

Systemy operacyjne 10

Spis treści

- 1 Budowa i wykonywanie skryptów powłoki
 - 1.1 Powłoka Bourne'a
 - 1.2 Jak to działa?
 - 1.3 Mechanizm śledzenia skryptów w czasie wykonania
 - 1.4 Pierwszy skrypt
- 2 Komunikacja skryptu z użytkownikiem
 - 2.1 Polecenie echo
 - 2.2 Polecenie read
- 3 Zmienne w skryptach
 - 3.1 Zmienne programowe
 - 3.2 Zmienne środowiskowe
 - 3.3 Zmienne specjalne
 - 3.4 Zmienne tablicowe
 - 3.5 Słowa zastrzeżone (ang. reserved words)
- 4 Cytowanie
- 5 Operacje arytmetyczne
- 6 Zadania

Budowa i wykonywanie skryptów powłoki

W każdym systemie Unix/Linux dostępnych jest kilka powłok (shell). Ich zmianę można dokonać za pomocą polecenia `chsh`, natomiast lista wszystkich dostępnych powłok i ich ścieżek zapisana jest w pliku `/etc/shells`. Za pomocą poleceń udostępnianych przez powłoki możemy w systemach unixowych tworzyć skrypty, które ułatwiają administrowanie systemem i wykonywanie powtarzających się pracochłonnych czynności.

Skrypt jest plikiem tekstowym, o pewnej strukturze, zawierającym polecenia charakterystyczne dla danej powłoki i nadanym atrybucie wykonywalności. Polecenia w nim zawarte będą wykonywane w takiej kolejności, w jakiej byłyby wpisywane z klawiatury.

Podstawowa struktura skryptu wygląda następująco:

```
#!/bin/bash
polecenie1
polecenie2
...
```

Istotną cechą skryptu jest, że zawsze zaczyna się znakami `#!` z dołączoną ścieżką powłoki w jakiej ma być wykonywany skrypt. Należy przestrzegać tego zapisu i stosować polecenia danej powłoki, ponieważ w innym przypadku skrypt nie będzie wykonywany. W trakcie tworzenia skryptu możliwe jest wykonywanie (testowanie) skryptu bez nadawania atrybutu wykonywania, w tym celu należy uruchomić go w następujący sposób:

```
. skrypt1
```

Nazwę skryptu poprzedzamy kropką i spacją a następnie wpisujemy nazwę skryptu. W przypadku wywołania skryptu w powyższy sposób, wykonywany on jest przez bieżącą powłokę. Jeśli wykonamy skrypt poprzez nadanie mu praw wykonania – zostanie utworzony nowy proces powłoki wykonujący skrypt. Różnica ta może mieć pewne znaczenie w przypadku różnych powłoki i bardziej złożonych skryptów (operujących na zmiennych).

```
#!/bin/bash
# Skrypt powitalny
echo Witaj $USER. Zalogowałeś się w systemie
echo Miłego dnia
echo Aktualnie w systemie są zalogowani następujący użytkownicy:
w
sleep 5
clear
```

Powyższy skrypt uruchamiany jest w powłoce bash (*/bin/bash*). Poleceniem echo wyświetlamy następujące po nim ciągi znaków oraz zawartości zmiennych. W tym przypadku nazwę logowanego użytkownika. Kolejne polecenie (w) wyświetla, kto jest aktualnie zalogowany, następnie odczeka 5 sekund i czyści ekran monitora (clear). Za pomocą znaku # umieszczonego w pierwszej kolumnie wiersza możemy w skrypcie umieszczać komentarze.

