

Utilisation avancée d'Unix et Programmation Shell

ENSIMAG

2017-2018



Outline

- 1 Introduction au cours
- 2 Introduction
- 3 Utilisation interactive du shell (bash)
- 4 Shell-scripts
- 5 Commandes utiles



Bienvenue dans le cours « Unix Avancé »

- Contenu :
 - ▶ Quelques aspects intéressants d'Unix qui vous facilitent la vie
 - ▶ Programmation avec le shell unix
 - ▶ Introduction aux gestionnaires de version (git)
- Utilité :
 - ▶ Gagner du temps au quotidien en automatisant des tâches répétitives et apprendre un langage de programmation (utilisé au moins dans le projet GL en 2A)
 - ▶ Apprendre à sauvegarder et partager efficacement son code
- Séances :
 - ▶ Aujourd'hui :
 - 1 Transparents (1h30)
 - 2 TP pas à pas (1h30, en salle machines)
 - ▶ Demain :
 - 3 Introduction à Git (gestionnaire de version) pas à pas (1h30, en salle machines)
 - ▶ Prochaines semaines : salles machines
 - 4 Suite du TP en libre-service encadré
 - 5 Fin (?) du TP en libre-service encadré



TP Libre Service

- Sujet : réaliser une page HTML de « vignettes » d'images
- Programmation shell
- Utilisation de Git (gestionnaire de version)
- En binômes
- Rendu sur TEIDE : fichier TAR avec le code, un rapport en html et l'historique git des versions du code et du rapport
- Note intégrée avec l'examen de TP Unix (matière « Unix : introduction et programmation »)
- Lecture du polycopié « Le Bourne Shell » indispensable, recherche d'information complémentaires sur Internet conseillée



Jeu de piste, partie 2

- Départ en bas de page :
http://ensiwiki.ensimag.fr/index.php/TP_Unix_-_Jeu_de_piste
- Aborde des notions un peu avancées (Unix, réseau, ...), mais largement faisable !
- Pas de note, mais amusez-vous bien ;-)



- 1960s Multics (Multiplexed Information and Computing Service),
- 1969 Ken Thompson et Dennis Ritchie écrivent la première version d'Unix, en assembleur.



"something as complex as an operating system, which must deal with time-critical events, had to be written exclusively in assembly language"

Ken (assis), Dennis (debout) devant un PDP-11, 1972

- 1973 Ré-écriture d'Unix en langage C
- 1988 Norme POSIX = « Portable Operating System Interface » for Unix
- 1991 Linux 0.0.1, écrit par Linus Torvalds à 21 ans
⇒ vers l'arrivée d'Unix sur les ordinateurs personnels

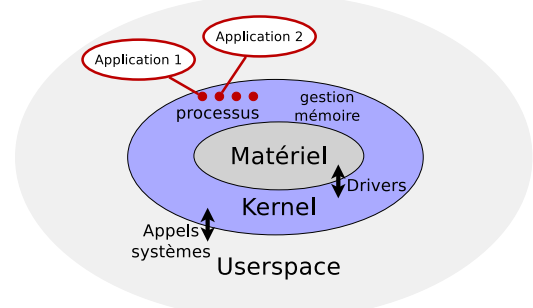


Unix aujourd'hui

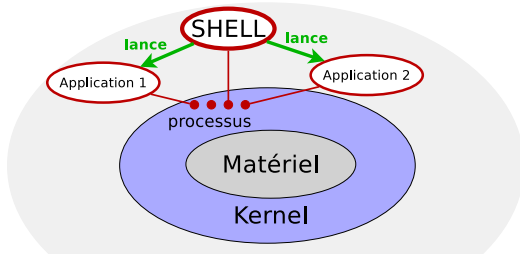
- Mac OS X est un Unix
- (GNU) Linux est un Unix (pas certifié officiellement)
- Les 500 plus gros ordinateurs de la planète fonctionnent sous Unix (dont 99.4 % sont des Linux)
- Et dans les smartphones ?
 - ▶ Android > 80 % du marché est basé sur Linux
 - ▶ iOS > 15 % du marché est un Unix
- Dans les box internet aussi, c'est du Linux
- ...



