `.` i `+` znaczą dokładnie to, co widać, a nie „dowolny znak" i „jeden lub więcej".

□□ **grep** = global regular expression print: szukaj wzorca w tekście

□□ TYLKO ODCZYT

Przełączenie na silnik PCRE: -P

```
grep -P '\d{3}-\d{2}-\d{4}' dane.txt
```

Włączamy silnik PCRE (Perl Compatible Regular Expressions) z klasami skrótowymi (`\d`, `\w`, `\s`) i asercjami wyprzedzającymi. Nie jest gwarantowane wszędzie — część buildów `grep` na BSD w ogóle go nie ma.

PCRE = Perl Compatible Regular Expressions: odmiana wyrażeń regularnych z językiem Perl, najbogatsza w możliwości spośród trzech · **BSD** = Berkeley Software Distribution: rodzina Uniksów z uczelni Berkeley, inna gałąź niż Linux, stąd czasem inna składnia tych samych poleceń ·**grep** = global regular expression print: szukaj wzorca w tekście

TYLKO ODCZYT

Dodanie kolejnego wzorca: -e

```
grep -e "error" -e "fail" -e "critical" app.log
```

Podajemy kilka niezależnych wzorców w jednym wywołaniu — linia pasuje, jeśli spełnia którykolwiek z nich. Przydatne, gdy wzorce zawierają znaki, które utrudniałyby sklejenie ich alternatywą `|`.

grep = global regular expression print: szukaj wzorca w tekście

TYLKO ODCZYT

Wczytanie wzorców z pliku: -f

```
grep -f wzorce.txt app.log
```

Czytamy listę wzorców z pliku, jeden na linię, zamiast wypisywać je wszystkie w linii poleceń. Wygodne, gdy lista jest długa albo generowana automatycznie.

grep = global regular expression print: szukaj wzorca w tekście

TYLKO ODCZYT

Wymuszenie kolorowania w potoku: --color=always

```
grep --color=always "error" app.log | less -R
```

`grep` domyślnie koloruje dopasowania tylko, gdy pisze na terminal — w potoku (np. do `less`) kolor by zniknął. `--color=always` wymusza go niezależnie od odbiorcy, a `less -R` przepuszcza kody koloru zamiast je pokazywać jako śmieci.

grep = global regular expression print: szukaj wzorca w tekście

TYLKO ODCZYT

Wymuszenie buforowania liniowego: --line-buffered

```
tail -F /var/log/messages | grep --line-buffered -i "error"
```

Bez tej flagi `grep`, którego wyjście trafia do potoku, buforuje dane blokowo i może milczeć długo, mimo że dopasowania faktycznie się pojawiają. `--line-buffered` każe wypisywać wynik natychmiast po każdej dopasowanej linii.

tail = pokaż koniec pliku, z flagą -f śledzi plik na żywo

TYLKO ODCZYT

Wszystkie flagi

Flaga	Co robi	Kiedy używasz
`-i`	ignoruje wielkość liter	szukasz bez pewności co do capsa
`-v`	neguje dopasowanie	chcesz zobaczyć, co NIE pasuje
`-n`	dodaje numer linii	trzeba wskazać komuś dokładne miejsce
`-c`	liczy pasujące linie, nie wypisuje ich	interesuje Cię tylko liczba
`-l`	wypisuje nazwy plików z dopasowaniem	szukasz w wielu plikach naraz
`-L`	wypisuje nazwy plików BEZ dopasowania	odwrotność `-l`
`-w`	dopasowanie całego słowa	unikasz trafień w środku innego słowa
`-x`	dopasowanie całej linii	walidujesz format wiersza, nie fragment
`-o`	wypisuje tylko dopasowany fragment	wyciągasz konkretne dane z linii
`-q`	brak wyjścia, liczy się kod wyjścia	używasz w `if` w skrypcie

