LINUX SHELL SCRIPTING

Emanuel Shmuelov

Linux Shell Scripting

3rd Edition, Nov 2010

Author: *Emanuel Shmuelov*

Cover Image & Design: *Eshed Yam/Dprint*

Legal Disclaimer

This book is intended to provide fundamental information about Linux operating system. Although we endeavor to keep this book as up-to-date as possible, we appreciate readers to share with us the concern for accuracy in the text and examples.

TABLE OF CONTENT

1. General Notes

- ✓ Linux operating system supports more than one way to execute many requested operations. This book describes the most common ways used to execute operations/requests.
- ✓ The shell used at the examples of this book is BASH.
- ✓ Lines starting with "`[user@localhost ~]#`" meaning command line prompt.
- ✓ At the command line explanation, content within [] square parenthesis indicates optional, and content within <> indicates mandatory.
- ✓ The Linux operating system provides a long list of command line utilities. Each utility usually supports many features, hence provides a range of properties. Prior to using a command line utility, read its manual by typing:

```
[user@localhost ~]# man <command name>
```

2. Linux Introduction

2.1. What is a shell?

Sometimes called *command shell*, a shell is the command processor interface. The command processor is the program that executes operating system commands. The shell, therefore, is the part of the command processor that accepts commands. After verifying that the commands are valid, the shell sends them to another part of the command processor to be executed.

Linux offers a choice between several different shells. The most popular are *Cshell (CSH), Bourne shell (BASH)*, *TCSH* and the *Korn shell (KSH).* Each offers a somewhat different command language.

2.2. Wildcards & Special Characters

Wildcards is a shell feature that makes the command line much more powerful than any GUI file manager. Wildcards are special characters allowing select filenames that match certain patterns of characters. This helps to select even a big group of files by typing just few characters.

Here's a list of the most commonly used wildcards in BASH:

Symbol	Meaning	Example
*	Match any string	`ls *`
?	Match any single string	`ls ?`
[…]	Match any enclosed characters	`cat [abc]test`
[…-…]	Match exactly one character in the given range	`mv [a-g]test /tmp`
{word1,word2}	Match exactly one entire word in the given options	`rm {con,self}test`

2.3. Quotes

The Linux command-line interpreter supports enclosing string in quotes as part of the command usage, as follows.

Quote	Name	Meaning
“	Double quote	Anything enclose in double quotes manipulated as **string**

Quote	Name	Meaning
		Note: except \ and $ and `
‘	Single quote	Anything enclosed in single quotes remains unchanged
`	Back quote	Anything enclosed is executed as **command**

In order to understand the quotes usage we will introduce the `echo` command first:

```
echo        -     Display a line of text
[user@localhost ~]# echo [options] [strings]
```

Common options	Description
-n	Do not output trailing newline (\n)
-e	Enable interpretation of the backslash-escaped characters

Examples for common uses of “**echo**” command:

```
[user@localhost ~]# echo "hello world"      # Print string
hello world

[user@localhost ~]# echo $USER              # Print system variable
root

[user@localhost ~]# echo -n "test"          # Print w/o new line
test[user@localhost ~]#

# Print while controlling the output display
[user@localhost ~]# echo -e "First line\n\tSecond line with tab"
First line
        Second line with tab
[user@localhost ~]#
```

Please refer to chapter 3.5.1 for more details about “**echo**” command.

Quotes usage examples:

```
[user@localhost ~]# A=123               # Define variable A as “123”
[user@localhost ~]# echo “A is: $A”     # Output: A is: 123
[user@localhost ~]# echo ‘A is: $A’     # Output: A is: $A
[user@localhost ~]# echo `A is: $A`     # Output: command not found
```

2.4. I/O redirections and Pipes

Most Linux commands read input and write output. By default, input is being given using the keyboard, and the output is displayed to the terminal.

The keyboard is the standard input (stdin) device, and the terminal is the standard output (stdout) device.

However, these default settings don't necessarily have to be applied. The standard output may be a file or a printing device and the standard input may be a text file etc.

Input and output redirection is done by the following operators:

Operator	Syntax	Description	Comments
>	command **>** filename	Command's output redirected to the BEGINNING of file	New file is created if filename does not exist
>>	command **>>** filename	Command's output redirected to the END of file (appended)	
<	command **<** filename	Input redirection to command from file	Not all commands support this redirection mode
|	command1 **|** command2	Pipe - redirect output of command1 as an input of command2	

Examples:

```
# Redirect output of "echo" to the BEGINNING of file 2.log
[user@localhost ~]# echo "hello world" > 2.log
```

```
# Append output of "echo" to the END of file 2.log
[user@localhost ~]# echo "hello world" >> 2.log
```

```
# Input redirection – sending mail using mail body from file instead
from user (stdin)
# Note: "mail" is a command line utility for delivering email
[user@localhost ~]# mail emanuel@gmail.com < mail_body.txt
```

```
# Redirect the output of "echo" to "mail" command
[user@localhost ~]# echo "Hello Emanuel" | mail emanuel@gmail.com
```

```
# Combine input and output redirection. The file text.txt is first
checked for spelling mistakes, and the output is redirected to log
file:
# Note: "spell" is a command-line utility for spell checking
[user@localhost ~]# spell < book.txt > spelling_mistakes.log
```

Output of one command can be piped into another command virtually many times, as long as these commands would normally read input from standard input and write output to the standard output.

Every open file in the system assigned with a unique file descriptor which is used by the operating system stream management to and from the file. Similar to any other open file in the system, the standard input and output also have unique file descriptor as if they were files.

Naturally, their file descriptors ID are the initial ones:

I/O	File descriptor ID	Device name
Standard **input**	0	/dev/stdin
Standard **output**	1	/dev/stdout
Standard **error**	2	/dev/stderr

If the file descriptor number is omitted, and the first character of the redirection operator is <, the redirection refers to the standard input (file descriptor 0).

If the first character of the redirection operator is >, the redirection refers to the standard output (file descriptor 1).

Tip: A very commonly used system device is the **/dev/null**. This is a special device discarding all data redirected to it.

It is possible to explicitly provide the file descriptor ID to the command. The syntax usage is as follows:

```
# List all files under /etc and redirect the standard output into
"file_list.txt"
[user@localhost ~]# ls /etc > file_list.txt
[user@localhost ~]# ls /etc 1> file_list.txt        # Equal result

# List to terminal all files under /etc and redirect the error messages
(standard error) to "list_err.txt"
[user@localhost ~]# ls /etc 2> list_err.txt
```

```
# List all files under /etc and redirect the standard output into
"file_list.txt" and error messages (standard error) to "list_err.txt"
[user@localhost ~]# ls -l /etc > file_list.txt 2> list_err.txt


# List all files under / and redirect the standard output into
"file_list.txt" and append error messages (standard error) to
"list_err.txt"
[user@localhost ~]# ls -l / > file_list.txt 2>> list_err.txt


# List all files under / and redirect the standard output into
"file_list.txt" and error messages (standard error) to /dev/null
[user@localhost ~]# ls -l / > file_list.txt 2> /dev/null


# List all files under / and redirect both the standard output and
standard error into "file_list.txt".
[user@localhost ~]# ls -l / > file_list.txt 2> file_list.txt
```

Making things complicated…

What the following command does?

```
[user@localhost ~]# kill -9 8198 > /dev/null 2>&1
```

Answer: kills process with PID 8198 (by sending signal SIGKILL [9] to the process), and redirects any output of the kill command including error outputs to /dev/null;

How come?

Lets explain it step by step:

```
[user@localhost ~]# ls /tmp > dirlist.txt
```

Redirect standard output to the file *dirlist.txt*.

```
[user@localhost ~]# ls /tmp > dirlist.txt 2>/dev/null
```

Redirect standard output to the file *dirlist.txt* and the standard error to /dev/null.

```
[user@localhost ~]# ls /tmp > /dev/null
```

Redirect standard output to /dev/null.

```
[user@localhost ~]# ls /tmp > /dev/null 2>&1
```

Redirect standard output to /dev/null and standard error to standard output - which is already redirected to /dev/null.

Pay attention: the location of `2>&1` is important and should be located at the end of the command line:

```
[user@localhost ~]# ls /tmp 2>&1 > dirlist.txt
```

At the example above, "ls" will only redirect standard output to *dirlist.txt* and the standard error to the terminal. This happens because the standard error is duplicated as standard output **before** the standard output was redirected to *dirlist.txt*.

The use of the ampersand (**&**) indicates that the number following is not a file name, but rather a file descriptor that the data stream is pointed to.

Important: the "&" sign should not be separated from the "2>" by spaces. If it would be separated, it would be pointing the output to a file again. The example below demonstrates this:

```
[user@localhost ~]# ls 2> tmp.log    # redirects ls errors to file
[user@localhost ~]# ls 2 > tmp.log   # Lists file named 2 to file
ls: 2: No such file or directory

# Following example will result with error due to the space set before
the "&" character
[user@localhost ~]# ls /tmp > dirlist.txt 2> &1
-bash: syntax error near unexpected token `&'

# Following examples are equal (space after the "&" sign is legal)
[user@localhost ~]# ls /tmp > dirlist.txt 2>&1
[user@localhost ~]# ls /tmp > dirlist.txt 2>& 1
```

2.5. Shell aliases

Shell aliases make it easier to use commands by facilitating abbreviated command names and by enabling pre-specification of common arguments. To establish a shell command alias, use the following command form:

```
alias name='command'
```

Examples:

```
[user@localhost ~]# alias ll="ls -ltr"     # defining new alias named
                                           "ll"
[user@localhost ~]# ll                     # using the new alias "ll"
                                           will list the current
                                           directory content
```

Linux usually has a pre-defined aliases set. In order to list all aliases run "alias" command with not parameters:

```
[user@localhost ~]# alias     # list current aliases set
alias cp='cp -i'
alias df='df -lh'
```

```
alias ll='ls -l --color=tty'
alias mv='mv -i'
alias rm='rm -i'
```

In order to unset existing alias use the “unalias” command:

```
unalias <alias name>
```

Example:

```
[user@localhost ~]# unalias ll       # del “ll” from the alias list
```

Note: Aliases are not expanded when the shell is not interactive (e.g. in scripts).

2.6. Processes and Job control

Linux operating system is a multi-user multi-process operating system, enabling multiple users running multiple commands at the same time.

While some programs initiate a single process like **rm**, others initiate a series of processes, such as **httpd** web server.

Every process is allocated with a unique ID during its execution called PID, and a dedicated directory containing full process details resides at the **/proc** filesystem.

The **/proc** filesystem is the kernel’s memory map. Its content is cleared on system shutdown, therefore while listing the content of the /proc filesystem, most of the files and directories are zero sized, and no physical allocation consumed from the hard drive.

In order to manage the processes running within the system, Linux operating system provides a set of process management utilities.

The commonly used utilities for process management are:

```
ps          -     Report a snapshot of the current processes
[user@localhost ~]# ps [options]
```

Common options	Description
-e or -A	List all processes on the system
-C <cmd name>	List all processes corresponding to the <cmd name>
-U <uid>	List all processes corresponding to the <user ID>
-o <format>	Format the output according to a predefined format
-f	List processes using full-format listing

Common options	Description
-F	List processes using extended full-format listing
-u	List processes using user-oriented format
-H	List processes hierarchy

Note: some options can't be used with others. Refer to the user manual of "ps" command for more information.