Composants d'un système complet



Composants d'un système complet



Shell : **application interactive qui permet d'en lancer d'autres !**

Exemples : explorateur de fichiers Windows (Shell graphique), interprète de commandes Unix, ...



Principe du shell interactif

- Pseudo-algorithme :

```
while True:
    commande = lire_commande() # (1)
    commande = expander(commande) # (2)
    executer(commande) # (3)
```

- Exemple : `$ ls -l *.c`

- Entré dans un éditeur de ligne (1)

- ▶ Saisie de caractères
- ▶ Navigation dans la ligne (flèches gauche/droite)
- ▶ Historique
- ▶ ...

- Expansé en `ls -l toto.c autre-fichier.c` (2)

- ▶ « wildcards » comme `*`, `[abc]`, `?` expansés à partir des noms de fichiers (appelé aussi expansion des « glob »),
- ▶ Variables comme `$toto` remplacées par leur valeur.
- ▶ ...

- Exécute la commande `ls` avec les arguments `-l`, `toto.c`, et `autre-fichier.c` (3)

- ▶ Fait appel au système d'exploitation
- ▶ (Rendez-vous en 2A pour les détails)



Éditeur de ligne du shell

- Complétion avec TAB
 - ▶ Moins de choses à taper
 - ▶ Moins de fautes de frappes
 - ▶ C'est la fonctionnalité qui fait que le shell est souvent plus efficace qu'une interface graphique
- Historique
 - ▶ Flèches haut/bas
 - ▶ ! (bang) : `!!`, `!n`, `!string`, `!?string`, ...
- Recherche en arrière
 - ▶ `C-r` (Control+r) pour rechercher une commande dans l'historique
- Complétion avancée
 - ▶ `Alt-g` affichage de l'expansion « glob »
 - ▶ `C-x*` expansion « glob » manuelle
- et bien d'autres raccourcis... `bind -P` donne une liste



Substitutions, expansions : les wildcards

- Wildcards : remplacements de motifs par rapport aux fichiers existants
 - ▶ `*` : n'importe quelle sous-chaine (sauf un point en début de nom de fichier),
 - ▶ `?` : n'importe quel caractère (sauf un point en début de nom de fichier),
 - ▶ `[abc]` un a, un b ou un c,
 - ▶ `[!abc]` (ou `[^abc]`, non-POSIX) n'importe quel caractère sauf un a, un b, ou un c.
 - ▶ `debut{un,deux,trois}fin` expansé en « `debutunfin` », « `debutdeuxfin` », « `debuttroisfin` » (sans rapport avec les fichiers existants) (marche en bash, non POSIX).
- Exemple : `rm *.od[tp]` ⇒ supprime tous les fichiers `.odt` (fichier OpenDocument Text) et `.odp` (fichier OpenDocument Presentation).



les wildcards : exercices

- Exercice : Comment faire pour reconnaître tous les fichiers et dossiers dans le répertoire courant (sauf `.` et `..`) ?
⇒ `*.[^.]*.*?`
 - ▶ Les fichiers ne commençant pas par `.`
 - ▶ Ceux commençant par `.` suivis d'autre chose qu'un point
 - ▶ Ceux commençant par `..` suivis d'autre chose
- Exercice : Comment renommer un fichier
`un-fichier-avec-un-nom-long.txt` en
`un-fichier-avec-un-nom-long.txt.bak` sans taper deux fois le nom du fichier ?
⇒ `mv un-fichier-avec-un-nom-long.txt{, .bak}`
 - ▶ `{, .bak}` avec la chaîne vide en première position
 - ▶ Expansé en `mv un-fichier-avec-un-nom-long.txt un-fichier-avec-un-nom-long.txt.bak`



Substitutions, expansions : les variables

- Principe
 - ▶ Définition (globale) :
 - ★ `x=toto`
 - ▶ Utilisation :
 - ★ `ls $x`
 - ★ `ls ${x}` (équivalent, mais `$(x)y` est différent de `$xy`)
- Précautions ...
`x="fichier avec espaces.txt"`
`rm ${x}`
⇒ essaye de supprimer les fichiers « `fichier` », « `avec` », et « `espaces.txt` »
- ⇒ c'est plus compliqué que ça n'en a l'air ...



