Wprowadzenie do pracy w środowisku Linux

Część 3: Skrypty – Pętle, Warunki i Przetwarzanie Danych

Opracowanie dla studentów matematyki

9 grudnia 2025

Spis treści

1 Instrukcje Warunkowe oraz Pętle **3**
1.1 Instrukcje warunkowe: `if` . . . 3
1.2 Pętle: `for` i `while` . . . 4
1.2.1 Pętla `for` – iteracja po liście elementów . . . 4
1.2.2 Pętla `while` – wykonywanie do spełnienia warunku . . . 4

2 Przekazywanie argumentów do skryptu **5**

3 Przetwarzanie Tekstu i Potoki **5**

4 Tablice Asocjacyjne **6**

5 Przetwarzanie Plików: `find` i `while read` **7**
5.1 Problem z "trudnymi"nazwami plików . . . 7
5.2 Rozwiązanie: Połączenie `find` i `while read` . . . 7

6 Sumy Kontrolne **7**

7 Laboratorium: Analiza i Modyfikacja Skryptów **9**

1 Instrukcje Warunkowe oraz Pętle

Prawdziwa moc skryptów polega na ich zdolności do **powtarzania** operacji i **podejmowania decyzji**. Te dwie koncepcje – pętle i instrukcje warunkowe – zamieniają prostą listę poleceń w inteligentny program.

1.1 Instrukcje warunkowe: `if`

Instrukcja `if` pozwala skryptowi na wykonanie określonego bloku kodu tylko wtedy, gdy pewien warunek jest prawdziwy. Działa to jak rozwidlenie dróg – skrypt wybiera ścieżkę w zależności od sytuacji.

Anatomia instrukcji `if`

Strukturę można rozbudowywać o kolejne warunki (`elif`) oraz blok domyślny (`else`), który wykona się, gdy żaden z poprzednich warunków nie będzie prawdziwy.

Terminal – Bash

```
if [[ warunek_1 ]]; then
    # Ten kod wykona się, jeśli warunek_1 jest prawdziwy.
    # Bash przejdzie od razu do `fi`.
elif [[ warunek_2 ]]; then
    # Ten kod wykona się, jeśli warunek_1 był fałszywy,
    # ale warunek_2 jest prawdziwy.
else
    # Ten kod wykona się, jeśli wszystkie powyższe warunki
    # okazały się fałszywe.
fi
```

Kluczowa składnia: W Bashu warunki najczęściej umieszcza się w podwójnych nawiasach kwadratowych `[[ ... ]]`. Jest to nowoczesna i bezpieczna forma.

Operatory testów – Twój zestaw narzędzi do sprawdzania warunków

Bash oferuje bogaty zestaw operatorów do budowania warunków. Oto najważniejsze z nich:

- **Testy plików (najczęstsze w skryptach):**
  - `[[ -e $sciezka ]]` – sprawdza, czy plik lub katalog **e**gzistuje (*exists*).
  - `[[ -f $sciezka ]]` – sprawdza, czy to zwykły **f**ile (plik).
  - `[[ -d $sciezka ]]` – sprawdza, czy to **d**irectory (katalog).
- **Testy napisów (zmiennych):**
  - `[[ -z "$zmienna"]]` – prawda, jeśli zmienna jest pusta (**z**ero length).
  - `[[ -n "$zmienna"]]` – prawda, jeśli zmienna **n**ie jest pusta (*non-empty*).

  - `[[ $a == $b ]]` – prawda, jeśli napisy są identyczne.

- **Testy liczb całkowitych:**
  - `[[ $a -eq $b ]]` – równe (*equal*).
  - `[[ $a -ne $b ]]` – nie równe (*not equal*).
  - `[[ $a -gt $b ]]` – większe (*greater than*).
  - `[[ $a -lt $b ]]` – mniejsze (*less than*).
- **Operator negacji `!`:**
  - `[[ ! -d "$sciezka"]]` – prawda, jeśli `$sciezka` **nie jest** katalogiem.

1.2 Pętle: `for` i `while`

Pętle służą do wielokrotnego wykonywania tego samego bloku kodu. Bez nich musielibyśmy ręcznie powielać polecenia dla każdego przetwarzanego elementu.

1.2.1 Pętla `for` – iteracja po liście elementów

Pętla `for` jest idealna, gdy mamy z góry znaną listę elementów (np. listę plików w katalogu) i chcemy wykonać jakąś operację dla każdego z nich.

