Scripts & Functions

SCRIPTS	
true	returns a successful (zero) return code
false	returns an unsuccessful (non-zero) return code
test <condition> | **[** <condition> **]** | **[[** <condition> **]]**	evaluates a specified condition and based on its result it returns a successful or unsuccessful exit status; to test the existence of specific files the following options are used with a file name as an argument: **-e** any file, **-f** regular file, **-b** block device, **-c** character device, **-d** directory, **-h** symbolic link, **-s** file with non-zero size, **-r** file with read permissions, **-w** file with write permissions, **-x** executable file, **-u** file with an SUID bit, **-g** file with an SGID bit, **-k** file with a sticky bit, **-p** named pipe, <file1> **-nt** <file2> file1 is newer than file2, <file1> **-ot** <file2> file1 is older than file2; to test integers the following operators are used: **-eq** is equal, **-ne** is not equal, **-lt** less than, **-le** less than or equal, **-gt** greater than, **-ge** greater than or equal; to test strings: **=** is equal, **!=** is not equal, **-z** <string> the length of string is zero, **-n** <string> the length of string is non-zero; extended testing is provided by operators: **-a** logical AND (both expressions are true), **-o** logical OR (at least one expression is true), **!** logical NOT (expression is false); **[[ ]]** provide more advanced expression testing such as <string> **=** / **==** <pattern> (the string contains a pattern), <string> **!=** <pattern> (the string does not contain a pattern), <string> **=~** <pattern> (the string matches a regular expression), **!** <string> **=~** <pattern> (the string does not match a regular expression), <string1> **<** <string2> or <string1> **>** <string2> (string1 is lexicographically before or after string2) and compound expressions like **&&** to evaluate the second expression if the first is true, **||** to evaluate the second expression if the first is false and **()** to group expressions `$ test -f *.txt	$ [-f *.txt]` (return code is 0 if only one file with the ".txt" extension exists) `$ [[-n $(find . -name "*.txt")]]` (return code is 0 if at least one file with the ".txt" extension exists) `$ [! -d PROD]` (return code is 0 if the directory does not exist) `$ [[abc == *c]]` (return code is 0 if the string "abc" contains a pattern "*c") `$ [["abc" =~ .*c]]` (return code is 0 if the string "abc" matches a regular expression ".*c") `$ [[a < z]]` (return code is 0 if the string "a" is lexicographically before a string "z") `$ [[$(awk -F ":" '/^root/ {print $4}' /etc/passwd) -eq 0]]` (return code is 0 if the root user's GID is 0) `$ [$? -ne 0] && echo $?` (prints the return code of the previous command if it failed) `$ [$PWD = $HOME] && echo "home" \|\| pwd` (prints "home" if the logged-in user is in the home directory, otherwise the path to the working directory is displayed) `# duid=$(awk -F ":" '{print $3}' /etc/passwd \| sort \| uniq -d); [[-z "$duid"]] && echo "NONE" || echo "$duid"` (prints duplicate UIDs in the system)
if <list>**; then** <list>**;** [**elif** <list>**; then** <list>**;**] ... [**else** <list>**;**] **fi**	**if** specifies a condition, if true, then **then** specifies a required action, if not true, **elif** specifies another condition and **then** specifies another action, otherwise **else** specifies an alternative action, the list of conditions is closed with **fi** `#!/bin/bash` `# The script checks if the logged-in user is root. If the user is root, it prompts for commands. If the user is not root, it informs them that they are not allowed to run the script and exits with an error message.` `if [[ $EUID -ne 0 ]]; then` `  echo "User '$(whoami)' is not allowed to run ${0##*/}."` `  exit 1` `fi` `read -p "Enter your commands: " commands` `eval "$commands"` `echo "Script completed successfully."` `#!/bin/bash` `# The script verifies if a specified package is installed.` `echo -n "Enter the package name: "` `while read PACKAGE` `do` `  rpm -q $PACKAGE > /dev/null` `  if [ $? -eq 0 ]` `  then` `  echo "YES, $PACKAGE is installed."` `  else` `  echo "NO, $PACKAGE is not installed."` `fi` `. $0` `done`

