Bash Lecture 2 - Advanced

Bibliography and learning materials

★ Bibliography:

https://www.rigacci.org/docs/biblio/online/sysadmin/toc.htm

https://www.tldp.org/LDP/abs/html/

★ Learning Materials:

http://www.ee.surrey.ac.uk/Teaching/Unix/

https://github.com/gtaffoni/Learn-Python/blob/master/Lectures/ShellLecture01.pdf

https://github.com/gtaffoni/Learn-Python/blob/master/Lectures/ShellLecture02.pdf

https://github.com/bertocco/bash_lectures

Arguments of this lesson

★ Bash scripting programming:

- Editors
- Scripts and redirection
- Variables
- Main programming elements (if, for while,…)
- Examples (using files)
- Basic `sed`
- Basic `awk`

Shell scripting abilities

Many shells have scripting abilities:

Put multiple commands in a script and the shell executes them as if they were typed from the keyboard.

Most shells offer additional programming constructs that extend the scripting feature into a programming language.

Editors

Editor

- ★In dictionary.cambridge.org: is a piece of software for editing text on a computer
- ★In www.merriam-webster.com: is a computer program that permits the user to create or modify data (such as text or graphics) especially on a display screen

Editor types

In Linux, text editor are of two kinds:

- ★ graphical user interface (GUI) based
  - gedit
  - bluefish
  - lime ……...
- ★ command line text editors (console or terminal)
  - nano
  - pico
  - vi/vim
  - emacs …….

Nano

Nano is the built-in basic text editor for many popular linux distros.
It doesn't take any learning or getting used to, and all its commands and prompts are displayed at the bottom.

★ Use Nano if:

- You're new to the terminal
- you just need to get into a file for a quick change.

Compared to more advanced editors in the hands of someone who knows what they're doing, some tasks are cumbersome and non-customizable.

★ How to Nano:

from your terminal, enter `nano` and the filename you want to edit.
If the file doesn't already exist, it will once you save it.
Commands are listed across the bottom and are triggered with the Control (CTRL) key. For example, to find something in your file, hold CTRL and press W, tell it what you're searching for, and press Enter.
Press CTRL+X to exit, then follow the prompts at the bottom of the screen.

GNU emacs

Emacs has so many available features like a terminal, calculator, calendar, email client, web browser, and Tetris, it's often spoken of as an operating system itself.
Starting Emacs is relatively simple, but more you learn, the more there is to learn.

★ How to Emacs:

Emacs commands are accessed through keyboard combinations of CTRL or ALT and another keystroke. When you see shortcuts that read C-h or M-x, C stands for the control key and M stands for the Alt key (or Escape, depending on your system).

Enter `emacs` in your terminal, and access the built-in tutorial with C-h t. That means, while holding CTRL, press H, then T.

Or, try key combination C-h r to open the manual within Emacs. You can also use the manual as a playground; just remember to quit without saving by pressing key combination C-x C-c.

Helpful Emacs links

- ★ Emacs wiki
  https://www.emacswiki.org/emacs/SiteMap
- ★ GNU Guided Tour
  https://www.gnu.org/software/emacs/tour/
- ★ Cornell Emacs Quick Reference
  https://www.cs.cornell.edu/courses/cs312/2006fa/software/quick-emacs.html

Main Emacs commands

Principali comandi di `emacs`	
Undo	`CTRL-x u` oppure `CTRL-_`
Salva il file	`CTRL-x CTRL-s`
Salva con nome diverso	`CTRL-x CTRL-w` *nome*
Apre un nuovo file	`CTRL-x CTRL-f` *nome*
Inserisce un file	`CTRL-x i` *nome*
Passa ad un altro buffer	`CTRL-x b`
Chiude un buffer	`CTRL-x k`
Divide la finestra in due	`CTRL-x 2`
Passa da una metà all'altra	`CTRL-x o`
Riunifica la finestra	`CTRL-x 1`
Refresh della finestra	`CTRL-l`
Quit da `emacs`	`CTRL-x CTRL-c`
Cursore a fine riga	`CTRL-e`
Cursore a inizio riga	`CTRL-a`
Cursore giù una pagina	`CTRL-v`
Cursore su una pagina	`ESC v`
Inizio del buffer	`ESC <`
Fine del buffer	`ESC >`
Vai alla linea...	`ESC x goto-line` *numero*

Cerca testo	`CTRL-s` *testo*
Sostituisce testo	`ESC %` *testo1 testo2*
Marca inizio di un blocco	`CTRL-SPACE`
Marca fine blocco e taglia	`CTRL-w`
Marca fine blocco e copia	`ALT-w`
Incolla blocco	`CTRL-y`
Pagina di aiuto	`CTRL-h CTRL-h`
Significato di un tasto	`CTRL-h k` *tasto*
Significato di tutti i tasti	`CTRL-h b`
Interrompe comandi complessi	`CTRL-g`
Apre una shell dentro `emacs`	`ESC x shell`
Aiuto psicologico	`ESC x doctor`
Torri di Hanoi	`ESC x hanoi`

http://www.di.unipi.it/~bozzo/fino/appunti/node2.htmla

vi/vim

Vi, typically comes with your distro-of-choice.
Vim is a vi successor with some improvements. It runs by default on OS X and some Linux distributions when `vi` is run.

