TECNICHE DI RAPPRESENTAZIONE E MODELLIZZAZIONE DEI DATI

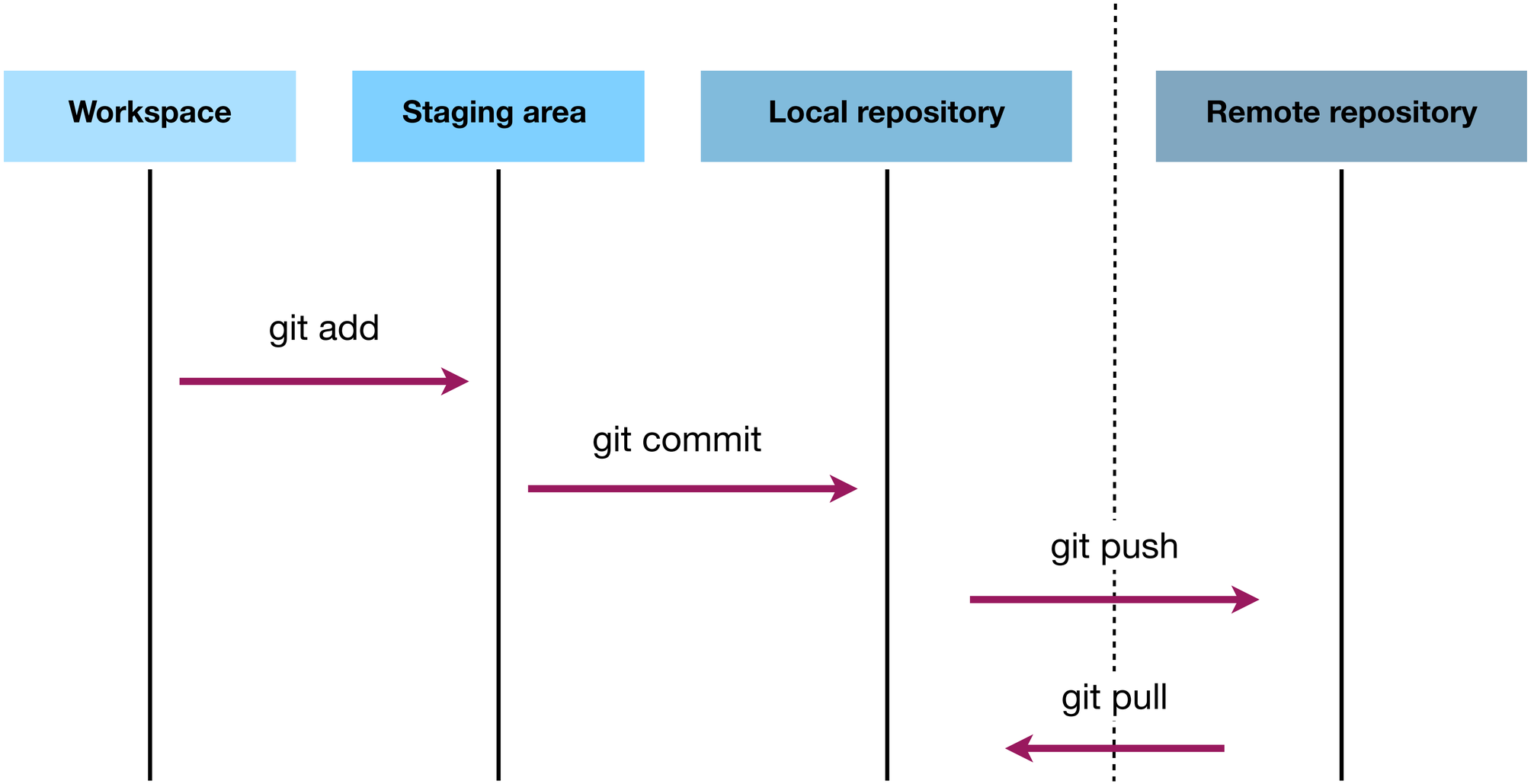
— Part 1 —

(2 CFU out of 6 total CFU)

Link moodle: https://moodle2.units.it/course/view.php?id=11703

Teams code: 0ftoqj8

Git: from local to remote repository



Git: hands-on

Exercise 1

Create a new directory. Create a file (e.g., my_script_1.py) in the directory and modify it (just write ‘import numpy’ in it).
Verify that the directory is NOT a git repository.
Initialize a git repository in the directory. Verify that it is a git repository now.

