DevOps external course

Linux administration with Bash. Lection 2

Lecture 7.2

Module 7 **Linux Administration + Bash**

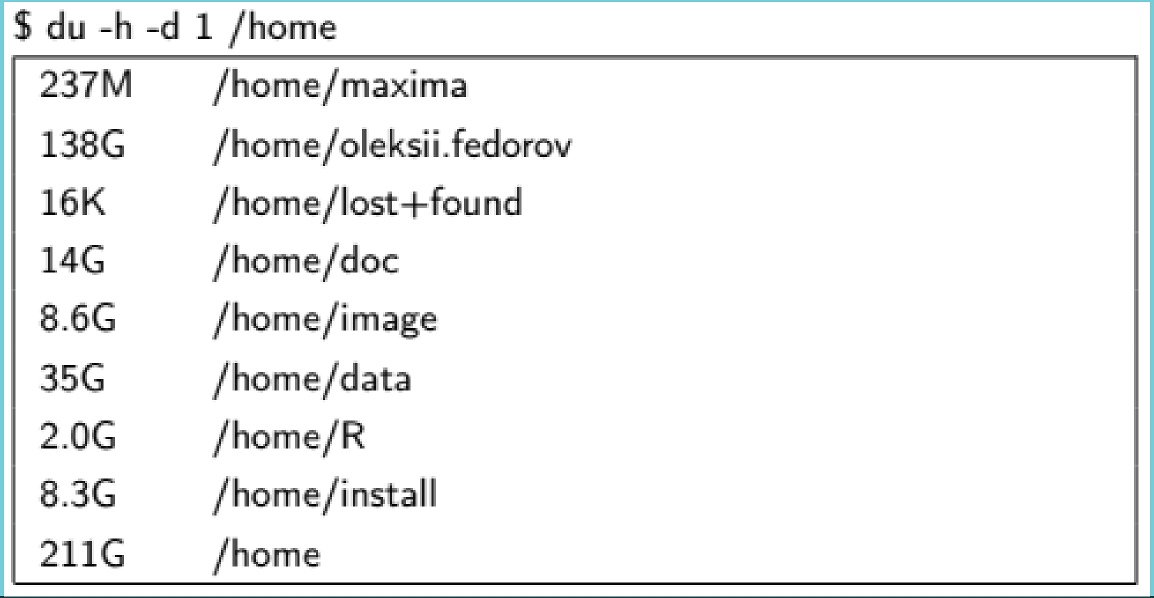
Serge Prykhodchenko

Agenda

- Administration
- Scripting
- Q&A

Monitoring of file system usage

Du (disk usage) command
The du command recursively displays the size of all directories in the specified (current) directory.
The -d parameter specifies the recursion depth.



```
$ du -h -d 1 /home
237M      /home/maxima
138G      /home/oleksii.fedorov
16K       /home/lost+found
14G       /home/doc
8.6G      /home/image
35G       /home/data
2.0G      /home/R
8.3G      /home/install
211G      /home
```

Daemon udevd

The udevd daemon works as follows.

1. The kernel sends a notification event called uevent to the udevd daemon.
2. The udevd daemon loads all the attributes contained in the uevent notification.
3. The udevd daemon parses the rules and then takes action or sets additional attributes based on the rules. An incoming uevent might look like this:

```
ACTION=change
DEVNAME=sde
DEVPATH=/ d e v i c e s / p c i 0 0 0 0 : 0 0 / 0 0 0 0 : 0 0 : 1 a . 0 / u sb1 /1 -1/1 -1.2/1 -1.2:1.0/
h o s t 4 / t a r g e t 4 : 0 : 0 / 4 : 0 : 0 : 3 / b l o c k / sd e
DEVTYPE=d i s k
DISK_MEDIA_CHANGE=1
MAJOR=8
MINOR=64
SEQNUM=2752
SUBSYSTEM=b l o c k
UDEV_LOG=3
```

Command udevadm

1. The udevadm command is the administration tool for the udevd manager. It can be used to reload udevd rules and trigger events.
2. The udevadm command allows you to search for and discover system devices and monitor uevents.