SCRIPTS	
case <word> **in** <pattern>[|<pattern>]...**)** <list>**;; esac**	executes a command based on the match of the word with the pattern; the word represents any variable, the pattern "*" specifies an action if there is no match, the list of patterns is closed with ")" and the group of commands ends with ";;" `#!/bin/bash` `# The script installs, updates or uninstalls a package, depending on the operation.` `echo -n 'Enter the operation and package: '` `read OPERATION PACKAGE` `case $OPERATION in` `  install)  yum install -y $PACKAGE;;` `  update)  yum update -y $PACKAGE;;` `  uninstall|remove)  yum remove -y $PACKAGE;;` `  *)  echo 'Unknown operation.'` `  echo 'Usage: {install|update|uninstall|remove} <package>';;` `esac`
read <variable ...>	reads the line from STDIN and splits it into words which are then assigned to the specified number of variables in the corresponding order for further processing; if there are more words than variables, the remaining words and their delimiters are assigned to the last variable, if there are more variables than words, the remaining variables are assigned empty values, **-a** <indexed_array> specifies an indexed array where the words are assigned to sequential indexes, **-n** <n> reads only a specified number of characters, **-p** <prompt> specifies a prompt that is displayed to the user before reading the input, **-r** suppresses a special meaning of the "\" character, **-s** does not display any characters on the input, **-t** <n> sets a time interval in seconds for reading the input `$ cat numbers.txt | { read a; read b; read c; echo "$a/$b/$c";}` (reads the first three lines of a file and prints them on one line separated by a slash) `#!/bin/bash` `# The script prompts the user for a username and password and confirms its acceptance.` `echo -n "Username: "; read username` `echo -n "Password: "; read -s password` `echo` `echo "Password for $username submitted."`
while <list1>**; do** <list2>**; done**	executes repeatedly "list2" until "list1" returns a successful return code `#!/bin/bash` `# The script opens four terminal screens.` `i=0` `while [ $i -lt 4 ]; do xterm &` `  i=$[$i+1]; done` `#!/bin/bash` `# The script creates a file with information about the currently logged-in users in an interval of five minutes.` `while true; do` `  who > info-$(date | awk '{print $2,$3,$4,$6}' | sed 's/ /_/g').txt` `  sleep 300; done` `#!/bin/bash` `# The script prompts the user for the name of the remote computer, on which it runs a command that detects the version of the Linux distribution and saves the result to a file on the local computer.` `echo -n "Hostname: "; while read hostname; do ssh $hostname 'echo "Linux distribution": $(cat /etc/redhat-release)' > ${hostname}_info && echo -n "Hostname: "; done`

SCRIPTS	
until <list1>**; do** <list2>**; done**	executes repeatedly "list2" until "list1" returns an unsuccessful return code #!/bin/bash # The script prints numbers 0–99. a=-1 until [$a -eq 99]; do a=$(expr $a + 1) echo $a; done #!/bin/bash # The script copies files from the home to the web directory, where it creates a new directory every hour, deleting directories older than 30 days if it is more than 90% full. while true; do DISKFUL=$(df -h $WEBDIR | grep "^/" | \ awk '{print $5}' | cut -d "%" -f1 -) until [$DISKFUL -ge 90]; do WEBDIR=/var/www/webcam DATE=$(date +%Y%m%d) HOUR=$(date +%H) mkdir $WEBDIR/"$DATE" while [$HOUR -ne 00]; do PICDIR=/home/user/pics DESTDIR=$WEBDIR/"$DATE"/"$HOUR" mkdir "$DESTDIR" cp $PICDIR/*.jpg "$DESTDIR"/ sleep 3600 HOUR=$(date +%H); done DISKFULL=$(df -h $WEBDIR | grep "^/" | \ awk '{print $5}' \| cut -d "%" -f1 -) done TOREMOVE=$(find $WEBDIR -type d -mtime +30) for i in $TOREMOVE; do rm -rf "$i"; done done

SCRIPTS	
for <variable> [**in** <word> ...**;**] **do** <list>**; done**	assigns the values of all words or all positional parameters of the script to the variable one by one and executes the list of commands for each of the items `#!/bin/bash` `# The script displays each command-line argument along with its corresponding number.` `count=1` `for arg; do` `  echo "Argument $count: $arg"` `  ((count++))` `done` `#!/bin/bash` `# The script creates users specified in the "users" file in the working directory, adds them to the "admins" group and assigns the "password" as their password.` `for user in $(cat users); do` `  useradd -g admins $user` `  echo password | passwd --stdin $user` `done` `#!/bin/bash` `# The script prints the contents of the working directory.` `for x in $(ls -F)` `do` `  echo "Directory $(pwd) contains file or directory: $x"` `done` `#!/bin/bash` `# The script counts the number of items in the working directory.` `POC=0` `for name in *` `  do` `  POC=$(($POC+1))` `  done` `echo "Directory $(pwd) contains $POC items."` `#!/bin/bash` `# The script converts the format of all .tif files to .jpg.` `for pic in *.tif; do` `  convert "$pic" "$(echo "$pic" | sed 's/\.tif/.jpg/')"` `done` `#!/bin/bash` `# The script checks which of the specified processes are running, prints their names including the number of lines they occupy and finally prints all running processes and the total number of lines.` `services="httpd pmon lsnr sap"` `for service in $services` `  do running_processes="$(ps -ef | grep $service \| egrep -v 'grep\|vnc\|client\|API')"` `  if [ -n "$running_processes" ]` `  then echo` `  echo $service:` `  echo` `  echo "$running_processes"` `  echo` `  echo "lines: $(echo "$running_processes" | wc -l)"` `  echo` `  echo "=========="` `  fi` `  done` `echo` `echo "Summary:"` `echo` `ps -ef` `echo` `echo "lines: $(ps -ef | wc -l)"` `#!/bin/bash` `# The script connects to all servers whose IP addresses are listed in the "servers" file, runs commands on them that find out the hostname and all its IP addresses, and writes the result to a file on the local computer.` `for ip in $(< servers); do ssh $ip 'hostname; ifconfig | awk '/inet / {print $2}' \| grep -v 127.0.0.1' >> servers_info.txt; done` `#!/bin/bash` `# The script connects to remote computers, where it installs software for the specified service, which then starts and verifies its status.` `for x in {a..c}; do echo "=== node${x} ==="; ssh node${x} 'yum install -y device-mapper-multipath; systemctl enable --now multipathd; systemctl status multipathd'; echo; done`

