OrSkl Academy

Unix Shell Scripting

By,

Kumar

Oracle DBA – Architect & Consultant

Contents

1. BASH SCRIPT BASICS 4
1.1. Shell programs 4
1.2. Executing commands 6
1.3. Building blocks 8
1.4. Creating and running script 10
1.5. Debugging scripts 16
2. BASH ENVIRONMENT 21
2.1. Initialization 21
2.2. Variables and characters 23
2.3. Expansions and aliases 31
2.4. Regular expressions 39
2.5. grep examples 41
2.6. Pattern matching 43
3. GNU 44
3.1. Interactive editing 44
3.2. Non interactive editing 46
3.3. AWK programming 47
3.4. AWK print program 47
3.5. AWK variables 49
4. CONDITIONS & INTERACTIVE SCRIPTS 53
4.1. IF clause 53
4.2. Advanced IF usage 57
4.3. CASE statements 60
4.4. Display user messages 62
4.5. User input 65
5. REPETITIVE TASKS & VARIABLES 69
5.1. FOR loop 69
5.2. WHILE loop 70
5.3. UNTIL loop 72
5.4. I/O redirection 74

5.5. Break and continue 74
5.6. Menu making 77
5.7. Variables 79
5.8. Operations on variables 81
6. FUNCTIONS & SIGNALS 82
6.1. Signals 82
6.2. Traps 83
6.3. Functions 85
6.4. Common shell features 87
6.5. Differing shell features 88
7. TOOLS DEVELOPMENT for DBA's 91
7.1. Storage check tool 91
7.2. Backups tool 91
7.3. Refresh tool 92
7.4. User creation tool 93

1. BASH SCRIPT BASICS

1.1.Shell programs

General shell functions -

The UNIX shell program interprets user commands, which are either directly entered by the user, or which can be read from a file called the shell script or shell program. Shell scripts are interpreted, not compiled. The shell reads commands from the script line per line and searches for those commands on the system, while a compiler converts a program into machine readable form, an executable file - which may then be used in a shell script.

Apart from passing commands to the kernel, the main task of a shell is providing a user environment, which can be configured individually using shell resource configuration file

Shell types-

Just like people know different languages and dialects, your UNIX system will usually offer a variety of shell types:

- **sh or Bourne Shell:** the original shell still used on UNIX systems and in UNIX-related environments. This is the basic shell, a small program with few features. While this is not the standard shell, it is still available on every Linux system for compatibility with UNIX programs.
- **bash or Bourne Again shell:** the standard GNU shell, intuitive and flexible. Probably most advisable for beginning users while being at the same time a powerful tool for the advanced and professional user. On Linux, bash is the standard shell for common users. This shell is a so-called superset of the Bourne shell, a set of add-ons and plug-ins. This means that the Bourne Again shell is compatible with the Bourne shell: commands that work in sh, also work in bash. However, the reverse is not always the case. All examples and exercises in this book use bash.
- **csh or C shell:** the syntax of this shell resembles that of the C programming language. Sometimes asked for by programmers.
- **tcsh or TENEX C shell:** a superset of the common C shell, enhancing user-friendliness and speed. That is why some also call it the Turbo C shell.
- **ksh or the Korn shell:** sometimes appreciated by people with a UNIX background. A superset of the Bourne shell; with standard configuration a nightmare for beginning users.

The file /etc/shells gives an overview of known shells on a Linux system:

```
[oracle@OEL-11g ~]$ cat /etc/shells

/bin/sh

/bin/bash

/sbin/nologin
```

```
/bin/dash

/bin/tcsh

/bin/csh

/bin/ksh
```

Your default shell is set in the /etc/passwd file, like this line for user

```
 [oracle@OEL-11g ~]$ echo $SHELL

/bin/bash

[oracle@OEL-11g ~]$ cat /etc/passwd | grep oracle

oracle:x:500:500::/home/oracle:/bin/bash
```

To switch from one shell to another, just enter the name of the new shell in the active terminal. The system finds the directory where the name occurs using the PATH settings, and since a shell is an executable file (program), the current shell activates it and it gets executed. A new prompt is usually shown, because each shell has its typical appearance:

```
[oracle@OEL-11g ~]$ echo $SHELL

/bin/bash

[oracle@OEL-11g ~]$ ksh

$ date

Sun Apr 26 21:34:28 IST 2015

$ exit

[oracle@OEL-11g ~]$

[oracle@OEL-11g ~]$ usermod oracle -s ksh

/usr/sbin/usermod: Permission denied.

[oracle@OEL-11g ~]$ cat /etc/passwd | grep oracle

oracle:x:500:500::/home/oracle:/bin/bash
```

Advantages of the Bourne Again SHell

- Interactive shells
- Non-Interactive shells
- Conditionals

- Shell arithmetic
- Aliases
- Arrays
- Directory stack
- The prompt
- The restricted shell

Activity:

- Check all the shells supported by linux.
```
cat /etc/shells
```
- Check the default shell in linux.
```
echo $SHELL
cat /etc/passwd | grep oracle
date
```
- Move from one shell to different shell.
```
ksh
date
```

1.2.Executing commands

General-

Bash determines the type of program that is to be executed. Normal programs are system commands that exist in compiled form on your system. When such a program is executed, a new process is created because Bash makes an exact copy of itself. This child process has the same environment as its parent, only the process ID number is different. This procedure is called forking.

After the forking process, the address space of the child process is overwritten with the new process data. This is done through an exec call to the system.

The fork-and-exec mechanism thus switches an old command with a new, while the environment in which the new program is executed remains the same, including configuration of input and output devices, environment variables and priority. This mechanism is used to create all UNIX processes, so it also applies to the Linux operating system. Even the first process, init, with process ID 1, is forked during the boot procedure in the so-called bootstrapping procedure.

Shell built-in commands

Built-in commands are contained within the shell itself. When the name of a built-in command is used as the first word of a simple command, the shell executes the command directly, without creating a new process. Built-in commands are necessary to implement functionality impossible or inconvenient to obtain with separate utilities.

Bash supports 3 types of built-in commands:

<u>Bourne Shell built-ins:</u>

:, ., break, cd, continue, eval, exec, exit, export, getopts, hash, pwd, readonly, return, set, shift, test, [, times, trap, umask and unset.

<u>Bash built-in commands:</u>

alias, bind, builtin, command, declare, echo, enable, help, let, local, logout, printf, read, shopt, type, typeset, ulimit and unalias.

<u>Special built-in commands:</u>

When Bash is executing in POSIX mode, the special built-ins differ from other built-in commands in three respects:

Special built-ins are found before shell functions during command lookup.

If a special built-in returns an error status, a non-interactive shell exits.

Assignment statements preceding the command stay in effect in the shell environment after the command completes.

The POSIX special built-ins are :, ., break, continue, eval, exec, exit, export, readonly, return, set, shift, trap and unset.

Most of these built-ins will be discussed in the next chapters. For those commands for which this is not the case, we refer to the Info pages.

Executing programs from a script

She-Bang - #! : this will decide in which shell the script has to be executed. It should always be defined in the first line of the script.

When the program being executed is a shell script, bash will create a new bash process using a fork. This subshell reads the lines from the shell script one line at a time. Commands on each line are read, interpreted and executed as if they would have come directly from the keyboard.

While the subshell processes each line of the script, the parent shell waits for its child process to finish. When there are no more lines in the shell script to read, the subshell terminates. The parent shell awakes and displays a new prompt.

1.3.Building blocks

Shell building blocks:

Shell syntax

If input is not commented, the shell reads it and divides it into words and operators, employing quoting rules to define the meaning of each character of input. Then these words and operators are translated into commands and other constructs, which return an exit status available for inspection or processing. The above fork-and-exec scheme is only applied after the shell has analyzed input in the following way:

- The shell reads its input from a file, from a string or from the user's terminal.
- Input is broken up into words and operators, obeying the quoting rules, see Chapter 3. These tokens are separated by meta characters. Alias expansion is performed.
- The shell parses (analyzes and substitutes) the tokens into simple and compound commands.
- Bash performs various shell expansions, breaking the expanded tokens into lists of filenames and commands and arguments.
- Redirection is performed if necessary, redirection operators and their operands are removed from the argument list.
- Commands are executed.
- Optionally the shell waits for the command to complete and collects its exit status.

Shell commands

A simple shell command such as touch file1 file2 file3 consists of the command itself followed by arguments, separated by spaces.

More complex shell commands are composed of simple commands arranged together in a variety of ways: in a pipeline in which the output of one command becomes the input of a second, in a loop or conditional construct, or in some other grouping. A couple of examples:

```
ls | more
```

```
gunzip file.tar.gz | tar xvf -
```

Shell functions

Shell functions are a way to group commands for later execution using a single name for the group. They are executed just like a "regular" command. When the name of a shell function is used as a simple command name, the list of commands associated with that function name is executed.

Shell functions are executed in the current shell context; no new process is created to interpret them.

Shell parameters

A parameter is an entity that stores values. It can be a name, a number or a special value. For the shell's purpose, a variable is a parameter that stores a name. A variable has a value and zero or more attributes. Variables are created with the declare shell built-in command.

If no value is given, a variable is assigned the null string. Variables can only be removed with the unset built-in.

Shell expansions

Shell expansion is performed after each command line has been split into tokens. These are the expansions performed:

- Brace expansion
- Tilde expansion
- Parameter and variable expansion
- Command substitution
- Arithmetic expansion
- Word splitting
- Filename expansion

Redirections

Before a command is executed, its input and output may be redirected using a special notation interpreted by the shell. Redirection may also be used to open and close files for the current shell execution environment.

Executing commands

When executing a command, the words that the parser has marked as variable assignments (preceding the command name) and redirections are saved for later reference. Words that are not variable assignments or redirections are expanded; the first remaining word after expansion is taken to be the name of the command and the rest are arguments to that command. Then redirections are performed, then strings assigned to variables are expanded. If no command name results, variables will affect the current shell environment.

An important part of the tasks of the shell is to search for commands. Bash does this as follows:

Check whether the command contains slashes. If not, first check with the function list to see if it contains a command by the name we are looking for.

If command is not a function, check for it in the built-in list.

If command is neither a function nor a built-in, look for it analyzing the directories listed in PATH. Bash uses a hash table (data storage area in memory) to remember the full path names of executables so extensive PATH searches can be avoided.

If the search is unsuccessful, bash prints an error message and returns an exit status of 127.

If the search was successful or if the command contains slashes, the shell executes the command in a separate execution environment.

If execution fails because the file is not executable and not a directory, it is assumed to be a shell script.

If the command was not begun asynchronously, the shell waits for the command to complete and collects its exit status.

Shell scripts

When a file containing shell commands is used as the first non-option argument when invoking Bash (without -c or -s, this will create a non-interactive shell. This shell first searches for the script file in the current directory, then looks in PATH if the file cannot be found there.

Activity:

- Check command is workable.

```
which cat
which cold
```

- Understand how a command gets executed on UNIX platform. Importance of PATH variable.

```
echo $PATH
```

- Write a command yourself and add location in path variable.

```
vi oracledate
#!/bin/bash
date
chmod +x oracledate
oracledate
cp oracledate /oracle/base/product/12.1.0/db/bin
oracledate
```

1.4.Creating and running script

Writing and naming

A shell script is a sequence of commands for which you have a repeated use. This sequence is typically executed by entering the name of the script on the command line. Alternatively, you can use scripts to automate tasks using the cron facility. Another use for scripts is in the UNIX boot and shutdown procedure, where operation of daemons and services are defined in init scripts.

To create a shell script, open a new empty file in your editor. Any text editor will do: vim, emacs, gedit, dtpad et cetera are all valid. You might want to chose a more advanced editor like vim or emacs, however, because these can be configured to recognize shell and Bash syntax and can be a great help in preventing those errors that beginners frequently make, such as forgetting brackets and semi-colons.

Tip Syntax highlighting in vim

In order to activate syntax highlighting in vim, use the command

```
:syntax enable
```

or

```
:sy enable
```

or

```
:syn enable
```

You can add this setting to your .vimrc file to make it permanent.

Put UNIX commands in the new empty file, like you would enter them on the command line. As discussed in the previous chapter (see Section 1.3), commands can be shell functions, shell built-ins, UNIX commands and other scripts.

Give your script a sensible name that gives a hint about what the script does. Make sure that your script name does not conflict with existing commands. In order to ensure that no confusion can rise, script names often end in .sh; even so, there might be other scripts on your system with the same name as the one you chose. Check using which, whereis and other commands for finding information about programs and files:

```
which -a script_name

whereis script_name

locate script_name
```

script1.sh

In this example we use the echo Bash built-in to inform the user about what is going to happen, before the task that will create the output is executed. It is strongly advised to inform users about what a script is doing, in order to prevent them from becoming nervous because the script is not doing anything. We will return to the subject of notifying users.

Write this script for yourself as well. It might be a good idea to create a directory ~/scripts to hold your scripts. Add the directory to the contents of the PATH variable:

```
export PATH="$PATH:~/scripts"
```

If you are just getting started with Bash, use a text editor that uses different colours for different shell constructs. Syntax highlighting is supported by vim, gvim, (x)emacs, kwrite and many other editors; check the documentation of your favorite editor.

Note Different prompts

The prompts throughout this course vary depending on the author's mood. This resembles much more real life situations than the standard educational $ prompt. The only convention we stick to, is that the root prompt ends in a hash mark (#).

Executing the script

The script should have execute permissions for the correct owners in order to be runnable. When setting permissions, check that you really obtained the permissions that you want. When this is done, the script can run like any other command:

```
willy:~/scripts> chmod u+x script1.sh

willy:~/scripts> ls -l script1.sh

-rwxrw-r--    1 willy        willy              456 Dec 24 17:11 script1.sh

willy:~> script1.sh
```

The script starts now.

```
Hi, willy!
```

I will now fetch you a list of connected users:

```
  3:38pm  up 18 days,  5:37,  4 users,  load average: 0.12, 0.22, 0.15

USER     TTY      FROM              LOGIN@   IDLE   JCPU   PCPU  WHAT

root     tty2     -                 Sat 2pm  4:25m  0.24s  0.05s  -bash

willy     :0      -                 Sat 2pm   ?     0.00s   ?     -

willy    pts/3    -                 Sat 2pm  3:33m 36.39s 36.39s  BitchX willy ir

willy    pts/2    -                 Sat 2pm  3:33m  0.13s  0.06s  /usr/bin/screen
```

I'm setting two variables now.

```
This is a string: black
```

```
And this is a number: 9

I'm giving you back your prompt now.

willy:~/scripts> echo $COLOUR

willy:~/scripts> echo $VALUE

willy:~/scripts>
```

This is the most common way to execute a script. It is preferred to execute the script like this in a subshell. The variables, functions and aliases created in this subshell are only known to the particular bash session of that subshell. When that shell exits and the parent regains control, everything is cleaned up and all changes to the state of the shell made by the script, are forgotten.

