awk: język do danych w kolumnach

◫◫ Poziom: średni **◫◫ Dotyczy: wszystkie dystrybucje** **⏱ Czas: ~20 min**

`awk` to pełnoprawny język programowania do danych w kolumnach, nie zestaw flag. Tam gdzie `grep` filtruje, a `sed` podmienia, `awk` liczy, sumuje, grupuje i formatuje — jednym poleceniem tam, gdzie inaczej trzeba by pisać skrypt.

◫◫ Skróty użyte na tej stronie (6)

Skrót	Co znaczy
B2B	business to business: sprzedaż i systemy skierowane do innych firm, nie do klientów indywidualnych
BSD	Berkeley Software Distribution: rodzina Uniksów z uczelni Berkeley, inna gałąź niż Linux, stąd czasem inna składnia tych samych poleceń
CSV	w Windowsie: Cluster Shared Volume, dysk widoczny naraz dla wszystkich węzłów klastra; w danych: Comma-Separated Values, plik z wartościami oddzielonymi przecinkami
IP	Internet Protocol: adres komputera w sieci; w elektryce: stopień ochrony obudowy (Ingress Protection)
POSIX	Portable Operating System Interface: wspólny zestaw zasad, dzięki któremu polecenie działa tak samo na różnych Uniksach
RHEL	Red Hat Enterprise Linux: komercyjna dystrybucja Linuksa

Fiszka

> **Pojęcie: rekord i pole.** `awk` czyta wejście linia po linii — każda linia to **rekord**. Rekord jest automatycznie dzielony na **pola**: fragmenty oddzielone separatorem, ponumerowane od `$1` w górę. Cały rekord bez podziału to `$0`. Jeśli plik to lista logów, rekordem jest jedna linia zdarzenia, a polami — data, host, kod odpowiedzi, każde osobno adresowalne.

> **Pojęcie: separator pól.** Znak (albo wzorzec), według którego `awk` tnie rekord na pola. Domyślny separator to dowolny ciąg białych znaków — spacji i tabulatorów, dowolnej długości, ze zwinięciem wielokrotnych spacji do jednego podziału. Ustawia się go jawnie flagą `-F` albo zmienną `FS`, gdy dane mają inny format, np. CSV z przecinkiem.

Dane testowe używane w tej Fiszce — fragment logu dostępowego:

```
cat > dostep.log <<'EOF'
2026-09-17 10:01:03 srv01 200 512
2026-09-17 10:01:05 srv01 404 128
2026-09-17 10:01:09 storage01.ad.zielona.example 200 2048
2026-09-17 10:02:01 srv01 500 64
2026-09-17 10:02:14 storage01.ad.zielona.example 200 1024
2026-09-17 10:03:00 storage01.ad.zielona.example 404 96
EOF
```

Pola: `$1` data, `$2` godzina, `$3` host, `$4` kod odpowiedzi, `$5` rozmiar w bajtach.

TYLKO ODCZYT

Akcja bez wzorca: `{print}`

```
awk '{ print }' dostep.log
```

Bez podanego wzorca akcja wykonuje się dla KAŻDEJ linii wejścia. Samo `print` bez argumentów wypisuje `$0` — cały rekord, efekt identyczny z `cat`.

Przykład:

```
2026-09-17 10:01:03 srv01 200 512
2026-09-17 10:01:05 srv01 404 128
2026-09-17 10:01:09 storage01.ad.zielona.example 200 2048
2026-09-17 10:02:01 srv01 500 64
2026-09-17 10:02:14 storage01.ad.zielona.example 200 1024
2026-09-17 10:03:00 storage01.ad.zielona.example 404 96
```

awk = od nazwisk autorów Aho, Weinberger, Kernighan: język do danych w kolumnach

TYLKO ODCZYT

Konkretne pole: `$1` / `$NF`

```
awk '{ print $1, $NF }' dostep.log
```

`$1` to pierwsze pole. `$NF` to OSTATNIE pole — wygodne, gdy liczba kolumn się zmienia zależnie od linii, bo nie trzeba znać dokładnego numeru.

Przykład:

```
2026-09-17 512
2026-09-17 128
2026-09-17 2048
2026-09-17 64
2026-09-17 1024
2026-09-17 96
```

awk = od nazwisk autorów Aho, Weinberger, Kernighan: język do danych w kolumnach

ZMIENIA SYSTEM

Ustawienie separatora pól wejściowych: `-F`

```
cat > serwery.csv <<'EOF'
app01,10.20.10.41,web
pg01,10.20.10.33,db
bkp01,10.20.10.61,storage
EOF

awk -F',' '{ print $1, $2 }' serwery.csv
```

Flagą `-F` ustawiamy separator, którym `awk` dzieli wejście na pola — tu przecinek zamiast domyślnych białych znaków.

Przykład:

```
app01 10.20.10.41
pg01 10.20.10.33
bkp01 10.20.10.61
```

cat = concatenate: sklej i wypisz pliki · **awk** = od nazwisk autorów Aho, Weinberger, Kernighan: język do danych w kolumnach

□□ TYLKO ODCZYT

Numer rekordu: `NR`

```
awk '{ print NR, $3 }' dostep.log
```

`NR` liczy rekordy od początku przetwarzania i rośnie bez przerwy — przy jednym pliku to po prostu numer bieżącej linii.

