Chapter 3a. shell variables

In this chapter we learn to manage environment **variables** in the shell. These **variables** are often needed by applications.

3a.1. $ dollar sign

Another important character interpreted by the shell is the dollar sign **$**. The shell will look for an **environment variable** named like the string following the **dollar sign** and replace it with the value of the variable (or with nothing if the variable does not exist).

These are some examples using $HOSTNAME, $USER, $UID, $SHELL, and $HOME.

```
[clim@sop ~]$ echo This is the $SHELL shell

This is the /bin/bash shell

[clim@sop ~]$ echo This is $SHELL on computer $HOSTNAME

This is /bin/bash on computer sop.ase.ro

[clim@sop ~]$ echo The userid of $USER is $UID

The userid of clim is 1045

[clim@sop ~]$ echo My homedir is $HOME

My homedir is /home/clim
```

3a.2. case sensitive

This example shows that shell variables are case sensitive!

```
[clim@sop ~]$ echo Hello $USER
Hello clim

[clim@sop ~]$ echo Hello $user
Hello
```

3a.3. creating variables

This example creates the variable **$MyVar** and sets its value. It then uses **echo** to verify the value.

```
[clim@sop gen]$ MyVar=555
[clim@sop gen]$ echo $MyVar 555

[clim@sop gen]$
```

3a.4. quotes

Notice that double quotes still allow the parsing of variables, whereas single quotes prevent this.

```
[clim@sop ~]$ MyVar=555
[clim@sop ~]$ echo $MyVar 555

[clim@sop ~]$ echo "$MyVar" 555

[clim@sop ~]$ echo '$MyVar'

$MyVar
```

The bash shell will replace variables with their value in double quoted lines, but not in single quoted lines.

```
clim@sop:~$ city=Sinaia

clim@sop:~$ echo "We are in $city today."
We are in Sinaia today.

clim@sop:~$ echo 'We are in $city today.'
We are in $city today.
```

3a.5. set

You can use the **set** command to display a list of environment variables. On Ubuntu and Debian systems, the **set** command will also list shell functions after the shell variables.Use **set | more** to see the variables then.

3a.6. unset

Use the **unset** command to remove a variable from your shell environment.

```
[clim@sop ~]$ MyVar=8472
[clim@sop ~]$ echo $MyVar
8472

[clim@sop ~]$ unset MyVar
[clim@sop ~]$ echo $MyVar



[clim@sop ~]$
```

3a.7. $PS1

The **$PS1** variable determines your shell prompt. You can use backslash escaped special characters like **\u** for the username or **\w** for the working directory. The **bash** manual has a complete reference.

In this example we change the value of **$PS1** a couple of times.

```
clim@sop:~$ PS1=prompt
prompt

promptPS1='prompt '
prompt

prompt PS1='> '

>

> PS1='\u@\h$ '
clim@sop$

clim@sop$ PS1='\u@\h:\W$'
clim@sop:~$
```

To avoid unrecoverable mistakes, you can set normal user prompts to green and the root prompt to red. Add the following to your **.bashrc** for a green user prompt:

```
# color prompt by clima
RED='\[\033[01;31m\]'
WHITE='\[\033[01;00m\]'
GREEN='\[\033[01;32m\]'
BLUE='\[\033[01;34m\]'

export PS1="${debian_chroot:+($debian_chroot)}$GREEN\u$WHITE@$BLUE\h$WHITE\w\$ "
```

3a.8. $PATH

The **$PATH** variable is determines where the shell is looking for commands to execute (unless the command is builtin or aliased). This variable contains a list of directories, separated by colons.

```
[[clim@sop ~]$ echo $PATH

/usr/kerberos/bin:/usr/local/bin:/bin:/usr/bin:
```

The shell will not look in the current directory for commands to execute! (Looking for executables in the current directory provided an easy way to hack PC-DOS computers). If you want the shell to look in the current directory, then add a . at the end of your $PATH.

```
[clim@sop ~]$ PATH=$PATH:.
[clim@sop ~]$ echo $PATH

/usr/kerberos/bin:/usr/local/bin:/bin:/usr/bin:.
[clim@sop ~]$
```

