Bash Shell Scripting Syntax

Basic Syntax and Special Characters:

1. Scripts require you to follow a standard language syntax. Rules delineate how to define variables and how to construct and format allowed statements, etc.
2. The below table lists some special character usages within bash scripts
3. There are other special characters and character combinations and constructs that scripts understand, such as (..), {..}, [..], &&, ||, ', ", $((...))

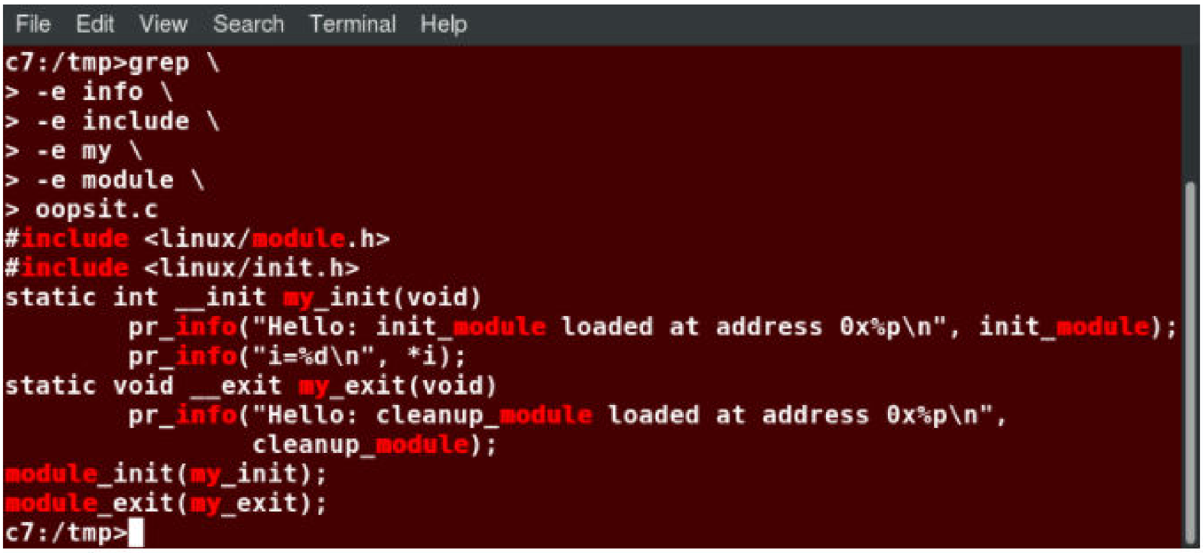
Character	Description
#	Used to add a comment, except when used as `\#`, or as `#!` when starting a script
\	Used at the end of a line to indicate continuation on to the next line
;	Used to interpret what follows as a new command to be executed next
$	Indicates what follows is an environment variable
>	Redirect output
>>	Append output
<	Redirect input
|	Used to pipe the result into the next command

Splitting Long Commands Over Multiple Lines:

1. Sometimes, commands are too long to either easily type on one line, or to grasp and understand (even though there is no real practical limit to the length of a command line).
2. In this case, the concatenation operator (\), the backslash character, is used to continue long commands over several lines.
3. Here is an example of a command installing a long list of packages on a system using Debian package management:

```
$~/> cd $HOME
$~/> sudo apt-get install autoconf automake bison build-essential \
    chrpath curl diffstat emacs flex gcc-multilib g++-multilib \
    libsdl1.2-dev libtool lzop make mc patch \
    screen socat sudo tar texinfo tofrodos u-boot-tools unzip \
    vim wget xterm zip
```

4. The above command is divided into multiple lines to make it look readable and easier to understand.
5. The \ operator at the end of each line causes the shell to combine (concatenate) multiple lines and executes them as one single command.



```
File   Edit   View   Search   Terminal   Help
c7:/tmp>grep \
> -e info \
> -e include \
> -e my \
> -e module \
> oopsit.c
#include <linux/module.h>
#include <linux/init.h>
static int __init my_init(void)
        pr_info("Hello: init_module loaded at address 0x%p\n", init_module);
        pr_info("i=%d\n", *i);
static void __exit my_exit(void)
        pr_info("Hello: cleanup_module loaded at address 0x%p\n",
                cleanup_module);
module_init(my_init);
module_exit(my_exit);
c7:/tmp>
```

Splitting Long Commands Over Multiple Lines

Putting Multiple Commands on a Single Line:

1. Users sometimes need to combine several commands and statements and even conditionally execute them based on the behavior of operators used in between them.
2. This method is called chaining of commands.
3. There are several different ways to do this, depending on what you want to do.
4. The **;** (semicolon) character is used to separate these commands and execute them sequentially, as if they had been typed on separate lines.
5. Each ensuing command is executed whether or not the preceding one succeeded.
6. Thus, the three commands in the following example will all execute, even if the ones preceding them fail:

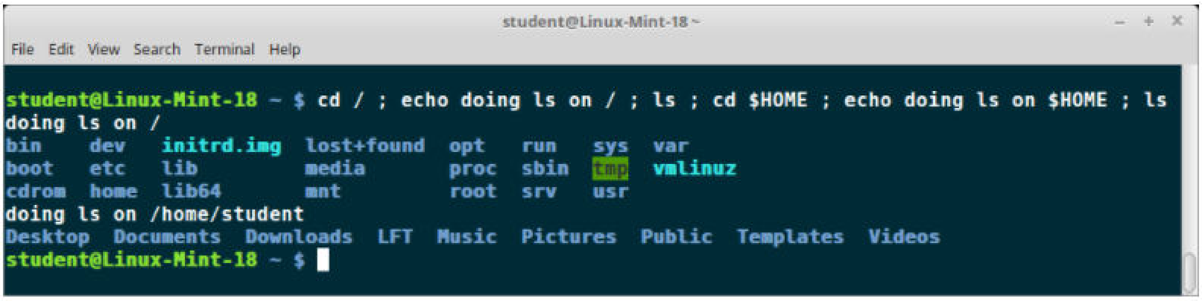
```
$ make ; make install ; make clean
```

7. However, you may want to abort subsequent commands when an earlier one fails. You can do this using the **&&** (and) operator as in:

```
$ make && make install && make clean
```

8. Chaining commands is not the same as piping them**;** in the later case succeeding commands begin operating on data streams produced by earlier ones before they complete, while in chaining each step exits before the next one starts.

```
$ cat file1 || cat file2 || cat file3
```



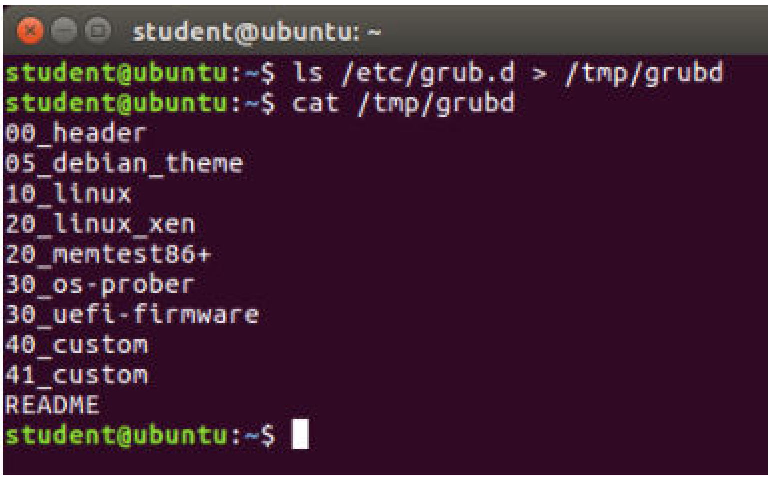
Putting Multiple Commands on a Single Line

Output Redirection:

1. Most operating systems accept input from the keyboard and display the output on the terminal.
2. However, in shell scripting you can send the output to a file.
3. The process of diverting the output to a file is called output redirection.
4. The **>** character is used to write output to a file. For example, the following command sends the output of free to /tmp/free.out:

```
$ free > /tmp/free.out
```

5. To check the contents of /tmp/free.out, at the command prompt type **cat /tmp/free.out.**
6. Two **>** characters **(>>)** will append output to a file if it exists, and act just like **>** if the file does not already exist.



Input Redirection:

1. Just as the output can be redirected to a file, the input of a command can be read from a file.
2. The process of reading input from a file is called input redirection and uses the **<** character.
3. The following three commands (using **wc** to count the number of lines, words and characters in a file) are entirely equivalent and involve input redirection, and a command operating on the contents of a file:

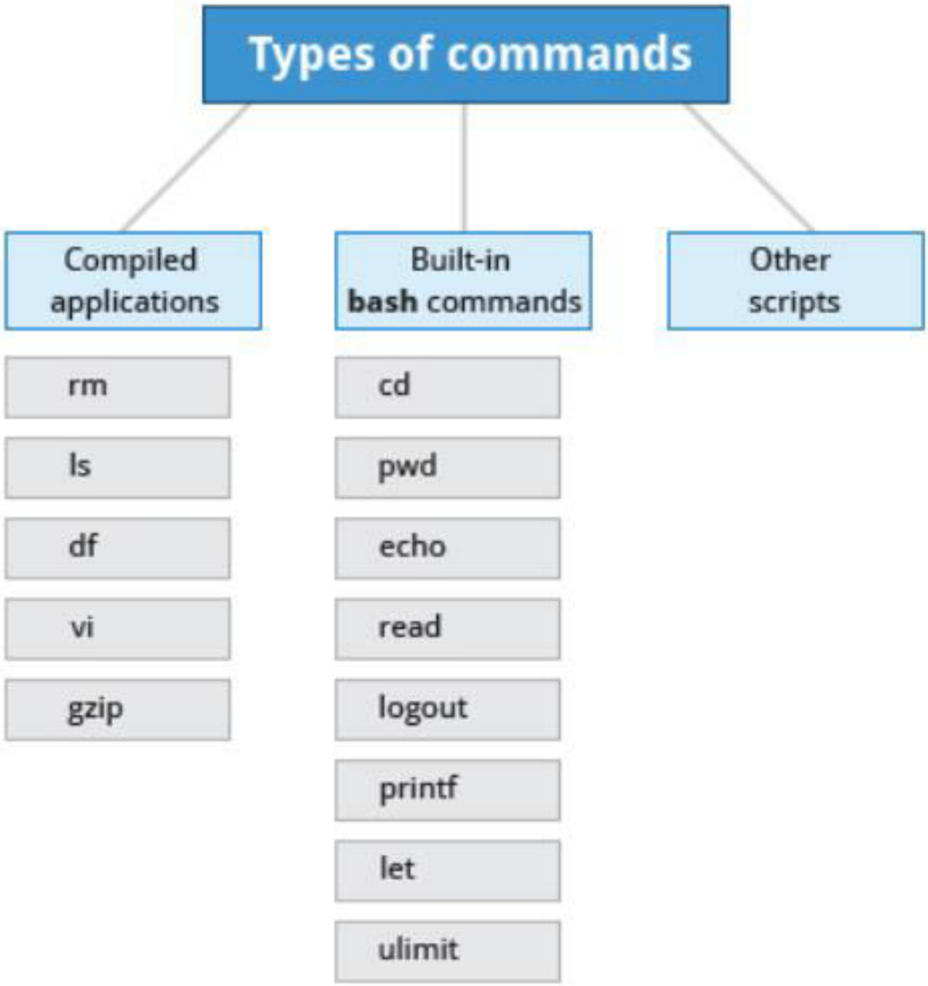
```
$ wc < /etc/passwd
49  105 2678 /etc/passwd

$ wc /etc/passwd
49  105 2678 /etcpasswd

$ cat /etc/passwd | wc
49  105 2678
```

Built-In Shell Commands:

1. Shell scripts execute sequences of commands and other types of statements. These commands can be:
   - Compiled applications
   - Built-in bash commands
   - Shell scripts or scripts from other interpreted languages, such as perl and Python.
2. Compiled applications are binary executable files, generally residing on the filesystem in well-known directories such as /usr/bin.
3. Shell scripts always have access to applications such as **rm, ls, df, vi, and gzip**, which are programs compiled from lower level programming languages such as C.
4. In addition, bash has many built-in commands, which can only be used to display the output within a terminal shell or shell script.
5. Sometimes, these commands have the same name as executable programs on the system, such as **echo**, which can lead to subtle problems.
6. bash built-in commands include **cd, pwd, echo, read, logout, printf, let, and ulimit.**
7. Thus, slightly different behavior can be expected from the built-in version of a command such as **echo** as compared to /bin/echo.
8. A complete list of bash built-in commands can be found in the bash man page, or by simply typing help, as we review on the next page.



Built-In Shell Commands

Commands Built in to bash:

1. We already enumerated which commands have versions built in to bash, in our earlier discussion of how to get help on Linux systems.
2. Once again, here is a screenshot listing exactly which commands are available.

```
File  Edit  View  Search  Terminal  Help
c7:/home/coop>help
GNU bash, version 4.2.46(1)-release (x86_64-redhat-linux-gnu)
These shell commands are defined internally.  Type `help' to see this list.
Type `help name' to find out more about the function `name'.
Use `info bash' to find out more about the shell in general.
Use `man -k' or `info' to find out more about commands not in this list.

A star (*) next to a name means that the command is disabled.

 job_spec [&]                                                   history [-c] [-d offset] [n] or history -anrw [filen>
 (( expression ))                                               if COMMANDS; then COMMANDS; [ elif COMMANDS; then CO>
 . filename [arguments]                                         jobs [-lnprs] [jobspec ...] or jobs -x command [args>
 :                                                              kill [-s sigspec | -n signum | -sigspec] pid | jobsp>
 [ arg... ]                                                     let arg [arg ...]
 [[ expression ]]                                               local [option] name[=value] ...
 alias [-p] [name[=value] ... ]                                 logout [n]
 bg [job_spec ...]                                              mapfile [-n count] [-O origin] [-s count] [-t] [-u f>
 bind [-lpvsPVS] [-m keymap] [-f filename] [-q name] [>         popd [-n] [+N | -N]
 break [n]                                                      printf [-v var] format [arguments]
 builtin [shell-builtin [arg ...]]                              pushd [-n] [+N | -N | dir]
 caller [expr]                                                  pwd [-LP]
 case WORD in [PATTERN [| PATTERN]...) COMMANDS ;;]...>         read [-ers] [-a array] [-d delim] [-i text] [-n ncha>
 cd [-L|[-P [-e]]] [dir]                                        readarray [-n count] [-O origin] [-s count] [-t] [-u>
 command [-pVv] command [arg ...]                               readonly [-aAf] [name[=value] ...] or readonly -p
 compgen [-abcdefgjksuv] [-o option]  [-A action] [-G >         return [n]
 complete [-abcdefgjksuv] [-pr] [-DE] [-o option] [-A >         select NAME [in WORDS ... ;] do COMMANDS; done
 compopt [-o|+o option] [-DE] [name ...]                        set [-abefhkmnptuvxBCHP] [-o option-name] [--] [arg >
 continue [n]                                                   shift [n]
 coproc [NAME] command [redirections]                           shopt [-pqsu] [-o] [optname ...]
 declare [-aAfFgilrtux] [-p] [name[=value] ...]                 source filename [arguments]
 dirs [-clpv] [+N] [-N]                                         suspend [-f]
 disown [-h] [-ar] [jobspec ...]                                test [expr]
 echo [-neE] [arg ...]                                          time [-p] pipeline
 enable [-a] [-dnps] [-f filename] [name ...]                   times
 eval [arg ...]                                                 trap [-lp] [[arg] signal_spec ...]
 exec [-cl] [-a name] [command [arguments ...]] [redir>         true
 exit [n]                                                       type [-afptP] name [name ...]
 export [-fn] [name[=value] ...] or export -p                   typeset [-aAfFgilrtux] [-p] name[=value] ...
 false                                                          ulimit [-SHacdefilmnpqrstuvx] [limit]
 fc [-e ename] [-lnr] [first] [last] or fc -s [pat=rep>         umask [-p] [-S] [mode]
 fg [job_spec]                                                  unalias [-a] name [name ...]
 for NAME [in WORDS ... ] ; do COMMANDS; done                   unset [-f] [-v] [name ...]
 for (( exp1; exp2; exp3 )); do COMMANDS; done                  until COMMANDS; do COMMANDS; done
 function name { COMMANDS ; } or name () { COMMANDS ; >         variables - Names and meanings of some shell variabl>
 getopts optstring name [arg]                                   wait [id]
 hash [-lr] [-p pathname] [-dt] [name ...]                      while COMMANDS; do COMMANDS; done
 help [-dms] [pattern ...]                                      { COMMANDS ; }
c7:/home/coop>
```

Commands Built in to bash

Script Parameters:

1. Users often need to pass parameter values to a script, such as a filename, date, etc.
2. Scripts will take different paths or arrive at different values according to the parameters (command arguments) that are passed to them.
3. These values can be text or numbers as in:

```
$ ./script.sh /tmp
$ ./script.sh 100 200
```

4. Within a script, the parameter or an argument is represented with a $ and a number or special character. The table lists some of these parameters.

Parameter	Meaning
`$0`	Script name
`$1`	First parameter
`$2`, `$3`, etc.	Second, third parameter, etc.
`$*`	All parameters
`$#`	Number of arguments

Using Script Parameters:

1. If you type in the script shown in the figure, make the script executable with chmod +x param.sh.
2. Then, run the script giving it several arguments, as shown. The script is processed as follows:

`$0` prints the script name: **`param.sh`**

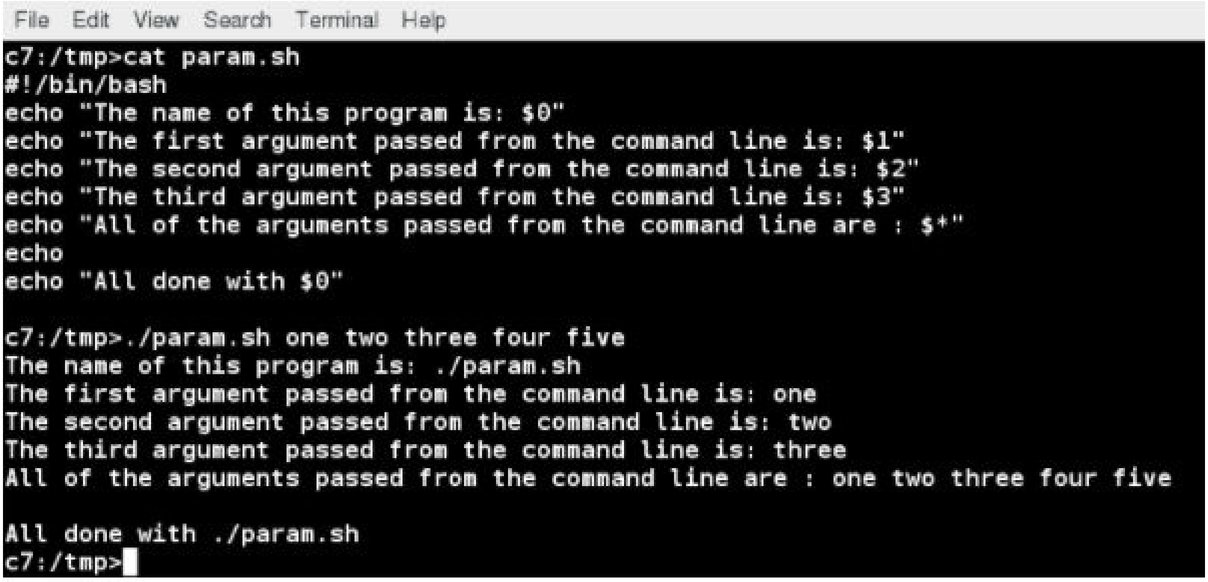
`$1` prints the first parameter: **`one`**

`$2` prints the second parameter: **`two`**

`$3` prints the third parameter: **`three`**

`$*` prints all parameters: **`one two three four five`**

The final statement becomes: **`All done with param.sh`**




```
c7:/tmp>cat param.sh
#!/bin/bash
echo "The name of this program is: $0"
echo "The first argument passed from the command line is: $1"
echo "The second argument passed from the command line is: $2"
echo "The third argument passed from the command line is: $3"
echo "All of the arguments passed from the command line are : $*"
echo
echo "All done with $0"

c7:/tmp>./param.sh one two three four five
The name of this program is: ./param.sh
The first argument passed from the command line is: one
The second argument passed from the command line is: two
The third argument passed from the command line is: three
All of the arguments passed from the command line are : one two three four five

All done with ./param.sh
c7:/tmp>
```

Using Script Parameters

Command Substitution:

1. At times, you may need to substitute the result of a command as a portion of another command. It can be done in two ways:
    - By enclosing the inner command in **$( )**
    - By enclosing the inner command with backticks **(`)**
2. The second, backticks form, is deprecated in new scripts and commands.
3. No matter which method is used, the specified command will be executed in a newly launched shell environment, and the standard output of the shell will be inserted where the command substitution is done.
4. Virtually any command can be executed this way.
5. While both of these methods enable command substitution, the **$( )** method allows command nesting.
6. New scripts should always use this more modern method. For example:

```
$ ls /lib/modules/$(uname -r)/
```

7. In the above example, the output of the command **uname –r** (which will be something like 5.13.3), is inserted into the argument for the ls command.

```
File  Edit  View  Search  Terminal  Help
c7:/tmp>uname -r
4.9.0
c7:/tmp>ls /lib/modules/`uname -r`
build           modules.alias.bin    modules.dep.bin   modules.symbols
kernel          modules.builtin      modules.devname   modules.symbols.bin
misc            modules.builtin.bin  modules.order     source
modules.alias   modules.dep          modules.softdep
c7:/tmp>ls /lib/modules/$(uname -r)
build           modules.alias.bin    modules.dep.bin   modules.symbols
kernel          modules.builtin      modules.devname   modules.symbols.bin
misc            modules.builtin.bin  modules.order     source
modules.alias   modules.dep          modules.softdep
c7:/tmp>
```

Command Substitution

Environment Variables:

1. Most scripts use variables containing a value, which can be used anywhere in the script.
2. These variables can either be user or system-defined.
3. As we discussed earlier, some examples of standard environment variables are **HOME, PATH,** and **HOST**.
4. When referenced, environment variables must be prefixed with the **$** symbol, as in **$HOME**.
5. You can view and set the value of environment variables. For example, the following command displays the value stored in the **PATH** variable:

```
$ echo $PATH
```

