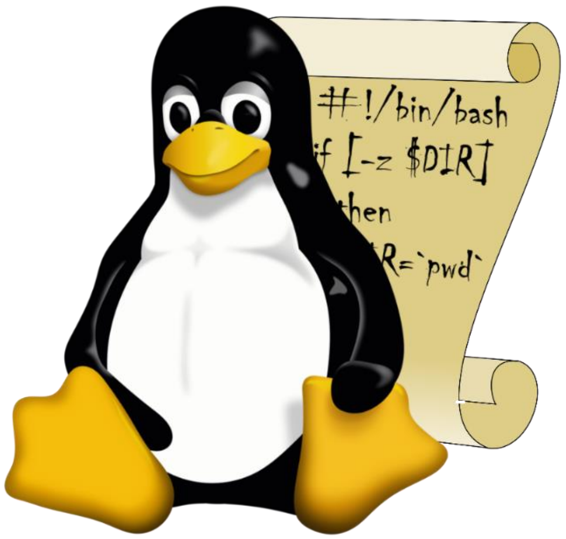

Linux Administrator BASH Scripting

Course manual

I invite you to visit the web page

http://www.kursyonline24.eu/

Review 2021-10-30

Table of Contents

Blank page inserted intentionally

Into – About the Course

When working with Linux in the professional way, we need to know plenty of commands, their options, syntax. That’s one of the biggest troubles for the beginners.

But command line has also huge advantage. When we connect together those simple commands, we can build our own tools, scripts, functions that may replace hundreds of other programs. Having such a new script, we just run it, and multiple actions will be executed at once.

What can be done with scripts? In short – if we can do something with commands, then we can do it also with script, and because on Linux everything can be done with commands, we can do everything with scripts. Some examples could be automated copying of files, installing software, configuring network and much much more.

But knowing the commands is not enough, to build the scripts. Apart from that, we need to know how to operate on variables, build conditional statements, loops, functions and more. And that’s the topic of this course. The course not only teaches how to do something, but often shows how this could be done in different way, what are the advantages and disadvantages of different methods. Step by step, we will show how to build functions and how to create a library of the scripts.

The course contains a set of short lessons, each of them focusing on another topic. In the course manual in the PDF format, for each topic, there is a short note with the most important information from the lesson and a set of exercises allowing to practice the knowledge. There are also the solution proposals, so one could say this course is an equivalent for a traditional classroom training, with the difference, that we don’t have lunch included.

After the course, you will be able to start working on Linux automation and there is no doubt, that automatic management of IT is the future. The cloud expansion caused that one admin is no more managing a couple of machines. One admin has under control hundreds or thousands of machines. That’s something what you can do as well!

Write once – execute multiple times.

Happy learning!

Rafał

How to learn?

That’s great that you are going to learn BASH scripting!

Learn by yourself under your conditions. That’s unfortunately your responsibility to organize time and resources to be able to successfully complete the course, but there are some hints allowing to make your chances bigger.

1. **Not too much at once** – one, maby two or three lessons a day should be enough.
2. **Work regularly** – it is not necessary to work every day, but having a plan to learn on Tuesday, Thursday and Saturday is already something more regular!
3. **Practice**. Only watching the movies is not enough. Solve the exercises.
4. **Modify exercises by yourself**. The exercises are only proposals for you. If you don’t understand something, you see some interesting opportunities – do it. The more you practice, the better for you
5. Don’t complain if according to you the learning goes too slowly. That’s normal that adults learn slower. What is more important - **from time to time do repetition** You have time for it.
6. **Come back to exercises** and solve them again. There is nothing bad about it.
7. If a topic is too difficult – **come back to the lesson**, watch it again. The second, third… time, you will understand more and more.
8. The notes in the book are for your convenience, but.. convenience is not far from laziness. Don’t be lazy. **Have your own notes** in a notebook, write down what you do and what you learn
9. If possible – print out this book, add your own notes, remarks.
10. When you reach a milestone, or when you complete the course, **give yourself a reward**. That will build your motivation!
11. **Use also other sources of knowledge**: books, blogs, forum etc. Some of them may be initially too complicated, but with time you will understand them.
12. When the course is completed – **update your CV on LinkedIn** (or other professional portals that you use), show your achievement. Show your new skills. Tell your friends about your efforts

    Good Luck!

    Raphael

Scripts? Who needs them today?

Note

- Scripts are integral part of Linux, they are used for:
  - Environment configuration
  - Software installation
  - Automation
  - Building simple scripts started in console
- Some of the operations and instructions used in scripts are: **if**, **while**, **for**, **case**, functions.
- Commented lines start with **#**
- Variable names are most often written in capitalized form
- Command **file** allows to check the file type: binary, text, script
- The first line in a script is often a shebang. It allows to define what a program should be used to interpret the script
- SHEBANG for bash files is:

```
#!/bin/bash
```

Lab

1. Review more information about shebang: https://en.wikipedia.org/wiki/Shebang_(Unix) Note the history part in last paragraphs
2. Determine what kind of files are (if they are available in your distribution of Linux):
   a. /sbin/ifcfg
   b. /usr/sbin/dhclient-script
   c. /usr/sbin/ifstat
3. Review the file /sbin/ifcfg and note some (hopefully) known to you instructions. Don't spend too much time on it.

Solution proposal (only the commands)

```
file /sbin/ifcfg
file /usr/sbin/dhclient-script
file /usr/sbin/ifstat
```

First script. Sometimes it is better to start from beginning

Note

- Files containing scripts:
    - Have an extension .sh but it is not mandatory
    - In the first line usually have shebang defining which interpreter should run the script. Shebang used by BASH is:

```
#!/bin/bash
```

- The BASH (and scripts) is sensitive to case (lower-/upper-case) and additional spaces
- One of the best practices is to end script with exit code:

```
exit 0
```

    - Exit code 0 means no error
    - Other exit code points to error. Using many non-zero exit codes allows to associate an error condition with the adequate lines of the script
- Exit code can be checked with a command:

```
echo $?
```

- To start a script:
    - Its name should be pointed with absolute or relative path, or the file should be saved in a directory being included into the variable **PATH**
    - The script needs to have execute permission:

```
chmod u+x script.sh
```

Lab

1. Develop script which:
    a. Assigns to variable SENTENCE the value ”If cats could talk, they wouldn’t.”
    b. Displays the text

   Try to apply all known best practices related to script development.
2. Run the script

Solution proposal (only the commands)

```
#!/bin/bash


SENTENCE="If cats could talk, they wouldn’t."
echo $SENTENCE

exit 0
```