For device information

```
$ udevadm info –query=all —name=/dev/sda
```

To track uevents notifications

```
$ udevadm [–kernel,–udev] monitor
```

blkid

To view the list of devices corresponding to them FS, as well as UUIDs, use the blkid (block ID) command:

```
#blkid
/dev/s d f 2 : UUID="a9011c2b -1c03 -4288-b3fe -8ba961ab0898 " TYPE="e x t 4 "
/dev/sda1 : UUID="70 ccd6e7 -6ae6 -44f6 -812c-51aab8036d29 " TYPE="e x t 4 "
/dev/sda5 : UUID="592 dcfd1 -58da-4769-9ea8-5f412a896980 " TYPE="swap "
/dev/sde1 : SEC_TYPE="msdos " UUID="3762-6138" TYPE=" v f a t “
/dev/sr0: UUID="2019-12-10-15-09-23-93" LABEL="VBox_GAs_6.1.0" TYPE="iso9660"
```

To mount a filesystem by its UUID, use the syntax UUID =

```
# mount UUID=a9011c2b-1c03-4288-b3fe-8ba961ab0898 /home/extra
```

Файл /etc/fstab

To mount filesystems at boot time and save the mount command from the tedious work, Linux permanently maintains a list of filesystems and their parameters in the / etc / fstab table.

Example of /etc/fstab

```
# /etc/fstab: static file system information.
#
# Use 'blkid' to print the universally unique identifier for a
# device; this may be used with UUID= as a more robust way to name devices
# that works even if disks are added and removed. See fstab(5).
#
# <file system> <mount point>   <type>  <options>       <dump>  <pass>
# / was on /dev/sda2 during curtin installation
/dev/disk/by-uuid/67820e75-cae0-4083-b1ed-69db08a44f8f / ext4 defaults 0 0
/swap.img       none    swap    sw      0       0
```

hostnamectl

The **hostnamectl** utility is part of systemd, and it is used to query and change the system hostname. It also displays the Linux distribution and kernel version:

```
# hostnamectl
  Static hostname:  linuxize.localdomain
        Icon name: computer-laptop
        Chassis: laptop
        Machine ID: af8ce1d394b844fea8c19ea5c6a9bd09
        Boot ID: 15bc3ae7bde842f29c8d925044f232b9
        Operating System: Ubuntu 18.04.2 LTS
        Kernel: Linux 4.15.0-54-generic
        Architecture: x86-64
```

You can use the grep command to filter out the Linux kernel version:

```
# hostnamectl | grep -i kernel
```

Ports scaning

A listening port is a network port that an application listens on. You can get a list of the listening ports on your system by querying the network stack with commands such as **ss**, **netstat** or **lsof**. Each listening port can be open or closed (filtered) using a firewall.

The following command issued from the console determines which ports are listening for TCP connections from the network:

sudo nmap -sT -p- 10.10.8.8

The -sT tells nmap to scan for TCP ports and -p- to scan for all 65535 ports. If -p- is not used nmap will scan only the 1000 most popular ports.

Netcat (or nc) is a command-line tool that can read and write data across network connections, using the TCP or UDP protocols.

For example to scan for open TCP ports on a remote machine with IP address 10.10.8.8 in the range 20-80 you would use the following command:

nc -z -v 10.10.8.8 20-80

The -z option tells nc to scan only for open ports, without sending any data and the -v is for more verbose information.

Ports scaning

Check Open Ports using Bash Pseudo Device

Another way to check whether a certain port is open or closed is by using the Bash shell /dev/tcp/.. or /dev/udp/.. pseudo-device.
When executing a command on a /dev/$PROTOCOL/$HOST/$PORT pseudo-device, Bash will open a TCP or UDP connection to the specified host on the specified port.

```
for PORT in {20..80}; do
  timeout 1 bash -c "</dev/tcp/10.10.8.8/$PORT &>/dev/null" &&  echo "port $PORT is open"
done
```

How does the code above works?

When connecting to a port using a pseudo-device, the default timeout is huge, so we are using the timeout command to kill the test command after 5 seconds. If the connection is established to 10.10.8.8 port 443 the test command will return true.

https://linuxize.com/post/check-open-ports-linux/

Functions

Like "real" programming languages, Bash has functions, though in a somewhat limited implementation. A function is a subroutine, a code block that implements a set of operations, a "black box" that performs a specified task. Wherever there is repetitive code, when a task repeats with only slight variations in procedure, then consider using a function.