Przykładem bardziej zaawansowanego zastosowania skryptu jest przekopiowywanie plików z rozszerzeniem bak do nowo utworzonego katalogu:

```
#!/bin/bash
echo Podaj nazwę katalogu
read zmienna
mkdir $zmienna
for i in *
do
cp $i.bak $zmienna/$i.bak
done
```

Skrypt wyświetla komunikat i czeka na wprowadzenie nazwy katalogu (read) zapamiętanej w zmiennej *zmienna* i tworzy w bieżącej lokalizacji ten katalog. Następnie wykonuje instrukcje pętli (for) przeglądając wszystkie pliki i kopiuje pliki z rozszerzeniem bak do nowoutworzonego katalogu

Powłoki dysponują bardziej zaawansowanymi sposobami wpływania na prace systemu niż to przedstawiono, z powodu ograniczeń czasowych, w niniejszej instrukcji. By móc je wykorzystywać należy zapoznać się z ich opisem dostępnym np. w manualu bash.

Powłoka Bourne'a

Powłoka Bourne'a jest zarówno interpreterem wiersza poleceń (przetwarza wtedy wprowadzone przez użytkownika komendy), jak i zaawansowanym językiem programowania (przetwarza skrypty przechowywane w plikach). Powłoka Bourne'a jest jedną z trzech dostępnych powłok w systemach z rodziny Unix. Inne powłoki to CShell oraz KornShell. BASH (ang. Bourne Again SHell) to stworzony przez Briana Foxa i Cheta Rameya zgodny z powłoką Bourne'a (czyli sh) interpreter poleceń, łączący

w sobie zalety powłoki ksh i csh. Powłoka ta jest to najbardziej popularna powłoka używana na systemach z rodziny Linux

Jak to działa?

Skrypt jest to zwykły plik, zawierający polecenia systemowe oraz polecenia sterujące jego wykonaniem, które są po kolei przetwarzane i wykonywane przez interpreter (np. /bin/bash), który tłumaczy polecenia zawarte w skrypcie na język zrozumiały dla procesora. Przykładowy skrypt przedstawiony został poniżej. Skrypt ten został zapisany w pliku przyklad1.

Zgodnie ze standardem POSIX taki plik rozpoczyna się od: #!/bin/bash, czyli ścieżki do interpretera (w zależności od tego jakiego chcemy używać), którą należy umieścić w pierwszej linii.

Aby możliwe było uruchomienie skryptu, plik w którym jest on zapisany musi być wykonywalny (atrybut execute).

```
$cat przyklad1
#!/bin/bash
echo "To jest prosty skrypt"
echo " zawierający polecenie echo"
echo " oraz trzy inne proste komendy"
echo
ps
echo
who
echo
ls
$
```

Domyślnie plik ten nie ma ustawionego tego uprawnienia. Próba uruchomienia takiego pliku skończy się komunikatem o błędzie.

```
$przyklad1
przyklad1: execute permission denied
$
```

Po właściwym zdefiniowaniu uprawnień do tego pliku, skrypt będzie można uruchomić, co przedstawia poniższy przykład.

```
$/przyklad1
To jest prosty skrypt
zawierający polecenie echo
oraz trzy inne proste komendy
PID TTY TIME COMMAND
10443 rt02120 0:01 bash
10427 rt02120 0:04 ksh
user rt021e0 Sep 7 13:26
root rt021b0 Sep 7 14:39
```

Mechanizm śledzenia skryptów w czasie wykonania

System Linux wyposażony jest w mechanizm śledzenia skryptu w czasie wykonania. Funkcja ta jest szczególnie przydatna w czasie procesu debbugingu. Aby skorzystać z tego mechanizmu należy

skorzystać z polecenia sh z opcją -x, która włącza mechanizm śledzenia. Spróbujmy na przykładzie przedstawionego wcześniej skryptu (plik przyklad1) zobaczyć jak ten mechanizm funkcjonuje.

```
$sh -x przyklad1
+ echo To jest prosty skrypt
To jest prosty skrypt
+ echo zawierający polecenie echo
zawierający polecenie echo
+ echo oraz trzy inne proste komendy
oraz trzy inne proste komendy
+ echo
+ ps
PID TTY TIME COMMAND
10443 rt01120 0:01 bash
10427 rt02120 0:04 ksh
+ echo
+ who
user rt021e0 Sep 7 13:26
root rt02120 Sep 7 14:39
+ echo
+ ls
memo
class_notes
$
```

Komendy odczytane ze skryptu poprzedzone są symbolem plus (+), natomiast wynik działania takiej komendy wyświetlany jest w następnej linii lub liniach. Dzięki temu możemy w prosty sposób prześledzić działanie napisanego przez nas skryptu.

