6.1

Shell programming

More features in bash scripts

- Debugging
  - Print the command before executing it
    - `set -x ./myscript.sh`
    - `bash -x ./myscript.sh`
  - Place the `set -x` inside the script
- Combining conditions
  - `[ $a -gt 3 ] && [ $a -gt 7 ]`
  - `[ $a -lt 3 ] || [ $a -gt 7 ]`
  - Example condition-examples.sh
- Shell arithmetic
  - Using `let`
    - `let a=$1+5`
    - `let "a= $1 + 5"`
  - Using `expr`
    - `expr $a +20`
    - `expr "$a + 20"`
    - `b=$( expr $a + 20 )`
  - Using `$[ expression ]`
    - `b=$[ $a + 10 ]`
  - Using `$(( expression ))`
    - `b=$(( $a + 10 ))`
    - `(( b++ ))` - Without $. Not intending to return. Useful for incrementing
  - Example arithmetic-example-1.sh

- ## expr command operators

Expression	Description
`a + b`	Return arithmetic sum of a and b
`a - b`	Return arithmetic difference of a and b
`a * b`	Return arithmetic product of a and b
`a / b`	Return arithmetic quotient of a divided by b
`a % b`	Return arithmetic remainder of a divided by b
`a > b`	Return 1 if a greater than b; else return 0

Expression	Description
`a >= b`	Return 1 if a greater than or equal to b; else return 0
`a < b`	Return 1 if a less than b; else return 0
`a <= b`	Return 1 if a less than or equal to b; else return 0
`a = b`	Return 1 if a equals b; else return 0
`a | b`	Return a if neither argument is null or 0; else return b
`a & b`	Return a if neither argument is null or 0; else return 0
`a != b`	Return 1 if a is not equal to b; else return 0
`str : reg`	Return the position upto anchored pattern match with BRE str
`match str reg`	Return the pattern match if reg matches pattern in str
`substr str n m`	Return the substring m chars in length starting at position n
`index str chars`	Return position in str where any one of chars is found else return 0
`length str`	Return numeric length of string str
`+ token`	Interpret token as string even if its a keyword
`(exprn)`	Return the value of expression exprn

- Example expr-examples.sh
- Bench Calculator
  - An arbitaty preciscion calculator language
  - `bc -l`
    - the `l` option loads the math library.
  - `12^6` or `12.6/3.6`
  - Can be used for floating point operations.
- heredoc feature
  - helps while passing long strings without having to worry about `\n` etc.
  - Example heredoc-example-1.sh
  - Example heredoc-example-2.sh
  - A hyphen tells bash to ignore leading tabs
- PATH variable
  - Example path-example.sh
    - IFS (Internal Field Seperator)

L6.3

- if-elif-else-fi loop

```
if condition1
then
        commandset1
else
        commandset2
fi
```

```
if condition1
then
        commandset1
elif condition2
then
        commandset2
elif condition3
then
        commandset3
else
        commandset4
fi
```

- case statement options
    - commandset4 is the default for values of $var not matching what are listed

```
case $var in
        op1)
                commandset1;;
        op2 | op3)
                commandset2;;
        op4 | op5 | op6)
                commandset3;;
        *)
                commandset4;;
esac
```

- c style for loop : one variable
    - extention of POSIX and maynot be available in all the shells
    - Adding `time` before a script command gives the amount of time taken to execute.

```
begin=1
finish=10
for (( a = $begin; a < $finish; a++ ))
do
        echo $a
done
```

- c style for loop : two variables
  - Note: Only one condition to close the for loop

```
begin1=1
begin2=10
finish=10
for (( a=$begin1, b=$begin2; a < $finish; a++, b-- ))
do
        echo $a $b
done
```

- processing output of a loop
  - Output of the loop is redirected to the tmp file

```
filename=tmp.$$
begin=1
finish=10
for (( a = $begin; a < $finish; a++ ))
do
        echo $a
done > $filename
```

- break
  - Break out of inner loop

```
n=10
i=0
while [ $i -lt $n ]
do
        echo $i
        (( i++ ))
        if [ $i -eq 5 ]
        then
                break
        fi
done



- Break out of outer loop



n=10
i=0
while [ $i -lt $n ]
do
        echo $i
        j=0
        while [ $j -le $i ]
        do
                printf "$j "
                (( j++ ))
                if [ $j -eq 7 ]
                then
                        break 2
                fi
        done
        (( i++ ))
done
```