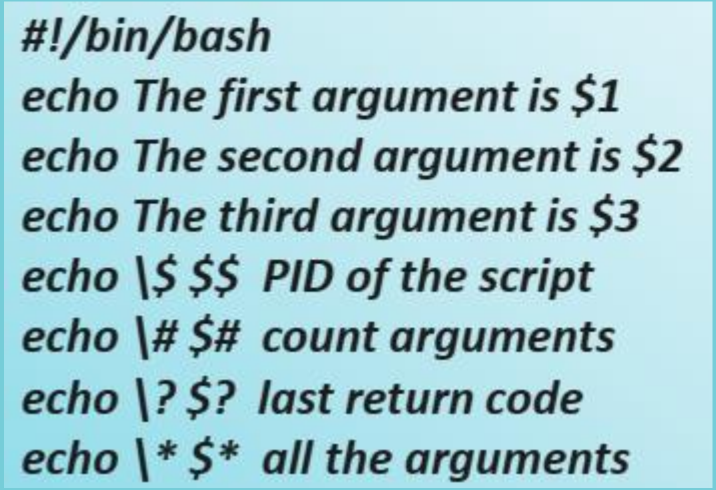
```
function function_name {
command...
}
or
function_name2 () {
command...
}

 # Now, call the functions.
function_name1
function_name2
exit $?
```

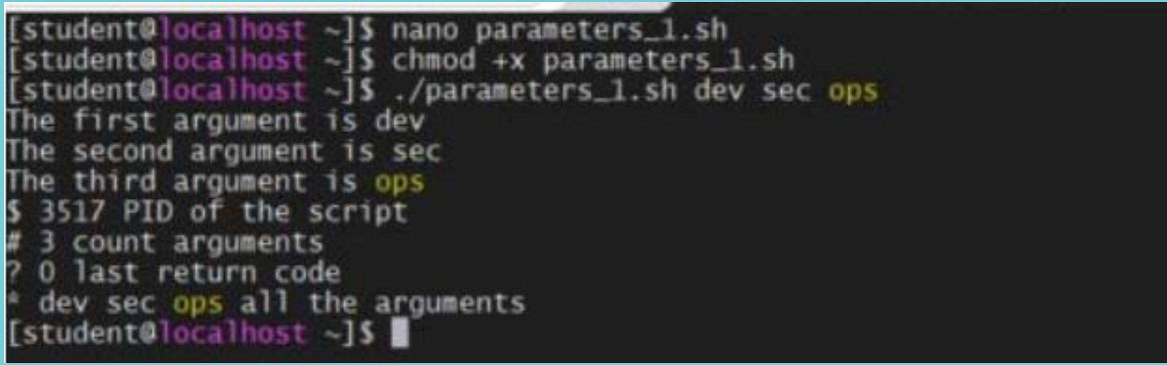
SCRIPTING

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.



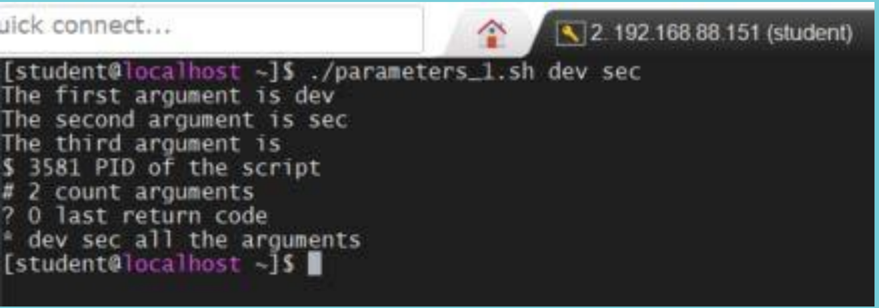
```
#!/bin/bash
echo The first argument is $1
echo The second argument is $2
echo The third argument is $3
echo \$ $$  PID of the script
echo \# $#  count arguments
echo \? $?  last return code
echo \* $*  all the arguments
```



```
[student@localhost ~]$ nano parameters_1.sh
[student@localhost ~]$ chmod +x parameters_1.sh
[student@localhost ~]$ ./parameters_1.sh dev sec ops
The first argument is dev
The second argument is sec
The third argument is ops
$ 3517 PID of the script
# 3 count arguments
? 0 last return code
* dev sec ops all the arguments
[student@localhost ~]$ 
```

Bash. Scripting parameters.

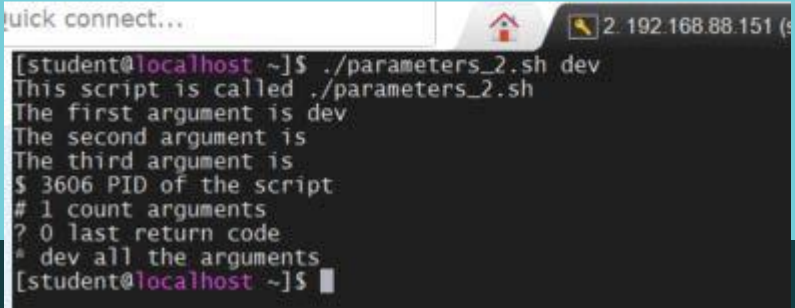
Once more the same script, but with only two parameters



```
[student@localhost ~]$ ./parameters_1.sh dev sec
The first argument is dev
The second argument is sec
The third argument is
$ 3581 PID of the script
# 2 count arguments
? 0 last return code
* dev sec all the arguments
[student@localhost ~]$
```

