Unix : Commandes et expressions régulières

Guy Tremblay
Professeur

Département d'informatique
UQAM
http://www.labunix.uqam.ca/~tremblay_gu

INF600A
11 septembre 2018
Labo. du 13 septembre 2018


find
awk?
grep?
sed?
pwd!


xargs
awk?
grep?
sed?
pwd!

Un graphe illustrant l'évolution d'Unix depuis ses débuts, ses différentes versions et déclinaisons...

http://www.levenez.com/unix/unix.pdf

Note : Il faut agrandir — énormément — l'unique page de ce document PDF !

Avertissement

- Ce chapitre = présentation partielle de quelques commandes Unix.
  - `man bash` : 196 pages
  - `man man` : 13 pages
    $\approx$ 30 options
  - `man ls` : 10 pages
    $\approx$ 40 options

- Commandes qu'on utilise typiquement au niveau ligne de commandes ou dans des scripts simples — pipelines !

- Principales commandes et options que j'utilise ou que j'ai utilisées dans ma pratique quotidienne, dans mes scripts, plus quelques autres... que j'aurais aimé connaitre ☺

Remarque

Les diapositives telles que celle-ci — fond bleu/gris pâle, en-tête plus pâle et symbole «⋆» à droite — présentent des compléments d'information. . . qui ne seront pas nécessairement vus/présentés en cours.

Ces éléments sont inclus par souci de complétude. Si certains d'entre eux s'avèrent nécessaires pour les laboratoires ou les devoirs, cela vous sera souligné en cours ou par courriel.

1. Philosophie Unix

La philosophie d'Unix et des outils Unix

This is the Unix philosophy : Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface.

Source: «*Basics of the Unix Philosophy*», D. McIlroy *

∗. L'inventeur des *pipes* Unix.

La philosophie d'Unix et des outils Unix

- *Do one thing well*
- *Process lines of text, not binary*
- *Use regular expressions*
- *Default to standard I/O*
- *Don't be chatty*
- *Generate the same format accepted as input*

Source: «*Classic Shell Scripting*», Robbins & Beebe, 2005.

La philosophie d'Unix et des outils Unix

- *Do one thing well*
- *Process lines of text, not binary*
- *Use regular expressions*
- *Default to standard I/O*

  Quel style de programmation est-ce que cela permet/favorise ?
- *Don't be chatty*

  Quel style de programmation est-ce que cela permet/favorise ?
- *Generate the same format accepted as input*

  Quel style de programmation est-ce que cela permet/favorise ?

Source: «*Classic Shell Scripting*», Robbins & Beebe, 2005.

La philosophie d'Unix et des outils Unix

- *Do one thing well*
- *Process lines of text, not binary*
- *Use regular expressions*
- *Default to standard I/O*

  Quel style de programmation est-ce que cela permet/favorise ?
- *Don't be chatty*

  Quel style de programmation est-ce que cela permet/favorise ?
- *Generate the same format accepted as input*

  Quel style de programmation est-ce que cela permet/favorise ?

Filtres et pipelines

Source: «*Classic Shell Scripting*», Robbins & Beebe, 2005.

La philosophie d'Unix et des outils Unix

«*Don't be chatty*»

Rule of silence = Developers should design programs so that they do not print unnecessary output. This rule aims to allow other programs and developers to pick out the information they need from a program's output without having to parse verbosity.

Source: E. Raymond, cité dans `https://en.wikipedia.org/wiki/Unix_philosophy`

«*Generate the same format as input*»

Expect the output of every program to become the input to another, as yet unknown, program. Don't clutter output with extraneous information. Avoid stringently columnar or binary input formats. Don't insist on interactive input.

Source: D. McIlroy, cité dans `http://www.catb.org/esr/writings/taoup/html/ch01s06.html`

2. Quelques commandes de base

2.1 Émission sur la sortie standard

La commande `cat`

Rôle = Copier le contenu d'un ou plusieurs fichiers vers la sortie standard

Copie d'un ou plusieurs fichiers

```
$ cat f1.txt
abc
def

$ cat f2.txt
xxx yyy zzz

$ cat f1.txt f2.txt
abc
def
xxx yyy zzz
```

La commande `cat`

Note : En rouge = caractères tapés au clavier

Un des fichiers peut être l'entrée standard...

```
$ cat
xxx
xxx
yyy
yyy
^D

$ cat f1.txt
abc
def

$ cat <f1.txt
abc
def
```