L'interprétation des blancs, ou « Découpage »

- En fait, l'interprétation de la ligne de commande est un peu plus compliquée :
 - ▶ Substitution des variables
 - ▶ Interprétation des blancs
 - ▶ Expansions des wildcards
- Interprétation des blancs = découpage de la ligne de commande (commande, argument 1, argument 2, ...)
- Exemple : `ls -l toto.c titi.c` découpé en « `ls` », « `-l` », « `toto.c` », « `titi.c` ».



Substitutions, expansions : l'interprétation des blancs

- L'interprétation des blancs arrive **après** les substitutions de variables
⇒ `x="fichier avec espace.txt"; ls -l $x` cherche trois fichiers ...
- L'interprétation des blancs arrive **avant** l'expansion des wildcards
⇒ `ls -l *` marche correctement même avec des espaces dans les noms de fichiers.



Jouer avec l'interprétation des blancs : les guillemets

Backslash \x considère x comme un caractère « normal »

- `ls -l fichier\ avec\ espaces.txt` fait ce qu'il faut.

Guillemets simples (single quotes) 'chaîne de caractère' : seul le guillemet simple est encore un caractère spécial. Les blancs, dollars et autres sont des caractères comme les autres.

- `ls -l 'fichier avec espaces.txt'` marche.
- `ls -l 'fich$avec<cars!speciaux.txt'` aussi.

Guillemets doubles (double quotes) "chaîne de caractères" : les blancs ne sont plus des caractères spéciaux, le shell ne coupera pas la chaîne en deux. \$, !, \ sont encore actifs, mais pas les wildcards (*, ...)

- `ls -l "$x"` est la manière correcte d'appeler `ls -l` sur un fichier contenu dans la variable x.
- `ls -l "\$x"` affiche le fichier « \$x ».



Guillemets et variable

- Règle d'or :

Si on n'a pas une bonne raison de faire autrement, on met **toujours** des guillemets doubles autour des variables.

- `"$toto" = :-)`
- `$toto = :-{`



Expansion de commande : \$(...)

- `ls -l $(commande)` exécute `commande`, et remplace `$(commande)` par le résultat.
- `ls -l $(find . -name "*.c")` va exécuter `ls -l` avec tous les fichiers `.c` trouvés dans un sous-répertoire du répertoire courant (attention, non robuste aux espaces dans les noms de fichiers);



Expansion arithmétique

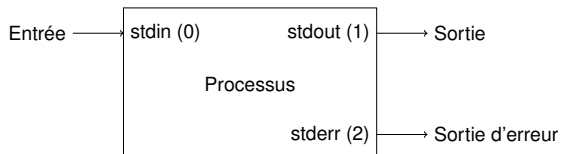
`$(...)` évalue le contenu « ... » comme une expression.

- Exemples
- `echo $((40 + 2))` (affiche 42)
 - `x=$((x + 1))` (incrmente x)
 - `x=$(($x + 1))` (idem)



Entrées-sorties dans un terminal Unix

- Sous Unix, chaque processus peut
 - ▶ Lire des entrées (au clavier),
 - ▶ Écrire une sortie (à l'écran),
 - ▶ Émettre des messages d'erreur (confondus avec la sortie par défaut).



Une des commandes les plus simples : « cat »

- `cat` sans argument :
 - ▶ Lit des lignes au clavier
 - ▶ Affiche la même chose à l'écran
- Pour envoyer une fin de fichier au clavier : `C-d`.