6. However, no prefix is required when setting or modifying the variable value.
7. For example, the following command sets the value of the **MYCOLOR** variable to blue:

```
$ MYCOLOR=blue
```

8. You can get a list of environment variables with the **env, set**, or **printenv** commands.

```
student@ubuntu: ~
student@ubuntu:~$ echo $MY_FAVORITE_OS

student@ubuntu:~$ MY_FAVORITE_OS=Linux
student@ubuntu:~$ echo $MY_FAVORITE_OS
Linux
student@ubuntu:~$ env | grep LANG
LANG=en_US.UTF-8
GDM_LANG=en_US
LANGUAGE=en_US
student@ubuntu:~$
```

Exporting Environment Variables:

1. While we discussed the export of environment variables in the section on the "**User Environment**", it is worth reviewing this topic in the context of writing bash scripts.

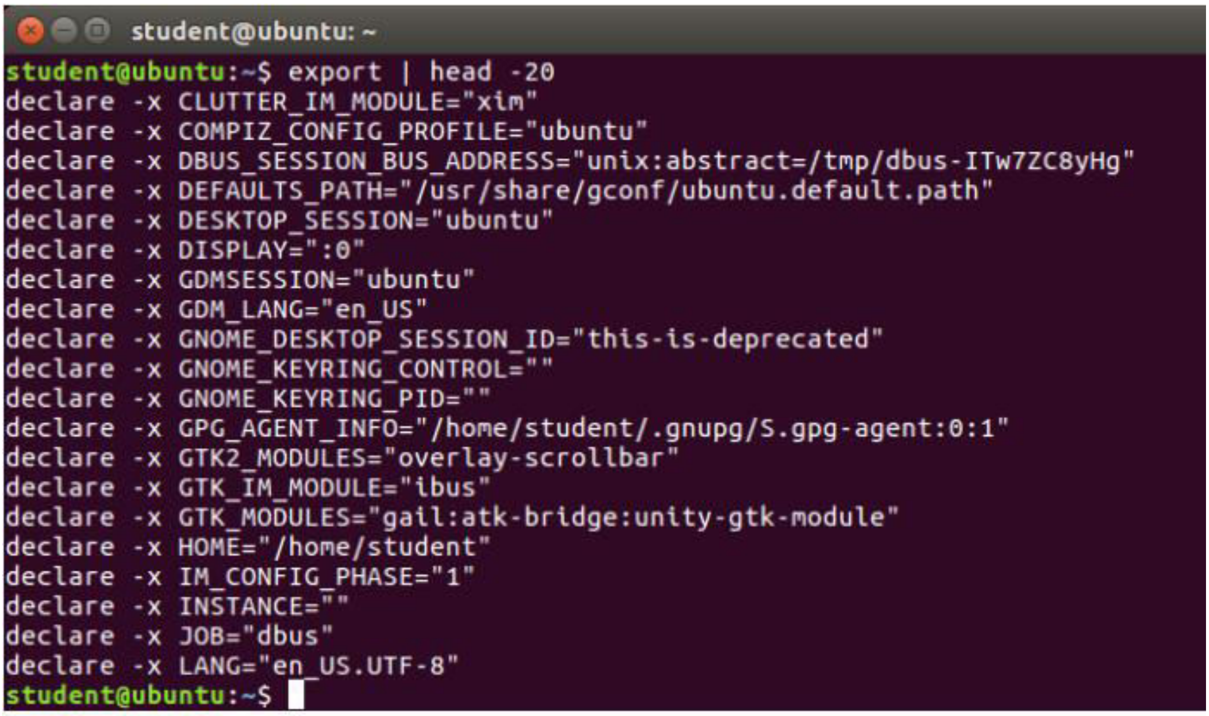
2. By default, the variables created within a script are available only to the subsequent steps of that script.
3. Any child processes (sub-shells) do not have automatic access to the values of these variables.
4. To make them available to child processes, they must be promoted to environment variables using the export statement, as in:

```
export VAR=value
```

