**Linux Shell Scripting: A project-based approach to learning**

24 June 2021

22:51

**ELEVENTH SCRIPT**

* Parsing command line options with getopts (shell built in)
* Getopts is used by shell procedures to parse positional parameters as options
* Syntax:

Getopts optstring name [arg]

* Getopts lets the user to specify flags such as -v -l or -vl followed by a command line argument while executing their bash scipt
* any option which needs to be have a mandatory value can be followed by a colon. Like in the luser-demo11a.sh script's case, '-l' option is a mandatory option
* If the statement is passed like:
  + while getopts vl:s OPTION

Then,

-ls is not equal to passing -sl since, -l requires an option and not a flag.

[vagrant@localusers vagrant]$ ./luser-demo11.sh -ls 8

head: s: invalid number of bytes

--working fine below:---

[vagrant@localusers vagrant]$ ./luser-demo11.sh -sl 8

9f3e81f6@

* Otherwise, normal if there is no mandatory flag then -ls is same as -ls
* Arithematic Expansion: NUM=$((1+2))
* Bash doesn’t support floating point arithmetic. In the sense, that it will just drop of the decimal point value.
* If we would like to work with floating point values or decimals then, the way to do it is to use an external program called bc. Also, called as Basic Calculator.
* To install bc:
  + Sudo yum install bc

[vagrant@localusers vagrant]$ NUM=$((1+2))

[vagrant@localusers vagrant]$ echo ${NUM}

3

[vagrant@localusers vagrant]$ NUM=$((10-1))

[vagrant@localusers vagrant]$ echo ${NUM}

9

[vagrant@localusers vagrant]$ NUM=$((10\*1))

[vagrant@localusers vagrant]$ echo ${NUM}

10

[vagrant@localusers vagrant]$ NUM=$((10/2))

[vagrant@localusers vagrant]$ echo ${NUM}

5

[vagrant@localusers vagrant]$ NUM=$((10/3))

[vagrant@localusers vagrant]$ echo ${NUM}

3

* Bc stands for basic calculator

[vagrant@localusers vagrant]$ type -a bc

bc is /usr/bin/bc

[vagrant@localusers vagrant]$ echo '6/4' | bc -l (-l option is used for using predefined math routines)

1.50000000000000000000

[vagrant@localusers vagrant]$ bc -l

bc 1.06.95

Copyright 1991-1994, 1997, 1998, 2000, 2004, 2006 Free Software Foundation, Inc.

This is free software with ABSOLUTELY NO WARRANTY.

For details type `warranty'.

5/3

1.66666666666666666666

1+2

3

^C

(interrupt) Exiting bc.

* We can also us awk to perform basix calculations:
* AWK is a pattern scanning and processing language
* Awk: Gawk is the GNU Project's implementation of the AWK programming language.
* It conforms to the definition of the language in the POSIX 1003.1 Standard.

This version in turn is based on the description in The AWK Programming Language, by Aho, Kernighan, and Weinberger.

* Gawk provides the additional features found in the current version of UNIX awk and a number of GNU-specific extensions.

[vagrant@localusers vagrant]$ awk 'BEGIN {print 6/4}'

1.5

[vagrant@localusers vagrant]$ DICEA='3'

[vagrant@localusers vagrant]$ DICEB='4'

[vagrant@localusers vagrant]$ TOTAL-$((DICEA+DICEB))

-bash: TOTAL-7: command not found

[vagrant@localusers vagrant]$ TOTAL=$((DICEA+DICEB))

[vagrant@localusers vagrant]$ echo ${TOTAL}

[vagrant@localusers vagrant]$

* '$' is used for substitution and here is DICEA and DICEB inside the double parenthesis we dont want that.

we are looking towards just changing the values of the variables and not substitution

[vagrant@localusers vagrant]$ NUM='1'

[vagrant@localusers vagrant]$ $(( NUM++ ))

-bash: 1: command not found

[vagrant@localusers vagrant]$ echo ${TOTAL}

8

[vagrant@localusers vagrant]$

[vagrant@localusers vagrant]$ echo ${NUM}

2

[vagrant@localusers vagrant]$ NUM='3'

[vagrant@localusers vagrant]$ $(( NUM-- ))

-bash: 3: command not found

[vagrant@localusers vagrant]$ echo ${NUM}

2

[vagrant@localusers vagrant]$ (( NUM += 5 ))

[vagrant@localusers vagrant]$ echo ${NUM}

7

[vagrant@localusers vagrant]$ NUM=$(( NUM+=5 ))

[vagrant@localusers vagrant]$ echo ${NUM}

* let: let arg [arg ...]

Evaluate arithmetic expressions.

Evaluate each ARG as an arithmetic expression. Evaluation is done in

fixed-width integers with no check for overflow, though division by 0

is trapped and flagged as an error. The following list of operators is

grouped into levels of equal-precedence operators. The levels are listed

in order of decreasing precedence.

id++, id-- variable post-increment, post-decrement

++id, --id variable pre-increment, pre-decrement

-, + unary minus, plus

!, ~ logical and bitwise negation

\*\* exponentiation

\*, /, % multiplication, division, remainder

* [vagrant@localusers vagrant]$ let NUM='78+54'

[vagrant@localusers vagrant]$ echo ${NUM}

132

* Expr is a built in command similar to bs (basic calculator)

[vagrant@localusers vagrant]$ type -a expr

expr is /usr/bin/expr

[vagrant@localusers vagrant]$ expr 1 + 1

2

[vagrant@localusers vagrant]$ NUM=$( expr 9 + 6)

[vagrant@localusers vagrant]$ echo ${NUM}

15

* Getopts does not change positional parameters. This means that the options are stored in $@, $1, $2 and $3
* OPTIND

# It is used to give the position of the next command line argument following the options

echo "OPTIND: ${OPTIND}"

See output below for OPTIND:

[vagrant@localusers vagrant]$ ./luser-demo11b.sh -sl 8 extra-stuff

Number of arguments: 3

All args: -sl 8 extra-stuff

First arg: -sl -> 1st, 2nd

Second arg: 8  -> 3rd hence OPTIND is 3

Third arg: extra-stuff

OPTIND: 3 <- because the first cmd arg 8 is at position 3 following the options -sl

9589f77e(

-------------------

[vagrant@localusers vagrant]$ ./luser-demo11b.sh -s extra-stuff

Number of arguments: 2

All args: -s extra-stuff

First arg: -s

Second arg: extra-stuff

Third arg:

OPTIND: 2

4fbb9d3cab05a3ebd449c0014a794ac4b272461a4babe216$