Flaga	Co robi	Kiedy używasz
`-A N` / `-B N` / `-C N`	N linii po / przed / wokół	analiza stack trace, logów wokół zdarzenia
`-r` / `-R`	rekurencja (bez / z podążaniem za symlinkami)	przeszukanie drzewa katalogów
`--include` / `--exclude`	filtr po wzorcu nazwy pliku	ograniczasz rekurencję do `*.conf` itp.
`--exclude-dir`	pomija cały katalog	omijasz `.git`, `node_modules`
`-E`	ERE — rozszerzone wyrażenia regularne	wzorzec z `+ ? \ ( ) ` bez masy backslashy
`-F`	traktuj wzorzec jako zwykły tekst	wzorzec ma znaki specjalne, których nie chcesz interpretować
`-P`	PCRE (nie wszędzie dostępne)	potrzebujesz `\d`, `\w`, lookahead
`-e`	kolejny wzorzec (można wielokrotnie)	szukasz kilku alternatyw naraz
`-f plik`	wzorce z pliku	lista wzorców jest długa albo generowana
`--color=always`	wymusza kolor mimo potoku	przekazujesz wynik dalej, np. do `less -R`
`--line-buffered`	wyłącza buforowanie blokowe	`grep` w potoku obserwującym plik na żywo
`-a`	traktuj plik binarny jak tekst	świadomie grepujesz plik binarny

Uważaj

- `ps aux | grep nginx` sam pojawia się w wyniku, bo `grep` widzi swój własny wiersz procesu z argumentem `nginx` w linii poleceń. Patrz sekcja *Częste błędy*.
- `grep -r` bez `--exclude-dir='.git'` w repozytorium Gita przeczyta też historię commitów zapisaną binarnie w `.git/objects` — wolno i bezcelowo.
- Potok `tail -f plik | grep wzorzec` bez `--line-buffered` potrafi milczeć godzinami, bo `grep` buforuje wyjście blokowo, gdy nie pisze na terminal.

Z życia wzięte

Z ŻYCIA WZIĘTE

Wtorek, 9:40. Tomek na `app01` sprawdza, czy aplikacja Javy Oli w ogóle działa, przez `ps aux | grep java`, widzi jeden wiersz i zgłasza, że wszystko gra. Proces padł kwadrans wcześniej: to, co złapał `grep`, to jego własny wpis z argumentem `java` w linii poleceń. Ola uczy go `pgrep -f java` zamiast tej kombinacji, żeby taka pomyłka się nie powtórzyła.

Po co to istnieje

Masz plik logu na 400 000 linii i szukasz jednego zdarzenia — na przykład wszystkich razy, kiedy usługa zgłosiła `connection refused`. Przewijanie ręczne w edytorze to praca na pół godziny. `grep "connection refused" /var/log/app.log` odpowie w ułamku sekundy.

To jest właściwy zakres `grep`: masz dużo tekstu i chcesz zobaczyć tylko te linie, które spełniają jakiś warunek — prosty (dosłowny fragment) albo złożony (wyrażenie regularne, osobna strona o BRE/ERE/PCRE). `grep` nie edytuje, nie sortuje, nie liczy statystyk — filtruje linie. Cała jego siła bierze się z tego, że robi jedną rzecz szybko, a potem daje się wpiąć w dowolny potok.

Analogia

Wyobraź sobie, że czytasz **gazetę z zakreślaczem w ręku**, szukając wzmianek o konkretnej firmie.

Nie czytasz artykułów ze zrozumieniem — przebiegasz wzrokiem każdą linijkę tekstu i sprawdzasz, czy pada w niej szukane słowo. Jeśli tak, zakreślasz całą linijkę, nie tylko samo słowo. Jeśli nie — lecisz dalej. Nie wracasz do poprzednich stron, nie łączysz kontekstu między artykułami, chyba że ktoś Cię o to poprosi — wtedy zakreślasz też linijkę przed i po trafieniu, żeby zobaczyć, o co

chodziło (to jest dokładnie `-A`/`-B`/`-C`).

Ta analogia niesie ograniczenie, które tłumaczy sporo zachowań `grep`. Zakreślacz reaguje na to, co widzi dosłownie — jeśli szukasz „kot", zaznaczy też „kotwica" i „kotlet", bo zawierają ten sam ciąg znaków. Żeby zakreślić tylko całe słowo „kot", trzeba dodać instrukcję „tylko gdy to osobne słowo" — to jest `-w`. Jeśli cała linijka ma brzmieć dokładnie „kot" i nic więcej — to `-x`.

I jeszcze jedno: czytelnik z zakreślaczem nie pamięta, co zaznaczył na poprzedniej stronie. Każda linia jest oceniana osobno, niezależnie od reszty pliku — `grep` nie potrafi powiedzieć „ta linia jest podobna do tamtej sprzed 200 wierszy", umie tylko sprawdzić bieżącą linię względem wzorca.