Przykład:

```
1 srv01
2 srv01
3 storage01.ad.zielona.example
4 srv01
5 storage01.ad.zielona.example
6 storage01.ad.zielona.example
```

□□ **awk** = od nazwisk autorów Aho, Weinberger, Kernighan: język do danych w kolumnach

□□ ZMIENIA SYSTEM

Numer rekordu w bieżącym pliku: `FNR`

```
cat > dostep2.log <<'EOF'
2026-09-18 09:00:01 srv02 200 256
2026-09-18 09:00:07 srv02 404 128
EOF

awk '{ print NR, FNR, FILENAME }' dostep.log dostep2.log
```

`FNR` liczy od nowa w każdym pliku, podczas gdy `NR` liczy łącznie przez wszystkie pliki podane w linii poleceń. Różnica widać dopiero przy drugim pliku.

Przykład:

```
1 1 dostep.log
2 2 dostep.log
3 3 dostep.log
4 4 dostep.log
5 5 dostep.log
```

```
6 6 dostep.log
7 1 dostep2.log
8 2 dostep2.log
```

cat = concatenate: sklej i wypisz pliki · **awk** = od nazwisk autorów Aho, Weinberger, Kernighan: język do danych w kolumnach

TYLKO ODCZYT

Liczba pól: `NF`

```
awk '{ print NF }' dostep.log
```

Liczba pól w bieżącej linii — tu zawsze 5, bo format logu jest stały. Przy plikach o zmiennej liczbie kolumn (np. komunikat błędu innej długości) `NF` pozwala odwołać się do ostatniego pola przez `$NF` albo przedostatniego przez `$(NF-1)`, bez znajomości konkretnego numeru.

Przykład:

```
5
5
5
5
5
5
```

awk = od nazwisk autorów Aho, Weinberger, Kernighan: język do danych w kolumnach

TYLKO ODCZYT

Separator pól wyjściowych: `OFS`

```
awk 'BEGIN { OFS="," } { print $1, $2, $3 }' dostep.log
```

`OFS` to separator, którego `awk` używa, gdy sam SKŁADA wynik z osobnych pól w `print` (przecinkiem między argumentami). Domyślnie to pojedyncza spacja — tu zmieniamy ją na przecinek.

Przykład:

```
2026-09-17,10:01:03,srv01
2026-09-17,10:01:05,srv01
2026-09-17,10:01:09,storage01.ad.zielona.example
2026-09-17,10:02:01,srv01
2026-09-17,10:02:14,storage01.ad.zielona.example
2026-09-17,10:03:00,storage01.ad.zielona.example
```

awk = od nazwisk autorów Aho, Weinberger, Kernighan: język do danych w kolumnach

TYLKO ODCZYT

Kod przed przetworzeniem: `BEGIN`

```
awk 'BEGIN { print "Raport dostępu:" } { print $3 }' dostep.log
```

Blok `BEGIN` wykonuje się DOKŁADNIE RAZ, zanim `awk` przeczyta pierwszą linię — zanim `$1` w ogóle ma sens. Typowe zastosowanie: nagłówek albo inicjalizacja zmiennych.

Przykład:

```
Raport dostępu:
srv01
srv01
storage01.ad.zielona.example
srv01
storage01.ad.zielona.example
storage01.ad.zielona.example
```

awk = od nazwisk autorów Aho, Weinberger, Kernighan: język do danych w kolumnach

TYLKO ODCZYT

Kod po przetworzeniu: `END`

```
awk '{ } END { print NR, "linii" }' dostep.log
```

Blok `END` wykonuje się RAZ, po ostatniej linii — tam mają sens podsumowania, bo dopiero wtedy zmienne zebrały wartości ze wszystkich rekordów.

Przykład:

```
6 linii
```

awk = od nazwisk autorów Aho, Weinberger, Kernighan: język do danych w kolumnach

TYLKO ODCZYT

Wzorzec regexowy

```
awk '/storage01/' dostep.log
```

Samo wyrażenie regularne w ukośnikach jako wzorzec — dopasowywane do `$0`, czyli do całej linii. Bez podanej akcji domyślną akcją jest `{ print $0 }`.

Przykład:

```
2026-09-17 10:01:09 storage01.ad.zielona.example 200 2048
2026-09-17 10:02:14 storage01.ad.zielona.example 200 1024
2026-09-17 10:03:00 storage01.ad.zielona.example 404 96
```

awk = od nazwisk autorów Aho, Weinberger, Kernighan: język do danych w kolumnach

TYLKO ODCZYT

Wzorzec warunkowy

```
awk '$4 == 404' dostep.log
```

Wzorzec może być też warunkiem logicznym na konkretnym polu — tu dokładne dopasowanie liczbowe, nie dopasowanie do całej linii jak przy regexie.

Przykład:

```
2026-09-17 10:01:05 srv01 404 128
2026-09-17 10:03:00 storage01.ad.zielona.example 404 96
```

awk = od nazwisk autorów Aho, Weinberger, Kernighan: język do danych w kolumnach

TYLKO ODCZYT

Zakres

```
awk 'NR==2,NR==4' dostep.log
```

Podobnie jak w `sed`, dwa wzorce oddzielone przecinkiem tworzą zakres: włącza się na linii pasującej do pierwszego, wyłącza na linii pasującej do drugiego (włącznie z obiema).