Exercise 2

Verify the status of the staging area. Create the first commit.
Verify the commit history of the repository.

Exercise 3

Modify my_script_1.py (to import scipy) and create a new commit.
Check the commit history.

Useful commands:

git init
git status
git add
git commit
git log
git branch
git switch
git merge
git remote add (origin)
git push (-u origin)
git pull

Git: hands-on

Exercise 4

Create a new branch called ‘feature’. In the new branch, create a new file called my_script_2.py. Modify my_script_1.py to import matplotlib instead of scipy. Commit the changes.

Exercise 5

Verify that on branch master there is not my_script_2.py. Modify my_script_1.py on master, so that the import is now for healpy. Merge branch feature into master. Resolve the ensuing conflict by importing both healpy and matplotlib.

Exercise 6

Create an empty remote repository and set it to track the master branch of the local one (hint: follow instructions on GitHub website, the splash page when you create a new repository). Create the file my_script_3.py in your local repository and push the changes to the remote one. Then create my_script_4.py in the remote repository and pull the changes in your local one.

Useful commands:

git init

git status

git add

git commit

git log

git branch

git switch

git merge

git remote add (origin)

git push (-u origin)

git pull

Bash: intro

Bash is a scripting/command language very close to the Unix operative system

Useful resources: https://www.gnu.org/software/bash/manual/

http://mywiki.wooledge.org/BashGuide

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

Bash: intro

Many shells have scripting abilities: put multiple commands in a script and the shell executes them as if they were typed from the keyboard.

A command or a script that you can execute consists of one or more processes. The processes can be categorized into different broad groups. The main two are:

- interactive processes, and
- batch processed

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

Bash: intro

Bash interprets a line of code as follows:

Bash divides each line into words separated by a tab/space character

The first word of the line is the name of the **command** to be execute

All the remaining words are arguments of that command (options, filenames, etc.).

Different types of commands:

aliases

functions

builtins

keywords

executables

Bash essentials

An **alias** is a way of shortening a command

example:

```
(base) MacBook-Pro-2:TRM_Dati milenavalentini$ ls
Best_file.txt   Useful          Worst_file.txt  file_1_save.txt file_3.txt      script_1.py
Untitled.ipynb  Useful_extra    file_1.txt      file_2.dat      file_4.txt      script_2.bash
(base) MacBook-Pro-2:TRM_Dati milenavalentini$
(base) MacBook-Pro-2:TRM_Dati milenavalentini$ alias ls='ls -la'
(base) MacBook-Pro-2:TRM_Dati milenavalentini$
(base) MacBook-Pro-2:TRM_Dati milenavalentini$ ls
total 48
drwxr-xr-x  17 milenavalentini  staff   544 Oct  2 19:36 .
drwxr-xr-x  10 milenavalentini  staff   320 Sep 25 15:19 ..
-rw-r--r--   1 milenavalentini  staff     0 Oct  2 19:36 .hidden_file_1.txt
-rw-r--r--   1 milenavalentini  staff     0 Oct  2 19:36 .hidden_file_2.txt
drwxr-xr-x   3 milenavalentini  staff    96 Sep 22 15:16 .ipynb_checkpoints
-rw-r--r--   3 milenavalentini  staff   121 Sep 18 16:54 Best_file.txt
-rw-r--r--   1 milenavalentini  staff  1289 Sep 22 15:28 Untitled.ipynb
drwxrw-rw-   6 milenavalentini  staff   192 Sep 17 18:20 Useful
drwxr-xr-x   4 milenavalentini  staff   128 Sep 19 00:01 Useful_extra
lrwxr-xr-x   1 milenavalentini  staff    10 Sep 18 18:47 Worst_file.txt -> file_3.txt
-rw-r--r--   3 milenavalentini  staff   121 Sep 18 16:54 file_1.txt
-rw-r--r--   1 milenavalentini  staff   121 Sep 18 16:59 file_1_save.txt
-rw-r--r--   1 milenavalentini  staff     0 Sep 18 16:52 file_2.dat
-rw-r--r--   1 milenavalentini  staff    26 Sep 18 16:56 file_3.txt
-rw-r--r--   2 milenavalentini  staff   173 Sep 18 17:06 file_4.txt
-rw-r--r--   1 milenavalentini  staff     0 Sep 17 12:58 script_1.py
-rw-r--r--   1 milenavalentini  staff     0 Sep 17 12:58 script_2.bash
(base) MacBook-Pro-2:TRM_Dati milenavalentini$
```

