Narzędzia Wspierające Programowanie

Bash – dalsze rozszerzenie

- W tym skrypcie omówimy szereg dalszych aspektów Basha:
  - procesy
  - strumienie i przetwarzanie potokowe
  - zmienne
  - pętle i warunki
  - skrypty.

  Ciekawe kompendium znajdziemy np. [tutaj].

- Jak obsługiwać procesy.

Podejrzyjmy je w monitorze: `$ top lub htop` (wyjście: q)

i wypiszmy w terminalu: `$ ps`

(Skrót PID oznacza numer procesu).
Zobaczyliśmy jednak tylko procesy w naszej sesji.
Aby zobaczyć wszystkie (nasze), wpiszmy:

`$ ps -u {login}`

Warto też spojrzeć na ich hierarchię: `$ ps -u {login} --forest`

Widzimy, że każdy proces jest „podpięty”
pod poprzednika, który go wywołał.

Skasujmy proces, podając jego PID: `$ kill {PID}`

kasowanie na silniejszym priorytecie: `$ kill -9 {PID}`

Uwaga: skasowanie procesu-rodzica wyśle sygnał kasowania do procesów-dzieci.
Zatem, skasowanie terminala (lub sesji ssh) wyłączy wszystkie programy pod to podpięte.

Zamknięcie okna graficznego: `$ xkill` (a teraz kliknij myszką w okno)

- *Dodatek*

Wypiszmy „rodziców” procesów (PPID): `$ ps -f`

Wypiszmy nr/y procesów, gdy znamy nazwę: `$ pidof {nazwa}`

Procesy c.d.: front, tło, pauzowanie

Włączmy w sesji program: `$ sleep 200`

Tryb, w którym działa teraz proces, nazywa się foreground (front):
proces ma kontakt z terminalem, może tam kierować napisy.

Zapauzujmy teraz nasz program: `$ [Ctrl Z]`

Wypiszmy listę zadań włączonych w naszej sesji:
```
$ jobs
[1]+  Stopped                 sleep 200
```

Nasze zadanie ma na tej liście nr. 1. Wznówmy je,
ale niech przejdzie do trybu background (tło): `$ bg %1`

⇒ nie blokuje terminala, ale działa (sprawdź: `jobs`).
Wysuńmy je teraz z tła na front: `$ fg %1`

Teraz przerwijmy nasz proces (zakończmy go): `[Ctrl C]`

Włączmy teraz program z oknem graficznym: `$ gedit`

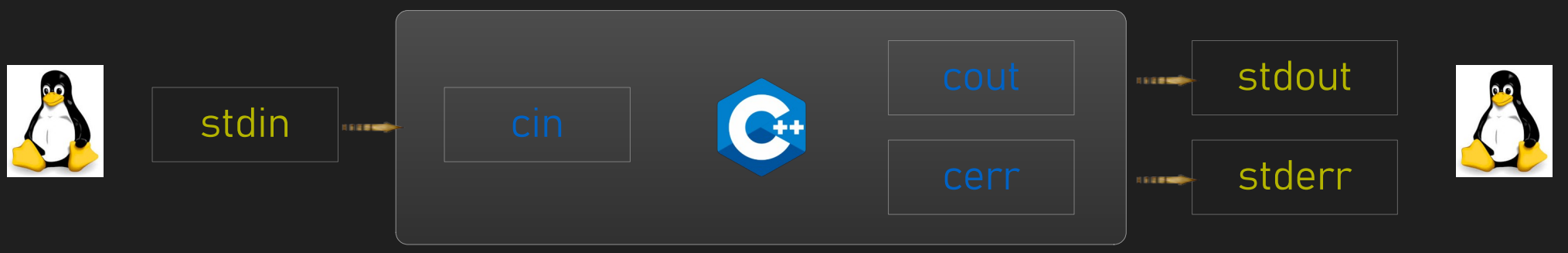
Włączenie grafiki blokuje terminal aż do jej wyłączenia
(lub zapauzowania przez [Ctrl Z]). Cofnijmy się, wyłączając gedit.