Demo



Fonctionnement de cat



Une autre commande très simple : « echo »

- `echo arguments` : affiche ses arguments à l'écran.

```
$ echo bonjour
bonjour
$ echo au revoir
au revoir
```



Redirection des entrées-sorties

Redirection de la sortie standard `echo bonjour > fichier.txt` :

- exécute `echo bonjour`
- met la sortie dans `fichier.txt` au lieu de l'afficher.

Redirection de l'entrée `commande < fichier.txt`

Exemples avec `cat`

- L'éditeur de texte le plus simple au monde :
`cat > fichier.txt`
- Afficher le contenu d'un fichier :
`cat < fichier.txt` (ou `cat fichier.txt`).



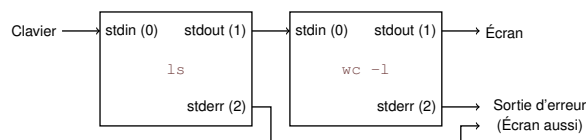
Redirection des erreurs

- `ls -l foo.c bar.c > sortie.txt 2> erreurs.txt`
 - ▶ `sortie.txt` reçoit la sortie (liste des fichiers)
 - ▶ `erreurs.txt` reçoit les erreurs (fichiers inexistant, erreurs d'accès disque...)
- Ignorer les erreurs : commande `2>/dev/null`
(`/dev/null` est un fichier spécial « puits sans fond »)



Redirections, pipes

- On peut mettre plusieurs processus bout à bout.
- Exemple : `ls | wc -l`



Question



Que fait ce `ls | wc -l` ?

- Combinables à volonté :
`cmd1 | cmd2 | cmd3`
`cmd1 < in.txt | cmd2 | cmd3 > out.txt`



Scripts shell Vs Shell interactif

- « Script » = « programme », en général « vite fait bien fait » et basé sur des chaînes de caractères.
- Tout ce qui peut être tapé dans un shell peut aussi être mis dans un script
- Et inversement !



Tout premier script

- Un script-shell = fichier commençant par `#!/bin/sh`
- Exemple :

```
#!/bin/sh  
  
echo 'Bonjour, ceci est mon premier script'  
echo "Ca y est, c'est deja fini ..."
```
- Il faut le rendre exécutable : `chmod +x fichier-script`
- Et on l'exécute : `./fichier-script`



Conditions, statut de retour, ...

- À la fin de l'exécution, un processus dit « si tout s'est bien passé » avec un nombre.
 - ▶ 0 si tout s'est bien passé
 - ▶ > 0 si il y a eu une erreur (grave ou pas)
- Accessible avec la variable `$?` :

```
$ ls fichier.txt  
fichier.txt  
$ echo $?  
0  
$ ls fichier-inexistant.txt  
ls: fichier-inexistant.txt: No such file or directory  
$ echo $?  
2
```
- En C, c'est la valeur renvoyée par la fonction `main`.



```
if ... then ... else ... fi
```

- Syntaxe :

```
if commande; then  
  instruction1  
  instruction2  
else  
  instruction3  
fi
```
- Sémantique : exécute `commande`, et la branche `then` ssi la commande a terminé sur un statut 0 (i.e. `$?` est 0).
- Exemple (simpliste) :

```
if ls toto.txt; then  
  echo "ls a reussi"  
else  
  echo "ls a echoue"  
fi
```



Conditions : `test`

- `test` = commande qui fait des tests en fonctions de ses arguments.
- Exemple : `test -f toto.txt` renvoie 0 si `toto.txt` est un fichier.
- Tests possibles (entre autres)
 - f `arg` `arg` est un fichier
 - d `arg` `arg` est un répertoire
 - x `arg` `arg` est exécutable
 - e `arg` `arg` existe (fichier, répertoire, ...)
 - ch1 = ch2 ch1 est égal à ch2
 - ch1 != ch2 ch1 n'est pas égal à ch2
- Comparaisons de nombres `-eq, -ge, -gt, -le, -lt, -ne` :
equal, greater or equal, greater than, lower than,
lower or equal, not equal.
- `man test`