VI has two modes of operation (is a “modal” editor):

- Command mode for navigating files: commands which cause action to be taken on the file
- Insert mode for editing text: in which entered text is inserted into the file.

Because Vi is navigated through the use of keyboard commands and shortcuts, it is better experienced than explained.

How to Vi or Vim

Enter `vi` or `vim` in your terminal.
When you enter Vi, you begin in command mode and navigate using keyboard commands and the H, J, K, and L keys to move left, down, up, and right, respectively (but arrows use is possible in the most recent versions).

To enter in editing mode press:
‘a’ to append to the file
‘i’ to insert
pressing the <Esc> (Escape) key turns off the Insert mode.

To exit Vim without saving, press ESC to enter command mode, then press : (colon) to access the command line (a prompt appears at the very bottom) and enter q!.
To save and quit, you could use that prompt and the key combination :wq, or hold down SHIFT and press Z two times (the shortcut SHIFT+ZZ).

The : (colon) operator begins many commands like :help for help, or :w to save.

If you’re stuck at the prompt and don’t remember the operator you want to use, enter : (colon), then press CTRL+D for a list of possibilities.

Helpful VI links

★ Basic vi Commands
https://www.cs.colostate.edu/helpdocs/vi.html

★ Swathmore's Tips and Tricks
https://www.cs.swarthmore.edu/oldhelp/vim/home.html

★ Linux Academy's Vim Reference Guide
https://linuxacademy.com/blog/wp-content/uploads/2016/06/vim-2.png

vi basic commands

Summary of most useful commands

Entering command mode

`[Esc]` Exit editing mode. Keyboard keys now interpreted as commands.

Moving the cursor

`h` (or left arrow key) move the cursor left.
`l` (or right arrow key) move the cursor right.
`j` (or down arrow key) move the cursor down.
`k` (or up arrow key) move the cursor up.
`[Ctrl] f` move the cursor one page **f**orward .
`[Ctrl] b` move the cursor one page **b**ackward.
`^` move cursor to the first non-white character in the current line.
`$` move the cursor to the end of the current line.
`G` **g**o to the last line in the file.
`nG` **g**o to line number *n*.
`[Ctrl] G` display the name of the current file and the cursor position in it.

Entering editing mode

`i` **i**nsert new text before the cursor.
`a` **a**ppend new text after the cursor.
`o` start to edit a new line after the current one.
`O` start to edit a new line before the current one.

Replacing characters, lines and words

`r` **r**eplace the current character (does not enter edit mode).
`s` enter edit mode and **s**ubstitute the current character by several ones.
`cw` enter edit mode and **c**hange the **w**ord after the cursor.
`C` enter edit mode and **c**hange the rest of the line after the cursor.

Copying and pasting

`yy` copy (**y**ank) the current line to the copy/paste buffer.
`p` **p**aste the copy/paste buffer after the current line.
`P` **P**aste the copy/paste buffer before the current line.

Deleting characters, words and lines

All deleted characters, words and lines are copied to the copy/paste buffer.

`x` delete the character at the cursor location.
`dw` **d**elete the current **w**ord.
`D` **d**elete the remainder of the line after the cursor.
`dd` **d**elete the current line.

Repeating commands

`.` repeat the last insertion, replacement or delete command.

Looking for strings

`/string` find the first occurrence of *string after the cursor.*
`?string` find the first occurrence of *string* before the cursor.
`n` find the **n**ext occurrence in the last search.

Replacing strings

Can also be done manually, searching and replacing once, and then using `n` (next occurrence) and `.` (repeat last edit).

`n,ps/str1/str2/g` between line numbers *n* and *p*, **s**ubstitute all (**g**: global) occurrences of *str1* by *str2*.
`1,$s/str1/str2/g` in the whole file ($: last line), **s**ubstitute all occurrences of *str1* by *str2*.

Applying a command several times - Examples

`5j` move the cursor 5 lines down.
`30dd` **d**elete 30 lines.
`4cw` **c**hange 4 **w**ords from the cursor.
`1G` **g**o to the first line in the file.

Misc

`[Ctrl] l` redraw the screen.
`J` **j**oin the current line with the next one
`u` **u**ndo the last action

Exiting and saving

`ZZ` save current file and exit vi.
`:w` **w**rite (save) to the current file.
`:w file` **w**rite (save) to the *file* file.
`:q!` **q**uit vi without saving changes.

Going further

`vi` has much more flexibility and many more commands for power users!
It can make you extremely productive in editing and creating text.

Learn more by taking the quick tutorial: just type `vimtutor`.

The editor war

Technologies available in Information Technology are a lot.

Often, to solve a problem, you can choose between different instruments. The rule to base your choose is:

It does not exist “the best tool” but “the best tool to solve *your specific problem*”.

Sometimes different tools are more or less equivalent.

This is the case of editors emacs and vi:

https://en.wikipedia.org/wiki/Editor_war

Choose your editor

Try an editor and its tutorial,
watch videos on how to use it for your intended purpose,
spend a day or two using it with real files training your fingers.

The best editor for you is the one that makes you feel like you're easily getting things done.

What is a script

A script is, in the simplest case, a list of system commands stored in a file.

Place commands in a script is useful

- to avoid having to retype them time and again
- to be able to modify and customize the script for a particular application
- to use the script as a program/command

The sha-bang #!