Your path might be different when using su instead of **su -** because the latter will take on the environment of the target user. The root user typically has **/sbin** directories added to the $PATH variable.

```
[clim@sop ~]$ su
Password:

[rooty@sop clim]# echo $PATH

/usr/local/bin:/bin:/usr/bin:/usr/S2sop/bin
[rooty@sop clim]# exit

[clim@sop ~]$ su -
Password:

[rooty@sop ~]# echo $PATH

/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin:
[rooty@sop ~]#
```

3a.9. env

The **env** command without options will display a list of **exported variables**. The difference with **set** with options is that **set** lists all variables, including those not exported to child shells.

But **env** can also be used to start a clean shell (a shell without any inherited environment). The **env -i** command clears the environment for the subshell.

Notice in this screenshot that **bash** will set the **$SHELL** variable on startup.

```
[clim@sop ~]$ bash -c 'echo $SHELL $HOME $USER'

/bin/bash /home/clim clim

[clim@sop ~]$ env -i bash -c 'echo $SHELL $HOME $USER'

/bin/bash
[clim@sop ~]$
```

You can use the **env** command to set the **$LANG**, or any other, variable for just one instance of **bash** with one command. The example below uses this to show the influence of the **$LANG** variable on file globbing (see the chapter on file globbing).

```
[clim@sop test]$ env LANG=C bash -c 'ls File[a-z]'

Filea  Fileb

[clim@sop test]$ env LANG=en_US.UTF-8 bash -c 'ls File[a-z]'
```

3a.10. export

You can export shell variables to other shells with the **export** command. This will export the variable to child shells.

```
[clim@sop ~]$ var3=three
[clim@sop ~]$ var4=four

[clim@sop ~]$ export var4
[clim@sop ~]$ echo $var3 $var4

three four

[clim@sop ~]$ bash [clim@sop ~]$
echo $var3 $var4

four
```

But it will not export to the parent shell (previous screenshot continued).

```
[clim@sop ~]$ export var5=five
[clim@sop ~]$ echo $var3 $var4 $var5
four five

[clim@sop ~]$ exit
exit

[clim@sop ~]$ echo $var3 $var4 $var5
three four

[clim@sop ~]$
```

3a.11. delineate variables

Until now, we have seen that bash interprets a variable starting from a dollar sign, continuing until the first occurrence of a non-alphanumeric character that is not an underscore. In some situations, this can be a problem. This issue can be resolved with curly braces like in this example.

```
[clim@sop ~]$ prefix=Super

[clim@sop ~]$ echo Hello $prefixman and $prefixgirl
Hello  and

[clim@sop ~]$ echo Hello ${prefix}man and ${prefix}girl
Hello Superman and Supergirl

[clim@sop ~]$
```

3a.12. unbound variables

The example below tries to display the value of the **$MyVar** variable, but it fails because the variable does not exist. By default the shell will display nothing when a variable is unbound (does not exist).

```
[clim@sop bin]$ echo $MyVar



[clim@sop bin]$
```

There is, however, the **nounset** shell option that you can use to generate an error when a variable does not exist.

```
clim@sop:~$ set -u
clim@sop:~$ echo $Myvar

bash: Myvar: unbound variable
clim@sop:~$ set +u
clim@sop:~$ echo $Myvar



clim@sop:~$
```

In the bash shell **set -u** is identical to **set -o nounset** and likewise **set +u** is identical to **set +o nounset**.

3a.13. practice: shell variables

3a.1.1. Use echo to display Hello followed by your username. (use a bash variable!)

3a.2.1. Create a variable **answer** with a value of **42**.

3a.3.1. Copy the value of $LANG to $MyLANG.

3a.4.1. List all current shell variables.

3a.5.1. List all exported shell variables.

3a.6.1. Do the **env** and **set** commands display your variable ?

6. Destroy your **answer** variable.

7. Create two variables, and **export** one of them.

8. Display the exported variable in an interactive child shell.

9. Create a variable, give it the value 'Dumb', create another variable with value 'do'. Use **echo** and the two variables to echo Dumbledore.

10. Find the list of backslash escaped characters in the manual of bash. Add the time to your **PS1** prompt.