La commande `cat`

Un des fichiers peut être l'entrée standard, y compris dans une liste de fichiers explicites où on utilise alors «−»

```
$ cat <f1.txt
abc
def

$ cat <f2.txt
xxx yyy zzz

$ cat f1.txt - f2.txt <f1.txt
abc
def
abc
def
xxx yyy zzz
```

Les commandes `head` et `tail`

Rôle = Obtenir le début ou la fin d'un fichier

```
$ cat f3.txt
ab
cd
ef
gh
ij
lk
```

```
$ head -2 f3.txt
ab
cd

$ tail -3 f3.txt
gh
ij
lk
```

Note : Valeur par défaut = 10 : «`tail`» == «`tail -10`»

Note : «`tail -F`» : La commande attend qu'une ligne soit ajoutée au fichier puis affiche cette ligne. Termine avec `^C`.

La commande `echo`

Rôle = Émettre sur `stdout` les arguments, i.e., une série de mots, séparés par une (1) espace... à moins que le texte ne soit entre guillemets

```
$ echo Bonjour le monde
Bonjour le monde

$ echo Bonjour le     monde ...
Bonjour le monde ...

$ echo "Bonjour le monde"
Bonjour le monde

$ echo "Bonjour le     monde ..."
Bonjour le     monde ...
```

Note : Très utile pour tester des expressions au niveau du *shell*.

La commande `echo`

Avec substitution de variables. . .

```
$ VAR=Bonjour

$ echo VAR
VAR

$ echo $VAR
Bonjour

$ echo "$VAR le monde"
??

$ echo '$VAR le monde'
??
```

??

La commande `echo`

Avec substitution de variables, ou non... selon le type de guillemets

```
$ VAR=Bonjour

$ echo VAR
VAR

$ echo $VAR
Bonjour

$ echo "$VAR le monde"
Bonjour le monde

$ echo '$VAR le monde'
$VAR le monde
```

Donc : Avec des guillemets (doubles), le contenu d'une variable est interpolé (*variable expansion*), mais pas avec des apostrophes (guillemets simples) !

La commande `echo`

Pour ne générer aucun saut de ligne

```
$ echo Bonjour le monde
Bonjour le monde

$ echo -n Bonjour
Bonjour$ echo le monde
le monde

$ echo -n Bonjour; echo -n " le monde"
Bonjour le monde$
```

La commande `echo`

Pour générer plusieurs lignes plutôt qu'une seule

```
$ echo "abc\ndef\nghi"
abc\ndef\nghi

$ echo -e "abc\ndef\nghi"
abc
def
ghi

$ echo -e "abc\ndef\nghi\n"
abc
def
ghi

$
```

La commande `printf`

Exécution sur `java.labunix` avec `LANG=fr_CA.utf8`

Rôle = Mettre en forme des données et les émettre sur `stdout`

```
$ printf "| %d | %8.2f | %6s |\n" 999 12,3 fin
| 999 |    12,30 |    fin |

$ printf "| %d | %-8.2f | %-6s |\n" 999 12,3 fin
??
```

Donc : Les spécifications de format sont semblables à celles de C. . .

La commande `printf`

Exécution sur `java.labunix` avec `LANG=fr_CA.utf8`

Rôle = Mettre en forme des données et les émettre sur `stdout`

```
$ printf "| %d | %8.2f | %6s |\n" 999 12,3 fin
| 999 |    12,30 |    fin |

$ printf "| %d | %-8.2f | %-6s |\n" 999 12,3 fin
| 999 | 12,30    | fin    |
```

Donc : Les spécifications de format sont semblables à celles de C. . .

La commande `printf`

Mais il y a quelques différences avec le `printf` de C !


```
$ printf "%s: %d\n" abc 10 def 30 ghi 40
abc: 10
def: 30
ghi: 40


$ printf "%s => %s; " 10 20 30
??
```

Donc : Lorsqu'il y a plus d'arguments que de «spécificateurs» dans le format, le `printf` est réexécuté avec les arguments subséquents jusqu'à ce tous les arguments soient traités.

La commande `printf`

Mais il y a quelques différences avec le `printf` de C !


```
$ printf "%s: %d\n" abc 10 def 30 ghi 40
abc: 10
def: 30
ghi: 40


$ printf "%s => %s; " 10 20 30
10 => 20; 30 => ; $
```

Donc : Lorsqu'il y a plus d'arguments que de «spécificateurs» dans le format, le `printf` est réexécuté avec les arguments subséquents jusqu'à ce tous les arguments soient traités.