Przykład:

```
2026-09-17 10:01:05 srv01 404 128
2026-09-17 10:01:09 storage01.ad.zielona.example 200 2048
2026-09-17 10:02:01 srv01 500 64
```

awk = od nazwisk autorów Aho, Weinberger, Kernighan: język do danych w kolumnach

TYLKO ODCZYT

Formatowanie: `printf`

```
awk '{ printf "%-25s %3d\n", $3, $4 }' dostep.log
```

`printf` daje pełną kontrolę nad formatem — `%-25s` to tekst wyrównany do lewej na szerokości 25 znaków, `%3d` to liczba wyrównana do prawej na szerokości 3. W przeciwieństwie do `print`, `printf` NIE dokłada automatycznie znaku nowej linii — trzeba go dopisać jako `\n`.

Przykład:

```
srv01                     200
srv01                     404
storage01.ad.zielona.example 200
srv01                     500
storage01.ad.zielona.example 200
storage01.ad.zielona.example 404
```

awk = od nazwisk autorów Aho, Weinberger, Kernighan: język do danych w kolumnach

TYLKO ODCZYT

Tablice asocjacyjne

```
awk '{ count[$3]++ } END { for (h in count) print h, count[h] }' dostep.log
```

Tablica `count` używa wartości pola jako KLUCZA — tu nazwy hosta. `count[$3]++` zwiększa licznik dla danego hosta; jeśli klucza jeszcze nie było, `awk` tworzy go z wartością 0 i od razu inkrementuje.

Przykład:

```
srv01 3
storage01.ad.zielona.example 3
```

awk = od nazwisk autorów Aho, Weinberger, Kernighan: język do danych w kolumnach

ZMIENIA SYSTEM

Deduplikacja bez sortowania: `!seen[$0]++`

```
cat > dostep_dubel.log <<'EOF'
2026-09-17 10:01:03 srv01 200 512
2026-09-17 10:01:03 srv01 200 512
2026-09-17 10:01:05 srv01 404 128
EOF

awk '!seen[$0]++' dostep_dubel.log
```

`seen[$0]` przed pierwszym zwiększeniem ma wartość pustą (fałsz), więc `!seen[$0]` jest prawdziwe za pierwszym razem — linia się wypisuje. Przy drugim identycznym wystąpieniu `seen[$0]` jest już niezerowe, `!` odwraca to na fałsz, i linia znika. W przeciwieństwie do `sort -u`, kolejność linii w wyniku zostaje taka, jak w oryginale.

Przykład:

```
2026-09-17 10:01:03 srv01 200 512
2026-09-17 10:01:05 srv01 404 128
```

cat = concatenate: sklej i wypisz pliki · **awk** = od nazwisk autorów Aho, Weinberger, Kernighan: język do danych w kolumnach

TYLKO ODCZYT

Długość: `length`

```
awk '{ print $3, length($3) }' dostep.log
```

`length()` bez argumentu zwraca długość `$0`; z argumentem — długość podanego pola albo napisu. Tu długość nazwy hosta.

Przykład:

```
srv01 5
srv01 5
storage01.ad.zielona.example 28
srv01 5
storage01.ad.zielona.example 28
storage01.ad.zielona.example 28
```

awk = od nazwisk autorów Aho, Weinberger, Kernighan: język do danych w kolumnach

TYLKO ODCZYT

Wycinek napisu: `substr`

```
awk '{ print substr($1, 6, 5) }' dostep.log
```

`substr(napis, start, długość)` wycina fragment zaczynając od pozycji `start` (licząc od 1). Tu z daty `2026-09-17` wycina znaki 6-10, czyli miesiąc i dzień.

Przykład:

```
09-17
09-17
09-17
09-17
09-17
09-17
```

awk = od nazwisk autorów Aho, Weinberger, Kernighan: język do danych w kolumnach

TYLKO ODCZYT

Rozbicie na tablicę: `split`

```
awk '{ n = split($1, d, "-"); print d[1], d[2], d[3] }' dostep.log
```

`split(napis, tablica, separator)` dzieli napis ręcznie i zwraca liczbę powstałych elementów. Tu rozbija datę po myślniku na rok, miesiąc, dzień.

Przykład:

```
2026 09 17
2026 09 17
2026 09 17
2026 09 17
2026 09 17
2026 09 17
```

awk = od nazwisk autorów Aho, Weinberger, Kernighan: język do danych w kolumnach

TYLKO ODCZYT

Podmiana wszędzie w linii: `gsub`

```
awk '{ gsub(/ad\.zielona\.example/, "zielona.example"); print $3 }' dostep.log
```

`gsub(wzorzec, tekst)` zamienia WSZYSTKIE dopasowania w `$0` (albo we wskazanym polu). Tu domenę `ad.zielona.example` zmieniamy na `zielona.example` wszędzie, gdzie wystąpi.

Przykład:

```
srv01
srv01
storage01.zielona.example
srv01
storage01.zielona.example
storage01.zielona.example
```

awk = od nazwisk autorów Aho, Weinberger, Kernighan: język do danych w kolumnach

TYLKO ODCZYT

Podmiana pierwszego wystąpienia: `sub`

```
echo "prod prod prod" | awk '{ sub(/prod/, "produkcja"); print }'
```

`sub()` różni się od `gsub()` dokładnie tak jak `s///` różni się od `s///g` w `sed` — zamienia TYLKO pierwsze dopasowanie w linii.