or

```
VAR=value ; export VAR
```

5. While child processes are allowed to modify the value of exported variables, the parent will not see any changes; exported variables are not shared, they are only copied and inherited.
6. Typing export with no arguments will give a list of all currently exported environment variables.



```
student@ubuntu: ~
student@ubuntu:~$ export | head -20
declare -x CLUTTER_IM_MODULE="xim"
declare -x COMPIZ_CONFIG_PROFILE="ubuntu"
declare -x DBUS_SESSION_BUS_ADDRESS="unix:abstract=/tmp/dbus-ITw7ZC8yHg"
declare -x DEFAULTS_PATH="/usr/share/gconf/ubuntu.default.path"
declare -x DESKTOP_SESSION="ubuntu"
declare -x DISPLAY=":0"
declare -x GDMSESSION="ubuntu"
declare -x GDM_LANG="en_US"
declare -x GNOME_DESKTOP_SESSION_ID="this-is-deprecated"
declare -x GNOME_KEYRING_CONTROL=""
declare -x GNOME_KEYRING_PID=""
declare -x GPG_AGENT_INFO="/home/student/.gnupg/S.gpg-agent:0:1"
declare -x GTK2_MODULES="overlay-scrollbar"
declare -x GTK_IM_MODULE="ibus"
declare -x GTK_MODULES="gail:atk-bridge:unity-gtk-module"
declare -x HOME="/home/student"
declare -x IM_CONFIG_PHASE="1"
declare -x INSTANCE=""
declare -x JOB="dbus"
declare -x LANG="en_US.UTF-8"
student@ubuntu:~$
```

Exporting Variables

Functions:

1. A function is a code block that implements a set of operations.
2. Functions are useful for executing procedures multiple times, perhaps with varying input variables.
3. Functions are also often called subroutines. Using functions in scripts requires two steps:

- Declaring a function
- Calling a function

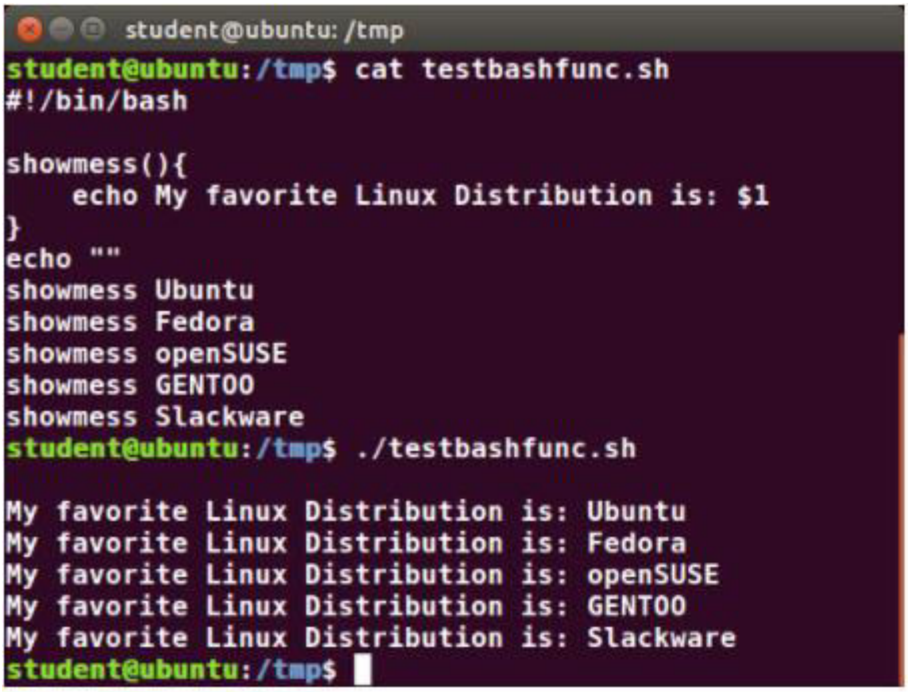
4. The function declaration requires a name which is used to invoke it. The proper syntax is:

```
function_name () {
   command...
}
```

5. For example, the following function is named display:

```
display () {
   echo "This is a sample function"
}
```

6. The function can be as long as desired and have many statements. Once defined, the function can be called later as many times as necessary.
7. In the full example shown in the figure, we are also showing an often-used refinement: how to pass an argument to the function.
8. The first argument can be referred to as **$1,** the second as **$2**, etc.



```
student@ubuntu: /tmp
student@ubuntu:/tmp$ cat testbashfunc.sh
#!/bin/bash

showmess(){
    echo My favorite Linux Distribution is: $1
}
echo ""
showmess Ubuntu
showmess Fedora
showmess openSUSE
showmess GENTOO
showmess Slackware
student@ubuntu:/tmp$ ./testbashfunc.sh

My favorite Linux Distribution is: Ubuntu
My favorite Linux Distribution is: Fedora
My favorite Linux Distribution is: openSUSE
My favorite Linux Distribution is: GENTOO
My favorite Linux Distribution is: Slackware
student@ubuntu:/tmp$
```

Functions

Lab : Working with Files and Directories in a Script:

Write a script which:

1. Prompts the user for a directory name and then creates it with **`mkdir.`**
2. Changes to the new directory and prints out where it is using **`pwd.`**
3. Using **`touch`**, creates several empty files and runs **ls** on them to verify they are empty.
4. Puts some content in them using **`echo`** and redirection.
5. Displays their content using **`cat.`**
6. Says goodbye to the user and cleans up after itself.