Linux ma operator & (ampersand) ,
który od razu wpuszcza program do tła. Spróbujmy: `$ gedit &`

Mamy kontakt i z terminalem, i z edytorem.

Nb.: procesy w tle są wciąż podpięte pod sesję: `$ ps -u {login} --forest`

⦿ Strumienie



Gdy program w Linuxie wypisuje tekst do terminala, to tak naprawdę przekazuje tekst do tzw. "standardowego strumienia wyjścia". W C++ polecenie `cout` przekazuje tekst do linuxowego strumienia stdout, a komenda `cerr` - do strumienia stderr (jest to strumień błędów) .
Z kolei, do programu Linux podłącza standardowy strumień wejścia stdin:
Gdy za pomocą `cin` użytkownik podaje tekst z klawiatury, tekst przechodzi przez ten strumień.

Strumienie wyjścia: `stdout` i `stderr`

Tekst w `stdout` można przekierować
z ekranu na plik, używając znaku `>`

```
$ ls > myout
```

Jeśli plik istniał wcześniej, to zostanie nadpisany.
Ale można poprzez `>>` dopisać do końca pliku:

```
$ cat "End of list" >> myout
```

Co jednak, gdy chcemy przekierować `stderr` na plik?
Okazuje się, że powyższy strumień "zwykły" ma nr 1,
a strumień błędów - nr 2. Napiszmy więc tak:

```
$ ls * i.dont.exist 1>myout 2>myerr
```

A jeśli chcemy, aby komunikaty o błędach trafiały
też do `stdout`, to piszemy tak:

```
$ ls * i.dont.exist 1>myout 2>&1
```

Można też przekierować strumień donikąd.
Służy temu miejsce docelowe `/dev/null`. Np:

```
$ ls * i.dont.exist 1>/dev/null
```

Strumień wejścia: `stdin`

Jak dotąd, użytkownik zapytany – odpowiadał z klawiatury.
Ale można pod `stdin` podpiąć plik, używając `<`.
Jako przykład, użyjmy polecenie `wc -w`,
które poda liczbę słów w badanym tekście.

```
$ wc -w < myout
```

Ale można też, używając operatora `<<`, podać tekst
ad hoc, umawiając się na hasło końca (np. EOF):

```
$ wc -w << EOF
a b c
d e f
EOF
```

→ odpowiedź: 6

Na koniec, używając `<<<`, podamy do `stdin`
jedną linię tekstu. Jeżeli w linii są jawnie spacje,
to trzeba ją otoczyć przez '' lub " ". Dla przykładu,
wczytajmy do kalkulatora `bc` działania:

```
$ bc -l <<< '2 / 3'
0.6666666666666
```

Zaczynamy dostrzegać, że Linux ma dedykowane narzędzia do konkretnych zadań.
To nie przypadek. Pierwsza zasada w "*Unix philosophy*" to: "*Make each program do one thing well.*"
Przyjrzyjmy się kilku narzędziom, które wkrótce zaprzęgniemy do tzw. "przetwarzania potokowego"

- `wc {plik}` lub `<strumień` (word count) : narzędzie do zliczania w tekście. Opcje, np:
  Opcją wskazujemy, co ma zliczać. Np.:

`-l`	podaj liczbę linii
`-w`	podaj liczbę słów
`-m`	podaj liczbę znaków

Np.:
```
$ wc -l myout
```

- `cut {opcja} {plik}` lub `<strumień` Wycina z tekstu kawałek i wypisuje go. Opcje:

`-c {zakres}`	znaki, akceptuj w zakresie { }
`-f {zakres}`	słowa (fields = pola), akceptuj w zakresie { }. Domyślny delimiter: Tab
`-f {zakres} -d {znak}`	Domyślny delimiter: {znak}
`--complement`	odwróć zasadę filtrowania

