#!/bin/bash

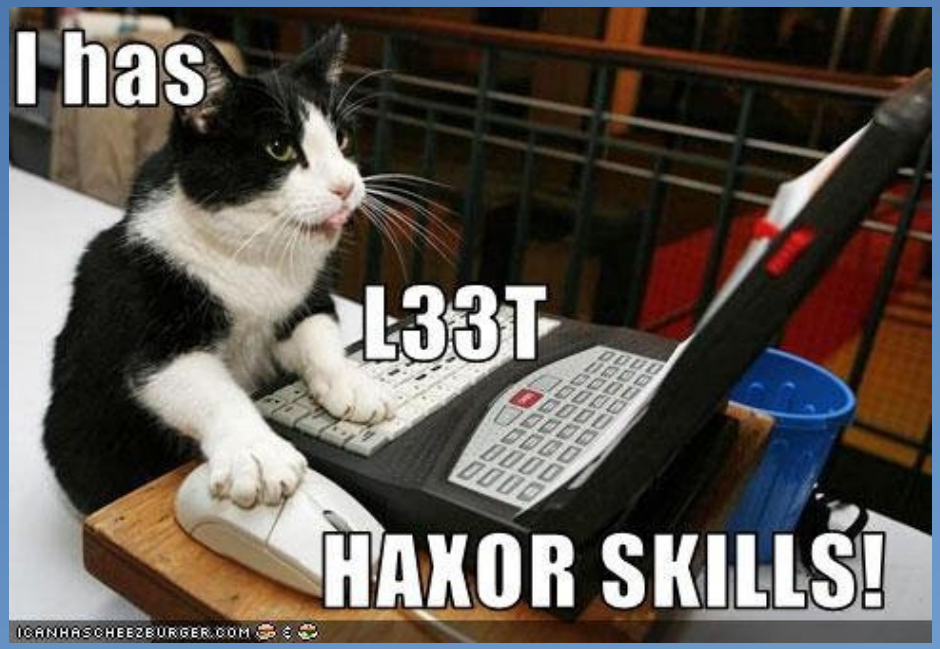
By: Greg savoie
For: Clibre AESS UQÀM

Disclaimer:

This presentation is ugly but it may contains useful informations...

Bash?

- Bourne-again shell
- To make the interactions between the computer and the human possible
- For 1337 h4x0rz only



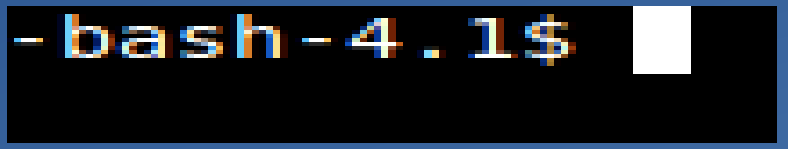
jk

The BB (Bash Basics (not Basic bitch))

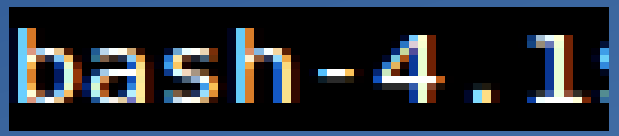
- How to get by with bash
- Customizing bash
- More complex commands
- Basic Bash scripting

How to get by with bash

The prompt

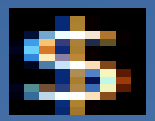


- The fuck is this?

  -> It's the name of the shell with its version (bash 4.1 in this case)

- The fuck is that?

  -> Show if you're the “superuser” or not

- The fuck is a superuser?

  -> we'll see that later.. jeez calm down...

- The prompt displays informations and waits for you to enter commands.

Commands 101

- Commands can take arguments

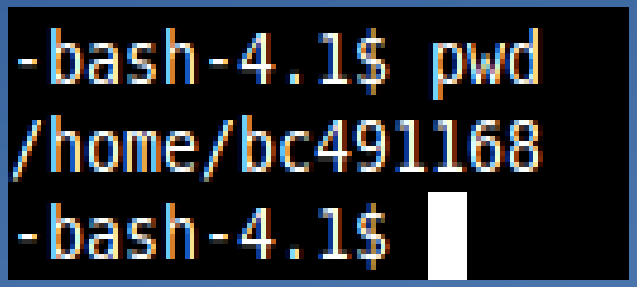
  -> command [arg1] [arg2] [...]
- Commands can take options

  -> command -[options]
- Commands can take both

  -> command -[options] [arg1] [arg2] [...]

Where the hell am I?

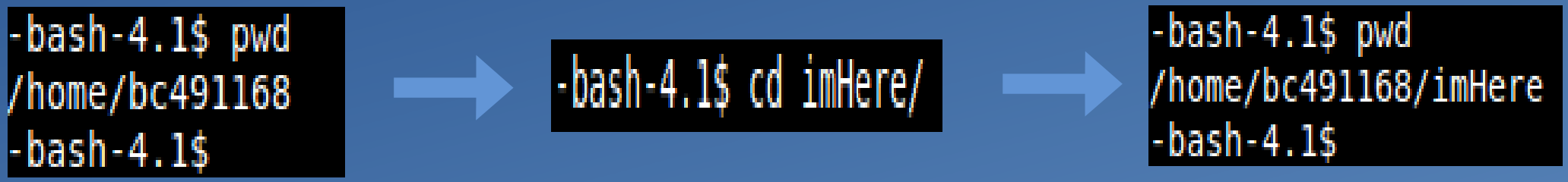
- pwd : Print Working Directory



- The first “/” is the root (it's like the C:\ for windows). As shown by the image, we're currently in a directory named “bc491168” under the home directory who is under the root directory!

How the hell do I get outta here?

- cd : Change Directory



- Absolute vs Relative path.

  Absolute: start from root

  Relative: start from the working directory

Knock Knock. Who's there?

- ls : list directory contents

```
-bash-4.1$ ls
Desktop  inf3105  inf3135  inf3180  inf600a  oracle.txt
-bash-4.1$
```

- Names in blue are sub-directories and names in white are files (may be different, depending on certain configurations...)

Knock Knock. Who's there? (part 2)

- ls with no args

  -> displays the content of the current directory

- ls with args

  -> displays the content of each directory

```
-bash-4.1$ ls /home/bd491021/
Inversions  oracle.txt
-bash-4.1$
```

- ls with -1 option

  -> displays the content of the directory on 1 column

```
-bash-4.1$ ls -1 /home/bd491021/
Inversions
oracle.txt
-bash-4.1$
```

- ls with -l option

  -> displays the content of the directory with long listing format

```
-bash-4.1$ ls -l /home/bd491021/
total 8
drwxr-xr-x 3 bd491021 domain^users 4096 Jan 25 16:39 Inversions
-rw------- 1 bd491021 domain^users  237 Sep 10 17:58 oracle.txt
-bash-4.1$
```

- ls with -a option

  -> displays the content of the directory (including hidden files)

```
-bash-4.1$ ls -a /home/bd491021/
.              .bash_logout   .cache
..             .bash_profile  .config
.bash_history  .bashrc        .dbus
-bash-4.1$
```

- ls with -R option

  -> displays the content of the directory and the content of the subdirectories

```
-bash-4.1$ ls -R /home/bd491021/
/home/bd491021/:
Inversions  oracle.txt

/home/bd491021/Inversions:
makefile  nb_inversions_bm.rb  nb_inve
-bash-4.1$
```

Move that there, copy this, oh w8 delete it, I won't need it!

- mv : move or rename a file

```
-bash-4.1$ pwd
/home/bc491168
-bash-4.1$ ls dummy/
dummy.txt
-bash-4.1$ mv dummy/dummy.txt /home/bc491168/
-bash-4.1$ ls
Desktop  dummy  dummy.txt  inf3105  inf3135  inf3180  inf600a  oracle.txt  resultats.dat
-bash-4.1$ ls dummy/
-bash-4.1$
```

- cp : copy a file (to copy a directory use -r option)

```
-bash-4.1$ ls dummy/
-bash-4.1$ ls
Desktop  dummy  dummy.txt  inf3105  inf3135  inf3180  inf600a  oracle.txt  resultats.dat
-bash-4.1$ cp dummy.txt dummy/
-bash-4.1$ ls dummy/
dummy.txt
-bash-4.1$
```

- rm : remove a file

```
-bash-4.1$ ls
Desktop  dummy  dummy.txt  inf3105  inf3135  inf3180  inf600a  oracle.txt  resultats.dat
-bash-4.1$ rm dummy.txt
-bash-4.1$ ls
Desktop  dummy  inf3105  inf3135  inf3180  inf600a  oracle.txt  resultats.dat
-bash-4.1$
```

Working with directories.. just because

- mkdir : make directory

```
-bash-4.1$ ls
Desktop  dummy  inf3105  inf3135  inf3180  inf600a  oracle.txt  resultats.dat
-bash-4.1$ mkdir newDir
-bash-4.1$ ls
Desktop  dummy  inf3105  inf3135  inf3180  inf600a  newDir  oracle.txt  resultats.dat
-bash-4.1$
```

- rmdir : remove directory (works only with empty directory)

```
-bash-4.1$ ls dummy/
dummy.txt
-bash-4.1$ rmdir dummy/
rmdir: failed to remove `dummy/': Directory not empty
-bash-4.1$
```

- rm -rf for directories that aren't empty

```
-bash-4.1$ ls dummy/
dummy.txt
-bash-4.1$ rm -rf dummy/
-bash-4.1$ ls
Desktop  inf3105  inf3135  inf3180  inf600a  newDir  oracle.txt  resultats.dat
-bash-4.1$
```

Is that all?