SCRIPTS	
select <variable> [**in** <word> ...**;**] **do** <list>**; done**	assigns the values of all words or all positional parameters of the script to the variable one by one, the numbered list of which is displayed together with a prompt for the user to enter the item number and then executes a list of commands for the selected item (interactive equivalent to the "for" command); the line read is saved in the variable "REPLY" `#!/bin/bash` `# The script displays a menu of services. After selecting one of them, a submenu of operations that can be performed appears.` `while true; do` `PS3="Select the service or quit: "` `select svc in cups httpd sshd quit` `do` `  case $svc in` `  cups)` `  PS3="Select the operation or quit: "` `  select opt in start stop restart status quit` `  do` `  case $opt in` `  start)` `  systemctl start cups` `  break;;` `  stop)` `  systemctl stop cups` `  break;;` `  restart)` `  systemctl restart cups` `  break;;` `  status)` `  systemctl status cups` `  break;;` `  quit)` `  break;;` `  *)` `  echo "Invalid option $REPLY"` `  break;;` `  esac` `  done` `  break;;` `  httpd)` `  PS3="Select the operation or quit: "` `  select opt in start stop restart status quit` `  do` `  case $opt in` `  start)` `  systemctl start httpd` `  break;;` `  stop)` `  systemctl stop httpd` `  break;;` `  restart)` `  systemctl restart httpd` `  break;;` `  status)` `  systemctl status httpd` `  break;;` `  quit)` `  break;;` `  *)` `  echo "Invalid option $REPLY"` `  break;;` `  esac` `  done` `  break;;` `  sshd)` `  PS3="Select the operation or quit: "` `  select opt in start stop restart status quit` `  do` `  case $opt in` `  start)` `  systemctl start sshd` `  break;;` `  stop)` `  systemctl stop sshd` `  break;;` `  restart)` `  systemctl restart sshd` `  break;;` `  status)` `  systemctl status sshd` `  break;;` `  quit)` `  break;;` `  *)` `  echo "Invalid option $REPLY"` `  break;;` `  esac` `  done` `  break;;` `  quit)` `  break 2;;` `  *)` `  echo "Invalid option $REPLY";;` `  esac` `done; done`