Every script starts with the sha-bang (#!) at the head, followed by the full path name of an interpreter.
Examples:
#!/bin/sh
#!/bin/bash
#!/usr/bin/perl

This tells your system that the file is a set of commands to be fed to the command interpreter indicated by the path.

The #! is a special marker that designates a file type, or in this case an executable shell script (type man magic for more details on this fascinating topic).

The command interpreter executes the commands in the script, starting at the top (the line following the sha-bang line), and ignoring comments.

Execute the script

★ The script execution requires the script has “execute” permissions:
chmod +rx scriptname (gives everyone read/execute permission)
chmod u+rx scriptname (gives only the script owner read/execute permission)

★ The script can be executed issuing:

./scriptname

★ The script can be made available as a command:

- moving the script to /usr/local/bin (as root), making it available to all users as a system wide executable. The script could then be invoked by simply typing scriptname [ENTER] from the command-line.
- Including the directory containing the script in the user's $PATH

Exercise: a first script

★ Write a script that upon invocation

1) Says “Hello!”
2) shows the time and date
3) The script then saves this information to a logfile

★ Make the script executable

★ Execute the script

★ Make the script available as a command

Special characters (1)

★ Special characters have a meaning beyond its literal meaning

Comments [#]. Lines beginning with a # (with the exception of #!)

```
# This line is a comment.
```

Comments may also occur following the end of a command.

```
echo "A comment will follow." # Comment here.
```

Comments may also follow whitespace at the beginning of a line.

```
    # Note
```

Command separator [semicolon ;]. Permits putting two or more commands on the same line.

```
echo hello; echo world
```

Escape [backslash \]. This is a mechanism to express litterally a special character.
For example the \ may be used to escape " and ' echoing a string:

```
echo This is a double quote \”  # This is a double quote
```

Special characters (2)

Command substitution [backquotes or backticks `]. The `command` construct makes available the output of command for assignment to a variable.
a=`pwd` (or a=$(pwd)) - (backtick AltGR+' on Linux, ALT+096 on Windows)
acho $a # display the path of your location

Wild card [asterisk *]. The * character serves as a "wild card", it matches every filename in a given directory or every character in a string.

Run job in background [and &]. A command followed by an & will run in the background.

```
bash$ sleep 10 &
[1] 850
[1]+  Done                    sleep 10
```

Within a script, commands and even loops may run in the background.
To bring the script in foreground type `fg` or `CTRL Z fg`
To bring the script in background type `fg` or `CTRL Z bg`

Complete reference:
https://www.tldp.org/LDP/abs/html/special-chars.html

Exercise: special characters

- Write a commented command and execute it
- Write two commands on the same row and execute them
- Make the echo of a string containing one or more escaped characters
- Make the echo of a command (like ls or pwd) output
- Use wildcard to list all files starting with ‘a’ in your directory

Redirection (1)

Each UNIX command (or program) is connected to three communication channels between the command and its environment:

- Standard input (stdin) where the command read its input
- Standard output (stdout) where the command writes its output
- Standard error (stderr) where the command writes its error

When a command is executed via an interactive shell, the streams are typically connected to the text terminal on which the shell is running, but can be changed with redirection or with a pipeline

redirect stdout to a file	redirect stderr and stdout to a file
redirect stderr to a file	redirect stderr and stdout to stdout
redirect stdout to stderr	redirect stderr and stdout to stderr
redirect stderr to stdout	

Standard Input, Standard Output and Standard Error Symbols:

standard input	0<
standard output	1>
standard error	2>

Redirection (2)

Redirection [> &> >& >>].

- Redirect stdout to file (overwrite filename if it already exists):

scriptname > filename

scriptname >> filename # appends the output of 'scriptname' to file 'filename'. If
filename does not already exist, it is created

- Redirect stderr to file (overwrite filename if it already exists):

scriptname 2> filename

- Redirect both the stdout and the stderr of command to filename:

command &> filename redirects both the stdout and the stderr of command to filename

- Redirects stdout of command to stderr:

command >&2

- Redirects stderr of command to stdout:

command 2>&1

Redirection: Examples

- Stdout redirected to file

find . -name pippo > find-output.txt

- Stderr redirected to file

find . -name pippo 2> find-errors.txt

- discards any errors that are generated by the find command

find / -name "*" -print 2> /dev/null

/dev/null is a simple device (implemented in software and not corresponding to any hardware device on the system).

/dev/null looks empty when you read from it.

Writing to /dev/null does nothing: data written to this device simply "disappear."

Often a command's standard output is silenced by redirecting it to /dev/null, and this is perhaps the null device's commonest use in shell scripting:

command > /dev/null

- Redirect both stdout and stderr to file

find . -name pippo &> out_and_err.txt

- Redirect stderr to stdout: find . -name filename 2>&1
- Redirect stdout to stderr: find . -name filename 1>&2

Special characters (3)

Pipe [|]. Passes the output (stdout) of a previous command to the input (stdin) of the next one, or to the shell. This is a method of chaining commands together.

```
echo ls -l | sh
#  Passes the output of "echo ls -l" to the shell,
#+ with the same result as a simple "ls -l".
```

```
cat *.lst | sort | uniq
# Merges and sorts all ".lst" files, then deletes duplicate lines.
```