If you did not put the scripts directory in your PATH, and . (the current directory) is not in the PATH either, you can activate the script like this:

```
./script_name.sh
```

A script can also explicitly be executed by a given shell, but generally we only do this if we want to obtain special behavior, such as checking if the script works with another shell or printing traces for debugging:

```
rbash script_name.sh

sh script_name.sh

bash -x script_name.sh
```

The specified shell will start as a subshell of your current shell and execute the script. This is done when you want the script to start up with specific options or under specific conditions which are not specified in the script.

If you don't want to start a new shell but execute the script in the current shell, you source it:

```
source script_name.sh
```

Tip source = .

The Bash source built-in is a synonym for the Bourne shell . (dot) command.

The script does not need execute permission in this case. Commands are executed in the current shell context, so any changes made to your environment will be visible when the script finishes execution:

```
willy:~/scripts> source script1.sh

--output ommitted--
```

```
willy:~/scripts> echo $VALUE

9

willy:~/scripts>
```

Which shell will run the script?

When running a script in a subshell, you should define which shell should run the script. The shell type in which you wrote the script might not be the default on your system, so commands you entered might result in errors when executed by the wrong shell.

The first line of the script determines the shell to start. The first two characters of the first line should be #!, then follows the path to the shell that should interpret the commands that follow. Blank lines are also considered to be lines, so don't start your script with an empty line.

For the purpose of this course, all scripts will start with the line

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

As noted before, this implies that the Bash executable can be found in /bin.

Adding comments

You should be aware of the fact that you might not be the only person reading your code. A lot of users and system administrators run scripts that were written by other people. If they want to see how you did it, comments are useful to enlighten the reader.

Comments also make your own life easier. Say that you had to read a lot of man pages in order to achieve a particular result with some command that you used in your script. You won't remember how it worked if you need to change your script after a few weeks or months, unless you have commented what you did, how you did it and/or why you did it.

Take the script1.sh example and copy it to commented-script1.sh, which we edit so that the comments reflect what the script does. Everything the shell encounters after a hash mark on a line is ignored and only visible upon opening the shell script file:

```
#!/bin/bash

# This script clears the terminal, displays a greeting and gives information

# about currently connected users.  The two example variables are set and displayed.

clear                               # clear terminal window

echo "The script starts now."

echo "Hi, $USER!"                   # dollar sign is used to get content of variable
```

```
echo

echo "I will now fetch you a list of connected users:"

echo

w                                   # show who is logged on and

echo                                # what they are doing

echo "I'm setting two variables now."

COLOUR="black"                                          # set a local shell variable

VALUE="9"                                     # set a local shell variable

echo "This is a string: $COLOUR"              # display content of variable

echo "And this is a number: $VALUE"           # display content of variable

echo

echo "I'm giving you back your prompt now."

echo
```

In a decent script, the first lines are usually comment about what to expect. Then each big chunk of commands will be commented as needed for clarity's sake. Linux init scripts, as an example, in your system's init.d directory, are usually well commented since they have to be readable and editable by everyone running Linux.

Activity:

- Writing a script.
  ```
  vi script1.sh
  ```
- Locating your scripts.
  ```
  find . -name script*
  locate script* - requires local database of linux to be
  updated
  updatedb
  locate script*
  ```
- Ways of executing the script.
  ```
  chmod 755 script1.sh
  chmod +x script1.sh
  vi script1.sh
  ./script1.sh
  sh script1.sh
  ```

```
ksh script1.sh
bash script1.sh
```

- Comment appropriately.

```
vi scirpt1.sh
#Finding the current system time stamp
```

- Use SHE-BANG in every script.

```
#!/bin/bash – this one should be existing.
./script1.sh
```

- Write a script and crontab it.

```
vi script2.sh
#!/bin/bash
date > /home/oracle/scripting/script2.out
chmod +x script2.sh
date
crontab -l
```

1.5.Debugging scripts

Debugging on the entire script

When things don't go according to plan, you need to determine what exactly causes the script to fail. Bash provides extensive debugging features. The most common is to start up the subshell with the -x option, which will run the entire script in debug mode. Traces of each command plus its arguments are printed to standard output after the commands have been expanded but before they are executed.

This is the commented-script1.sh script ran in debug mode. Note again that the added comments are not visible in the output of the script.
willy:~/scripts> bash -x script1.sh

```
+ clear

+ echo 'The script starts now.'

The script starts now.

+ echo 'Hi, willy!'

Hi, willy!

+ echo

+ echo 'I will now fetch you a list of connected users:'

I will now fetch you a list of connected users:
```

```
+ echo

+ w

  4:50pm  up 18 days,  6:49,  4 users,  load average: 0.58, 0.62, 0.40

USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU  WHAT

root     tty2     -                Sat 2pm  5:36m  0.24s  0.05s  -bash

willy     :0      -                Sat 2pm   ?     0.00s   ?     -

willy     pts/3   -                Sat 2pm 43:13  36.82s 36.82s  BitchX willy ir

willy    pts/2    -                Sat 2pm 43:13   0.13s  0.06s  /usr/bin/screen

+ echo

+ echo 'I'\''m setting two variables now.'

I'm setting two variables now.

+ COLOUR=black

+ VALUE=9

+ echo 'This is a string: '

This is a string:

+  echo 'And this is a number: '

And this is a number:

+ echo

+ echo 'I'\''m giving you back your prompt now.'

I'm giving you back your prompt now.

+ echo
```

There is now a full-fledged debugger for Bash, available at SourceForge. These debugging features are available in most modern versions of Bash, starting from 3.x.

Debugging on part(s) of the script

Using the set Bash built-in you can run in normal mode those portions of the script of which you are sure they are without fault, and display debugging information only for troublesome zones. Say we are not sure what the w command will do in the example commented-script1.sh, then we could enclose it in the script like this:

```
set -x                    # activate debugging from here

w

set +x                    # stop debugging from here
```

Output then looks like this:

```
willy: ~/scripts> script1.sh

The script starts now.

Hi, willy!
```

I will now fetch you a list of connected users:

```
+ w

  5:00pm  up 18 days,  7:00,  4 users,  load average: 0.79, 0.39, 0.33

USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU  WHAT

root     tty2     -               Sat 2pm  5:47m  0.24s  0.05s  -bash

willy    :0       -              Sat 2pm   ?     0.00s   ?     -

willy    pts/3    -               Sat 2pm 54:02  36.88s 36.88s  BitchX willyke

willy    pts/2    -               Sat 2pm 54:02   0.13s  0.06s  /usr/bin/screen

+ set +x
```

I'm setting two variables now.

This is a string:

And this is a number:

I'm giving you back your prompt now.

```
willy: ~/scripts>
```

You can switch debugging mode on and off as many times as you want within the same script.

The table below gives an overview of other useful Bash options:

Table 2-1. Overview of set debugging options

Short notation	Long notation	Result
set -f	set -o noglob	Disable file name generation using metacharacters (globbing).

set -v	set -o verbose	Prints shell input lines as they are read.
set -x	set -o xtrace	Print command traces before executing command.

The dash is used to activate a shell option and a plus to deactivate it. Don't let this confuse you!

In the example below, we demonstrate these options on the command line:
willy:~/scripts> set -v

```
willy:~/scripts> ls

ls

commented-scripts.sh   script1.sh

willy:~/scripts> set +v

set +v

willy:~/scripts> ls *

commented-scripts.sh    script1.sh

willy:~/scripts> set -f

willy:~/scripts> ls *

ls: *: No such file or directory

willy:~/scripts> touch *


willy:~/scripts> ls

*   commented-scripts.sh    script1.sh

willy:~/scripts> rm *

willy:~/scripts> ls

commented-scripts.sh    script1.sh
```

Alternatively, these modes can be specified in the script itself, by adding the desired options to the first line shell declaration. Options can be combined, as is usually the case with UNIX commands:

```
#!/bin/bash -xv
```

Once you found the buggy part of your script, you can add echo statements before each command of which you are unsure, so that you will see exactly where and why things don't work. In the example commented-script1.sh script, it could be done like this, still assuming that the displaying of users gives

us problems:
echo "debug message: now attempting to start w command"; w

In more advanced scripts, the echo can be inserted to display the content of variables at different stages in the script, so that flaws can be detected:
echo "Variable VARNAME is now set to $VARNAME."

Activity:

- create issue in the script.

```
vi script1.sh
date
dummy
cat /etc/hosts
```

- Full script debugging ways.

```
bash -x script1.sh

vi script1.sh
set -x
./script1.sh

vi script1.sh
#!/bin/bash -x
./script1.sh
```

- Partial debugging ways.

```
vi script1.sh
set -x
dummy
set +x
```

2. BASH ENVIRONMENT

2.1.Initialization

System-wide configuration files

/etc/profile

When invoked interactively with the --login option or when invoked as sh, Bash reads the /etc/profile instructions. These usually set the shell variables PATH, USER, MAIL, HOSTNAME and HISTSIZE.

On some systems, the umask value is configured in /etc/profile; on other systems this file holds pointers to other configuration files such as:

/etc/inputrc, the system-wide Readline initialization file where you can configure the command line bell-style.

the /etc/profile.d directory, which contains files configuring system-wide behavior of specific programs.

/etc/bashrc

On systems offering multiple types of shells, it might be better to put Bash-specific configurations in this file, since /etc/profile is also read by other shells, such as the Bourne shell. Errors generated by shells that don't understand the Bash syntax are prevented by splitting the configuration files for the different types of shells. In such cases, the user's ~/.bashrc might point to /etc/bashrc in order to include it in the shell initialization process upon login.

You might also find that /etc/profile on your system only holds shell environment and program startup settings, while /etc/bashrc contains system-wide definitions for shell functions and aliases. The /etc/bashrc file might be referred to in /etc/profile or in individual user shell initialization files.

Individual user configuration files

Note I don't have these files?!

These files might not be in your home directory by default; create them if needed.

~/.bash_profile

This is the preferred configuration file for configuring user environments individually. In this file, users can add extra configuration options or change default settings: franky~> cat .bash_profile

~/.bash_login

This file contains specific settings that are normally only executed when you log in to the system.

~/.bashrc

Today, it is more common to use a non-login shell, for instance when logged in graphically using X terminal windows. Upon opening such a window, the user does not have to provide a user name or password; no authentication is done. Bash searches for ~/.bashrc when this happens, so it is referred to in the files read upon login as well, which means you don't have to enter the same settings in multiple files.

In this user's .bashrc a couple of aliases are defined and variables for specific programs are set after the system-wide /etc/bashrc is read.

~/.bash_logout

This file contains specific instructions for the logout procedure. In the example, the terminal window is cleared upon logout. This is useful for remote connections, which will leave a clean window after closing them. franky ~> cat .bash_logout

Changing shell configuration files

When making changes to any of the above files, users have to either reconnect to the system or source the altered file for the changes to take effect. By interpreting the script this way, changes are applied to the current shell session

Most shell scripts execute in a private environment: variables are not inherited by child processes unless they are exported by the parent shell. Sourcing a file containing shell commands is a way of applying changes to your own environment and setting variables in the current shell.

This example also demonstrates the use of different prompt settings by different users. In this case, red means danger. When you have a green prompt, don't worry too much.

Note that `source resourcefile` is the same as `. resourcefile`.

Activity:

- Check the list of files in the user directory.
```
ls -la
```
- Check content in each of the hidden initialization files.
```
cat .bash_profile
cat .bashrc
cat .bash_logout
```
- Open /etc/profile and understand the working model.
- Understand working model of .bashrc
- Edit .bash_profile and test.
```
echo $TEST
vi .bash_profile
TEST="Its user defined variable"
```

```
then log off and login
echo $TEST
```

2.2.Variables and characters

Types of variables

As seen in the examples above, shell variables are in uppercase characters by convention. Bash keeps a list of two types of variables:

Global variables

Global variables or environment variables are available in all shells. The env or printenv commands can be used to display environment variables. These programs come with the sh-utils package.

Local variables

Local variables are only available in the current shell. Using the set built-in command without any options will display a list of all variables (including environment variables) and functions. The output will be sorted according to the current locale and displayed in a reusable format.

Variables by content

Apart from dividing variables in local and global variables, we can also divide them in categories according to the sort of content the variable contains. In this respect, variables come in 4 types:

String variables

Integer variables

Constant variables

Array variables

Creating variables

Variables are case sensitive and capitalized by default. Giving local variables a lowercase name is a convention which is sometimes applied. However, you are free to use the names you want or to mix cases. Variables can also contain digits, but a name starting with a digit is not allowed: prompt> export 1number=1

bash: export: `1number=1': not a valid identifier

To set a variable in the shell, use

VARNAME="value"

Putting spaces around the equal sign will cause errors. It is a good habit to quote content strings when assigning values to variables: this will reduce the chance that you make errors.

Exporting variables

A variable created like the ones in the example above is only available to the current shell. It is a local variable: child processes of the current shell will not be aware of this variable. In order to pass variables to a subshell, we need to export them using the export built-in command. Variables that are exported are referred to as environment variables. Setting and exporting is usually done in one step:

```
export VARNAME="value"
```

Reserved variables

Bourne shell reserved variables

Bash uses certain shell variables in the same way as the Bourne shell. In some cases, Bash assigns a default value to the variable. The table below gives an overview of these plain shell variables:

Table 3-1. Reserved Bourne shell variables

Variable name	Definition
CDPATH	A colon-separated list of directories used as a search path for the cd built-in command.
HOME	The current user's home directory; the default for the cd built-in. The value of this variable is also used by tilde expansion.
IFS	A list of characters that separate fields; used when the shell splits words as part of expansion.
MAIL	If this parameter is set to a file name and the MAILPATH variable is not set, Bash informs the user of the arrival of mail in the specified file.
MAILPATH	A colon-separated list of file names which the shell periodically checks for new mail.
OPTARG	The value of the last option argument processed by the getopts built-in.
OPTIND	The index of the last option argument processed by the getopts built-in.
PATH	A colon-separated list of directories in which the shell looks for commands.
PS1	The primary prompt string. The default value is "'\s-\v\$ '".
PS2	The secondary prompt string. The default value is "'> '".

Bash reserved variables

These variables are set or used by Bash, but other shells do not normally treat them specially.