- passwd: modifies your password
- history: consults the history of your commands
- jobs: lists of your pending processes
- cat: concatenates files and prints to the standart output
- more / less : displays to the standard output a file page by page
- tail / head : displays the first n lines or the last n lines of a file
- touch : changes the modification date of a file
- chmod : changes the privileges of a file
- echo : displays a line of text
- sort : sorts lines of a text file
- wc : word count
- which : shows the full path of a command
- pushd / popd : comparable to cd but uses the directories stack
- dirs : shows the directories stack
- cmp / diff : compares files for differences
- find : finds files and directories
- And many many more...

In case of doubts: RTFM

- man [command name]

sudo = Sudan Development Organisation

- sudo : super-user do
- allows users to run programs with the security privileges of another user, by default the superuser.
- e.g installing a program (sudo apt-get install for the debian-like OSes)

But don't forget...

Redirections

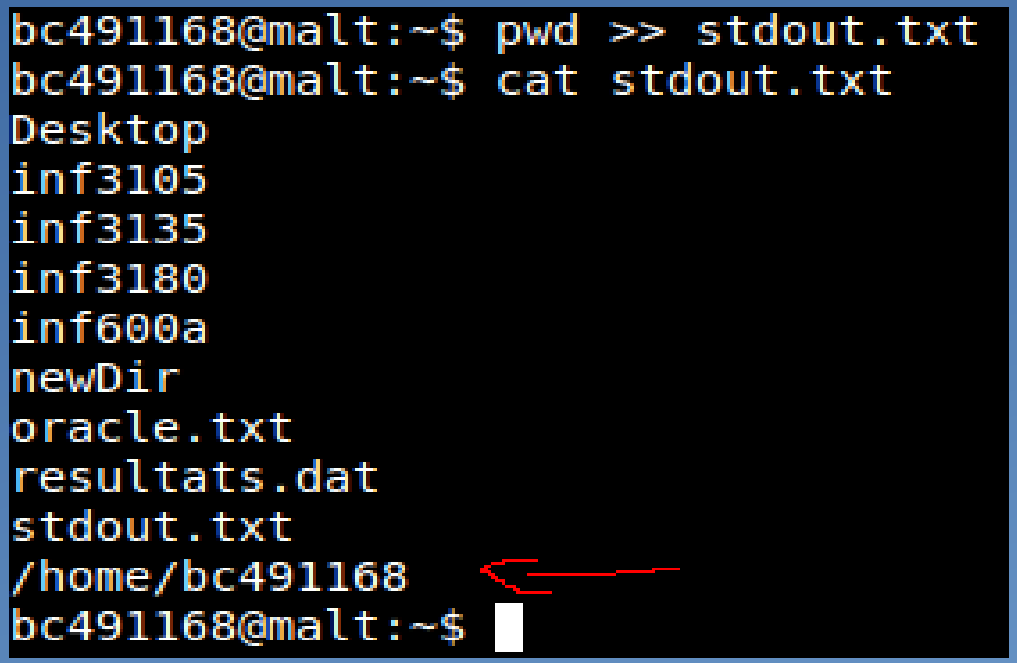
output

- [command] > [file name]: redirects the output of the command in *filename* (creates it if it doesn't exist, overwrites the file if it does exist)

```
bc491168@malt:~$
bc491168@malt:~$ ls > stdout.txt
bc491168@malt:~$ cat stdout.txt
Desktop
inf3105
inf3135
inf3180
inf600a
newDir
oracle.txt
resultats.dat
stdout.txt
bc491168@malt:~$
```

output (part 2)

- [command] >> [filename]: redirects the output of the command in *filename* (creates it if it doesn't exist, appends to the end if it does exist)



errors

- [command] 2> [filename]: redirects the errors of the command in *filename* (creates it if it doesn't exist, overwrites the file if it does exist)

```
bc491168@malt:~$ rm bilbo.txt
rm: cannot remove `bilbo.txt': No such file or directory
bc491168@malt:~$ rm bilbo.txt 2> stderr.txt
bc491168@malt:~$ cat stderr.txt
rm: cannot remove `bilbo.txt': No such file or directory
bc491168@malt:~$
```

errors (part 2)

- [command] 2>> [filename]: redirects the errors of the command in *filename* (creates it if it doesn't exist, appends to the end if it does exist)

```
bc491168@malt:~$ rm baggins.txt 2>> stderr.txt
bc491168@malt:~$ cat stderr.txt
rm: cannot remove `bilbo.txt': No such file or directory
rm: cannot remove `baggins.txt': No such file or directory
bc491168@malt:~$
```

output & errors

- [command] &> [filename]: redirects the output and the errors of the command in *filename* (creates it if it doesn't exist, overwrites it if it does exist)

- [command] &>> [filename]: redirects the output and the errors of the command in *filename* (creates it if it doesn't exist, appends to the end if it does exist)

input

- [command] < [filename]: passes the content of *filename* to the command (the file must exist!) as its input.

```
bc491168@malt:~$ cat input.sh
#!/bin/bash

cat
bc491168@malt:~$ cat input.txt
Hello, world!
bc491168@malt:~$ ./input.sh < input.txt
Hello, world!
bc491168@malt:~$
```

pipe

- [command 1] | [command 2]: takes the output of a command as the input of another.

```
bc491168@malt:~$ ls | grep inf
inf3105
inf3135
inf3180
inf600a
bc491168@malt:~$
```

- Grep is a useful command we'll talk about a bit later

Customizing Bash

2 important files

- ~/.bash_profile: loaded once => when you log in
- ~/.bashrc: loaded each time a terminal is opened
- Typically, there's a line of code in bash_profile that includes bashrc. ([-f ~/.bashrc] && . ~/.bashrc)

Fogell A.K.A McLovin (bash aliases)

- Aliases are useful to create keyboard shortcuts of other commands.
- Aliases can be defined in ~/.bashrc or in any other file (in this case, a line in your bashrc should be added: [-f full/path/filename] && . full/path/filename. Usually, filename is ~/.bash_aliases)
- Syntax: alias [alias_name]='[command]'
- Exemple: alias gcc='gcc -Wall'

Environment variables

- They are used to store data, set configuration options and to customize the shell environment
- There's many ways to declare environment variables: we will declare them in ~/.bashrc (so they will be loaded each time a shell opens)
- Syntax: Export MY_VAR=”MY VAR CONTENT”
- One important environment variable for customization is the PS1 environment variable.

PS1 != PlayStation 1

- This variable is used to customize the prompt.

  Before

```
bc491168@malt:~$ This prompt is so ugly...
```

  After

```
(17:07:15) @malt [~] {0} $  This prompt is so beautiful...
```

Anatomy of the wild PS1 variable

- \e[1;36m(\t) @\H [\W] {\j} \$\e[m
- \e[and \e[m => indicate the beginning and the end of a color
- 1;36 => bold, cyan color
- \t => the current time in 24-hour HH:MM:SS format
- \H => the hostname
- \W => the basename of the current working directory, with $HOME abbreviated with a tilde
- \j => the number of jobs currently managed by the shell
- \$ => if the effective UID is 0 (root), a #, otherwise a $
- Everything uncolored hold no signification and is interpreted as a normal string.

For more PS1 codes, see the PS1 cheat sheet in cheat_sheets/ps1_cheat_sheet.pdf

More complex commands

The commands we'll see are very powerful and are used mostly for text manipulation. Hence, a good knowledge of regular expressions is needed to understand the following slides.

We'll do a quick reminder of regexes. For a more complete reference on regexes, see the bash cheat sheet in cheat_sheets/bash_cheat_sheet.pdf.

Regex

- It's a sequence of characters representing a pattern. The pattern matches a set of strings.

  Ex: ^[ab]\{0,2\}c$ (simple regex)

  matches: c, ac, aac , abc, bac, bc, bbc
- There's 3 types of regexes: simple, extended and file globing (ok, there's probably more than 3, but these are the most important...)