A pipe sends the stdout of one process to the stdin of another. In a typical case, a command, such as cat or echo, pipes a stream of data to a command that transforms it in input for processing:

```
cat $filename1 $filename2 | grep $search_word
```

Redirection with pipe and tee examples

Examples of redirection of the output of a command to be used as input of another:

- Display the output of a command (in this case ls) by pages:
  ls -la | less
- Count files in a directory:
  ls -l | wc -l
- Count the number of rows containing of the word “canadesi” in the file vialactea.txt
  grep canadesi vialactea.txt | wc -l
- Count the number of words in the rows containing the word “canadesi”

`tee` is useful to redirect output both to stdout and to a file. Example:
find . -name filename.ext 2>&1 | tee -a log.txt
This will take stdout and append it to log file. The stderr will then get converted to stdout which is piped to tee which appends it to the log and sends it to stdout which will either appear on the tty or can be piped to another command.

To go deep: https://stackoverflow.com/questions/2871233/write-stdout-stderr-to-a-logfile-also-write-stderr-to-screen

Exercise: redirection

Create a directory and file tree like this one:

my_examples /ex1.dir
/ex2.txt
/ex3.dir
/*ex3.dir/*file1.txt
/*ex3.dir/*file2.txt
/*ex3.dir/*file3.txt

Remove read permissions to directory /*ex2.dir*
Redirect output on a file. Error is displayed on terminal
Redirect error on a file. Output is displayed on terminal
Verify the content of the files
Stderr redirected to file
Redirect output and errors symultaneously

Use pipe to redirect the output of a command to another command and to a file
Use tee to redirect output both to stdout and to a file

UNIX Variables

★ Variables are how programming and scripting languages represent data. A variable is a label, a name assigned to a location holding data.

★ Standard UNIX variables are split into two categories:

- environment variables:
  if set at login, are valid for the duration of the session
- shell variables:
  apply only to the current instance of the shell and are used to set short-term working conditions;

By convention, environment variables have UPPER CASE and shell variables have lower case names.

★ Environment variables are a way of passing information from the shell to programs when you run them. Programs look "in the environment" for variables and if found, will use the values stored.

★ Variables can be set: by the system, by you, by the shell, by any program that loads another program.

bash variables

Variable in bash are untyped.

★ Bash variables are character strings: can contain a number, a character, a string of characters.

★ Depending on context, bash permits arithmetic operations and comparisons on variables. The determining factor is whether the value of a variable contains only digits or not.

★There is no need to declare a variable, just assigning a value to its reference will create it.

bash variables: assignment (1)

It must distinguish between the name (left value) of a variable and its value (right value).

If variable1 is the name of a variable,
then $variable1 is a reference to its value, i.e. the data item it contains.

$variable1 is actually a simplified form of ${variable1}. In contexts where the $variable syntax causes an error, the longer form ${variable} may work.

Referencing (retrieving) the variable value is called variable substitution.

=> No space permitted on either side of = sign when initializing variables.

Example:

```
a=375      # Initialize variable
hello=$a   # No space permitted on either side of = sign when initializing variables.
#     ^ ^
# What happens if there is a space? Bash will treat the variable name as a program to
# execute, and the = as its first parameter. TRY
#
echo hello       # hello ## Not a variable reference, just the string "hello" ...
echo $hello      # 375 ##  This *is* a variable reference, i.e. shows the value.
echo ${hello}    # 375 ##  Likewise a variable reference, as above.
```

assignment disambiguation with {}

In the previous slide: “In contexts where the $variable syntax causes an error, the longer form ${variable} may work”. This is called variable disambiguation.

Example:
If the variable $type contains a singular noun and we want to transform it on a plural one adding an ‘s’, we can't simply add an ‘s’ character to $type since that would turn it into a different variable, $types.
Although we could utilize code contortions such as
echo "Found 42 "$type"s"

the best way to solve this problem is to use curly braces:
echo "Found 42 ${type}s",
which allows us to tell bash where the name of a variable starts and ends

Exercise: bash variables

Try:

1) STR=’Hello World!’
   echo $STR

2) Try assignment and echo the variable content:

   a=5324

   a=(1 3 4 6 5 “otto”) # array

3) Very simple backup script example:
   OF=/tmp/my-backup-$(date +%Y%m%d).tgz
      tar -czf $OF ./subdir_of_where_i_am

bash variables: assignment examples

“naked variable”, i.e. lacking ‘$’ in front, is when a variable is being assigned, rather than referenced.

```
# Assignment simple
a=879 ;  echo "The value of \"a\" is $a."

# Assignment using 'let' (arithmetic expression)
let a=16+5;  echo "The value of \"a\" is now $a."

# In a 'for' loop (see for details later in this lesson):
echo -n "Values of \"a\" in the loop are: "
for a in 7 8 9 11
do
  echo -n "$a "
done

# In a 'read' statement (also a type of assignment):
echo -n "Enter \"a\" "
read a
echo "The value of \"a\" is now $a."
```

bash variables: assignment examples(2)

```
#!/bin/bash
# With command substitution

a=$(echo Hello\!)   # Assigns result of 'echo' command to 'a' ...
echo $a

a=$(ls -l)          # Assigns result of 'ls -l' command to 'a'
echo $a             # Unquoted, however, it removes tabs and newlines.

echo "$a"           # The quoted variable preserves whitespace.
```

Exercise 3: practice with variables assignment

Try different variable assignments and print the variable content to standard output

- Simple assignment
- Command output assignment

bash variables: quoting

Quoting means just that, bracketing a string in quotes.
This has the effect of protecting special characters in the string from reinterpretation or expansion by the shell or shell script. (A character is "special" if it has an interpretation other than its literal meaning. For example, the asterisk * represents a wild card character in Regular Expressions).