Pierwszy skrypt

Spróbujmy stworzyć nasz pierwszy skrypt wyświetlający na ekranie napis. W tym celu należy stworzyć odpowiedni plik, w którym umieścimy kod. Następnie przy pomocy dowolnego edytora dostępnego w systemie Linux wprowadzimy odpowiednie polecenia:

```
#!/bin/bash
#Dowolny komentarz
echo "Hello world"
```

Znak # (hash) oznacza komentarz. A więc wszystko, co znajduje się za nim w tej samej linii, jest pomijane przez interpreter.

Polecenie echo "Hello world" wydrukuje na standardowym wyjściu (stdout) napis: Hello world.

Po nadaniu naszemu skryptowi uprawnienia execute skrypt ten można uruchomić. Jeśli katalog bieżący, w którym znajduje się skrypt nie jest dopisany do zmiennej PATH i skrypt został zapisany w pliku o nazwie przykładowy_skrypt to można go uruchomić w następujący sposób:

```
./przykładowy_skrypt
```

Komunikacja skryptu z użytkownikiem

Polecenie echo

Polecenie echo umożliwia wydrukowanie na standardowym wyjściu napisu podanego jako parametr.

Składnia:

```
echo parametry tekst_do_wyświetlenia
```

Parametry:

- n
nie jest wysyłany znak nowej linii
- e
włącza interpretację znaków specjalnych:
 - \a
czyli alert,
 - \b
czyli backspace
 - \c
pomija znak kończący nową linię
 - \f
czyli escape
 - \n
znak nowej linii
 - \r
form feed
 - \t
tabulacja pozioma
 - \v
tabulacja pionowa
 - \\
backslash
 - \nnn
znak, którego kod ASCII ma wartość ósemkową
 - \xnnn
znak, którego kod ASCII ma wartość szesnastkową

Przykłady:

```
#!/bin/bash
echo -n "Napis1"
echo "Napis2"
```

Polecenie read

Polecenie read umożliwia odczytanie ze standardowego wejścia pojedynczego wiersza.

Składnia:

```
read parametry nazwa_zmiennej
```

Spróbujmy użyć polecenia read w skrypcie. Utwórzmy skrypt o nazwie odczyt. Kod tego skryptu przedstawiony jest poniżej.

```
#!/bin/bash
echo -ne "Wprowadź tekst:\a"
read wpis
echo "$wpis"
```

Po uruchomieniu otrzymamy następujący rezultat:

```
$/odczyt
Wprowadź tekst:
Jakiś tekst
Jakiś tekst
$
```

Wprowadzony przez użytkownika tekst zostanie zapisany w zmiennej wpis (polecenie read wpis). Zmienna ta zostanie następnie wypisana na ekran przy użyciu polecenia echo "\$wpis". Polecenie read pozwala również na przypisanie kilku wartości kilku zmiennym. Przedstawia to skrypt odczyt_wielu zaprezentowany poniżej.

```
#!/bin/bash
echo "Wpisz trzy wyrazy:"
read a b c
echo $a $b $c
echo "Wartość zmiennej a to: $a"
echo "Wartość zmiennej b to: $b"
echo "Wartość zmiennej c to: $c"
```

