Linux administration with Bash. Lection 2

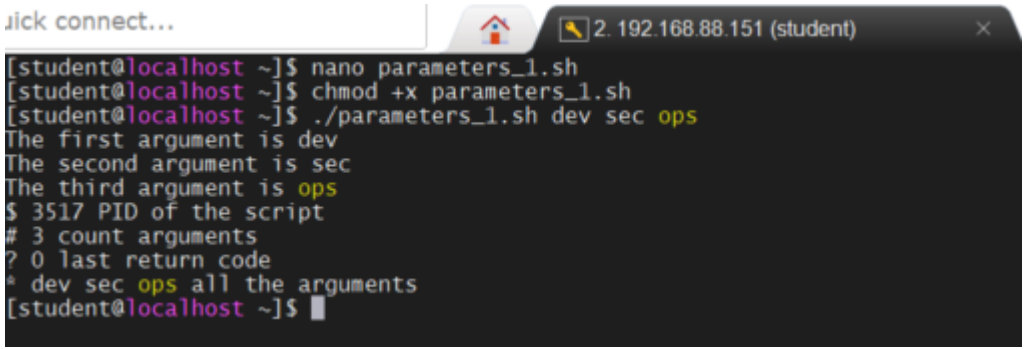
Bash

- Scripting
- Q&A.

Bash. Scripting parameters.

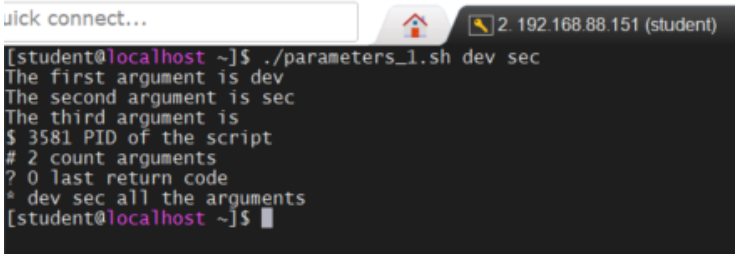
A ***bash*** shell script can have parameters. The numbering you see in the script below continues if you have more parameters. You also have special parameters containing the number of parameters, a string of all of them, and also the process id, and the last return code. The ***man*** page of ***bash*** has a full list.

```
#!/bin/bash
echo The first argument is $1
echo The second argument is $2
echo The third argument is $3
echo \$ $$  PID of the script
echo \# $#  count arguments
echo \? $?  last return code
echo \* $*  all the arguments
```



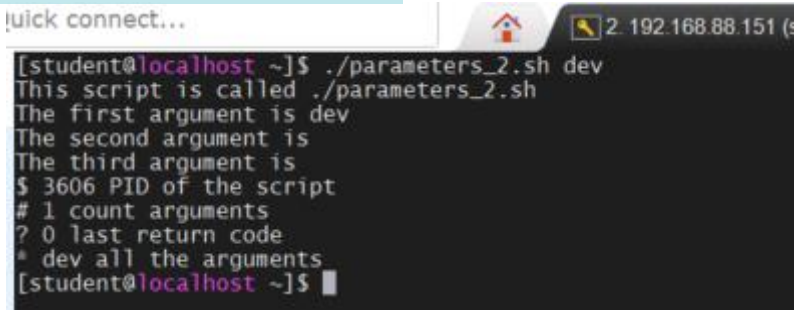
Bash. Scripting parameters.

Once more the same script, but with only two parameters



Here is another example, where we use $0.
The $0 parameter contains the name of the script.