Simple regex

- \ => escape character: delete the meaning of a special character
- . => any character
- * => 0, 1 or many repetitions of the last character / sequence
- ^ => the beginning of the line
- $ => the end of the line
- [...] => every character that is in the class (ex: [0-9], [a-z])
- [^...] => every character that isn't in the class (ex: [^0-9])
- \(...\) => used to capture a sequence. The sequence can then be referenced with \1, \2, etc.. (backreferences)
- \{n\} => n times the last character / sequence
- \{n,\} => at least n times the last character / sequence
- \{n,m\} => n to m times the last character / sequence

Extended regex

- \ => escape character: delete the meaning of a special character
- . => any character
- * => 0, 1 or many repetitions of the last character / sequence
- + => 1 or more repetitions of a the character / sequence
- ? => The last character or sequence can be present or not
- ^ => the beginning of the line
- $ => the end of the line
- [...] => every characters that is in the class (ex: [0-9], [a-z])
- [^...] => every characters that isn't in the class (ex: [^0-9])
- s1|s2 => either s1 or s2, not both
- (...) => changes the priority
- {n} => n times the last character / sequence
- {n,} => at least n times the last character / sequence
- {n,m} => n to m times the last character / sequence

File globbing

- \ => escape character: delete the meaning of a special character.
- ? => any character once
- * => any character 0, 1, or many times
- [...] => any character that is in ... (ex: [0-9])
- [^...] => any character that isn't in ... (ex: [^0-9])
- {s1, s2,...,sn} => s1 or s2 or sn

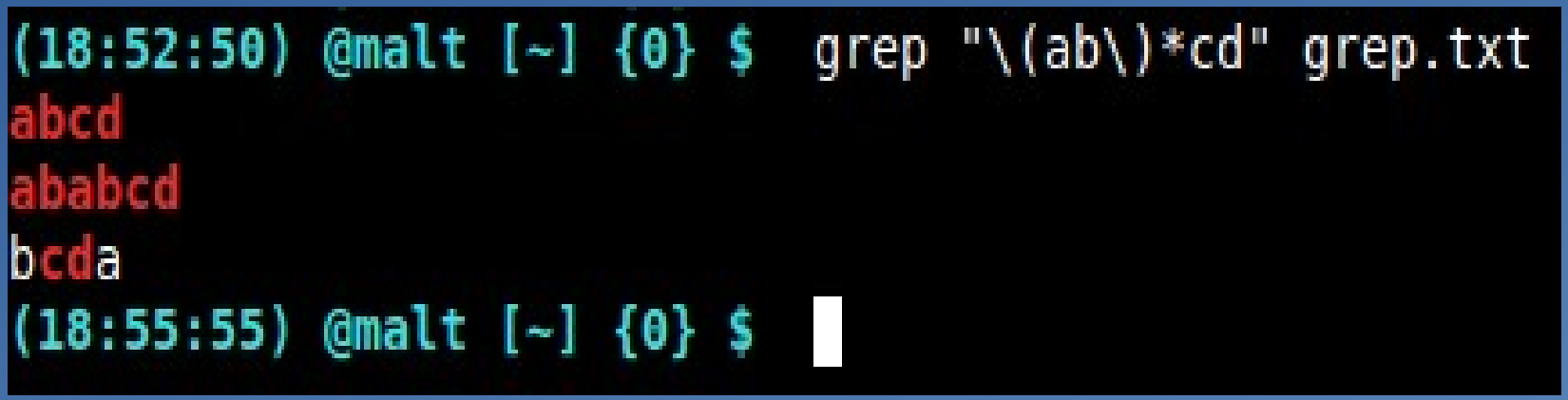
- Simple regex: grep, sed
- Extended regex: awk
- File globbing: ls, find

Grep (global regular expression print)

- It prints the lines of the file that match the regex
- Uses simple regex
- Syntax: grep [options...] 'regex' [files...]
- If no files are specified, grep will read from stdin
- Grep writes on stdout
- If many files are specified, grep will output the name of the file before each line that matches

```
(18:50:33) @malt [~] {0} $  cat grep.txt
abcd
ababcd
bcda
abba
(18:50:35) @malt [~] {0} $
```

```
(18:52:50) @malt [~] {0} $  grep "\(ab\)*cd" grep.txt
```



- What is in red is the part of the line that has been matched by the regex (it may be different in your shell)

Let's go through some grep options...

-v : print the lines that don't match the pattern

```
(19:01:03) @malt [~] {0} $  cat grep.txt
abcd
ababcd
bcda
abba
(19:03:14) @malt [~] {0} $  grep -v "\(ab\)*cd" grep.txt
abba
(19:03:16) @malt [~] {0} $
```

-i: ignore case

```
(19:06:42) @malt [~] {0} $  cat grep.txt
abcd
ababcd
bcda
abba
(19:06:43) @malt [~] {0} $  grep "\(a\)*Cd" grep.txt
(19:06:47) @malt [~] {0} $  grep -i "\(a\)*Cd" grep.txt
abcd
ababcd
bcda
(19:06:57) @malt [~] {0} $
```

-l(lowercase L): outputs only the name of the files that have at least one match

```
(19:14:11) @malt [~] {0} $  cat grep.txt
abcd
ababcd
bcda
abba
(19:14:14) @malt [~] {0} $  cat grep2.txt
hjjkl
jkl
hkl
(19:14:16) @malt [~] {0} $  grep -l "\(ab\)*cd" grep.txt grep2.txt
grep.txt
(19:14:47) @malt [~] {0} $
```

-o: output only the piece of line that has been matched

```
(19:17:00) @malt [~] {0} $  cat grep.txt
abcd
ababcd
bcda
abba
(19:17:04) @malt [~] {0} $  grep -o "\(ab\)*cd" grep.txt
abcd
ababcd
cd
(19:17:15) @malt [~] {0} $
```

For more options:

man grep

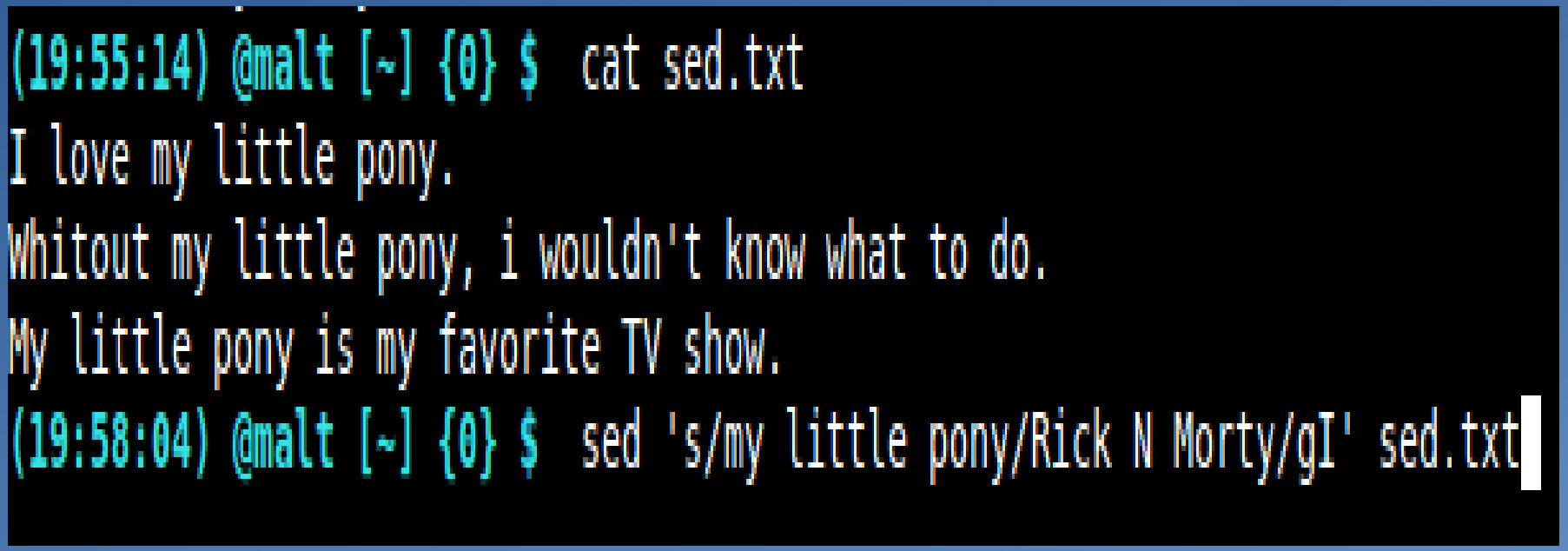
Sed (stream editor)

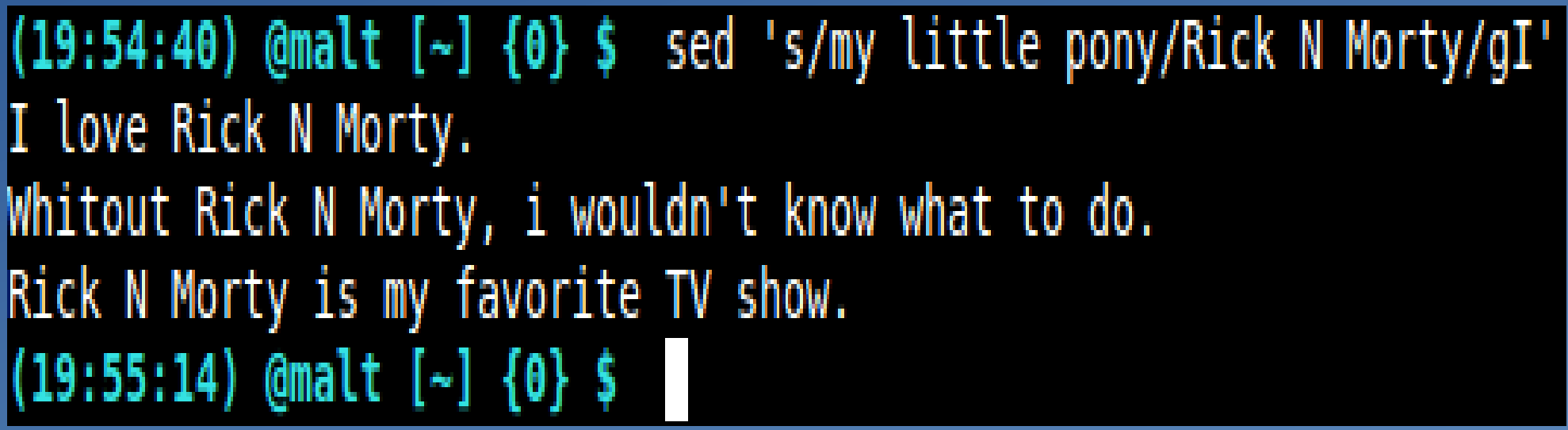
- Parses and transforms text using regexes
- Uses simple regexes
- Syntax: sed [options...] 'sed script' [files...]
- If no files are specified, sed will read from stdin
- Sed writes on stdout
- Sed processes the file line by line