Przykład:

```
produkcja prod prod
```

□□ TYLKO ODCZYT

Pozycja dopasowania: `match`

```
awk -F',' '{ match($0, /[0-9]+\.[0-9]+\.[0-9]+\.[0-9]+/); print $1, RSTART, RLENGTH }'
serwery.csv
```

`match(napis, wzorzec)` szuka wzorca i ustawia zmienne wbudowane `RSTART` (pozycja początku dopasowania, licząc od 1) i `RLENGTH` (długość dopasowania) — albo `RSTART=0`, jeśli nic nie pasuje.

Przykład:

```
app01 7 11
pg01 6 11
bkp01 7 11
```

□□ **awk** = od nazwisk autorów Aho, Weinberger, Kernighan: język do danych w kolumnach

□□ TYLKO ODCZYT

Zamiana na wielkie litery: `toupper`

```
awk '{ print toupper($3) }' dostep.log
```

Zamieniamy cały argument na wielkie litery. Symetryczna funkcja `tolower()` robi odwrotnie.

Przykład:

```
SRV01
SRV01
STORAGE01.AD.ZIELONA.EXAMPLE
SRV01
```

```
STORAGE01.AD.ZIELONA.EXAMPLE
STORAGE01.AD.ZIELONA.EXAMPLE
```

awk = od nazwisk autorów Aho, Weinberger, Kernighan: język do danych w kolumnach

TYLKO ODCZYT

Przekazanie zmiennej z zewnątrz: `-v`

```
kod="404"
awk -v szukany="$kod" '$4 == szukany { print $3 }' dostep.log
```

Flagą `-v` ustawiamy zmienną `awk` PRZED przetworzeniem wejścia, poza treścią skryptu. Bezpieczniejsze niż wklejanie `$kod` bezpośrednio w skrypt w podwójnych cudzysłowach, bo skrypt zostaje wtedy jednym, spójnym blokiem w apostrofach, którego powłoka w ogóle nie dotyka.

Przykład:

```
srv01
storage01.ad.zielona.example
```

awk = od nazwisk autorów Aho, Weinberger, Kernighan: język do danych w kolumnach

TYLKO ODCZYT

Wszystkie zmienne i funkcje

Element	Co robi	Kiedy używasz
`$0` / `$N` / `$NF`	cały rekord / pole N / ostatnie pole	dostęp do danych w kolumnach
`NR` / `FNR`	numer rekordu globalnie / w bieżącym pliku	rozróżnienie przy wielu plikach
`NF`	liczba pól w bieżącym rekordzie	odwołanie do ostatnich pól bez znajomości numeru
`-F` / `FS`	separator pól wejściowych	CSV, `/etc/passwd`, dowolny format
`OFS`	separator pól na wyjściu	scalanie pól w `print` innym znakiem
`BEGIN` / `END`	kod przed / po przetworzeniu wszystkich rekordów	inicjalizacja, podsumowanie

Element	Co robi	Kiedy używasz
`/regex/`	wzorzec dopasowywany do `$0`	filtrowanie jak `grep`
`wyr1,wyr2`	zakres — od dopasowania do dopasowania	wycinanie bloku rekordów
`printf`	formatowanie z pełną kontrolą	tabele, wyrównane kolumny
`tablica[klucz]++`	zliczanie i grupowanie	unikaty, sumy per kategoria
`!seen[$0]++`	deduplikacja bez sortowania	usuwanie powtórzeń z zachowaniem kolejności
`length` / `substr` / `split`	długość / wycinek / rozbicie na tablicę	praca z tekstem wewnątrz pola
`gsub` / `sub`	podmiana wszędzie / tylko pierwsza	odpowiedniki `s///g` i `s///` z `sed`
`match`	pozycja i długość dopasowania	`RSTART`, `RLENGTH`
`toupper` / `tolower`	zmiana wielkości liter	normalizacja danych
`-v zmienna=wartość`	bezpieczne przekazanie wartości z powłoki	zamiast wklejania `$ZMIENNA` w skrypt

Uważaj

- `awk "{ print $1 }"` (podwójny cudzysłów) to nie to samo co `awk '{ print $1 }'` — powłoka widzi `$1` jako SWOJĄ zmienną.
- Domyślny `FS` (dowolna liczba spacji/tabulatorów) potrafi cicho źle podzielić CSV z pustymi polami — użyj jawnego `-F','`.
- `awk` to nazwa POSIX, ale za `/usr/bin/awk` kryje się różny program zależnie od dystrybucji (gawk, mawk, one-true-awk) — patrz *Częste błędy*.

CSV = w Windowsie: Cluster Shared Volume, dysk widoczny naraz dla wszystkich węzłów klastra; w danych: Comma-Separated Values, plik z wartościami oddzielonymi przecinkami · **POSIX** = Portable Operating System Interface: wspólny zestaw zasad, dzięki któremu polecenie działa tak samo na różnych Uniksach

Z życia wzięte

Z ŻYCIA WZIĘTE

W piątek o 16:10 sklep B2B na my01 zaczyna zwracać błędy 500, a Ola pyta, ile dokładnie żądań to dotyczyło w ostatniej godzinie. Zamiast przeglądać dziesięć tysięcy linii logu ręcznie, jednym poleceniem awk liczymy sumę bajtów i liczbę kodów 500 dla każdego hosta

> osobno. Po dwóch minutach mamy odpowiedź: awaria dotyczyła tylko jednego z trzech serwerów aplikacji, więc reszta ruchu może iść dalej normalnie.