Partial quoting consists in enclosing a referenced value in double quotes (" ... ").
This does not interfere with variable substitution. Sometimes referred also as "weak quoting."

Full quoting consists in using single quotes ('...').
It causes the variable name to be used literally, and no substitution will take place.

Examples (Try):

```
a=352
echo $a    # 352
echo “$a”  # 352
echo ‘$a’  # $a
```

=> Quoting a variable preserves whitespaces.

Exercise 2: variables assignment and quoting

In a bash script:

- Assign a variable
- Print the variable value
- Print a string containing the variable value
- Print a string containing the partial quoted variable
- Print the same string fully quoted
- Assign a variable containing multiple spaces
- Print this new variable
- Print this new variable quoted

- Run the script
- Run the script redirecting the output on a file

Bash Arithmetic Expansion (1)

★Arithmetic expansion provides a powerful tool for performing (integer) arithmetic operations. Translating a string into a numerical expression is relatively straightforward using `expr`, backticks, double parentheses, or let.

★ Backticks examples:

```
z=15
z=$(expr $z + 3)
echo $z
```

Demonstrating some of the uses of 'expr'

```
# Arithmetic Operators
a=$(expr 5 + 3)
echo "5 + 3 = $a"

a=$(expr $a + 1)
# incrementing a variable
echo "a + 1 = $a"

# modulo
a=$(expr 5 % 3)
echo "5 mod 3 = $a"
```

Bash Arithmetic Expansion (2)

★Parentheses examples:

$((EXPRESSION)) is arithmetic expansion.

Not to be confused with + command substitution.

Examples:

```
n=0
echo "n = $n"                    # n = 0
(( n += 1 ))                     # Increment.
echo "n = $n"                    # n = 1

# (( $n += 1 ))                  # is incorrect!
echo (( $n += 1 ))
```

Bash Arithmetic Expansion (3)

★`let` let does exactly what (( )) do.

Examples:

```
z=0
let z=z+3
let "z += 3"  #  Quotes permit the use of spaces in
               #  variable assignment.
               #  The 'let' operator actually performs
              #  arithmetic evaluation,
               #  rather than expansion.
echo $z
```

`set`

`Set` sets shell attributes (and positional parametrs)

Example:

```
$ set foo=baz
$ echo "$1"
foo=baz
```

Note that baz is not assigned to foo, it simply becomes a literal positional parameter.

`set` also prints variables that are not exported.

To see other possible operations: `help set`.

Note: `export` exports to children of the current process, by default they are not exported.

Example:

```
$ foo=bar
$ echo "$foo"
bar
$ bash -c 'echo "$foo"'

$ export foo
$ bash -c 'echo "$foo"'
bar
```

Array in bash

★Initialize an array: arrays in Bash can contain both numbers and strings:

- Initialization with all elements of the same type (numbers)

```
myArray=(1 2 4 8 16 32 64 128)
```

- Initialization with mixed types elements

```
myArray=(1 2 "three" 4 "five")
```

★Make sure to leave no spaces around the equal sign. Otherwise, Bash will treat the variable name as a program to execute, and the = as its first parameter!

Retrieve array elements in bash

★Although Bash variables don't generally require curly brackets, they are required for arrays.

In turn, this allows us to specify the index to access:

echo ${myArray[1]} returns the second element of the array (indexes starts from zero).

★Not including brackets

echo $allThreads[1] leads Bash to treat [1] as a string and output it as such.

Some useful array operations

Syntax	Result
arr=()	Create an empty array
arr=(1 2 3)	Initialize array
${arr[2]}	Retrieve third element
${arr[@]}	Retrieve all elements
${!arr[@]}	Retrieve array indices
${#arr[@]}	Calculate array size
arr[0]=3	Overwrite 1st element
arr+=(4)	Append value(s)
str=$(ls)	Save ls output as a string
arr=($(ls))	Save ls output as an array of files
${arr[@]:s:n}	Retrieve n elements starting at index s

Exercise: retrieve array elements

★Initialize three arrays:

- One with only numbers
- One with only strings
- One with mixed elements

★ Retrieve the first, and the third element of each one

★ Write a script that:

- enter the home directory (hint. HOME environmental variable);
- save all the files in the home directory as bash array;
- print the last element of the array

Conditional execution

Conditional statements:

- ★If … then
- ★ If … then … else
- ★ If … then … elif
- ★ case

Conditional statement “if...then”

The if construction allows you to specify different courses of action to be taken in a shell script, depending on the success or failure of a command.

The most compact syntax of the if command is:

```
if TEST-COMMANDS; then COMMANDS; fi
```

Which is the same, less compact:

```
if TEST-COMMANDS
    then COMMANDS
fi
```

The TEST-COMMAND list is executed, and if its return status is one (True), the COMMANDS are executed. The return status is the exit status of the last command executed, or zero if the condition tested is False.

Example of conditional statement “if...then”

- Testing exit status

The ? variable holds the exit status of the previously executed command (the most recently completed foreground process).