Let's go through some basic sed scripts

Substitute

- s/pattern/newString/gI
- s is for “substitute”
- Pattern is a simple regex
- Replacement is the string by which the matched string will be replaced
- g is for “global”: every occurence matched of the line will be subsituted (it is optional)
- I (uppercase I) is for “case insensitive” (it is optional)


(19:55:14) @malt [~] {0} $ cat sed.txt
I love my little pony.
Whitout my little pony, i wouldn't know what to do.
My little pony is my favorite TV show.
(19:58:04) @malt [~] {0} $ sed 's/my little pony/Rick N Morty/gI' sed.txt


(19:54:40) @malt [~] {0} $ sed 's/my little pony/Rick N Morty/gI'
I love Rick N Morty.
Whitout Rick N Morty, i wouldn't know what to do.
Rick N Morty is my favorite TV show.
(19:55:14) @malt [~] {0} $

print

- /pattern/pI
- Pattern is the regex that needs to be matched
- p is for “print”
- I (uppercase i) is for “case insensitive” (it is optional)
- This script will write to stdout every line that matches the pattern

Print (part 2)

- /pattern/p is not very useful since grep can do the same.
- However, with sed, it is possible to print a range of text delimited by two patterns
- /pattern1/,/pattern2/p
- This script will write to stdout every lines between the first line that matches pattern1 and the last line that matches pattern2 inclusively.

```
(13:28:40) @malt [~] {0} $  cat sed2.txt
this is the first line.
this is another line.
this is the third line.
this is not the first line.
this is a simple line.
this is yet another line.
and finally, this is the last line.
(13:28:43) @malt [~] {0} $
```

```
(13:30:58) @malt [~] {0} $  sed -n '/first/,/another/p' sed2.txt
```

```
(13:30:58) @malt [~] {0} $  sed -n '/first/,/another/p' sed2.txt
this is the first line.
this is another line.
this is not the first line.
this is a simple line.
this is yet another line.
(13:32:38) @malt [~] {0} $
```

- -n : print only the concerned lines (otherwise sed would have printed every line repeating the ones that mathed the pattern...)

delete

- /pattern/d
- Pattern is the regex that need to be matched
- d is for “delete”
- This script will write to stdout every line that doesn't match the pattern
- Like print, you can use a range: /pattern1/,/pattern2/d

```
(13:41:03) @malt [~] {0} $  cat sed2.txt
this is the first line.
this is another line.
this is the third line.
this is not the first line.
this is a simple line.
this is yet another line.
and finally, this is the last line.
(13:41:05) @malt [~] {0} $  sed '/the/d' sed2.txt
this is a simple line.
(13:41:31) @malt [~] {0} $
```

-i : in place changes

- Does the modifications in the specified files and doesn't print to stdout (equivalent of: sed 'script' file > file)
- Syntax: sed -i[.ext] 'sed script' [files...]
- If .ext is specified, sed creates a backup of the original file in file.ext

```
(13:50:26) @malt [~] {0} $  cat sed.txt
I love my little pony.
Whitout my little pony, i wouldn't know what to do.
My little pony is my favorite TV show.
(13:50:28) @malt [~] {0} $  sed -i.bkp 's/my little pony/Rick N Morty/gI' sed.txt
(13:50:53) @malt [~] {0} $  cat sed.txt
I love Rick N Morty.
Whitout Rick N Morty, i wouldn't know what to do.
Rick N Morty is my favorite TV show.
(13:50:56) @malt [~] {0} $  cat sed.txt.bkp
I love my little pony.
Whitout my little pony, i wouldn't know what to do.
My little pony is my favorite TV show.
(13:51:01) @malt [~] {0} $
```

Once again...

- If you want to learn more about sed
- man sed

Awk (Aho Weinberger Kernighan)

- It's a programming language specially designed to parse and process CSV style files.
- Awk uses extended regex
- Syntax: awk [options] 'awk script' file
- The delimiter can be anything and must be specified with the -Fc option where “c” is the delimiter character

Examples of CSV style files...

```
(16:20:06) @malt [~] {0} $  cat awk.txt
10,20,30,
40,50,60,
70,80,90,
11,12,13,
14,15,16,
17,18,19,
(16:20:13) @malt [~] {0} $  cat awk2.txt
Savoie%Gregory%gregsavoie@mail.com%23/09/1993
Lavallee%Eric%ericlavallee@mail.com%23/07/75
Chamberland%Francois%frankchamberland@mail.com%23/05/74
(16:20:17) @malt [~] {0} $
```

- Awk uses extended regexes.
- Awk reads the file line by line.
- It separates each field delimited by the delimiter and associate that field to a variable.
- $1 for the first field, $2 for the second, and so on.
- There are special variables:
- $0 : is the full line
- NR : is the record number
- NF : is the number of field in the record
- FS : is the delimiter (field separator)

Awk script

- 'BEGIN {statements} /pattern/ {script statements} END {statements}'
- BEGIN {} and END {} are optional. The statements will be executed once at the very beginning of the script and once at the very end.
- /pattern/ (optional) is an extended regex. It serves as a filter. Only lines that match the regex will be processed. If you want to print a range of lines you can use /pattern1/,/pattern2/ (every line from the first line that matches pattern1 to the first line that matches pattern2 will be processed)
- {script statements} is the core of the script. It can be really complex but we'll only see a few actions and control structures. (printf and if/else if/else).

- Printf: a C-style formatter and printer (man printf)
- If / else if / else control structures

  if ($1 == “blue”) printf “%s\n”, “sky”

  else if ($1 == “green”) printf “%s\n”, “grass”

  else printf “%s\n”, “undefined”
- Other control structures exist such as while and for.

Printing the first field of each line

```
(19:37:30) @malt [~] {0} $  cat awk2.txt
Savoie%Gregory%gregsavoie@mail.com%23/09/1993
Lavallee%Eric%ericlavallee@mail.com%23/07/75
Chamberland%Francois%frankchamberland@mail.com%23/05/74
(19:37:40) @malt [~] {0} $  awk -F% '{printf "%s\n", $1}' awk2.txt
Savoie
Lavallee
Chamberland
(19:37:43) @malt [~] {0} $
```

Print every person born before 1990

```
(15:48:36) @malt [~] {0} $  cat awk2.txt
Savoie%Gregory%gregsavoie@gmail.com%23/09/1993
Lavallee%Eric%ericlavallee@gmail.com%23/07/1975
Chamberland%Francois%frankchamberland@gmail.com%23/05/1974
(15:48:39) @malt [~] {0} $  awk --re-interval -F% \
> '/[0-9]{2}\/[0-9]{2}\/19[0-8][0-9]/ {printf "%s %s\n", $2, $1}' \
> awk2.txt
Eric Lavallee
Francois Chamberland
(15:51:14) @malt [~] {0} $
```

- **--re-interval option is mandatory to use the {n}, {n,} and {n,m} in the regex...

There's many more fun things you can do with awk.. For the last time, if you want to learn more:

man awk

Find (find files... duh!)

- Syntax: find *directory* -name *pattern*
- directory: is the directory you want to search in. (it will search recursively)
- -name: it's an option that specifies that you want to search for file and directory name that match “pattern”
- pattern: file globing

Find every shell scripts in the inf600a directory

```
(10:05:34) @malt [~] {0} $  ls inf600a/
ExercicesSed  TestsAcceptation  tp1  tp1_base
(10:05:37) @malt [~] {0} $  find inf600a/ -name "*.sh"
inf600a/ExercicesSed/trouver-urls.sh
inf600a/ExercicesSed/supprimer-blancs.sh
inf600a/ExercicesSed/trouver-usagers.sh
inf600a/ExercicesSed/trouver-enseignants.sh
inf600a/ExercicesSed/changer-remote.sh
inf600a/TestsAcceptation/tester_filtre.sh
inf600a/TestsAcceptation/tmp/aruba/nb_mots.sh
inf600a/tp1_base/BiblioBash/biblio.sh
inf600a/tp1/BiblioBash/biblio.sh
(10:05:56) @malt [~] {0} $
```

xargs (x args...)

- It uses the output of a command as arguments for another command
- Syntax: command1 | xargs [options] command2
- If you need the output in your second command, use the -i[string] option. (exemple will follow)
- The string will be available as [string] or as {} if none is specified

Renaming every files ending with the .old extension

```
(17:39:27) @malt [~] {0} $  ls
Desktop  haha.old  hehe.old  hihi.old  hoho.old  inf3105  inf3135  inf3180  inf600a  input.txt  oracle.txt
(17:39:27) @malt [~] {0} $  ls -1 *.old | awk -F. '{print $1}' | xargs -i mv {}.old {}.new
(17:39:43) @malt [~] {0} $  ls
Desktop  haha.new  hehe.new  hihi.new  hoho.new  inf3105  inf3135  inf3180  inf600a  input.txt  oracle.txt
(17:39:45) @malt [~] {0} $
```

Don't forget to read the man...

man xargs

Basic Bash Scripting

What are they used for?

