Chapter 3. The Bash environment:

Global user configuration files

/etc/profile: shell variables PATH, USER, MAIL, HOSTNAME and HISTSIZE *($ set , $ printenv)* 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

Individual user configuration files

~/.bash_profile or ~/.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:

~/.bash_login

This file contains specific settings that are normally only executed when you log in to the system. In the example, we use it to configure the umask value and to show a list of connected users upon login. This user also gets the calendar for the current month.

~/.profile

In the absence of ~/.bash_profile and ~/.bash_login, ~/.profile is read. It can hold the same configurations, which are then also accessible by other shells. Mind that other shells might not understand the Bash syntax.

~/.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

```
$ cat .bashrc
# /home/sandip/.bashrc
# Source global definitions
if [ -f /etc/bashrc ]; then
. /etc/bashrc fi
# shell options set -o noclobber
# my shell variables
export PS1="\[\033[1;44m\]\u \w\[\033[0m\] " export
PATH="$PATH:~/bin:~/scripts"
```

```
# my aliases
alias cdrecord='cdrecord -dev 0,0,0 -speed=8'
alias ss='ssh octarine'
alias ll='ls -la'
```

~/.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.

Changing shell configuration files

Prompting Variables : Different prompts for different users

```
export PS1="\[\033[1;44m\]\u@\h \w\[\033[0m\] # "
PS1="\[\033[1;44m\]sandip is in \w\[\033[0m\]"
PS2=""> "
PS4="+ "
```

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

Shell Variables

VARNAME="value"

```
# export sandip=1
# echo $sandip
1
```

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.

```
# export 1sandip=1
-bash: export: `1sandip=1': not a valid identifier
```

A subshell can change variables it inherited from the parent, but the changes made by the child don't affect the parent. This is demonstrated in the example:

```
stud ~> full_name="Sandip Raaje"
stud ~> bash
stud ~> echo $full_name
stud ~> exit
stud ~> export full_name
stud ~> bash
stud ~> echo $full_name
Sandip Raaje
stud ~> export full_name="Wikram the Great"
stud ~> echo $full_name
Wikram the Great
stud ~> exit
stud ~> echo $full_name
Sandip Raaje
stud ~>
```

Reserved variables

Bourne shell reserved variables (sh)

Variable name	Definition
CDPATH	A colon-separated list of directories used as a search path for the **cd** built-incommand.
HOME	The current user's home directory; the default for the **cd** built-in. The value ofthis 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 fornew 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 (bash)

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 completionfacility.
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 -eoption 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, quicksubstitution, 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.

Variable name	Definition
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/hoststhat 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 thesole 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 specificallyselected with a variable starting with LC_.
LC_ALL	This variable overrides the value of LANGand 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-quotedstrings preceded by a "$" sign.
LC_NUMERIC	This variable determines the locale category used for numberformatting.
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 MAILPATHor MAILvariables.

Variable name	Definition
OLDPWD	The previous working directory as set by the **cd** built-in.
OPTERR	If 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 entersPOSIX 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 theprinting 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 echoedwhen the -xoption 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 and32767 is generated. Assigning a value to this variable seeds the randomnumber generator.
REPLY	The default variable for the **read** built-in.
SECONDS	This variable expands to the number of seconds since the shell wasstarted.
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 thatnumber of seconds if input does not arrive.
UID	The numeric, real user ID of the current user.

Special parameters

$*	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 IFSspecial 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.

$* 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. Consider the following example, a very simple script that makes a backup of sandip's home directory to a remote server:

```
#!/bin/bash
# This script makes a backup of my home directory.
cd /home
# This creates the archive
tar cf /var/tmp/home_sandip.tar sandip > /dev/null 2>&1
# First remove the old bzip2 file. Redirect errors because this generates
some if the archive
# does not exist. Then create a new compressed file.
rm /var/tmp/home_sandip.tar.bz2 2> /dev/null
bzip2 /var/tmp/home_sandip.tar
# Copy the file to another host - we have ssh keys for making this work
without intervention.
scp /var/tmp/home_sandip.tar.bz2 bordeaux:/opt/backup/sandip > /dev/null
2>&1
```

```
# Create a timestamp in a logfile.
date >> /home/sandip/log/home_backup.log
echo backup succeeded >> /home/sandip/log/home_backup.log
```