`{zakres}` : W jaki sposób możemy go zapisać? np. 2 2- -5 2-5

Np.:
```
$ cut -c 6- <<< 'I am just a nice line.'                    → just a nice line.
$ cut -c -4 <<< 'I am just a nice line.'                    → I am
$ cut -c 6-9 <<< 'I am just a nice line.'                   → just
$ cut -f 5 -d ' ' <<< 'I am just a nice line.'              → nice
$ cut -f 5 -d ' ' --complement <<< 'I am just a nice line.' → I am just a line.
```

- `tr {z} {na} {plik}` lub `<strumień` Przekształca tekst, wypisując go. Warianty:

`-d 'zbiór'`	skasuje znaki ze zbioru
`-s 'znak'`	skasuje bezpośrednie powtórzenia znaku

Np.:
```
$ tr a b <<< 'I am just a nice line.'              →  I bm just b nice line.
$ tr 'a-f' 'A-F' <<< 'I am just a nice line.'      →  I Am just A niCE linE.
$ tr '1-4' 'a-d' <<< 7654321                       →  765dcba
$ tr -d '1-4' <<< 7654321                          →  765
$ tr -s ' ' <<< '| One  Two   Tree |'              →  | One Two Tree |
```

Uwaga: jeśli do tr podamy plik, to tr nie zmieni jego wnętrza. Jedynie wypisze przeprocesowany tekst.
Aby zmienić frazę w środku pliku, propozycja: `sed -i 's/stare_slowo/nowe_slowo/g' {mojPlik}`

Plik z kolumnami danych

W naszym katalogu dostępny jest plik `data.txt` (uwaga: delimiterem kolumn jest tu tab)
Wykonajmy kilka manipulacji na tych danych:

```
$ head -2 data.txt                    wypisze tylko 2 pierwsze linie pliku
$ tail -2 data.txt                    wypisze tylko 2 ostatnie linie

$ cut -f 2 --complement data.txt      usunie kolumnę (drugą; domyślny delimiter = tab)
```

- `sort {plik}` lub `<strumien` — sortuje linie pliku. Opcje:

`-r`	w odwróconym porządku
`-k {nr kol.}`	względem danej kolumny

  Np. `$ sort -k 2 -r data.txt`

- `paste {plik1} {plik2}` — zestawia dwa pliki w sąsiednie kolumny.

  Np. rozdzielmy dane na 2 pliki: `names.txt` i `values.txt`, a następnie scalmy obok siebie.

```
$ cut -f -2 data.txt > names.txt
$ cut -f 3- data.txt < values.txt
$ paste names.txt values.txt
```

⦿ Przetwarzanie potokowe

Poznaliśmy spory zestaw komend pracujących potokowo (`stdin` → komenda → `stdout`/`stderr`). Linux ma operator `|` (`pipe`), który treść w `stdout` wstawia do `stdin` nowego polecenia. Przykłady:

```
$ ls -1 | wc -l                          Dowiemy się, ile plików/ścieżek ma katalog.

$ cat data.txt | sort -k 3 | cut -f 1    Ukaże imiona, posortowane po 3. kolumnie

$ cat data.txt | grep 2 | grep 3 | wc -l  Zliczy linie mające znak 2 oraz 3.
```

Oczywiście, na końcu wynik można zapisać do pliku, znakiem `>` .