- continue
  - Continue will skip rest of the commands in the loop and goes to next iteration

```
n=9
i=0
while [ $i -lt $n ]
do
        printf "\n loop $i:"
        j=0
        (( i++ ))
        while [ $j -le $i ]
        do
                (( j++ ))
                if [ $j -gt 3 ] && [ $j -lt 6 ]
                then
                        continue
                fi
                printf "$j "
        done
done
```

- shift
  - shift will shift the command line arguments by one to the left.
  - shift is destructive - after the arguments are shifted t the left they are gone. This is only helpful if you don't need the arguments later.
  - n checks if it is a non-zero argument
  - except $0, which is the name of the script, the rest of the arguments get popped

```
i=1
while [ -n "$1" ]
do
        echo argument $i is $1
        shift
        (( i++ ))
done
```

- exec

  - `exec ./my-executable --my-options --my-args`
  - To replace shell with a new program or to change i/o settings
  - If new program is launched successfully, it will not return control to the shell
  - If new program fails to launch, the shell continues

- eval

  - `eval my-arg`
  - Execute argument as a shell command
  - Combines arguments into a single string
  - Returns control to the shell with exit status
  - Example eval-example.sh

- function

  - Example function-example.sh

- getopts

  - This script can be invoked with only three options: `a` , `b` , `c` . The options `b` and `c` will take arguments.
  - Example getopts-example.sh

```
while getopts "ab:c:" options;
do
        case "${options}" in
                b)
                        barg=${OPTARG}
                        echo accepted: -b $barg
                        ;;
                c)
                        carg=${OPTARG}
                        echo accepted: -c $carg
                        ;;
                a)
                        echo accepted: -a
                        ;;
                *)
                        echo Usage: -a -b barg -c carg
                        ;;
        esac
done
```

- select loop
  - Text Menu
  - Example select-example.sh

```
echo select a middle one
select i in {1..10}
```

```
do
        case $i in
                1 | 2 | 3)
                        echo you picked a small one;;
                8 | 9 | 10)
                        echo you picked a big one;;
                4 | 5 | 6 | 7)
                        echo you picked the right one
                        break;;
        esac
done
echo selection completed with $i
```

- Additional notes
  - Warning : Never `eval` a user supplied string on any command line
  - eval is sending the strings to the shell and printing them out.
  - Can source a file with functions to use it in a shell script
  - `source mylib.sh`
  - Do not give set uid permission to the scripts unless you know what you are doing

L6.4

awk

A language for processing fields and records

- Introduction
  - awk is a programming language, quick to code and fast in execution
  - awk is an abbreviation of the names of three people who developed it: **A**ho, **W**einberger & **K**ernighan
  - It is a part of POSIX, IEEE 1003.1-2008
  - Variants: nawk, gawk, mawk ...
  - gawk contains features that extend POSIX (normally seen on GNU Linux systems with a symbolic link from awk to gawk)
  - Though awk is viewed as a scripting language it has enough mathematical functions to use for routine calculations. IT can do things that spreadsheets cannot do.
- Execution model
  - Input stream is a set of records
  - Eg., using "\n" as record separator, lines are records
  - Each record is a sequence of fields
  - Eg., using " " as field separator, words are fields. Evan a regular expression can be used as an FS.
  - Splitting of records to fields is done automatically

- Each code block executes on one record at a time, as matched by the pattern of that block

- Usage
  - Single line at the command line
    - `cat /etc/passwd | awk -F”:” ‘{print $1}’`
  - Script interpreted by awk
    - `./myscript.awk /etc/passwd`
    - `myscript.awk`

```
#!/usr/bin/gawk -f
BEGIN {
       FS=":"
       }
{
       print $1
}
```

- Examples
  - block-ex-1.awk
    - `./block-ex-1.awk block-ex-1.input`
    - `cat block-ex-1.input | ./block-ex-1.awk`
    - For each line Default block will be processed once.
    - You can have as many begin and end blocks wherever required in the awk script. BEGIN will be processed before the default block and and END will be processed after the default block.
    - We don't need `;` at the end of every statement unless you need to write multiple statements on a single line.
    - `$0` represents the line(record) which is currently being processed.