Po co to istnieje

Masz plik `access.log` z milionem linii i pytanie: „które adresy IP generują najwięcej ruchu, i ile requestów ma każdy z top 10?". `grep` przefiltruje linie, ale nie policzy i nie pogrupuje. `sed` podmieni tekst, ale nie zsumuje kolumny. Potrzebujesz czegoś, co potraktuje każdą linię logu jak rekord z polami — adres, znacznik czasu, metoda, kod odpowiedzi, rozmiar — i pozwoli napisać: „dla każdej linii zwiększ licznik dla pola 1, na końcu wypisz wszystkie liczniki".

To jest dokładnie to, do czego `awk` został stworzony w 1977 roku (nazwa to inicjały autorów: Aho, Weinberger, Kernighan) — język do raportów z danych w kolumnach, bez pisania programu w C czy Pythonie tam, gdzie wystarczy kilka linijek. Różnica względem `grep` i `sed` jest fundamentalna: `awk` ma zmienne, tablice, arytmetykę, warunki i pętle. To nie komenda z opcjami, tylko mały, kompletny język programowania wyspecjalizowany w tekście podzielonym na kolumny.

Analogia

Pomyśl o **arkuszu kalkulacyjnym przetwarzanym wiersz po wierszu**.

Wklejasz dane do arkusza — każda linia pliku wejściowego staje się jednym wierszem, a spacje (albo przecinki, jeśli tak ustawisz) wyznaczają granice kolumn: A, B, C... czyli w języku `awk` `$1`, `$2`, `$3`. Różnica względem prawdziwego arkusza jest taka, że nie klikasz komórek — piszesz jedną formułę, a `awk` wykonuje ją automatycznie dla **każdego wiersza z osobna**, od góry do dołu, bez Twojego udziału przy każdym wierszu.

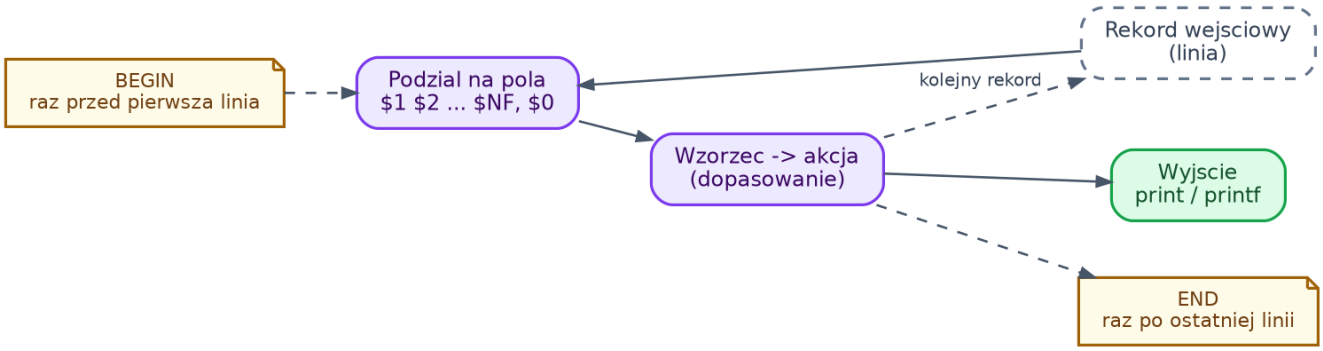
Formuła w komórce arkusza zwykle widzi tylko swój wiersz — `=B2*C2` nie wie nic o wierszu piątym. `awk` domyślnie działa tak samo: akcja `{ suma += $2 }` widzi tylko bieżącą linię. Ale tak jak w arkuszu możesz mieć osobną komórkę z sumą bieżącą, która pamięta wynik między wierszami (i aktualizuje się przy każdym nowym), tak `awk` ma zmienne i tablice, które **przeżywają** przejście do kolejnej linii — to na nich buduje się sumowanie, zliczanie i grupowanie.

Ograniczenie tej analogii pokazuje realną pułapkę: jeśli komórka ma spację w środku tekstu (np. „Jan Kowalski" w kolumnie „imię i nazwisko"), zwykłe wklejenie z pliku tekstowego porozjeżdża kolumny, bo separator trafia się też wewnątrz danych, nie tylko między nimi. Dokładnie to samo dzieje się w `awk`, kiedy separator pól jest źle dobrany do formatu danych — więcej o tym w *Częste błędy*.

Jak to działa

Rozwiń: jak to działa pod spodem

Program `awk` to seria par `wzorzec { akcja }`. Dla każdego rekordu wejścia `awk`:



Awk dzieli rekord na pola, dopasowuje kolejne wzorce z programu i wykonuje ich akcje, a BEGIN i END dopinają początek i koniec przetwarzania (kliknięcie otwiera pełny rozmiar)

1. Dzieli linię na pola według `FS` (Field Separator) i wypełnia `$1`, `$2`, ..., `$NF`, zachowując też całość jako `$0`.
2. Sprawdza po kolei każdy `wzorzec` w programie. Jeśli pasuje (albo go nie ma — wtedy pasuje zawsze), wykonuje powiązaną `{ akcję }`.
3. Jeśli przy wzorcu nie podano akcji, domyślna akcja to `{ print $0 }` — stąd `awk '/error/' plik.txt` działa jak `grep error plik.txt`.
4. Przechodzi do kolejnej linii, aż wejście się skończy.