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
```

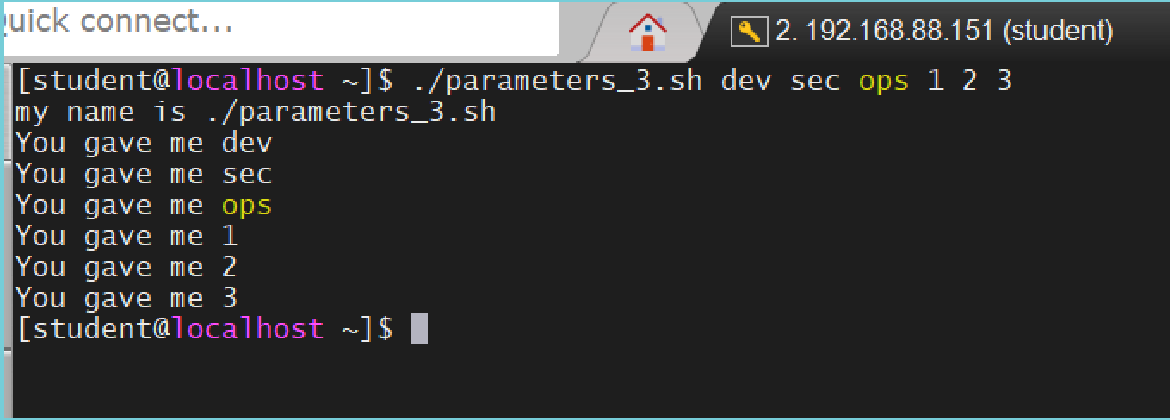


```
[student@localhost ~]$ ./parameters_2.sh dev
This script is called ./parameters_2.sh
The first argument is dev
The second argument is
The third argument is
$ 3606 PID of the script
# 1 count arguments
? 0 last return code
* dev all the arguments
[student@localhost ~]$
```

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
```

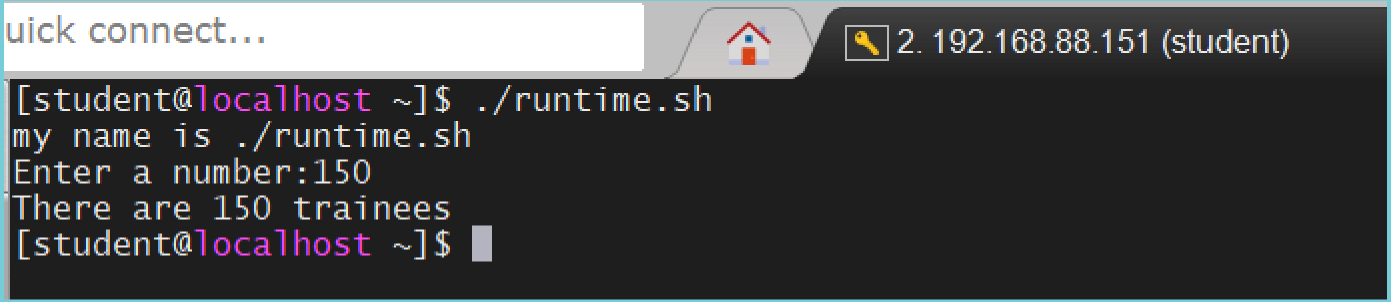


```
[student@localhost ~]$ ./parameters_3.sh dev sec ops 1 2 3
my name is ./parameters_3.sh
You gave me dev
You gave me sec
You gave me ops
You gave me 1
You gave me 2
You gave me 3
[student@localhost ~]$
```