Spróbujmy przy użyciu mechanizmu śledzenia, zobaczyć jak działa ten skrypt.

```
$sh -x odczyt_wielu
+ echo 'Wpisz trzy wyrazy:'
Wpisz trzy wyrazy:
+ read a b c
to sa wiecej niz trzy wyrazy
+ echo to sa wiecej niz trzy wyrazy
to sa wiecej niz trzy wyrazy
+ echo 'Wartość zmiennej a to: to'
Wartość zmiennej a to: to
+ echo 'Wartość zmiennej b to: sa'
Wartość zmiennej b to: sa
+ echo 'Wartość zmiennej c to: wiecej niz trzy wyrazy'
Wartość zmiennej c to: wiecej niz trzy wyrazy
$
```

W skrypcie tym widać, iż spacje pomiędzy wyrazami są separatorami dla wartości przypisanych do zmiennych a, b oraz c. Do zmiennej a została przypisana wartość „to”, do zmiennej b została przypisana wartości „sa”, natomiast dla zmiennej c pozostały fragment wprowadzonej przez użytkownika linii (czyli „wiecej niz trzy wyrazy”).

Wybrane parametry polecenie read

-p

Pokaże znak zachęty bez kończącego znaku nowej linii

```
#!/bin/bash
read -p "Pisz:" odp
echo "$odp"
```

-a

Kolejne wartości przypisywane są do kolejnych indeksów zmiennej tablicowej

```
#!/bin/bash
echo "Podaj elementy zmiennej tablicowej:"
read tablica
echo "${tablica[*]}"
```

-e

Jeśli nie podano żadnej nazwy zmiennej, wiersz trafia do \$REPLY

```
#!/bin/bash
echo "Wpisz coś:"
read -e
echo "$REPLY"
```

Zmienne w skryptach

W językach skryptowych wyróżniamy kilka rodzajów zmiennych. Są to

- zmienne programowe;
- zmienne środowiskowe;
- zmienne specjalne;
- zmienne tablicowe.

Zmienne programowe

Zmienne programowe są to zmienne, które zostały zdefiniowane samodzielnie przez użytkownika. W skrypcie definiujemy je pisząc po prostu:

```
nazwa_zmiennej="wartość"
```

np.:

```
x="napis",
```

a odwołujemy się do niej poprzez polecenie:

```
$nazwa_zmiennej,  
np.  
read x  
echo $x
```

Do wyzerowania zmiennej używamy polecenia:

```
nazwa_zmiennej=.
```

Należy pamiętać, że **nie może być spacji** po obu stronach znaku =, dlatego też linia `x = "napis"` jest błędna

Za zmienną można również podstawić polecenie umieszczone w tzw. odwrotnych apostrofach (```). W czasie wykonywania takiego skryptu, interpreter zastąpi komendę, która się tam znajduje, wynikiem jej działania. Przedstawia to poniższy przykład.

```
$cat listuj  
#!/bin/bash  
dir=`pwd`  
echo 'Aktualnie korzystasz z katalogu ' $dir 'na Twoim dysku'  
$
```

Efekt działania powyższego przykładu przedstawia poniższy wydruk:

```
$/listuj  
Aktualnie korzystasz z katalogu /home/user/skrypt na Twoim dysku  
$
```

Przykład ten pokazał dwa istotne elementy. Pierwszy z nich to mechanizm podstawiania poleceń. Drugi element, na który warto zwrócić uwagę, to sklejanie łańcuchów znaków, z którego skorzystaliśmy w tym przykładzie.

Istnieje jeszcze jeden sposób podstawiania poleceń. Zrealizowane to może zostać przy użyciu mechanizmu rozwijania zawartości nawiasów: `$(polecenie)`. Działanie tego mechanizmu wygląda podobnie, jak poprzednio, co pokazuje poniższy przykład.

```
$cat listuj2  
#!/bin/bash  
dir=$(pwd)  
echo "Aktualnie korzystasz z katalogu $dir na Twoim dysku"  
$
```


Zmienne środowiskowe

Zmienne środowiskowe służą do zdefiniowania środowiska użytkownika dostępnego dla wszystkich procesów potomnych. Dzielimy je na:

- globalne
- lokalne.