Variables in Scripts

Note

- Variables allow to store data in memory. Once defined, can be multiple times reused
- To display all the variables a command **set** can be used
- Following instruction creates and next displays the variable's value:

```
IMPORT_DIR=/tmp/import
echo $IMPORT_DIR
```

- When constructing a text, that will contain variables, take following points into consideration:
  - The variables are replaced with values only when the text in enclosed in the quotes
  - The variables are not replaced with values when the text in enclosed in the apostrophes
  - If part of the text should be replaced with a value and the other part should not be replaced, add sign **backslash \** before the **$**-sign where the variable should not be replaced with the value

```
echo ”this is $IMPORT_DIR”
echo ‘this is $IMPORT_DIR’
echo “the value of \$IMPORT_DIR is $IMPORT_DIR”
```

- To write information int the system log (/var/log/messages lub /var/log/syslog) command **logger** can be used:

```
logger ”Starting installation”
```

Lab

1. Your task is to write a script, which will move files generated by camera-monitoring system from one directory to another archive directory
2. That's are the requirements:
   a. The source catalog is /tmp/ftp_data (In real-world it would be most probably a different directory, but… that's just an excercise)
   b. The destination catalog is /tmp/monitor_arch
   c. Before moving the files, a massage should be sent to system log. The same should be done when the script finishes its work
   d. The script should inform the user about the progress
   e. The assumption is, that the directories already exist
   f. Try to use variables whenever possible
3. Test the script by creation of a few files in the source directory and run the script. Validate if the files are indeed moved.

Solution proposal

```
# prepare files and directories:

mkdir /tmp/ftp_data
mkdir /tmp/monitor_arch

touch /tmp/ftp_data/1.txt
touch /tmp/ftp_data/2.txt



# the content of the script:

cat > archive.sh
#!/bin/bash

INPUT_DIR=/tmp/ftp_data
OUTPUT_DIR=/tmp/monitor_arch

logger "Starting moving files from $INPUT_DIR to $OUTPUT_DIR"
echo "Starting moving files from $INPUT_DIR to $OUTPUT_DIR"

mv $INPUT_DIR/* $OUTPUT_DIR

echo "done"
logger "Moving files from $INPUT_DIR to $OUTPUT_DIR finished!"

exit 0



# preparing and starting the script
chmod u+x archive.sh
./archive.sh



# validating the results
sudo tail -f /var/log/syslog
ls -R /tmp
```

Tricks from .bash_profile and .bashrc

Note

- The user environment is configured with following files:
    - /etc/profile – started for every user after log-in. System file
    - .bash_profile – started for user after log-in, can be modified by a user
    - .bashrc – started for a user, each time when a new bash shell is started. Can be configured by a user.
- By default, variables defined in bash are not visible for child processes. To pass the variable to child processes, such a variable needs to be exported:

```
export INSTALL_DIR=/usr/finbook
```

- Script may be divided into two parts: static with variable definition in one file and dynamic script code in the other file. The static file can be read into the script with the command **source** or a **dot** (**.**)

```
source config.sh
```

Lab

1. You are a real "console fan", but you like to see the calendar on your console's screen like the windows guys. Add to your profile a command displaying the calendar for the current month (use command cal). The calendar should be displayed once after login.
2. You create an application, that once every few hours should download content of the file:
   http://feeds.bbci.co.uk/news/world/rss.xml
   and save it in the file news.xml. First, validate if this can be done by using following command:

```
wget --quiet --output-document=news.xml  http://feeds.bbci.co.uk/news/world/rss.xml
```

3. Develop a script, that will download the web page to the file, but:
    a. Web site address should be saved in a separate file as a parameter WWW_ADDRESS
    b. File name should also be saved in this file as parameter named FILE_NAME
4. Test the script

Solution proposal

```
cat > config.sh
WWW_ADDRESS=http://feeds.bbci.co.uk/news/world/rss.xml
FILE_NAME=news.xml
<CTRL+D>
```

```
cat > download_news.sh
#!/bin/bash

source ./config.sh

echo "Downloading $WWW_ADDRESS to $FILE_NAME..."
wget --quiet --output-document=$FILE_NAME $WWW_ADDRESS
echo "Done!"

exit 0
<CTRL+D>
```

```
chmod u+x download_news.sh
```

```
./download_news.sh
```

How to count in BASH?

Note

- The most popular way of counting in bash uses syntax of $ and two brackets::

```
x=8
y=4
echo $((x+y))
```

- This syntax allows to perform more operations in one step. Additionally, the results can be saved to another variable. The value returned by a following expression is the result of the last operation:

```
echo $((z=x+y, u=x-y, v=u/z))
```

- It is possible also to perform some basic comparison. (Note: When the result is 1 – the condition was True, and when the result is 0 – the condition was False)

```
echo $((x==y))
echo $((x>y))
```

- An older method of counting is the command **expr**. Here it is mandatory to use the $ before the variables and to separate the variables and operations with space. The multiplication operator ***** must be preceded with a **backslash \**

```
expr $x + $y
expr $x \* $y
```

- To save the result in a variable the notation with **$(…)** or reverse apostrophe **\`…\`** should be used:

```
z=$(expr $x + $y)
u=`expr $x \* $y`
```

- The command bc is a kind of an interactive calculator. It can be also used to execute some more complicated mathematical operations in shell. To use it in batch mode, send on standard input the text that should be interpreted as an expression:

```
echo ”$x + $y” | bc
```

Lab

1. On disk there is a file of size 356 GB. You are going to compress it and save the compressed file on a file system with 220 GB of free space. An expected compression ratio is 60% (the target file size will be of 60% of original size). Is it possible to perform planned actions?
2. In the point (1) we assumed, that the compression ratio is 60%. This value is approximate and we want to have an additional margin. We assume, that the operation should succeed if the destination file system has free disk space of 60% of original file plus 10 GB. Is it possible to perform the planned actions now?
   When counting use variables and BASH

Solution proposal

```
FILESIZE=356
FREESPACE=220
COMPRESSPERCENT=60

NEWSIZE=$((FILESIZE*COMPRESSPERCENT/100))

echo $NEWSIZE
213

echo $((FREESPACE>NEWSIZE))
1




FILESIZE=356
FREESPACE=220
COMPRESSPERCENT=60
MARGIN=10

NEWSIZE=$(((FILESIZE*COMPRESSPERCENT/100)+MARGIN))

echo $NEWSIZE
223

echo $((FREESPACE>NEWSIZE))
0
```