Examples for common uses of the "**ps**" command:

```
[user@localhost ~]# ps -eF
[user@localhost ~]# ps -AH
[user@localhost ~]# ps -u
[user@localhost ~]# ps -U root
[user@localhost ~]# ps H
[user@localhost ~]# ps -C sleep
[user@localhost ~]# ps -o "%u%g%p%c"
```

```
top        -     Linux task manager
[user@localhost ~]# top
```

Common options	Description
W	Save configuration changes
h	Open help menu
q	Quit
d	Set update interval
1	Toggle single CPU vs. separate CPU information
f	Select sort field

```
pstree     -     Display system's processes using tree view
[user@localhost ~]# pstree [options]
```

Common options	Description
-a	Show command line arguments
-p	Show process ID

```
pidof       -     Find the process ID of a running program
[user@localhost ~]# pidof [options] <program name>
```

Common options	Description
-s	Show the PID of the first process found
-x	Show process ID of scripts in addition to programs

Examples for common uses of "**pidof**" command:

```
[user@localhost ~]# pidof bash
[user@localhost ~]# pidof -x test.sh
[user@localhost ~]# pidof -s httpd
```

```
kill        -     Notify process with signal
[user@localhost ~]# kill [options] <process ID>
```

Common options	Description
-s <signal id>	Send specific signal to process
-l	Print available signals

The kill command enables to send a wide range of signals to a process in order to control the process's execution, i.e. stop, suspend, abort, alarm etc.

More than 60 different signals are supported by the Linux operating systems. For more information regarding signals and their variation please refer to the manual of signal (7) as follows:

```
[user@localhost ~]# man 7 signal
```

In case no signal is provided by the command line, by default, the TERM signal is sent to the process.

Examples for common uses of the "**kill**" command:

```
[user@localhost ~]# kill 4450         # Send TERM signal to process
[user@localhost ~]# kill -s 9 4454    # Send signal 9 to process
[user@localhost ~]# kill -SIGHUP 458  # Send SIGHUP signal to process
```

```
time        -     Monitor execution time of a command
[user@localhost ~]# time <command line>
```

Examples for common uses of the "**time**" command:

```
[user@localhost ~]# time ps -f       # monitor running time of "ps -f"
UID         PID   PPID   C STIME TTY          TIME CMD     # Output of the
```

```
root      6212  6210  0 16:43 pts/2    00:00:00 -bash  "ps -f" command
root      9234  6212  0 19:57 pts/2    00:00:00 ps -f

real    0m0.064s              # Output by the "time" command
user    0m0.006s
sys     0m0.024s
```

Linux command line supports execution of several commands in queue written in one sentence by separating the commands with ";" symbol, for example:

```
# The following example shows execution of 3 commands, one after the
other using the ";" symbol separating each. The commands are executed
according to the written order.
[user@localhost ~]# ls -l ; rm -f /tmp/1.log ; cp ./file /root
```

2.6.1. Job control and Signals

Linux processes might be running in two modes:

- ✓ Run in *foreground*
- ✓ Run in *background*

Run in *foreground* - every regular command/program which is executed via the shell is by default running in foreground, means, the shell session is occupied during its execution.

Run in *background* - process is executed, but the shell session can accept new commands during its execution.

Processes in background might reside in two modes: *Running* or *Stopped* as follows:

Running - the process is working in background

Stopped - the process is hanged until signaled to continue run

Job control refers to the ability to selectively stop the execution of a process and resume its execution at a later time.

When the shell starts a job asynchronously, it prints a line that looks like: **`[1] 25433`**

Meaning:

`[1]` - Indicating the index number of the job at the shell's job control list

`25433` - Indicating the process unique ID

A process can be switched from running in foreground to run in background and vice versa. There are several ways to change process's run mode:

Mode	Operations to be taken
Run in fg	Execute the command regularly
Run in bg	Add "**&**" to the end of the command line
Switch from running in **fg** to running in **bg**	When the program is running in foreground do: 1. Press **Ctrl+z** (suspends the program execution) 2. Type **bg**
Switch from running in **bg** to running in **fg**	1. Run **jobs** command and verify the command's index 2. Run **fg n** (where "n" is the process's index ID from the jobs output)

Pressing the Ctrl+z sequence while a process is running causes the process to be stopped and returns control to the shell. A Ctrl+z takes effect immediately.

Examples:

```
# Run the program in foreground
[user@localhost ~]# sleep 1000

# Run the program in background
[user@localhost ~]# sleep 1000 &
[3] 8507

# Switch running mode from foreground to background
[user@localhost ~]# sleep 1500        # run program regularly
Ctrl+z                                # suspend it using Ctrl+z
[1]+  Stopped         sleep 1500      # process suspended
[user@localhost ~]# bg 1              # change mode to background
[1]+ sleep 1500 &

# Switch running mode from background to foreground
[user@localhost ~]# sleep 1000 &      # run program in background
[1] 8584
[user@localhost ~]# sleep 1500 &      # run program in background
[2] 8585
[user@localhost ~]# sleep 2000 &      # run program in background
[3] 8586
[user@localhost ~]# jobs              # list background processes
```

```
[1]   Running          sleep 1000 &
[2]-  Running          sleep 1500 &
[3]+  Running          sleep 2000 &
[user@localhost ~]# fg 2              # run program [2] in foreground
sleep 1500
```

The "+" symbol indicates the last job which was stopped or executed in the background, and the "-" symbol indicates the previous job executed in background.

```
wait        -     Wait  for the specified process to end
[user@localhost ~]# wait <process ID>
```

Examples for common uses of "**wait**" command:

```
[user@localhost ~]# sleep 200 &      # Run sleep in background
[1] 19741
[user@localhost ~]# sleep 400 &      # Run another sleep in background
[1] 19742
 [user@localhost ~]# wait 19741      # Waiting for the first "sleep"
                                     command to end using its PID.
                                     Afterwards return control to the
                                     shell

[user@localhost ~]# sleep 200 &      # Run sleep in background
[1] 19741
[user@localhost ~]# wait             # Waiting for the last command to
                                     end, afterwards return control to
                                     the shell
```

Note: `wait` command cannot be executed in background or suspended using the Ctrl+Z signal.

In case the user attempts to exit the shell while jobs are stopped, the shell will print a message warning that there are stopped jobs. The `jobs` command may then be used to inspect their status. If a second attempt to exit is made without an intervening command, the shell will not print another warning, and result in:

1. Stopped jobs will be terminated
2. The background running jobs will continue running having init process as their parent process.

```
# Logging out while there are stopped processes
[user@localhost ~]# logout
There are stopped jobs              # Shell message
```

In addition to the process run modes described above, some processes are called “daemon”. Daemon is a process (usually a system service), which runs in background and its parent process is usually the “init” process, since its executing shell was terminated.

These processes are typically represented inside square parentheses [], for example:

```
[user@localhost ~]# ps -eF
UID    PID  PPID C    STIME   TTY    TIME CMD
Root   1    0    0    15:12   ?      00:00:01   init [3]
Root   2    1    0    15:12   ?      00:00:00   [migration/0]    #daemon
```

2.7. Useful basic commands

The following commands, among a long list of shell commands, are basic and commonly used for strings and file manipulation via scripts.

```
head       -     Output the first part of input
[user@localhost ~]# head [file]
```

Common options	Description
-n <N>	Print the first N lines of the input

Example:

```
# Print first 10 lines of “passwd” file
[user@localhost ~]# head -n 10 /etc/passwd
```

```
Tail       -     Output the last part of input
[user@localhost ~]# tail [file]
```

Common options	Description
-f	Keep updating the output as the file grows
-n <N>	Print the last N lines of the input

Example:

```
# Keep printing to output the last lines of the log file "test.log"
[user@localhost ~]# tail -f /tmp/test.log
```

```
sort        -     Sort lines of text input
[user@localhost ~]# sort [file]
```

Common options	Description
-n	Numeric sort
-r	Sort in reverse order
-o FILE	Write the sorted output to FILE

Example:

```
[user@localhost ~]# sort /tmp/test.log
```

A commonly used command for searching patterns in stream is the "grep" command. It supports wide range of properties including extended regular expressions:

```
grep        -     Print line matching defined pattern
[user@localhost ~]# grep [options] <pattern> <file name>
```

Common options	Description
-c	Count number of appearances of pattern in file
-E	Interpret PATTERN as an extended regular expression
-i	Ignore case while searching pattern
-l	Print only file names of those files which pattern was found
-R	Search recursively in directory
-s	Suppress error messages about nonexistent or unreadable files
-v	Invert the sense of matching
--color	Surround the matching string with color

Examples:

```
# Print all lines containing the string "test" from file /tmp/file1
[user@localhost ~]# grep --color "test" /tmp/file1


# Search process named "ssh" from current running processes
[user@localhost ~]# ps -ef | grep ssh


# Combine output redirection and piping.
# In the example below the output of the history command is piped to
the grep command as input, in which the pattern "rm" is searched and
all output is saved under /tmp/ directory:
[user@localhost ~]# history | grep "rm" > /tmp/rm_commands.log
```

The "diff" command compares two files and returns the difference between the two.

```
diff       - Compare two files
[user@localhost ~]# diff [options] <file1> <file2>
```

Common options	Description
-b	Ignore changes in amount of white space
-q	Report only whether the files differ, not the details of the differences
-i	Ignore changes in case
-r	Recursively compare any subdirectories found
-w	Ignore white space when comparing lines
-y	Use the side by side output format

The output of the comparison looks as follows:

```
[user@localhost ~]# diff file1 file2
4c7               # "4" indicates line number on left file (file1) vs.
                  "7" line number compared on right side (file2)
< joe             # line on left side file (file1)
---
> jim             # different line on right side file (file2)
```

Linux shell provides several utilities to help us find the location of files and directories. One of these utilities is "find":

```
find                - Searches for files in a directory hierarchy
[user@localhost ~]# find <path> [options]
```

Common options	Description
-name	The name of the file
-type	Define the type of the file to find (e.g. `f` for file, `d` for directory etc)
-maxdepth <levels>	Set the directory max depth to search in
-mount	Look for files located only in the current filesystem
-user <username>	Look for files owner by specific user
-printf <format>	Print results in specific format

Examples:

```
# Look for files or directories named "file1" starting from "/" path
[user@localhost ~]# find / -name "file1"

# Look under "/etc" for files names starting with "pass" pattern
[user@localhost ~]# find /etc -name "pass*" -type f

# Look for files or directories with name containing "pass", start
searching from "/" path for max directory depth of 2 levels
[user@localhost ~]# find / -name "*pass*" -maxdepth 2

# Look for files or directories with name containing "pass", start
searching from "/" path and limit to the partition of the "/" location
[user@localhost ~]# find / -name "*pass*" -mount

# Look for files or directories starting with "pass" starting from "/"
path and owner by user "joe"
[user@localhost ~]# find / -name "pass*" -user "joe"

# Look under "/" for files or directories names starting with "pass"
pattern and display the output according to the format provided
[user@localhost ~]# find / -name "pass*" -printf "%u:%h%p\n"
```

```
cat         -     Concatenate files and print to the standard output
[user@localhost ~]# cat <file name>
```

Example:

```
[user@localhost ~]# cat /etc/passwd   # Print to terminal the content of
                                      "passwd" file
```

3. Getting started

Writing shell scripts is not hard to learn, the syntax is simple and straightforward, similar to that of invoking and chaining together commands at the command line, and there are only a few "rules" to learn.

Shell scripting hearkens back to the classical UNIX philosophy of breaking complex projects into simpler subtasks, of chaining together components and utilities. Many consider this a better, or at least more esthetically approach to problem solving than using one of the new generation of high powered all-in-one languages, such as Perl, which forces you to alter your thinking processes to fit the tool.

When **not** to use shell scripts:

- ✓ Resource-intensive tasks, especially where speed is a factor (sorting, hashing, etc.)
- ✓ Procedures involving heavy-duty math operations, especially floating point arithmetic, arbitrary precision calculations, or complex numbers (use C++ or FORTRAN instead)
- ✓ Cross-platform portability required (use Java instead)
- ✓ Complex applications, where structured programming is a necessity (needs type checking of variables, function prototypes, etc.)
- ✓ Mission-critical applications
- ✓ Situations where security is important
- ✓ Extensive file operations required (Bash is limited to serial file access, and that only in a particularly clumsy and inefficient line-by-line fashion)
- ✓ Multi-dimensional arrays, or data structures, such as linked lists or trees are required
- ✓ Generating or manipulate graphics or GUIs
- ✓ Directing access to system hardware, port or socket I/O
- ✓ Proprietary, closed-source applications (shell scripts are necessarily Open Source)

If any of the above applies, consider a more powerful scripting language (e.g. Perl, Tcl, Python, or possibly a high-level compiled language such as C, C++, or Java).