- Automation
- Sys admin related task
- Text manipulation
- Yeah.. mostly that...

Bash scripts usually end with “.sh” extension

The first line of a bash script should be “#!/bin/bash”. This is called the shebang. It tells which program should run the script. (in this case, my bash is in /bin/bash. To know your bash location juste type this command: which bash)

Now I can run a bash script with the following command: ./scriptName.sh

Without the shebang I would have to specify which program would run the script and the command would have been something like: bash scripName.sh

The script must be executable: chmod +x scriptName.sh

Variables

- They work just like environment variable.
- VARNAME=VARVALUE
- After being declared, variables can be used with the “$” sign before its name
- They are global (available anywhere in the program => there's other way to declare variables with other scope. We won't see it in this tutorial / presentation...)
- Echo $VARNAME

```
(16:48:47) @malt [~] {1} $  cat first.sh
VARNAME=VARVALUE
echo $VARNAME
(16:48:58) @malt [~] {1} $  ./first.sh
VARVALUE
(16:49:02) @malt [~] {1} $
```

How to use variables: arithmetics

- Variables can hold numeric data, but if you want to do arithmetics, you will have to use the double parentheses.

```
(17:01:59) @malt [~] {1} $  cat first.sh
VARNAME=5
(( VARNAME=$VARNAME + 5 ))
echo $VARNAME
(17:02:02) @malt [~] {1} $  ./first.sh
10
(17:02:06) @malt [~] {1} $
```

How to use variables: inside a string

- You can use variables in string.
- A variable inside double quotes will be replaced by its value at the moment of execution/output.
- A variable inside single quotes will not.

- Double quotes

```
(17:11:59) @malt [~] {1} $  cat first.sh
animal=lion
fav_animal="My favorite animal is the $animal"
echo $fav_animal
(17:12:01) @malt [~] {1} $  ./first.sh
My favorite animal is the lion
(17:12:07) @malt [~] {1} $
```

- Single quotes

```
(17:14:00) @malt [~] {1} $  cat first.sh
animal=lion
fav_animal='My favorite animal is the $animal'
echo $fav_animal
(17:14:02) @malt [~] {1} $  ./first.sh
My favorite animal is the $animal
(17:14:07) @malt [~] {1} $
```

- Double quotes in single quotes

```
(17:17:13) @malt [~] {1} $  cat first.sh
animal=lion
fav_animal='My favorite animal is the "$animal"'
echo $fav_animal
(17:17:16) @malt [~] {1} $  ./first.sh
My favorite animal is the "$animal"
(17:17:19) @malt [~] {1} $
```

- Single quotes in double quotes

```
(17:18:59) @malt [~] {1} $  cat first.sh
animal=lion
fav_animal="My favorite animal is the '$animal'"
echo $fav_animal
(17:19:01) @malt [~] {1} $  ./first.sh
My favorite animal is the 'lion'
(17:19:06) @malt [~] {1} $
```

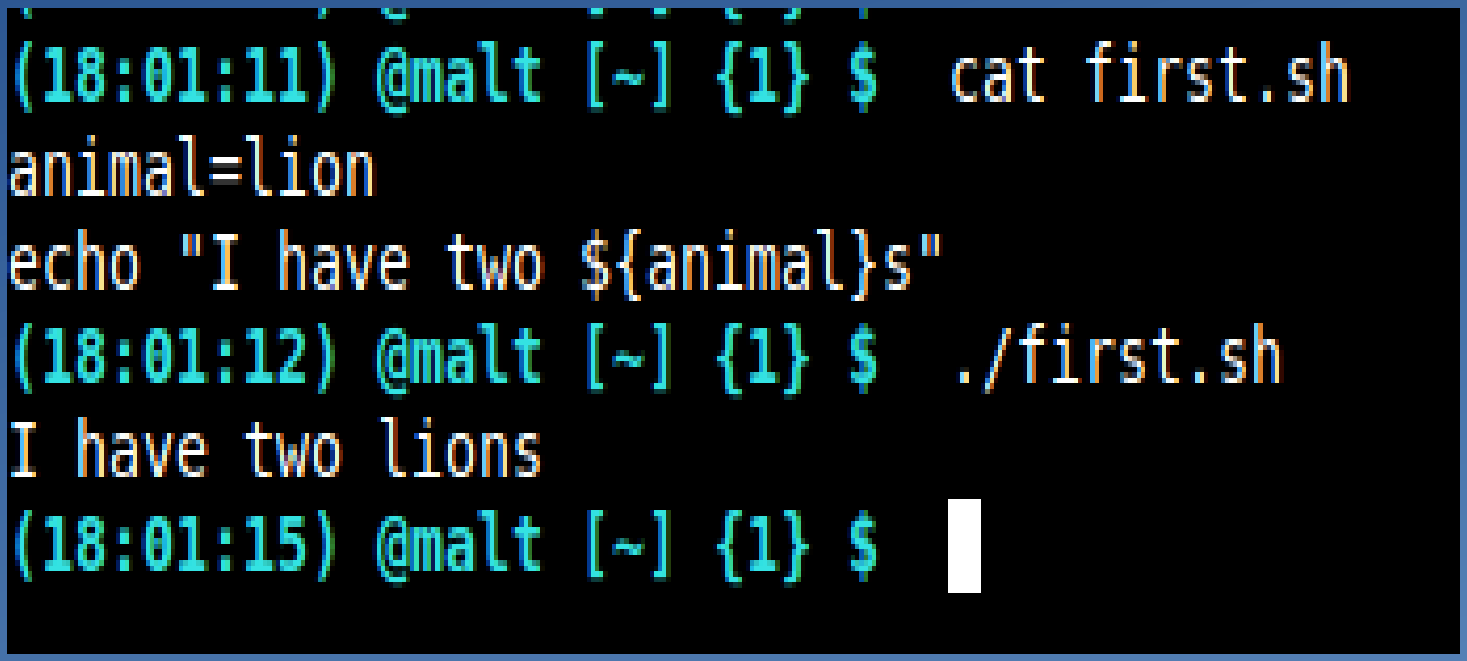
How to use variables: concatenation

```
(17:55:47) @malt [~] {0} $  cat first.sh
animal=lion
echo "I have two $animal"
(17:55:48) @malt [~] {0} $  ./first.sh
I have two lion
(17:55:51) @malt [~] {0} $
```

- What if I want to have the plural form of “lion”??

```
(17:58:16) @malt [~] {1} $  cat first.sh
animal=lion
echo "I have two $animals"
(17:58:18) @malt [~] {1} $  ./first.sh
I have two
(17:58:21) @malt [~] {1} $
```

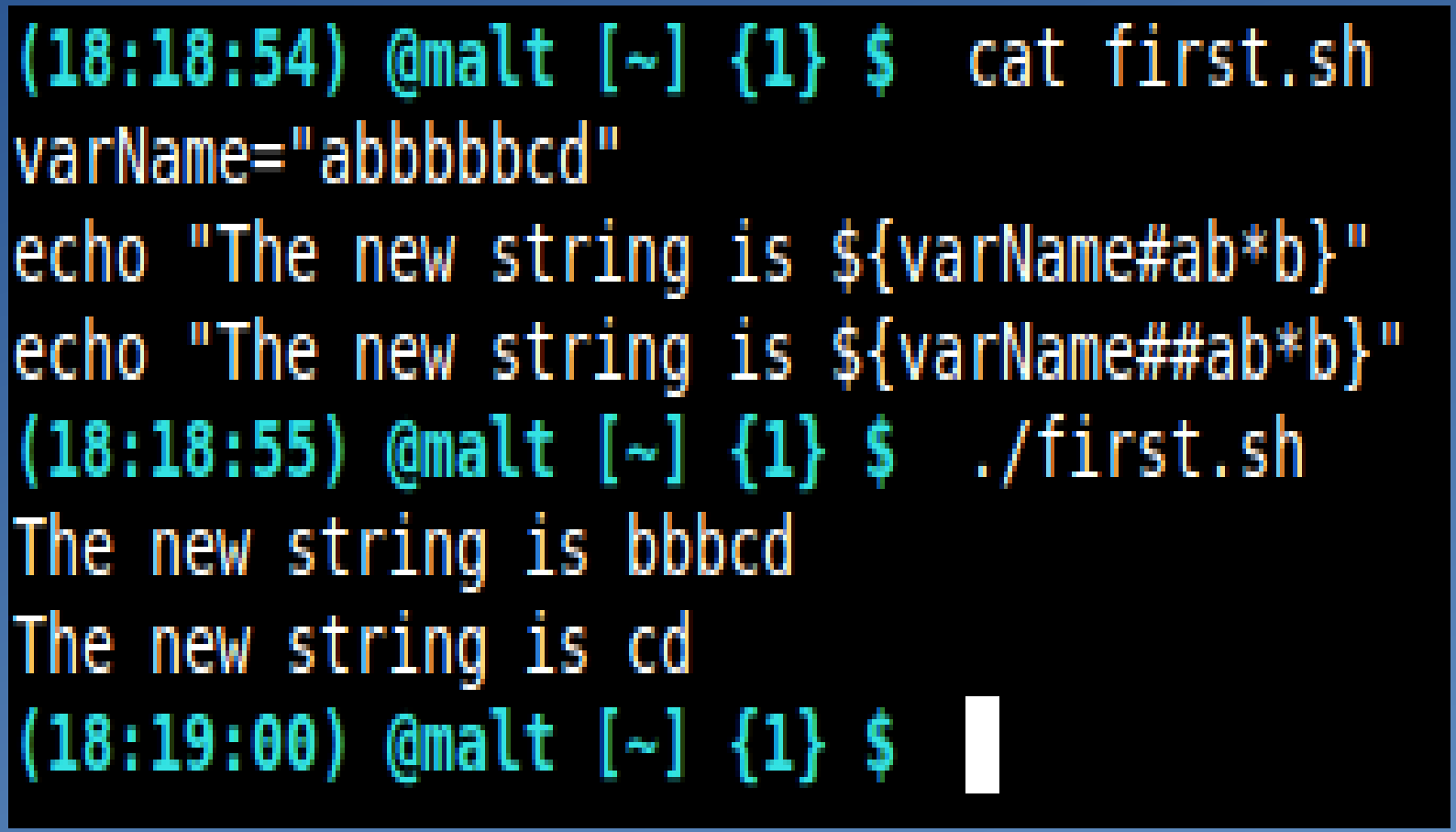
- Here I want to echo “lion” at the plural form
- This doesn't work because the interpreter is looking for a variable called “animals” which doesn't exist.