Working with Text

Note

- „Text cutting” can be done with a use of a mask defining what should be removed from a text:
    - ${VARNAME%/*} - remove the shortest text matching mask /* starting from the end
    - ${VARNAME%%/*} - remove the longest text matching the mask /* starting from the end
    - ${VARNAME#*/} - remove the shortest text matching the mask */ starting from beginning
    - ${VARNAME##*/} - remove the longest text matching the mask */ starting from the beginning
- Text can be cut pointing to the number of characters that should be skipped and copied from the original text:
    - ${VARNAME:x:y} - skip x letters and next copy y letters. The values can be in special case equal zero

Lab

1. Using command touch create a new file:

```
touch  /tmp/data.csv
```

2. Save the file name in a variable named FILEPATH.
3. Cut from the FILEPATH only the directory name. The result save in DIRONLY
   Note: There is a special command **dirname** that can do the same, but let’s suppose you don’t know it 😉
4. Cut from FILEPATH only the file name and save in the variable FILEONLY
   Note: There is a special command **basename**, that can do the same, but let’s suppose you don’t remember it 😉
5. Cut from FILEONLY only the file name, without extension and save it in the variable FILENOEXT
6. Create a new variable named NEWPATH, that will be created from:
    a. Directory name taken from DIRONLY
    b. Character /
    c. File name taken from FILENOEXT
    d. And extension .old
7. Using variables, copy the file pointed by FILEPATH to NEWPATH

Solution proposal

```
touch /tmp/data.csv


FILEPATH=/tmp/data.csv


DIRONLY=${FILEPATH%/*}
echo $DIRONLY
/tmp


FILEONLY=${FILEPATH##*/}
echo $FILEONLY
data.csv


FILENOEXT=${FILEONLY%.*}
echo $FILENOEXT
data


NEWPATH=$DIRONLY/$FILENOEXT.old
echo $NEWPATH
/tmp/data.old


cp $FILEPATH $NEWPATH
```

Input, Output & Error Output

Note

- Every program in Unix has by default 3 devices available:
    - 0 – standard input – by default keyboard
    - 1 – standard output – by default screen
    - 2 – standard error output – by default screen
- Those devices can be redirected

```
# redirect output to a file
./script  >  ./output.txt
./script 1>  ./output.txt

# redirect error output to a file
./script 2>errors.txt

# redirect error output to a file by appending the content
./script 2>>errors.txt


# redirect output to a file and standard error output to the same
# device as standard output
./script > /output.txt 2>&1

# redirect errors to /dev/null
./script 2>/dev/null

# redirect input to take it from following lines of the script
./script << EOF
1
2
3
EOF
```

Lab

1. You want to save information about currently logged in users. Write script that:
    a. Displays current date and time using command **date**
    b. Displays computer name using command **hostname**
    c. Displays list of logged in users using command **who**
2. After testing the script redirect its output to a file report.txt
3. After a few seconds execute the script once again adding the new results to the existing file report.txt
4. Using command

```
du -sh  /var/*
```

check size of the subdirectories of the directory /var. Because the command was stared without sudo, you can expect some error messages. Redirect them to /dev/null

5. Can we trust such results? Is skipping the errors a good practice? Run the command once again but this time use **sudo**. Compare the results with results of the previous test

Propozycja rozwiązania

```
cat > report.sh
#!/bin/bash

date
hostname
who

exit 0


chmod u+x report.sh

./report.sh > report.txt
./report.sh >> report.txt



du -sh /var/*
4.8M    /var/backups
du: cannot read directory '/var/cache/apt/archives/partial': Permission denied
du: cannot read directory '/var/cache/lightdm': Permission denied
du: cannot read directory '/var/cache/private': Permission denied
du: cannot read directory '/var/cache/cups': Permission denied
…
du: cannot read directory '/var/tmp/systemd-private-971ebb5aeaef4f37906320dff3bdd1bf-
rtkit-daemon.service-sIR8dY': Permission denied
du: cannot read directory '/var/tmp/systemd-private-971ebb5aeaef4f37906320dff3bdd1bf-
systemd-timesyncd.service-y6HEWb': Permission denied
12K     /var/tmp

du -sh /var/* 2>/dev/null
4.8M    /var/backups
737M    /var/cache
146M    /var/lib
4.0K    /var/local
0       /var/lock
798M    /var/log
4.0K    /var/mail
4.0K    /var/opt
0       /var/run
36K     /var/spool
100M    /var/swap
12K     /var/tmp

sudo du -sh /var/* 2>/dev/null
4.8M    /var/backups
737M    /var/cache
146M    /var/lib
4.0K    /var/local
0       /var/lock
798M    /var/log
4.0K    /var/mail
4.0K    /var/opt
0       /var/run
48K     /var/spool
100M    /var/swap
20K     /var/tmp
```

Saving Command Output in Variable

Note

- Variables in the programs and in the scripts hold information that can be reused in the script.
- A huge advantage of using variables is that they give the numerical or text values a logical meaning defined by the value's name. This makes developer life easier. Using value 1000 is more mysterious than using variable MAX_SIZE.
- The variables can also hold values returned by other commands. To save the command result in a variable use the syntax **$(…)**

```
DISK_FREE=$(df -h /home | tr -s ' ' | grep -v 'Filesystem' | cut -d ' ' -f 4)
echo $DISK_FREE
16G
```

- Older scripts can also use a reverse apostrophe `` `…` ``:

```
DISK_FREE=`df -h /home | tr -s ' ' | grep -v 'Filesystem' | cut -d ' ' -f 4`
echo $DISK_FREE
16G
```

Lab

1. Due to some dependencies in your script you want to use variable to check the size of subdirectories of the directory /var:
   a. Go to directory /var
   b. In the variable DIRS save the result of the command **ls**
   c. Using $DIRS and the command **du -sh** display size of all subdirectories in /var
   d. Running the command use **sudo** or redirect the standard error output to /dev/null
2. You want to have a script counting number of sessions currently established for your user:
   a. To variable ME save result of command **whoami**
   b. To variable SESSION_COUNT save number of sessions open for user $ME. Hints:
      i. Run command **who**, and results pass to …
      ii. Command **grep**, that will filter out only rows containing username $ME and will send them in pipeline to …
      iii. To the command **wc -l**, that will count the number of lines
      iv. Display message communicating the number of open sessions for the current user

Solution proposal

```
cd /var

DIRS=$(ls /var)

sudo du -sh $DIRS




ME=$(whoami)

echo $ME
pi

who | grep $ME
pi       tty1         2021-07-28 22:32
pi       pts/0        2021-07-31 14:56 (192.168.2.114)

who | grep $ME | wc -l
2

SESSION_COUNT=$(who | grep $ME | wc -l)
echo "$ME has $SESSION_COUNT session(s)"
pi has 2 session(s)
```