Poniższy przykład obrazuje różnicę między nimi. W tym celu należy uruchomić polecenie xterm (terminal w środowisku X Window) i wpisać:

```
x="Przykładowy napis"  
echo $x  
xterm
```

Ostatnie polecenie spowoduje wywołanie kolejnego terminala. Jeśli teraz w nowym terminalu wydamy polecenie echo \$x, to nie zobaczymy nic, gdyż zmienna x jest zmienną lokalną, niewidoczną w innych podpowłokach.

Aby zmienną x uczynić globalną, należy wydać polecenie export x="napis", dzięki czemu będzie ona widoczna w innych podpowłokach. Polecenie export nadaje zmiennym atrybut zmiennych globalnych, natomiast wywołane bez parametrów wyświetla listę aktualnie eksportowanych zmiennych.

W przypadku powłoki bash na liście tej pojawi się polecenie declare. Jest to wewnętrzne polecenie tej powłoki służące do definiowania zmiennych i nadawania im odpowiednich atrybutów (-x parametr odpowiadający poleceniu export). Należy pamiętać, że polecenie to dostępne jest jedynie dla powłoki bash i nie występuje w innych powłokach.

Przykładowe zmienne środowiskowe to m.in.:

```
$HOME #ścieżka do katalogu domowego użytkownika  
$USER #login użytkownika  
$HOSTNAME #nazwa hosta na który zalogowany jest użytkownik  
$OSTYPE #rodzaj systemu operacyjnego
```

Wszystkie dostępne zmienne środowiskowe można wyświetlić za pomocą polecenia:

```
printenv | more
```

Zmienne specjalne

Zmienne specjalne są to najbardziej prywatne zmienne powłoki, które są udostępniane użytkownikowi tylko do odczytu (z pewnymi wyjątkami). Poniżej przedstawiono kilka przykładów:

`$0`

nazwa bieżącego skryptu lub powłoki. Przykład:

```
#!/bin/bash
echo "$0"
Skrypt ten wyświetli nazwę naszego skryptu.
```

`$1..$9`

Parametry przekazywane do skryptu (użytkownik może modyfikować ten rodzaj

zmiennych specjalnych.

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

Wywołanie tego skrypt z parametrami spowoduje, że zostaną one przypisane zmiennym od \$1 do \$9.

`$@`

Pokaże wszystkie parametry, które zostały przekazywane do skryptu (możliwość

modyfikacji), równoważne `$1 $2 $3...`, jeśli nie podane są żadne parametry `$@` interpretowany jest jako pusty:

```
#!/bin/bash
echo "Skrypt uruchomiono z parametrami: $@"
```

`$?`

kod powrotu ostatnio wykonywanego polecenia

`$$`

PID procesu bieżącej powłoki

Zmienne tablicowe

BASH pozwala na stosowanie jednowymiarowych zmiennych tablicowych. W BASH'u nie ma górnego ograniczenia rozmiaru tablic. Kolejne wartości zmiennej tablicowej indeksowane są przy pomocy kolejnych liczb całkowitych, zaczynających się od 0.

Składnia: **zmienna=(wartości wartosc2 wartoscJ ... wartoscN)**

Przykład:

```
#!/bin/bash
tablica=(element1 element2 element3)
echo ${tablica[0]}
echo ${tablica[1]}
echo ${tablica[2]}
```

Zadeklarowana zmienna tablicowa o nazwie: *tablica*, zawiera trzy elementy: *element1 element2 element3*. Polecenie: **echo \$ {tablica [0]}** wyświetli na ekranie pierwszy elementu tablicy. W powyższym przykładzie w ten sposób wypisana zostanie cała zawartość tablicy. Do elementów tablicy odwołujemy się za pomocą indeksów.