Lab Solution: Working with Files and Directories in a Script

Create a file named `testfile.sh`, with the content below.

```
#!/bin/bash

# Prompts the user for a directory name and then creates it  with mkdir.

echo "Give a directory name to create:"
read NEW_DIR

# Save original directory so we can return to it (could also just use pushd, popd)

ORIG_DIR=$(pwd)

# check to make sure it doesn't already exist!

[[ -d $NEW_DIR ]] && echo $NEW_DIR already exists, aborting && exit
mkdir $NEW_DIR

# Changes to the new directory and prints out where it is using pwd.

cd $NEW_DIR
pwd

# Using touch, creates several empty files and runs ls on them to verify they are empty.

for n in 1 2 3 4
do
    touch file$n
done

ls file?

# (Could have just done touch file1 file2 file3 file4, just want to show do loop!)

# Puts some content in them using echo and redirection.

for names in file?
do
    echo This file is named $names > $names
done

# Displays their content using cat

cat file?

# Says goodbye to the user and cleans up after itself

cd $ORIG_DIR
rm -rf $NEW_DIR
echo "Goodbye My Friend!"
```

Make it executable and run it:

```
$ chmod +x testfile.sh
./testfile.sh
```

```
Give a directory name to create:
```

```
/tmp/SOME_DIR
```

```
/tmp/SOME_DIR
file1  file2  file3  file4
This file is named file1
This file is named file2
This file is named file3
This file is named file4
Goodbye My Friend
```

Lab: Passing Arguments:

Write a script that takes exactly one argument, and prints it back out to standard output. Make sure the script generates a usage message if it is run without giving an argument.

Lab Solution: Passing Arguments

Create a file named `testarg.sh`, with the content below.

```
#!/bin/bash
#
# check for an argument, print a usage message if not supplied.
#
if [ $# -eq 0 ] ; then
       echo "Usage: $0 argument"
       exit 1
fi
echo $1
exit 0
```

Make it executable and run it:

```
student:/tmp> chmod +x testarg.sh
student:/tmp> ./testarg.sh Hello

Hello

student:/tmp>./testarg.sh

Usage: ./testarg.sh argument
student:/tmp>
```

Lab: Environment Variables

Write a script which:

1. Asks the user for a number, which should be "1" or "2". Any other input should lead to an error report.
2. Sets an environmental variable to be "Yes" if it is "1", and "No" if it is "2".
3. Exports the environmental variable and displays it.

Lab Solution: Environment Variables

Create a file named `testenv.sh`, with the content below.

```
#!/bin/bash

echo "Enter 1 or 2, to set the environmental variable EVAR to Yes or No"
read ans

# Set up a return code
RC=0

if [ $ans -eq 1 ]
then
    export EVAR="Yes"
else
    if [ $ans -eq 2 ]
    then
        export EVAR="No"
    else
# can only reach here with a bad answer
        export EVAR="Unknown"
        RC=1
    fi
fi
echo "The value of EVAR is: $EVAR"
exit $RC
```

Make it executable and run it:

```
student:/tmp> chmod +x testenv.sh
student:/tmp> ./testenv.sh

Enter 1 or 2, to set the environmental variable EVAR to Yes or No

1

The value of EVAR is: Yes

student:/tmp> ./testenv.sh

Enter 1 or 2, to set the environmental variable EVAR to Yes or No

2

The value of EVAR is: No

student:/tmp> ./testenv.sh

Enter 1 or 2, to set the environmental variable EVAR to Yes or No

3
```

The value of EVAR is: Unknown

Lab: Working with Functions

Write a script which:

1. Asks the user for a number (1, 2 or 3).
2. Calls a function with that number in its name. The function should display a message with its name included.

Lab Solution: Working with Functions

Create a file named **testfun.sh**, with the content below.

```
#!/bin/bash

# Functions (must be defined before use)
func1() {
echo " This message is from function 1"
}
func2() {
echo " This message is from function 2"
}
func3() {
echo " This message is from function 3"
}

# Beginning of the main script

# prompt the user to get their choice
echo "Enter a number from 1 to 3"
read n

# Call the chosen function
func$n
```

Make it executable and run it:

```
student:/tmp> chmod +x testfun.sh
student:/tmp> ./testfun.sh

Enter a number from 1 to 3

2

 This message is from function 2

$ ./testfun.sh

Enter a number from 1 to 3

7

./testfun.sh: line 21: func7: command not found
```