Anatomia pętli `for`

Terminal – Bash

```
for zmienna in element1 element2 element3; do
    # Ten blok kodu wykona się 3 razy.
    # W pierwszej iteracji $zmienna będzie miała wartość
    ↪  "element1",
    # w drugiej "element2", itd.
    echo "Przetwarzam: $zmienna"
done
```

Najczęstszym zastosowaniem jest iteracja po plikach: `for plik in sciezka/*`. Powłoka Bash automatycznie rozwija wzorzec `*` na listę wszystkich plików i katalogów w danej lokalizacji.

1.2.2 Pętla `while` – wykonywanie do spełnienia warunku

Pętla `while` działa inaczej: wykonuje blok kodu tak długo, jak długo jej warunek jest prawdziwy. Jest idealna w sytuacjach, gdy nie wiemy, ile będzie iteracji, np. podczas wczytywania danych z pliku linia po linii.

Anatomia pętli `while`

Terminal – Bash

```
# Przykład: odliczanie od 5 do 1
licznik=5
while [[ $licznik -gt 0 ]]; do
    echo "Odliczanie: $licznik"
    # Ważne: musimy samodzielnie modyfikować warunek,
    # w przeciwnym razie pętla będzie nieskończona!
    licznik=$((licznik - 1))
done
echo "Start!"
```

Pętla `while` sprawdza warunek na początku każdej iteracji. Jeśli warunek od razu jest fałszywy, pętla nie wykona się ani razu.

2 Przekazywanie argumentów do skryptu

Skrypty mogą przyjmować argumenty z wiersza poleceń. Wewnątrz skryptu mamy dostęp do tych argumentów za pomocą specjalnych zmiennych:

- `$0` – nazwa samego skryptu.
- `$1`, `$2`, ... – pierwszy, drugi, itd. argument.
- `$#` – całkowita liczba przekazanych argumentów.

3 Przetwarzanie Tekstu i Potoki

Jedną z najpotężniejszych cech powłoki jest możliwość łączenia prostych narzędzi w złożone sekwencje za pomocą **potoków (pipes)**, reprezentowanych przez znak `|`.

Potok (pipe)

Potok przekierowuje standardowe wyjście jednego polecenia na standardowe wejście następnego. Działa to jak linia montażowa: każde narzędzie wykonuje swoją pracę, a wynik przekazuje dalej.

W skryptach laboratoryjnych używamy potoków do wyodrębnienia kategorii z pliku. Przeanalizujmy tę linię:

Terminal – Bash

```
category=$(grep -i '^CATEGORY:' "$file" | cut -d':' -f2 | tr -d
↪  '[:space:]')
```

- `grep -i '^CATEGORY:' "$file"`
  Etap 1: Filtrowanie. Polecenie `grep` odnajduje w pliku interesującą nas linię.
  - `-i`: Ignoruj wielkość liter (*ignore case*).

  - `'^CATEGORY:'`: Wzorzec. Znak `^` to "kotwica", która oznacza "początek linii". Szukamy więc linii, które **zaczynają się** od `CATEGORY:`.

- `cut -d':' -f2`
  Etap 2: Wycinanie. Wynik z `grep` (cała linia) jest przekazywany do `cut`, które wycina z niego tylko potrzebny fragment.
  - `-d':'`: Ustawia separator (delimiter) na dwukropek. Dzieli linię na pola względem `:`.
  - `-f2`: Każe wybrać drugie pole (*field*), czyli wszystko po pierwszym dwukropku.
- `tr -d '[:space:]'`
  Etap 3: Czyszczenie. Fragment tekstu z `cut` jest przekazywany do `tr`, które usuwa z niego niepotrzebne białe znaki.
  - `-d`: Usuń znaki (*delete*).
  - `'[:space:]'`: Specjalna klasa znaków oznaczająca wszystkie białe znaki (spacje, tabulatory, itp.).

4 Tablice Asocjacyjne

Gdy potrzebujemy przechowywać zbiór powiązanych danych (np. mapowanie sum kontrolnych na ścieżki plików), idealnym rozwiązaniem są tablice asocjacyjne.

Tablica Asocjacyjna

To struktura danych przechowująca pary **klucz-wartość**, podobnie do słownika w Pythonie. Deklarujemy ją za pomocą `declare -A nazwa_mapy`.