Syntaxe alternative

```
if [ -f "$fichier" ]; then ...
```

- Commande [équivalente à `test`, mais exige que son dernier argument soit]



Opérations booléennes

```
[ ... ] && [ ... ] « et » booléen  
[ ... ] || [ ... ] « ou » booléen  
[ ! ... ] ou bien ! [ ... ] « non » booléen
```

Exemple :

```
if [ -r "$file" ] && ! [ -L "$file" ]  
then  
  echo "$file is readable and"  
  echo "is not a symbolic link"  
fi
```



Exemple de if/then/else : les fichiers

```
if test -f "$fichier"; then  
  echo "$fichier est un fichier"  
elif test -d "$fichier"; then  
  echo "$fichier est un repertoire"  
else  
  echo "c'est autre chose"  
fi
```



Exemple de if/then/else : chaines, nombres

```
if [ "$chaine" = "toto" ] || [ "$chaine" = "tutu" ]  
then  
  echo "chaine est egal a toto ou tutu"  
elif [ "$nombre" -lt 4 ]  
then  
  echo "$nombre est plus petit que 4"  
elif [ "$nombre" -ge 4 ]  
then  
  echo "$nombre est plus grand ou egal a 4"  
fi
```

⚠ Les espaces sont importants!



```
for ... in ... done
```

```
for i in un deux trois; do  
  echo "cette fois, i vaut $i"  
done  
affiche  
  
cette fois, i vaut un  
cette fois, i vaut deux  
cette fois, i vaut trois
```



Exemple utile avec for

```
for i in *.py; do  
  ... $i ...  
done
```



```
while ... do ... done
```

```
while commande; do  
  bloc d'instructions  
done
```

- Exécute la commande, et si elle renvoie 0, exécute le bloc d'instructions et reboucle.
- Très utile avec `test`, comme pour `if`.



```
case ... in ... esac
```

```
case "$i" in  
  "valeur")  
    echo "i vaut valeur"  
    ;;  
  --*)  
    echo "i commence par tiret-tiret"  
    ;;  
  "un"|"deux")  
    echo "i est soit un soit deux"  
    ;;  
  *)  
    echo "i est autre chose"  
    ;;  
esac
```



Passage de paramètres à un script

- `$1, $2, ..., ${42}` : premier, deuxième, ..., quarante-deuxième arguments du script.
- `"$@"` : tous les arguments
- `$#` : nombre d'arguments
- `$0` : nom de l'exécutable (ou pas)
- Exemple :

```
#!/bin/sh
```

```
echo "l'exécutable est $0"  
echo "le premier argument est $1"  
echo "le deuxième est $2"  
echo "et au total, il y a $# arguments"
```



Passage de paramètres

- `set arg1 arg2...` : remplace les arguments `$1, $2, ...` par `arg1, arg2, ...`
- `shift` : oublie `$1`, et décale `$2, $3, ...` vers la gauche.
- Exemple :

```
#!/bin/sh

echo "$0" "$@"
set un deux trois
echo "$0" "$@"
shift
echo "$0" "$@"

et ./mon-script one two affichera :
./mon-script one two
./mon-script un deux trois
./mon-script deux trois
```



Exercice

- Faire un script qui, pour chaque argument :
 - ▶ Affiche « option un » si l'argument est « --un »
 - ▶ Affiche « autre option » si l'argument commence par « -- »
 - ▶ Affiche « autre chose : argument » sinon.
- Utiliser `while`, `test`, `case`, `shift`, `$#` et `$1`.