Jak to działa

Rozwiń: jak to działa pod spodem

Nazwa `grep` pochodzi z edytora tekstu `ed`. W `ed` istniała komenda `g/re/p` — **g**lobal / **re**gular expression / **p**rint: „dla każdej linii (global), jeśli pasuje do wyrażenia regularnego (re), wypisz ją (print)". Ken Thompson w 1973 roku wydzielił tę funkcję z `ed` jako osobny program, bo okazała się użyteczna na tyle, że nie chciano za każdym razem otwierać edytora. Nazwa skróciła się do `grep`.

Mechanicznie `grep` robi pętlę: czyta plik (albo standardowe wejście) linia po linii, sprawdza dopasowanie do wzorca silnikiem wyrażeń regularnych, i jeśli dopasowanie wystąpiło (albo nie wystąpiło, przy `-v`), wypisuje linię. Domyślny silnik to BRE — dokładny opis różnic między BRE, ERE i PCRE jest na osobnej stronie.

> **Pojęcie: metaznak.** Znak w wyrażeniu regularnym, który nie oznacza samego siebie, tylko instrukcję dla silnika dopasowania — np. `.` znaczy „dowolny znak", a `*` znaczy „zero lub więcej poprzedniego znaku", nie dosłowną kropkę czy gwiazdkę. Żeby dopasować taki znak dosłownie, trzeba go zescapować backslashem (`\.`) albo użyć `-F`, które wyłącza interpretację całego wzorca jako wyrażenia regularnego. To, które znaki są metaznakami, zależy od dialektu wyrażeń regularnych — temat rozwinięty na osobnej stronie.

`grep` kończy działanie z jednym z trzech kodów wyjścia, i to jest istotne w skryptach:

- **0** — znaleziono co najmniej jedno dopasowanie,
- **1** — nie znaleziono żadnego dopasowania (to nie jest błąd, to informacja),
- **2** — wystąpił błąd, np. plik nie istnieje albo wzorzec jest niepoprawny składniowo.

Dzięki temu `grep -q wzorzec plik` da się użyć jako warunek w `if`, bez wypisywania czegokolwiek na ekran — liczy się tylko kod wyjścia:

```
if grep -q "^root:" /etc/passwd; then
    echo "użytkownik root istnieje w pliku passwd"
fi
```

Krok po kroku

1. Znajdź linie z błędem, z numerem linii.

```
grep -n "error" /var/log/app.log
```

```
142:2026-09-17 08:12:03 error: connection refused
987:2026-09-17 09:03:41 error: timeout waiting for lock
```

2. Policz, ile razy coś wystąpiło, bez czytania linii.

```
grep -c "error" /var/log/app.log
```

```
2
```

3. Sprawdź kontekst wokół awarii.

```
grep -B 2 -A 5 "Traceback" /var/log/app.log
```

Zobaczysz dwie linie przed `Traceback` (często timestamp i wywołanie funkcji) i pięć linii po nim (najczęściej treść wyjątku).

4. Przeszukaj katalog konfiguracyjny rekurencyjnie, tylko pliki `.conf`.

```
grep -rn --include='*.conf' "listen" /etc/nginx
```

```
/etc/nginx/nginx.conf:12:    listen 80;
/etc/nginx/conf.d/ssl.conf:3:    listen 443 ssl;
```

5. Wyciągnij same adresy IP z logu, bez reszty linii.

```
grep -o -E '[0-9]{1,3}(\.[0-9]{1,3}){3}' /var/log/nginx/access.log | sort -u
```

`-o` zmienia zachowanie z „wypisz całą pasującą linię" na „wypisz tylko dopasowany fragment". Bez tej flagi dostałbyś całe linie logu, z `-o` — same adresy, po jednym na linię, gotowe do dalszego przetwarzania.

6. Znajdź proces po nazwie bez łapania samego `grep`.

```
pgrep -a nginx
```

```
1842 nginx: master process /usr/sbin/nginx
1843 nginx: worker process
```

`pgrep` jest zaprojektowane do tego zadania — nie musisz wywoływać `ps`.