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
```



```
[student@localhost ~]$ ./runtime.sh
my name is ./runtime.sh
Enter a number:150
There are 150 trainees
[student@localhost ~]$
```

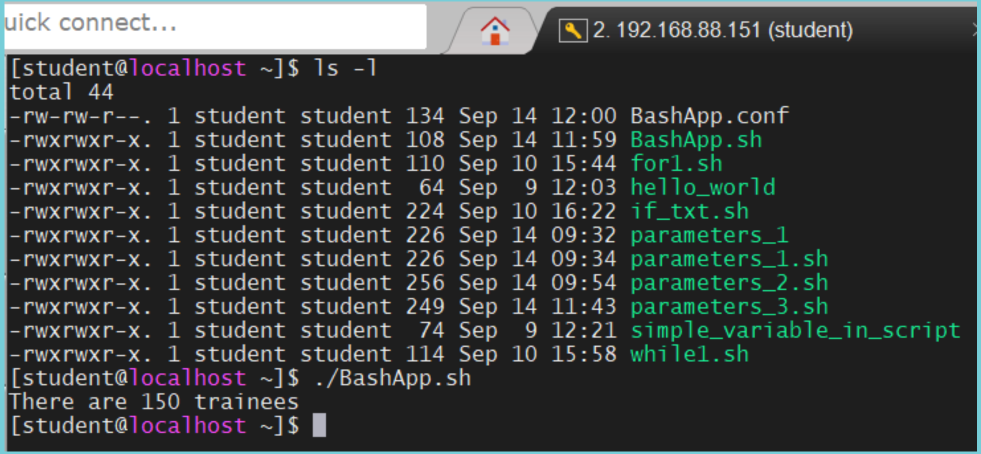
Bash. Sourcing a config file.

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



```
[student@localhost ~]$ ls -l
total 44
-rw-rw-r--. 1 student student 134 Sep 14 12:00 BashApp.conf
-rwxrwxr-x. 1 student student 108 Sep 14 11:59 BashApp.sh
-rwxrwxr-x. 1 student student 110 Sep 10 15:44 for1.sh
-rwxrwxr-x. 1 student student  64 Sep  9 12:03 hello_world
-rwxrwxr-x. 1 student student 224 Sep 10 16:22 if_txt.sh
-rwxrwxr-x. 1 student student 226 Sep 14 09:32 parameters_1
-rwxrwxr-x. 1 student student 226 Sep 14 09:34 parameters_1.sh
-rwxrwxr-x. 1 student student 256 Sep 14 09:54 parameters_2.sh
-rwxrwxr-x. 1 student student 249 Sep 14 11:43 parameters_3.sh
-rwxrwxr-x. 1 student student  74 Sep  9 12:21 simple_variable_in_script
-rwxrwxr-x. 1 student student 114 Sep 10 15:58 while1.sh
[student@localhost ~]$ ./BashApp.sh
There are 150 trainees
[student@localhost ~]$ 
```

Bash. case.

case (in) / esac

The case construct is the shell scripting analog to switch in C/C++. It permits branching to one of a number of code blocks, depending on condition tests. It serves as a kind of shorthand for multiple if/then/else statements and is an appropriate tool for creating menus.

```
case "$variable" in

 "$condition1" )
 command...
 ;;
 "$condition2" )
 command...
 ;;
esac
```

```
#!/bin/bash
# Testing ranges of characters.

echo; echo "Hit a key, then hit return."
read Keypress

case "$Keypress" in
  [[:lower:]]   ) echo "Lowercase letter";;
  [[:upper:]]   ) echo "Uppercase letter";;
  [0-9]         ) echo "Digit";;
  *             ) echo "Punctuation, whitespace, or other";;
esac      #  Allows ranges of characters in [square brackets],
          #+ or POSIX ranges in [[double square brackets.
```

Bash. case

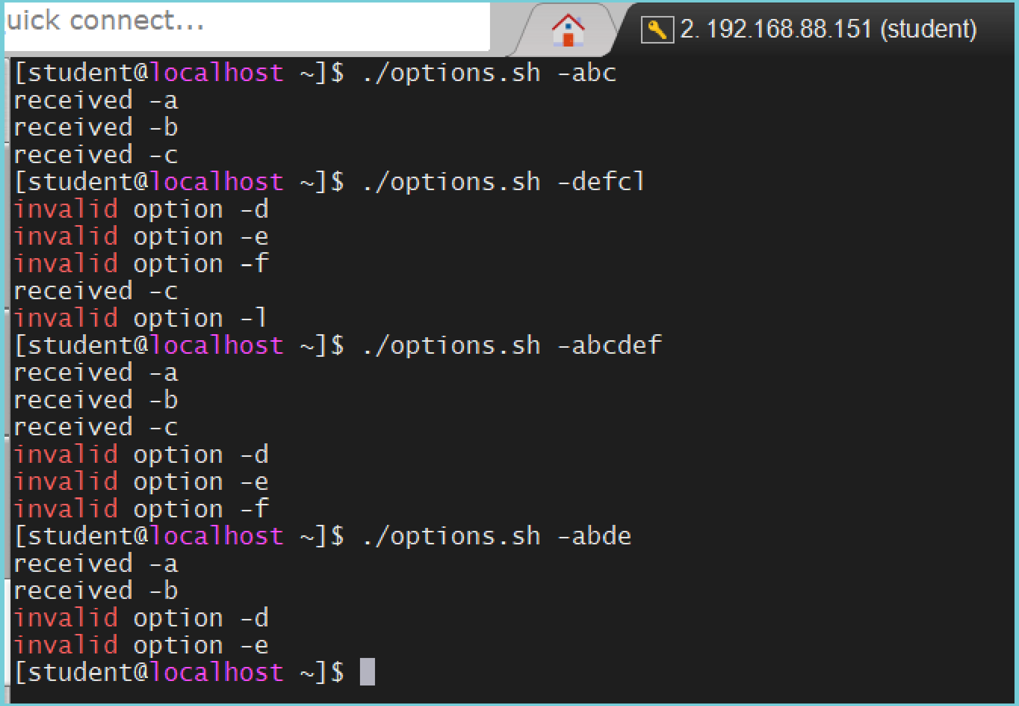
You can sometimes simplify nested ***if*** statements with a ***case*** construct