Exercice : solution

```
#!/bin/sh
while test $# -ne 0; do
  case "$1" in
    "--un")
      echo "option un"
      ;;
    "--*")
      echo "autre option"
      ;;
    *)
      echo "autre chose"
      ;;
  esac
  shift
done
```

⇒ Très utile pour « parser » la ligne de commande
(nb : on peut aussi utiliser `getopt` (man 1 getopt) ou `getopts` (man bash))



Fonctions en shell

- Déclaration :

```
ma_fonction () {
  echo "appel de ma_fonction avec arguments"
  echo "$1, $2, $3 ... ($# au total)"
}
```
- Appel : comme une commande

```
ma_fonction "premier argument" arg2 arg3 arg4
```



man : manuel

- Attention, toute cette section donne une vue très succincte des possibilités de chaque commande
- `man commande` pour les détails
- `man man...`



cat

- `cat` sans argument : lit sur son entrée standard, recopie sur sa sortie standard (pas très utile)
- `cat fichier` : affiche le contenu du fichier sur la sortie standard
- `cat fichier1 fichier2 ...` : affiche la concaténation des fichiers sur la sortie standard.
- Exercice : donner une formulation plus simple de `cat /etc/passwd | wc -l` (UUOC = Useless Use Of Cat)
⇒ `wc -l < /etc/passwd` (ou `wc -l /etc/passwd`)



grep : Global Regular Expression Print

- `grep toto fichier.txt` : affiche toutes les lignes de `fichier.txt` contenant `toto`.
- `commande | grep toto` : lance `commande`, mais n'affiche que les lignes de la sortie contenant `toto`.
- `grep 'to.o' fichier.txt` : affiche toutes les lignes de `fichier.txt` contenant la chaîne `to` suivie de n'importe quel caractère, suivi d'un `o`.
- ⇒ `grep` recherche en fait une expression régulière ...



Expressions régulières (regex)

- Basée sur la théorie des langages... avec une syntaxe texte :
 - ▶ `a` : le caractère `a`
 - ▶ `abc` : la chaîne `abc`
 - ▶ `.` : n'importe quel caractère
 - ▶ `[abc]` : un des caractères `a, b, ou c`
 - ▶ `[a-z0-9]` : un caractère compris entre `a` et `z` ou entre `0` et `9`
 - ▶ `[^abc]` : ni `a`, ni `b`, ni `c`
 - ▶ `\(expression\)` : l'expression, avec parenthèses de groupement
 - ▶ `expression*` : expression, répétée un nombre quelconque de fois
 - ▶ `\(expr1\|expr2\)` : `expr1` ou `expr2`
 - ▶ `expression?` : l'expression, ou la chaîne vide (ne marche qu'avec `grep -E`)
 - ▶ `^` : début de ligne
 - ▶ `$` : fin de ligne
 - ▶ `\., \?, ...` : le caractère `.`, le caractère `?`, ...



Expressions régulières : exemple

- Liste des connexions d'un utilisateur : last

```
telesun:> last
[...]
```

```
moy pts/12 bauges.imag.fr Fri Apr 11 15:02 still logged in
autre pts/42 quelque.part.fr Fri Apr 11 15:01 - 15:02 (00:01)
moy pts/7 bauges.imag.fr Fri Apr 11 15:00 still logged in
moy pts/7 bauges.imag.fr Fri Apr 11 15:00 - 15:00 (00:00)
[...]
```
- Liste de mes connexions encore ouvertes :

```
telesun:>last | grep '^moy.*still logged in *$'
moy pts/12 bauges.imag.fr Fri Apr 11 15:01 still logged in
moy pts/7 bauges.imag.fr Fri Apr 11 15:00 still logged in
```
- Explications : on affiche chaque ligne qui
 - Commence par moy (^moy),
 - puis n'importe quoi (.*),
 - puis la chaîne still logged in,
 - puis une suite quelconque d'espaces (*) avant la fin de la ligne (\$).