Checking Logical Conditions

Note

- Evaluation of logical expressions can be done in three ways (Note the spaces!):
    - **test** – the oldest and available everywhere
    - **[]** – the ancestor of test, nowadays implemented everywhere
    - **[[ ]]** – more modern and easier to use (more intuitive string operations and logical expressions && ||). If compatibility is not a problem, that's the recommended method

```
test -f /etc/passwd
[ -f /etc/passwd ]
[[ -f /etc/passwd ]]
```

- Test can perform multiple conditional comparisons (full list in **man test**):
    - -f check if the file exists and is a regular file
    - -d check if directory exists
    - $x -gt $y x is greater than y
    - $x -le $y x less or equal than y
    - -z "$x" the string x is empty
    - -n "$x" the string x is non-empty
    - $x = $y the string x is equal to string y (in case of [[ ]] a symbol == should be used)
    - -a AND – conjunction – in case of [[ ]] symbol && should be used
    - -o OR – alternative – in case of [[ ]] symbol || should be used
- The result of the test is available in variable $? The value of 0 designates TRUE and the value of 1 – FALSE. The instruction checking the result of the test must be the next instruction after the test.

Lab

1. Define variables LIMIT=16, AVERAGE=10, PEAK=20 i USED=15 and write tests of following conditions:
    a. Is USED lower than LIMIT
    b. Is USED greater or equal than AVERAGE
    c. Is LIMIT greater than PEAK
    d. Is USED lower than LIMIT and greater than AVERAGE
2. Define variable MEDIA=/media/cdrom and TMPLOC=/tmp and check:
    a. Does the directory MEDIA exist?
    b. Does the file TMPLOC exist?
    c. Does the directory TMPLOC exist?
3. Without defining the variable NAME, check if the text saved in variable NAME is empty or non-empty.
4. Define the variable NAME and repeat the tests from the previous point.

Solution proposal

```
LIMIT=16
AVERAGE=10
PEAK=20
USED=15

test $USED -lt $LIMIT
echo $?
0

[ $USED -ge $AVERAGE ]
echo $?
0

[[ $LIMIT -gt $PEAK ]]
echo $?
1

[[ $USED -lt $LIMIT && $USED -gt $AVERAGE ]]
echo $?
0



MEDIA=/media/cdrom
TMPLOC=/tmp

test -d $MEDIA
echo $?
1

test -f $TMPLOC
echo $?
1

test -d $TMPLOC
echo $?
0



[[ -z $NAME ]]
echo $?
0

[[ -n $NAME ]]
echo $?
1

NAME=Max

[[ -z $NAME ]]
echo $?
1

[[ -n $NAME ]]
echo $?
0
```

Conditional Execution - if

Note

- **If** allows to conditionally execute a part of the script
  - The command starts with **if**, after which there is a condition to be checked. After the word **then** there are instructions that will be executed if the condition was true
  - Optionally, there may be the instruction **elif**, which like **if** checks the condition and allows to execute following commands, when the condition was true
  - Optionally at the end there may be the instruction **else**. The commands after it will be executed when none of the previous conditions was met
  - The statement must be finished with the word **fi**

```
if [[ -d $DIR ]]
then
   echo "Directory $DIR already exists. Nothing to do."
elif [[ -e $DIR ]]
then
   echo "Path $DIR already exists, but it is not a directory!"
else
   echo "Creating directory $DIR"
   mkdir $DIR
fi
```

Lab

1. Prepare to lab running following commands:

```
touch /tmp/output.txt
sudo groupadd DEV
sudo useradd mat -g DEV
```

2. Create script named permissions.sh, and put in it commands that:
   a. Will create variables:

```
USER=mat
GROUP=DEV
FILE=/tmp/output.txt
```

   b. Check if USER or GROUP are empty texts. If yes, display a message and exit the script with code 11
   c. Check if FILE points to an existing file
      i. If yes, display a message informing about changing ownership of the **file** and change the owner of FILE to user USER and group GROUP
      ii. In other case, if that's a catalog, display message about changing ownership of the **directory** and change ownership of FILE **recursively** to USER and GROUP
      iii. In other case, display a message informing that FILE is incorrect and finish the script with exit code 12
3. Test the script. Remember to use sudo. Test different values for variables, so that all possible scenarios will be checked and different messages/exit codes will be returned

Solution proposal

```
#!/bin/bash

USER=mat
GROUP=DEV
FILE=/tmp/output.txt


if [[ -z $USER || -z $GROUP ]]; then
   echo "USER and GROUP cannot be empty"
   exit 11
fi


if [[ -f $FILE ]]; then
   echo "Changing permissions to file"
   chown $USER:$GROUP $FILE
elif [[ -d $FILE ]]; then
   echo "Changing permissions to directory"
   chown -R $USER:$GROUP $FILE
else
   echo "The FILE is incorrect"
   exit 12
fi

exit 0
```

How to Work with Exit Codes?

Note

- Script may define a list of variables with possible exit codes. When the exit code should be returned, instead of returning a “magic” number, the variable with descriptive name can be used.
- Each command started within the script also returns an exit code. Following the unix standard the returned code can be:
    - Zero in case of successful operation
    - Non-zero in case of error
- If the exit code of an external command should be saved in a variable, after the command execution check the value from $?

```
mkdir $CAT
RESULT=$?
if [[ $RESULT -ne 0 ]]; then
   exit 1
fi
```

- This entry could be shortened:

```
mkdir $CAT
if [[ $? -ne 0 ]]; then
   exit 1
fi
```

- The external program could be also started directly in the if condition:

```
if mkdir $CAT; then
   exit 1
fi
```

Lab

1. To the script from the previous lesson add lines checking for errors that could be met when::
    a. The file owner is changed
    b. The directory owner is changed
2. Test the script in different scenarios

Solution proposal

```
#!/bin/bash

USER=mat
GROUP=DEV
FILE=/tmp/output.txt


if [[ -z $USER || -z $GROUP ]]; then
   echo "USER and GROUP cannot be empty"
   exit 11
fi


if [[ -f $FILE ]]; then
   echo "Changing permissions to file"
   chown $USER:$GROUP $FILE
   if [[ $? -ne 0 ]]; then
      exit 13
   fi
elif [[ -d $FILE ]]; then
   echo "Changing permissions to directory"
   chown -R $USER:$GROUP $FILE
   if [[ $? -ne 0 ]]; then
      exit 13
   fi
else
   echo "The FILE is incorrect"
   exit 12
fi

exit 0
```