`\${nazwa_zmiennej[indeks]}`

Indeksy to liczby od 0 do n-1 (gdzie n jest liczbą elementów tablicy) oraz @ i *. Gdy odwołując się do zmiennej nie podamy indeksu: **`\${nazwa_zmiennej}`** to nastąpi odwołanie do elementu tablicy o indeksie 0. Jeśli jako indeks podamy @ lub * to zwrócone zostaną wszystkie elementy tablicy.

Przykład:

```
#!/bin/bash
tablica=(element1 element2 element3)
echo `${tablica[*]}`
```

BASH umożliwia też uzyskanie informacji o długości (liczbie znaków) danego elementu tablicy:

`\${#nazwa_zmiennej[indeks]}`.

Przykład:

```
#!/bin/bash
tablica=(element1 element2 element3)
echo `${#tablica[0]}`
```

Polecenie echo **`\${##tablica[0]}`** wydrukuje liczbę znaków, z jakich składa się pierwszy element tablicy (czyli 8). W podobny sposób można otrzymać liczbę wszystkich elementów tablicy - wystarczy jako indeks podać @ lub *. Przykład:

```
#!/bin/bash
tablica=(element1 element2 element3)
echo `${#tablica[@]}`
```

Aby zmienić lub dodać element tablicy należy użyć następującej składni:

nazwa_zmiennej[indeks]=wartosc.

Przykład:

```
#!/bin/bash
tablica=(element1 element2 element3)
tablica[3]=element4
echo `${tablica[@]}`
```

Dany element tablicy usuwa się za pomocą polecenia **unset**, o składni:

unset nazwa_zmiennej[indeks]

Przykład:

```
#!/bin/bash
tablica=(element1 element2 element3)
echo ${tablica[@]}
unset tablica[2]
echo ${tablica[*]}
```

Usunięty został ostatni element tablicy. Aby usunąć całą tablicę wystarczy podać jako indeks: @ lub *. Przykład:

```
#!/bin/bash
tablica=(element1 element2 element3)
unset tablica[*]
echo ${tablica[@]}
```

Słowa zastrzeżone (ang. *reserved words*)

Słowa zastrzeżone mają dla powłoki specjalne znaczenie i nie mogą być wykorzystywane jako nazwy zmiennych, funkcji oraz plików zawierających skrypty. Poniżej przedstawiona jest lista tych słów:

- !
- case
- do
- done
- elif
- else
- esac
- fi
- for
- function
- if
- in
- select
- then
- until
- while
- {
- }
- time
- [
-]
- test

Cytowanie

Znaki cytowania pozwalają na wyłączenie mechanizmu interpretacji znaków specjalnych przez powłokę. Wyróżniamy następujące znaki cytowania:

cudzysłów (ang. double quote) - " "

Między cudzysłowami umieszcza się dowolny tekst oraz zmienne poprzedzielane spacjami. Cudzysłowy zachowują znaczenie specjalne trzech znaków:

- \$ - wskazuje na nazwę zmiennej, umożliwiając podstawienie jej wartości
- \ - znak maskujący
- `` - odwrotny apostrof, umożliwia zacytowanie polecenia

```
#!/bin/bash
x=2
echo "Wartość zmiennej x to $x"
echo -ne "Dzwonek\a"
echo "Polecenie date pokaże: `date`"
```

apostrof (ang. single quote) - ' '

Elementy ujęte w symbol apostrofu traktowane są jak łańcuch tekstowy. Apostrof wyłącza interpretowanie wszystkich znaków specjalnych.

```
#!/bin/bash
echo '$USER' #nie wypisze twojego loginu
```

odwrotny apostrof (ang. backquote) - ``

Pozwala na zacytowanie polecenia (przydatne gdy chcemy podstawić pod zmienną wynik działania jakiegoś polecenia):

```
#!/bin/bash
x=`ls -la $PWD`
echo $x
```

Powłoka *bash* udostępnia również alternatywny zapis podstawienia polecenia, w postaci \$(...).
Na przykład:

```
x=$(ls -la $PWD)
```