Wzorce mogą być wyrażeniem regularnym (`/error/`, dopasowywanym do `$0`), warunkiem logicznym (`$5 > 100`, `NR == 1`), zakresem dwóch takich warunków oddzielonych przecinkiem, i specjalnymi słowami `BEGIN` (wykonywane raz, przed przeczytaniem pierwszej linii — zanim `$1` w ogóle ma sens) i `END` (raz, po ostatniej linii — tu wypisuje się podsumowania).

Pola liczone są od `$1`. `$0` to cały rekord bez podziału. `$NF` to ostatnie pole — wygodne, gdy liczba kolumn się zmienia w zależności od linii (np. w logach, gdzie komunikat błędu ma różną długość). `NF` samo w sobie to liczba pól w bieżącej linii, więc `$(NF-1)` to przedostatnie pole, a `$(NF+1)="x"` **dopisuje** nowe pole i przelicza `$0` na nowo z bieżącym `OFS`.

`FS` kontra `OFS`. `FS` to separator, którym `awk` dzieli WEJŚCIE na pola. `OFS` to separator, którego `awk` używa, gdy sam SKŁADA `$0` na nowo z powodu przypisania do jakiegoś pola albo do `NF`.

Domyślnie oba to pojedyncza spacja (a `FS` domyślnie oznacza specjalnie: „dowolny ciąg białych znaków, z pominięciem wiodących/końcowych" — nie dosłowną jedną spację). Jeśli `FS` ma więcej niż jeden znak, `awk` traktuje go jako wyrażenie regularne, np. `-F'[,;]'` dzieli po przecinku LUB średniku.

`NR` kontra `FNR`. `NR` liczy linie od początku przetwarzania i rośnie bez przerwy, nawet gdy `awk` przechodzi do drugiego pliku na liście argumentów. `FNR` liczy linie od nowa w każdym pliku. Przy jednym pliku wejściowym są identyczne — różnica ujawnia się dopiero przy `awk '...' plik1 plik2`.

Krok po kroku

1. Wybierz konkretne kolumny.

Skrypt i wynik pokazane w Fiszce, w sekcji „`$1` / `$NF` — konkretne pole". Przecinek między argumentami `print` wstawia między nimi `OFS` (domyślnie spację); bez przecinka (`print $1 $3`) pola sklejają się bez separatora.

2. Filtruj po dokładnej wartości pola.

```
awk '$3 == "srv01"' dostep.log
```

Bez podanej akcji domyślną akcją jest wypisanie całej linii — czyli efekt jak `grep`, ale z dopasowaniem do KONKRETNEGO pola, nie do całej linii. `grep srv01 dostep.log` złapałby też linię, gdzie „srv01" pojawia się gdziekolwiek indziej.

```
2026-09-17 10:01:03 srv01 200 512
2026-09-17 10:01:05 srv01 404 128
2026-09-17 10:02:01 srv01 500 64
```

3. Dopasuj pole do wyrażenia regularnego, nie do dokładnej wartości.

```
awk '$3 ~ /storage/' dostep.log       # trzecia kolumna ZAWIERA "storage"
awk '$4 !~ /^2/' dostep.log           # czwarta kolumna NIE zaczyna się od "2"
```

`~` to „pasuje do regexu", `!~` to negacja. Różnica względem `==` jest taka sama jak między `grep` a porównaniem tekstu: `==` wymaga identyczności, `~` wystarczy, że wzorzec gdzieś w polu wystąpi.

4. Porównania liczbowe i logiczne.

```
awk '$5 > 500' dostep.log
```

`awk` sam próbuje rozpoznać, czy pole jest liczbą — `$5 > 500` porówna liczbowo, jeśli `$5` wygląda jak liczba, a leksykograficznie (jak teksty), jeśli nie. To źródło subtelnych błędów przy polach, które są liczbami zapisanymi z wiodącymi zerami czy jednostką (`"100ms"`).

```
2026-09-17 10:01:09 storage01.ad.zielona.example 200 2048
2026-09-17 10:02:14 storage01.ad.zielona.example 200 1024
```

5. Sumuj kolumnę.

```
awk '{ suma += $5 } END { print suma }' dostep.log
```

Zmienna `suma` nie wymaga deklaracji ani inicjalizacji na zero — `awk` tworzy zmienne przy pierwszym użyciu z wartością pustą, która w kontekście liczbowym (`+=`) zachowuje się jak 0. Wynik wypisujesz w `END`, bo tam `suma` ma już zsumowaną wartość ze wszystkich linii.

```
3872
```

6. Policz unikatowe wartości w kolumnie.

```
awk '{ c[$4]++ } END { for (k in c) print k, c[k] }' dostep.log
```

```
200 3
404 2
500 1
```

Tablica asocjacyjna `c` używa wartości pola jako klucza (nie musi to być liczba — klucz w `awk` zawsze jest w praktyce ciągiem znaków). Pętla `for (k in c)` w `END` przechodzi po wszystkich zebranych kluczach — w kolejności, która NIE jest gwarantowana (zależy od implementacji), więc jeśli zależy Ci na sortowaniu wyniku, dołóż `sort` w potoku.