Shortened Conditional Statement - && ||

Note

- When BASH interprets logical conditions, it follows a few rules:
  - The expression is analyzed from left to the right
  - If it is possible to return the final result of the tested condition, without evaluation of the following subexpressions, the work ends and the result is returned
- Each command returns exit code, that can be interpreted as a logical value
- Following the mentioned here rules, the if statement can be often shortened to one line:
  - If user yoda exists, don't run echo
  - But if the user yoda doesn't exist, run echo

```
grep yoda /etc/passwd || echo ”No such user”
```

  - If the user yoda exists run the echo
  - But if the user yoda doesn't exist, don't run echo

```
grep yoda /etc/passwd && echo ”No such user”
```

- In such a way also more conditions can be evaluated
- This syntax allows in a compact way to write simple if expressions

Lab

1. Create a command that will create a directory /tmp/workdir in case when the directory doesn't exist yet. Do it using 2 methods and in both use the short syntax:
   a. Once use an operator &&
   b. And once use an operator ||
2. Write a command, that in case of existence of the file /tmp/stop.txt will display message “The script execution cannot be continued” and will end with exit code 1

Solution proposal

```
[ ! -d /tmp/workdir ] && mkdir /tmp/workdir
[ -d /tmp/workdir ] ||  mkdir /tmp/workdir



[ -f /tmp/stop.txt ] && echo "Cannot continue" && exit 1
```

Loop for – 3, 2, 1 start

Note

- **for** allows to execute the same code multiple times:

```
for i in {1..5}; do
   echo $i
done
```

- **for** loops may be nested:

```
for i in {1..5}; do
   for j in {1..5}; do
      echo "$i * $j = $((i * j))"
   done
done
```

- **for** may be used to work with each line of a text file separately:

```
for server in $(cat ./servers.txt); do
   echo $server
done
```

- **for** may process all the files from a directory:

```
for file in files/*.txt; do
    echo $file
done
```

- **for** may also iterate over words in text variable

```
DAYS="Mon Tue Wed Thu Fri Sat Sun"

for day in $DAYS; do
  echo $day
done
```

Lab

1. Create a loop displaying all the file names from the directory /var/log. (I know it can be done with a simple **ls**, but here let's use **for**, it will be the first step in the next lab.
2. Create a loop processing each line of the file /etc/fstab (each line corresponds to one mounted file system). The lines in the file contain spaces, what causes, that **for** treats separately each part separated by space. To solve the problem, add in the script somewhere before **for** following line:

```
IFS=''
```

   It will cause that space loses its special meaning and will be treated as other characters
3. Create a loop, that for the numbers starting from 1 and ending with 20 will display a square of that number (That's a less-sysadmin example, but… sometimes admins also need to count something)

Solution Proposal

```
#!/bin/bash

DIR=/var/log

for m in $DIR/*; do
   echo  $m
done

exit 0
```

```
#!/bin/bash

FILE=/etc/fstab
IFS=''

for line in $(cat $FILE); do
   echo "$line"
done

exit 0
```

```
#!/bin/bash

for i in {1..10}; do
   echo "$i   $(( $i * $i ))"
done

exit 0
```

Loop for – Use Case

Note

- **for** is perfect for some scenarios, where a command or set of commands need to be executed multiple times like:
    - for every file in a directory
    - for every line in a file
    - for every word in a text
- Similar activities could be developed also with different types of the loop, but for-loop could be considered as the best choice whenever the number of executions is good defined already at the beginning of execution

Lab

1. Create a file folders.txt. In this file write down in separate lines names of new folders that should be created. In the next steps we will use content of the file to create new subdirectories in the /tmp. Use names that indeed can be the names of folders (avoid special characters etc).
2. Write a loop **for**, which will process all the lines from this file and for each of them will create the subdirectory in /tmp.
3. Create a file folders-du.txt and in lines of the file write down the following names:

    a. /etc
    b. /usr/lib
    c. /var/mail
    d. /tmp
    e. …
4. Create a loop **for**, that for every line from this file (so for every mentioned here directory), will start **du** command. Make sure, that potential errors emitted by the commands will stay hidden.

```
du -sh
```

Solution proposal

```
#!/bin/bash

FILE=folders.txt
DIR=/tmp

for folder in $(cat $FILE); do

    CATALOG=$DIR/$folder
    echo "Creating directory $CATALOG"
    mkdir $CATALOG

done

exit 0


#!/bin/bash

FILEDU=./folders-du.txt

for folder in $(cat $FILEDU); do

   du -sh $folder 2> /dev/null

done
```

Loop while - Introduction

Note

- **while** is a loop that will be executed so long, while the loop-condition is true
- **while** loop is usually used, when the number of executions is not known at the beginning. However, depending on the condition, it may also behave as a **for** loop
- **while** loop can be also executed a given number of times:

```
i=0
while [[ $i -lt 10]]; do
       echo ”$((++i))”
done
```

- **while** loop can be also executed for each line of the input file:

```
while read line; do
       echo line
done < my_file.txt
```

Lab

1. You want to stop the script for 10 seconds, however, during that pause, you wish also to display a dot every second. Hints:
   a. **sleep 1** halts the script execution for 1 second
   b. **echo -n ‘.’** Displays a dot without producing a new line

   Create such a script

Solution proposal

```
#!/bin/bash

MAXTIME=10
TIME=1

while [[ $TIME -le $MAXTIME ]]; do

  echo -n '.'
  sleep 1
  TIME=$(( $TIME + 1 ))

done

echo ''

exit 0
```

Loop while – Use Case

Note

- **while** can serve in a script as a block waiting for an event that should with time happen on the system:
    - Waiting until a file will be created
    - Waiting until the server starts responding to a ping
    - Waiting until the user would log off
- To correctly build such a "waiting" script, the **while** loop needs to have correctly defined condition controlling execution of the loop
- Both **while** and **for** can be nested

Lab

1. If there aren't any additional users on your system, create one using command:

```
sudo useradd helen
sudo passwd helen
```

2. Create a script, that will display dots so long, as the user hellen logs on. To check if the user is logged on you can use following command:

```
who | grep helen
```

   The condition of the loop may check if the result of this query is an empty string. Where possible try to use variables

3. At the end, so after the user hellen logs on, display on the terminal an adequate message and generate a bell sound. To generate a bell use following command:

```
echo -e "\\07"
```

   (If the bell is not working for you, the reason can be that the terminal is not interpreting it correctly. I such case, don't spend too much time on it …)

4. Test the script:
    a. Run the script – it should display a dot every second.
    b. In new terminal log on as hellen. Just after it, the first script should be stopped and a sound should be hearable.