First of all, you are more likely to make errors if you name files and directories manually each time you need them. Secondly, suppose sandip wants to give this script to Wikram.

```
#!/bin/bash
# This script makes a backup of my home directory.
# Change the values of the variables to make the script work for you:
BACKUPDIR=/home
BACKUPFILES=sandip
TARFILE=/var/tmp/home_sandip.tar
BZIPFILE=/var/tmp/home_sandip.tar.bz2
SERVER=bordeaux
REMOTEDIR=/opt/backup/sandip
LOGFILE=/home/sandip/log/home_backup.log
cd $BACKUPDIR
# This creates the archive
tar cf $TARFILE $BACKUPFILES > /dev/null 2>&1
# First remove the old bzip2 file. Redirect errors because this generates
some if the archive
# does not exist. Then create a new compressed file. rm $BZIPFILE 2>
/dev/null
bzip2 $TARFILE
# Copy the file to another host - we have ssh keys for making this work
without intervention.
scp $BZIPFILE $SERVER:$REMOTEDIR > /dev/null 2>&1 # Create a timestamp in
a logfile.
date >> $LOGFILE
echo backup succeeded >> $LOGFILE
```

Aliases

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.

```
# alias
alias cp='cp -i'
alias l.='ls -d .* --color=tty'
alias ll='ls -l --color=tty'
alias ls='ls --color=tty'
alias mc='. /usr/share/mc/bin/mc-wrapper.sh'
alias mv='mv -i'
alias rm='rm -i'
alias which='alias | /usr/bin/which --tty-only --read-alias --show-dot --
show-tilde'
# alias dh='df -Ph'
```

```
# dh
Filesystem Size Used Avail Use% Mounted on
/dev/mapper/rootvg-rootVol00     25G   20G 3.3G 87% /
/dev/mapper/rootvg-appsVol00     25G 173M   23G   1% /apps
/dev/mapper/rootvg-varVol00 30G 2.6G   25G 10% /var
/dev/mapper/rootvg-tmpVol00 9.7G 163M 9.1G  2% /tmp
/dev/sda1  494M  32M 438M   7% /boot
tmpfs 12G   16K   12G   1% /dev/shm
tmpfs 4.0K  0 4.0K     0% /dev/vx 169.254.10.47:/vol/opsvol 1.1T 754G 271G
74% /data 169.254.10.46:/vol/Hpthbatch 20G   56M        20G   1% /Hpthbatch
#
# unalias dh # dh
-bash: dh: command not found
```

Arithmetic expansion

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

$((EXPRESSION))

```
# echo $((3*5)) 15
# num1=10
# num2=5
# echo $(($num1+$num2)) 15
# echo $((num1++)) 10
# echo $((num1++)) 11
# echo $((num1++)) 12
# echo $num1 13
```

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

Bash options

We already discussed a couple of Bash options that are useful for debugging your scripts. In this section,we will take a more in-depth view of the Bash options.

Displaying options

```
# set -o
allexport  off
braceexpand     on
emacs      on
errexit    off
errtraceoff
functrace  off
hashall    on
histexpand on
history    on
ignoreeof  off
interactive-comments  on
keyword    off
monitor    on
noclobber  off
noexec     off
noglob     off
nolog off
notify     off
nounset    off
onecmd     off
physical   off
pipefail   off
posix      off
privileged off
verbose    off
vi         off
xtrace     off
```

Changing options

`noclobber` option, which prevents existing files from being overwritten by redirection operations.

```
# set -o noclobber
# touch test1 # date > test1
-bash: test1: cannot overwrite existing file # set +o noclobber
# date > test1 # cat test1
Sat May 12 04:05:14 PDT 2012
```

noglob option, tells the shell not to expand wildcard characters in file names. This command is occasionally useful if you are entering command lines that contain several characters that would normally be expanded.

```
# set -o noglob # touch *
# ls -l *
```

```
-rw-r--r-- 1 root root 0 May 12 04:14 *

rm \*
rm: remove regular empty file `*'? y # rm *
rm: cannot lstat `*': No such file or directory
```

shopt

bash 2.0 introduced a new built-in for configuring shell behaviour, shopt. This built-in is meant as a replacement for option configuration originally done through environment variables and the set command.

```
shopt [-pqsu] [-o] [optname …]
```