Istnieją polecenia, które nie czytają ze `stdin` , tylko z opcji wywołania. Choćby `echo` .
Aby `stdout` z poprzednika podać do opcji wywołania następnika, korzysta się z `xargs` . Np.:

```
$ ls | xargs echo
```

Uwaga: `xargs` działa osobno na każdej linii `stdout` poprzednika. Zajdzie więc różnica:

```
$ ls -l | wc -l                          z listy plików, zostanie podana jej długość
$ ls -l | xargs wc -l                    dla każdego pliku, wc poda jego długość
```

Bywa jednak, że chcemy poprzedni `stdout` umieścić w innym miejscu następnego polecenia. Do tego służy opcja `-I {symbol}` . W ten sposób uczymy Linux, że gdy napotka taki symbol, to ma w jego miejsce wkleić treść linii w `stdout` . Np:

```
$ ls -1 a*.txt | cut -d '.' -f 1 | xargs -I {}  mv -v {}.txt {}.dat
```

Skoro mówimy o wykonywaniu operacji na kolejnych liniach wejścia, to można też wyszukiwać plików i na nich wykonywać nasze operacje:

```
$ find . -name {wzor_pliku} -exec {komenda} {}  \;
```

⬑ na każdym znalezionym pliku o danym wzorze wykona komendę, podmieniając nazwę pliku pod {}

```
$ find . -name {wzor_pliku} -execdir {komenda} {}  \;
```

⬑ na każdym znalezionym pliku o danym wzorze, przejdzie do jego ścieżki i wykona komendę, podmieniając nazwę pliku pod {}

⊕ *Dodatek*

Istnieje też polecenie `tee {plik}` , które `stdout` z poprzednika bocznikuje do pliku. Np.:

```
$ cat data.txt | sort -k 3 | tee sorted.txt | cut -f 3      pośredni wynik → sorted.txt
```

⦿ Globbing

Niedługo przejdziemy do pętli zakresowych w Bashu. Są to pętle m. in. po tablicach stringów czy tablicach nazw plików. Omówmy najpierw potrzebną tu cegiełkę.
Globbing = ekspandowanie zakresu stringów oraz nazw (istniejących) plików.

- Nawiasy klamrowe { } służą do ekspandowania stringu w tablicę, według wzorca. Np:

```
$ echo a{1..5}b                    → a1b a2b a3b a4b a5b
$ echo 1{a..e}2                    → 1a2 1b2 1c2 1d2 1e2
$ echo a{15..6..3}                 → a15 a12 a9 a6
$ echo name{1,7,9}.dat             → name1.dat name7.dat name9.dat
$ echo name{06..12..3}.dat         → name06.dat name09.dat name12.dat
```

Stosowanie nawiasów jest bardzo pomocne przy zmienianiu nazwy pliku:

```
$ mv abc.{dat,txt}                 podmiana rozszerzenia
$ mv output{,_2}.dat               wstawienie do środka numeru wersji
$ mv {,prefix_}name.out            wstawienie prefixu
```

Uwaga: powyższe nawiasy nie umieją wytwarzać ułamków dziesiętnych. Potrafi to polecenie `seq` :

```
$ seq 1 5                          → 1 2 3 4 5          (liczby jedna pod drugą)
$ seq 1 3 10                       → 1 4 7 10                 (j.w.)
$ seq 1.25 0.5 2.75                → 1,25 1,75 2,25 2,75      (j.w.)
```

Nb. możemy nie lubić przecinka jako separatora ułamków. Sposób obejścia:

```
$ LC_NUMERIC=en_US seq 1.25 0.5 2.75   → 1.25 1.75 2.25 2.75
```

⦿ Dodatek: więcej o nawiasach { } [tutaj]

Do dalszego kroku wytwórzmy serię plików (będą puste).

```
$ touch {a..d}{1..4}.txt
$ ls ??.txt
```

- nawiasy kwadratowe [ ] . Ekspandują nazwy istniejących plików (nie po-prostu stringi). Np.:

```
$ ls a[13]*                        zaakceptuje pliki mające tu w nazwie 1 lub 3
$ ls b[1-3]*                       zaakceptuje pliki mające tu w nazwie od 1 do 3
$ ls [ad][2-4]*
$ ls [!ac]*                        odrzuci pliki mające tu w nazwie litery: a lub c
```

⦿ Dodatek: więcej o nawiasach [ ] [tutaj] i [tutaj]