- This is the right way to do it (variable interpolation).

Variable interpolation and pattern matching

- You can do pattern matching (file globing) with variable interporlation to return only a piece of the variable.
- ${varName#pattern}: will return a substring of “varName” where the smallest string (starting from the beginning) matching “pattern” will be cut.
- ${varName##pattern}: will return a substring of “varName” where the longest string (starting from the beginning) matching “pattern” will be cut.
- % and %% will do the same but the pattern to be matched will start from the end.


(18:18:54) @malt [~] {1} $ cat first.sh
varName="abbbbbcd"
echo "The new string is ${varName#ab*b}"
echo "The new string is ${varName##ab*b}"
(18:18:55) @malt [~] {1} $ ./first.sh
The new string is bbbcd
The new string is cd
(18:19:00) @malt [~] {1} $

Bash special variables

- $? : the exit status of the last command or function executed (usally 0 when everything went right and any other number when something went wrong)
- $# : Number of arguments passed to the script or function
- $0 : Name of the script
- $@ : List of all the arguments passed to the script or the function
- $1, $2, ..., $N : arguments passed to the script or the function.

Bash control structures (if)

```
If <expression> ; then
    [statements...]
elif <expression>; then
    [statements...]
else
    [statements...]
fi
```

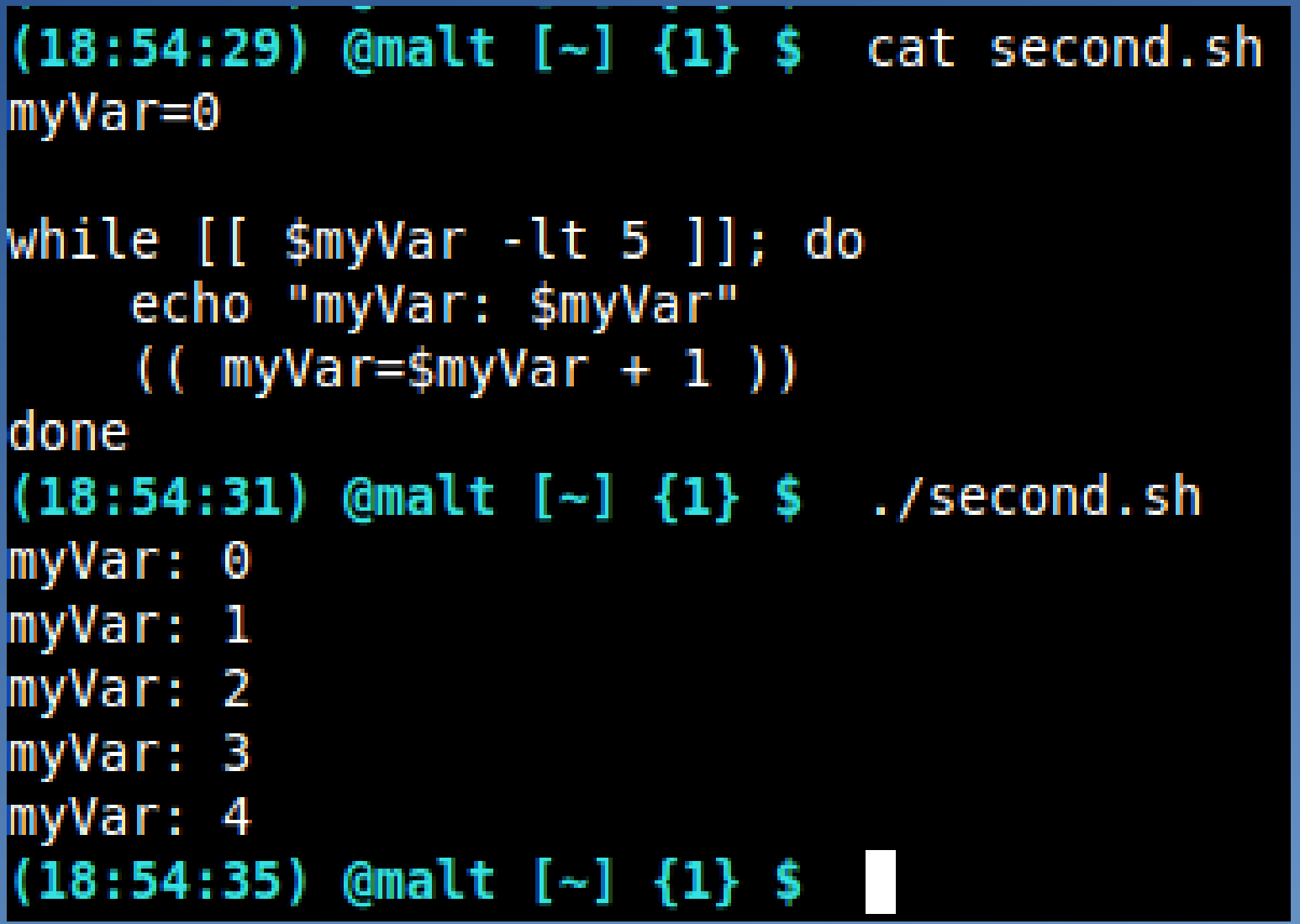
```
(18:47:12) @malt [~] {0} $  cat second.sh
light=yellow

if [[ $light == "red" ]]; then
    echo "stop"
elif [[ $light == "green" ]]; then
    echo "go"
else
    echo "caution, the light is turning red!"
fi
(18:47:14) @malt [~] {0} $  ./second.sh
caution, the light is turning red!
(18:47:20) @malt [~] {0} $
```

Bash control structures (while)

While <expression>; do

    [statements...]

done


(18:54:29) @malt [~] {1} $ cat second.sh
myVar=0
while [[$myVar -lt 5]]; do
echo "myVar: $myVar"
((myVar=$myVar + 1))
done
(18:54:31) @malt [~] {1} $ ./second.sh
myVar: 0
myVar: 1
myVar: 2
myVar: 3
myVar: 4
(18:54:35) @malt [~] {1} $