Table 3-2. Reserved Bash variables

Variable name	Definition
auto_resume	This variable controls how the shell interacts with the user and job control.
BASH	The full pathname used to execute the current instance of Bash.
BASH_ENV	If this variable is set when Bash is invoked to execute a shell script, its value is expanded and used as the name of a startup file to read before executing the script.
BASH_VERSION	The version number of the current instance of Bash.
BASH_VERSINFO	A read-only array variable whose members hold version information for this instance of Bash.
COLUMNS	Used by the select built-in to determine the terminal width when printing selection lists. Automatically set upon receipt of a SIGWINCH signal.
COMP_CWORD	An index into ${COMP_WORDS} of the word containing the current cursor position.
COMP_LINE	The current command line.
COMP_POINT	The index of the current cursor position relative to the beginning of the current command.
COMP_WORDS	An array variable consisting of the individual words in the current command line.
COMPREPLY	An array variable from which Bash reads the possible completions generated by a shell function invoked by the programmable completion facility.
DIRSTACK	An array variable containing the current contents of the directory stack.
EUID	The numeric effective user ID of the current user.
FCEDIT	The editor used as a default by the -e option to the fc built-in command.
FIGNORE	A colon-separated list of suffixes to ignore when performing file name completion.
FUNCNAME	The name of any currently-executing shell function.
GLOBIGNORE	A colon-separated list of patterns defining the set of file names to be ignored by file name expansion.
GROUPS	An array variable containing the list of groups of which the current user is a member.
histchars	Up to three characters which control history expansion, quick substitution, and tokenization.

HISTCMD The history number, or index in the history list, of the current command.

HISTCONTROL Defines whether a command is added to the history file.

HISTFILE The name of the file to which the command history is saved. The default value is ~/.bash_history.

HISTFILESIZE The maximum number of lines contained in the history file, defaults to 500.

HISTIGNORE A colon-separated list of patterns used to decide which command lines should be saved in the history list.

HISTSIZE The maximum number of commands to remember on the history list, default is 500.

HOSTFILE Contains the name of a file in the same format as /etc/hosts that should be read when the shell needs to complete a hostname.

HOSTNAME The name of the current host.

HOSTTYPE A string describing the machine Bash is running on.

IGNOREEOF Controls the action of the shell on receipt of an EOF character as the sole input.

INPUTRC The name of the Readline initialization file, overriding the default /etc/inputrc.

LANG Used to determine the locale category for any category not specifically selected with a variable starting with LC_.

LC_ALL This variable overrides the value of LANG and any other LC_ variable specifying a locale category.

LC_COLLATE This variable determines the collation order used when sorting the results of file name expansion, and determines the behavior of range expressions, equivalence classes, and collating sequences within file name expansion and pattern matching.

LC_CTYPE This variable determines the interpretation of characters and the behavior of character classes within file name expansion and pattern matching.

LC_MESSAGES This variable determines the locale used to translate double-quoted strings preceded by a "$" sign.

LC_NUMERIC This variable determines the locale category used for number formatting.

LINENO The line number in the script or shell function currently executing.

LINES Used by the select built-in to determine the column length for printing selection lists.

MACHTYPE A string that fully describes the system type on which Bash is executing, in the standard GNU CPU-COMPANY-SYSTEM format.

MAILCHECK How often (in seconds) that the shell should check for mail in the files specified in the MAILPATH or MAIL variables.

OLDPWD The previous working directory as set by the cd built-in.

OPTERRIf set to the value 1, Bash displays error messages generated by the getopts built-in.

OSTYPE A string describing the operating system Bash is running on.

PIPESTATUS An array variable containing a list of exit status values from the processes in the most recently executed foreground pipeline (which may contain only a single command).

POSIXLY_CORRECT If this variable is in the environment when bash starts, the shell enters POSIX mode.

PPID The process ID of the shell's parent process.

PROMPT_COMMAND If set, the value is interpreted as a command to execute before the printing of each primary prompt (PS1).

PS3 The value of this variable is used as the prompt for the select command. Defaults to "'#? '"

PS4 The value is the prompt printed before the command line is echoed when the -x option is set; defaults to "'+ '".

PWD The current working directory as set by the cd built-in command.

RANDOM Each time this parameter is referenced, a random integer between 0 and 32767 is generated. Assigning a value to this variable seeds the random number generator.

REPLY The default variable for the read built-in.

SECONDS This variable expands to the number of seconds since the shell was started.

SHELLOPTS A colon-separated list of enabled shell options.

SHLVL Incremented by one each time a new instance of Bash is started.

TIMEFORMAT The value of this parameter is used as a format string specifying how the timing information for pipelines prefixed with the time reserved word should be displayed.

TMOUT If set to a value greater than zero, TMOUT is treated as the default timeout for the read built-in. In an interative shell, the value is interpreted as the number of seconds to wait for input after issuing the primary prompt when the shell is interactive. Bash terminates after that number of seconds if input does not arrive.

UID The numeric, real user ID of the current user.

Check the Bash man, info or doc pages for extended information. Some variables are read-only, some are set automatically and some lose their meaning when set to a different value than the default.

Special parameters

The shell treats several parameters specially. These parameters may only be referenced; assignment to them is not allowed.

Table 3-3. Special bash variables

Character	Definition
$*	Expands to the positional parameters, starting from one. When the expansion occurs within double quotes, it expands to a single word with the value of each parameter separated by the first character of the IFS special variable.
$@	Expands to the positional parameters, starting from one. When the expansion occurs within double quotes, each parameter expands to a separate word.
$#	Expands to the number of positional parameters in decimal.
$?	Expands to the exit status of the most recently executed foreground pipeline.
$-	A hyphen expands to the current option flags as specified upon invocation, by the set built-in command, or those set by the shell itself (such as the -i).
$$	Expands to the process ID of the shell.
$!	Expands to the process ID of the most recently executed background (asynchronous) command.
$0	Expands to the name of the shell or shell script.
$_	The underscore variable is set at shell startup and contains the absolute file name of the shell or script being executed as passed in the argument list. Subsequently, it expands to the last argument to the previous command, after expansion. It is also set to the full pathname of each command executed and placed in the environment exported to that command. When checking mail, this parameter holds the name of the mail file.

Note $* vs. $@

The implementation of "$*" has always been a problem and realistically should have been replaced with the behavior of "$@". In almost every case where coders use "$*", they mean "$@". "$*" Can cause bugs and even security holes in your software.

Script recycling with variables

Apart from making the script more readable, variables will also enable you to faster apply a script in another environment or for another purpose.

Quoting characters

Why?

A lot of keys have special meanings in some context or other. Quoting is used to remove the special meaning of characters or words: quotes can disable special treatment for special characters, they can prevent reserved words from being recognized as such and they can disable parameter expansion.

Escape characters

Escape characters are used to remove the special meaning from a single character. A non-quoted backslash, \, is used as an escape character in Bash. It preserves the literal value of the next character that follows, with the exception of newline. If a newline character appears immediately after the backslash, it marks the continuation of a line when it is longer that the width of the terminal; the backslash is removed from the input stream and effectively ignored.

Single quotes

Single quotes ('') are used to preserve the literal value of each character enclosed within the quotes. A single quote may not occur between single quotes, even when preceded by a backslash.

Double quotes

Using double quotes the literal value of all characters enclosed is preserved, except for the dollar sign, the backticks (backward single quotes, ``) and the backslash.

The dollar sign and the backticks retain their special meaning within the double quotes.

The backslash retains its meaning only when followed by dollar, backtick, double quote, backslash or newline. Within double quotes, the backslashes are removed from the input stream when followed by one of these characters. Backslashes preceding characters that don't have a special meaning are left unmodified for processing by the shell interpreter.

A double quote may be quoted within double quotes by preceding it with a backslash.

ANSI-C quoting

Words in the form "$'STRING'" are treated in a special way. The word expands to a string, with backslash-escaped characters replaced as specified by the ANSI-C standard. Backslash escape sequences can be found in the Bash documentation.

Locales

A double-quoted string preceded by a dollar sign will cause the string to be translated according to the current locale. If the current locale is "C" or "POSIX", the dollar sign is ignored. If the string is translated and replaced, the replacement is double-quoted.

Activity:

```
o  Check default variables:
   echo $HOME
   echo $MAIL
   echo $PPID
o  Execute successful command and failure command and check
   the result using $?
   date
   echo $?
   system
   echo $?
o  Write a script to understand functionality of $1 $2 $3
   and $#
   vi script3.sh
   #!/bin/bash
   # This script reads 3 positional parameters and prints
   them out.
   POSPAR1="$1"
   POSPAR2="$2"
   POSPAR3="$3"
   echo "$1 is the first positional parameter, \$1."
   echo "$2 is the second positional parameter, \$2."
   echo "$3 is the third positional parameter, \$3."
   echo
   echo "The total number of positional parameters is $#."

   chomod +x script3.sh
   ./script3.sh one two three

o  Understand purpose of \, ' and "
   TEST=1234
   echo $TEST
   echo \$TEST
   echo '$TEST'
   echo "$TEST"
o  Load variables into a file and run source command
   vi variables.log
```

```
TEST1="Its test variable 1"
TEST2="Its test variable 2"
echo $TEST1
echo $TEST2
source variables.log
echo $TEST1
echo $TEST2
```

2.3.Expansions and aliases

General

After the command has been split into tokens (see Section 1.4.1.1), these tokens or words are expanded or resolved. There are eight kinds of expansion performed, which we will discuss in the next sections, in the order that they are expanded.

After all expansions, quote removal is performed.

Brace expansion

Brace expansion is a mechanism by which arbitrary strings may be generated. Patterns to be brace-expanded take the form of an optional PREAMBLE, followed by a series of comma-separated strings between a pair of braces, followed by an optional POSTSCRIPT. The preamble is prefixed to each string contained within the braces, and the postscript is then appended to each resulting string, expanding left to right.

Brace expansions may be nested. The results of each expanded string are not sorted; left to right order is preserved.

Brace expansion is performed before any other expansions, and any characters special to other expansions are preserved in the result. It is strictly textual. Bash does not apply any syntactic interpretation to the context of the expansion or the text between the braces. To avoid conflicts with parameter expansion, the string "${" is not considered eligible for brace expansion.

A correctly-formed brace expansion must contain unquoted opening and closing braces, and at least one unquoted comma. Any incorrectly formed brace expansion is left unchanged.

Tilde expansion

If a word begins with an unquoted tilde character ("~"), all of the characters up to the first unquoted slash (or all characters, if there is no unquoted slash) are considered a tilde-prefix. If none of the characters in the tilde-prefix are quoted, the characters in the tilde-prefix following the tilde are treated as a possible login name. If this login name is the null string, the tilde is replaced with the value of the HOME shell variable. If HOME is unset, the home directory of the user executing the shell is substituted instead. Otherwise, the tilde-prefix is replaced with the home directory associated with the specified login name.

If the tilde-prefix is "~+", the value of the shell variable PWD replaces the tilde-prefix. If the tilde-prefix is "~-", the value of the shell variable OLDPWD, if it is set, is substituted.

If the characters following the tilde in the tilde-prefix consist of a number N, optionally prefixed by a "+" or a "-", the tilde-prefix is replaced with the corresponding element from the directory stack, as it would be displayed by the dirs built-in invoked with the characters following tilde in the tilde-prefix as an argument. If the tilde-prefix, without the tilde, consists of a number without a leading "+" or "-", "+" is assumed.

If the login name is invalid, or the tilde expansion fails, the word is left unchanged.

Each variable assignment is checked for unquoted tilde-prefixes immediately following a ":" or "=". In these cases, tilde expansion is also performed. Consequently, one may use file names with tildes in assignments to PATH, MAILPATH, and CDPATH, and the shell assigns the expanded value.

Example: franky ~> export PATH="$PATH:~/testdir"

~/testdir will be expanded to $HOME/testdir, so if $HOME is /var/home/franky, the directory /var/home/franky/testdir will be added to the content of the PATH variable.

Shell parameter and variable expansion

The "$" character introduces parameter expansion, command substitution, or arithmetic expansion. The parameter name or symbol to be expanded may be enclosed in braces, which are optional but serve to protect the variable to be expanded from characters immediately following it which could be interpreted as part of the name.

When braces are used, the matching ending brace is the first "}" not escaped by a backslash or within a quoted string, and not within an embedded arithmetic expansion, command substitution, or parameter expansion.

The basic form of parameter expansion is "${PARAMETER}". The value of "PARAMETER" is substituted. The braces are required when "PARAMETER" is a positional parameter with more than one digit, or when "PARAMETER" is followed by a character that is not to be interpreted as part of its name.

If the first character of "PARAMETER" is an exclamation point, Bash uses the value of the variable formed from the rest of "PARAMETER" as the name of the variable; this variable is then expanded and that value is used in the rest of the substitution, rather than the value of "PARAMETER" itself. This is known as indirect expansion.

Command substitution

Command substitution allows the output of a command to replace the command itself. Command substitution occurs when a command is enclosed like this:

$(command)

or like this using backticks:

\`command\`

Bash performs the expansion by executing COMMAND and replacing the command substitution with the standard output of the command, with any trailing newlines deleted. Embedded newlines are not deleted, but they may be removed during word splitting.

When the old-style backquoted form of substitution is used, backslash retains its literal meaning except when followed by "$", "\`", or "\". The first backticks not preceded by a backslash terminates the command substitution. When using the "$(COMMAND)" form, all characters between the parentheses make up the command; none are treated specially.

Command substitutions may be nested. To nest when using the backquoted form, escape the inner backticks with backslashes.

If the substitution appears within double quotes, word splitting and file name expansion are not performed on the results.

Arithmetic expansion

Arithmetic expansion allows the evaluation of an arithmetic expression and the substitution of the result. The format for arithmetic expansion is:

$((EXPRESSION))

The expression is treated as if it were within double quotes, but a double quote inside the parentheses is not treated specially. All tokens in the expression undergo parameter expansion, command substitution, and quote removal. Arithmetic substitutions may be nested.

Evaluation of arithmetic expressions is done in fixed-width integers with no check for overflow - although division by zero is trapped and recognized as an error. The operators are roughly the same as in the C programming language. In order of decreasing precedence, the list looks like this:

Table 3-4. Arithmetic operators

Operator	Meaning
VAR++ and VAR--	variable post-increment and post-decrement
++VAR and --VAR	variable pre-increment and pre-decrement
- and +	unary minus and plus
! and ~	logical and bitwise negation
**	exponentiation

*, / and %	multiplication, division, remainder
+ and -	addition, subtraction
<< and >>	left and right bitwise shifts
<=, >=, < and >	comparison operators
== and !=	equality and inequality
&	bitwise AND
^	bitwise exclusive OR
|	bitwise OR
&&	logical AND
||	logical OR
expr ? expr : expr	conditional evaluation
=, *=, /=, %=, +=, -=, <<=, >>=, &=, ^= and |=	assignments
,	separator between expressions

Shell variables are allowed as operands; parameter expansion is performed before the expression is evaluated. Within an expression, shell variables may also be referenced by name without using the parameter expansion syntax. The value of a variable is evaluated as an arithmetic expression when it is referenced. A shell variable need not have its integer attribute turned on to be used in an expression.