3.1. Creating basic shell scripts

A script is nothing more than a list of system commands stored in a file, for example:

```
# Simple script to remove all files located under /tmp directory
echo “Cleaning temporary directory content”
rm -rf /tmp/*
echo “Done.”
```

In order to make the script executable, add “execution” permission to the script file (called “cleaner”) as follows:

```
[user@localhost ~]# chmod ugo+x cleaner
```

This gives the owner, members of the group, and everyone else the ability to execute the file.

Finally, to execute the script simply call it:

```
[user@localhost ~]# bash ./cleaner
Cleaning temporary directory content
Done.
```

Where the “bash” indicates which shell should execute the script “cleaner”.

The advantages of placing commands in a script is for not having to retype them again and again.

3.2. Execution shell

It is possible to define which interpreter will be executing the script by providing its path at the beginning of the script, for example:

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

Where “***#!***” (also called *Shebang*) indicates to the system that this file is a set of commands to be executed by a command interpreter, and the “/bin/bash” indicated afterwards is the interpreter.

Following list is a partial list of commonly supported interpreters:

```
#!/bin/sh
#!/bin/bash
#!/bin/bash
#!/bin/tcsh
#!/bin/csh
#!/usr/bin/perl
#!/usr/bin/tcl
#!/bin/sed -f
#!/usr/awk -f
```

Tip: Using **`#!/bin/sh`**, the default shell in most commercial variants of UNIX, makes the script portable to non-Linux machines, though you may have to sacrifice a few Bash-specific features.

Once the required interpreter is defined at the beginning of the script, it is allowed to run the script same as any other command (eliminating the shell to be interpreting the script), for example:

```
[user@localhost ~]# chmod ugo+x cleaner
[user@localhost ~]# ./cleaner
Cleaning temporary directory content
Done.
```

3.3. Comments

During the development of any code, developer's commenting is highly recommended for maintenance, readability etc.

Like on most programming languages, shell scripting platform also supports comments, using the following syntax:

- ✓ Comments are set by '#' character.
- ✓ Comments can be set at the beginning of a line or middle of text.

```
#!/bin/bash

# This is a comment
ls -l                                   # This is another comment
```

Note: There is no method of terminating the comment; therefore command may not follow comment on the same line.

Comments cannot be set for blocks of lines, like in other programming languages (C, Java etc.)

Yet, take a look at the following examples:

```
#!/bin/bash

echo "The # here does not begin a comment."
echo 'The # here does not begin a comment.'
echo The \# here does not begin a comment.
echo The # here begins a comment.
```

3.4. Exit – Finishing a Script

Every command returns an exit code on termination. The exist code indicates whether the command finished its operation successfully or failed due to some kind of error.

Normally, process which finished its task successfully return exit code 0, while process which failed during its task due to any reason usually return an exit code other than 0 (according to the error set).

The exit code can be determined using the ***$?*** environment variable which is constantly updated, according to the exit code of the last executed command.

Determining the EXIT code of the last program could be done using the following command: "**echo $?**" right after its termination, for example:

```
[user@localhost ~]# sleep 5     # Delay the shell for 5 seconds
[user@localhost ~]# echo $?     # Print the "sleep" exit code
0                               # The exit code was "0" - success
[user@localhost ~]# sleep 5     # Delay the shell for 5 seconds
Ctrl+c                          # Stop the process using Ctrl+C signal
[user@localhost ~]# echo $?     # Print the "sleep" exit code
130                             # The exit code was "130" - failure
```

Script finishes execution with the exit code set by the last command executed in the script**.**

```
#!/bin/bash

cp /etc/my.cnf /tmp
rm -rf /tmp/*
# Since no EXIT code is defined, the script's exit code will be
according to the "rm" command exist code
```

When a script ends with an **exit** that has no parameter, the exit code of the script is the exit code of the last command executed in the script**.**

```
#!/bin/bash

cp /etc/my.cnf /tmp
rm -rf /tmp/*
exit
# Since no parameter was provided to the "exit" command, the script's
exit code will be according to the "rm" command exist code
```

It is possible to explicitly set exit code using the `exit` command as follows:

```
#!/bin/bash

cp /etc/my.cnf /tmp
rm -rf /tmp/*
exit 1
```

Similar to processes, the exit code of the script can be determined from the shell using the $? environment variable via the following command: "**echo $?**" right after its termination.

3.5. Printing to output

Linux has several utilities to print to the standard output text, expressions or variables. Two common utilities are the `echo` and `printf`, which are stand alone commands providing a wide range of capabilities.

3.5.1. Using echo

Prints text, expression and variables to the standard output:

```
echo        -     Display a line of text to standard output
[user@localhost ~]# echo [options] [strings]
```

Common options	Description
-n	Do not output trailing newline (\n)
-e	Enable interpretation of the backslash-escaped characters

Examples for common uses of the "**echo**" command:

```
#!/bin/bash

echo "hello world"  # Output: hello world
echo $USER          # Output: root
echo -n "test"      # Output: "test" w/o new line
```

Using `echo` command with "-e" option supports special backslash escape sequences which are interpolated as follows:

Option	Description
\a	Bell
\b	Backspace
\c	Suppress trailing newline
\f	Form feed
\n	New line
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab

Examples:

```
#!/bin/bash

# Print while controlling the output display
echo -e "First line\n\tSecond line with tab"
```

Output:

```
First line
        Second line with tab
```

```
# The following example will print the "123" string and will delete the
last character "3" due to the "\b" which is interpolated as backspace.

echo -en "123\b"  # Output: 12
```

3.5.2. Using ***printf***

The ***printf*** command prints text, expression and variables to the standard output using "c" style formatted printing syntax:

```
printf     -     Format and print data to standard output
[user@localhost ~]# printf <format> [arguments]
```

The ***printf*** command supports the following formatting controls:

Option	Description
\\	Backslash
\a	Alert (bell)

Option	Description
\b	Backspace
\c	Produce no further output
\f	Form feed
\n	New line
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab

Examples for common uses of "***printf***" command:

```
#!/bin/bash

param1=”125”
param2=”Success: the output is OK”

# Print the values of $param1 and $param2 each on its own line
printf "The value is: %s\n\t%s\n" $param1 $param2

#Output:
The value is: 125
    Success: the output is OK
```

4. Variables

Working with variables is divided into two parts: Assignment and Referencing.

Assigning values to the variable is called assignment/substitution, and getting its value is called value referencing.

4.1. Variable Naming

The name of a variable, which is a placeholder for its value, is **case sensitive** and must begin with alphanumeric character or underscore character (_), followed by one or more alphanumeric character or underscore, for example:

Variable assignment	Status
abc=123	Valid
_abc=123	Valid
ab1c=123	Valid
ab_c=123	Valid
1abc=123	Not valid !
@abc=123	Not valid !
ab-c=123	Not valid !

Variable name case sensitive example:

```
platform=linux     # Two different variables due to their
Platform=linux     # case difference
```

4.2. Variable Substitution

Variable assignment is usually (except assignment in loop commands) substituted using the following syntax:

```
<variable name>=<variable value>
```

Pay attention: no space permitted on either side of '=' sign when assigning value to variable:

```
# Correct assignment
VAR1=123
VAR1=               # Null/empty value assignment

# Incorrect assignment
VAR1 =123
VAR1= 123
```

In case the value of the variable contains white spaces, the assignment must be enclosed with double quotes:

```
# Correct assignment
VAR1=”My Name”

# Incorrect assignment
VAR1=My Name
```

Referring to the variable’s value is done by using the variable’s name with addition of the “$” character as a prefix of its name as follows: **$<variable name>**, or example:

```
# Variable assignment
VAR1=”123”

# Value referencing
echo $VAR1          # Will print the variable’s value: 123
```

Tip: Variable’s scope is within the whole script from the point the variable was declared**.**

In addition to static values substitution, assignment can be for dynamic values as well:

- ✓ Substituting values using other pre-defined variables
- ✓ Substituting the output of command execution

4.2.1. Substituting values using other pre-defined variables

In order to use previously defined variables during assignment, use the name of the variable with the “$” prefix as follows:

```
A=123
```

Simple usage of pre-defined variable substitution:

```
B=”$A”
echo $B             # Output: 123
```

Enclosing a referenced value in double quotes (" ") does not interfere with variable substitution. This is called partial quoting, sometimes referred to as "weak quoting". Using single quotes (' ') causes the variable name to be used literally, and no substitution will take place. This is full quoting, sometimes referred to as "strong quoting".

Substitution of special characters ($, \, `, !, etc.) as literal values can be done as follows:

1. Enclose the value with single quotes.
2. Enclosing the value with double quotes, use the backslash (\) as prefix of the special character.

Examples:

```
A=123
B="\$A"             # Use backslash before special characters
echo $B             # Output: $A
B='$A'              # Use single quote during assignment
echo $B             # Output: $A
```

Concatenation of a string to a value of a predefined variable should be done by enclosing the variable's name with curly brackets {}, for example:

```
B="$A45"
echo $B             # will print nothing since A45 not defined

B="${A}45"
echo $B             # Output: 12345
```

4.2.2. Substituting the output of command execution

Variable substitution allows assigning output of commands during variable assignment to the value of the variable by enclosing the command with " ` " (apostrophe), for example:

```
A=`cat /etc/passwd`             # "A" will contain the content of the
                                /etc/passwd file

echo $A                         # Output: Content of /etc/passwd file

A=`grep ^root /etc/passwd`      # "A" will contain only lines starting
                                with the word root from the /etc/passwd
                                file
```

4.3. Variable Referencing

As mentioned before, referring to the variable's value is done by the variable's name and using the "$" character as a prefix to its name as follows: **$<variable name>.**

To prevent error cases, a good practice will be to refer to the variable name enclosed by double quotes "", for example:

```
# VAR1 assignment
VAR1=”/var/log/log file”      # File name containing white spaces

# Reference error case: the interpreter interprets following line as
follows: ls -l My Documents, and will generate error since the files
“/var/log/log” and “file” are missing
ls -l  $VAR2

# The interpreter interprets following line as follows:
# ls -l “/var/log/log file” and will succeed
ls -l  “$VAR2”
```

The only time a variable appears "naked", without the $ prefix, is when declared or assigned, when *unset* or when exported.

Tip: Note that `$variable` is actually a simplified alternate form of `${variable}`. In contexts where the `$variable` syntax causes an error, the longer form may work (i.e. in cases the content of the variable contains white spaces, etc.).

Sometimes it is needed to concatenate the variable’s value to a string when referencing its value.

In order to indicate to the shell which part of the string is variable and which is not, enclose the variable’s name with curly brackets {}, for example:

```
# Generate file with name convention containing current logged-in
“username_log”:
touch /tmp/$UID_log           # The interpreter will fail since is not
                              familiar with variable “UID_log”
touch /tmp/${UID}_log         # Will create file: /tmp/root_log
```

In order to unset a declared variable use the “unset” command with the variable’s name (**without** using $ as prefix), for example:

```
A=123
echo $A             # Will print: “123”
unset A
echo $A             # Will print nothing
```

4.4. Variable Types

Unlike many other programming languages, BASH does not segregate its variables by "type". Essentially, BASH variables are character strings, but, depending on context, BASH permits integer operations and comparisons on variables. The determining factor is whether the value of a variable contains only digits or not.

Variables visible only within a code block or function. In spite of that, variables can be exported using the “export” command, and will be accessible to the script’s child processes, that is, only to commands or processes which that particular script initiates. A script invoked from the command line *cannot* export variables back to the command line environment.