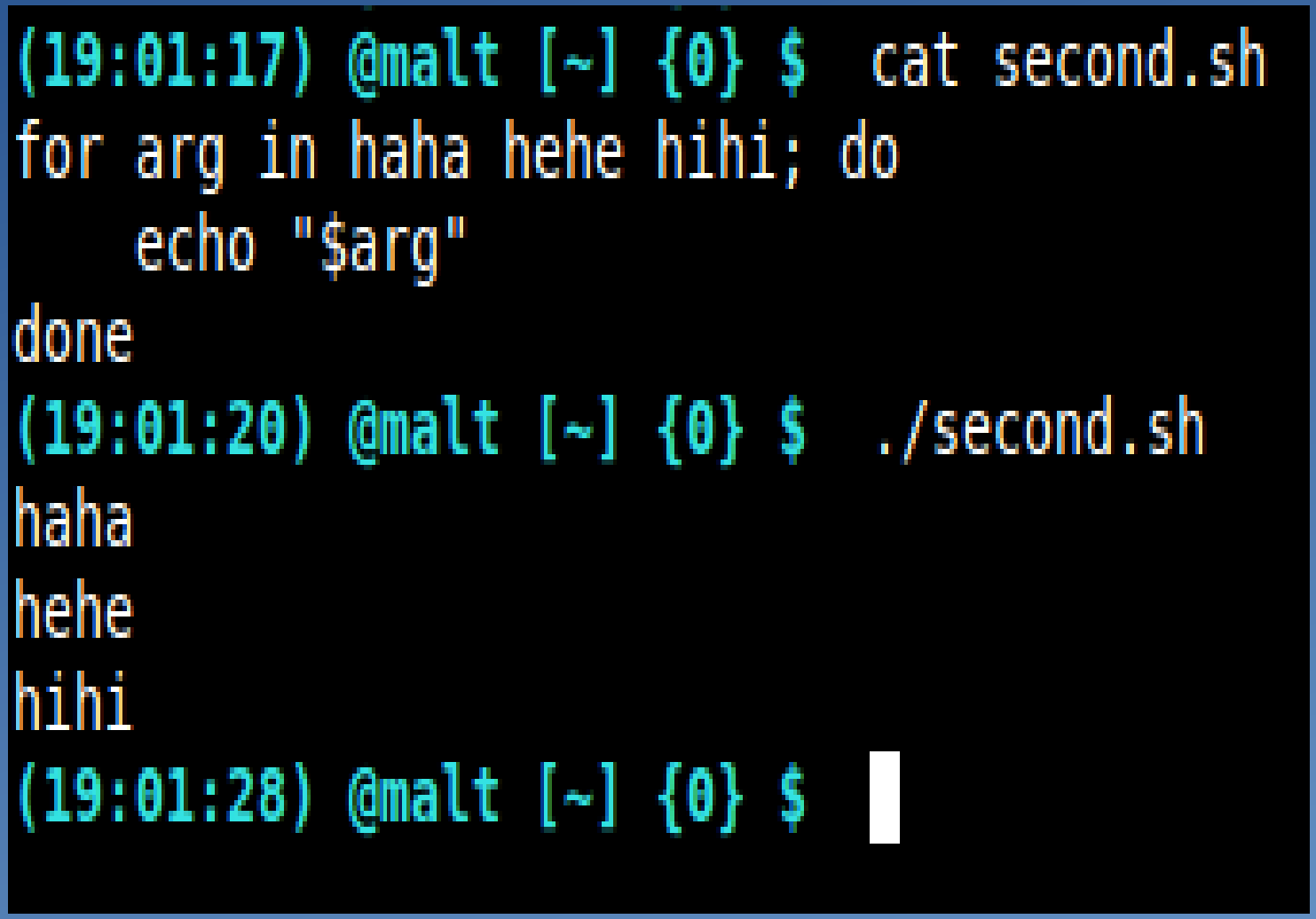
Bash control structures (for)

For var in <expression>; do

    [statements...]

done

- $var will be accessible inside the for loop.


(19:01:17) @malt [~] {0} $ cat second.sh
for arg in haha hehe hihi; do
echo "$arg"
done
(19:01:20) @malt [~] {0} $ ./second.sh
haha
hehe
hihi
(19:01:28) @malt [~] {0} $

Bash control structures (case)

```
Case <expression> in
    pattern1)
        [statements...]
        ;;
    pattern2)
        [statements...]
        ;;
    *)
         [statements...]
         ;;
esac
```

- Patterns are file globing regexes

```
(19:09:17) @malt [~] {1} $  cat second.sh
case $1 in
     dog)
          echo "woof woof"
          ;;
     cat)
          echo "meow moew"
          ;;
     *)
          echo "insert alien noises [here]!"
esac
(19:09:22) @malt [~] {1} $  ./second.sh dog
woof woof
(19:09:26) @malt [~] {1} $  ./second.sh cat
meow moew
(19:09:29) @malt [~] {1} $  ./second.sh goat
insert alien noises [here]!
(19:09:54) @malt [~] {1} $
```

Function

```
Function functionName {
    [statements...]
    [return X]
}
```

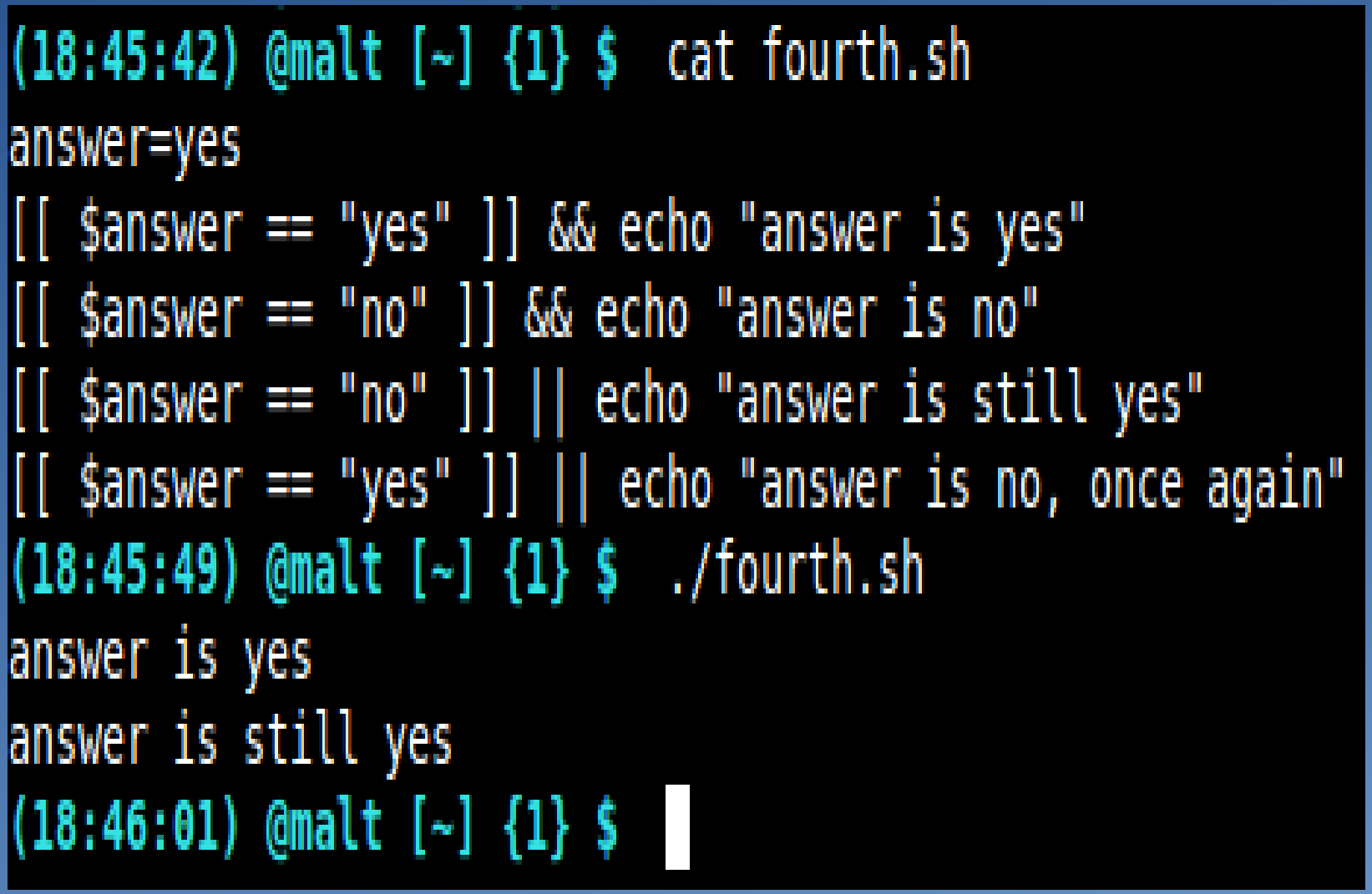
- *To call a function: functionName param1 param2 paramN*
- Parameters to be passed to the function is never precised in its declaration
- Return statement are not mandatory, but if there's no return statement, bash assumes that the exit code is 0.

- In the execution of a function, the value of the special variables $@, $#, and $1 ... $N change.
  (ex: $@ is no more the list of arguments passed to the script, it is now the list of arguments passed to the function...)

```
(18:27:15) @malt [~] {1} $  cat third.sh
function myFunction {
    i=1
    string=""
    for arg in "$@"; do
        if [[ $i -eq $# ]]; then
            string="${string}and $arg are my friends"
        else
            string="${string}${arg}, "
        fi
        (( i=$i + 1 ))
    done
    echo $string
}

myFunction Eddie Bruce Paul
myFunction Eddie Bruce Paul Joe
echo $?
(18:27:18) @malt [~] {1} $  ./third.sh
Eddie, Bruce, and Paul are my friends
Eddie, Bruce, Paul, and Joe are my friends
0
(18:27:28) @malt [~] {1} $
```