Constants with a leading 0 (zero) are interpreted as octal numbers. A leading "0x" or "0X" denotes hexadecimal. Otherwise, numbers take the form "[BASE'#']N", where "BASE" is a decimal number between 2 and 64 representing the arithmetic base, and N is a number in that base. If "BASE'#'" is omitted, then base 10 is used. The digits greater than 9 are represented by the lowercase letters, the uppercase letters, "@", and "_", in that order. If "BASE" is less than or equal to 36, lowercase and uppercase letters may be used inter changably to represent numbers between 10 and 35.

Operators are evaluated in order of precedence. Sub-expressions in parentheses are evaluated first and may override the precedence rules above.

Wherever possible, Bash users should try to use the syntax with square brackets:

$[EXPRESSION]

Process substitution

Process substitution is supported on systems that support named pipes (FIFOs) or the /dev/fd method of naming open files. It takes the form of

<(LIST)

or

>(LIST)

The process LIST is run with its input or output connected to a FIFO or some file in /dev/fd. The name of this file is passed as an argument to the current command as the result of the expansion. If the ">(LIST)" form is used, writing to the file will provide input for LIST. If the "<(LIST)" form is used, the file passed as an argument should be read to obtain the output of LIST. Note that no space may appear between the < or > signs and the left parenthesis, otherwise the construct would be interpreted as a redirection.

When available, process substitution is performed simultaneously with parameter and variable expansion, command substitution, and arithmetic expansion.

Word splitting

The shell scans the results of parameter expansion, command substitution, and arithmetic expansion that did not occur within double quotes for word splitting.

The shell treats each character of $IFS as a delimiter, and splits the results of the other expansions into words on these characters. If IFS is unset, or its value is exactly "'<space><tab><newline>'", the default, then any sequence of IFS characters serves to delimit words. If IFS has a value other than the default, then sequences of the whitespace characters "space" and "Tab" are ignored at the beginning and end of the word, as long as the whitespace character is in the value of IFS (an IFS whitespace character). Any character in IFS that is not IFS whitespace, along with any adjacent IF whitespace characters, delimits a field. A sequence of IFS whitespace characters is also treated as a delimiter. If the value of IFS is null, no word splitting occurs.

Explicit null arguments ("""" or "''") are retained. Unquoted implicit null arguments, resulting from the expansion of parameters that have no values, are removed. If a parameter with no value is expanded within double quotes, a null argument results and is retained.

Note Expansion and word splitting

If no expansion occurs, no splitting is performed.

File name expansion

After word splitting, unless the -f option has been set (see Section 2.3.2), Bash scans each word for the characters "*", "?", and "[". If one of these characters appears, then the word is regarded as a PATTERN, and replaced with an alphabetically sorted list of file names matching the pattern. If no matching file names are found, and the shell option nullglob is disabled, the word is left unchanged. If the nullglob option is set, and no matches are found, the word is removed. If the shell option nocaseglob is enabled, the match is performed without regard to the case of alphabetic characters.

When a pattern is used for file name generation, the character "." at the start of a file name or immediately following a slash must be matched explicitly, unless the shell option dotglob is set. When matching a file name, the slash character must always be matched explicitly. In other cases, the "." character is not treated specially.

The GLOBIGNORE shell variable may be used to restrict the set of file names matching a pattern. If GLOBIGNORE is set, each matching file name that also matches one of the patterns in GLOBIGNORE is removed from the list of matches. The file names . and .. are always ignored, even when GLOBIGNORE is set. However, setting GLOBIGNORE has the effect of enabling the dotglob shell option, so all other file names beginning with a "." will match. To get the old behavior of ignoring file names beginning with a ".", make ".*" one of the patterns in GLOBIGNORE. The dotglob option is disabled when GLOBIGNORE is unset.

What are aliases?

An alias allows a string to be substituted for a word when it is used as the first word of a simple command. The shell maintains a list of aliases that may be set and unset with the alias and unalias built-in commands. Issue the alias without options to display a list of aliases known to the current shell.

```
franky: ~> alias

alias ..='cd ..'

alias ...='cd ../..'

alias ....='cd ../../..'

alias PAGER='less -r'

alias Txterm='export TERM=xterm'

alias XARGS='xargs -r'

alias cdrecord='cdrecord -dev 0,0,0 -speed=8'

alias e='vi'

alias egrep='grep -E'

alias ewformat='fdformat -n /dev/fd0u1743; ewfsck'

alias fgrep='grep -F'

alias ftp='ncftp -d15'

alias h='history 10'

alias fformat='fdformat /dev/fd0H1440'

alias j='jobs -l'
alias ksane='setterm -reset'

alias ls='ls -F --color=auto'

alias m='less'

alias md='mkdir'

alias od='od -Ax -ta -txC'

alias p='pstree -p'

alias ping='ping -vc1'

alias sb='ssh blubber'

alias sl='ls'

alias ss='ssh octarine'

alias tar='gtar'

alias tmp='cd /tmp'

alias unaliasall='unalias -a'

alias vi='eval `resize`;vi'

alias vt100='export TERM=vt100'

alias which='type'

alias xt='xterm -bg black -fg white &'

franky ~>
```

Aliases are useful for specifying the default version of a command that exists in several versions on your system, or to specify default options to a command. Another use for aliases is for correcting incorrect spelling.

The first word of each simple command, if unquoted, is checked to see if it has an alias. If so, that word is replaced by the text of the alias. The alias name and the replacement text may contain any valid shell input, including shell metacharacters, with the exception that the alias name may not contain "=". The first word of the replacement text is tested for aliases, but a word that is identical to an alias being expanded is not expanded a second time. This means that one may alias ls to ls -F, for instance, and Bash will not try to recursively expand the replacement text. If the last character of the alias value is a space or tab character, then the next command word following the alias is also checked for alias expansion.

Aliases are not expanded when the shell is not interactive, unless the expand_aliases option is set using the shopt shell built-in.

Creating and removing aliases

Aliases are created using the alias shell built-in. For permanent use, enter the alias in one of your shell initialization files; if you just enter the alias on the command line, it is only recognized within the current shell. franky ~> alias dh='df -h'

```
franky ~> dh

Filesystem          Size  Used Avail Use% Mounted on

/dev/hda7           1.3G  272M 1018M  22% /

/dev/hda1           121M  9.4M  105M   9% /boot

/dev/hda2            13G  8.7G  3.7G  70% /home

/dev/hda3            13G  5.3G  7.1G  43% /opt

none               243M     0  243M   0% /dev/shm

/dev/hda6           3.9G  3.2G  572M  85% /usr

/dev/hda5           5.2G  4.3G  725M  86% /var

franky ~> unalias dh

franky ~> dh

bash: dh: command not found

franky ~>
```

Bash always reads at least one complete line of input before executing any of the commands on that line. Aliases are expanded when a command is read, not when it is executed. Therefore, an alias definition appearing on the same line as another command does not take effect until the next line of input is read. The commands following the alias definition on that line are not affected by the new alias. This behavior is also an issue when functions are executed. Aliases are expanded when a function definition is read, not when the function is executed, because a function definition is itself a compound command. As a consequence, aliases defined in a function are not available until after that function is executed. To be safe, always put alias definitions on a separate line, and do not use alias in compound commands.

Aliases are not inherited by child processes. Bourne shell (sh) does not recognize aliases.

Activity:

- Example for brace expansion.
```
echo sp{el,il,al}l
```
- Example of tilde expansion
```
echo ~
cat ~oracle/scripting/script3.sh
```
- Example using backticks
```
echo $DATE
DATE=`date`
echo $DATE
DATE2=$(date)
```
- Example to use arithmetic expansion.
```
echo $((365*24))
echo $[365*24]
```
- Example of redirecting output.
```
echo "Redirecting data into file" > output.log
```
- Check list of aliases on the terminal.
```
alias
```
- Create and remove temporary alias and verify its functionality.
```
alias DT='date'
DT
unalias DT
DT
```
- Create permanent alias
```
vi .bash_profile
alias DT='date'
source .bash_profile
DT
```

2.4.Regular expressions

What are regular expressions?

A regular expression is a pattern that describes a set of strings. Regular expressions are constructed analogously to arithmetic expressions by using various operators to combine smaller expressions.

The fundamental building blocks are the regular expressions that match a single character. Most characters, including all letters and digits, are regular expressions that match themselves. Any metacharacter with special meaning may be quoted by preceding it with a backslash.

Regular expression metacharacters

A regular expression may be followed by one of several repetition operators (metacharacters):

Table 4-1. Regular expression operators

Operator	Effect
.	Matches any single character.
?	The preceding item is optional and will be matched, at most, once.
*	The preceding item will be matched zero or more times.
+	The preceding item will be matched one or more times.
{N}	The preceding item is matched exactly N times.
{N,}	The preceding item is matched N or more times.
{N,M}	The preceding item is matched at least N times, but not more than M times.
-	represents the range if it's not first or last in a list or the ending point of a range in a list.
^	Matches the empty string at the beginning of a line; also represents the characters not in the range of a list.
$	Matches the empty string at the end of a line.
\b	Matches the empty string at the edge of a word.
\B	Matches the empty string provided it's not at the edge of a word.
\<	Match the empty string at the beginning of word.
\>	Match the empty string at the end of word.

Two regular expressions may be concatenated; the resulting regular expression matches any string formed by concatenating two substrings that respectively match the concatenated subexpressions.

Two regular expressions may be joined by the infix operator "|"; the resulting regular expression matches any string matching either subexpression.

Repetition takes precedence over concatenation, which in turn takes precedence over alternation. A whole subexpression may be enclosed in parentheses to override these precedence rules.

Basic versus extended regular expressions

In basic regular expressions the metacharacters "?", "+", "{", "|", "(", and ")" lose their special meaning; instead use the backslashed versions "\?", "\+", "\{", "\|", "\(", and "\)".

Check in your system documentation whether commands using regular expressions support extended expressions.

2.5.grep examples

What is grep?

grep searches the input files for lines containing a match to a given pattern list. When it finds a match in a line, it copies the line to standard output (by default), or whatever other sort of output you have requested with options.

Though grep expects to do the matching on text, it has no limits on input line length other than available memory, and it can match arbitrary characters within a line. If the final byte of an input file is not a newline, grep silently supplies one. Since newline is also a separator for the list of patterns, there is no way to match newline characters in a text.

Grep and regular expressions

Note If you are not on Linux

We use GNU grep in these examples, which supports extended regular expressions. GNU grep is the default on Linux systems. If you are working on proprietary systems, check with the -V option which version you are using. GNU grep can be downloaded from http://gnu.org/directory/.

Character classes

A bracket expression is a list of characters enclosed by "[" and "]". It matches any single character in that list; if the first character of the list is the caret, "^", then it matches any character NOT in the list. For example, the regular expression "[0123456789]" matches any single digit.

Within a bracket expression, a range expression consists of two characters separated by a hyphen. It matches any single character that sorts between the two characters, inclusive, using the locale's collating sequence and character set. For example, in the default C locale, "[a-d]" is equivalent to "[abcd]". Many locales sort characters in dictionary order, and in these locales "[a-d]" is typically not equivalent to "[abcd]"; it might be equivalent to "[aBbCcDd]", for example. To obtain the traditional interpretation of bracket expressions, you can use the C locale by setting the LC_ALL environment variable to the value "C".

Finally, certain named classes of characters are predefined within bracket expressions. See the grep man or info pages for more information about these predefined expressions.

Wildcards

Use the "." for a single character match. If you want to get a list of all five-character English dictionary words starting with "c" and ending in "h" (handy for solving crosswords): cathy ~> grep '\<c...h\>' /usr/share/dict/words

If you want to display lines containing the literal dot character, use the -F option to grep.

For matching multiple characters, use the asterisk. This example selects all words starting with "c" and ending in "h" from the system's dictionary: cathy ~> grep '\<c.*h\>' /usr/share/dict/words

Activity:

- Use grep to filter an expression from the file.
  ```
  grep root /etc/passwd
  ```
- Grep expression from file with line number.
  ```
  grep -n root /etc/passwd
  ```
- Grep everything from a file other than expression.
  ```
  grep -v root /etc/passwd
  ```
- Grep expression from group of files in a directory.
  ```
  grep -i bash ./*.sh
  ```
- Exclude an expression in grep from group of files.
  ```
  grep -i bash ./*.sh | grep -v bin
  ```
- Grep from a file with lines starting with regular expression.
  ```
  grep ^root /etc/passwd
  ```
- Grep from a file with lines ending with regular expression.
  ```
  grep bash$ /etc/passwd
  ```
- Grep regular expression with blank spaces on either side.
  ```
  grep -w / /etc/passwd
  ```

- Grep individual characters in a regular expression.
  ```
  grep [yf] /etc/group
  ```
- Grep for regular expression starting and ending with a character and fixed number of characters in between.
  ```
  grep '\<c...h\>' /usr/share/dict/words
  ```
- Grep for regular expression starting and ending with a character and floating number of characters in between.
  ```
  grep '\<c.*h\>' /usr/share/dict/words
  ```

2.6.Pattern matching

Character ranges

Apart from grep and regular expressions, there's a good deal of pattern matching that you can do directly in the shell, without having to use an external program.

As you already know, the asterisk (*) and the question mark (?) match any string or any single character, respectively. Quote these special characters to match them literally.

Character classes

Character classes can be specified within the square braces, using the syntax [:CLASS:], where CLASS is defined in the POSIX standard and has one of the values

"alnum", "alpha", "ascii", "blank", "cntrl", "digit", "graph", "lower", "print", "punct", "space", "upper", "word" or "xdigit".

Activity:

- Filter files with range of characters in the location.
  ```
  cd /etc
  ls -ld [a-cx-z]*
  ```
- Filer files with digits in the name or upper case character in the name.
  ```
  ls -ld [[:digit:]]*
  ls -ld [[:upper:]]*
  ```

3. GNU

3.1.Interactive editing

What is sed?

A Stream EDitor is used to perform basic transformations on text read from a file or a pipe. The result is sent to standard output. The syntax for the sed command has no output file specification, but results can be saved to a file using output redirection. The editor does not modify the original input.

What distinguishes sed from other editors, such as vi and ed, is its ability to filter text that it gets from a pipeline feed. You do not need to interact with the editor while it is running; that is why sed is sometimes called a batch editor. This feature allows use of editing commands in scripts, greatly easing repetitive editing tasks. When facing replacement of text in a large number of files, sed is a great help.

sed commands

The sed program can perform text pattern substitutions and deletions using regular expressions, like the ones used with the grep command.