SCRIPTS	
break [<n>]	exits definitively the specific cycle within a "for", "while", "until", or "select" loop based on certain criteria, *n* specifies the number of levels of nested cycles `#!/bin/bash` `# The script executes a loop that should iterate through numbers from 1 to 5, but prematurely exits the loop during the third iteration when the value of the "cycle" variable becomes 3.` `for cycle in {1..5}; do` `  if [ "$cycle" -eq 3 ]; then` `  echo "Breaking at iteration $cycle"` `  break` `  fi` `  echo "Iteration: $cycle"` `done` `echo "Loop ended."`
continue [<n>]	skips the specific cycle within a "for", "while", "until", or "select" loop based on certain criteria and proceeds to the next one, *n* specifies the cycle level `#!/bin/bash` `# The script executes a loop that should iterate through numbers from 1 to 5, but skips the third iteration when the value of the "cycle" variable becomes 3 and proceeds to the next iteration.` `for cycle in {1..5}; do` `  if [ "$cycle" -eq 3 ]; then` `  echo "Skipping iteration $cycle"` `  continue` `  fi` `  echo "Iteration: $cycle"` `done` `#!/bin/bash` `# The script converts uppercase letters in file names to lowercase.` `LIST=$(ls)` `for name in $LIST; do` `  if [[ $name != *[[:upper:]]* ]]; then` `  continue; fi` `ORIG=$name` `NEW=$(echo $name | tr 'A-Z' 'a-z')` `mv $ORIG $NEW` `echo "New name for $ORIG is $NEW"` `done`
getopts <optstring> <name> [<arguments>]	parses (processes) positional parameters (options and arguments) that are passed to a shell script; <optstring> specifies the list of supported option characters, if a character is followed by a colon (:), the option is expected to have an argument, if the first character is preceded by a colon, silent error reporting is used; each time "getopts" is invoked, it obtains the next option from the positional parameters and places the option character in the variable <name>, if the option does not match those defined in <optstring>, "getopts" sets variable <name> to a question mark (?) and, if silent error reporting is used, the option character found is placed in the "OPTARG" variable; if an option requires an argument, "getopts" places that argument in the "OPTARG" variable, if the required argument is not found and silent error reporting is used, a colon is placed in the <name> variable and "OPTARG" is set to the option character found; when executing the script, the options that require an argument must be followed by their arguments before another option is used `#!/bin/bash` `# The script allows to use only strictly defined options and expected arguments for its execution.` `usage() {` `  echo "Usage: $0 {-x\|-y <string>}"` `  exit 1` `}` `[[ ! "$@" =~ -.+ ]] && usage` `while getopts :xy: option` `do` `  case $option in` `  x)` `  echo "Option '-x' is used."` `  ;;` `  y)` `  echo "Option '-y' with argument '$OPTARG' is used."` `  ;;` `  \?)` `  echo "Invalid option '-$OPTARG'."` `  usage` `  ;;` `  :)` `  echo "Option '-$OPTARG' requires an argument."` `  usage` `  ;;` `  esac` `done`
exit [<n>]	exits a script with the specified return code, if not specified, the return code of the script is the return code of the last command executed within the script

FUNCTIONS	
function <function> **{** <list>**;}** | <function> **() {** <list>**;}**	defines a function whose contents are enclosed in braces; the function is executed in the current shell and is used especially in cases where a specific operation (sequence of commands) needs to be performed repeatedly; it is executed by entering its name and by default returns the return code of the last executed command, unless otherwise specified by the "return" command; inside the function, a local variable can be defined with the "local" command (it is also inherited into nested functions), if there is a global variable with the same name, it is suppressed in the function; the function can be used permanently (even in future shell executions) by including it in *~/.bashrc*; information about arguments passed to scripts or functions is stored in special variables ("$#" = total number of arguments, "$<n>" = specified argument in order ranging from 1–9, "${<n>}" = specified argument in order in any range, "$*" or "$@" = all arguments) `# The "test" function creates a file that is specified as an argument and sets appropriate permissions to it. If a file with the same name already exists or the specified name contains only a space or is completely missing, an error message is displayed.` `function test` `{` `if ! [[ -e $* || $* = '' ]]; then` `  touch $*; chmod 755 $*` `else` `  echo "Error"` `fi` `echo "Number of arguments: $#"` `echo "All arguments: $*"` `echo "Second argument: $2"` `}` `# The "user" function checks if the user specified as an argument exists in the system and, if so, prints detailed information about the user.` `user ()` `{` `if [ $(grep -ic ^$1 /etc/passwd) -eq 0 ]; then` `  echo "User $1 does not exist."` `else` `  echo "User $1 exists." && finger $1` `fi` `}` `# The "percentage" function calculates the percentage value of the second numeric argument from the first one, where the first argument represents a value of 100%.` `percentage () {` `local x=$1 y=$2` `printf "$x = 100%%\n$y = x%%\n"` `echo "$y = $[$(( 100 * $y )) / $x]% from $x"` `}` `# The "usage" function prints possible options to execute the script, if no valid option has been used, it returns a return code 1.` `usage () { echo "Usage: $0 {-start|-stop|-restart}"; exit 1;}` `# The "size" function calculates the size of the specified directory.` `size() {` `if [ -z "$1" ]; then echo "Usage: size <directory>"; return 1; fi` `[ -d "$1" ] || { echo "Directory not found: $1"; return 1; }` `echo "Scanning directory tree: $1 ..."` `find "$1" -type f -printf '%s\n' | \` `awk '{sum += $1} END {printf "Bytes: %d | MB: %.2f | GB: %.2f\n", sum, sum/1024/1024, sum/1024/1024/1024}'` `}`

FUNCTIONS	
local [<variable>[=<value>] ...]	defines a local function variable, **-a** specifies an indexed array, **-A** specifies an associative array; with no argument all local function variables are displayed
return [<n>]	exits a function with the specified return code, if not specified, the return code of the function is the return code of the last command executed within the function