- Built-in variables

Variable	Description
`ARGC`	Number of arguments supplied on the command line (except those that came with -f & -v options)
`ARGV`	Array of command line arguments supplied; indexed from 0 to ARGC-1
`ENVIRON`	Associative array of environment variables
`FILENAME`	Current filename being processed
`FNR`	Number of the current record, relative to the current file
`FS`	Field separator, can use regex
`NF`	Number of fields in the current record
`NR`	Number of the current record
`OFMT`	Output format for numbers

Variable	Description
`OFS`	Output fields separator
`ORS`	Output record separator
`RS`	Record separator
`RLENGTH`	Length of string matched by match() function
`RSTART`	First position in the string matched by match() function
`SUBSEP`	Separator character for array subscripts
`$0`	Entire input record
`$n`	nth field in the current record

awk scripts

`pattern {procedure}`

- pattern (optional. If not given the code block is called default block and it is applied to every line in the input stream.)
  - **BEGIN**
  - **END**
  - general expression
  - **regex**
  - Relational Expression
  - Pattern-matching expression
- procedure (will be applied to all recors that match the pattern)
  - Variable assignment
  - Array assignment
  - Input / output commands
  - Build-in functions
  - User-defined functions
  - Control loops

Execution

- `BEGIN { commands; }`
  - Executed once, before files are read
  - Can appear anywhere in the script
  - Can appear multiple times
  - Can contain program code
- `END { commands; }`
  - Executed once, after files are read
  - Can appear anywhere in the script
  - Can appear multiple times

        - Can contain program code
    - `pattern { commands; }`
        - Patterns can be combined with `&&` `||` `!`
        - Range of records can be specified using comma
        - Executed each record pattern evalutes to true
        - Script can have multiple such blocks
    - `{ commands; }`
        - Executed for all records
        - Can have multiple such blocks

operators

- Assignment
    - `=` `+=` `-=` `*=` `/=` `%=` `^=` `**=`
- Logical
    - `||` `&&`
- Algebraic
    - `+` `-` `*` `/` `%` `^` `**`
- Relational
    - `>` `<=` `>` `>=` `!=` `==`

Operation	Description
`expr ? a : b`	Conditional expression
`a in array`	Array membership
`a ~ /regex/`	Regular expression match
`a !~ /regex/`	Negation of regular expression match
`++`	Increment, both prefix and postfix
`--`	decrement, both prefix and postfix
`$`	Field reference
	Blank is for concatenation

- Adding 0 to a string makes it get interpreted as a number.

Functions and commands

Operation	Commands
Arithmetic	`atan2` `cos` `exp` `int` `log` `rand` `sin` `sqrt` `srand`
String	`asort` `asorti` `gsub` `index` `length` `match` `split` `sprintf` `strtonum` `sub` `su`
Control Flow	`break` `continue` `do` `while` `exit` `for` `if` `else` `return`
Input / Output	`close` `fflush` `getline` `next` `nextline` `print` `printf`

Operation	Commands
Programming	`extension` `delete` `function` `system`
bit-wise	`and` `compl` `lshift` `or` `rshift` `xor`

- Example
  - block-ex-2.awk
  - block-ex-3.awk
    - Blocks get executed based on whether the line has `alpha` , `alnum` or `digits`
  - block-ex-4.awk
    - Matching only the first field in the record with a pattern
  - block-ex-5.awk
    - Field Separator as regular expression
    - Number of fields as condition

arrays

- Associative arrays
- Sparse storage
- Index need not be integer
- `arr[index]=value`
- `for (var in arr)`
- `delete arr[index]`

Loops

```
for (a in array)
{
	print a
}



if (a > b)
{
	print a
}



for (i=1;i<n;i++)
{
	print i
}



while (a < n)
{
        print a
}


do
{
        print a
} while (a<n)
```

Functions

- `cat infile |awk -f mylib -f myscript.awk`
- `mylib`

```
function myfunc1()
{
        printf “%s\n”, $1
}
function myfunc2(a)
{
        return a*rand()
}
```

- `myscript.awk`

```
BEGIN
{
        a=1
}
{
        myfunc1()
        b = myfunc2(a)
        print b
}
```

Pretty printing

- `printf "format", a, b, c`
  - format - `%[modifier]control-letter`
    - modifier
      - `width`
      - `prec`
      - `-`
    - control-letter
      - `c` ascii char
      - `d` integer
      - `i` integer

    - `e` scientific notation
    - `f` floating notation
    - `g` shorter of scientific & float
    - `o` octal value
    - `s` string text
    - `x` hexadecimal value
    - `X` hexadecimal value in caps

🔗 bash + awk

- Including awk inside shell script
- heredoc feature
- Use with other shell scripts on command line using pipe