Bash essentials

An **alias** is a way of shortening a command

example:

```
alias ls='ls -la'
```

- Word mapped into a string
- What is useful for: to change the default behaviour of a command
- What is not aimed at: complex tasks (functions should be preferred)
- Not used in scripts, only in the interactive mode

Bash essentials

A **function** is a set of shell commands

example:

```
JUST_A_SECOND=1

fun ()
{ # A somewhat more complex function.
        local i=0
        REPEATS=30
        sleep $JUST_A_SECOND # Hey, wait a second!
        while [ $i -lt $REPEATS ]
        do
                echo '<---------FUN--------->'
                let 'i+=1'
        done
}

fun
exit $?
```

- When a function is called, commands in it are executed
- A function behaves like a (short) script
- It can be used in scripts
- Functions can take arguments and create local variables

Bash essentials

Builtins are basic commands that bash has built in it

.	dirs	history	shopt
:	disown	jobs	source
[	echo	kill	suspend
alias	enable	let	test
bg	eval	local	times
bind	exec	logout	trap
break	exit	popd	true
builtin	export	printf	type
caller	false	pushd	typeset
cd	fc	pwd	ulimit
command	fg	read	umask
compgen	getopts	readonly	unalias
complete	hash	return	unset
continue	help	set	wait
declare		shift	

- They are simple functions already provided to the users
- Include builtin keywords

Bash essentials

Keywords are reserved words that have a special meaning to the shell.

- They are used to begin and end the shell command
- Similar to builtins
- Keywords are recognised when unquoted
- They are interpreted in a special way

```
if
then
else
elif
fi
case
esac
for
select
while
until
do
done
in
function
time
{
}
!
[[
]]
```

Bash essentials

Executables are external commands or applications

- They are usually invoked by their name
- Bash has to be able to find them

- If a command is not an alias, a function, a builtin nor a keyword and it is specified without a file path, bash searches through the directories listed in the environmental variable PATH

- Directories are scanned from left to right and the first executable that matches is run

Bash scripting

Bash scripts are sequences of bash commands in a file

The first line of a script should be reserved for an interpreter directive also called *hashbang* or *shebang*. This is used when the kernel executes a non-binary file. Use one of the two following alternatives

- `#!/bin/bash`
- `#!/usr/bin/env bash`

Bash scripting

- Script execution requires the script has “execute” permissions

- The script can be executed in either of the following ways:

  bash scriptname # the shebang is considered a comment

  ./scriptname # the shebang is used

- The script can be made available as a command:
  - move the script to /usr/local/bin , 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.
  - Or include the directory containing the script in the user's $PATH

Bash scripting

- Write a bash script that:
  - Says “Hello!”
  - Displays date and time
  - Saves this information in a file
- Make the script executable and execute it
- Make the script available as a command

Bash scripting

- Write a bash script that:
  - Says “Hello!”
  - Displays date and time
  - Saves this information in a file
- Make the script executable and execute it
- Make the script available as a command

```
(base) MacBook-Pro-2:TRM_Dati milenavalentini$ man date
(base) MacBook-Pro-2:TRM_Dati milenavalentini$
(base) MacBook-Pro-2:TRM_Dati milenavalentini$ date
Tue Oct  3 01:00:43 CEST 2023
(base) MacBook-Pro-2:TRM_Dati milenavalentini$ date "+DATE: %Y-%m-%d%nTIME: %H:%M:%S"
DATE: 2023-10-03
TIME: 01:00:46
(base) MacBook-Pro-2:TRM_Dati milenavalentini$
```

Bash scripting

- Write a bash script that:
  - Says “Hello!”
  - Displays date and time
  - Saves this information in a file
- Make the script executable and execute it
- Make the script available as a command

```
#!/bin/bash

echo 'Hello!'

date

touch output_file
echo 'Hello!' > output_file
date >> output_file
```

Shell scripting

Many shells have scripting abilities: put multiple commands in a script and the shell executes them as if they were typed from the keyboard.