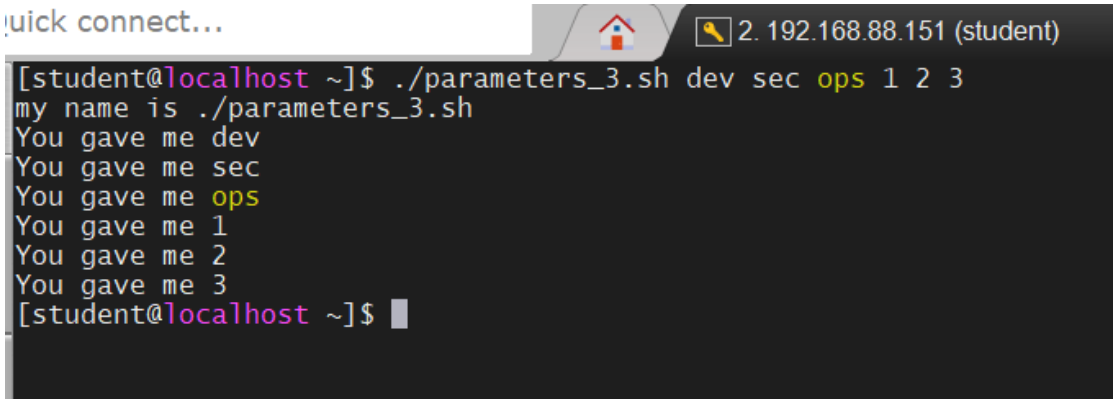
```
!/bin/bash
echo This script is called $0
echo The first argument is $1
echo The second argument is $2
echo The third argument is $3
echo \$ $$  PID of the script
echo \# $#  count arguments
echo \? $?  last return code
echo \* $*  all the arguments
```



Bash. Shift through parameters.

The shift statement can parse all parameters one by one. This is a sample script.

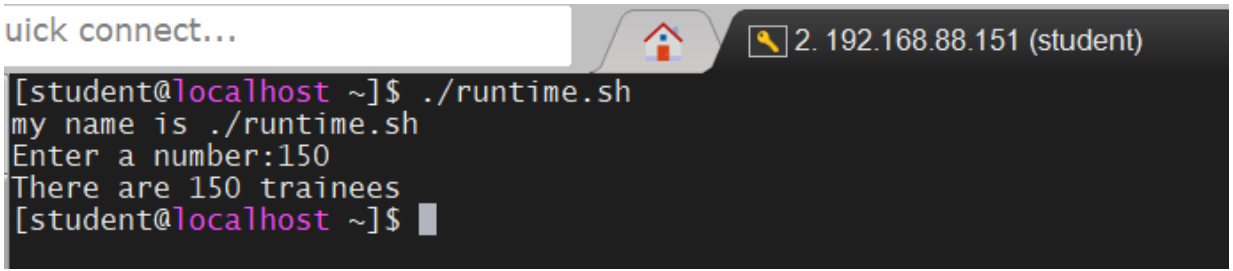
```
#!/bin/bash
echo my name is $0
if [ "$#" == "0" ]  then
        echo You have to give at
least one parameter.
        exit 1
fi
while (( $# ))
do
        echo You gave me $1
        shift
done
```



Bash. Runtime input.

You can ask the user for input with the ***read*** command in a script

```
#!/bin/bash
echo my name is $0
echo -n Enter a number:
read number
echo There are $number trainees
```



Bash. Sourcing a config file.

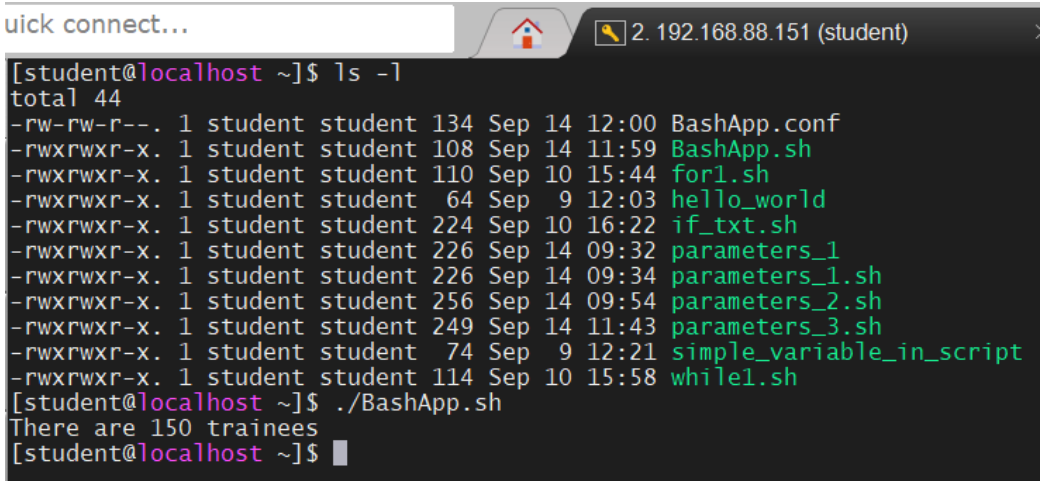
The ***source*** can be used to source a configuration file. Below a sample configuration file for an application

```
# The config file of BashApp
# Enter the path here
BashAppPath=/home/student/myApp
# Enter the number of trainees here
trainees=150
```