Sprawdzanie istnienia klucza

Niezawodnym sposobem na sprawdzenie, czy klucz **istnieje** w tablicy, jest składnia:
`[[ -n "${mapa[$klucz]+x}" ]]`

Tablice asocjacyjne w praktyce: śledzenie duplikatów

Wyobraźmy sobie, jak skrypt do wyszukiwania duplikatów używa tablicy `checksum_map` do śledzenia napotkanych plików:

1. Skrypt analizuje plik `raport.txt`. Oblicza jego sumę kontrolną: `abc...`.
2. Sprawdza, czy klucz `abc...` istnieje w mapie. **Nie istnieje**.
3. Dodaje wpis do mapy: `checksum_map[abc...] = "/home/user/raport.txt"`.
4. Skrypt analizuje plik `dane.csv`. Oblicza jego sumę: `def...`.
5. Sprawdza, czy klucz `def...` istnieje w mapie. **Nie istnieje**.
6. Dodaje wpis: `checksum_map[def...] = "/home/user/dane.csv"`.

7. Skrypt analizuje plik `raport_kopia.txt`, który jest identyczny jak pierwszy.
8. Oblicza jego sumę kontrolną: `abc...`.
9. Sprawdza, czy klucz `abc...` istnieje w mapie. **Tak, istnieje!**
10. Skrypt wie, że plik `raport_kopia.txt` jest duplikatem, ponieważ jego "odcisk palca"jest już w kartotece. Raportuje znalezisko.

5 Przetwarzanie Plików: `find` i `while read`

5.1 Problem z "trudnymi"nazwami plików

Pętla `for plik in *` zawodzi, gdy nazwy plików zawierają spacje lub inne znaki specjalne.

5.2 Rozwiązanie: Połączenie `find` i `while read`

Standardowym i w 100% niezawodnym sposobem na przetwarzanie listy plików jest połączenie polecenia `find` z pętlą `while read`.

Wzorzec niezawodnego przetwarzania plików

Terminal – Bash

```
while IFS= read -r -d '' nazwa_pliku; do
    # Tutaj bezpiecznie przetwarzamy plik, którego ścieżka jest w
    ↪  "$nazwa_pliku"
    echo "Przetwarzam: $nazwa_pliku"
done < <(find /sciezka/startowa -type f -print0)
```

- `find ... -print0`: Generuje strumień ścieżek oddzielonych znakiem **NULL**.
- `< <(...)`: **Podstawienie procesu** przekierowuje ten strumień na wejście pętli `while`.
- `IFS= read -r -d ''`: Każe poleceniu `read` czytać dane aż do napotkania znaku NULL (`-d ''`), bez interpretowania znaków specjalnych (`-r`) i bez dzielenia według spacji (`IFS=`).

6 Sumy Kontrolne

W skryptach do wyszukiwania duplikatów kluczową rolę odgrywa mechanizm sum kontrolnych. Pozwala on w sposób jednoznaczny i efektywny sprawdzić, czy dwa pliki są identyczne pod względem zawartości, ignorując ich nazwy czy daty modyfikacji.

Suma Kontrolna (Checksum)

Suma kontrolna to **cyfrowy odcisk palca** pliku. Jest to unikalny, krótki ciąg znaków o stałej długości, wygenerowany na podstawie całej zawartości pliku za pomocą algorytmu kryptograficznego.

Właściwości sumy kontrolnej są analogiczne do ludzkiego odcisku palca:

- **Unikalność:** Dwa różne pliki prawie na pewno będą miały zupełnie inne sumy kontrolne.
- **Determinizm:** Ten sam plik zawsze wygeneruje dokładnie tę samą sumę kontrolną.
- **Efekt lawinowy:** Nawet najmniejsza zmiana w pliku (np. zmiana jednej litery) powoduje, że nowa suma kontrolna jest **całkowicie inna**.
- **Jednokierunkowość:** Na podstawie samej sumy kontrolnej nie da się odtworzyć oryginalnego pliku.

`sha256sum` – Cyfrowy Skaner Odcisków Palców

`sha256sum` to standardowe polecenie w Linuksie, które implementuje algorytm **SHA-256** (Secure Hash Algorithm, 256-bit). Jest to "urządzenie", które pobiera cyfrowy odcisk palca. Wynikiem jego działania jest 256-bitowy hash, reprezentowany jako 64 znaki szesnastkowe.

Efekt lawinowy w praktyce

Zobacz, jak dramatycznie zmienia się suma kontrolna po dodaniu jednego znaku:

Terminal – Bash

```
# Używamy `echo -n`, aby uniknąć dodania znaku nowej linii na
 ↪  końcu
$ echo -n "Witaj swiecie" | sha256sum
2db65224495123998781987 5e3f3ac7c9615a1334237c1a8397a216439e6a037
 ↪  -

$ echo -n "Witaj swiecie." | sha256sum
266228333f28d81084b423f7e5015e58983e20042472d423982e5d52e5b74108
 ↪  -
```

7 Laboratorium: Analiza i Modyfikacja Skryptów

Celem laboratorium jest zrozumienie działania, a następnie samodzielne rozbudowanie trzech skryptów automatyzujących pracę z plikami.