- Niech K będzie komendą, która wypisze tekst do `stdout` .
Istnieją operatory podstawienia: `` `K` `` oraz `$(K)` , które wkleją ten tekst w dane miejsce. Np.:

```
$ echo Poczatek `seq 1 3` koniec       → Poczatek 1 2 3 Koniec
$ echo Poczatek $(seq 1 3) koniec      → Poczatek 1 2 3 Koniec
```

⦿ Zmienne (Variables)

Bash oferuje zmienne, które są bez typów. Mogą mieścić dowolną kombinację znaków.
Jeśli zawierają tylko liczby, to mogą być traktowane jako liczby.
Jeśli w środku występują spacje, to treść trzeba objąć w ' ' lub " ".

```
$ var="123 abc"
$ echo $var                                   Aby użyć zmiennej, poprzedzamy ją $
```

Można wstawić zmienną do środka napisu. Ale trzeba uniemożliwić zlanie jej z otoczeniem: ${var}.

```
$ num=1234
$ filename=prefix_${num}.dat
$ echo $filename                              → prefix_1234.dat
```

W kombinacji z operatorem przypisania, zmienna może przechować wypis z polecenia:

```
$ mydate=`date`
$ myfiles=$(ls)
```

Aby wczytać do zmiennej (np. z klawiatury), używamy komendy read. Za opcją -p można dać monit:

```
$ read -p "Wpisz cos: " var
```

Można tej komendy użyć również do czytania ze strumienia lub pliku. Np.:

```
$ read var <<< 'To tylko przyklad.'
```

- Zasięg zmiennych. Domyślnie, zmienne utworzone w naszej sesji – istnieją tylko w niej.
  W Bashu tę klasę zmiennych nazywa się "shell variables" (zmienne powłoki).
  Tych zmiennych nie widzą: ani sesja nadrzędna, ani w terminalu obok, ani podprocesy naszej sesji.

  Ale można zmienną uwidocznić dla naszych podprocesów. Robimy to poleceniem export :

  ```
  $ export filename                           Udostępni tę zmienną podprocesom
  $ export var2=test                          Utworzy var2 i udostępni ją podprocesom
  ```

  Tę klasę zmiennych nazywa się "environment variables" (zmienne środowiskowe).

  ```
  $ ls -l | wc -l                  z listy plików, zostanie podana jej długość
  $ ls -l | xargs wc -l                dla każdego pliku, po kolei wc poda jego długość
  ```

- Linux oferuje predefiniowane zmienne środowiskowe. Niektóre z nich:

PWD	obecna ścieżkę pracy
HOME	ścieżka domowa użytkownika
USER	login użytkownika
PATH	zestaw ścieżek, na których Linux będzie szukał aplikacji
RANDOM	poda losową liczbę całkowitą z przedziału [0 .. 32767]
$	nr procesu aktualnej sesji

- Przydatne polecenia do manipulacji na nazwach plików:

```
$ readlink -f mojplik.txt
/home/login/mojplik.txt
```
rozwinie nazwę, dodając jej pełną ścieżkę

```
$ dirname /home/login/mojplik.txt
/home/login
```
wyciągnie samą ścieżkę, wycinając nazwę

```
$ basename /home/login/mojplik.txt
mojplik.txt
```
wyciągnie samą nazwę, wycinając ścieżki

```
$ basename /home/login/mojplik.txt .txt
mojplik
```
na dodatek usunie rozszerzenie `.txt`

Wynik takiej operacji możemy np. wstawić do zmiennej:

```
$ AnalysisDir=$(dirname /home/login/mojplik.txt)
```

⦿ Skrypt w Bashu: to plik z poleceniami, interpretowany przez Basha.