The editing commands are similar to the ones used in the vi editor:

Sed editing commands

Command	Result
a\	Append text below current line.
c\	Change text in the current line with new text.
d	Delete text.
i\	Insert text above current line.
p	Print text.
r	Read a file.
s	Search and replace text.
w	Write to a file.

Apart from editing commands, you can give options to sed. An overview is in the table below:

Sed options

Option Effect

-e SCRIPT Add the commands in SCRIPT to the set of commands to be run while processing the input.

-f Add the commands contained in the file SCRIPT-FILE to the set of commands to be run while processing the input.

-n Silent mode.

-V Print version information and exit.

Printing lines containing a pattern

This is something you can do with grep, of course, but you can't do a "find and replace" using that command. This is just to get you started.

Activity:

- Create a file with some content in it.

```
vi file.txt
This is the first line of an example text.
It is a text with erors.
Lots of erors.
So much erors, all these erors are making me sick.
This is a line not containing any errors.
This is the last line.
```

- Use sed now to search a pattern called erors.

```
sed '/erors/p' file.txt
```

- Remove repeated outputs using sed while searching for a pattern.

```
sed -n '/erors/p' file.txt
```

Exclude lines containing a pattern

Activity:

- Use sed now to exclude a pattern called erors.

```
sed -n '/erors/d' file.txt
```

Range of lines

Activity:

- Remove the lines from 2 to 4 from the file in the output.
  ```
  sed '2,4d' file.txt
  ```
- Print the lines starting from 3rd line until end.
  ```
  sed '3,$d' file.txt
  ```
- Print from one pattern until next pattern in a file.
  ```
  sed -n '/a text/,/This/p' file.txt
  ```

Find and replace

Activity:

- Find pattern erors and replace with errors.
  ```
  sed 's/erors/errors/' file.txt - only first occurrences
  in the line will be replaced.
  ```
- To replace all the patterns in all the lines.
  ```
  sed 's/erors/errors/g' file.txt
  ```
- To insert string at beginning of every line.
  ```
  sed 's/^/> /' file.txt
  ```
- To insert pattern at ending of every line.
  ```
  sed 's/$/EOL/' file.txt
  ```

3.2.Non interactive editing

Reading sed commands from a file

Multiple sed commands can be put in a file and executed using the -f option. When creating such a file, make sure that:

- No trailing white spaces exist at the end of lines.
- No quotes are used.
- When entering text to add or replace, all except the last line end in a backslash.
- AWK programming

Activity:

- Redirect the output of the sed command to the different file.
  ```
  sed 's/erors/errors/' file.txt > corrected.txt
  rm file.txt
  ```

3.3.AWK programming

What is gawk?

Gawk is the GNU version of the commonly available UNIX **awk** program, another popular stream editor. Since the **awk** program is often just a link to gawk, we will refer to it as **awk**.

The basic function of **awk** is to search files for lines or other text units containing one or more patterns. When a line matches one of the patterns, special actions are performed on that line.

Programs in awk are different from programs in most other languages, because awk programs are "data-driven": you describe the data you want to work with and then what to do when you find it. Most other languages are "procedural." You have to describe, in great detail, every step the program is to take. When working with procedural languages, it is usually much harder to clearly describe the data your program will process. For this reason, awk programs are often refreshingly easy to read and write.

Gawk commands

When you run awk, you specify an awk program that tells awk what to do. The program consists of a series of rules. (It may also contain function definitions, loops, conditions and other programming constructs, advanced features that we will ignore for now.) Each rule specifies one pattern to search for and one action to perform upon finding the pattern.

There are several ways to run awk. If the program is short, it is easiest to run it on the command line:

```
awk PROGRAM inputfile(s)
```

If multiple changes have to be made, possibly regularly and on multiple files, it is easier to put the awk commands in a script. This is read like this:

```
awk -f PROGRAM-FILE inputfile(s)
```

3.4.AWK print program

Printing selected fields

The **print** command in awk outputs selected data from the input file.

When awk reads a line of a file, it divides the line in fields based on the specified input field separator, FS, which is an awk variable. This variable is predefined to be one or more spaces or tabs.

The variables `$1, $2, $3, ..., $N` hold the values of the first, second, third until the last field of an input line. The variable `$0 (zero)` holds the value of the entire line. This is depicted in the image below, where we see six colums in the output of the df command:

Activity:

- Understand the column positional numbers

  ```
  df -h
  ```
- Use awk programming to print specified columns.

  ```
  ls -l | awk '{ print $5 $9 }'
  ```

Formatting fields

Without formatting, using only the output separator, the output looks rather poor. Inserting a couple of tabs and a string to indicate what output this is will make it look a lot better:

Activity:

- Print the fields with formatted output.

  ```
  ls -ldh * | grep -v total | awk '{ print "Size is " $5 "
  bytes for " $9 }'
  ```
- To format the output from df command.

  ```
  df -h | sort -rnk 5 | head -3 | awk '{ print "Partition "
  $6 "\t: " $5 " full!" }'
  ```

Formatting characters for gawk

Sequence	Meaning
\a	Bell character
\n	Newline character
\t	Tab

The print command and regular expressions

A regular expression can be used as a pattern by enclosing it in slashes. The regular expression is then tested against the entire text of each record. The syntax is as follows:

```
awk 'EXPRESSION { PROGRAM }' file(s)
```

Activity:

- The following example displays only local disk device information, networked file systems are not shown:

  ```
  df -h | awk '/dev\/hd/ { print $6 "\t: " $5 }'
  ```

- Below another example where we search the /etc directory for files ending in ".conf" and starting with either "a" or "x", using extended regular expressions:
  ```
  ls -l | awk '/\<(a|x).*\.conf$/ { print $9 }'
  ```

Special patterns

In order to precede output with comments, use the BEGIN statement.

```
ls -l | awk 'BEGIN { print "Files found:\n" } /\<[a|x].*\.conf$/
{ print $9 }'
```

The END statement can be added for inserting text after the entire input is processed.

```
ls -l | awk '/\<[a|x].*\.conf$/ { print $9 } END { print "Can I
do anything else for you?" }'
```

Gawk scripts

As commands tend to get a little longer, you might want to put them in a script, so they are reusable. An awk script contains awk statements defining patterns and actions.

As an illustration, we will build a report that displays our most loaded partitions.

Activity:

- Write AWK script.
  ```
  cat diskrep.awk
  BEGIN { print "*** WARNING WARNING WARNING ***" }
  /  \<[8|9][0-9]%  / { print "Partition " $6 "\t: " $5 "
  full!" }
  END { print "*** Give money for new disks URGENTLY! ***"
  }
  ```

- Use this script now for df command.
  ```
  df -h | awk -f diskrep.awk
  ```

3.5.AWK variables

As awk is processing the input file, it uses several variables. Some are editable, some are read-only.

The input field separator

The field separator, which is either a single character or a regular expression, controls the way awk splits up an input record into fields. The input record is scanned for character sequences that match the separator definition; the fields themselves are the text between the matches.

The field separator is represented by the built-in variable FS. Note that this is something different from the IFS variable used by POSIX-compliant shells.

The value of the field separator variable can be changed in the awk program with the assignment operator =. Often the right time to do this is at the beginning of execution before any input has been processed, so that the very first record is read with the proper separator. To do this, use the special BEGIN pattern.

Activity:

- In the example below, we build a command that displays all the users on your system with a description:

```
awk 'BEGIN {FS=":"} { print $1 "\t" $5 }' /etc/passwd
```

The default input field separator is one or more whitespaces or tabs.

The output field separator

Fields are normally separated by spaces in the output. This becomes apparent when you use the correct syntax for the print command, where arguments are separated by commas:

Activity:

- Let us create a file with content as below.

```
cat test
record1         data1
record2         data2
```

- Let us print output without field separator.

```
awk '{ print $1 $2}' test
```

- Let us print output with field separator ,

```
awk '{ print $1, $2}' test
```

The output record separator

The output from an entire print statement is called an output record. Each print command results in one output record, and then outputs a string called the output record separator, ORS. The default value for this variable is "\n", a newline character. Thus, each print statement generates a separate line.

Activity:

- To change the way output fields and records are separated, assign new values to OFS and ORS:

```
awk 'BEGIN {OFS=";" ; ORS="\n-->\n" }{ print $1,$2}' test
```

- If the value of ORS does not contain a newline, the program's output is run together on a single line.

The number of records

The built-in NR holds the number of records that are processed. It is incremented after reading a new input line. You can use it at the end to count the total number of records, or in each output record.

Activity:

- Write an awk script as below.

```
cat processed.awk
BEGIN { OFS="-" ; ORS="\n--> done\n" }
{ print "Record number " NR ":\t" $1,$2 }
END { print "Number of records processed: " NR }
```

- Let us run this on test file.

```
awk -f processed.awk test
```

User defined variables

Apart from the built-in variables, you can define your own. When awk encounters a reference to a variable which does not exist (which is not predefined), the variable is created and initialized to a null string. For all subsequent references, the value of the variable is whatever value was assigned last. Variables can be a string or a numeric value. Content of input fields can also be assigned to variables.

Values can be assigned directly using the = operator, or you can use the current value of the variable in combination with other operators.

Activity:

- Create a file with content in it as.

```
cat revenues
20021009        20021013        consultancy     BigComp
2500
20021015        20021020        training        EduComp
2000
20021112        20021123        appdev          SmartComp
10000
20021204        20021215        training        EduComp
5000
```

- Write an awk script now.

```
cat total.awk
{ total=total + $5 }
{ print "Send bill for " $5 " dollar to " $4 }
END { print "---------------------------------\nTotal
revenue: " total }
```

- Run the awk script now on this file.

```
awk -f total.awk revenues
```

4. CONDITIONS & INTERACTIVE SCRIPTS

4.1.IF clause

General

At times you need to specify different courses of action to be taken in a shell script, depending on the success or failure of a command. The if construction allows you to specify such conditions.

The most compact syntax of the if command is:

```
if TEST-COMMANDS; then CONSEQUENT-COMMANDS; fi
```

The TEST-COMMAND list is executed, and if its return status is zero, the CONSEQUENT-COMMANDS list is executed. The return status is the exit status of the last command executed, or zero if no condition tested true.

The TEST-COMMAND often involves numerical or string comparison tests, but it can also be any command that returns a status of zero when it succeeds and some other status when it fails. Unary expressions are often used to examine the status of a file. If the FILE argument to one of the primaries is of the form /dev/fd/N, then file descriptor “N” is checked. stdin, stdout and stderr and their respective file descriptors may also be used for tests.

Expressions used with if

The table below contains an overview of the so-called “primaries” that make up the TEST-COMMAND command or list of commands. These primaries are put between square brackets to indicate the test of a conditional expression.

Primary expressions

Primary	Meaning
[-a FILE]	True if FILE exists.
[-b FILE]	True if FILE exists and is a block-special file.
[-c FILE]	True if FILE exists and is a character-special file.
[-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.
[-g FILE]	True if FILE exists and its SGID bit is set.
[-h FILE]	True if FILE exists and is a symbolic link.
[-k FILE]	True if FILE exists and its sticky bit is set.
[-p FILE]	True if FILE exists and is a named pipe (FIFO).

Primary	Meaning
[-r FILE]	True if FILE exists and is readable.
[-s FILE]	True if FILE exists and has a size greater than zero.
[-t FD]	True if file descriptor FD is open and refers to a terminal.
[-u FILE]	True if FILE exists and its SUID (set user ID) bit is set.
[-w FILE]	True if FILE exists and is writable.
[-x FILE]	True if FILE exists and is executable.
[-O FILE]	True if FILE exists and is owned by the effective user ID.
[-G FILE]	True if FILE exists and is owned by the effective group ID.
[-L FILE]	True if FILE exists and is a symbolic link.
[-N FILE]	True if FILE exists and has been modified since it was last read.
[-S FILE]	True if FILE exists and is a socket.
[FILE1 -nt FILE2]	True if FILE1 has been changed more recently than FILE2, or if FILE1 exists and FILE2 does not.
[FILE1 -ot FILE2]	True if FILE1 is older than FILE2, or if FILE2 exists and FILE1 does not.
[FILE1 -ef FILE2]	True if FILE1 and FILE2 refer to the same device and inode numbers.
[-o OPTIONNAME]	True if shell option “OPTIONNAME” is enabled.
[-z STRING]	True if the length of “STRING” is zero.
[-n STRING] or [STRING]	True if the length of “STRING” is non-zero.
[STRING1 == STRING2]	True if the strings are equal. “=” may be used instead of “==” for strict POSIX compliance.
[STRING1 != STRING2]	True if the strings are not equal.
[STRING1 < STRING2]	True if “STRING1” sorts before “STRING2” lexicographically in the current locale.
[STRING1 > STRING2]	True if “STRING1” sorts after “STRING2” lexicographically in the current locale.
[ARG1 OP ARG2]	“OP” is one of -eq, -ne, -lt, -le, -gt or -ge. These arithmetic binary operators return true if “ARG1” is equal to, not equal to, less than, less than or equal to, greater than, or greater than or equal to “ARG2”, respectively. “ARG1” and “ARG2” are integers.

Combining expressions

Operation	Effect
[! 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.

The [(or test) built-in evaluates conditional expressions using a set of rules based on the number of arguments. More information about this subject can be found in the Bash documentation. Just like the if is closed with fi, the opening square bracket should be closed after the conditions have been listed.

Commands following the then statement

The CONSEQUENT-COMMANDS list that follows the then statement can be any valid UNIX command, any executable program, any executable shell script or any shell statement, with the exception of the closing fi. It is important to remember that the then and fi are considered to be separated statements in the shell. Therefore, when issued on the command line, they are separated by a semi-colon.

In a script, the different parts of the if statement are usually well-separated.