7. Grupuj sumę po kluczu.

```
awk '{ suma[$3] += $5 } END { for (h in suma) print h, suma[h] }' dostep.log
```

To samo co zliczanie, tylko zamiast `++` sumujesz wartość piątego pola pod kluczem z trzeciego — klasyczne „suma ruchu per host" w jednym przebiegu pliku.

```
srv01 704
storage01.ad.zielona.example 3168
```

8. Usuń duplikaty linii bez sortowania.

Skrypt i wynik pokazane w Fiszce, w sekcji „`!seen[$0]++` — deduplikacja bez sortowania".

9. Formatuj wynik z `printf`.

Skrypt i wynik pokazane w Fiszce, w sekcji „`printf` — formatowanie".

10. Przekaż wartość z powłoki bezpiecznie, przez `-v`.

Skrypt i wynik pokazane w Fiszce, w sekcji „`-v` — zmienna z zewnątrz". Pełne wytłumaczenie, dlaczego mieszanie cudzysłowów prowadzi do zjadania `$1` przez powłokę, jest na stronie *Cudzysłowy, escapowanie i ekspansja*.

Częste błędy

Cudzysłów zamiast apostrofu — powłoka zjada `$1`, zanim `awk` go zobaczy.

```
awk "{ print $1 }" dostep.log
```

```
2026-09-17 10:01:03 srv01 200 512
2026-09-17 10:01:05 srv01 404 128
2026-09-17 10:01:09 storage01.ad.zielona.example 200 2048
```

Wygląda, jakby zadziałało — ale to CAŁE linie, nie pierwsza kolumna. W podwójnych cudzysłowach powłoka wykonuje własną ekspansję zmiennych PRZED uruchomieniem `awk` — `$1` to dla powłoki jej własny pierwszy parametr pozycyjny, w zwykłej interaktywnej sesji pusty. Po podstawieniu pustego `$1` skrypt, który faktycznie dostaje `awk`, to `{ print }` — czyli `print` bez argumentów, co w `awk` oznacza „wypisz `$0`", całą linię. Błąd jest podstępny właśnie dlatego, że output wygląda sensownie, tylko nie robi tego, co miał robić. Gorzej, jeśli akurat wcześniej w skrypcie ustawiłeś pozycyjne `$1` powłoki na jakieś słowo (`set -- cos`) — wtedy `awk` dostanie `{ print cos }`, a `cos` jako niezainicjalizowana zmienna `awk` da puste linie. Poprawnie: apostrofy, które nic nie ekspandują, więc `$1` dociera do `awk` w całości:

```
awk '{ print $1 }' dostep.log
```

Pomylenie `NR` z `FNR` przy wielu plikach.

```
awk '{ print NR, $0 }' dostep.log dostep2.log
```

Skoro `dostep.log` ma 6 linii, pierwsza linia `dostep2.log` dostanie numer 7, nie 1 — bo `NR` liczy łącznie przez wszystkie pliki podane w linii poleceń. Jeśli chodziło Ci o numerację od nowa w każdym pliku (np. do komunikatu „błąd w linii N pliku X"), potrzebujesz `FNR`:

```
awk '{ print FNR, FILENAME, $0 }' dostep.log dostep2.log
```

Domyślny `FS` na pliku CSV z pustymi kolumnami.

```
awk '{ print $2 }' serwery.csv
```

Jeśli `serwery.csv` ma zawartość rozdzieloną przecinkami, a Ty nie ustawiłeś `-F','`, to `awk` w ogóle nie zobaczy przecinków jako separatorów — cała linia trafi do `$1` jako jeden ciąg, bo domyślny `FS` dzieli po białych znakach. Rozwiązanie oczywiste: `-F','`. Pułapka pojawia się nawet PO ustawieniu przecinka: domyślny `FS` (białe znaki) po cichu **zwija** wielokrotne separatory i ignoruje wiodące/końcowe, więc `"a  b"` (dwie spacje) da dwa pola, tak jak `"a b"`. Przecinek jako `FS` tego NIE robi — pusta kolumna między dwoma przecinkami zostaje pustym polem, co jest poprawnym zachowaniem dla CSV, ale zaskakuje kogoś przyzwyczajonego do domyślnego zachowania z białymi znakami.

```
awk -F',' '{ print "pole2=[" $2 "]" }' serwery.csv
```

```
pole2=[10.20.10.41]
pole2=[10.20.10.33]
pole2=[10.20.10.61]
```

`printf` bez `\n` — wszystko w jednej linii.

```
awk '{ printf "%s %s", $3, $4 }' dostep.log
```

```
srv01 200srv01 404storage01.ad.zielona.example 200srv01 500storage01.ad.zielona.example
200storage01.ad.zielona.example 404
```

W przeciwieństwie do `print`, `printf` nie dokłada automatycznie znaku końca linii. Wynik dla wszystkich linii wejścia skleja się w jeden ciąg. Trzeba dopisać `\n` jawnie w formacie: `printf "%s %s\n", $3, $4`.

Sklejanie zamiast dodawania przy sumowaniu.

```
awk '{ suma = suma $5 } END { print suma }' dostep.log
```

Jeden znak różnicy od poprawnej wersji (`suma += $5`), a wynik zupełnie inny: `=` z samym `$5` bez operatora arytmetycznego to **konkatenacja tekstu**, nie dodawanie. Dla kolumny `512`, `128`, `2048` ... dostaniesz `suma` równe tekstowi złożonemu ze sklejonych cyfr, nie liczbie. `awk` nie zgłosi błędu — dopiero dziwny wynik na końcu zdradza pomyłkę.