```
#!/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. 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
```

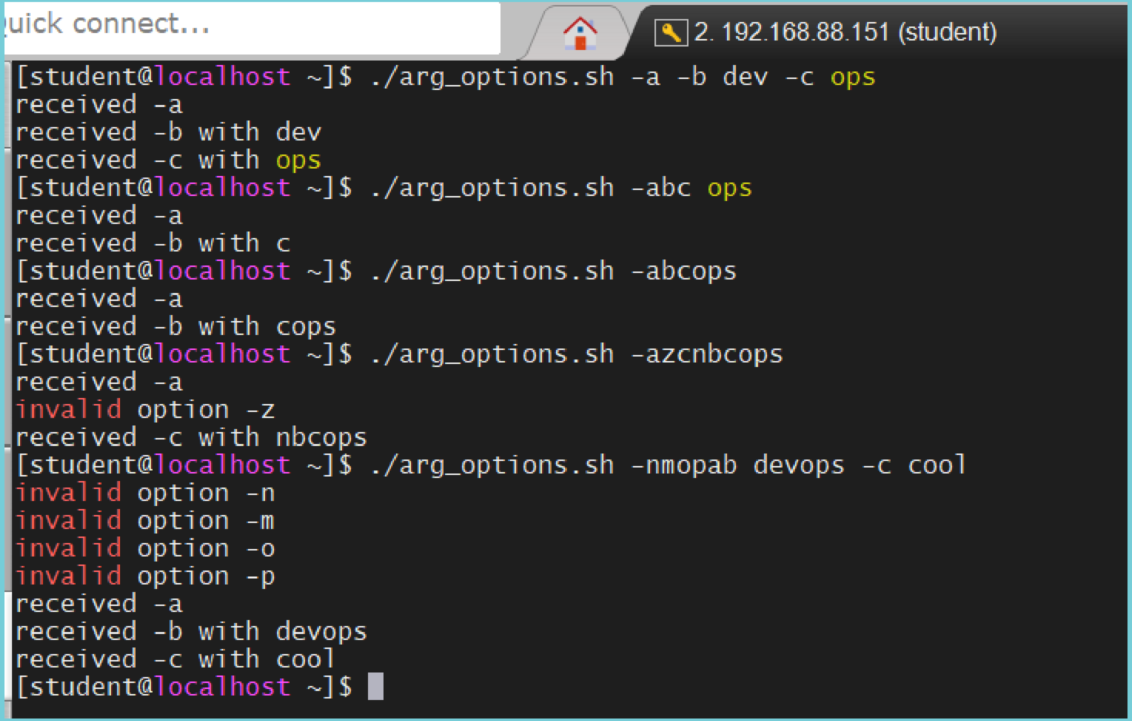


```
[student@localhost ~]$ ./options.sh -abc
received -a
received -b
received -c
[student@localhost ~]$ ./options.sh -defcl
invalid option -d
invalid option -e
invalid option -f
received -c
invalid option -l
[student@localhost ~]$ ./options.sh -abcdef
received -a
received -b
received -c
invalid option -d
invalid option -e
invalid option -f
[student@localhost ~]$ ./options.sh -abde
received -a
received -b
invalid option -d
invalid option -e
[student@localhost ~]$
```

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
```



```
[student@localhost ~]$ ./arg_options.sh -a -b dev -c ops
received -a
received -b with dev
received -c with ops
[student@localhost ~]$ ./arg_options.sh -abc ops
received -a
received -b with c
[student@localhost ~]$ ./arg_options.sh -abcops
received -a
received -b with cops
[student@localhost ~]$ ./arg_options.sh -azcnbcops
received -a
invalid option -z
received -c with nbcops
[student@localhost ~]$ ./arg_options.sh -nmopab devops -c cool
invalid option -n
invalid option -m
invalid option -o
invalid option -p
received -a
received -b with devops
received -c with cool
[student@localhost ~]$
```

Bash. Additional scripting elements.