And here an application that uses this file:

```
#!/bin/bash
#
# Welcome to the BashApp
application
#
. ./BashApp.conf
echo There are $trainees trainees
```

Output:

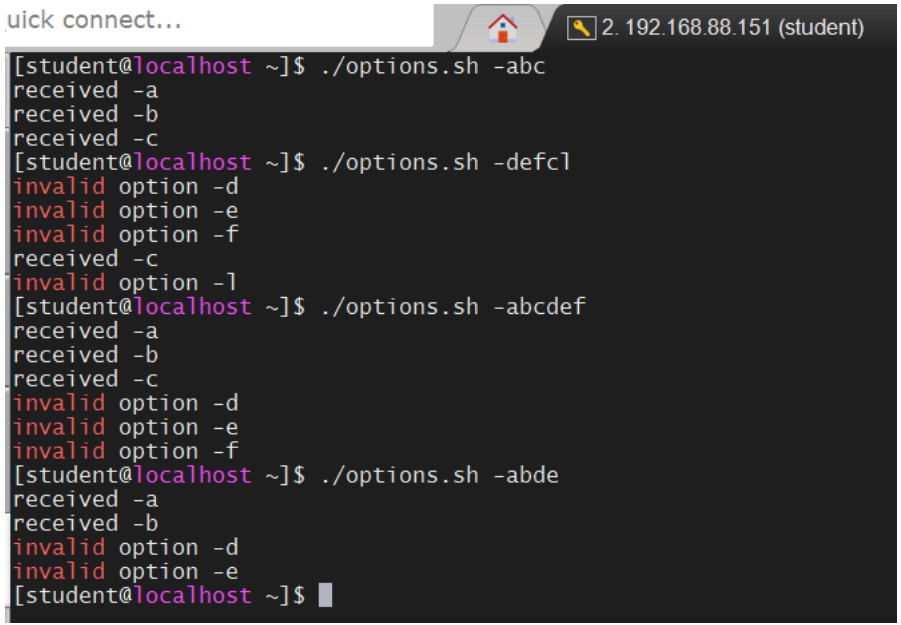


Bash. Get script options with getopts.

The ***getopts*** function allows you to parse options given to a command. The following script allows for any combination of the options a, b and c

```
#!/bin/bash
while getopts ":abc" option;
do
 case $option in
  a)
      echo received -a   ;;
  b)
      echo received -b   ;;
  c)
      echo received -c   ;;
  *)
      echo "invalid option -$OPTARG"    ;;
 esac
done
```

Output:

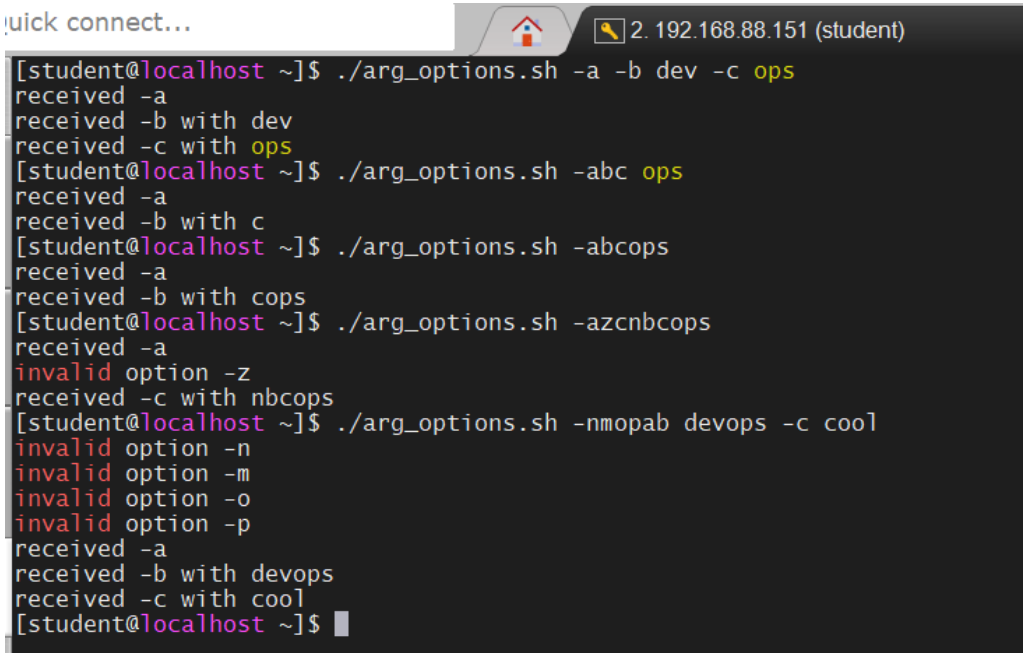


Bash. Get script options with getopts.