Example

Test to check if a command has been successfully executed:

```
ls -l
if [ $? -eq 0 ]
   then echo 'That was a good job!'
fi
```

- Numeric comparisons

The example below use numerical comparisons:

```
num=$(less work.txt | wc -l)
echo $num
If [[  "$num" -gt "150"  ]]
then echo ; echo "you've worked hard enough for today."
fi
```

Main conditional operators

Relational operators

- -lt (<) lower-than
- -gt (>) greather-then
- -le (<=) lower-equal
- -ge (>=) greather-equal
- -eq (==) equal
- -ne (!=) not equal

Boolean operators

- && and
- || or
- | not

Files operators:

- if [-x "$filename"]; then # if filename is executable
- if [-e "$filename"]; then # if filename exists
- ……

Condition check

The [[ ]] construct is the more versatile Bash version of [ ].
This is the extended test command.

No filename expansion or word splitting takes place between [[and]], but there is parameter expansion and command substitution.

```
file=/etc/passwd
if [[ -e $file ]]
then
  echo "Password file exists."
fi
```

Using the [[...]] test construct, rather than [...] can prevent many logic errors in scripts. For example, the &&, ||, <, and > operators work within a [[ ]] test, despite giving an error within a [ ] construct.

Exercise: True and false result

```
a=3

((($a>10))
echo $?          # print 1 because the condition is false

((($a>2))
echo $?          # print 0 because the condition is true
```

Strings comparison example (try)

```
#!/bin/bash
s1='string'
s2=''
if [ $s1 != $s2 ]
then
        echo "s1 ('$s1') is not equal to s2 ('$s2')"
fi
if [ $s1 = $s1 ]
then
        echo "s1('$s1') is equal to s1('$s1')"
fi
```

Some issue? Try the [[ ]] construct

Check if a variable is empty example

Try:

```
if [[ X == X$variable_to_check ]]
  then
      echo “variable is empty”
 else
      echo “variable value is $variable_to_check”
fi
```

Then try:

```
variable_to_check=”I_am_not_empty”
if [[ X == X$variable_to_check ]]
  then
      echo “variable is empty”
 else
      echo “variable value is $variable_to_check”
fi
```

Nested conditional if...then statement

```
a=3

if [ "$a" -gt 0 ]
then
  if [ "$a" -lt 5 ]
  then
    echo "The value of \"a\" lies somewhere between 0 and 5."
  fi
fi

# Same result as:

if [ "$a" -gt 0 ] && [ "$a" -lt 5 ]
then
  echo "The value of \"a\" lies somewhere between 0 and 5."
fi
```

Conditional statement “if...then...else”

```
if [ condition-true ]
then
   command 1
   command 2
   ...
else  # Adds default code block executing if original condition tests false.
   command 3
   command 4
   ...
fi
```

Note:
When if and then are on same line in a condition test, a semicolon must terminate the if statement. Both if and then are keywords. Keywords (or commands) begin statements, and before a new statement on the same line begins, the old one must terminate.

Exercise: “if...then...else”

Write a simple example of the construct if...then...else

Suggestion:
Basic example of if .. then ... else:

```
#!/bin/bash
if [ "foo" = "foo" ]; then
   echo expression evaluated as true
else
   echo expression evaluated as false
fi
```

Example of condition with variables:

```
#!/bin/bash
t1="foo"
t2="bar"
if [ "$t1" = "$t2" ]; then
    echo expression evaluated as true
else
    echo expression evaluated as false
fi
```

Conditional statement “else if and elif”

elif is a contraction for else if. The effect is to nest an inner if/then construct within an outer one.

```
if [ condition1 ]
then
   command1
   command2
else if [ condition2 ]
then
   command3
   command4
else
   default-command
fi
```

```
if [ condition1 ]
then
   command1
   command2
elif [ condition2 ]
then
   command3
   command4
else
   default-command
fi
```

Exercise: “else if and elif”

Translate the previously seen “Nested if...then” example in an “if...elif” form

Case

The BASH CASE statement takes some value once and test it multiple times.
Use the CASE statement if you need the IF-THEN-ELSE statement with many ELIF elements.
Syntax:

```
case $variable in
    pattern-1)
        commands
        ;;
    pattern-2)
        commands
        ;;
    pattern-3|pattern-4|pattern-5)
        commands
        ;;
    pattern-N)
        commands
        ;;
    *)
        commands
        ;;
esac
```

Exercise: case

```
#!/bin/bash
printf 'Which Linux distribution do you know? '
read DISTR

case $DISTR in
    ubuntu)
        echo "I know it! It is an operating system based on Debian."
        ;;
    centos|RedHat)
        echo "Hey! It is my favorite Server OS!"
        ;;
    windows)
        echo "Very funny..."
        ;;
    *)
        echo "Hmm, seems I've never used it."
        ;;
esac
```

Loops

Loop statements:

- ★ for
- ★ while
- ★ until

for loop

Executes an iteration on a set of words.
It is slightly different from other languages (like C) where the iteration is done respect to a numerical index.