4.5. Special Variables

In addition to the environment variables which can be used by the script during its execution, there is an additional list of built-in variables transferred to the script on execution, containing useful information which might be used during its execution, as follows:

Variable Name	Variable Description
$$	Script’s Process ID
$PPID	Script’s parent Process ID
$_	Last argument of previous command
$?	Exit status
$!	PID of last executed background command
$-	Options given to the shell
$RANDOM	A random number generator in range 0-32767

Some examples of variable assignment and substitution examples:

```
# Setting empty variable (or '' for empty set)
PACKED_JARS=""
PACKED_JARS=

# Using special variables
PID=$$
LAST_EXEC_RESULT=$?
ARGC=$#
ARGV=$@
```

```
FIRST_ARGUMENT=$1

# Setting array
ARRAY[0]=one
ARRAY[1]=two
ARRAY[2]=three
echo "Value is: ${ARRAY[0]}" # Output: Value is: one

# Variables usage
file="/test/abc"    # Define variable “file”
echo $filetxt       # Output: variable “filetxt” unknown
echo ${file}.txt    # Output: /test/abc.txt

# Initialize variable dir_list using output of command
dir_list=`ls -l /tmp 2>/dev/null`
```

5. Parameters

5.1. Positional parameters

To handle options on the command line, we use a facility in the shell called *positional parameters*. Positional parameters are a series of special variables that contain the contents of the command line.

For example, using the following command line:

```
[user@localhost ~]# rm -r -f /tmp/*
```

The “rm” command receives from the shell the following list of parameters provided to it via the command line:

- -r
- -f
- /tmp/*

Same mechanism is supported by shell scripts, when required to write more complicated scripts which should receive inputs from the user and execute accordingly.

The positional parameters variables are accessible using the following naming convention:

Variable Name	Variable Description
$*	List of arguments passed to the script as one string
$#	Number of arguments passed to the script
$@	List of arguments passed to the script each as a quoted string
$1...$N	Arguments passed to the script from the command line ($0 is the name of the script, $1 is the first argument etc.)

Example:

```
[user@localhost ~]# myscript.sh -t -s /var/log/messages
```

The parameters are stored at the following variables:

Variable Name	Variable Value
$*	myscript.sh -t -s /var/log/messages
$#	3
$@	myscript.sh -t -s /var/log/messages
$0	myscript.sh

Variable Name	Variable Value
$1	-t
$2	-s
$3	/var/log/messages

Here is a tiny shell script to try this out:

```
#!/bin/bash

echo "Positional Parameters"
echo '$0 = ' $0
echo '$1 = ' $1
echo '$2 = ' $2
echo '$3 = ' $3
echo '$# = ' $#
echo '$@ = ' $@
echo '$* = ' $*
```

Please refer to chapter 7, for using positional parameters together with conditional statements and loops.

5.2. Using shift

Parsing command line arguments provided to the script can be done using the following methods:

1. Go through all parameters starting to $1 to $N and interact accordingly.
2. Work with $1 parameter only, and once considered, move the next parameter ($2) into $1 place until no parameters are left.

In order to work using the second method, use the "***shift***" command.

The ***shift*** command moves the current values stored in the positional parameters (command line args) to the left (by default one position):

$1 <--- $2, $2 <--- $3, $3 <--- $4, etc.

The old `$1` is overwritten with $2, while $2 is overwritten with $3 and so on, for example, refer to the following script named "myscript.sh":

```
#!/bin/bash

echo "Positional Parameters"
echo '$1 = ' $1
```

```
shift
echo '$1 = ' $1
shift
echo '$1 = ' $1
```

Execution of the script will produce the following output:

```
[user@localhost ~]# myscript.sh -param1 -param2 -param3
$1 = -param1
$1 = -param2
$1 = -param3
```

The value of *$0 (the script name) does not change*.

Shift can be used with integer values to shift more than one place, i.e. "shift 2" will shift 2 places meaning the new value of $1 will be the old value of $3 and so forth:

```
#!/bin/bash

echo "Positional Parameters"
echo '$1 = ' $1
shift 2
echo '$1 = ' $1
shift
echo '$1 = ' $1
```

Execution of the script will produce the following output:

```
[user@localhost ~]# myscript.sh -param1 -param2 -param3
$1 = -param1
$1 = -param3
$1 =                # Empty output since no positional parameters exist
```

The ***shift*** command works in a similar fashion on parameters passed to a function. Please refer to chapter 9 for more information about shell functions.

5.3. The **getopts** command

The ***getopts*** command parses command-line options provided to the script. It permits passing and concatenating multiple options and associated arguments to a script (for example **./scriptname -abc -e /usr/local**).

getopts syntax looks as follows:

```
getopts optstring op [arg ...]
```

A colon following the option name in the declaration indicates that the option expects an argument, for example:

```
getopts ab:cd:e option
```

The ***getopts*** construct uses ***op*** to store the option currently processed, and two more implicit variables: OPTIND and OPTARG as follows:

1. *$OPTARG* - (OPTion ARGument) contains the argument attached to the option.
2. *$OPTIND* - (OPTion INDex) contains the index of the argument will be processed next.

Usage requirements: The arguments passed from the command-line to the script must be preceded by a dash (−). It is the prefixed − that lets **getopts** recognize command-line arguments as *options*.

Usage example:

```
getopts ab:cd:e cli_opt
```

Means, options **b** and **d** require argument while **a c** and **e** do not, in other words, the following command-line is expected:

```
-a -b <argument> -c -d <argument> -e
```

Following shorter command-line is supported as well, since both **a c** and **e** do not require argument to be provided:

```
-acb <argument> -ed <argument>
```

In case provided command-line does not fit the **opstring** syntax, an error message will be provided to the terminal, unless is silenced by leading the **opstring** with ":", for example:

```
getopts :ab:cd:e cli_opt
```

At the example above, no error message will be prompted once user provides unknown option via command-line, due to the leading ":".

Example:

```
# The following script expect command line of the form:
#./script.sh -a -b <arg> -c -d <arg>
#!/bin/bash
```

```
# The ":' means that the both 'b' and 'd' options expect argument
while getopts "ab:cd:e" op
do
     case "$op" in
          a) echo "Option \"a\"";;
          b) echo "Option \"b\" $OPTARG" ;;
          c) echo "Option \"c\"";;
          d) echo "Option \"d\" $OPTARG" ;;
          *) break;;
     esac
done
```

```
# Script execution output
[user@localhost ~]$ ./script.sh -ab 12 -d 129
Option "a"
Option "b" 12
Option "d" 129
```

6. Decision Making

A basic requirement for programming language is the ability to test for a condition, then act according to its result.

Linux BASH shell provides the following testing mechanism methods:

- ✓ **bracket** and **parenthesis** testing operators
- ✓ ***if/then*** construct
- ✓ ***test*** command
- ✓ ***case*** command

6.1. Brackets and Parenthesis

In Linux shell, ***brackets*** and ***parenthesis*** are built-in commands, acting either as conditional or grouping constructs.

Conditional constructs

Parentheses	Description	Example
[[expression]] [expression]	Test expression & return execution status (0 or 1) depending on the evaluation of the conditional expression	[[$a < 200]] [$a -gt 200]
((expression))	Arithmetic evaluation. If the value of the expression is non-zero, the return status is 0; otherwise the return status is 1.	((RES=10+14)) echo $RES

Grouping constructs

Parentheses	Description	Example
{ commands list; }	Grouping commands to be executed within the **current** shell environment	{ echo 1; echo 2; }
(list)	Grouping commands to be executed within a **separated** shell environment	(ls –l ; rm -f /tmp*)

6.2. If/else/then/fi construct

The ***if*** construct is one of the most important features of many programming languages, shell scripting included. It allows for conditional execution of code fragments.

The expression of the ***If*** construct is evaluated to its Boolean value; in case of ***TRUE*** - the construct flow is directed to "***then***" block, otherwise directed to the "***else***" block (if "***else***" block defined).

Basic ***if/then/fi*** command syntax looks as follows:

```
if test statement
then                # Block area for test statement success
...
fi                  # closing the if construct
```

For more testing options using ***if/then/else/fi*** command use following syntax:

```
if test statement
then                # Block area for test statement success
...
else                # If test statement fails, execute Else block
...
fi                  # closing the if construct
```

For all testing options using ***if/then/else/fi*** command use following syntax:

```
if test statement
then                # Block area for test statement success
...
elif test statement # Else run additional test statement block
then                # Block area for test statement success
...
else                # Else block start (optional)
...
fi                  # closing the if construct
```

Tip: The *if* and the *then* can reside on same line using the ";" separator as follows:

```
if test statement ; then
```

If statement supports a shorten syntax, eliminating the "***if***" "***then***" "***else***" strings as follows:

```
# Short IF constructs
# ECHO will be executed only if the tested expression is TRUE
[ "$1" = "test" ] && echo OK
```

```
# ECHO will be executed only if the tested expression is FALSE
[ "$1" = "test" ] || echo OK

# ECHO TRUE will be executed if the tested expression is TRUE otherwise
ECHO FALSE will be executed
[ "$1" = "test" ] && echo TRUE || echo FALSE
```

The test conditions can be used with variant types of testing operators:

- ✓ Arithmetic test operators
- ✓ File test operators
- ✓ String comparison operators

Even though the shell does not explicitly support variable typesetting, i.e. string, Boolean, integer etc., yet the content of the variables will be implicitly assumed according to the operator used. For more information please refer to the next chapters.

6.2.1. Arithmetic test operators

Test statements support *arithmetic* comparison operators to compare two variables or quantities.

The following table lists the supported arithmetic test operators:

Condition	Description	Parentheses
-gt	Greater than	[[test]] OR [test]
-lt	Less than	
-le	Less or Equal	
-eq	Equal	
-ne	Not equal	
>	Greater than	((test))
<	Less than	
>=	Greater or Equal	
<=	Less or Equal	
==	Equal	
!=	Not Equal	

Note: Arithmetic and String comparison use a different set of operators.

Examples:

```
# Binary tests using []
A=5
if [ $A -lt 2 ] ; then
     echo "\$A is smaller than 2"
elif [ $A -gt 2 ] ; then
     echo "\$A is greater than 2"
else
     echo "\$A equals 2"
fi
```

```
# Binary tests using (( ))
A=5
if (( $A < 2 )) ; then
     echo "\$A is smaller than 2"
elif (( $A > 2 )) ; then
     echo "\$A is greater than 2"
else
     echo "\$A equals 2"
fi
```

Tip: Generally in shell, it is a good practice to place the variable within double quotes:

```
if (( "$A" < 2 )) ; then
```

6.2.2. String test operators

Test statements support *string* comparison operators to compare two variables or strings.

The following table lists the supported string test operators:

Cond.	Description	Parentheses
-z	String length is zero (null)	[[test]] OR
-n	String is not null	

Cond.	Description	Parentheses
= ==	Equal	[test]
!= <>	Not equal	
>	Is less than, in ASCII alphabetical order	[[test]]
<	Is greater than, in ASCII alphabetical order	

Examples:

```
# A is initialized with a capital "H". String comparison is evaluated
A="Hello"
if [ $A = "hello" ] ; then
     echo "\$A equals hello"
elif [ -z $A ] ; then
     echo "\$A is empty"
elif [ -n $A ] ; then
     echo "\$A contains $A"
fi

# B is not initialized (null) hence the else block will be executed
B=''
if [ $B ] ; then
     echo $B
else
     echo "\$B is empty"
fi

# A & B are defined with integer values; yet the string comparison will
consider them as strings (casting) rather than integers
A=3
B=4
if [ $A != $B ] ; then
     echo "\$A does not equal \$B"
else
     echo "\$A equals \$B"
fi

# Here A & B will be compare as two strings rather than integers since
the tests uses > within square parentheses
```

```
A=3
B=4
if [[ $A > $B ]] ; then      # The > mark will verify if ASCII code of
                             3 is greater than ASCII code of 4 instead
                             of integer comparison
     echo "\$A in ASCII is greater than \$B"
fi
```

6.2.3. File test operators

Test statements support *file* testing operators to verify file attributes. The return value of the test is either "***True***" or "***False***".