Solution proposal

```
#!/bin/bash

USER='helen'

while [[ -z $(who | grep $USER) ]]; do

   echo -n '.'
   sleep 1

done


echo -e "\\07"
echo "$USER is on the system"
echo -e "\\07"
```

Arrays

Note

- Arrays can store data used during the execution of the script (like used space on file systems, , total space on disk, size of disk space consumed by a list of users etc.)
- The syntax for operations working with arrays is very specific:

```
# Define groups:
GROUPS=(office managers support)

# Show positions number zero – NOTE! – Numbering starts from zero!
echo ${GROUPS[0]}

# Display the last item:
echo ${GROUPS[-1]}

# Display all the items from the list:
echo ${GROUPS[@]}

# Display indexes of the array:
echo ${!GROUPS[@]}

# Iterate over the array:
for group in ${GROUPS[@]}: do
  echo $group
done

# Display number of items in the array:
echo ${#GROUPS[@]}

# Change value of the item on position 1
GROUPS[1]='admins'

# Add new item
GROUPS+=('devops')

# Display 4 items from the list starting from the second
echo ${GROUPS[@]:2:4}
```

Lab

1. You need to prepare a report that should display some information for each day of the week. The report will be used by Italians. Create a table and initialize it with Italian week days names: Lunedi Martedi Mercoledi Giovedi Venerdi
2. Ops.. weekend should be also included. Add: Sabato Domenica
3. Number of today's day can be returned with the command date. Save result of the following command in the variable today:

```
Date  +%w
```

4. Display Italian week day using $today and $weekdays
5. Display Italian names of working week.

Solution proposal

```
weekdays=(Lunedi Martedi Mercoledi Giovani Venerdi)

echo ${weekdays[@]}
Lunedi Martedi Mercoledi Giovani Venerdi

weekdays+=(Sabato Domenica)

echo ${weekdays[@]}
Lunedi Martedi Mercoledi Giovani Venerdi Sabato Domenica

date +%w
5

today=$(date +%w)

echo $today
5

echo ${weekdays[$today]}
Sabato

echo ${weekdays[@]:0:5}
Lunedi Martedi Mercoledi Giovani Venerdi
```

Use Case - Array

Note

- Arrays can be used to save temporary, serial data in the script
- The data can be read from the user interactively, but can also be static or be generated by external commands.
- The data can be saved in the tables in the initial part of the script, a next in the following phases be used for processing. This is usually done with a loop statement iterating over the array with temporary data

Lab

1. This example is not using an array, but presents similar steps that should be executed when working with array
2. Create a script, that for each logged in user will count, how many processes this user has started.:
   a. Hint: some useful commands that may help with the task are:
      i. Get a list of currently logged in users:

```
who | cut -d ' ' -f 1 | uniq
```

      ii. Count number of processes for the user helen:

```
ps --no-headers -u helen | wc -l
```

   b. First save the list of currently logged in users to variable USERS. You can use the command (i)
   c. Next, using a loop go through the list USERS and for each user count the number of processes of the user. Use the command (ii)
   d. Display result

Solution proposal

```
#!/bin/bash

USERS=$(who | cut -d ' ' -f 1 | uniq)

for u in $USERS; do

   count=$(ps --no-headers -u $u | wc -l)
   echo "$u - $count"

done

exit 0
```

Script Arguments

Note

- To run script with parameters, append them after the script name in the command line. Separate parameters with spaces.
- In the script, extract the value of parameters:
    - $0 – the name of the script
    - $1. $2. $3,… ${10}, ${11}… - parameters values
    - $# - number of parameters
    - $@ - parameter list (array)
    - $* - parameter list (a text separated with spaces)
- To process the parameters one after another, create a loop::
    - Run the loop so long as $# is greater than 0
    - Read parameter value from variable $1
    - Shift the arguments using command **shift** (shift will change value of $1 and $#)

```
#!/bin/bash

while [[ $# -gt 0 ]]; do

   echo $1
   shift

done
```

Lab

1. In the lesson related to for loop use-case, we developed a script estimating size of the directories saved in a text file. Here an example script:

```
#!/bin/bash

FILE=folders.txt
DIR=/tmp

for folder in $(cat $FILE); do

    CATALOG=$DIR/$folder
    echo "Creating directory $CATALOG"
    mkdir $CATALOG

done

exit 0
```

2. Change the script so, that the directories could be passed by arguments. Script should be able to process any number of arguments (that means you need to use a loop).
3. Test the script. Try to analyze also directories containing spaces in the name
   Hint: during tests don't redirect error output to /dev/null. This will allow to analyze potencial errors.

Solution proposal

```
#!/bin/bash
for folder in "$@"; do
   du -sh "$folder"        # 2> /dev/null
done



sudo mkdir /home/folder\ with\ space
./du.sh /tmp /root ‘/home/folder with space’
```

Script Arguments. Instruction case

Note

- Script accepting parameters may be divided into three logical parts:
  - Declaration of variables matching to parameters, the initial values may be incorrect
    - **while** loop – in the loop the current value of parameter is taken from $1. Every time shift moves the parameters changing $1 and $#. The condition of the loop:

```
while [[ $# -gt 0 ]]; do
```

    - In the loop the arguments are mapped to the variables. The parameters and values are taken from $1
    - The options may be single letters or long options
    - If the argument and its value should be read, than the option name can be taken from $1 and associated value from $2. In such case shift should be executed twice
    - Other/incorrect options may be serviced by condition *

```
case $1 in
   -r|--report)
      echo ”using -r”
      MAKE_REPORT=’Y’
      shift
      ;;
   # put here other options
   *)
      echo “incorrect option $1”
      exit 1
   ;;
esac
```

  - The validity and consistency of options should be performed
  - The essential part of the script taking actions accordingly to variables defined and changed earlier

Lab

1. Create a script displaying information about user accepting following arguments
   a. -n <user_name> - user name of the user that’s information should be shown
   b. -i   will run **id <user_name>**
   c. -l   will run **last <user_name>**
   d. -d will run **du -sh /home/<user_name>**
   e. -h will display short help/syntax
   f. Passing incorrect arguments should cause an error
   g. Before running the command, the user_name should be validated – it cannot be empty
   h. Warning! Some of the commands must be started with sudo, so calling the script use sudo as well
2. Save the script for the next lab.