La commande `printf`

Mais il y a quelques différences avec le `printf` de C !


```
$ printf "%s\n" "abc def"
abc def

$ printf "%q\n" "abc def"
abc\ def


$ printf "%s\n" abc\ def
abc def

$ printf "%q\n" abc\ def
abc\ def
```

Donc : Le format «`%q`» utilisé avec une chaine imprime l'argument pour qu'il soit utilisable comme entrée du *shell* — le «`q`» est pour «`quote`»

La commande `tee`

Rôle = Copier l'entrée standard sur la sortie standard **et** dans un fichier

```
$ cat f1.txt
abc def
ghi

$ cat f2.txt
cat: f2.txt: Aucun fichier ou dossier de ce type

$ cat f1.txt | tee f2.txt
abc def
ghi

$ cat f2.txt
abc def
ghi
```

Note : Utile pour voir la sortie d'un programme au fur et à mesure où elle est produite tout en la conservant dans un fichier pour utilisation ultérieure.

2.2 Comparaison de fichiers

La commande `cmp`

Pour déterminer «rapidement» si deux fichiers sont égaux ou pas

Rôle = Comparer deux fichiers octet par octet

```
$ cmp f1.txt f1.txt
$ echo $?
0

$ cmp f2.txt f3.txt
f2.txt f3.txt sont différents: octet 1, ligne 1
$ echo $?
1

$ cmp --quiet f2.txt f3.txt
$ echo $?
1
```

Note : La variable «`$?`» contient le statut retourné lors de l'exécution de la dernière commande (*exit status*) :

- $=$ 0 : Tout est ok !
- $\neq$ 0 : Erreur ou indicateur pour condition spéciale

La commande `diff`

Pour comparer de façon plus détaillée des fichiers de texte

Rôle = Trouver les différences entre deux fichiers

```
$ cat f0.txt
bc
def
ghi
jkl

$ cat f1.txt
abc
def
ghi
```

```
$ diff f0.txt f1.txt
1c1
< bc
---
> abc
4d3
< jkl
```

La commande `diff`

Pour comparer de façon plus détaillée des fichiers de texte

Rôle = Trouver les différences entre deux fichiers

```
$ cat f0.txt
bc
def
ghi
jkl

$ cat f1.txt
abc
def
ghi
```

```
$ diff f1.txt f0.txt
1c1
< abc
---
> bc
3a4
> jkl
```

Détails des changements indiqués par `diff`

Pas souvent nécessaire d'analyser cela en détail !

'lar' *Add the lines in range r of the second file after line l of the first file.*

For example, '8a12,15' means append lines 12–15 of file 2 after line 8 of file 1 ; or, if changing file 2 into file 1, delete lines 12–15 of file 2.

'fct' *Replace the lines in range f of the first file with lines in range t of the second file. This is like a combined add and delete, but more compact.*

For example, '5,7c8,10' means change lines 5–7 of file 1 to read as lines 8–10 of file 2 ; or, if changing file 2 into file 1, change lines 8–10 of file 2 to read as lines 5–7 of file 1.

'rdl' *Delete the lines in range r from the first file ; line l is where they would have appeared in the second file had they not been deleted.*

For example, '5,7d3' means delete lines 5–7 of file 1 ; or, if changing file 2 into file 1, append lines 5–7 of file 1 after line 3 of file 2.

Source: `http://www.chemie.fu-berlin.de/chemnet/use/info/diff/diff_3.html`

La commande `diff`

Pour comparer des fichiers de texte avec un contexte


```
$ cat f0.txt
bc
def
ghi
jkl

$ cat f1.txt
abc
def
ghi
```

Légende :

- `!` Changement
- `+` Ajout
- `-` Suppression

```
$ diff -c f0.txt f1.txt
*** f0.txt  2016-01-12 15:31:18.480792781 +
--- f1.txt  2016-01-12 15:31:28.568792962 +
***************
*** 1,4 ****
! bc
  def
  ghi
- jkl
--- 1,3 ----
! abc
  def
  ghi
```

La commande `diff`

Pour comparer des fichiers de texte avec un contexte


```
$ cat f0.txt
bc
def
ghi
jkl

$ cat f1.txt
abc
def
ghi
```

Légende :

- `!` Changement
- `+` Ajout
- `-` Suppression

```
$ diff -c f1.txt f0.txt
*** f1.txt  2016-01-12 15:31:28.568792962 +
--- f0.txt  2016-01-12 15:31:18.480792781 +
***************
*** 1,3 ****
! abc
  def
  ghi
--- 1,4 ----
! bc
  def
  ghi
+ jkl
```