The following table lists the common supported *file* test operators:

Operand	Description
-a <file>	True if file exists
-d <file>	True if file exists and is a directory
-e <file>	True if file exists
-f <file>	True if file exists and is a regular file
-p <file>	True if file exists and is a named pipe (FIFO)
-r <file>	True if file exists and is readable
-s <file>	True if file exists and has a size greater than zero
-w <file>	True if file exists and is writable
-x <file>	True if file exists and is executable
-L <file>	True if file exists and is a symbolic link
-S <file>	True if file exists and is a socket

For more supported ***file*** testing operators, refer to ***man bash***.

Examples:

```
# Check whether the file /usr/bin/sum exists and executable
if [ -x /usr/bin/sum] ; then
     echo "File /usr/bin/sum exists and executable"
elif [ -a /usr/bin/sum ] ; then
     echo "File /usr/bin/sum exists but not executable"
else
     echo "File /usr/bin/sum does not exist"
fi
```

6.2.4. Nested decisions

Sometimes it is needed to continue with condition testing once the first condition has been verified. For these situations, nested tests are supported by the shell ***if/then*** construct.

Nested ***if/then/fi*** command syntax looks as follows:

```
if test statement
then                          # Block area for test statement success
   if test statement
        then                  # Nested Block area for test statement
             ...
   fi                         # closing the nested if construct
fi                            # closing the if construct
```

For more nested testing options using ***if/then/else/fi*** command use following syntax:

```
if test statement
then                          # Block area for test statement success
   if test statement
   then                       # Nested Block area for test statement
        ...
   else
        ...
   fi                         # closing the nested if construct
else                          # Else block start
   if test statement
   then                       # Nested Block area for test statement
        ...
   else
        ...
   fi                         # closing the nested if construct
fi                            # closing the if construct
```

Examples:

```
# Check whether the file /usr/bin/sum exists and executable using
nested statements
if [ -a /usr/bin/sum] ; then
     echo "File /usr/bin/sum exists"
     if [ -d /usr/bin/sum ] ; then
```

```
          echo "File /usr/bin/sum is a directory"
     else
          if [ -x /usr/bin/sum ] ; then
               echo "File /usr/bin/sum is executable"
          fi
     fi
else
     echo "File /usr/bin/sum does not exist"
fi
```

Nested ***if*** construct can be avoided by using compound comparison syntax.

The following table lists the supported compound operators:

Op	Description	Parentheses
-a	Logical AND	[test1 Op test2]
-o	Logical OR	
&&	Logical AND	[[test1 Op test2]]
||	Logical OR	
!	Logical NOT	Either

Examples:

```
# Check whether the file /usr/bin/sum exists and executable using
compound comparison
if [ -a /usr/bin/sum -a -x /usr/bin/sum ] ; then
     echo "File /usr/bin/sum exists and is executable"
fi

# Using &&
if [[ -a /usr/bin/sum && -x /usr/bin/sum ]] ; then
     echo "File /usr/bin/sum exists and is executable"
fi

# Divided parentheses
if [ -a /usr/bin/sum ] && [ -x /usr/bin/sum ] ; then
     echo "File /usr/bin/sum exists and is executable"
fi
# File /usr/bin/sum exists but NOT executable
```

```
if [ -a /usr/bin/sum ] && [ ! -x /usr/bin/sum ] ; then
     echo "File /usr/bin/sum exists and is NOT executable"
fi
```

```
# Using more than one condition
if [ $# -lt 2 –o $# -gt 3 ] ; then … ; fi

# Mathematical arguments comparison including logical OR/AND
if [ $a -lt $b –o $a -gt $b –a $a -eq $b ]; then ; fi
```

6.3. Using test statement

The ***test*** command is a Bash built-in command which evaluates the conditional expression and returns status 0 or 1 accordingly.

Tip: Do not confuse with the */usr/bin/test* binary, which checks file types and compare values, and not used during shell scripts (unless explicitly defined).

```
# "test" command syntax
test expr
```

Expressions may be combined using the following operators, listed in decreasing order of precedence:

Expression	Description
! expr	True if expr is false
(expr)	Returns the value of expr. This may be used to override the normal precedence of operators
expr1 -a expr2	True if both expr1 and expr2 are true
expr1 -o expr2	True if either expr1 or expr2 is true

Examples:

```
# Test arithmetic comparison
A=3
B=4
test $A -gt $B && echo True || echo false
```

```
# Test file properties
test -d "$HOME" ;echo $?
```

6.4. The case statement

The ***case*** construct is the shell scripting analog to *switch* in C/C++. It permits branching to one of a number of code blocks, depending on condition tests. It serves as a kind of shorthand for multiple ***if/then/else*** statements and is an appropriate tool for creating menus.

case statement syntax

```
case "$variable" in
     "$condition1")            # Condition1 definition
          commands...          # Condition1 commands block
     ;;                        # Ending condition block
     "$condition2")
          commands...
     ;;
     *)                        # Default option in case none of the
                               conditions matches
          commands...
     ;;
esac
```

Example:

```
read answer
case $answer in
     [yY] | [yY][eE][sS] )
          echo "User has agreed!"
          agreed=yes
          ;;
     [nN] | [nN][oO] )
          echo "Goodbye..."
          exit 1
          ;;
     *) printf "Please enter \"yes\" or \"no\"."
          ;;
esac
```

7. Loops

Loop is one of the most important features of many programming languages, shell scripting included. It allows iteration of block of code as long as the loop control condition is true.

7.1. The while, until & for loops

There are several loop constructs supported by the BASH shell scripting language:

- ✓ ***For*** loops
- ✓ ***While*** loops
- ✓ ***Until*** loops

7.1.1. **For** Loop

For loop is the basic looping construct. The ***for*** loop construct syntax looks as follows:

```
for arg in [list]
do
     command(s)...
done
```

- ✓ During each pass through the loop, ***arg*** takes on the value of each successive variable in the ***list***.
- ✓ The argument ***list*** may contain wild cards.
- ✓ If ***do*** is on same line as ***for***, there needs to be a semicolon after ***list***.

```
for arg in [list] ; do
```

Examples:

```
# Following loop will run the "echo" command 4 times since the list a b
c d contains 4 values
#!/bin/bash

for i in a b c d
do
     echo $i
done
```

Output:

```
a
b
```

```
c
d
# Following loop will run the "echo" command once since the list "a b c
d" is referred as list with single value due to the quotes
#!/bin/bash

for i in "a b c d"
do
     echo $i
done
```

Output:

```
a b c d
```

```
# Following loop will run the "echo" command 4 times since the list is
dynamically generated by executing command `seq 1 4`
#!/bin/bash

for i in `seq 1 4`; do
     echo $i
done
```

Output:

```
1
2
3
4
```

```
# Following loop will run the "echo" command 4 times since the list is
made of the variable LIST which contains "a b c d"
#!/bin/bash

LIST="a b c d"
for i in $LIST; do
     echo $i
done
```

Output:

```
a
b
c
d
```

```
# Following loop will run the "echo" command once since the list is
made of the variable "$LIST" (inside quotes) which contains "a b c d"
#!/bin/bash

LIST="a b c d"
for i in "$LIST"; do
     echo $i
done
```

Output:

```
a b c d
```

```
# C style for loop syntax
#!/bin/bash

for (( i=1 ; i<=4 ; i++ )) ; do
     echo $i
done
```

Output:

```
1
2
3
4
```

Piping output of ***for*** loop to another command:

```
# The output of the for loop is piped to the sort command
#!/bin/bash

LIST="d b a c"
for i in $LIST; do
     echo $i
done | sort
```

Output:

```
a
b
c
d
```

7.1.2. **`While`** Loop

`While` loop construct tests for a condition at the top of a loop, and keeps looping as long as that condition is true (returns a 0 exit status).

The ***`while`*** loop is useful in situations where the number of loop repetitions is not known beforehand.

The syntax of the ***`while`*** loop construct is:

```
while [ condition ]
do
     command(s)...
done
```

The bracket construct in a ***`while`*** loop is the test brackets, same as used in an ***`if/then`*** test.

Same as with ***`for`*** loops, placing the ***`do`*** on the same line as the condition test requires a semicolon.

```
while [ condition ] ; do
```

Tip: the *test brackets* are *not* mandatory in a *while* loop

Examples:

```
# Run the while loop forever since the condition (true) is always TRUE.
# The condition is set within () which means: execute command true
#!/bin/bash

while ( true ) ; do
     echo "Line X"
     echo "Line Y"
done
```

```
# Run the while loop until count reaches value 10
#!/bin/bash

count=0
while [[ $count -lt 10 ]] ; do
     echo $count
     ((count=count+1))
done
```

```
# Printing file's lines one by the other, omitting the while test
brackets
#!/bin/bash

i=0;
cat /tmp/file | while read file_lines
do
     echo "Line $i: $file_lines"
     ((i+=1))
done
```

7.1.3. `Until` Loops

`Until` loop construct tests for a condition at the top of a loop, and keeps looping as long as that condition is **false** (opposite of the ***`while`*** loop).

The ***`until`*** loop is useful in situations where the number of loop repetitions is not known beforehand.

The syntax of the ***`until`*** loop construct is:

```
until [ condition-is-true ]
do
     command(s)...
done
```

The bracket construct in an ***`until`*** loop is the test brackets, same as used in an ***`if/then`*** test.

Same as with ***`for`*** loops, placing the ***`do`*** on the same line as the condition test requires a semicolon.

```
until [ condition ] ; do
```

Tip: the *test brackets* are *not* mandatory in an *until* loop

Examples:

```
# Printing file's lines one by the other, omitting the while test
brackets
#!/bin/bash

max=10
i=0
```

```
until (( $i > $max ))
do
     echo "Still in loop"
     (( i++ ))
done
```

7.2. Break and Continue

The ***break*** command terminates the loop (breaks out of it) regardless to the condition of the loop.

The ***continue*** causes a jump to the next iteration of the loop, skipping all the remaining commands in that particular loop cycle.

Examples:

```
# Using BREAK
# Print all numbers from 1 to 49
#!/bin/bash

i=0
while ( true ) ; do
     ((i++))
     if [[ `echo $i % 50 | bc` -eq 0 ]] ; then
          break
     fi
     echo $i
done
```

```
# Using CONTINUE
# Endless loop, prints all even numbers by checking if the number is
divided by 2 with no rest
#!/bin/bash
i=0
while ( true ) ; do
     ((i++))
     if [[ `echo $i % 2 | bc` -eq 0 ]] ; then
          continue
     fi
     echo $i
done
```

Like with ***if*** construct, loop constructs also support nested loops.

Using ***continue*** command in a nested loop, the ***continue*** command supports arguments which define the level of loop to jump to.

To clarify follow the next example:

```
# Using CONTINUE to jump to a higher loop level
#!/bin/bash

while ( true ) ; do           # Loop level 1
     i=0
     while ( true ) ; do      # Loop level 2
           ((i++))
           if [[ `echo $i % 50 | bc` -eq 0 ]] ; then
                 continue 2
                 # Continue 2 indicates to continue to the upper loop
                 (loop level 1) rather than with current loop level
                 (loop level 2)
           fi
           echo $i
     done
done
```

8. Arithmetic operations

8.1. Declaring integer variables

Though we have mentioned before that shell variables do not have type set, it is not totally accurate.

Shell supports typesetting only to integer types variables. Other types, like float, double, strings and so on have no explicit declaration support.

In order to set variable to be of type integer use the ***typeset*** command:

```
typeset -i name[=value]
```

In order to set variable as integer use the "-i" option as follows:

```
typeset -i var1
```

OR

```
typeset -i var1=100
```

In case of mistakenly assigning any string into $var1, its value will be "0".