A command or a script that you can execute consists of one or more processes. The processes can be categorized into different broad groups. The main two are:

- interactive processes, and
- batch processed

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

Quoting

Single quotes: ‘string’ — what’s inside becomes a string

Shell scripting

Quoting

Single quotes: ‘string’ — what’s inside becomes a string

Double quotes: “ … ” — needed to perform actions, e.g. substitutions that begin with $

— they allow the shell to interpret dollar sign ($), backtick (`), backslash (\ ) and exclamation mark (!). These characters have special meaning when used with double quotes, and before display, they are evaluated

Shell scripting

```
(base) milena:~ milenavalentini$ echo 'Hello world'
Hello world
(base) milena:~ milenavalentini$ STRING='hello'
(base) milena:~ milenavalentini$ echo '$STRING world'
$STRING world
(base) milena:~ milenavalentini$ echo "$STRING world"
hello world
```

Shell scripting

Special characters

Single quotes: ‘string’

Double quotes: “ … ”

Whitespace: used to determine where words begin and end. The first word is the command name.

$: introduces different types of **expansion**

**** : put in front of a metacharacter removes its special meaning

: introduces a comment till the end of line

Shell scripting

Special characters

Single quotes: ‘string’

Double quotes: “ … ”

Whitespace: used to determine where words begin and end. The first word is the command name.

$: introduces different types of **expansion**

**** : put in front of a metacharacter removes its special meaning

: introduces a comment till the end of line

! : negate keyword, reverses a test or an exit status

= : assignment (white space not allowed on either side)

[[ ]] : testing keywords, allows to evaluate a conditional expression to determine if “true” or “false”

Shell scripting

Special characters

Single quotes: ‘string’

Double quotes: “ … ”

Whitespace: used to determine where words begin and end. The first word is the command name.

$: introduces different types of **expansion**

**** : put in front of a metacharacter removes its special meaning

: introduces a comment till the end of line

! : negate keyword, reverses a test or an exit status

= : assignment (white space not allowed on either side)

[[ ]] : testing keywords, allows to evaluate a conditional expression to determine if “true” or “false”

> >> < : redirection of a command’s output or input to a file

| : the pipeline sends the output from one command to the input of another command

; : command separator of multiple commands that are on the same line

Shell scripting

```
(base) milena:~ milenavalentini$ echo 'Hello world'
Hello world
(base) milena:~ milenavalentini$ STRING='hello'
(base) milena:~ milenavalentini$ echo '$STRING world'
$STRING world
(base) milena:~ milenavalentini$ echo "$STRING world"
hello world
(base) milena:~ milenavalentini$ a=1+2
(base) milena:~ milenavalentini$ echo 'One plus Two is $a'
One plus Two is $a
(base) milena:~ milenavalentini$ echo "One plus Two is $a"
One plus Two is 1+2
(base) milena:~ milenavalentini$ ((b=1+2))
(base) milena:~ milenavalentini$ echo 'One plus Two is $b'
One plus Two is $b
(base) milena:~ milenavalentini$ echo "One plus Two is $b"
One plus Two is 3
```

(( )) : within an arithmetic expression, **mathematical operators (+ - * /)** are used for calculations. Round brackets used for:

- variable assignments ((a=1+4))
- tests evaluation ((a<b))

Shell scripting

```
(base) milena:~ milenavalentini$ echo 'Hello world'
Hello world
(base) milena:~ milenavalentini$ STRING='hello'
(base) milena:~ milenavalentini$ echo '$STRING world'
$STRING world
(base) milena:~ milenavalentini$ echo "$STRING world"
hello world
(base) milena:~ milenavalentini$ a=1+2
(base) milena:~ milenavalentini$ echo 'One plus Two is $a'
One plus Two is $a
(base) milena:~ milenavalentini$ echo "One plus Two is $a"
One plus Two is 1+2
(base) milena:~ milenavalentini$ ((b=1+2))
(base) milena:~ milenavalentini$ echo 'One plus Two is $b'
One plus Two is $b
(base) milena:~ milenavalentini$ echo "One plus Two is $b"
One plus Two is 3
(base) milena:~ milenavalentini$ printf 'go to \\n new line'
go to \n new line(base) milena:~ milenavalentini$ printf "go to \\n new line"
go to
 new line(base) milena:~ milenavalentini$ printf "go to \\nnew line \\n"
go to
new line
(base) milena:~ milenavalentini$
```