□□ Ćwiczenia

>

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

Ćwiczenie 1 — raport z logu dostępowego

```
cd /tmp
cat > log.txt <<'EOF'
2026-09-17 10:01:03 srv01 200 512
2026-09-17 10:01:05 srv01 404 128
2026-09-17 10:01:09 storage01.ad.zielona.example 200 2048
2026-09-17 10:02:01 srv01 500 64
2026-09-17 10:02:14 storage01.ad.zielona.example 200 1024
2026-09-17 10:03:00 storage01.ad.zielona.example 404 96
EOF
```

Zadania:

1. Wypisz same nazwy hostów (kolumna 3) i kody odpowiedzi (kolumna 4).
2. Policz, ile razy wystąpił każdy kod odpowiedzi (kolumna 4).
3. Zsumuj rozmiar (kolumna 5) osobno dla każdego hosta (kolumna 3).
4. Wypisz tylko linie z kodem 200 lub więcej niż 100 bajtów, sformatowane jako `host: rozmiar`.

Odpowiedź

```
# 1.
awk '{ print $3, $4 }' log.txt

# 2.
awk '{ c[$4]++ } END { for (k in c) print k, c[k] }' log.txt

# 3.
awk '{ s[$3] += $5 } END { for (h in s) print h, s[h] }' log.txt

# 4.
awk '$4 == 200 || $5 > 100 { printf "%s: %d\n", $3, $5 }' log.txt
```

W zadaniu 4 warunek jest sprawdzany dla CAŁEJ linii przed akcją — `$4 == 200` porównuje liczbowo (bo `$4` wygląda jak liczba), a `||` łączy go z drugim warunkiem. Kolejność sprawdzania

i wykonania akcji jest dokładnie taka, jak w modelu `wzorzec { akcja }` opisanym wyżej.

Pytania kontrolne

1. Dlaczego `awk "{ print $1 }" plik.txt` (z podwójnym cudzysłowem) w typowej interaktywnej sesji bash wypisze CAŁE linie zamiast pierwszej kolumny — zamiast, powiedzmy, błędu albo pustego wyjścia?

Odpowiedź

Bo cały skrypt jest w podwójnym cudzysłowie, więc powłoka wykonuje na nim własną ekspansję zmiennych, zanim `awk` w ogóle go zobaczy. `$1` to dla powłoki jej własny pierwszy parametr pozycyjny — w zwykłej interaktywnej sesji pusty. Po podstawieniu pustego `$1` `awk` dostaje dosłownie `{ print }`, czyli `print` bez argumentów — a to w `awk` oznacza „wypisz `$0`", całą linię. Dlatego wynik wygląda jak działające polecenie, tylko robi coś innego niż zamierzone. Rozwiązanie: apostrofy (`'...'`), które nie podlegają ekspansji powłoki.

2. Masz `awk '{ print NR }' a.txt b.txt` i widzisz numerację ciągnącą się od 1 do sumy linii obu plików, zamiast zaczynać od nowa w `b.txt`. To błąd w `awk`?

Odpowiedź

Nie, to zamierzone działanie `NR` — liczy linie globalnie przez wszystkie pliki podane w linii poleceń. Żeby dostać numerację od nowa w każdym pliku, trzeba użyć `FNR` zamiast `NR`.

3. Dlaczego `awk '{ print $2 }' plik.csv` na pliku z polami rozdzielonymi przecinkami może wypisać całą linię zamiast drugiej kolumny?

Odpowiedź

Bo domyślny separator pól (`FS`) to dowolny ciąg białych znaków, nie przecinek. Bez jawnego `-F','` (albo `BEGIN { FS="," }`) `awk` widzi całą linię CSV jako jedno pole, `$1`, o ile w linii nie ma spacji ani tabulatorów. `$2` wtedy po prostu nie istnieje (jest puste).

4. Czy `awk`, `gawk` i `mawk` to trzy nazwy tego samego programu?

Odpowiedź

Nie. `awk` to nazwa polecenia zdefiniowana przez POSIX, ale konkretny plik binarny pod tą nazwą różni się między dystrybucjami: na RHEL/Rocky `/usr/bin/awk` to zwykle `gawk` (GNU awk, z rozszerzeniami jak `gensub()`, `asort()`, `strftime()`), na Debianie/Ubuntu domyślnie bywa `mawk` (szybszy, ale bez większości rozszerzeń gawk), a na BSD/macOS to *one true awk* (`nawk`), też bez rozszerzeń gawk. Skrypt używający funkcji spoza standardu POSIX może więc działać na jednym systemie i wywalić się z błędem `calling undefined function` na innym.

Powiązane: sed: edytor strumieniowy · Wyrażenia regularne: BRE, ERE, PCRE · Cudzysłowy, escapowanie i ekspansja · sort i uniq: porządkowanie i zliczanie · cut, paste, join: praca na kolumnach

Źródła: `man 1 awk` · `man 1 gawk` · GNU Awk User's Guide (`gawk.gnu.org`) · POSIX.1-2024, `awk`

Revision #19
Created 2026-09-17 15:07:00 UTC by Tomasz Konieczny
Updated 2026-10-03 19:08:00 UTC by Tomasz Konieczny