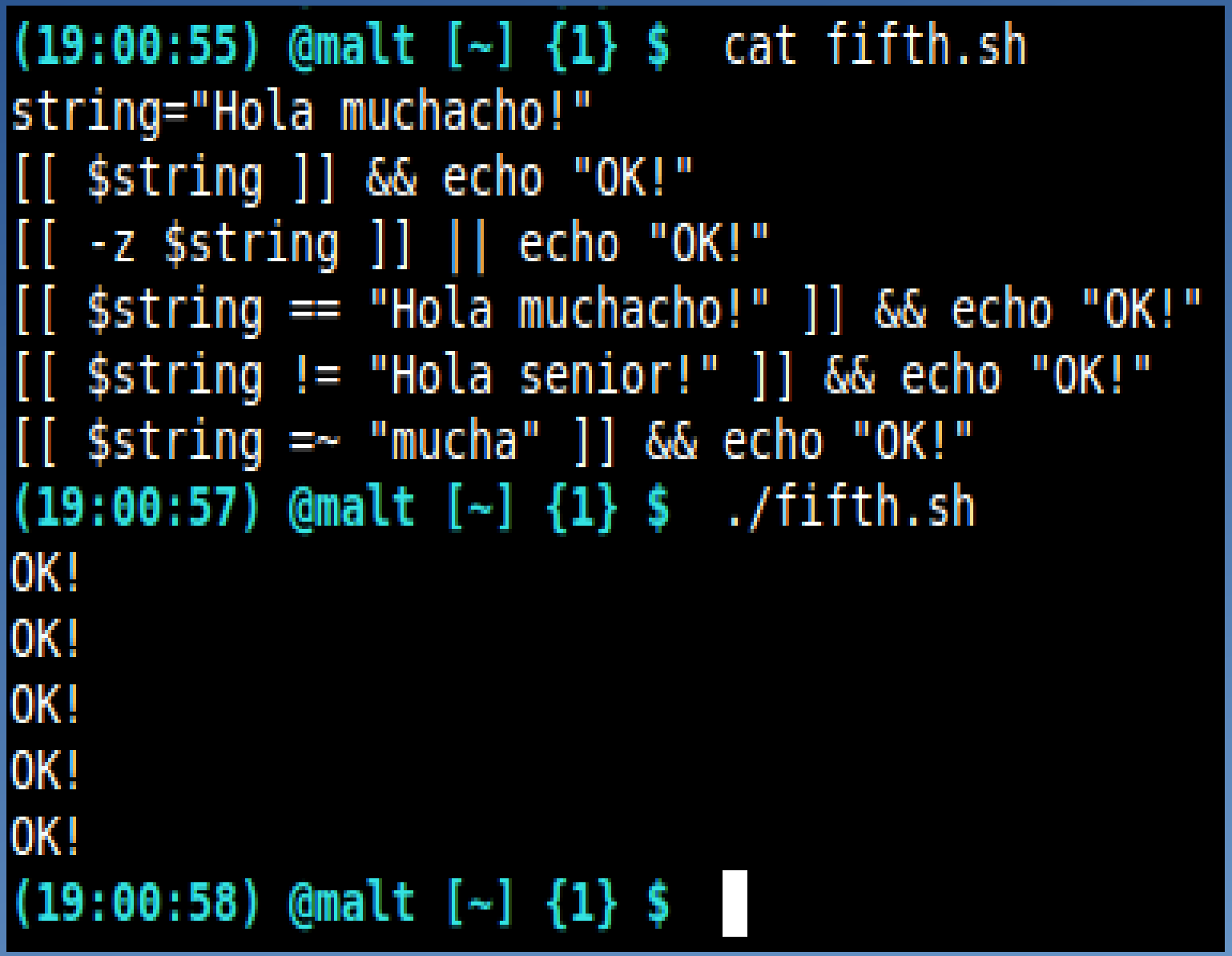
Bash tests

- Use the keyword “[[” to begin the test and the keyword “]]” to end the test
- && : It's the logical “and” operator. It will execute the second statement only if the first is true (0)
- || : It's the logical “or” operator. It will execute the second statement only if the first is false (other than 0)


(18:45:42) @malt [~] {1} $ cat fourth.sh
answer=yes
[[$answer == "yes"]] && echo "answer is yes"
[[$answer == "no"]] && echo "answer is no"
[[$answer == "no"]] || echo "answer is still yes"
[[$answer == "yes"]] || echo "answer is no, once again"
(18:45:49) @malt [~] {1} $ ./fourth.sh
answer is yes
answer is still yes
(18:46:01) @malt [~] {1} $

Tests on strings

- [[stringName]] : return 0 if the string is not empty
- [[-z stringName]] : return 0 if the string is empty
- [[string1 == string2]] : return 0 if the strings are equal
- [[string1 != string2]] : return 0 if the strings are not equal
- [[stringName =~ pattern]] : return 0 if the string matches the pattern (extended regex)


(19:00:55) @malt [~] {1} $ cat fifth.sh
string="Hola muchacho!"
[[$string]] && echo "OK!"
[[-z $string]] || echo "OK!"
[[$string == "Hola muchacho!"]] && echo "OK!"
[[$string != "Hola senior!"]] && echo "OK!"
[[$string =~ "mucha"]] && echo "OK!"
(19:00:57) @malt [~] {1} $ ./fifth.sh
OK!
OK!
OK!
OK!
OK!
(19:00:58) @malt [~] {1} $

Tests on files

- [[-e fileName]] : return 0 if the file exists
- [[-d fileName]] : return 0 if the file is a directory
- [[-f fileName]] : return 0 if the file is a regular file
- [[-x fileName]] : return 0 if the file is executable

```
(19:13:47) @malt [~] {0} $  ls
Desktop   first.sh   haha.new  hihi.new  inf3105  inf3180  input.txt  oracle.txt  sixth.sh  third.sh
fifth.sh  fourth.sh  hehe.new  hoho.new  inf3135  inf600a  lala.sh    second.sh   test.sh
(19:15:16) @malt [~] {0} $
```

```
(19:13:42) @malt [~] {0} $  cat sixth.sh
dir="inf600a"
file="first.sh"
[[ -e $file ]] && echo "$file exists"
[[ -e $dir ]] && echo "$dir exists"
[[ -d $file ]] && echo "$file is a directory"
[[ -d $dir ]] && echo "$dir is a directory"
[[ -f $file ]] && echo "$file is a file"
[[ -f $dir ]] && echo "$dir is a file"
[[ -x $file ]] && echo "$file is executable"
[[ -x $dir ]] && echo "$dir is executable"
(19:13:44) @malt [~] {0} $  ./sixth.sh
first.sh exists
inf600a exists
inf600a is a directory
first.sh is a file
first.sh is executable
inf600a is executable
(19:13:47) @malt [~] {0} $
```

Executing commands

- You can execute any bash command in a bash script.

```
(19:25:12) @malt [~] {0} $  cat seventh.sh
ls -1 *.sh
(19:25:14) @malt [~] {0} $  ./seventh.sh
fifth.sh
first.sh
fourth.sh
second.sh
seventh.sh
sixth.sh
third.sh
(19:25:15) @malt [~] {0} $
```

- Sometimes, you will want to move the result of a command in a variable or in a for loop statement.

```
(19:41:18) @malt [~] {1} $  cat seventh.sh
var=ls -1 *.sh
echo $var

for arg in ls -1 *.sh; do
    echo "$arg"
done
(19:41:28) @malt [~] {1} $  ./seventh.sh
./seventh.sh: line 1: -1: command not found

ls
-1
fifth.sh
first.sh
fourth.sh
second.sh
seventh.sh
sixth.sh
third.sh
(19:41:36) @malt [~] {1} $
```

- This is clearly not the expected result...

- If you want to move the result of a command in a variable or use it in a for loop, you have to surround the command with “$()”.

```
(19:45:40) @malt [~] {1} $  cat seventh.sh
var=$(ls -1 *.sh)
echo $var

for arg in $(ls -1 *.sh); do
    echo "$arg"
done
(19:45:42) @malt [~] {1} $  ./seventh.sh
fifth.sh first.sh fourth.sh second.sh seventh.sh sixth.sh third.sh
fifth.sh
first.sh
fourth.sh
second.sh
seventh.sh
sixth.sh
third.sh
(19:45:46) @malt [~] {1} $
```

Thank you senpai

- Bash version: 4.1.2
- If you find anything wrong in this tutorial you can contact me: greg.savoie23@gmail.com
- You'll find exercises on almost every subjects covered in this presentation in the exercises/ directory
- There's cheat sheets on bash and PS1 variable under the cheatSheets/ directory

References / tutorials

- Bash
- http://www.tldp.org/LDP/abs/html/
- http://tldp.org/LDP/Bash-Beginners-Guide/html/
- http://tldp.org/HOWTO/Bash-Prog-Intro-HOWTO-1.html
- https://www.gnu.org/software/bash/manual/html_node/index.html#Top
- https://en.wikibooks.org/wiki/Bash_Shell_Scripting

- Some websites to test your regexes...
- https://regex101.com/
- http://regexr.com/
- http://www.regextester.com/
- http://regexstorm.net/tester

- Grep
- http://www.panix.com/~elflord/unix/grep.html
- http://www.uccs.edu/~ahitchco/grep/
- http://www.tutorialspoint.com/unix_commands/grep.htm
- http://www.grymoire.com/Unix/Grep.html

- Sed
- http://www.grymoire.com/Unix/Sed.html
- http://www.tutorialspoint.com/sed/index.htm
- http://www.panix.com/~elflord/unix/sed.html
-

- Awk
- http://www.grymoire.com/Unix/Awk.html
- http://www.vectorsite.net/tsawk.html
- http://www.math.utah.edu/docs/info/gawk_toc.html

- PS1 variable
- http://www.cyberciti.biz/tips/howto-linux-unix-bash-shell-setup-prompt.html
- http://www.thegeekstuff.com/2008/09/bash-shell-ps1-10-examples-to-make-your-linux-prompt-like-angelina-jolie/
- http://www.linuxnix.com/linuxunix-shell-ps1-prompt-explained-in-detail/
- http://misc.flogisoft.com/bash/tip_colors_and_formatting