-s Enable (set) each *optname*.
-u Disable (unset) each *optname*.
-q Suppresses normal output; the return status indicates whether the *optname* is set or unset. If multiple *optname* arguments are given with -q, the return status is zero if all *optname*s are enabled; non-zero otherwise.
-o Restricts the values of *optname* to be those defined for the -o option to the set builtin.

```
# shopt -o
allexport       off
braceexpand     on
emacs           on
errexit         off
errtrace        off
functrace       off
hashall         on
histexpand      on
history         on
ignoreeof       off
interactive-comments    on
keyword         off
monitor         on
noclobber       off
noexec          off
noglob          off
nolog           off
notify          off
nounset         off
onecmd          off
physical        off
pipefail        off
posix           off
privileged      off
verbose         off
vi              off
xtrace          off
```

Command search path

```
echo $PATH
```

```
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/root/bin
```

Command hashing

You may be thinking that having to go and find a command in a large list of possible places would take a long time, and you'd be right. To speed things up, *bash* uses what is known as a *hash* table.

You can see what is currently in the hash table with the command **hash**:

```
$ hash
hits command
2 /bin/cat
1 /usr/bin/stat
2 /usr/bin/less
1 /usr/bin/man
2 /usr/bin/apropos
2 /bin/more
1 /bin/ln
3 /bin/ls
1 /bin/ps
2 /bin/vi
```

Directory search path and variables

CDPATH is a variable whose value, like that of **PATH**, is a list of directories separated by colons. Its purpose is to augment the functionality of the **cd** built-in command.

```
# pwd
/root
# echo $CDPATH

# CDPATH=/tmp
# echo $CDPATH
/tmp
# cd my
/tmp/my
#
```

bash provides another shorthand mechanism for referring to directories; if you set the shell option **cdable_vars** using **shopt**,

Exercises

1. Create 3 variables, VAR1, VAR2 and VAR3; initialize them to hold the values "thirteen", "13"and "Happy Birthday" respectively.

2. Display the values of all three variables.

3. Are these local or global variables?

4. Remove VAR3.

5. Can you see the two remaining variables in a new terminal window?

7. Make sure that newly created users also get a nice, personalized prompt which informs them on which system in which directory they are working. Test your changes by adding a new user and loggingin as that user.

8. Write a script in which you assign two integer values to two variables. The script should calculatethe surface of a rectangle which has these proportions. It should be aired with comments and generate elegant output.

Chapter 4. Basic Shell Programming

Shell Scripts and Functions

A script (a file that contains shell commands) is a shell program. Your .bash_profile and environment files, discussed in the previous chapter, are shell scripts.
Loading variables & running script.
source ~/.bashrc

Sha-bang!!: Default interpreter, first line in your script.
#!

Hello World
As usual Your first bash script!

```
#!/bin/bash
echo "Hello World"
```