Solution proposal

```
#!/bin/bash


USERNAME=''
RUN_ID='N'
RUN_LAST='N'
RUN_DU='N'


while [[ $# -gt 0 ]]; do

  case $1 in

    -i|--id)
      echo "id will be started"
      RUN_ID='Y'
      shift
      ;;
    -l|--last)
      echo "last will be started"
      RUN_LAST='Y'
      shift
      ;;
    -d|--du)
      echo "du will be started"
      RUN_DU='Y'
      shift
      ;;
    -h|--help)
      echo "USAGE:  $0 [-i|--id] [-l|--last] [-d|--du] [-n|--name <user_name>]"
      exit 0
      ;;
    -n|--name)
      echo "Information about user $2 will be shown"
      USERNAME=$2
      shift
      shift
      ;;
    *)
      echo "Unknown option! Use $0 -h to display help"
      exit 1
      ;;

  esac

done


[[ -z $USERNAME ]] && echo "User parameter is mandatory - use option -n or display
help using -h" && exit 1

if [[ $RUN_ID == 'Y' ]]; then
  echo "id $USERNAME"
  id $USERNAME
fi

if [[ $RUN_LAST == 'Y' ]]; then
  echo "last $USERNAME"
  last $USERNAME
fi

if [[ $RUN_DU == 'Y' ]]; then
  echo "du -sh /home/$USERNAME"
  du -sh /home/$USERNAME
fi

exit 0
```

Reading Script Parameters with getopts

Note

- getopts is a dedicated instruction allowing to process parameters of the script
- The first parameter for getopts is a definition of acceptable script parameters
    - A colon after a letter means that the option defined with that letter requires additionally a value
    - A colon at the beginning means, that getops should handle errors in automatic way
- When getopts processes parameters, it puts the parameter letter in the variable opt
- If the parameter accepts a value, it is assigned to variable OPTARG

```
while getopts ":rdhf:" opt; do
   case $opt in
      r)
         MAKE_REPORT='Y'
         ;;
      f)
         FILE=$OPTARG
         ;;
    esac
done
```

- While getopts reads the parameters, the variable OPTIND points to the number of the next argument that should be processed. It allows at the end of the process analyzing the parameters to shift all the processed parameters. The final shift should move the parameters by OPTIND-1:

```
shit $((OPTIND – 1))
```

- If the message emitted by a script should be treated as an error, at the end of the echo command the redirection >&2 should be added

```
Echo ‘missing parameter’ >&2
```

Lab

1. Change code from the previous lab, so that it will use getopts
    a. Adapt the script to all limitations introduced by the getopts function.

Propozycja rozwiązania

```
#!/bin/bash


USERNAME=''
RUN_ID='N'
RUN_LAST='N'
RUN_DU='N'

while getopts ":ildhn:" OPT; do

  case $OPT in

    i)
      echo "id will be started"
      RUN_ID='Y'
      ;;
    l)
      echo "last will be started"
      RUN_LAST='Y'
      ;;
    d)
      echo "du will be started"
      RUN_DU='Y'
      ;;
    h)
      echo "USAGE:  $0 [-i] [-l] [-d] [-n <user_name>]"
      exit 0
      ;;
    n)
      echo "Information about user $OPTARG will be shown"
      USERNAME=$OPTARG
      ;;
    *)
      echo "Unknown option! Use $0 -h to display help"
      exit 1
      ;;

  esac

done


shift $((OPTIND - 1))


[[ -z $USERNAME ]] && echo "User parameter is mandatory - use option -n or display
help using -h" && exit 1

if [[ $RUN_ID == 'Y' ]]; then
  echo "id $USERNAME"
  id $USERNAME
fi

if [[ $RUN_LAST == 'Y' ]]; then
  echo "last $USERNAME"
  last $USERNAME
fi

if [[ $RUN_DU == 'Y' ]]; then
  echo "du -sh /home/$USERNAME"
  du -sh /home/$USERNAME
fi

exit 0
```

Introduction to functions

Note

- Functions allow to use the same code multiple times, and are available in the most of programming languages. A function is a named set of commands.
- Functions may return exit code with the key word **return**.
- Functions may also output text. This text could be captured by the instruction calling the function.
- Syntax:

```
function  is_weekend()
{
    WEEKDAY=$(date +%w)
    if [[  $WEEKDAY -eq 0  ||  $WEEKDAY -eq 6 ]]; then
       echo “it is a weekend”
       return 0
    else
        echo “it is a working day”
        return 1
    fi
}
```

- To call the function use its name and:
  - If the function returns exit code, consume the result:

```
if is_weekend; then
     echo ”Hey it it a weekend”
fi
```

  - If the function outputs text, it can be consumed using $()

```
MESSAGE=$(is_weekend)
echo $MESSAGE
```

Lab

1. You are going to build s library of scripts that will be used to automate the system. Create a function that will return 0 when the /rmp/logs consumes less than 1 MB or 1 if the directory is bigger:
   a. Create directory /tmp/logs and copy there some files
   b. Create and test the function is_small returning 0 or 1 depending on the directory size
2. Change the function so, that it will additionaly display information about directory size. Change the line calling the function so, that the generated by this function output will be saved in a variable.
3. Save the function. It will be needed in the next labs.

Solution proposal

```
#!/bin/bash

function is_small()
{
  DIR=/tmp/logs
  SIZE_KB=$(du -s $DIR | cut -f 1)

  if [[ $SIZE_KB -lt 1024 ]]; then
    echo "Directory $DIR is still small $SIZE_KB KB"
    return 0
  else
    echo "Directory $DIR is big ($SIZE_KB KB)"
    return 1
  fi
}


MESSAGE=$(is_small)
IS_SMALL=$?

echo $IS_SMALL

if [[ $IS_SMALL -eq 0 ]]; then
  echo "No action needed: $MESSAGE"
else
  echo "ACTION NEEDED: $MESSAGE"
fi

exit 0
```

Passing Arguments to Functions

Note

- Function parameters work similarly to script parameters:
    - To access parameter's value use $1, $2, $3 etc.
    - To call a function with parameters, just add the parameter's values after the name of the function
- If the function defines new variables, or changes values of existing variables, the changes will be visible also outside of the function
- If the variable should stay as visible only internally in the function (not visible from outside of the function), it should be declared as local, add before the variable name the word **local**:

```
function user_dir()
{
   local USER=$1
   local homedir=$(grep $USER /etc/passwd | cut -d : -f 6)
   echo $homedir
}

user_dir   lucy
```

Lab

1. Change definition of the function from the previous lab, so that::
    a. The path of the tested directory could be passed as the first parameter ($1)
    b. The size of the directory that allows to determine the tested directory as small or big should be also sent as a parameter ($2)
    c. All the variables defined internally in the function should be local (visible only inside the function and not visible from outside)