Znak maskujący (ang. backslash) - \

Symbol ten jest przydatny, gdy chcemy na ekranie wyświetlić się napis jakiś symbol dla systemu zastrzeżony np. napis \$HOME. Obrazuje to poniższy przykład.

```
echo "$HOME" #na ekranie /home/katalog_domowy_uzytkownika
echo \$HOME #na ekranie pojawi się napis $HOME
```

Pierwsza linia wypisze na ekranie katalog domowy użytkownika, który uruchomił ten skrypt, w drugiej linii dzięki użyciu \ została wyłączona interpretacja przez powłokę zmiennej \$HOME.

Operacje arytmetyczne

Podstawą operacji arytmetycznych w powłoce jest polecenie `expr`. `expr` wykonuje obliczenie i zapisuje rezultat na standardowe wyjście. Każdy żeton wyrażenia musi być oddzielnym argumentem. Operandy mogą być liczbami lub ciągami znaków. Łącuchów znaków nie cytuje się dla `expr`, choć możesz być zmuszonym do tego, by ochronić je przed powłoką (znaki i łańcuchy o specjalnym znaczeniu dla powłoki, np. spacja). Przykład:

```
$ expr 2 + 2
4
$ expr \( 12 + 43 \) / 3
18
$ expr 2 == 3
0
$ expr 4 \> 5
0
```

Można także użyć `expr` do wykonywania operacji na zmiennych:

```
$ a=2
$ a=`expr $a + 1`
$ echo $a
3
```

Dopuszczalne są następujące operacje:

- operacje arytmetyczne na liczbach całkowitych

`+`, `-`, `*`, `/`;

- operacje logiczne. Wartość prawdziwa oznaczane jest liczbą 1, zaś fałszywa 0 (!)

`<`, `<=`, `=`, `==`, `!=`, `>=`, `>`

Operatory o specjalnym znaczeniu dla powłoki (np. `<` lub `*`) należy poprzedzić znakiem `\`.

Powłoka `bash` udostępnia alternatywny, znaczenie wygodniejszy zapis w postaci `$((...))`. Operacja wewnątrz nawiasów zgodna jest ze składnią języka C. Na przykład:

```
$ echo $((2+2))
4
$ echo $(((12+43)/3))
18
$ echo $((2==3))
0
$ echo $((4>5))
0
$ a=2
$ a=$((a+1))
$ echo $a
3
```

Zadania

- Co to jest skrypt?
- W jaki sposób można wykorzystać mechanizm śledzenia skryptów?
- Jakie uprawnienia powinien posiadać plik zawierający skrypt, aby możliwe było jego uruchomienie?
- Co oznacza zmienna środowiskowa \$USER ?
- Jakie informacje przechowuje zmienna specjalna \$\$?
- Co oznacza znak # w skrypcie?
- Co powoduje dodanie do polecenia echo opcji -e ?
- Gdzie zostanie umieszczony wprowadzony tekst po wydaniu polecenia read -e ?
- Napisz skrypt, który wypisze na ekran twoje imię i nazwisko oraz zawartość twojego katalogu domowego.
- Napisz skrypt, który będzie Cię pytał o datę urodzin, a następnie wyświetlał ją na ekranie w postaci komunikatu: Urodziłeś się
- Skrypt z poprzedniego zadania zmodyfikuj tak, aby data urodzin była wprowadzana w postaci parametru.
- Napisz skrypt, który poprosi Cię o wpisanie dowolnego zdania, następnie poprosi o numer wyrazu w zdaniu i wyświetli ten wyraz.
- Napisz skrypt wyświetlający bieżący czas.
- Napisz skrypt, który będzie pytał się o nazwę użytkownika a następnie zakładał mu konto wraz z katalogiem domowym.
- Napisz skrypt, który będzie kopiował wybrany plik do katalogu, o nazwie w formacie daty: yy.mm.dd. Nazwa pliku ma być podawana jako parametr skryptu.