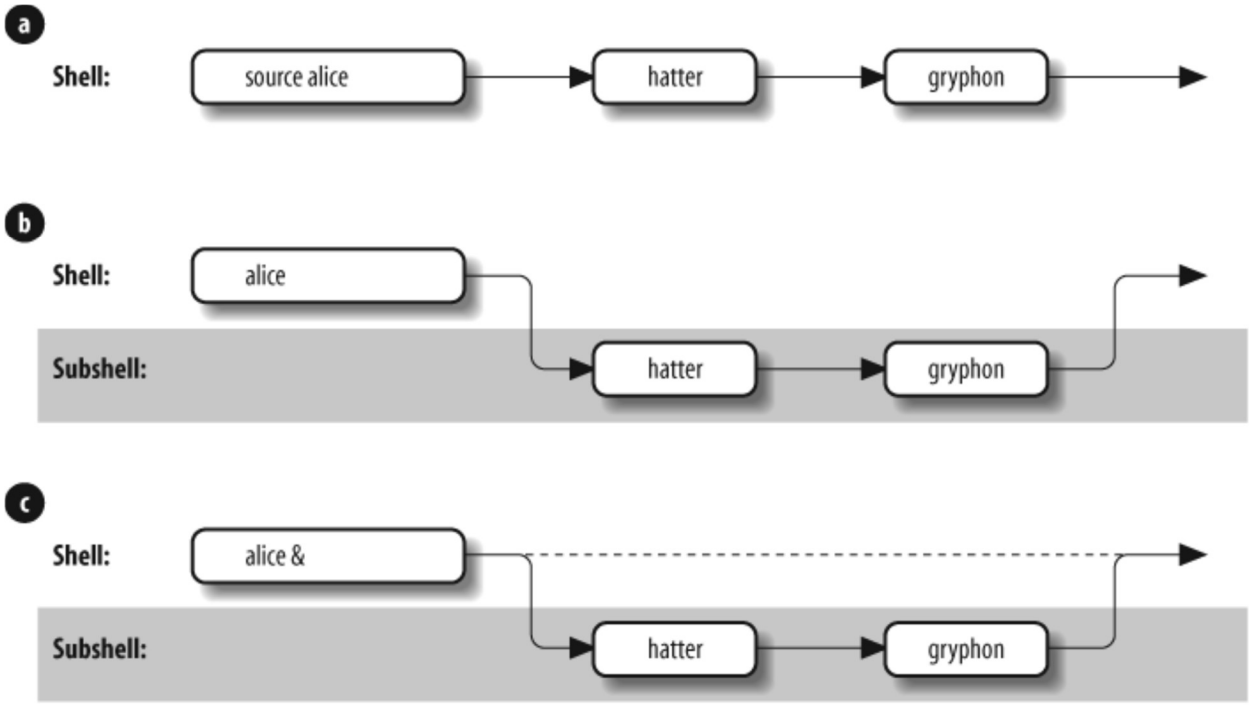
How to run ?
```
$ source hello_world.sh
Hello World
$ sh hello_world.sh
Hello World
$ . hello_world.sh
Hello World
$ ./hello_world.sh
-bash: ./hello_world.sh: Permission denied
```

WHY?
File permissions to run script: “x”
```
# chmod +x
# chmod +ux
# chmod +gx
# chmod +ox
```

Octal Notations for permission
```
1 = executable : --x
2 = write      : -w-
4 = read       : r--
7 ->           : rwx
```

HOW TO CHECK PERMISSIONS?

Shell & Subshells



type

```
$ type -t bash
file
$ type -t if
keyword
$ type -all ll
ll is aliased to `ls -l --color=auto'


$ alias auther="Sandip"
$ type -all auther
auther is aliased to `Sandip'

$ auther ()
> {
> echo "The Auther name is Sandip"
> }
$ auther is ./auther
The Auther name is Sandip
$ auther
The Auther name is Sandip
$
```

Positional Parameters

The most important special, built-in variables are called positional parameters. These hold the “command-line” arguments to scripts when they are invoked. Positional parameters have the names 1, 2, 3, etc., meaning that their values are denoted by $1, $2, $3, etc. There is also a positional parameter 0, whose value is the name of the script.

EG:

```
# print script name
echo $0
echo $_

# print all parameters with *, not preferred
echo $*

# print all parameters with @
echo $@

# print number of parameters
echo $#
```

Functions

bash's function feature is an expanded version of a similar facility in the System V Bourne shell and a few other shells. A function is sort of a script-within-a-script; you use it to define some shell code by name and store it in the shell's memory, to be invoked and run later.

EG:

```
function functname
{
shell commands
}
```

or:

```
functname ( )
{
shell commands
}
```

IMP: Functions have their own positional parameters, So what are local variables to the functions?

EG:

```
#!/bin/bash

function countargs
{
echo "Function has $# parameters/arguments."
echo "$@"
}

echo "Script has $# parameters/arguments"
echo "$@"
countargs Sandip Wikram Pratik
```

There is no functional difference, You can also delete a function definition with the command **$ unset -f *functname***.

You can find out what functions are defined in your login session by typing "**declare -f "** .
If you just want to see the names of the functions, you can use "**declare -F"**.

TRY IT YOURSELF

```
#!/bin/bash

# print script name
echo $0
echo $_

# print all parameters with @
echo $@

# print number of parameters
echo $#

testfunction()
{
# print script name
echo $0
echo $_

# print all parameters with *, not preffered
echo "Parameters passed to function : " $*

# print all parameters with @
echo "Parameters passed to function : " $@

# print number of parameters
echo $#
}
```