Chapter 1

Bash scripts

1.1 For very beginners

Most of this scripts might not be so helpful for you, but you should get the idea how to create something complex from them. And most of scripts would be one-liners.
Let's start from simple rules:

- Most of this scripts would be executed inside "putty" or any other tool that allows you to access Solaris (or Linux) environment.
- You can join output from one command with input of another, so called 'pipe'.
- You can join few commands into one line, if you put ';' between them

1.2 Basic commands

Below are some basic day-to day commands, which are mostly related to logs management. Use them as building blocks for your scripts.

Listing 1.1 : Clear file contents

```
echo "" >  /file_to_clean.log;
```

Listing 1.2 : Comress contents of folder

```
tar -cvf name_of_arc.tar *;
gzip name_of_arc.tar;
```

Listing 1.3 : Comress only one file

```
gzip file_to_compress;
```

Listing 1.4 : Example of 2 joined commands

```
cd /home/Sockets/; ./sockets.sh;
```

Listing 1.5 : If your command fails - you can put error message and continue script execution

```
rm access.log0* || echo "no access.log0* files";
```

Listing 1.6 : Pipe example - result of process search is output to filter grep

```
ps -aef | grep managed
```

Listing 1.7 : Get current date in particular format

```
date '+%d-%m-%Y %H:%M:%S'
```

Listing 1.8 : List file contents

```
cat somefile > resultfile.txt
```

Listing 1.9 : Output result to file

```
cat somefile > resultfile.txt
```

Listing 1.10 : Output result to file, append

```
cat somefile >> resultfile.txt
```

Listing 1.11 : Monitor file changes, from 15 last lines

```
tail -f somefile
```

Listing 1.12 : Output result to file, but show it on screen at same time

```
tail -f somefile | tee resultfile.txt
```

1.3 Variables and Functions

Listing 1.13 : Getting result from function in one line or multiple lines

```
declare RESULT=$(./myscript)
echo "$RESULT" #To get multiple in lines
echo $RESULT #To get in one line
```

Listing 1.14 : Defining function

```
function function_name {
    echo "Parameter #1 is $1"
    echo "Parameters count = is $#"
    echo "All params in one variable = is $*"
}
```

1.4 String result manipulation

Listing 1.15 : Split string by symbol and get first result

```
string | cut -d'<sybmol>' -f1
```

Listing 1.16 : Replace all symbols with others

```
echo $var | sed 's/ //g'  # remove all spaces
echo $var | sed 's/ /*/g' # replace spaces with *
```

Listing 1.17 : Split string by symbol and get first result

1.5 Grep

Grep is used to filter output using some rules. (often using regular expressions)

Listing 1.18 : Grep example, showing only processes that have managed in name

```
ps -aef | grep managed
```

Grep -i ignores case of your search

Listing 1.19 : Filtering all sockets, that have established in name

```
netstat -an | grep -i "established";
```

You can use Grep in exclude mode:

Listing 1.20 : Filtering all sockets, except one's that are "idle"

```
netstat -an | grep -v "Idle";
```

Grep few possible variants - one or other

Listing 1.21 : Show only CloseWait or Established sockets - using grep OR

```
netstat -an | egrep "CLOSE_WAIT|ESTABLISHED";
```

Let's add more pipes - sort result of grep using 3rd column.

Listing 1.22 : Sort table by column

```
#in this case we list all process that contain 'sh'
#and sort them by 3rd field - PID
ps -aef | grep sh | sort -nk3
```

Listing 1.23 : Get only first argument of found result

```
ps -aef | grep mmog | awk '{print $2;}'
```

Listing 1.24 : Next step - transpose this result into row

```
ps -aef | grep mmog | awk '{print $2;} | xargs'
```

Listing 1.25 : Kill processes with PID's found before

```
kill $(ps -aef | grep mmog | grep -v grep | awk '{print $2;}' | xargs)
```

1.6 Arrays

Listing 1.26 : Add element to Array

```
declare -a array=()
function addToArray { array=("${array[@]}" $1);}
addToArray Something
```

Listing 1.27 : Get all elements from Array

```
${processes_pids[*]}
```

Listing 1.28 : Itterate over array

```
declare -a arr=("element1" "element2" "element3")

for i in "${arr[@]}"
do
   echo "$i"
done
```

1.7 Debug and investigation

Listing 1.29 : Find program location

```
whereis <command or executable>
```

Listing 1.30 : Trace application execution

```
strace -o out.txt ./visualvm
```

Listing 1.31 : Get process arguments

```
pargs <pid>
```

Listing 1.32 : Get process tree

```
ptree <pid>
```

Listing 1.33 : Get ports assigned to current PID

```
/usr/proc/bin/pfiles $PID | grep port
#or if you know name of process and it's only one proces with this name:
/usr/proc/bin/pfiles $(ps -aef | grep jstatd | grep -v grep | awk '{print $2;}')
```

Listing 1.34 : Fast memory dump to screen

```
jmap -histo <pid> | head -100
```

Listing 1.35 : Memory dump to file

```
jmap -dump:file=/tmp/mon/dump.hprof <pid>
```

Listing 1.36 : Add some flag to already runnig JVM

```
jinfo -flag +ExtendedDTraceProbes <Java-process-ID> )
```

1.8 JVM

Listing 1.37 : Already mentioned above - get PID of managed server

```
ps -aef | grep managed | grep -v grep | awk '{print $2;}'
```

Listing 1.38 : Thread Dump

```
jstack <pid>
```

1.9 Conditions and loops

Listing 1.39 : if / not if condition one-liner example

```
if [ actual -eq expected ]; then echo "equals"; fi;
if [ actual -ne expected ]; then echo "not equals"; fi;
```

Listing 1.40 : if / else condition example

```
if [ "foo" = "foo" ]; then
        echo expression evaluated as true
else
        echo expression evaluated as false
fi
```

Listing 1.41 : while loop example

```
while true #or some condition
do
#... do something
done
```

1.10 Method/file arguments

You can define your own methods in bash. To do this - use function command.