You can also check for options that need an argument, as this example shows.

```
#!/bin/bash
while getopts ":ab:c:" option;
do
 case $option in
  a)
   echo received -a   ;;
  b)
   echo received -b with $OPTARG   ;;
  c)
   echo received -c with $OPTARG  ;;
  :)
   echo "option -$OPTARG needs an argument"   ;;
  *)
   echo "invalid option -$OPTARG"    ;;
esac
done
```

Output:



Bash. Additional scripting elements.

eval reads arguments as input to the shell (the resulting commands are executed).
This allows using the value of a variable as a variable.

> answer=42
> word=answer
> eval x=\$$word ; echo $x
> 42

In bash the arguments can be quoted

> answer=42
> word=answer
> eval "y=\$$word" ; echo $x
> 42

Bash. Additional scripting elements.

Sometimes the ***eval*** is needed to have correct parsing of arguments. Consider this example where the ***date*** command receives one parameter ***1 week ago***

```
[student@localhost ~]$ date --date="1 week ago"
Mon Sep  7 23:38:02 EEST 2020
```

When we set this command in a variable, then executing that variable fails unless we use eval

```
[student@localhost ~]$ lastweek='date --date="1 week ago"'
[student@localhost ~]$ $lastweek
date: extra operand ‘ago"’
Try 'date --help' for more information.
[student@localhost ~]$ eval $lastweek
Mon Sep  7 23:44:33 EEST 2020
[student@localhost ~]$
```

Bash. Additional scripting elements.

The (( )) allows for evaluation of numerical expressions

```
> (( 42 > 33 )) && echo true || echo false
> true
>  (( 42 > 1201 )) && echo true || echo false
> false
> var42=42
> (( 42 == var42 )) && echo true || echo false
> true
> (( 42 == $var42 )) && echo true || echo false
> true
> var42=33
> (( 42 == var42 )) && echo true || echo false
> false
```

Bash. Additional scripting elements.

The let built-in shell function instructs the shell to perform an evaluation of arithmetic expressions.

```
[student@localhost ~]$ let x="3 + 4" ; echo $x
7
[student@localhost ~]$ let x="10 + 100/10" ; echo $x
20
[student@localhost ~]$ let x="10-2+100/10" ; echo $x
18
[student@localhost ~]$ let x="10*2+100/10" ; echo $x
30
```

The shell can also convert between different bases.

```
[student@localhost ~]$ let x="0xFF" ; echo $x
255
[student@localhost ~]$ let x="0xC0" ; echo $x
192
[student@localhost ~]$ let x="0xA8" ; echo $x
168
[student@localhost ~]$ let x="8#70" ; echo $x
56
[student@localhost ~]$ let x="8#77" ; echo $x
63
[student@localhost ~]$ let x="16#c0" ; echo $x
192
```

There is a difference between assigning a variable directly, or using let to evaluate the arithmetic expressions (even if it is just assigning a value).

```
[student@localhost ~]$ dec=15 ; oct=017 ; hex=0x0f
[student@localhost ~]$ echo $dec $oct $hex
15 017 0x0f
[student@localhost ~]$ let dec=15 ; let oct=017 ; let hex=0x0f
[student@localhost ~]$ echo $dec $oct $hex
15 15 15
```

Bash. Additional scripting elements.