The output of commands can be used as arguments to another command, to set a variable, and even for generating the argument list in a for loop.

```
textfile_listing=`ls *.txt`
# Variable contains names of all *.txt files in current working directory.
echo $textfile_listing

textfile_listing2=$(ls *.txt)   # The alternative form of command substitution.
echo $textfile_listing2
# Same result.
```

Bash. Additional scripting elements.

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

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
```

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).

The shell can also convert between different bases.

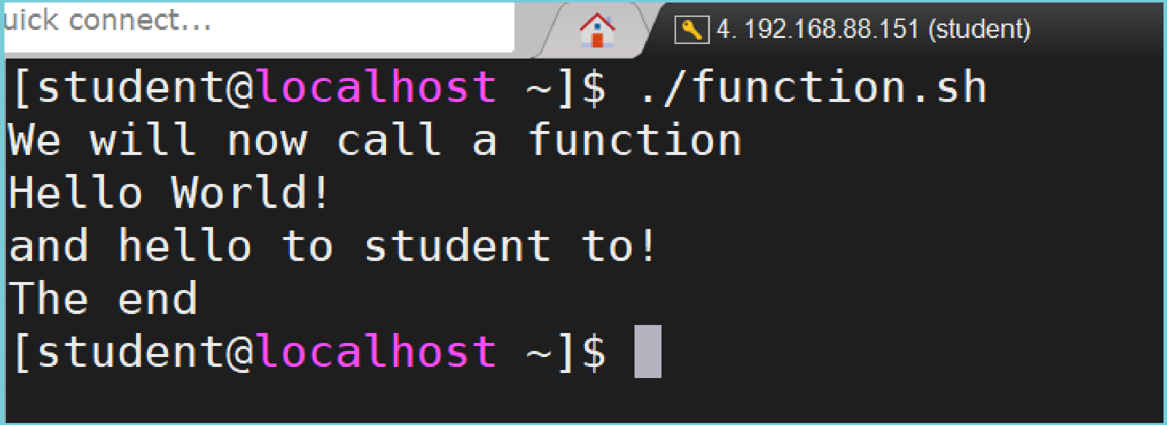
```
[student@localhost ~]$ let x="0xFF" ; echo $x
255
[student@localhost ~]$ let x="0xC0" ; echo $x
192
[student@localhost ~]$ let x="0xA8" ; echo $x
168
[student@localhost ~]$ let x="8#70" ; echo $x
56
[student@localhost ~]$ let x="8#77" ; echo $x
63
[student@localhost ~]$ let x="16#c0" ; echo $x
192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

```
[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.

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

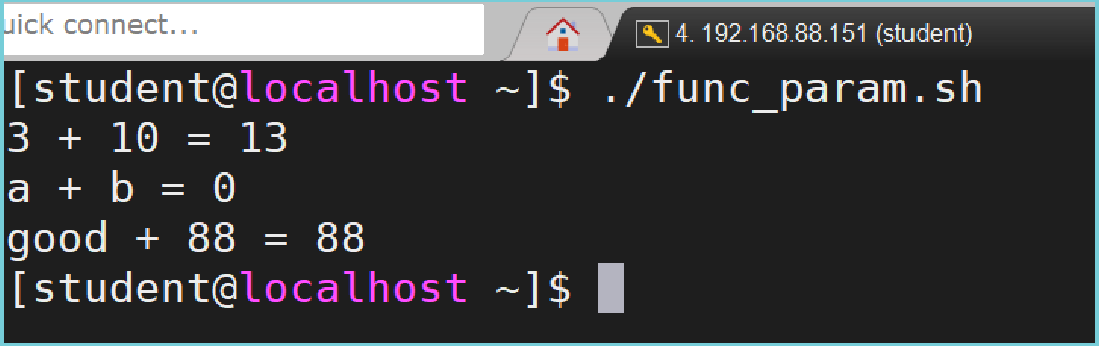


