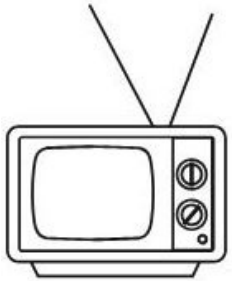
Announcements

Upcoming IE3

- 3 questions on RE
- 1 question on grep / egrep feasible without RE
- 1 globbing
- 1 cut & paste
- 1 join
- 1 redirections / piping
- TOTAL = 8 questions (vs 10) → give you more time for the RE ones

Upcoming case study

- First session on Monday 11/14/2022



Previously On... CIS4930

- M1 – Basics
  - Aliases, filesystem, processes & signals
- M2 – Serious CLI
  - Shell quoting & escaping, Bash initialization files
  - Globbing, variables expansion, command expansion…
- M3 – Linux way
  - Redirections, piping, filters
- M4 – Regular Expressions
  - More filters: grep / egrep, Both Basic & Extended RegExs

M5 – Bash Scripting
- Special variables
- Arithmetic
- Conditionals
- Iteratives
- I/O

Module

M05
Bash Scripting

Menu for this module

T1.1	Anatomy of a Bash Script
T1.2	Bash scripts' variables
T2.1	Arithmetic (expr)
T2.2	Arithmetic (let)
T2.3	Arithmetic Expansion
T3	Conditional Statements
T4	Iterative Statements
T5	I/O

Topic

T1.1 – Anatomy of a Bash script

Before we get started…

What is a script?

- text file with command lines
- Anything that works at the command line interface can be put in a script
- E.g.,
  - put a few ls commands in a script
  - Built-ins are ok too → throw in some echo statements

The 1st thing in the file must be a shebang

- …a what now?

Shebang: #!/bin/bash

- When a text file with a shebang is used as if it is an executable in a Unix-like operating system, the program loader mechanism parses the rest of the file's initial line as an interpreter directive.
- The loader executes the specified interpreter program, passing to it as an argument using the path that was initially used when attempting to run the script, so that the program may use the file as input data.
  - For example, if a script is named with the path path/to/script, and it starts with the following line, #!/bin/sh, then the program loader is instructed to run the program /bin/sh, passing path/to/script as the first argument.
- In Linux, this behavior is the result of both kernel and user-space code.
- The shebang line is usually ignored by the interpreter, because the "#" character is a comment marker in many scripting languages; some language interpreters that do not use the hash mark to begin comments still may ignore the shebang line in recognition of its purpose.

https://en.wikipedia.org/wiki/Shebang_(Unix)

Alternative ways to execute a script

#1 – Tell bash to interpret it:

```
bash myscript.sh
```

#2 – But we can do the following

```
./myscript.sh
```

```
myscript.sh
```

–> nope, except if you have `./` in your `PATH`

For this to work, however, we need the following:

```
chmod a+x myscript.sh
```


<INTERLUDE>
FILE PERMISSIONS

Linux notation for basic File Permissions

`ls -l myscript.sh`

→Let's explain the permissions

- Directory `d` or regular file `-`
- `rwx` → triplet for owner (`u`), group (`g`), others (`o`) or all (`a`)
- `r` → read
- `w` → write or delete or rename
- `x` → execute (for file), or traverse (for directory)

chmod & symbolic notation https://youtu.be/SmSuyfTfK0k

chmod Symbolic notation, setting new permissions

```
chmod u=rx    something.txt
chmod g=rwx   something.txt
chmod o=      somefolder/

chmod a=x     *.sh

chmod u=rwx,g=rx,o=   myFolder/
```

chmod Symbolic notation, adding permissions

```
chmod u+x     something.txt
chmod g+w     something.txt
chmod o+rwx   somefolder/

chmod a+x     *.sh

chmod u+x,g+w,o+rwx   myFolder/
```

chmod Symbolic notation, removing permissions

```
chmod u-x     something.txt
chmod g-w     something.txt
chmod o-rwx   somefolder/

chmod a-x     *.sh

chmod u-x,g-w,o-rwx   myFolder/
```

chmod & octal notation

Owner(u)	Group(g)	Others(o)
rwx	rwx	rwx
---	---	---

Specifying each group as an octal number (2^3==8)

```
chmod 666 myfile.txt
```

→ `rw-rw-rw-`

Permission Triplet	Binary	Octal
---	000	0
--x	001	1
-w-	010	2
-wx	011	3
r--	100	4
r-x	101	5
rw-	110	6
rwx	111	7

chmod & octal notation  https://youtu.be/N6_0l2CIN7c

While we are at it – chown

- Getting to see the owner / group with `ls -l`

```
sudo chown alessio myscript.sh
sudo chown -R alessio myfolder/

sudo chown alessio:faculty myscript.sh
```