find

- Rechercher un fichier,
- find . : afficher tous les fichiers dans le répertoire courant et ses sous-répertoires,
- find /home/ : tous les fichiers dans /home/ ou ses sous-répertoires,
- find . -name '*.py' : tous les fichiers dont le nom correspond à *.py
- find . -type d : tous les répertoires
- find . -name '*~' -exec rm -i {} \; : exécuter la commande rm -i sur tous les fichiers terminant par ~ dans le répertoire courant et ses sous-répertoires.



diff

- Comparer deux fichiers
- Fichiers identiques : statut 0 et pas de sortie.

```
$ diff foo.txt bar.txt
$ echo $?
0
```
- Fichiers différents : statut > 0 et visualisation des différences.

```
$ diff -u hello.c bonjour.c
--- hello.c      2008-04-11 19:49:31.000000000 +0200
+++ bonjour.c   2008-04-11 19:49:49.000000000 +0200
@@ -1,5 +1,5 @@
 int main () {
-     printf("Hello, world\n");
+     printf("Bonjour tout le monde\n");
     return 0;
 }
$ echo $?
1
```



Manipuler des noms de fichiers

- basename /path/to/toto.txt : nom du fichier sans le répertoire (toto.txt)
- basename /path/to/toto.txt .txt : nom du fichier sans le répertoire ni le suffixe donné (toto)
- dirname /path/to/toto.txt : nom du répertoire (/path/to)



cut

- Découper un texte en colonnes
- cut -f 2 : récupérer la deuxième colonne (délimiteur = tabulation)
- cut -f 3 -d : : récupérer la troisième colonne (délimiteur : deux-points)
- Exemple : cut -f 5 -d : /etc/passwd : récupérer les noms des utilisateurs.



Trier

- sort : trie les lignes de l'entrée par ordre alphabétique
- uniq : supprime les doublons dans un ensemble de lignes triées
- Exemple : last | cut -f 1 -d ' ' | sort | uniq : liste des utilisateurs qui apparaissent au moins une fois dans last.



xargs

- Construit et exécute une commande à partir de son entrée standard
- cmd1 | xargs cmd2 va exécuter cmd1, obtenir une sortie sortie puis construire puis exécuter la commande cmd2 sortie.
- Exemple :
 - find . -name '*.py' | grep toto : cherche tous les fichiers *.py et n'affiche que ceux dont le nom contient toto,
 - find . -name '*.py' | xargs grep toto : cherche les *.py, et exécute grep toto fichier1.py fichier2.py ... (i.e. fait une recherche sur le contenu, pas le nom)



sed : Stream Editor

- « Éditeur de texte », mais non-interactif. Très puissant pour faire des transformations syntaxiques sur du texte.
- Principale utilité : substitution d'expressions.
sed 's/expr/chaîne/g'
ou sed 's#expr#chaîne#g'. L'option g à la fin permet de remplacer expr plusieurs fois sur une même ligne.
- Exemples :
 - sed 's/toto/titi/g' < fich1.txt > fich2.txt : remplacer tous les toto par des titi dans fich1.txt et mettre le résultat dans fich2.txt
 - pwd | sed 's#^.*/# #' : obtenir le nom du répertoire courant (équivalent à basename \$(pwd))
 - last | sed 's/^\([^]*\) .*\$/\1/' : extraire seulement la première colonne
 - le \1 est remplacé par ce à quoi la première paire de parenthèses a correspondu
 - Le contenu des \([^]*\) s'arrête au premier espace.



wc

- Compter les mots, les lignes, les caractères
- `wc *.py`
- `ls | wc`
- ...



read

- Commande interne
- Poser des questions à l'utilisateur du script, attendre et exploiter la réponse
- lire une ligne (jusqu'à `<enter>`) et stocker la valeur dans une variable
- ou bien lire un nombre fixe de caractère
- Exemples :
 - ▶ `read -p "Entrez une ligne" ligne` affiche une phrase et attend l'entrée d'une ligne en réponse. La ligne est stockée dans la variable `$ligne`
 - ▶ `read -s -n1` attend que l'utilisateur tape une touche sans l'afficher et stocke la touche dans la variable `$REPLY`

