Basic Shell Scripting

Important things to keep in mind.

- Linux is case sensitive. (for example: a "**PATH**" is not the same as "**path**")

- Make sure that your scripts have **#!/usr/bin/bash** as the first entry on the first line of the script.

- Scripts are read sequentially, so Line after line.

- It is a good idea to start your scripts with **variables** and then **functions** and then the main part of the script.

- It is best practise to add comments to what commands do in your script so that other know what is meant to happen.

- If you get stuck in any way, use the internet or ask your neighbour to help.

VARIABLES

There are different types of variables:

- Regular variables

A regular variable is set in the process of each shell independently. Some regular variables are set by default and should not be changed by the user.

For example: **PWD**, **SHELL**, **PID** and more are set by the system.

Users can set their own variables. They will only be set in the shell the user sets them in. They will not be set in the shell of another user.

For example:
```
MYVAR=abcd
counter=100
times=2
```

Regular variables can also be set in the script by asking the user to fill them. This is done by using the 'read' command.

For example:
```
read MYVAR
read counter
```

- Positional parameters

A positional parameter is passed as an argument to the script you start.

For example: `~/myscript param1 param2 param3`
In this example you can use the variables param1 param2 and param3

The names of positional parameters depend on the position when you start the script.
In the above example **~/myscript** is **$0**, **param1** is **$1** et cetera.

If the user does not pass the variables when starting the script, you can use the 'read' command to fill them and then use the 'set' command to set the parameters.

For example:
```
read first
read second
set $first $second
```

Now $1 contains the value of $first and $2 contains the value of $second

- Special variables

Special variables are set automatically. The user does not set them.

- **$!** (The PID of the last background process started by the shell)
- **$?** (The return code of the exited child process)
- **$*** (the value of all positional parameters)
- **$#** (the number of positional parameters)
- **$$** (the PID of the current shell)
- **$_** (the last argument of the last started command)

ALIASES and FUNCTIONS

Aliases and Function have the same purpose. The difference is that an alias is always a single line. It is like putting a command or a sequence of commands in a single variable. A function is like a script that you create in the shell and not on disk.

For example:

```
alias ll="ls -l"
alias dir1="cd /home/rocky/ ; pwd"

function list {
      ls ~
      echo "these are the files in your login dir."
      echo -n "are you happy with this?"
      read ANSWER
      echo "Thanks."
            }

function cleanup {
      clear
      echo "bye"
      sleep 2
      exit
      }
```

Usage of functions in scripts is very common. When you have repetitive sequences of commands, it is best practise to create functions so that you can simply call the functions in the script instead of having to run a collection of commands multiple times.

test and while

The test instruction tests whether something is true or not. For example, to test whether the value of a particular variable is greater, less then or equal to a number.

```
var=10
[ $var -lt 20 ]
echo $?
0
```

This checks whether the value of the var variable is less than 20 or not. If it is less than 20 the return code will be 0 (so it is true), if it is not less than 20 the return code will be 1 (which is false).

Another example: check whether a file is executable or not.

```
ls -l f1
-rw-r--r--. 1 root root 0 Mar 12 19:12 f1
[ -x f1 ]
echo $?
1
```

The test command is commonly used with another command like for example the **while** command. The while command will run one or more commands as long as the outcome of the **test** command is **true**.

For example: to display a variable until it drops below a particular value.

```
var=10
while [ $var -gt 9 ]
> do
> echo $var is greater than 9
> read var
> done
```

Another example. This is an infinite loop. The true command always returns 0. So 0 is always true and true is always 0.

```
while true
> do
> echo true is always true
> sleep 2
```

```
> done
true is always true
true is always true
true is always true
true is always true
^C
```

The if command

The if command is a form of branching. This means that, based upon a particular check, the command will perform one set of action or another set of actions.

So for example: the script asks a user to choose between “yes” or “no”. If the user types **yes**, then the script will other commands than if the user types **no**.

To determine which branch to take, the **if** command checks whether something is true or not,

So, for example. to check whether the user enter “yes” or “no”, you could use the following commands:

```
echo -n “enter yes or no”
read answer
if [ $answer == yes ]
then
      echo “you entered yes”
else
      echo “you entered no”
fi
```

In the above example the test command returns true or false. Depending on that outcome the if command takes one branch or the other.

Another example: You want to check whether a file exists or not and if it does not exist your have to create it.

```
if ls myfile  2> /dev/null
then
      echo “myfile exists”
else
      touch myfile
fi
```

The **if** statement in the above script checks whether the **ls** command returns a 0 or not, and acts upon that.

:

The case command

The case command is like the if command. It checks whether something is true or not. This command is very often used to create a menu.

First an example:

```
clear
echo "1=reboot"
echo "2=halt"
echo "3=exit"
echo -n "input : "
read input

case $input in
1)reboot;;
2)halt;;
3)exit;;
*)echo "incorrect choice";;
esac
```

The above script first displays a menu, then asks for input.
If the input is 1 the reboot command will be run; if 2, the halt command will be run; if 3 the script wil exit; if anything else, an error will be shown.

The for loop

The for loop will run one or more commands multiple times.
For example, if you want to run display the value of a variable for a predefined number of times:

```
for var in a b c d e f g
> do
> echo $var
> done
a
b
c
d
e
f
g
```

Another example. Run the echo command 10 times.

```
for number in {1..10}
> do
> echo $number
> done
1
2
3
4
5
6
7
8
9
10
```

The above example could also be like this:

```
for number in 1 2 3 4 5 6 7 8 9 10
>do
>echo $number
done
```

A practical example. In a directory you want to rename all files by adding **.txt** to the name. NOTE! **$(ls)** is instruction substitution and in this case it will list all files in a directory and rename each file by adding **.txt**.

```
ls
a  b  c  d

for name in $(ls)
> do
> mv $name $name.txt
> done

ls
a.txt  b.txt  c.txt  d.txt
```

Debugging scripts

To help you finding mistakes in your script you can add the following line at the top of your script: **set -x**

This will print how commands are executed.
For example. The following script asks for a number less than 10.

```
#!/usr/bin/bash
set -x
echo -n "enter a number that is less than 10 : "
read num
if [ $num  -lt 10 ]
then
      echo "ok"
else
      echo "not ok"
fi
```

And when you run the script it will tell you what it does.

```
./check
+ echo -n 'enter a number that is less than 10 : '
enter a number that is less than 10 : + read num
4
+ '[' 4 -lt 10 ']'
+ echo ok
ok
```