```
[student@localhost ~]$ ./function.sh
We will now call a function
Hello World!
and hello to student to!
The end
[student@localhost ~]$
```

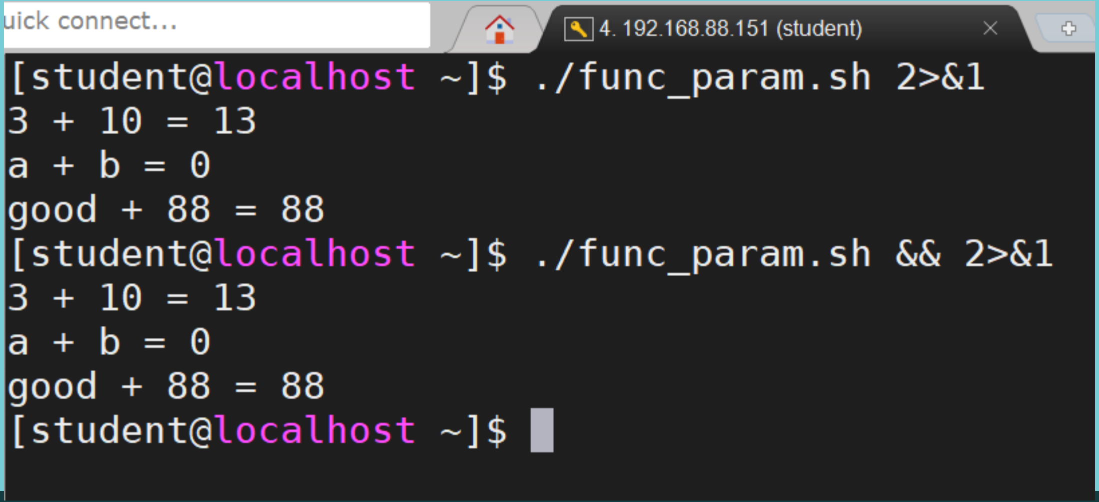
Bash. Additional scripting elements.

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
```



```
[student@localhost ~]$ ./func_param.sh
3 + 10 = 13
a + b = 0
good + 88 = 88
[student@localhost ~]$
```



```
[student@localhost ~]$ ./func_param.sh 2>&1
3 + 10 = 13
a + b = 0
good + 88 = 88
[student@localhost ~]$ ./func_param.sh && 2>&1
3 + 10 = 13
a + b = 0
good + 88 = 88
[student@localhost ~]$
```

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
```

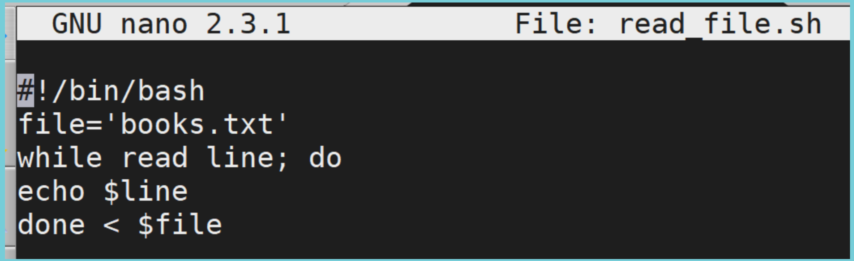
The bash shell will replace variables with their value in double quoted lines, but not in single quoted lines.

```
$ 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.

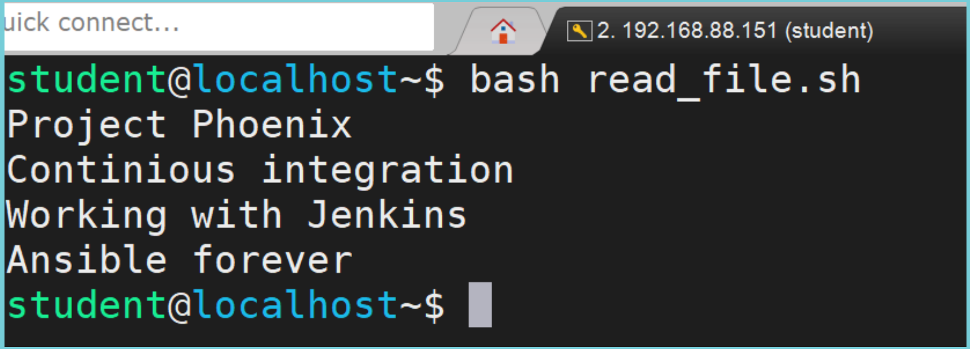
Read a File:

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
```



```
student@localhost~$ bash read_file.sh
Project Phoenix
Continious integration
Working with Jenkins
Ansible forever
student@localhost~$
```

QUESTIONS & ANSWERS

THANK YOU!