W pliku `hello.sh` napiszmy najprostszy skrypt Basha :

```
#!/bin/bash
echo "Hello, World!"
```

Pierwsza linia to tzw "shebang". Mówimy w niej, jaki język ma interpretować kod (tu: bash)
Pytanie, jak wywołać ten skrypt. Próba wprost (`./hello.sh`) się nie uda.
Aby zrozumieć powód, wylistujmy:

```
$ ls -og hello.sh
```

Jak widać, plikowi brak praw do wykonania.
Wkrótce je nadamy, ale na razie włączmy skrypt inaczej:

```
$ source hello.sh     lub:
$ . hello.sh
```

Na skutek takiego wywołania skryptu, kod będzie częścią naszej sesji (i procesu) .
Teraz nadajmy skryptowi prawa wywołania i następnie wywołajmy go jak każdą aplikację:

```
$ chmod 755 hello.sh
$ ./hello.sh
```

Linux otworzy dla niego podproces i w nim wywoła skrypt. Aby "poczuć" różnicę, utwórzmy zmienną (pamiętamy, że bez `export` zmienna nie przenika do podprocesu).

```
$ var=123
```

I zmieńmy skrypt na taki:

```
#!/bin/bash
echo _${var}_
```

Teraz wywołajmy skrypt na dwa odmienne sposoby:

```
$ source hello.sh                    →          _123_
$ ./hello.sh                         →          __
```

Oczywiście, można wymusić widoczność zmiennej w podprocesach:

```
$ export var
$ ./hello.sh                         →          _123_
```

- Skrypt c.d.: zmienne specjalne

  Aby umożliwić wywołanie skryptu z opcjami wejścia, w kodzie dostępne są zmienne specjalne:

  ```
  $0                    nazwa bieżącego skryptu lub sesji
  $#                    ilość opcji, jaką wpisał użytkownik przy wywołaniu skryptu
  $1 $2 $ ...           kolejne opcje wywołania
  $@                    lista opcji wywołania, oddzielonych spacją
  ```

  Napiszmy skrypt `inputarg.sh` :

  ```
  #!/bin/bash
  echo $#
  echo $0 $1 $2 $3
  ```

  i wywołajmy go tak:

  ```
  $ ./inputarg.sh -opt abc 123          →        3
                                                 -bash -opt abc 123
  ```

- Pętle. Bash ma kilka rodzajów pętli: for, while, until, select.

  Pętle są konieczne przy seryjnej obróbce danych. Również potrzebne przy posyłaniu zadań na farmę. Omówimy tu tylko pętlę for . Są dwa rodzaje takiej pętli.

  - for - jako pętla zakresowa. Iteruje ona po liście: liczb/stringów, ale i plików/ścieżek. Typowa składnia:

    ```
    for {zmienna} in {lista} ; do
      {działania}
    done
    ```

    Przykłady różnych rodzajów list w nagłówkach „zakresowego for” :

    ```
    for i in a1 b2 c3 ;
    for i in {03..12..3} ;
    for i in $(seq 3 3 12) ;
    for i in a*.txt ;              ←   po plikach w katalogu, zgodnych ze wzorcem
    for i in */ ;                  ←   po wszystkich podścieżkach katalogu
    for i in $@ ;                  ←   po opcjach wywołania
    ```

    Przykładowy skrypt z pętlą zakesową:

    ```
    #!/bin/bash
    for i in {01..05} ; do
      echo $RANDOM > myrandom_${i}.dat      ← każdemu z plików wpisze liczbę losową
    done
    ```

⦿ Pętle c.d.

Inny przykład:

```
#!/bin/bash
for i in a*.txt ; do
  readlink -f $i                ←   każdy plik z listy wypisze z pełną ścieżką
done
```

Właściwie, pętlę for można nawet napisać w 1 linii, w prompcie. To się nazywa “one-liner”.
Powyższy przykład w wariancie one-liner tworzy plik z listą nazw plików:

```
$ for i in a*.txt; do readlink -f $i ; done > myfiles.list
```

⚠ One-linery są poręczne, ale trudna czytelność może być dla nas pułapką.
A błąd przy pracy z plikami może nas sporo kosztować. Dlatego stosujmy je bardzo ostrożnie.