Activity:

- If clause to check file is existing.

```
vi checkfile.sh
#!/bin/bash
echo "This scripts checks the existence of the messages
file."
echo "Checking..."
if [ -f /var/log/messages ]
then
   echo "/var/log/messages exists."
else
   echo "/var/log/messages DO NOT exists."
fi
echo
echo "...done. "
chmod +x checkfile.sh
./checkfile.sh
```

- Check if the file is protected against accidental overwriting using redirection.

```
if [ -o noclobber ]
```

```
  then
   echo "Your files are protected against accidental
overwriting using redirection."
fi
```

- Testing exit status with if.

```
if [ $? -eq 0 ]
then
   echo "Earlier command executed successfully"
else
   echo "Earlier command DID NOT execute successfully"
fi
```

- Testing with a command and then $?

```
grep oracle /etc/passwd
if [ $? -eq 0 ]
then
   echo "Earlier command executed successfully"
else
   echo "Earlier command DID NOT execute successfully"
fi
grep oraC /etc/passwd
if [ $? -eq 0 ]
then
   echo "Earlier command executed successfully"
else
   echo "Earlier command DID NOT execute successfully"
fi
```

- Testing this user existence within if clause.

```
if ! grep oracle /etc/passwd
then
   echo "User doesn’t exists"
else
   echo "User exists"
fi
```

- Working with numeric.

```
num=`wc -l revenues`
if [ "$num" -gt "4" ]
then echo ; echo "Condition is true"
else; echo "Condition is false"
```

```
fi
```

- Working with string comparisons.

```
if [ "$(whoami)" != 'root' ]
then
        echo "You have no permission to run $0 as non-
root user."
fi
```

4.2.Advanced IF usage

Checking command line arguments

Instead of setting a variable and then executing a script, it is frequently more elegant to put the values for the variables on the command line.

We use the positional parameters $1, $2, ..., $N for this purpose. $# refers to the number of command line arguments. $0 refers to the name of the script.

Activity:

- Example with two arguments we passed.

```
vi weight.sh
#!/bin/bash
# This script prints a message about your weight if you
give it your
# weight in kilos and height in centimeters.
weight="$1"
height="$2"
idealweight=$[$height - 110]
if [ $weight -le $idealweight ] ; then
  echo "You should eat a bit more fat."
else
  echo "You should eat a bit more fruit."
fi
```

- Execute this script with debug mode to verify the process flow.

```
bash -x weight.sh 55 169
```

- Let us edit this script to also include verifying number of arguments passed.

```
vi weight.sh
#!/bin/bash
# This script prints a message about your weight if you
give it your
# weight in kilos and height in centimeters.
```

```
if [ ! $# == 2 ]; then
  echo "Usage: $0 weight_in_kilos length_in_centimeters"
  exit
fi

weight="$1"
height="$2"
idealweight=$[$height - 110]
if [ $weight -le $idealweight ] ; then
  echo "You should eat a bit more fat."
else
  echo "You should eat a bit more fruit."
fi
```

if/then/elif/else constructs

General

This is the full form of the if statement:

```
if TEST-COMMANDS; then

CONSEQUENT-COMMANDS;

elif MORE-TEST-COMMANDS; then

MORE-CONSEQUENT-COMMANDS;

else ALTERNATE-CONSEQUENT-COMMANDS;

fi
```

The TEST-COMMANDS list is executed, and if its return status is zero, the CONSEQUENT-COMMANDS list is executed. If TEST-COMMANDS returns a non-zero status, each elif list is executed in turn, and if its exit status is zero, the corresponding MORE-CONSEQUENT-COMMANDS is executed and the command completes. If else is followed by an ALTERNATE-CONSEQUENT-COMMANDS list, and the final command in the final if or elif clause has a non-zero exit status, then ALTERNATE-CONSEQUENT-COMMANDS is executed. The return status is the exit status of the last command executed, or zero if no condition tested true.

Activity:

- Example with complex variable and simple if.
  vi disktest.sh

```
#!/bin/bash
# This script does a very simple test for checking disk
space.
space=`df -h | awk '{print $5}' | grep % | grep -v Use |
sort -n | tail -1 | cut -d "%" -f1 -`
alertvalue="80"
if [ "$space" -ge "$alertvalue" ]; then
  echo "At least one of my disks is nearly full!"
else
  echo "Disk space normal"
fi
```

- Example with nested if clauses.

```
vi testleap.sh
#!/bin/bash
# This script will test if we're in a leap year or not.
year=`date +%Y`
if [ $[$year % 400] -eq "0" ]; then
  echo "This is a leap year.  February has 29 days."
elif [ $[$year % 4] -eq 0 ]; then
        if [ $[$year % 100] -ne 0 ]; then
          echo "This is a leap year, February has 29
days."
        else
          echo "This is not a leap year.  February has 28
days."
        fi
else
  echo "This is not a leap year.  February has 28 days."
fi
```

- Using exit numbers in the if clauses.

```
#!/bin/bash
#This script lets you present different menus to Tux.  He
will only be happy
# when given a fish.  We've also added a dolphin and
(presumably) a camel.
menu="$1"
animal="$2"
if [ "$menu" == "fish" ]; then
  if [ "$animal" == "penguin" ]; then
    echo "Hmmmmmm fish... Tux happy!"
  elif [ "$animal" == "dolphin" ]; then
```

```
      echo "Pweetpeettreetppeterdepweet!"
    else
      echo "*prrrrrrrt*"
    fi
  else
    if [ "$animal" == "penguin" ]; then
      echo "Tux don't like that.  Tux wants fish!"
      exit 1
    elif [ "$animal" == "dolphin" ]; then
      echo "Pweepwishpeeterdepweet!"
      exit 2
    else
      echo "Will you read this sign?!"
      exit 3
    fi
  fi
```

4.3.CASE statements

Simplified conditions

Nested if statements might be nice, but as soon as you are confronted with a couple of different possible actions to take, they tend to confuse. For the more complex conditionals, use the case syntax:

```
case EXPRESSION in CASE1) COMMAND-LIST;; CASE2) COMMAND-LIST;;
... CASEN) COMMAND-LIST;; esac
```

Each case is an expression matching a pattern. The commands in the COMMAND-LIST for the first match are executed. The "|" symbol is used for separating multiple patterns, and the ")" operator terminates a pattern list. Each case plus its according commands are called a clause. Each clause must be terminated with ";;". Each case statement is ended with the esac statement.

Activity:

- In the example, we demonstrate use of cases for sending a more selective warning message with the disktest.sh script:

```
#!/bin/bash
# This script does a very simple test for checking disk
space.
space=`df -h | awk '{print $5}' | grep % | grep -v Use |
sort -n | tail -1 | cut -d "%" -f1 -`
```

```
case $space in
[1-6]*)
  Message="All is quiet."
  ;;
[7-8]*)
  Message="Start thinking about cleaning out some stuff.
There's a partition that is $space % full."
  ;;
9[1-8])
  Message="Better hurry with that new disk...  One
partition is $space % full."
  ;;
99)
  Message="I'm drowning here!  There's a partition at
$space %!"
  ;;
*)
  Message="I seem to be running with an nonexistent
amount of disk space..."
  ;;
esac

echo $Message
```

- Case on string value comparisions.

```
case "$1" in
        start)
            start
            ;;
        stop)
            stop
            ;;
        status)
            status anacron
            ;;
        restart)
            stop
            start
            ;;
        condrestart)
            if test "x`pidof anacron`" != x; then
                stop
                start
            fi
            ;;
        *)
            echo $"Usage: $0
{start|stop|restart|condrestart|status}"
            exit 1
esac
```

4.4.Display user messages

Interactive or not?

Some scripts run without any interaction from the user at all. Advantages of non-interactive scripts include:

- The script runs in a predictable way every time.
- The script can run in the background.

Many scripts, however, require input from the user, or give output to the user as the script is running. The advantages of interactive scripts are, among others:

- More flexible scripts can be built.
- Users can customize the script as it runs or make it behave in different ways.
- The script can report its progress as it runs.

When writing interactive scripts, never hold back on comments. A script that prints appropriate messages is much more user-friendly and can be more easily debugged. A script might do a perfect job, but you will get a whole lot of support calls if it does not inform the user about what it is doing. So include messages that tell the user to wait for output because a calculation is being done. If possible, try to give an indication of how long the user will have to wait. If the waiting should regularly take a long time when executing a certain task, you might want to consider integrating some processing indication in the output of your script.

When prompting the user for input, it is also better to give too much than too little information about the kind of data to be entered. This applies to the checking of arguments and the accompanying usage message as well.

Bash has the **echo and printf** commands to provide comments for users, and although you should be familiar with at least the use of echo by now, we will discuss some more examples in the next sections.

Using the echo built-in command

The echo built-in command outputs its arguments, separated by spaces and terminated with a newline character. The return status is always zero. echo takes a couple of options:

-e: interprets backslash-escaped characters.

-n: suppresses the trailing newline.

Activity:

- Let us write two scripts in a better format than earlier with user messages using echo.

-
  ```
  #!/bin/bash
  # This script lets you present different menus to Tux.
  He will only be happy
  # when given a fish.  To make it more fun, we added a
  couple more animals.
  if [ "$menu" == "fish" ]; then
    if [ "$animal" == "penguin" ]; then
      echo -e "Hmmmmmm fish... Tux happy!\n"
    elif [ "$animal" == "dolphin" ]; then
      echo -e "\a\a\aPweetpeettreetppeterdepweet!\a\a\a\n"
    else
      echo -e "*prrrrrrrt*\n"
    fi
  else
    if [ "$animal" == "penguin" ]; then
      echo -e "Tux don't like that.  Tux wants fish!\n"
      exit 1
    elif [ "$animal" == "dolphin" ]; then
      echo -e "\a\a\a\a\a\aPweepwishpeeterdepweet!\a\a\a"
      exit 2
    else
      echo -e "Will you read this sign?!  Don't feed the
  "$animal"s!\n"
      exit 3
    fi
  fi
  ```

- Example which includes case conditional statements.
  ```
  #!/bin/bash
  # This script acts upon the exit status given by
  penguin.sh
  ```

```
if [ "$#" != "2" ]; then
  echo -e "Usage of the feed script:\t$0 food-on-menu
animal-name\n"
  exit 1
else
  export menu="$1"
  export animal="$2"
  echo -e "Feeding $menu to $animal...\n"
  feed="/home/oracle/scripting/penguin.sh"
$feed
result="$?"
  echo -e "Done feeding.\n"
case "$result" in
  1)
    echo -e "Guard: \"You'd better give'm a fish, less
they get violent...\"\n"
    ;;
  2)
    echo -e "Guard: \"No wonder they flee our
planet...\"\n"
    ;;
  3)
    echo -e "Guard: \"Buy the food that the Zoo provides
at the entry, you ***\"\n"
    echo -e "Guard: \"You want to poison them, do
you?\"\n"
    ;;
  *)
    echo -e "Guard: \"Don't forget the guide!\"\n"
    ;;
  esac
fi

echo "Leaving..."
echo -e "\a\a\aThanks for visiting the Zoo, hope to see
you again soon!\n"
```

- Run feed.sh to re-run penguin.sh script.

```
./feed.sh apple camel
```

Escape sequences used by the echo command

Sequence	Meaning
\a	Alert (bell).
\b	Backspace.
\c	Suppress trailing newline.
\e	Escape.
\f	Form feed.
\n	Newline.
\r	Carriage return.
\t	Horizontal tab.
\v	Vertical tab.
\\	Backslash.
\0NNN	The eight-bit character whose value is the octal value NNN (zero to three octal digits).
\NNN	The eight-bit character whose value is the octal value NNN (one to three octal digits).
\xHH	The eight-bit character whose value is the hexadecimal value (one or two hexadecimal digits).

4.5.User input

Using the read built-in command

The read built-in command is the counterpart of the echo and printf commands. The syntax of the read command is as follows:

```
read [options] NAME1 NAME2 ... NAMEN
```

One line is read from the standard input, or from the file descriptor supplied as an argument to the -u option. The first word of the line is assigned to the first name, NAME1, the second word to the second name, and so on, with leftover words and their intervening separators assigned to the last name, NAMEN. If there are fewer words read from the input stream than there are names, the remaining names are assigned empty values.

The characters in the value of the IFS variable are used to split the input line into words or tokens; see the section called "Word splitting". The backslash character may be used to remove any special meaning for the next character read and for line continuation.

If no names are supplied, the line read is assigned to the variable REPLY.

The return code of the read command is zero, unless an end-of-file character is encountered, if read times out or if an invalid file descriptor is supplied as the argument to the -u option.

Options to the read built-in

Option	Meaning
-a `ANAME`	The words are assigned to sequential indexes of the array variable `ANAME`, starting at 0. All elements are removed from `ANAME` before the assignment. Other `NAME` arguments are ignored.
-d `DELIM`	The first character of `DELIM` is used to terminate the input line, rather than newline.
-e	**readline** is used to obtain the line.
-n `NCHARS`	**read** returns after reading `NCHARS` characters rather than waiting for a complete line of input.
-p `PROMPT`	Display `PROMPT`, without a trailing newline, before attempting to read any input. The prompt is displayed only if input is coming from a terminal.
-r	If this option is given, backslash does not act as an escape character. The backslash is considered to be part of the line. In particular, a backslash-newline pair may not be used as a line continuation.
-s	Silent mode. If input is coming from a terminal, characters are not echoed.
-t `TIMEOUT`	Cause **read** to time out and return failure if a complete line of input is not read within `TIMEOUT` seconds. This option has no effect if **read** is not reading input from the terminal or from a pipe.
-u `FD`	Read input from file descriptor `FD`.

Activity:

- Writing a better leaptest.sh script.

```
#!/bin/bash
# This script will test if you have given a leap year or
not.
year="$1"
if (( ("$year" % 400) == "0" )) || (( ("$year" % 4 ==
"0") && ("$year" % 100 != "0") )); then
echo "$year is a leap year."
else
  echo "This is not a leap year."
fi
```

- Writing a script to prompt user input.

```
vi friends.sh
#!/bin/bash

# This is a program that keeps your address book up to
date.

friends="/home/oracle/scripting/friends"

echo "Hello, "$USER".  This script will register you in
Kumar's friends database."

echo -n "Enter your name and press [ENTER]: "
read name
echo -n "Enter your gender and press [ENTER]: "
read -n 1 gender
echo

grep -i "$name" "$friends"

if  [ $? == 0 ]; then
  echo "You are already registered, quitting."
  exit 1
elif [ "$gender" == "m" ]; then
  echo "$name $gender" >> "$friends"
  echo "You are added to Kumar's friends list."
  exit 1
else
  echo -n "How old are you? "
```

```
    read age
    if [ $age -lt 25 ]; then
      echo -n "Which colour of hair do you have? "
      read colour
      echo "$name $age $colour" >> "$friends"
      echo "You are added to Kumar's friends list.  Thank
  you so much!"
    else
      echo "You are added to Kumar's friends list."
      exit 1
    fi
  fi
```

5. REPETITIVE TASKS & VARIABLES

5.1.FOR loop

How does it work?

The for loop is the first of the three shell looping constructs. This loop allows for specification of a list of values. A list of commands is executed for each value in the list.

The syntax for this loop is:

```
for NAME [in LIST]; do COMMANDS; done
```

The return status is the exit status of the last command that executes. If no commands are executed because LIST does not expand to any items, the return status is zero.