Shell scripting

```
(base) milena:test_bash milenavalentini$ echo ((c=10+1))
-bash: syntax error near unexpected token `('
(base) milena:test_bash milenavalentini$
(base) milena:test_bash milenavalentini$ ((c=10+1))
(base) milena:test_bash milenavalentini$ echo $c
11
(base) milena:test_bash milenavalentini$ c=10+1
(base) milena:test_bash milenavalentini$ echo $c
10+1
(base) milena:test_bash milenavalentini$ d=$((15+2))
(base) milena:test_bash milenavalentini$ echo d
d
(base) milena:test_bash milenavalentini$ echo $d
17
```

$(( )) : comparable to (( )), but the expression is replaced with the result of its evaluation

$(...) : command substitution

Special characters

$: introduces different types of **expansion**

Shell scripting

Expansions. They come in different flavours:

Brace expansion: mechanism by which arbitrary strings may be generated

```
(base) milena:~ milenavalentini$ echo a{d,c,b}e
ade ace abe
```

Tilde expansion: the unquoted ~ character is usually replaced with the value of the home shell variable.

```
(base) milena:~ milenavalentini$ echo $HOME
/Users/milenavalentini
(base) milena:~ milenavalentini$ cd ~
(base) milena:~ milenavalentini$ pwd
/Users/milenavalentini
(base) milena:~ milenavalentini$ cd Desktop/
(base) milena:Desktop milenavalentini$ cd ~
(base) milena:~ milenavalentini$ pwd
/Users/milenavalentini
(base) milena:~ milenavalentini$ cd Desktop/WorkingOn/
(base) milena:WorkingOn milenavalentini$ ls ~/Downloads/
```

Shell scripting

Expansions. They come in different flavours:

Brace expansion: mechanism by which arbitrary strings may be generated

```
(base) milena:~ milenavalentini$ echo a{d,c,b}e
ade ace abe
```

Tilde expansion: the unquoted ~ character is usually replaced with the value of the home shell variable.

```
(base) milena:WorkingOn milenavalentini$ ls ~/Downloads/
```

Command substitution: allows the output of a command to replace the command itself. It occurs when:

```
$(command)
```

Arithmetic expansion: allows the evaluation of the expression and the substitution of the result. Format:

```
(( expression ))
```

Filename expansion: After word splitting, unless the -f option has been set, Bash scans each word for the characters ‘ * ’, ‘ ? ’, and ‘ [’. If one of these characters appears (and is not quoted), then the word is regarded as a *pattern*, and replaced with an alphabetically sorted list of filenames matching the pattern.

Shell scripting

Expansions. They come in different flavours:

Filename expansion: After word splitting, unless the -f option has been set, Bash scans each word for the characters ‘ * ’, ‘ ? ’, and ‘ [’. If one of these characters appears (and is not quoted), then the word is regarded as a *pattern*, and replaced with an alphabetically sorted list of filenames matching the pattern.

```
(base) milena:test_bash milenavalentini$ ls
file_100.txt    file_101.txt    file_102.txt    file_103.txt    script.bash
(base) milena:test_bash milenavalentini$
(base) milena:test_bash milenavalentini$ cat script.bash
#!/bin/bash

for a in file_*; do
  echo "$a"
done

(base) milena:test_bash milenavalentini$ ./script.bash
file_100.txt
file_101.txt
file_102.txt
file_103.txt
(base) milena:test_bash milenavalentini$ _
```

Shell scripting

Expansions. They come in different flavours:

Brace expansion: mechanism by which arbitrary strings may be generated

```
(base) milena:~ milenavalentini$ echo a{d,c,b}e
ade ace abe
```

Tilde expansion: the unquoted ~ character is usually replaced with the value of the home shell variable.

```
(base) milena:WorkingOn milenavalentini$ ls ~/Downloads/
```

Command substitution: allows the output of a command to replace the command itself. It occurs when:

```
$(command)
```

Arithmetic expansion: allows the evaluation of the expression and the substitution of the result. Format:

```
(( expression ))
```

Filename expansion: characters ‘ * ’, ‘ ? ’, and ‘ [’ that can be regarded as a *pattern*.

And also: shell parameter expansions, word splitting, …

Shell scripting

Exercises.

Exercise 1
Create a new directory, move into it and run the command

touch file{1..20}{.{dat,png,txt},\ backup.dat,_bkp.png}

a. — Understand what happened using ls.
b. — List only files with the .dat extension.
c. — List only files with number 13 in the name.
d. — List only backup files.
e. — List only files containing a space in the name.
f. — List files with a number that is multiple of 5 before the dot.

Write a bash script which performs all the tasks above, and execute it.

Make the script write the commands and the output of the commands on a file.

Make the script available as a command.