Syntax:
```
for CONDITION; do
    COMMANDS
done
```

Examples:
```
#!/bin/bash
for i in $( ls ); do
    echo item: $i
done
```

C-like for:
```
#!/bin/bash
for i in $(seq 1 10)
do
        echo $i
done
```

for loop examples (try)

Counting:

```
#!/bin/bash
for i in {1..25}
do
   echo $i
done
```

or:

```
#!/bin/bash
for ((i=1 ; i<=25 ; i++))
do
   echo $i
done
```

Counting on "n" steps

```
#!/bin/bash
for i in {0..25..5}
do
   echo $i
done
```

That will count with 5 to 5 steps.

Counting backwards

```
#!/bin/bash
for i in {25..0..-5}
do
   echo $i
done
```

Acting on files

```
#!/bin/bash
for file in ~/*.txt
do
echo $file
done
```

That example will just list all files with "txt" extension. It is the same as ls *.txt

Calculate prime numbers

```
#!/bin/bash
read -p "How many prime
numbers ?: " num
c=0
k=0
n=2

numero=$[$num-1]
while [ $k -ne $num ]; do
    for i in `seq 1 $n`;do
      r=$[$n%$i]
            if [ $r -eq 0 ]; then
                  c=$[$c+1]
            fi
    done

     if [ $c -eq 2 ]; then
           echo "$i"
           k=$[$k+1]
      fi
      n=$[$n+1]
      c=0
done
```

break statement in for loop

break statement is used to break the loop before it actually finish executing.
You are looking for a condition to be met, you can check the status of a variable for that condition.
Once the contidition is met, you can break the loop. Pseudo-code example:

```
for i in [series]
do
        command 1
        command 2
        command 3
        if (condition) # Condition to break the loop
        then
                command 4 # Command if the loop needs to be broken
                break
        fi
        command 5 # Command to run if the "condition" is never true
done
```

With the use of if ... then you can insert a condition, and when it is true, the loop will be broken with the break statement

continue statement in for loop

continue stop the execution of the commands in the loop and jump to the next value in the series. It is similar to continue which completely stop the loop.

Pseudo-code example:

```
for i in [series]
do
    command 1
    command 2
    if (condition) # Condition to jump over command 3
            continue # skip to the next value in "series"
    fi
    command 3
done
```

break statement in iteration

break command is used to exit out of current loop completely before the actual ending of loop.
Break command can be used in scripts with multiple loops. If we want to exit out of current working loop whether inner or outer loop, we simply use break but if we are in inner loop & want to exit out of outer loop, we use break 2.
Example

```
#!/bin/bash
# Breaking outer loop from inner loop
for (( a = 1; a < 5; a++ ))
do
echo “outer loop: $a”
for (( b = 1; b < 100; b++ ))
do
if [ $b –gt 4 ]
then
break 2
fi
echo “Inner loop: $b ”
done
done
```

The script start with a=1 & move to inner loop and when it reaches b=4, it break the outer loop.

Exercise:
In this same script, use break instead of break 2, to break inner loop & see how it affects the output.

continue statement in iteration

continue command is used in script to skip current iteration of loop & continue to next iteration of the loop.

Example

```
#!/bin/bash
# using continue command
for i in 1 2 3 4 5 6 7 8 9
do
if [ $i –eq 5 ]
then
echo “skipping number 5”
continue
fi
echo “I is equal to $i”
done
```

while loop

Executes one or more instructions while a condition is true.
It stops when the control condition is true or when the execution is intentionally stopped by the programmer with an explicit interruption instruction (break or continue)

Syntax:

```
while CONDITION; do
    COMMANDS
done
```

Example:

```
#!/bin/bash
counter=0
while [  $counter -lt 10 ]; do
    echo The counter is $counter
    let counter=counter+1
done
```

Example of break statement in while loop

Interrupt the loop at number … (try)

```
#!/bin/bash

num=1
while [ $num -lt 10 ]
do
if [ $num -eq 5 ]
then
echo “$num equal to 5 so I interrupt the loop”
break
fi
echo $num
let num+=1
done
echo “Loop is complete”
```

until loop

Executes one or more instructions until a condition is false.

Syntax:

```
until CONDITION; do
    COMMANDS
done
```

Example:

```
#!/bin/bash
counter=20
until [  $counter -lt 10 ]; do
    echo counter $counter
    let counter-=1
done
```

until vs. while

Until is similar to while, but it is a slightly difference:
Until is executed while the condition is false,
While is executed while the condition is true.
What means it?

Try the following code and check the output:

```
num=1
while [[ $num -lt 10 ]]
do
if [[ $num -eq 5 ]]
then
break
fi
echo $num
let num=num+1
done
echo “Loop while is complete”
```

```
num1=1
until [[ $num1 -lt 10 ]]
do
if [[ $num1 -eq 5 ]]
then
break
fi
echo $num1
let num1=num1+1
done
echo “Loop until is complete”
```

Loop through array elements

${myArray[@]} return all the elements of an array
replace the numeric index with the @ symbol can be tought as standing for all.

Example: Loop on all elements of the array:

```
myArray=(1, 3, 5, ”try” , ”this” ,1)

for t in ${myArray[@]}; do
  echo array element $t
done
```

Loop through array indices

${!allThreads[@]} returns all the indexes in an array.

Example: Loop on all indexes of the array:

```
myArray=(1, 3, 5, ”try” , ”this”, 1)

for i in ${!myArray[@]}; do
  echo “Array element ${i} is = ${myArray[$i]}”
done
```

Functions

Functions are use to group sets of commands logically related making them reusable without the need to re-write them.