NAME can be any variable name, although `i` is used very often. LIST can be any list of words, strings or numbers, which can be literal or generated by any command. The COMMANDS to execute can also be any operating system commands, script, program or shell statement. The first time through the loop, NAME is set to the first item in LIST. The second time, its value is set to the second item in the list, and so on. The loop terminates when NAME has taken on each of the values from LIST and no items are left in LIST.

Activity:

- Using command substitution for specifying LIST items

```
touch file1.txt file2.txt file3.txt
echo file1.txt > list
echo file2.txt >> list
echo file3.txt >> list
for i in `cat list`
do
cp "$i" "$i".bak
done
```

- Using for loop for fixed number of iterations.

```
for i in {1..5}
do
echo $i
done
for (( i=1; i<=5; i++ ))
do
echo $i
done
```

5.2.WHILE loop

What is it?

The while construct allows for repetitive execution of a list of commands, as long as the command controlling the while loop executes successfully (exit status of zero). The syntax is:

```
while CONTROL-COMMAND; do CONSEQUENT-COMMANDS; done
```

CONTROL-COMMAND can be any command(s) that can exit with a success or failure status. The CONSEQUENT-COMMANDS can be any program, script or shell construct.

As soon as the CONTROL-COMMAND fails, the loop exits. In a script, the command following the done statement is executed.

The return status is the exit status of the last CONSEQUENT-COMMANDS command, or zero if none was executed.

Activity:

- Simple example using while.

```
#!/bin/bash
# This script opens 4 terminal windows.
i="0"
while [ $i -lt 4 ]
do
xterm &
i=$[$i+1]
done
for i in {0..3}
do
xterm &
done
```

- Nested while loops

```
#!/bin/bash
# This script copies files from my scripting into the
backup directory.
# A new directory is created every hour.

PICSDIR=/home/oracle/scripting
WEBDIR=/home/oracle/backup
```

```
while true; do
  DATE=`date +%Y%m%d`
  HOUR=`date +%H`
  mkdir $WEBDIR/"$DATE"

  while [ $HOUR -ne "00" ]; do
       DESTDIR=$WEBDIR/"$DATE"/"$HOUR"
       mkdir -p "$DESTDIR"
       cp $PICSDIR/* "$DESTDIR"/
       sleep 3600
       HOUR=`date +%H`
  done
done
```

- Using keyboard input to control the while loop

```
vi /home/oracle/scripting/fortune
#!/bin/bash
echo "Received your input as $1"

chmod +x /home/oracle/scripting/fortune

#!/bin/bash
# This script runs the executable
FORTUNE=/home/oracle/scripting/fortune
while true; do
echo "On which topic do you want advice?"
cat << LIST
politics
Startrek
kernelnewbies
sports
bofh-excuses
magic
love
literature
drugs
education
LIST

echo
echo -n "Make your choice: "
read topic
echo
```

```
echo "Free advice on the topic of $topic: "
echo
$FORTUNE $topic
echo
done
```

- Calculating an average

```
#!/bin/bash

# Calculate the average of a series of numbers.

SCORE="0"
AVERAGE="0"
SUM="0"
NUM="0"

while true; do

  echo -n "Enter your score [0-1000] ('q' for quit): ";
read SCORE;
echo -n "Enter your number of attempts : "; read NUM;

  if (("$SCORE" < "0"))  || (("$SCORE" > "1000")); then
    echo "Be serious.  Common, try again: "
  elif [ "$SCORE" == "q" ]; then
    echo "Average rating: $AVERAGE%."
    break
  else
    SUM=$[$SUM + $SCORE]
    AVERAGE=$[$SUM / $NUM]
  echo "Your average is : $AVERAGE"
  break
  fi

done
echo "Exiting."
```

5.3.UNTIL loop

What is it?

The until loop is very similar to the while loop, except that the loop executes until the TEST-COMMAND executes successfully. As long as this command fails, the loop continues. The syntax is the same as for the while loop:

```
until TEST-COMMAND; do CONSEQUENT-COMMANDS; done
```

The return status is the exit status of the last command executed in the CONSEQUENT-COMMANDS list, or zero if none was executed. TEST-COMMAND can, again, be any command that can exit with a success or failure status, and CONSEQUENT-COMMANDS can be any UNIX command, script or shell construct.

As we already explained previously, the ";" may be replaced with one or more newlines wherever it appears.

Activity:

- Simple example using UNTIL.

```
#!/bin/bash

# This script copies files from my homedirectory into the
webserver directory.
# A new directory is created every hour.
# If the pics are taking up too much space, the oldest
are removed.

PICSDIR=/home/oracle/scripting
WEBDIR=/home/oracle/backup

while true; do
  DISKFUL=$(df -h $WEBDIR | grep -v File | awk '{print
$5 }' | cut -d "%" -f1 -)

  until [ $DISKFUL -ge "90" ]; do

            DATE=`date +%Y%m%d`
            HOUR=`date +%H`
            mkdir $WEBDIR/"$DATE"

            while [ $HOUR -ne "00" ]; do
                 DESTDIR=$WEBDIR/"$DATE"/"$HOUR"
                 mkdir "$DESTDIR"
                 mv $PICDIR/*.jpg "$DESTDIR"/
                 sleep 3600
                 HOUR=`date +%H`
            done
```

```
   DISKFULL=$(df -h $WEBDIR | grep -v File | awk '{ print
$5 }' | cut -d "%" -f1 -)
   done

   TOREMOVE=$(find $WEBDIR -type d -a -mtime +30)
   for i in $TOREMOVE; do
        rm -rf "$i";
   done

done
```

5.4.I/O redirection

Input redirection

Instead of controlling a loop by testing the result of a command or by user input, you can specify a file from which to read input that controls the loop. In such cases, read is often the controlling command. As long as input lines are fed into the loop, execution of the loop commands continues. As soon as all the input lines are read the loop exits.

Since the loop construct is considered to be one command structure (such as while TEST-COMMAND; do CONSEQUENT-COMMANDS; done), the redirection should occur after the done statement, so that it complies with the form

```
command < file
```

This kind of redirection also works with other kinds of loops.

Output redirection

```
echo TEST > file.txt

echo TEST >> file.txt
```

5.5.Break and continue

The break built-in

The break statement is used to exit the current loop before its normal ending. This is done when you don't know in advance how many times the loop will have to execute, for instance because it is dependent on user input.

Activity:

- Simple example using break.

```
#!/bin/bash

# This script provides wisdom
# You can now exit in a decent way.

FORTUNE=/home/oracle/scripting/fortune

while true; do
echo
echo "On which topic do you want advice?"
echo "1.  politics"
echo "2.  startrek"
echo "3.  kernelnewbies"
echo "4.  sports"
echo "5.  bofh-excuses"
echo "6.  magic"
echo "7.  love"
echo "8.  literature"
echo "9.  drugs"
echo "10. education"
echo

echo -n "Enter your choice, or 0 for exit: "
read choice
echo

case $choice in
     1)
     $FORTUNE politics
     ;;
     2)
     $FORTUNE startrek
     ;;
     3)
     $FORTUNE kernelnewbies
     ;;
     4)
     $FORTUNE sports
     ;;
```

```
        5)
        $FORTUNE bofh-excuses
        ;;
        6)
        $FORTUNE magic
        ;;
        7)
        $FORTUNE love
        ;;
        8)
        $FORTUNE literature
        ;;
        9)
        $FORTUNE drugs
        ;;
        10)
        $FORTUNE education
        ;;
        0)
        echo "OK, see you!"
        break
        ;;
        *)
        echo "That is not a valid choice, try a number from
   to 10."
        ;;
   esac

   done
```

The continue built-in

The continue statement resumes iteration of an enclosing for, while, until or select loop.

When used in a for loop, the controlling variable takes on the value of the next element in the list. When used in a while or until construct, on the other hand, execution resumes with TEST-COMMAND at the top of the loop.

Activity:

- Example to show how continue works in a loop.

```
touch test1.log teST2.log teSt3.log
```

```
#!/bin/bash

# This script converts all file names containing upper
case characters into file# names containing only lower
cases.

for name in `ls te*.log`; do

if [[ "$name" != *[[:upper:]]* ]]; then
continue
fi

ORIG="$name"
NEW=`echo $name | tr 'A-Z' 'a-z'`

mv "$ORIG" "$NEW"
echo "new name for $ORIG is $NEW"
done
```

5.6.Menu making

General

Use of select

The select construct allows easy menu generation. The syntax is quite similar to that of the for loop:

```
select WORD [in LIST]; do RESPECTIVE-COMMANDS; done
```

LIST is expanded, generating a list of items. The expansion is printed to standard error; each item is preceded by a number. If in LIST is not present, the positional parameters are printed, as if in $@ would have been specified. LIST is only printed once.

Upon printing all the items, the PS3 prompt is printed and one line from standard input is read. If this line consists of a number corresponding to one of the items, the value of WORD is set to the name of that item. If the line is empty, the items and the PS3 prompt are displayed again. If an EOF (End Of File) character is read, the loop exits. Since most users don't have a clue which key combination is used for the EOF sequence, it is more user-friendly to have a break command as one of the items. Any other value of the read line will set WORD to be a null string.

The read line is saved in the REPLY variable.

The RESPECTIVE-COMMANDS are executed after each selection until the number representing the break is read. This exits the loop.

Activity:

- Example to create a simple menu.

```
#!/bin/bash

echo "This script can make any of the files in this
directory private."
echo "Enter the number of the file you want to protect:"

select FILENAME in *;
do
     echo "You picked $FILENAME ($REPLY), it is now only
accessible to you."
     chmod 700 "$FILENAME"
done
```

- Improved menu experience than above.

```
#!/bin/bash

echo "This script can make any of the files in this
directory private."
echo "Enter the number of the file you want to protect:"

PS3="Your choice in file number: "
QUIT="QUIT THIS PROGRAM - I feel safe now."
touch "$QUIT"

select FILENAME in *;
do
  case $FILENAME in
        "$QUIT")
          echo "Exiting."
          break
          ;;
        *)
          echo "You picked $FILENAME ($REPLY)"
          chmod 700 "$FILENAME"
          ;;
  esac
done
```

```
    rm "$QUIT"
```

5.7.Variables

General assignment of values

As we already saw, Bash understands many different kinds of variables or parameters. Thus far, we haven't bothered much with what kind of variables we assigned, so our variables could hold any value that we assigned to them.

Activity:

```
    VARIABLE=12
    echo $VARIABLE
    VARIABLE=string
    echo $VARIABLE
```

There are cases when you want to avoid this kind of behavior, for instance when handling telephone and other numbers. Apart from integers and variables, you may also want to specify a variable that is a constant. This is often done at the beginning of a script, when the value of the constant is declared. After that, there are only references to the constant variable name, so that when the constant needs to be changed, it only has to be done once. A variable may also be a series of variables of any type, a so-called array of variables (VAR0VAR1, VAR2, ... VARN).

Using the declare built-in

Using a declare statement, we can limit the value assignment to variables.

The syntax for declare is the following:

```
declare OPTION(s) VARIABLE=value
```

The following options are used to determine the type of data the variable can hold and to assign it attributes:

Options to the declare built-in

Option	Meaning
`-a`	Variable is an array.

Option	Meaning
`-f`	Use function names only.
`-i`	The variable is to be treated as an integer; arithmetic evaluation is performed when the variable is assigned a value.
`-p`	Display the attributes and values of each variable. When `-p` is used, additional options are ignored.
`-r`	Make variables read-only. These variables cannot then be assigned values by subsequent assignment statements, nor can they be unset.
`-t`	Give each variable the *trace* attribute.
`-x`	Mark each variable for export to subsequent commands via the environment.

Activity:

```
    declare -i VARIABLE=12
    VARIABLE=string
    echo $VARIABLE
    declare -p VARIABLE
```

Constants

In Bash, constants are created by making a variable read-only. The readonly built-in marks each specified variable as unchangeable. The syntax is:

```
readonly OPTION VARIABLE(s)
```

The values of these variables can then no longer be changed by subsequent assignment. If the -f option is given, each variable refers to a shell function; see Chapter 11, Functions. If -a is specified, each variable refers to an array of variables. If no arguments are given, or if -p is supplied, a list of all read-only variables is displayed. Using the -p option, the output can be reused as input.

The return status is zero, unless an invalid option was specified, one of the variables or functions does not exist, or -f was supplied for a variable name instead of for a function name.

```
readonly TUX=penguinpower
```

```
TUX=Mickeysoft
```

5.8.Operations on variables

Activity:

- Example to find Length of a variable.
```
echo $SHELL
echo ${#SHELL}
```
- Example to substitute.
```
${VAR:-WORD}
```
If VAR is not defined or null, the expansion of WORD is substituted; otherwise the value of VAR is substituted:
```
export TEST=" Its user defined variable"
echo ${TEST:-test}
echo ${TEST2:-test}
```

- Removing substrings

  To strip a number of characters, equal to OFFSET, from a variable, use this syntax:
```
${VAR:OFFSET:LENGTH}
```
The LENGTH parameter defines how many characters to keep, starting from the first character after the offset point. If LENGTH is omitted, the remainder of the variable content is taken.
```
export STRING="thisisaverylongname"
echo ${STRING:4}
echo ${STRING:6:5}
```

- Replacing parts of variable names.

  This is done using the
```
${VAR/PATTERN/STRING}
or
${VAR//PATTERN/STRING}
```
syntax. The first form replaces only the first match, the second replaces all matches of PATTERN with STRING
```
echo ${STRING/name/string}
```

6. FUNCTIONS & SIGNALS

6.1.Signals

Finding the signal man page

Your system contains a man page listing all the available signals, but depending on your operating system, it might be opened in a different way. On most Linux systems, this will be man 7 signal. When in doubt, locate the exact man page and section using commands like

```
man -k signal | grep list
```

or

```
apropos signal | grep list
```

Signal names can be found using `kill -l`.

Signals to your Bash shell

In the absence of any traps, an interactive Bash shell ignores SIGTERM and SIGQUIT. SIGINT is caught and handled, and if job control is active, SIGTTIN, SIGTTOU and SIGTSTP are also ignored. Commands that are run as the result of a command substitution also ignore these signals, when keyboard generated.

SIGHUP by default exits a shell. An interactive shell will send a SIGHUP to all jobs, running or stopped; see the documentation on the disown built-in if you want to disable this default behavior for a particular process. Use the huponexit option for killing all jobs upon receiving a SIGHUP signal, using the shopt built-in.

Sending signals using the shell

Control signals in Bash

Standard key combination	Meaning
Ctrl+C	The interrupt signal, sends SIGINT to the job running in the foreground.
Ctrl+Y	The *delayed suspend* character. Causes a running process to be stopped when it attempts to read input from the terminal. Control is returned to the shell, the user can foreground, background or kill the process. Delayed suspend is only available on operating systems supporting this feature.
Ctrl+Z	The *suspend* signal, sends a *SIGTSTP* to a running program, thus stopping it and returning control to the shell.