Example:

```
#!/bin/bash

# Setting A to be integer
typeset -i A

# Assigning A with integer value
A=100
echo $A                         # Output: 100

# Assigning A with string
A="test"
echo $A                         # Output: 0
```

8.2. Arithmetic operators

There are several ways to evaluate arithmetic operation using different utilities and syntax. The common ways to evaluate arithmetic operations are:

1. Using the built-in shell command: (( )) and ***let***.
2. Using external utilities: ***expr*** and ***bc***.

8.2.1. Using shell built-in commands

The following arithmetic operators are supported by the shell's built-in commands:

Operator	Description
+	Addition - Adds values on either side of the operator
-	Subtraction - Subtracts right hand operand from left hand operand
*	Multiplication - Multiplies values on either side of the operator
/	Division - Divides left hand operand by right hand operand
%	Modulus - Divides left hand operand by right hand operand and returns remainder
**	Exponentiation
+=	Increment variable by a constant
-=	Decrement variable by a constant
*=	Multiply variable by a constant
/=	Divide variable by a constant
%=	*Remainder* of dividing variable by a constant

The *(( ))* command

The shell supports a build-in command for evaluating arithmetic operations by placing the arithmetic operation inside double brackets as follows:

```
A=100
B=3

((C=$A+$B))                    # C=103
((A=$A%$B))                    # A=1
((A*=$B))                      # Short style; A=300
C=$(($A+$B))                   # C=300
```

The *let* command

The ***let*** command carries out *arithmetic* operations on variables. In many cases, it functions as a less complex version of ***expr***.

```
A=100
B=3
```

```
# Arithmetic operations using let command
let C=$A**$B                          # C=1000000
let "C = $A ** $B"                    # Using spaces
let A*=$B                             # A=300
let A=$A%$B                           # A=1
```

8.2.2. Using external utilities

The *expr* command

The ***expr*** command concatenates and evaluates the arguments according to the given operation (arguments must be separated by spaces). Operations may be arithmetic, comparison, string, or logical.

The following arithmetic operators are supported by the ***expr*** command:

Operator	Description
+	Addition - Adds values on either side of the operator
-	Subtraction - Subtracts right hand operand from left hand operand
*	Multiplication - Multiplies values on either side of the operator
/	Division - Divides left hand operand by right hand operand
%	Modulus - Divides left hand operand by right hand operand and returns remainder
>	Logical test: bigger than (returns 1 on true, 0 on false)
<	Logical test: less than (returns 1 on true, 0 on false)
>=	Logical test: bigger or equals to (returns 1 on true, 0 on false)
<=	Logical test: smaller or equals to (returns 1 on true, 0 on false)
!=	Logical test: not equal (returns 1 on true, 0 on false)
|	Use left argument if it is neither null nor 0, otherwise use right argument
\&	Use left argument if neither arguments are null nor 0, otherwise return 0

Note: some operators must be escaped when used in an arithmetic expression with **expr** due to shell limitations.

Examples:

```
#!/bin/bash
# Arithmetic operations using expr command
A=100
B=3

expr $A \* $B                  # Output: 300
expr 300 + $B                  # Output: 303
A=`expr $A % $B`               # A=1
expr 100 \> 2                  # Output: 1 (true)
expr 100 \< 2                  # Output: 0 (false)
expr 100 \& 0                  # Output: 0 (logical AND)
```

The *bc* utility

The ***bc*** is a command line calculator, supporting both interactive and non-interactive modes. One of the incentives to use ***bc*** is in cases were precise results are required rather than integer ones.

In order to use ***bc*** in a non-interactive mode, ***echo*** the arithmetic operation to it via pipe as follows:

```
echo "<erithmetic expression>" | bc
```

For example:

```
#!/bin/bash

echo "32*2" | bc
```

Script's execution output:

```
64
```

In order to support higher precision results, provide the ***scale*** property to ***bc*** during the ***echo*** command, for example:

```
#!/bin/bash

echo "scale=3 ; 32/7" | bc
```

Script's execution output:

```
4.571
```

8.3. Making arithmetic tests

Please refer to chapter 6.2.1 for detailed information regarding making arithmetic tests.

9. Functions

Similar to other programming languages, shell scripting also supports developing and using functions, though with much more limited implementation.

Function is a code block implementing dedicated operation by aggregating a flow of set of commands.

Functions are best to be implemented when a repetition of code is required, or when a task repeats itself with a slight set of changes.

9.1. Writing functions

The syntax of a function construct looks as follows:

```
function function_name {
     set of commands...
}
```

OR

```
# C style syntax
function_name () {
     set of commands...
}
```

Triggering a function is done by invoking its name. Calling function is equal to calling any other command (i.e. echo, ls etc.).

Note that function must be declared prior invoking it. There is no method of declaring a function (like in other programming language).

Example:

```
# The following script echoes 3 time the string "My name is Emanuel",
and sleeping 1 second between each output
#!/bin/bash

my_name () {
     sleep 1
     echo "My name is Emanuel"
}

count=0
while [[ $count -lt 3 ]]
do
     my_name
     let "count+=1"
done
```

9.2. Passing arguments

Functions support process arguments passed to them and return status on exit to the script for further processing.

In order to pass arguments to a function use the following syntax:

```
function_name [argument list]
```

The function refers to the arguments provided to it the same as a script refers to arguments provided to it via command line - positional arguments, i.e. ***$1 $2 ..$N***. Accordingly, the ***shift*** command can be used to shift the positional argument left side (please refer to chapter 5.2 for more information using ***shift*** command).

Example:

```
# The following script echoes 3 time the string "My name is", each
iteration with: 1) different name 2) different sleep timeout
#!/bin/bash

my_name () {
     # Sleeping timeout according to first argument
     sleep $1
     # Echoing name according to second argument
     echo "My name is $2"
}

count=0

# Initializing array "name" with 3 values
name[0]="Emanuel"
name[1]="Shmuelov"
name[2]="Scripting"

while [[ $count -lt 3 ]]
do
        my_name $count ${name[$count]}
        let "count+=1"
done
```

Output:

```
My name is Emanuel
My name is Shmuelov
My name is Scripting
```

9.2.1. Call by reference

Normally in shell scripting, arguments passed to function by value and not by reference.

Hence, changing the value of the argument will not be kept by the variable once the function finishes.

Nevertheless, there is a workaround in order to implement “call-by-reference” method usually used with regular programming languages.

Please follow the next example for “call-by-reference” implementation:

```
# Call by reference example
#!/bin/bash

function func1()
{
     DATE=”June 06”
     eval "$1=\”$DATE\”"      # Updating the value of BIRTHDAY
     # Since $1 corresponds to BIRTHDAY, the eval command above
     interpreted to: eval “BIRTHDAY=$DATE”
}

func1 BIRTHDAY
echo "My birthday is on $BIRTHDAY"
```

Output:

```
My birthday is on June 06
```

9.3. Function Return Values

Normally in shell scripting, function can return only integer value called “exit status” which implicitly set by the return value of the last command executed in the function. Yet, it is possible to explicitly define different return value using the ***return n*** command, were ***n*** is the integer value to return for example:

```
#!/bin/bash

# Implicitly return (integer) value
func1() {
     ls -l | wc
     # The return value is set according to the result of the last
     command (in this case: wc)
}
```

```
# Explicitly return value
func2() {
     ls -l | wc
     return 2
     # The return value can be only numeric number
}

func1 >/dev/null 2>&1
echo $?                # Output: 0 (according to wc return code)

func2 >/dev/null 2>&1
echo $?                # Output: 2
```

9.4. Setting Local variable

Unlike other programming languages, in BASH, all variables are global unless explicitly declared as local, and then is visible only within the block of code in which it was declared as local.

Note: Variables can be declared as local only inside functions.

Examples:

```
#!/bin/bash

function func1 {
     # Declaring local_var1 as local variable
     local local_var1=10
     # local_var2 is global variable
     local_var2=20
}

func1

echo $local_var1              # Output: none since variable unknown
echo $local_var2              # Output: 20
```

10. Writing interactive scripts

10.1. Prompting & Reading user input

Shell scripting might be interactive by prompting messages and output to the user and getting input from the user.

Prompting messages to the user can be achieved using the ***echo***, ***printf*** and other prompting commands. For more information please refer to chapter 3.5.

On the other hand, reading user input can be achieved using ***read*** command. The ***read*** command reads a line from the standard input and saves it to a variable. The input can be provided either via keyboard or by redirecting output to stdin descriptor using redirection methods.

The ***read*** command supports several features including: prompting message prior to accepting input from the user, limiting input time etc.

In case of using ***read*** without providing the variable name to save the input into, the input will be saved inside a default variable called: REPLY.

The syntax of the ***read*** command looks as follows:

```
read [options] [variable name]
```

Common options	Description
-p [prompt]	Display prompt on standard output
-t [timeout]	Limit the input to “timeout” seconds
-n [nchars]	Stop input after n characters
-s	Silent mode, input does not appear on terminal

Examples:

```
#!/bin/bash

# Reading input from user. Since no variable was explicitly defined,
the input is saved to “REPLY” variable
read

# Reading input from user to “var1” variable
read var1

# Prompting message to terminal and saving input to “fname”
```

```
read -p "Enter first name: " fname
# Prompting message to terminal, limiting the input time to 10 seconds
and saving the input to "lname". If no input was provided within the
time limit, assigning lname with null
read -t 10 -p "Enter last name: " lname


# Prompting message to terminal and saving input to "pass" - the user's
input is not displayed to the terminal
read -s -p "Enter Password: "  pass


# Prompting message to terminal and saving input to "phone". The input
is limited to first 11 characters. The command terminates reading after
the 11th character.
read -n 11 -p "Enter phone number: " phone


# Provide input to "read" by redirecting output of a file as input to
read. Using while construct to keep reading all lines of the file
cat /etc/passwd | while read pw_line ; do
     echo $pw_line
done
```

10.2. Implementing menus with **`select`** command

A powerful tool to build user interactive menus is the ***`select`*** construct. This prompts the user to enter one of the choices presented in the variable list.

The syntax of the ***`select`*** construct is:

```
select variable [in list]
do
     command...
     break
done
```

Note: If the ***`break`*** command will not be provided, the ***`select`*** will keep asking for user's input.

In order to provide prompt to the user either use the ***`echo`*** command with any string or set the ***`PS3`*** variable with the requested string to be prompted to the user.

Example:

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

```
PS3='Choose your favorite car: ' # Sets the prompt string.

echo

select car in "Ferrari" "Porsche" "Mercedes" "Lexus" "BMW"
do
     echo "Your favorite car is $car."
     break
done
```

Output:

```
1) Ferrari
2) Porsche
3) Mercedes
4) Lexus
5) BMW
Choose your favorite car:
```

11. Regular Expressions[1]

11.1. Using regular expressions

A regular expression is a pattern describing a certain amount of text. The shell supports a level of regular expressions within the command line, but not its entire standard.

The power of regular expressions is when it is needed to manipulate some strings from text files, i.e. email address, and to automate the operation.

The tables below provide the syntax rules with corresponding examples at the table afterwards.

Example for utilities support regular expressions are the “grep” and “egrep”. These utilities print lines matching a pattern according to the regular expression provided.