Częste błędy

`ps aux | grep nazwa` łapie sam siebie.

```
ps aux | grep nginx
```

```
adm_user   3021  0.0  0.0   7488  2176 pts/0    S+   10:14   0:00 grep --color=auto nginx
```

Jeśli akurat nginx nie działa, dostaniesz jeden wiersz — sam proces `grep`, bo w liście procesów widnieje jego własna linia poleceń zawierająca słowo `nginx`. To wygląda jak działający proces, a to tylko artefakt. Trzy poprawki, w kolejności preferencji:

```
pgrep -a nginx                   # najlepsze — właściwe narzędzie do zadania
ps aux | grep [n]ginx            # klasa znakowa [n] rozbija literalny ciąg "nginx"
                                  # w linii poleceń grepa, więc grep siebie nie widzi
ps aux | grep nginx | grep -v grep # działa, ale brzydkie i zawodne przy nazwach
                                  # zawierających "grep" jako podciąg
```

Wzorzec ze znakami specjalnymi bez cudzysłowu.

```
grep error* /var/log/app.log
```

Jeśli w bieżącym katalogu są pliki pasujące do wzorca `error*` (np. `error.txt`), powłoka rozwinie go **przed** przekazaniem do `grep` — i `grep` dostanie zupełnie coś innego niż zamierzałeś: nazwę pliku jako wzorzec, a Twój plik logu jako drugi argument (czyli plik do przeszukania). Zawsze cytuj wzorzec:

```
grep "error*" /var/log/app.log
```

I pamiętaj, że wtedy `*` w regexie oznacza „zero lub więcej poprzedniego znaku", nie „dowolny ciąg" jak w globbingu powłoki — `error*` w regexie dopasuje „erro", „error", „errorr", nie „error123".

Oczekiwanie ERE tam, gdzie działa BRE.

```
grep "error|warning" app.log
```

To nie zwróci nic sensownego — w domyślnym trybie BRE znak `|` jest **literalny**, więc `grep` szuka dosłownego ciągu `error|warning`, a nie alternatywy. Trzeba albo escapować `|`, albo przełączyć się na ERE:

```
grep "error\|warning" app.log     # BRE z escapowanym |
grep -E "error|warning" app.log   # ERE, czytelniejsze
```

Pełne wytłumaczenie różnic między dialektami jest na stronie *Wyrażenia regularne: BRE, ERE, PCRE*.

Cisza w potoku na żywo.

```
tail -f /var/log/messages | grep -i error
```

Wygląda, jakby nic się nie działo, mimo że błędy faktycznie lecą do logu. Powód: `grep`, gdy jego wyjście trafia do potoku (nie na terminal), przełącza się z buforowania liniowego na blokowe — czeka, aż uzbiera pełny blok danych (zwykle 4 kB), zanim cokolwiek wypisze. Przy rzadkich wpisach możesz czekać godzinami. Rozwiązanie:

```
tail -F /var/log/messages | grep --line-buffered -i error
```

Więcej o samym `tail -f` i pułapkach obserwacji plików na żywo — na stronie *Obserwacja plików na żywo: tail -f, less +F, watch*.

Grepowanie pliku binarnego bez `-a` i zdziwienie komunikatem.

```
grep "libssl" /usr/sbin/sshd
```

```
Binary file /usr/sbin/sshd matches
```

`grep` wykrywa plik binarny (obecność bajtu `NUL` we wczesnym fragmencie) i domyślnie nie wypisuje jego zawartości na ekran, żeby nie rozsypać terminala sekwencjami sterującymi — tylko informuje, że dopasowanie wystąpiło. Jeśli naprawdę chcesz zobaczyć pasujący fragment jako tekst:

```
grep -a "libssl" /usr/sbin/sshd
```

Ćwiczenia

> Wszystko poniżej wykonujesz na własnej maszynie, w katalogu `/tmp`. Nic nie wymaga uprawnień administratora.