Skrypt 1: Automatyczne Sortowanie Plików

Przeznaczenie: Skrypt automatycznie klasyfikuje pliki z katalogu `inbox` do podkatalogów w folderze `classified` na podstawie zawartości każdego pliku.

Terminal – Bash

```
#!/usr/bin/env bash
INCOMING="inbox"; OUTDIR="classified"
mkdir -p "$OUTDIR"
for file in "$INCOMING"/*; do
    [ -e "$file" ] || continue
    category=$(grep -i '^CATEGORY:' "$file" | cut -d':' -f2 | tr
    ↪  -d '[:space:]')
    if [[ -z "$category" ]]; then
        category="unknown"
    fi
    mkdir -p "$OUTDIR/$category"
    cp "$file" "$OUTDIR/$category"/
    echo "Copied $(basename "$file") -> $OUTDIR/$category/"
done
echo "Classification complete."
```

Kluczowe koncepcje: Pętla `for`, potoki (‘|‘), polecenia ‘grep‘, ‘cut‘, ‘tr‘, instrukcja warunkowa `if`.

Skrypt 2: Wyszukiwanie Duplikatów (Wersja Podstawowa)

Przeznaczenie: Skrypt znajduje duplikaty plików w jednym, wskazanym katalogu. Porównuje pliki na podstawie ich zawartości za pomocą sum kontrolnych `sha256sum`.

Terminal – Bash

```
#!/usr/bin/env bash
if [[ $# -ne 1 ]]; then echo "Usage: $0 <directory>"; exit 1; fi
DIR="$1"
if [[ ! -d "$DIR" ]]; then echo "Error: '$DIR' is not a
↪ directory."; exit 1; fi
declare -A checksum_map
for file in "$DIR"/*; do
    [[ -f "$file" ]] || continue
    checksum=$(sha256sum "$file" | cut -d ' ' -f1)
    if [[ -n "${checksum_map[$checksum]+x}" ]]; then
        echo "Duplicate found:"
        echo "  Original: ${checksum_map[$checksum]}"; echo "
        ↪ Duplicate: $file"
    else
        checksum_map[$checksum]="$file"
    fi
done
```

Kluczowe koncepcje: Argumenty skryptu, walidacja danych, tablice asocjacyjne, sumy kontrolne.

Skrypt 3: Wyszukiwanie Duplikatów (Wersja Zaawansowana)

Przeznaczenie: Ulepszona wersja poprzedniego skryptu. Jest rekurencyjna i bezpieczna dla nietypowych nazw plików.

Terminal – Bash

```
#!/usr/bin/env bash
if [[ $# -ne 1 || ! -d "$1" ]]; then echo "Usage: $0
↪ <directory>"; exit 1; fi
DIR="$1"
declare -A checksum_map
while IFS= read -r -d '' file; do
    checksum=$(sha256sum "$file" | cut -d ' ' -f1)
    if [[ -n "${checksum_map[$checksum]+x}" ]]; then
        echo "Duplicate found:"
        echo "  Original: ${checksum_map[$checksum]}"; echo "
        ↪ Duplicate: $file"
    else
        checksum_map[$checksum]="$file"
    fi
done < <(find "$DIR" -type f -print0)
```

Kluczowe koncepcje: Niezawodne przetwarzanie plików za pomocą `find -print0` i pętli `while read`.

Zadania do wykonania

1. **Modyfikacja Skryptu 1 (Sortowanie):**

- Zmień skrypt tak, aby **przenosił** pliki (`mv`) zamiast je kopiować (`cp`).
- Dodaj obsługę drugiego argumentu, który będzie określał katalog docelowy (zamiast na stałe wpisanego `"classified"`). Jeśli argument nie zostanie podany, skrypt powinien użyć wartości domyślnej.

2. **Modyfikacja Skryptu 3 (Duplikaty):**

   - Dodaj do skryptu nową funkcjonalność: po znalezieniu duplikatu, skrypt powinien zapytać użytkownika, czy chce go usunąć. Użyj polecenia `read -p "Pytanie"zmienna`, aby wczytać odpowiedź. Wewnątrz pętli ‘while‘ dodaj instrukcję warunkową, która sprawdzi odpowiedź użytkownika (np. "t"lub "T").
   - (Zaawansowane) Rozbuduj skrypt tak, aby raportował nie tylko pierwszy duplikat, ale wszystkie. (Wskazówka: wartość w tablicy asocjacyjnej może być listą plików, a nie pojedynczym plikiem).