At the following example, the “egrep” will list all lines which the sub-string “root” is located ONLY at the beginning of the line:

```
[user@localhost ~]# egrep "^root" /etc/passwd
root:x:0:0:root:/root:/bin/bash
```

11.2. Metacharacters

Brackets & Positions

Syntax	Description	Example #
[]	Match anything inside the square brackets for one character position once and only once	12,13
-	The - (dash) **inside square brackets** is the 'range separator' and allows us to define a range	12,13
^	Match the beginning of the string	8,10
$	Match the end of the string	8
\b	Match the beginning or end of a word	1-5,7,9,12, 13,15,16,19

Repetitions & Metacharacters

Syntax	Description	Example #
*	Repeat any number of times	2,5,10,15, 19

[1] Reference: http:#www.regular-expressions.info/reference.html

Syntax	Description	Example #
+	Repeat one or more times	6,11,15,18
?	Repeat zero or one time	16-18,20
{*n*}	Repeat *n* times	4,7-8,14
{*n,m*}	Repeat at least *n*, but no more than *m* times	9,14
{*n*,}	Repeat at least *n* times	20
|	Find the left OR right values	12,13
*?	Repeat any number of times, but as few as possible	
+?	Repeat one or more times, but as few as possible	
??	Repeat zero or one time, but as few as possible	
{*n*,*m*}?	Repeat at least *n*, but no more than *m* times, but as few as possible	
{*n*,}?	Repeat at least *n* times, but as few as possible	

Special Characters

Syntax	Description	Example #
.	Match any character except newline	2
\w	Match any alphanumeric character	5,7,9,10, 15-19
\s	Match any whitespace character	15,18
\d	Match any digit	3-4,6,8,14
\<num>	Back reference - match the text that was captured by group number <num>	

Negatives of Special Characters

Syntax	Description	Example #
\W	Match any character that is NOT alphanumeric	
\S	Match any character that is NOT whitespace	11
\D	Match any character that is NOT a digit	
\B	Match a position that is NOT the beginning or end of a word	

Syntax	Description	Example #
[^abc]	Match any character that is NOT one of the characters *abc*	19

Grouping

Syntax	Description	Example #
(exp)	Match exp and capture it in an automatically numbered group	13-18
(?<name>exp)	Match exp and capture it in a group named name	
(?:exp)	Match exp, but do not capture it	
Lookarounds		
(?=exp)	Match any position preceding a suffix exp	16,18
(?<=exp)	Match any position following a prefix exp	17,18
(?!exp)	Match any position after which the suffix exp is not found	20
(?<!exp)	Match any position before which the prefix exp is not found	
(?#comment)	Comment	

Examples:

#	Expressions	Description
1	\b**elvis**\b	Find *elvis* as a whole word
2	\b**elvis**\b.*\b**alive**\b	Find text with "*elvis*" followed by "alive"
3	\b\d\d\d-\d\d\d\d	Find seven-digit phone number
4	\b\d{3}-\d{4}	Find seven-digit phone number a better way
5	\ba\w*\b	Find words that start with the letter a
6	\d+	Find repeated strings of digits
7	\b\w{6}\b	Find six letter words
8	^\d{3}-\d{4}$	Validate a seven-digit phone number from beginning of text
9	\b\w{5,6}\b	Find all five and six letter words
10	^\w*	The first word in the line or in the text

#	Expressions	Description
11	\S+	All strings that do not contain whitespace characters
12	\b05[0|2|5]-[0-9]{3}	Lines containing numbers of type: 050-XXX, 052-XXX, 054-XXX
13	\b((050)|(052)|(054))-[0-9]{3}	Same as above
14	(\d{1,3}\.){3}\d{1,3}	A simple IP address finder
15	\b(\w+)\b\s*\1\b	Find repeated words
16	\b\w+(?=ing\b)	The beginning of words ending with "ing"
17	(?<=\bre)\w+\b	The end of words starting with "re"
18	(?<=\s)\w+(?=\s)	Alphanumeric strings bounded by whitespace
19	\b\w*q[^u]\w*\b	Words with "q" followed by NOT "u"
20	\d{3,}(?!\d)	At least three digits not followed by another digit

12. Text Parsing using *sed* and *awk*

12.1. The *sed* utility

Sed is the ultimate stream editor scripting language; hence you can use it as a command line utility, or aggregate a set of ***sed*** commands into a script file and execute it.

Note: ***sed*** operates on patterns found in the input stream, and when a pattern is matched, the modified output is generated, and the **rest** of the input line is scanned. Therefore, sed is not a recursive language.

Out of all ***sed's*** available commands, the most common commands we will explain are:

- ✓ Stream Substitute
- ✓ Stream Search
- ✓ Stream Delete
- ✓ Stream Append
- ✓ Stream Insert
- ✓ Stream Change

12.1.1. Stream Substitute

The most common used command when using ***sed*** is the substitute command. This command changes all occurrences of a regular expression to a new value.

The substitute is achieved using ***sed's*** 's' command as follows:

```
sed 's/<pattern to find>/<pattern to replace>/' <file name>
```

sed does not replace the content of the file, it only reads its content, manipulates the stream according to the commands and outputs the changes to the standard output. Therefore, in order to save the changes redirect the output to a file.

Example:

```
# Replace all matches of the word "error" with the word "warning" found
in file log.1; The output is redirected to the terminal
[user@localhost ~]# sed 's/error/warning/' log.1


# Replace all matches of the word "error" with the word "warning" found
in file log.1; The output is redirected to file log.2
[user@localhost ~]# sed 's/error/warning/' log.1 > log.2
```

There are four parts to this command:

s	Substitute command
/../../	Delimiter
error	Regular Expression Search Pattern
warning	Replacement string

Tip: The character used as delimiter is not limited to be '/'. i.e. it can be any other character such as '_' or '%' or any other preferred. It is important in cases the pattern to find or to replace with should contain the '/' character, for example:

```
[user@localhost ~]# sed 's_/var/log_/tmp/log_' log.1
```

At the example above, the pattern to find is "/var/log" and the replacement pattern is "/tmp/log", hence it is more convenient to use '_' as delimiters rather than '/'.

Anyhow, according to regular expression rules, special characters can be backslashed as follows:

```
[user@localhost ~]# sed 's/\/var\/log/\/tmp\/log/' log.1
```

Cases in which manipulating pattern using regular expression, where the string itself is not known but needs to be used later-on, can be achieved using the '&' character, representing the found pattern as follows:

```
sed 's/<reg exp>/<text>&<text>/' <file name>
```

Examples:

```
# Stream is searched for the following pattern: beginning with '+',
continuing with a digit and any character afterwards; sed adds the
words "Num is: " as a prefix to the pattern found.
[user@localhost ~]# echo "Phone +972.3.5509988" | sed 's/+[0-9].*/Num
is: &/'
```

Output:

```
Phone Num is: +972.3.5509988
```

```
# Stream is searched for the following pattern: beginning with '+',
continuing with a digit and any character afterwards; the pattern found
is outputted twice.
[user@localhost ~]# echo "Phone +972.3.5509988" | sed 's/+[0-9].*/& &/'
```

Output:

```
Phone +972.3.5509988 +972.3.5509988
```

Substitution supports the following flags to indicate to the "s" command which occurrence to replace:

g - Replace all occurrences

index number - Replace the pattern according to its index number

Example:

```
# Replace all occurrences of 123 with 456
[user@localhost ~]# echo "123 123 123" | sed 's/123/456/g'
456 456 456

# Replace second occurrence of 123 with 456
[user@localhost ~]# echo "123 123 123" | sed 's/123/456/2'
123 456 123

# Replace all occurrences from the second occurrence of 123 with 456
[user@localhost ~]# echo "123 123 123" | sed 's/123/456/2g'
123 456 456
```

12.1.2. Stream Search

sed supports pattern search using the '/' command as follows:

```
sed '/<pattern>/' <file name>
```

Note: ***sed*** does not support stream search as a standalone command. The search command needs to be used by any other commands as a prefix.

Example:

```
# Search for the pattern "emanuel" at /etc/passwd - usage ERROR
[user@localhost ~]# sed '/emanuel/' /etc/passwd
sed: -e expression #1, char 9: missing command

# Search for the pattern "emanuel" and replace the pattern "101" with
"200" at this line only
[user@localhost ~]# sed '/emanuel/ s/101/200/' /etc/passwd
```

12.1.3. Stream Delete

sed supports line deletion of lines containing the searched pattern using the 'd' command as follows:

```
sed '/<pattern>/ d' <file name>
```

Example:

```
# Search for the pattern "root" and delete the line
[user@localhost ~]# sed '/root/ d' /etc/passwd
```

12.1.4. Stream Append

sed supports line appending after lines containing the searched pattern using the 'a' command as follows:

```
sed '/<pattern>/ a<Text>' <file name>
```

Example:

```
# Search for the pattern "emanuel" and add new line with some text
after each line having the pattern
[user@localhost ~]# sed '/emanuel/ aThis is a new line' /etc/passwd


# Search for the pattern "emanuel" and add new empty line after each
line having the pattern
[user@localhost ~]# sed '/emanuel/ a\n' /etc/passwd
```

12.1.5. Stream Insert

sed supports line insertion before lines containing the searched pattern using the 'i' command as follows:

```
sed '/<pattern>/ i<Text>' <file name>
```

Example:

```
# Search for the pattern "emanuel" and add new line with some text
before each line having the pattern
[user@localhost ~]# sed '/emanuel/ iThis is a new line' /etc/passwd


# Search for the pattern "emanuel" and add new empty line before each
line having the pattern
[user@localhost ~]# sed '/emanuel/ i\n' /etc/passwd
```

12.1.6. Stream Change

sed supports changing lines containing the searched pattern using the 'c' command as follows:

```
sed '/<pattern>/ c<Text>' <file name>
```

Example:

```
# Search for the pattern "emanuel" and change the line with new line
with some text
[user@localhost ~]# sed '/emanuel/ cReplacement new line' /etc/passwd
```

12.1.7. Save to File

After ***sed*** finishes to manipulating the stream, the output can be either redirected to a file or can be saved using the 'w' command as follows:

```
# Replace all occurrences of 123 with 456 and save to "res.txt" file
[user@localhost ~]# echo "123 123 123" | sed 's/123/456/gw res.txt'
456 456 456


# Get stream from orig.file, replace all occurrences of 123 with 456
and save to "new.file".
Important: Only the updated lines will be saved to the file
[user@localhost ~]# sed 's/123/456/gw new.file' orig.file
456 456 456
```

12.1.8. Executing Multiple Commands

sed supports executing multiple commands via a single command using the '-e' flag as follows:

```
sed -e <command 1> <command 2> … <command N>
```

Example:

```
# Replace all occurrences of "root" with "ROOT" and "emanuel" with
"EMANUEL"
[user@localhost ~]# sed -e 's/root/ROOT/' -e 's/emanuel/EMANUEL/'
/etc/passwd
```

12.2. The *awk* utility

awk is a data driven programming language supporting regular expressions, designed for processing text based data.

Alfred V. Aho, one of ***awk's*** developers described ***awk***:

"A file is treated as a sequence of records, and by default each line is a record. Each line is broken up into a sequence of fields, so we can think of the first word in a line as the first field, the second word as the second field, and so on. An AWK program is of a sequence of pattern-action statements. AWK reads the input a line at a time. A line is scanned for each pattern in the program, and for each pattern that matches, the associated action is executed."

The organization of ***awk*** programming language is based on the following form:

```
pattern { action }
```

awk, like any other standard programming language, supports conditional statements, loops, stream output, variables and more syntax constructs.