‣ Istnieje też pętla for - jako pętla „w stylu C” (C-style for loop). Przebiega ona po indeksie.
Typowa składnia:

```
for (( i=3; i<8; i+=2 )) ; do        ← nie poprzedzamy zmiennych znakiem $
  echo $i
done
```

⊕ Dodatek: nawiasy (( )) służą arytmetyce na liczbach całkowitych. [tutaj]
Wyczerpujące omówienie wszystkich nawiasów: [tutaj]

- Blok warunkowy if / elif / else. Jego schemat ogólny wygląda tak:

```
if {warunek logiczny} ; then
  {działania}
elif {warunek logiczny} ; then      ← opcjonalnie
  {działania}
else                                ← opcjonalnie
  {działania}
fi
```

Czym jest ten warunek logiczny i jak go kodujemy?
Warunek sprawdza relację pomiędzy liczbami czy stringami. Ale też status pliku czy zmiennej.
Dobre ściągawki z operatorami warunku są np. [tutaj] i [tutaj].
Zobaczmy kilka przykładów warunku logicznego, najpierw z operatorem `[ ]` :

```
[ 1 == 2 ]                          ← koniecznie wszędzie spacje
[ $var1 == $var2 ]
[ "$var1" == "1" ]                  ← bezpieczniej jest otoczyć obiekty cudzysłowami
![ 1 == 2 ]                         ← aby zanegować, poprzedzamy nawias znakiem !
```

Jednak operatory mniejszości/większości, które można wpisać do [ ], wyglądają nieefektownie:
`[ 1 -lt 2 ]`

Dawno temu wymyślono więc lepszą wersję tych operatorów: `[[ warunek ]]` :
`[[ 1 < 2 ]]`

A w przypadku porównywania wyrażeń arytmetycznych (w tym w zmiennych) użyć można:
`(( 1 < 2 ))`

(Pokazujemy wszystkie warianty, bo w skryptach i w internecie możecie je wszystkie zobaczyć).

Jednak warunek logiczny może dotyczyć statusu pliku/ścieżki czy zmiennej. Wypiszmy kilka:

```
[ -e {plik} ]                       ←  czy plik istnieje?
[ -d {ścieżka} ]                    ←  czy to jest katalog?
[ -v var ]                          ←  czy zmienna var istnieje?
[ -n "$var" ]                       ←  czy zmienna var ma zawartość?
```

- Blok warunkowy `if` c.d.

  Przykładowy skrypt z pętlą for i warunkiem if:

```
#!/bin/bash
for i in {a..e} ; do
  for j in {1..5} ; do
    filename=${i}${j}.txt
    if ! [ -e $filename ] ; then
      echo Creating file $filename
      touch $filename
    fi
  done
done
```

  Prosty one-liner w prompcie z blokiem if: (nie nadużywajmy tej techniki z uwagi na ryzyko błędu)

```
$ if [ -e a1.txt ] ; then echo "This file exists" ; fi
```

- W bloku warunkowym można też użyć poleceń `break` i `continue`, tak jak w C/Python:

  Przykładowy skrypt:

  #!/bin/bash

```
for i in {01..10} ; do
  filename=myrandom_${i}.dat
  if ! [ -e $filename ] ; then
    echo "File $filename does not exist."
    break
  fi
  read val <$filename
  if (( ${val} < 15000 )) ; then
    continue
  fi
  echo "In File $filename val = $val ≥ 15000"
done
```

- Z racji ograniczeń czasowych, tu kończymy naukę Basha.
  Jeżeli chcesz nauczyć się o nim więcej, w następnych krokach rozważ zapoznanie się z:
  - funkcjami (np. [tutaj] i [tutaj])
  - tablicami (arrays; np. [tutaj] i [tutaj])