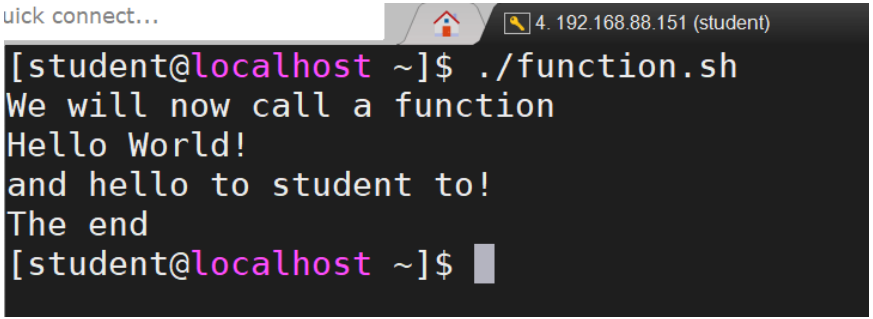
You can sometimes simplify nested ***if*** statements with a ***case*** construct

```
#!/bin/bash
# Job Helpdesk Advisor :-)
echo -n "What job do you want ? "
read job
case $job in
        "devops")
                echo "Excellent"
        ;;
        "dev")
                echo "Good"
        ;;
        "test")
                echo "not bad."
        ;;
        "frontend")
                echo "Really???"
        ;;
        *)
                echo "Make your choise once more from: devops, dev, test and frontend"
        ;;
esac
```

Bash. Additional scripting elements.

Shell ***functions*** can be used to group commands in a logical way.

```
#!/bin/bash
function greetings {
echo Hello World!
echo and hello to $USER to!
}
echo We will now call a function
greetings
echo The end
```

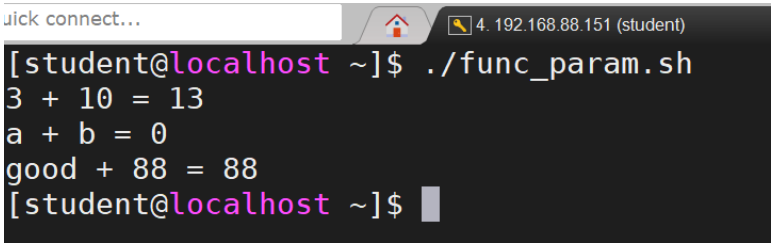


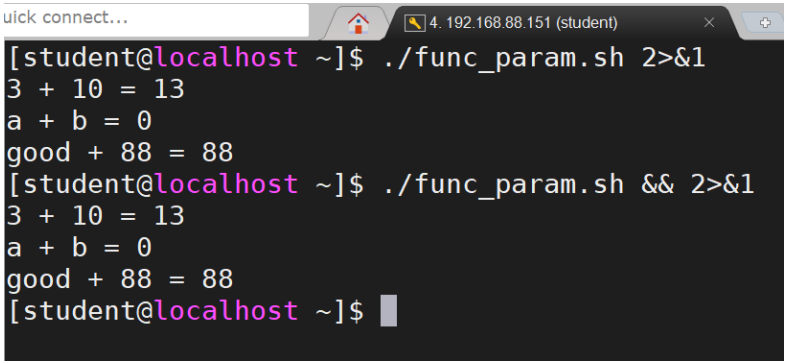
Bash. Additional scripting elements.

A shell ***function*** can also receive parameters

```
#!/bin/bash
function plus {
 let result="$1 + $2"
 echo  $1 + $2 = $result
}

plus 3 10
plus a b
plus good 88
```





Bash. Shell expansions.Quotes

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

$ MyVar=555
$ echo $MyVar
555
$ echo "$MyVar"
555
$ echo '$MyVar'
MyVar

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

$ city=Burtonville
$ echo "We are in $city today."
We are in Burtonville today.
$ echo 'We are in $city today.'
We are in $city today.

Bash. Shell expansions. Backticks or single quotes

Single embedding can be useful to avoid changing your current directory. The screenshot below uses backticks instead of dollar-bracket to embed.

```
$ echo `cd /etc; ls -d * | grep pass`
passwd passwd- passwd.OLD
$
```

Placing the embedding between backticks uses one character less than the dollar and parenthesis combo. Be careful however, backticks are often confused with single quotes. The technical difference between ' and ` is significant!

```
$ echo `var1=5;echo $var1`
5
$ echo 'var1=5;echo $var1'
var1=5;echo $var1
$
```

Bash. Shell expansions.

Read a File:
You can read any file line by line in bash by using loop. Create a file named, '**read_file.sh**' and add the following code to read an existing file named, '**book.txt**'.

```
  GNU nano 2.3.1                File: read_file.sh

#!/bin/bash
file='books.txt'
while read line; do
echo $line
done < $file
```

```
uick connect...    2. 192.168.88.151 (student)
student@localhost~$ bash read_file.sh
Project Phoenix
Continious integration
Working with Jenkins
Ansible forever
student@localhost~$
```

Q & A

Thank you!