This tutorial will describe only the support of the commonly used built-in variables:

Variable	Name	Description
$1, $2,….$N	Positional Parameters	Variable which holds the fields on the line (according to the separating separator)
FS	Input Field Separator	Variable defines the fields separator when reading input
OFS	Output Field Separator	Variable defines the fields separator when printing output
NF	Number of Fields	Variable which counts the number of parsed field (in the line)
RS	Record Separator	Variable defines the records separator to be used when reading input instead of using the "\n" (newline) as default
ORS	Output Record Separator	Variable defines the records separator to be used when printing output instead of using the "\n" (newline) as default
NR	Number of Records	Variable which counts the number parsed line
FILENAME	The current Filename	Variable which holds the name of file which is parsed

The term **Record** refers to line parsed, while the term **Field** refers to the streams between the separators, for example:

```
# Using the whitespace as the default separator, the following text has
5 records (lines) each has 4 fields
212.23.2.9 - [11/Nov/2010:06:07:25 -0700]
182.3.212.19 - [11/Dec/2010:03:16:03 -0700]
212.23.2.9 - [11/Nov/2010:06:07:25 -0700]
182.3.212.19 - [11/Dec/2010:03:16:03 -0700]
66.13.28.119 - [18/Dec/2010:13:22:21 -0700]
```

Examples:

```
# Following examples are based on the content of the following file
named "log.1":
212.23.2.9 - [11/Nov/2010:06:07:25 -0700]
182.3.212.19 - [11/Dec/2010:03:16:03 -0700]
```

```
# Print only first field (any text until first whitespace) of all lines
since the separator is by default the whitespace (" ")
[user@localhost ~]# awk '{print $1}' /tmp/log.1
212.23.2.9
182.3.212.19
```

```
# Print first & second fields (separated by whitespaces).
[user@localhost ~]# awk '{print $1 $2}' /tmp/log.1
212.23.2.9-
182.3.212.19-
```

```
# Print first & second fields (separated by whitespaces). The ","
indicates to print output record separator which is by default the
white space (" ")
[user@localhost ~]# awk '{print $1, $2}' /tmp/log.1
212.23.2.9 -
182.3.212.19 -
```

```
# Print first & second fields (separated by whitespaces). The ","
indicates to print output field separator which is set to "#" using OFS
variable.
[user@localhost ~]# awk '{OFS="#" ; print $1, $2}' /tmp/log.1
212.23.2.9#-
182.3.212.19#-
```

```
# Print first & second fields using field separator "-" (by setting FS
variable).
[user@localhost ~]# awk '{FS="-" ; OFS="#" ; print $1, $2}' /tmp/log.1
212.23.2.9 # [11/Nov/2010:06:07:25
182.3.212.19 # [11/Dec/2010:03:16:03
```

```
# Print first record and the number of fields found per line using NF
variable.
[user@localhost ~]# awk '{OFS="#" ; print $1, NF}' /tmp/log.1
212.23.2.9#4
182.3.212.19#4
```

```
# Print first and last field.
# Pay attention that this time we use "$NF" instead of "NF", which
indicates to output the value of the last field.
[user@localhost ~]# awk '{FS="-" ; OFS="#" ; print $1, $NF}' /tmp/log.1
```

```
212.23.2.9#0700]
182.3.212.19#0700]
```

```
# Print first record and the number of records using NR variable.
[user@localhost ~]# awk '{OFS="#" ; print $1, NR}' /tmp/log.1
212.23.2.9#1
182.3.212.19#2
```

13. Signaling Scripts

13.1. What are signals?

Signal is an asynchronous notification sent to a process in order to notify it of an event that has occurred. When a signal is sent to a process, the operating system interrupts the process's normal flow of execution. Execution can be interrupted during any non-atomic instruction. If the process has previously registered a **signal handler**, that routine is executed. Otherwise the default signal handler is executed.

Scripts may detect signals once they were sent using kill command, and divert control to a handler function or external program. This is often used to perform clean-up actions before exiting, or restart certain procedures.

Execution resumes where it left off, if the signal handler returns.

13.2. Catching signals with trap

Like a regular programming language, Linux shell supports signal handling once sent to the script. The shell provides a built-in command named trap which catches the signal, and calls the signal handler accordingly.

The syntax of the trap command looks as follows:

```
trap [[arg] sigspec ...]
```

The command ***arg*** is to be read and executed when the shell receives signal(s) ***sigspec***. If ***arg*** is absent (and there is a single ***sigspec***) or -, each specified signal is reset to its original disposition.

Example:

```
# The following script catches signal 2 (SIGINT, Ctrl+C) and 15
(SIGTERM), prints message and exits with code 1
#!/bin/bash

trap '{ echo "Signal catched…" ; exit 1; }' 2 15          # trap signals

while ( true )
do
     echo "Waiting for signal…"
     sleep 1
done
```

The ***arg*** of the ***trap*** command can be an internal function developed by the script, and internal shell command or any external program. At the example above, the ***trap*** command uses the shell's internal "***echo***" command.

14. Script debugging

Shell provides some useful built-in options to help script debugging.

For debugging a BASH script, some parameters might be set to the script interpreter provided at the beginning of the script (i.e. #!/bin/bash) as follows:

Parameter	Description
-x	Echo the command after command line processing
-v	Verbose output
-n	Don't run the script, check syntax only

Some built in variables BASH provides to be used during the script's debugging are:

$**LINENO** – This variable is the line number of the shell script in which this variable appears.

$**SECONDS** – The number of seconds the script has been running

$**FUNCNAME** – The name of the running function

15. Appendix I - Shell commands examples

```
# Trap signals before running some long operation.
# Read 'man 7 signal' for more info on signals.
trap 'rm -f /tmp/1.log; exit 1' HUP INT QUIT TERM
trap function_name INT TERM EXIT

# Do arithmetical calculations
expr $NEW_COUNT - $OLD_COUNT
echo 4^3 | bc
NUM=1
((NUM+=1))            # Value of NUM will be: 2
echo $((NUM+200))     # Will print 202 (NUM will stay 2)

# Pause for some time
sleep 4               # sleep in seconds
usleep 1000           # sleep in microseconds

# Machine's date & date manipulation
date                  # Output: Thu Jan  7 17:23:37 GMT 2010
# Convert some date to seconds since 1970 (Output: 1292834230)
date --date="Sun Dec 20 08:37:10 UTC" +%s
date --date='2009-12-20 08:37:10 UTC' +%s
date --date='2009/12/20 08:37:10 UTC' +%s
# Calculate future date
date -d '2010-01-21 + 2 weeks 3 days'
date -d +2months17days
date -d yesterday-10days

# Using "sed" to Replace characters
sed "s/search pattern/replacement pattern/g" <filename>
# To extract a range of lines, say lines 2 to 4
sed -n 2,4p somefile.txt
# To extract lines 1,2 and 4
sed -n -e 1,2p -e 4p somefile.txt
# Replace character "1" with "2"
echo 1234 | tr "1" "2"
```

Note: sed/awk/grep etc. support extended regular expressions. Please refer the corresponding Appendix for detailed usage of regular expressions

```
# Basic awk parsing command
# Print the first string until space (Output: emanuel)
echo "emanuel shmuelov" | awk '{print $1}'
```

```
# Using delimiter "," and print the lines without the first two segments
(Output: shmuelov)
echo "emanuel=shmuelov" | awk -F"=" '{print $2}'

# In case number of result values is unknown
echo "my,name,is,emanuel,shmuelov" | awk -F"," '{OFS=":" ;
$1=$2="" ; print $0}' # Output is "::is:emanuel:shmuelov"

echo "my,name,is,emanuel,shmuelov" | awk -F"," '{OFS=":" ;
$1=$2="" ; print $0}' | sed "s/::#g" # Output is
"is:emanuel:shmuelov"

dirname "/tmp/dir1/file"   # Output: /tmp/dir1
basename "/tmp/dir1/file1" # Output: file1

# Convert string/file case from lower to upper and vice versa
dd if=input_file of=output_file conv=ucase
dd if=input_file of=output_file conv=lcase
tr '[:lower:]' '[:upper:]' < input.txt > output.txt
awk '{ print toupper($0) }' input.txt > output.txt

# Using grep command
grep 'apples\|oranges' fruits.txt  #Look for apples OR oranges
grep -e "apples" -e "oranges" fruits.txt  # Same as above

# Same but using extended regular expressions
grep -E 'green apples|red oranges' fruits.txt

# Using regular expressions: search for all lines with
# following pattern:
# STARTING with 5, then any num of characters, then "," and
# then "-1", then and set of characters and ENDING with 0
grep -E "^5.*,.*,\-1*0$" file

# Colorize output
grep --color=auto abc  a_file.txt

# Preventing from grep to appears when using "ps"
ps -ef | grep emac[s]

# Provide the  pids of all "less" processes
pgrep less
# Provide the  pids of all "less" processes owned by "root"
pgrep -u root less
```

```
# Teach your shell new commands
alias grep="grep --color=auto"
alias ll='ls -lth --color=tty'

# xargs - delivering arguments to command
# Find files named ‘core’ inside the directory /tmp and delete them
find /tmp -name core -type f -print0 | xargs -0 /bin/rm -f

# List the home directory content of all users in the system
cut -d: -f6 < /etc/passwd | sort | xargs ls -l

# Read input from stdin (user)
read LINE_FROM_USER
echo $LINE_FROM_USER
```

16. Appendix II - Using VIM

Command	Description
:q	Exit vi
:q!	Exit Vim without saving changes
:wq	Write the file and exit
u	Undo last action
Ctrl + r	Redo
.	Repeat last action
Moving around	
b	To the beginning of a word
e	To the end of a word
:n	Jump to line number *n*
Editing	
i	Insert before cursor
a	Append after cursor
A	Append to the end of the current line
o	Open a new line below and insert
O	Open a new line above and insert
C	Change the rest of the current line
r	Overwrite one character
ESC	Exit insert/overwrite mode
x	Delete characters under the cursor
dd	Delete the current line
~	Change the case of characters
yy	Copy one line
p	Paste the copied line
Visual mode	
v	Start highlighting characters

Command	Description
V	Start highlighting lines
Search & Replace	
/string	Search the file for *pattern*
n	Scan for next search match in the same direction
N	Scan for next search match but opposite direction
:<R>s/*foo*/*bar*/<*A*>	Substitute *foo* with *bar*. *r* determines the range and *a* determines the arguments <u><R> can be:</u> nothing – work on same line number – work on line number % - work on all file <u><A> can be:</u> g – Replace all occurrences i – ignore case c – confirm each substitution

17. Appendix III – Common Linux Commands

Command	Description
alias	Create your own name for a command
arch	Print machine architecture
awk/gawk	pattern scanning and processing language
basename	Remove directory and suffix from a file name
bash	GNU Bourne-Again Shell
bc	Command line calculator
bunzip2	Unzip .bz2 files
cat	Concatenate a file print it to the screen
chmod	Change file permissions
chown	Change the owner of a file
clear	Clear terminal screen (command line)
cp	Copy command
cut	Print selected parts of lines to standard output
date	Display date and time
dc	Command line calculator
df	Show amount of disk space free
diff	Determine difference between two files
diff3	Determine difference between 3 files
dos2unix	Converts plain text files in DOS/MAC format to UNIX format
du	Show disk usage
echo	Display a line of text
egrep	Print lines matching a pattern
env	Display the path
false	Exit with a status code indicating failure
find	Find a file

Command	Description
free	Display free memory
grep	Search for a pattern using regular expression
gunzip	Unzip .gz files
gzip	Compress using Lempel-Ziv coding (LZ77)
halt	Stop the system
head	Print the first 10 lines of a file to standard output
history	Display entire command history
hostname	Show or set the system's host name
id	Print information for username, or the current user
ifconfig	Display network and hardware addresses
kill	Terminate a process
ln	Create a link to the specified TARGET with optional LINK_NAME
locate	Locate a file
ls	List directory contents
lynx	Command to start the Lynx browser
mac2unix	Converts plain text files in DOS/MAC format to UNIX format
man	Display a particular manual entry
mkdir	Create a directory
mktemp	Make temporary filename (unique)
more	Page through text one screen at a time.
mv	Move and / or rename files
pidof	Get process id using process name
ping	Send ICMP ECHO_REQUEST to network hosts
pwd	Print Working Directory
reboot	Stop the system, power off, reboot
rename	Rename files
rm	Remove files or directories

Command	Description
rmdir	Remove a directory
rpm	rpm command options
sed	Stream editor
sh	Shell (BASH)
sleep	Delay for a specified amount of time
sort	Sort lines of a text file
ssh	Secure shell connection command
su	Become super user (root)
tail	Print the last 10 Lines of a file standard output
tar	Create an Archive
tcsh	Enhanced completely compatible version of the Berkeley UNIX C shell, csh
tee	Copy standard input to each file, and also to standard output
telnet	User interface to the telnet protocol
touch	Change file timestamps
tree	Display file tree
true	Exit with a status code indicating success
uname	Print system information (kernel version)
uniq	Remove duplicate lines from sorted file
unlink	Call the unlink function to remove the specified file
untar	Un-archive a file
unzip	Unzip .zip files
usleep	Sleep a given number of microseconds. default is 1
vi	Start the vi editor
w	Show who is logged on and what they are doing
wc	Word count of a file
wget	Non-interactive download of files from the Web

Command	Description
which	Find command path
who	Show who is logged on
whoami	Print effective userid
whois	Client for the whois service
zip	Compression and file packaging utility
zipinfo	List detailed information about a ZIP archive