Ćwiczenie 1 — podstawowe filtry

Przygotuj plik logu i przećwicz podstawowe filtry.

```
cd /tmp
cat > app.log <<'EOF'
2026-09-17 08:01:03 info: start service
2026-09-17 08:12:03 error: connection refused
2026-09-17 08:12:04 info: retry attempt 1
2026-09-17 08:45:11 warning: high memory usage
2026-09-17 09:03:41 error: timeout waiting for lock
2026-09-17 09:03:42 info: lock acquired
EOF
```

Zadania:

1. Wypisz same linie z błędami, z numerem linii.
2. Policz, ile jest linii z `error` LUB `warning`, jedną komendą z `-E`.
3. Wypisz nazwy plików z bieżącego katalogu, które zawierają słowo `error` (masz tylko jeden plik — sprawdź działanie `-l` i `-L`).
4. Wyciągnij same znaczniki czasu (pierwsze 19 znaków każdej linii) przy pomocy `grep -o`.

Odpowiedź

```
grep -n "error" app.log
grep -Ec "error|warning" app.log
grep -l "error" app.log
grep -L "error" app.log
grep -o -E '^[0-9]{4}-[0-9]{2}-[0-9]{2} [0-9]{2}:[0-9]{2}:[0-9]{2}' app.log
```

Zadanie 3: `-l` wypisze `app.log` (zawiera dopasowanie), `-L` nic nie wypisze, bo nie masz drugiego pliku bez `error` do porównania — dorzuć `printf 'wszystko ok\n' > clean.log` i powtórz `grep -L "error" *.log`, żeby zobaczyć różnicę.

Pytania kontrolne

1. Dlaczego `grep "a+b" plik` i `grep -E "a+b" plik` mogą dać różne wyniki na tym samym pliku?

Odpowiedź

Bez `-E` grep działa w trybie BRE, gdzie `+` jest znakiem literalnym — wzorzec szuka dosłownego ciągu „a+b". Z `-E` (ERE) `+` jest kwantyfikatorem — wzorzec szuka jednego lub więcej `a`, po których następuje `b`, czyli dopasuje też „aab", „aaab" itd. To jeden z najczęstszych powodów „grep mi nie działa" u osób przechodzących z innych języków, gdzie `+` zawsze jest kwantyfikatorem.

2. Masz skrypt, który ma się zatrzymać z błędem, jeśli w pliku konfiguracyjnym NIE MA linii `enabled=true`. Jak to sprawdzić przy pomocy `grep` bez wypisywania czegokolwiek na ekran?

Odpowiedź

```
if ! grep -q "^enabled=true$" config.ini; then
    echo "Brak enabled=true" >&2
    exit 1
fi
```

`-q` tłumi wyjście, liczy się tylko kod wyjścia: 0, jeśli linia istnieje, 1, jeśli nie. `!` neguje warunek w `if`. Kotwice `^` i `$` (opisane na stronie o wyrażeniach regularnych) wymuszają dopasowanie całej linii, a nie fragmentu — bez nich `enabled=truefalse` też by przeszło.

3. Dlaczego `ps aux | grep nginx` czasem pokazuje jeden wynik, mimo że nginx w ogóle nie działa?

Odpowiedź

Ten jeden wynik to sam proces `grep` — w liście procesów widnieje jego linia poleceń, która zawiera słowo `nginx` jako argument wywołania. `grep` dopasowuje samego siebie. Trzeba użyć `pgrep -a nginx`, `ps aux | grep [n]ginx` albo dopisać `| grep -v grep`.

4. Masz plik z 2 GB logów w potoku `tail -F plik | grep -i error` i nic się nie wyświetla, choć wiesz, że błędy lecą co kilka minut. Co sprawdzisz najpierw?

Odpowiedź

Buforowanie wyjścia `grep` w potoku. Gdy `grep` nie pisze bezpośrednio na terminal, przełącza się na buforowanie blokowe i czeka na uzbieranie pełnego bloku danych, zanim cokolwiek wypisze. Rozwiązanie: `tail -F plik | grep --line-buffered -i error`.

Powiązane: Wyrażenia regularne: BRE, ERE, PCRE · Obserwacja plików na żywo: tail -f, less +F, watch · Potoki i filozofia Unix · Kod wyjścia, $?, &&, ||, średnik · Cudzysłowy, escapowanie i ekspansja

Źródła: `man 1 grep` · `info grep` · POSIX.1-2024, `grep` · historia `ed` i `g/re/p` — Kernighan, „Unix: A History and a Memoir"

Revision #19
Created 2026-09-17 15:06:52 UTC by Tomasz Konieczny
Updated 2026-10-03 19:07:52 UTC by Tomasz Konieczny