Usage of signals with kill

Most modern shells, Bash included, have a built-in kill function. In Bash, both signal names and numbers are accepted as options, and arguments may be job or process IDs. An exit status can be reported using the -l option: zero when at least one signal was successfully sent, non-zero if an error occurred.

Using the kill command from /usr/bin, your system might enable extra options, such as the ability to kill processes from other than your own user ID and specifying processes by name, like with pgrep and pkill.

Both kill commands send the TERM signal if none is given.

This is a list of the most common signals:

Common kill signals

Signal name	Signal value	Effect
SIGHUP	1	Hang-up
SIGINT	2	Interrupt from keyboard
SIGKILL	9	Kill signal
SIGTERM	15	Termination signal
SIGSTOP	17,19,23	Stop the process

SIGKILL and SIGSTOP cannot be caught, blocked or ignored.

When killing a process or series of processes, it is common sense to start trying with the least dangerous signal, SIGTERM. That way, programs that care about an orderly shutdown get the chance to follow the procedures that they have been designed to execute when getting the SIGTERM signal, such as cleaning up and closing open files. If you send a SIGKILL to a process, you remove any chance for the process to do a tidy cleanup and shutdown, which might have unfortunate consequences.

But if a clean termination does not work, the INT or KILL signals might be the only way. For instance, when a process does not die using Ctrl+C, it is best to use the kill -9 on that process ID.

6.2.Traps

General

There might be situations when you don't want users of your scripts to exit untimely using keyboard abort sequences, for example because input has to be provided or cleanup has to be done. The trap statement catches these sequences and can be programmed to execute a list of commands upon catching those signals.

The syntax for the trap statement is straightforward:

```
trap [COMMANDS] [SIGNALS]
```

This instructs the trap command to catch the listed SIGNALS, which may be signal names with or without the SIG prefix, or signal numbers. If a signal is 0 or EXIT, the COMMANDS are executed when the shell exits. If one of the signals is DEBUG, the list of COMMANDS is executed after every simple command. A signal may also be specified as ERR; in that case COMMANDS are executed each time a simple command exits with a non-zero status. Note that these commands will not be executed when the non-zero exit status comes from part of an if statement, or from a while or until loop. Neither will they be executed if a logical AND (&&) or OR (||) result in a non-zero exit code, or when a command's return status is inverted using the ! operator.

The return status of the trap command itself is zero unless an invalid signal specification is encountered. The trap command takes a couple of options, which are documented in the Bash info pages.

How Bash interprets traps

When Bash receives a signal for which a trap has been set while waiting for a command to complete, the trap will not be executed until the command completes. When Bash is waiting for an asynchronous command via the wait built-in, the reception of a signal for which a trap has been set will cause the wait built-in to return immediately with an exit status greater than 128, immediately after which the trap is executed.

Activity:

- Detecting when a variable is used.
  When debugging longer scripts, you might want to give a variable the trace attribute and trap DEBUG messages for that variable. Normally you would just declare a variable using an assignment like VARIABLE=value. Replacing the declaration of the variable with the following lines might provide valuable information about what your script is doing:

```
declare -t VARIABLE=value
trap "echo VARIABLE is being used here." DEBUG
# rest of the script
```

- Removing rubbish upon exit.
  The whatis command relies on a database which is regularly built using the makewhatis.cron script with cron:

```
#!/bin/bash
LOCKFILE=/var/lock/makewhatis.lock
# Previous makewhatis should execute successfully:
[ -f $LOCKFILE ] && exit 0
# Upon exit, remove lockfile.
```

```
trap "{ rm -f $LOCKFILE ; exit 255; }" EXIT
touch $LOCKFILE
makewhatis -u -w
exit 0
```

- Program not to allow user to user Ctrl+C in the shell execution.

```
#!/bin/bash
trap "echo You are not allowed to use Ctrl+C" SIGINT
echo -n "Please give your name: "
read name
echo "Thank you - You are $name"
```

6.3.Functions

What are functions?

Shell functions are a way to group commands for later execution, using a single name for this group, or routine. The name of the routine must be unique within the shell or script. All the commands that make up a function are executed like regular commands. When calling on a function as a simple command name, the list of commands associated with that function name is executed. A function is executed within the shell in which it has been declared: no new process is created to interpret the commands.

Special built-in commands are found before shell functions during command lookup. The special built-ins are: break, :, ., continue, eval, exec, exit, export, readonly, return, set, shift, trap and unset.

Function syntax

Functions either use the syntax

```
function FUNCTION { COMMANDS; }
```

or

```
FUNCTION () { COMMANDS; }
```

Both define a shell function FUNCTION. The use of the built-in command function is optional; however, if it is not used, parentheses are needed.

The commands listed between curly braces make up the body of the function. These commands are executed whenever FUNCTION is specified as the name of a command. The exit status is the exit status of the last command executed in the body.

Positional parameters in functions

Functions are like mini-scripts: they can accept parameters, they can use variables only known within the function (using the local shell built-in) and they can return values to the calling shell.

A function also has a system for interpreting positional parameters. However, the positional parameters passed to a function are not the same as the ones passed to a command or script.

When a function is executed, the arguments to the function become the positional parameters during its execution. The special parameter # that expands to the number of positional parameters is updated to reflect the change. Positional parameter 0 is unchanged. The Bash variable FUNCNAME is set to the name of the function, while it is executing.

If the return built-in is executed in a function, the function completes and execution resumes with the next command after the function call. When a function completes, the values of the positional parameters and the special parameter # are restored to the values they had prior to the function's execution. If a numeric argument is given to return, that status is returned.

Displaying functions

All functions known by the current shell can be displayed using the set built-in without options. Functions are retained after they are used, unless they are unset after use. The which command also displays functions.

Recycling

There are plenty of scripts on your system that use functions as a structured way of handling series of commands. On some Linux systems, for instance, you will find the /etc/rc.d/init.d/functions definition file, which is sourced in all init scripts. Using this method, common tasks such as checking if a process runs, starting or stopping a daemon and so on, only have to be written once, in a general way. If the same task is needed again, the code is recycled.

You could make your own /etc/functions file that contains all functions that you use regularly on your system, in different scripts. Just put the line

. /etc/functions

somewhere at the start of the script and you can recycle functions.

Activity:

- Create a normal function in a script to execute set of commands.

```
#!/bin/bash
function listdb
{
if [ -f /etc/oratab ]
then
cat /etc/oratab | grep -v ^# | awk 'FS=":" {print $1}'|
grep -v ^$
else
echo "Oracle database not installed on this server"
fi
}
listdb
```

- Create a function on command line.

```
function listdb
{
if [ -f /etc/oratab ]
then
cat /etc/oratab | grep -v ^# | awk 'FS=":" {print $1}'|
grep -v ^$
else
echo "Oracle database not installed on this server"
fi
}
listdb
```

6.4.Common shell features

The following features are standard in every shell. Note that the stop, suspend, jobs, bg and fg commands are only available on systems that support job control.

Common shell features list.

Command	Meaning
>	Redirect output
>>	Append to file
<	Redirect input
<<	"Here" document (redirect input)
|	Pipe output
&	Run process in background.
;	Separate commands on same line
*	Match any character(s) in filename
?	Match single character in filename
[ ]	Match any characters enclosed
( )	Execute in subshell
` `	Substitute output of enclosed command

Command	Meaning
" "	Partial quote (allows variable and command expansion)
' '	Full quote (no expansion)
\	Quote following character
$var	Use value for variable
$$	Process id
$0	Command name
$n	nth argument (n from 0 to 9)
#	Begin comment
bg	Background execution
break	Break from loop statements
cd	Change directories
continue	Resume a program loop
echo	Display output
eval	Evaluate arguments
exec	Execute a new shell
fg	Foreground execution
jobs	Show active jobs
kill	Terminate running jobs
newgrp	Change to a new group
shift	Shift positional parameters
stop	Suspend a background job
suspend	Suspend a foreground job
time	Time a command
umask	Set or list file permissions
unset	Erase variable or function definitions
wait	Wait for a background job to finish

6.5.Differing shell features

The table below shows major differences between the standard shell (sh), Bourne Again SHell (bash), Korn shell (ksh) and the C shell (csh).

sh	bash	ksh	csh	Meaning/Action
$	$	$	%	Default user prompt
	>|	>|	>!	Force redirection

sh	bash	ksh	csh	Meaning/Action
> file 2>&1	&> file or > file 2>&1	> file 2>&1	>& file	Redirect stdout and stderr to file
	{ }		{ }	Expand elements in list
\`command\`	\`command\` or $(command)	$(command)	\`command\`	Substitute output of enclosed command
$HOME	$HOME	$HOME	$home	Home directory
	~	~	~	Home directory symbol
	~+, ~-, dirs	~+, ~-	=-, =N	Access directory stack
var=value	VAR=value	var=value	set var=value	Variable assignment
export var	export VAR=value	export var=val	setenv var val	Set environment variable
	${nnnn}	${nn}		More than 9 arguments can be referenced
"$@"	"$@"	"$@"		All arguments as separate words
$#	$#	$#	$#argv	Number of arguments
$?	$?	$?	$status	Exit status of the most recently executed command
$!	$!	$!		PID of most recently backgrounded process
$-	$-	$-		Current options
. file	source file or . file	. file	source file	Read commands in file
	alias x='y'	alias x=y	alias x y	Name x stands for command y
case	case	case	switch or case	Choose alternatives
done	done	done	end	End a loop statement
esac	esac	esac	endsw	End case or switch
exit n	exit n	exit n	exit (expr)	Exit with a status
for/do	for/do	for/do	foreach	Loop through variables
	set -f, set -o nullglob|dotglob|nocaseglob|noglob		noglob	Ignore substitution characters for filename generation

sh	bash	ksh	csh	Meaning/Action
hash	hash	alias -t	hashstat	Display hashed commands (tracked aliases)
hash cmds	hash cmds	alias -t cmds	rehash	Remember command locations
hash -r	hash -r		unhash	Forget command locations
	history	history	history	List previous commands
	ArrowUp+Enter or !!	r	!!	Redo previous command
	!str	r str	!str	Redo last command that starts with “str”
	!cmd:s/x/y/	r x=y cmd	!cmd:s/x/y/	Replace “x” with “y” i n most recent command starting with “cmd”, then execute.
if [$i -eq 5]	if [$i -eq 5]	if ((i==5))	if ($i==5)	Sample condition test
fi	fi	fi	endif	End if statement
ulimit	ulimit	ulimit	limit	Set resource limits
pwd	pwd	pwd	dirs	Print working directory
read	read	read	$<	Read from terminal
trap 2	trap 2	trap 2	onintr	Ignore interrupts
	unalias	unalias	unalias	Remove aliases
until	until	until		Begin until loop
while/do	while/do	while/do	while	Begin while loop

7. TOOLS DEVELOPMENT for DBA's

7.1.Storage check tool

Let us create a tool that can perform the following.

- Check tablespace utilizations on the databases running on the server.
- Resize a datafile.

Logical flow:

1. Check tablespace utilizations on the databases running on the server.
    a. Ask for which database it has to check.
    b. Set environment variables of that database.
    c. Connect to the database and list tablespaces existing.
    d. Ask user to pick the tablespace he wants to check.
    e. Give the complete analysis of that tablespace – used size, free size, extensible size, Datafiles, mount point usages where datafiles reside.
    f. Quit from the tool.
2. Resize a datafile.
    a. Ask for which database it has to connect.
    b. Set environment variables of that database.
    c. Connect to the database and list tablespaces and datafiles.
    d. Ask which datafile to resize and receive the next size to set.
    e. Resize the datafile and quit the tool.

7.2.Backups tool

Let us create a backup tool that can perform the following.

- List out the summary of backups taken.
- Take full backup of database.
- Take incremental backup.
- Take archivelog backup.
- Clean obsolete backups.

Logical flow:

1. List out the summary of backups taken.
    a. Ask for which database it has to check.
    b. Set environment variables of that database.
    c. Connect to the RMAN of that database.
    d. Show the summary of all the backups taken by RMAN.

   e. Quit.
2. Take full backup of database.
   a. Ask for which database it has to check.
   b. Set environment variables of that database.
   c. Connect to the RMAN of that database.
   d. Take full backup of database and Quit.
3. Take incremental backup.
   a. Ask for which database it has to check.
   b. Set environment variables of that database.
   c. Connect to the RMAN of that database.
   d. Ask which incremental backup of database to take and Quit.
4. Take archivelog backup.
   a. Ask for which database it has to check.
   b. Set environment variables of that database.
   c. Connect to the RMAN of that database.
   d. Show the available archivelogs on the disk.
   e. Confirm from the user if all the archivelogs has to be backedup.
   f. Quit.
5. Clean obsolete backups.
   a. Ask for which database it has to check.
   b. Set environment variables of that database.
   c. Connect to the RMAN of that database.
   d. List obsolete backups and ask user to confirm the deletion of the list of backups.

7.3.Refresh tool

Let us develop a tool to perform:

- Table refresh
- Schema refresh

Logical flow:

1. Table refresh
   a. Ask source database name, schema name and table name.
   b. Verify if details are correct.
   c. Ask destination database name, schema name and table name.
   d. Verify if details are correct.
   e. Ask to which directory backup should be taken.
   f. Verify if location exists.
   g. Take the backup of that table and refresh.

   h. Quit.

2. Schema refresh.
   a. Ask source database name, schema name.
   b. Verify if details are correct.
   c. Ask destination database name, schema name.
   d. Verify if details are correct.
   e. Ask to which directory backup should be taken.
   f. Verify if location exists.
   g. Take the backup of that schema and refresh.
   h. Quit.

7.4.User creation tool

User creation tool would be useful if you regularly get requests to create user with set of roles to assign to the user all time based on the application.

Logical flow:

- List the databases, Connect to it.
- Ask for user to create.
- Verify if he is not existing.
- Create user and assign the roles.
- Quit

Thank you!!! It is our mission to mould you as a successful professional Oracle DBA.

I wish you refer us to your friends/colleagues and like us in social media.

For any queries, please email to kumar@orskl.com or WhatsApp to +919951696808.

We appreciate your feedback – Please send it to admin@orskl.com

Subscribe in our website and let us help you with latest updates on interesting articles on Oracle DBA – www.orskl.com

Like and Subscribe to our YouTube channel for following our videos – www.youtube.com/c/OrSklAcademyteam

Follow us on Linkedin - https://www.linkedin.com/company/orskl-corp?trk=biz-companies-cym

Follow us on twitter – www.twitter.com/orskl

Like us on facebook – www.facebook.com/orsklacademy