Listing 1.42 : function example

```
function getAll()
{
#... Something going on here
}
```

What is different from other development languages - is a way to pass and use arguments. This approach is same for functions/files:

Listing 1.43 : Some hints on arguments

```
$# - count of arguments
$1,$2,$3 - reference to any of argument
$@ - all arguments at once
```

Let's see some simple example on how-to define and call function with arguments:

Listing 1.44 : Function with arguments example

```
function PrintAllArguments(){
for var in "$@"
do
    echo "$var"
done
echo "Number of arguments="$#
}

#Notice the way, that you need to call this function:
PrintAllArguments First second third "string argument"
```

1.11 Call bash scripts from other files

Recently I had task - we had 10 SQL scrips, each of them contained date inside. The goal was to run only one file which accept this date argument and pushes it to other scripts. Here is how i did it:

Listing 1.45 : Passing argument between files

```
#main execution script, which accepts date
if [ $# -ne "1" ];
then
echo "Wrong number of args. Use: ./runall.sh "'2014/01/20 23:30'" "; exit -1;
fi

#argument is assigned to variable
after=$@

#Script is executed with this variable
./SQL/SqlScript1.sh $after > ./results/GPLoad.txt; echo "GpLoad done";
```

1.12 more examples

Listing 1.46 : Count response distribution for mmog file

```
#this is really short-goal script, created without any optimizations in mind.
```

```
#input should be something like:
# /tmp/mon/./bashCount.sh $(tail -1000 ip-http_access_log | cut -d' ' -f4 | xargs)
#or  /tmp/mon/./bashCount.sh $(cat ip-http_access_log | cut -d' ' -f4 | xargs)

range=(0 0 0 0 0 0 0 0 0)

for X in "$@"
do
        if ((X<100000))
        then
                range[1]=$[range[1]+1]
        elif ((100000<=X && X<500000))
        then
                range[2]=$[range[2]+1]
        elif ((500000<=X && X<1000000))
        then
                range[3]=$[range[3]+1]
        elif ((1000000<=X && X<5000000))
        then
                range[4]=$[range[4]+1]
        elif ((5000000<=X && X<10000000))
        then
                range[5]=$[range[5]+1]
        elif ((10000000<=X && X<50000000))
        then
                range[6]=$[range[6]+1]
        elif ((50000000<=X && X<100000000))
        then
                range[7]=$[range[7]+1]
        elif ((X>100000000))
        then
                range[8]=$[range[8]+1]
        fi
done

echo "Number of arguments="$#

echo "          X < 100000="     ${range[1]}
echo "100000   <=X < 500000="     ${range[2]}
echo "500000   <=X < 1000000="    ${range[3]}
echo "1000000  <=X < 5000000="    ${range[4]}
echo "5000000  <=X < 10000000="   ${range[5]}
echo "10000000<=X < 50000000="   ${range[6]}
echo "50000000<=X < 100000000=" ${range[7]}
echo "          X > 100000000=" ${range[8]}
```

Listing 1.47 : Bash export system variable

```
JAVA_HOME=/usr/j2se
export JAVA_HOME
```

Listing 1.48 : Gather CPU information only for particular processes

```

declare -a processes=(AdminServer LINQUS_MANAGED1_PID LINQUS_MANAGED3_PID)
declare -a processes_pids=()

function join { local IFS="$1"; shift; echo "$*"; }
function joinpid { processes_pids=("${processes_pids[@]}" $1);}
function getpid
{
        procid=$(/usr/ucb/ps -auxww | grep $1 | grep -v grep);
        procid=$(echo ${procid:0:25} | awk '{print $2;}');
        echo $1 $procid;
        joinpid $procid;
}

echo "getting PID's"
for i in "${processes[@]}"
do
        getpid "$i"
done

```

```
p9=$(join , $(ps -aef | grep mmog | grep -v grep | awk '{print $2;}' | xargs))
joinpid $p9

prstat -p $(join , ${processes_pids[*]}) -c 10 | nawk '$1=="PID" { system("date +%H:%M:%S") } 1'
```

1.13 SQL code execution from bash

Listing 1.49 : Sql code, called from bash

```
outtemp=`mktemp`

USER=<user>
PASS=<password>
export ORACLE_SID=<sid>
sqlplus -s ${USER}/${PASS}@${ORACLE_SID} > /dev/null << EOF_SQL
set serverout off term off echo off newpage 0 space 0 pagesize 0 feed off head off trimspool on showmode off

spool ${outtemp}
select count (*) from TSM_OPERATIONS where C_STATUS like 'ON_GOING';
spool off
EOF_SQL

d=$(date '+%d-%m-%Y %H:%M:%S')
h=$(cat $outtemp | tr -d ' ' | tr -d "\n\r")

echo "$d $h"
```

1.14 ETC

Listing 1.50 : Set time

```
#++Linux:
date +%T -s "10:13:13"

#Solaris (under root):
date 1013.13
```