Solution proposal

```
#!/bin/bash

function is_small()
{
  local DIR=$1
  local MAX_SIZE=$2
  SIZE_KB=$(du -s $DIR | cut -f 1)

  if [[ $SIZE_KB -lt $MAX_SIZE ]]; then
    echo "Directory $DIR is still small $SIZE_KB KB"
    return 0
  else
    echo "Directory $DIR is big ($SIZE_KB KB)"
    return 1
  fi
}


MESSAGE=$(is_small '/tmp/logs1' 2048)
IS_SMALL=$?

echo $IS_SMALL

if [[ $IS_SMALL -eq 0 ]]; then
  echo "No action needed: $MESSAGE"
else
  echo "ACTION NEEDED: $MESSAGE"
fi

exit 0
```

Use Case – Recurrent Function

Note

- Functions in BASH can generally perform similar activities as functions in other programming languages like:
  - Call other functions
  - Call itself (recurrency)
  - Take parameters and return exit code
  - Ask the user for interactive input or in opposite perform only batch activities without user interaction
  - Can define a set of commands that will be reused multiple times in the script.
  - Can perform numerical calculations, analyze text, perform different activities on operating system level

Lab

1. In this lab you will prepare function that will be able to ask a user interactively for decision. The function should accept yes/no response. The actions taken by function:
   a. Display a message with question text – this text should be sent as an argument
   b. Display YES or NO prompt, so the user knows what answer is accepted
   c. Read the user answer
   d. Check the correctness of the answer. Following answers are accepted:
      i. YES or yes or Y or y – the function should return 0 and finish
      ii. NO or no or N or n – the function should return 1 and finish
      iii. In other cases, the function should display the question, prompt, and read the answer again, until it will be one of the accepted options
2. Test the function using following code:

```
if (read_yes_no 'Should I continue with disk formatting?'); then
  echo "CONTINUING"
else
  echo "STOPPING"
fi
```

3. Hints:
   a. If you wish to have a "never ending" loop, the condition may be defined like this:

```
while true; do
….
done
```

   b. Comparing texts by using the syntax [[ ]], use doubled equality symbol ==, to build an alternative use double pipe character ||
   c. To display question saved in the variable **QUESTION** and to read the user's answer into variable **response** use command **read**:

```
read -p $QUESTION response
```

Solution proposal

```
#!/bin/bash

function read_yes_no()
{
  local QUESTION=$1

  while true; do

    echo $QUESTION
    local response=''
    read -p 'Enter YES or NO: ' response

    if [[ $response == 'YES' || $response == 'Y' || $response == 'yes' || $response ==
'y' ]]; then
      return 0
    elif [[ $response == 'NO' || $response == 'N' || $response == 'no' || $response ==
'n' ]]; then
      return 1
    fi

  done
}

if (read_yes_no 'Should I continue with disk formatting?'); then
  echo "CONTINUING"
else
  echo "STOPPING"
fi

exit 0
```

Your own library of BASH Scripts

Note

- Libraries of functions allow to use multiple times the same functions in different scripts
- Library is a file that contains function definitions (without instructions calling them). The file don't have the exit statement.
- Library should have a kind of documentation, put in the comments at the beginning like:
    - Short description of the functions, the type of the problem, that they solve
    - Information about author
    - Version, last modification date etc
- Additionally, each of the functions also should have a short description/help, an example showing how to use a function is always welcome
- To use a function, it can be read into the script:
    - Using command source, or
    - Using a dot:

```
. ./fun_lib.sh
```

- The library can be also imported into the user's session. In such case all the defined in the script functions, will be ready to use by the user in his session. This makes impression of having new set of multiple useful commands

Lab

1. Create a library file, describe it as „course library" in version „1.0", enter information about athor and the reason of creating the library etc.
2. Add to the file, the functions created already earlier in the flow of the course. Each of the functions should be described. There should be example showing how to call the function.
3. Create script that:
    a. In the loop
        i. Asks user to enter user name (command **read**)
        ii. Get the path to a home directory of this user(function **user_dir**)
        iii. Estimates the size of the directory – if it is small or big (function **is_small**)
        iv. Asks, if the action should be performed once again. If yes – the loop should be executed again and if not, the loop and the script should end. When asking the user use the function **yes_no**)

Solution proposal

```
## FILE lib.sh





#!/bin/bash

# Title       : Course Library
# Version     : 1.0
# Author      : Mobilo24.eu
# Description : This is a final script from course "Scripting with BASH"


# Function finds home dir of a user sent as a parameter
# Returns: home directory path of a user
#
#  ----- Example 1  ------
#  user_dir lucy

function user_dir()
{
   local USER=$1
   local homedir=$(grep $USER /etc/passwd | cut -d : -f 6)
   echo $homedir
}


# Function checks if the directory is "small" or not
# Returns: 0 if the dir is small and 1 in other case
#
#  ----- Example 1  ------
#  MESSAGE=$(is_small '/tmp/logs1' 2048)
#  IS_SMALL=$?

function is_small()
{
  local DIR=$1
  local MAX_SIZE=$2
  SIZE_KB=$(du -s $DIR | cut -f 1)

  if [[ $SIZE_KB -lt $MAX_SIZE ]]; then
    echo "Directory $DIR is still small $SIZE_KB KB"
    return 0
  else
    echo "Directory $DIR is big ($SIZE_KB KB)"
    return 1
  fi
}


# Function reads answer yes/no or y/n or YES/NO or Y/N
# Returs: 0 for "yes" and 1 for "no"
#
#  ----- Example 1  ------
#  if (read_yes_no 'Should I continue with disk formatting?'); then
#     echo "CONTINUING"
#  else
#     echo "STOPPING"
#  fi

function read_yes_no()
{
  local QUESTION=$1

  while true; do
```

```
    echo $QUESTION
    local response=''
    read -p 'Enter YES or NO: ' response

    if [[ $response == 'YES' || $response == 'Y' || $response == 'yes' || $response ==
'y' ]]; then
      return 0
    elif [[ $response == 'NO' || $response == 'N' || $response == 'no' || $response ==
'n' ]]; then
      return 1
    fi

  done
}




## FILE my_script.sh




#!/bin/bash

. ./lib.sh


while $true; do

  NAME=''
  read -p "Enter name of the user that's home dir should be checked " NAME

  HOMEDIR=$(user_dir $NAME)
  echo "Home dir of user $NAME is $HOMEDIR"

  if (is_small "$HOMEDIR" 1024); then
    echo "This directory is small"
  else
    echo "This directory can be checked - it is big"
  fi

  if ! read_yes_no "Continue for a next user? " ; then
    break
  fi

done

exit 0
```

See also

PowerShell
essential course with labs
Mobilo

SAP ASE 16 / Sybase
Installation and Configuration
Mobilo
Linux