A function does not need to be declared.

Function example:

```
#!bin/bash
function quit {
    exit
}
function hello {
    echo Hello!
}
hello
quit
echo foo
```

Syntax:

```
function func_name {
    command1
    command2
    …..
}
```

How to call the function in a script:

```
func_name
```

Functions parameters/arguments

Parameters does not need to be declared.
It is good practice

- to put a comment before the function definition describing parameters and their meaning
- Read the parameters at the beginning of the function

Function with parameters example:

```
#!/bin/bash
function quit {
    exit
}
# input parameter msg=”a message”
function my_func {
    msg=$1
   echo $msg
}
my_func Hello
my_func World
quit
echo foo
```

Syntax with parameters:

```
function func_name {
        command1
        command2
        …..
}
```

How to call the function with parameters in a script:

```
func_name para1 param2 ...
```

Add help to a script

cat usage.sh

```
#!/bin/bash

display_usage() {
     # echo "This script must be run with super-user privileges."
     echo -e "\nUsage:\n$0 [arguments] \n"
}

# if less than two arguments supplied, display usage
if [[  $# -le 1 ]]
  then
    display_usage
    exit 1
fi
```

Add help to a script

Example

```
#!/bin/bash
if [ -z "$1" ]; then        # check if one parameter exists
    echo usage: $0 directory
    exit
fi
srcd=$1
bakd="/tmp/"
mkdir $bakd
of=home-$(date +%Y%m%d).tgz
tar -czf $bakd$of $srcd
```

Positional parameters

Positional parameters are a series of special variables ($0 through $9) that contain the contents of the command line.
If my_script is a bash shell script, we could read each item on the command line because the positional parameters contain the following:
$0 would contain "some_program"
$1 would contain "parameter1"
$2 would contain "parameter2"
…..

This way, if I call my_script with two parameters:
my_script Hello world
Then inside the script I can read them with:

```
#!/bin/bash
script_name=$0
first_word=$1
second_word=$2
Echo “$script_name says $first_word $second_word
```

The mechanism is the same to read functions parameters.

Read the user's input examples

- Example on how to read the user's input:

```
#!/bin/bash
echo Please, enter your name
read NAME
echo "Hi $NAME!"
```

- Example on how to read multiple user's input:

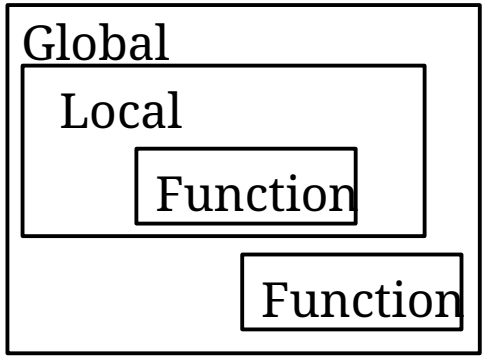
```
#!/bin/bash
echo Please, enter your firstname and lastname
read FN LN
echo "Hi! $LN, $FN !"
echo "How are you?"
```

Scope of variables

In general you can distinguish between

Global
Local
Function

Scope



Bash (like Python) doesn't have block scope in conditionals.

It has local scope within functions, it is also possible to use the ‘local’ modifier which is a keyword to declare the local variables.
Local variables are visible only within the block of code.

Variable scope (visibility) is related mainly to the shell.
Exported variables are visible in all subshells.

Scope of variables

A variable exported is a global variable.

A variable defined in the main body of the script is called a local variable.
- It will be visible throughout the script,
- A variable which is defined inside a function is local to that function.
- It is accessible from the point at which it is defined until the end of the function, and exists for as long as the function is executing.

- Global variables can have unintended consequences because of their wide-ranging effects: we should almost never use them

Exercise: Scope of variables

```
#!/bin/bash
e=2
echo At beginning e = $e
function test1() {
  e=4
  echo "hello. Now in the function1 e = $e"
}
function test2() {
  local e=4
  echo "hello. Now in the function2 e = $e"
}
test1
echo "After calling the function1 e = $e"

e=2
echo In the file before to call func2 reassign e = $e
test2
echo "After calling the function2 e = $e"
```

Justify the result !

Sed

Sed is a non interactive editor.
It is generally used to parse and transform text, using a simple, compact programming language.

It allows to modify a file usinf scripts with instructions for sed editing plus the filename. Example of string substitution:

```
$sed 's/old_text/new_text/g' /tmp/testfile
```

Sed substitute the string 'old_text' with the string 'new_text' reading from file /tmp/testfile. The result is redirected to stdout, but it can be redirected also to a file using '>'

```
$sed 12, 18d /tmp/testfile
```

Sed displays all the rows from 12 to 18. The original file is not modified by this command, but if you redirect stdout on a new file, if is different from the original one (try).

Awk

Awk match a string on the base of a regular expression and execute a required action:

Create a file /tmp/filetext as follow:

```
cat filetext <
test123
test
Tteesstt
EOF
        $awk '/test/ {print}' /tmp/filetext
test123
test
```

The regular expression requires to match the string 'test'

The required action is to 'print'the string containing ‘test’ when found.

```
        $awk '/test/ {i=i+1} END {print i}' /tmp/filetext
3
```

How to check your scripts

Create a script which launch one of the script you wrote by exercise,
Test the output of the command,
Write if the execution is ok or not.