- → usually, each user comes with their own group

chown  https://youtu.be/Ckv6j6-UrDU


</INTERLUDE>
FILE PERMISSIONS

Topic

T1.2 – Bash scripts' variables

Bash special variables: arguments

```
./myscript.sh one two three
```

- Demo for $0

Bash variable	Meaning
`$0`	Name of the script
`$1`	First command line argument
…	
`$9`	9th command line argument
`${10}`	10th command line argument
…	

https://youtu.be/QqbkBaJlW9w

Bash special variables:

Bash variable	Meaning
`$?`	Expands to the exit status of the most recently executed foreground pipeline.
`$#`	Number of arguments
`$@`	All arguments (as an array)
`$*`	All arguments (as a single string)
`$-`	Expands to the current option flags as specified upon invocation, by the set builtin command, or those set by the shell itself (such as the -i option).
`$$`	Expands to the process ID of the shell. In a subshell, it expands to the process ID of the invoking shell, not the subshell.
`$BASHPID`	Expands to the process ID of the subshell itself.
`$!`	Expands to the process ID of the job most recently placed into the background, whether executed as an asynchronous command or using the bg builtin

https://www.gnu.org/software/bash/manual/html_node/Special-Parameters.html

Side Note: $* vs. $@

- *$* - Expands to the positional parameters, starting from one. When the expansion occurs within double quotes, it expands to a single word with the value of each parameter separated by the first character of the IFS special variable.*
- *$@ - Expands to the positional parameters, starting from one. When the expansion occurs within double quotes, each parameter expands to a separate word.*

-- Bash Beginner Guide

→ It won't matter until we cover `for` loops!

https://unix.stackexchange.com/questions/129072/whats-the-difference-between-and

Bash scripting Little trick

- At the beginning of the script, assign the arguments to variables with more descriptive names

```
./mycopy file1.txt file2.txt 128
```

- Inside the script:

```
SOURCE_FILE=$1
TARGET_FILE=$2
DATA_AMOUNT_IN_KILOBYTES=$3
```

Bash scripting Little trick #2

- If you call a command in your script, set it up in a variable
- Inside the script:

```
CMD='/bin/ls'
OPT='-ld'
…
$CMD $OPT $@
# applies command on all arguments of this script
```

Topic

T2.1 – Bash Arithmetics: expr

https://youtu.be/tDAl9ofk680

Warning

- We are going to only deal with INTEGER arithmetic
- For floating point arithmetic, use the `bc` command instead.

Displaying results

- By default, `expr` displays results on the screen
- Works as expected for (almost) all basic operators…

```
type expr
expr is hashed (/usr/bin/expr)



expr 5 + 3
8
expr 5 - 3
2
expr 5 / 3
1
expr 5 % 3
2
```

Things to watch out for

- Parentheses must be back slashed
- So does the multiplication operator
- Strings are displayed as-is (“5 * 3” and 5+3)
- Globbing substitution may play havoc with forgetting spaces (5*4)

```
expr \( 5 + 20 \) / 7
3


expr 5 * 3
expr: syntax error
expr "5 * 3"
5 * 3
expr 5+3
5+3
expr 5*4
504    → If we have a 504 file


expr 5 \* 3
15
```

Assigning results

- Use command substitution to run `expr`

```
result=`expr 5 + 3`
echo $result
8


result=$(expr 5 + 3)
echo $result
8
```

Topic

T2.2 – Bash Arithmetics: let

https://youtu.be/k2biw0TCrhY

Differences w/ `expr`

```
type let
let is a shell builtin


let n=3*5
echo $n
15

# no spacing required
# no \* for multiplication


let n=3**5
echo $n
243

let n=5*4
echo $n
20 → even if I have a file 504
```

Assignment operators

```
let n++
echo $n
244

let n--
echo $n
243

let n+=5
echo $n
248

let n=6/5
echo $n
1
```

Remarks

- Double quoting allows spacing between operators
- Do we need the $ in front of variables?

```
let "n = 5 * 5"
echo $n
25




let "$n = 42"
Attempted assignment to non-variable


let "result = n + 1"
echo $n
42
echo $result
43

→ Works also with $n + 1
```

Topic

T2.3 – Bash Arithmetics: $(( )) (( ))

https://youtu.be/AC35dqGC6Es

Basic Examples

- Just another Bash expansion…
- `n` or `$n` are ok in expressions
- Exponentiation available

```
result=$((42 + 9))
echo $result
51

n=42
result=$((n + 9))
echo $result
51
result=$(($n + 9))
echo $result
51


result=$((3 ** 5))
echo $result
243
```

What about assigning?

- We removed the $
- Need to switch notation:
  - $(( )) → (( ))
- Similar to let

```
(( result = 42 ))
echo $result
42

(( result += 42 + 4 ))
echo $result
88


(( result++ ))
echo $result
89
```