La commande `diff`

Pour ignorer certains aspects lors des comparaisons


```
$ cat fich0.txt
abc def ghi
xx yy

z

$ cat fich1.txt
a  BCDEF ghi
x x yy
Z
```

```
$ diff fich0.txt fich1.txt
1,4c1,3
< abc def ghi
< xx yy
<
< z
---
> a  BCDEF ghi
> x x yy
> Z
```

La commande `diff`

Pour ignorer la casse (minuscule vs. MAJUSCULE)


```
$ cat fich0.txt
abc def ghi
xx yy

z

$ cat fich1.txt
a  BCDEF ghi
x x yy
Z
```

```
$ diff -i fich0.txt fich1.txt
1,3c1,2
< abc def ghi
< xx yy
<
---
> a  BCDEF ghi
> x x yy
```

La commande `diff`

Pour ignorer «les espaces blanches lors de la comparaison de lignes»


```
$ cat fich0.txt
abc def ghi
xx yy

z

$ cat fich1.txt
a  BCDEF ghi
x x yy
Z
```

```
$ diff -iw fich0.txt fich1.txt
3d2
<
```

La commande `diff`

Pour ignorer ??


```
$ cat fich0.txt
abc def ghi
xx yy

z

$ cat fich1.txt
a  BCDEF ghi
x x yy
Z
```

```
$ diff -iwB fich0.txt fich1.txt
??
```

La commande `diff`

Pour ignorer la casse, les espaces blanches et les lignes blanches


```
$ cat fich0.txt
abc def ghi
xx yy

z

$ cat fich1.txt
a  BCDEF ghi
x x yy
Z
```

```
$ diff -iwB fich0.txt fich1.txt
```

La commande `sdiff`

Pour comparer les deux fichiers côtes à côtes


```
$ cat f0.txt
bc
def
ghi
jkl
```

```
$ cat f1.txt
abc
def
ghi
```

```
$ sdiff f0.txt  f1.txt
bc                                  | abc
def                                   def
ghi                                   ghi
jkl                                 <
```

2.3 Tri de fichiers

La commande `sort`

Tri alphanumérique

Rôle = Trier les lignes d'un fichier

```
$ cat ff.txt
ab cd
DEF
DEF
xx
abc
XX
xx
```

Note :

`sort -u f`

=

`sort f | uniq`

Avec ou sans doublons

```
$ sort ff.txt
abc
ab cd
DEF
DEF
XX
xx
xx

$ sort -u ff.txt
abc
ab cd
DEF
XX
xx
```

La commande `sort`

Tri numérique


```
$ cat ff0.txt
123345
89
123
```

Note : L'option «`-r`» peut aussi s'employer pour un tri alphanumérique.

Tri alphanumérique en ordre, en ordre inverse vs. numérique

```
$ sort ff0.txt
123
123345
89

$ sort -n ff0.txt
89
123
123345

$ sort -nr ff0.txt
123345
123
89
```

3. Expressions régulières

Il y a deux sortes d'expressions régulières

Certains outils utilisent une sorte, d'autres outils utilisent l'autre sorte

- Les expressions régulières simples (*basic reg. ex.*) :

```
\   .   *   ^   $   [...]   [^...]

\{n\}   \{n,\}   \{n,m\}
\(...\)   \1   \2 ... \9
```

- Les expressions régulières étendues (*extended reg. ex.*) :

```
\   .   *   ^   $   [...]   [^...]

{n}   {n,}   {n,m}
(...)   +   ?   |
```

Note : `n` et `m` sont des entiers non négatifs (≥ 0)

Il y a deux sortes d'expressions régulières

Certains outils utilisent une sorte, d'autres outils utilisent l'autre sorte

Expr. rég. simples	**Expr. rég. étendues**
`grep`	`grep -E`
`sed`	`sed -E`
	`awk`
	`[[ chaine =~` *motif* `]]`

Note : `grep -E` (moderne) == `egrep` (ancien)

Ne pas confondre «expressions régulières» et «*file globbing*»

L'expansion des noms de fichiers (*file globbing*) utilise certains caractères spéciaux, caractères utilisés aussi comme patrons dans les instructions `case`, **différents de ceux des expr. rég.** :

Caractère spécial	**Signification pour le *file globbing***
`?`	1 caractère arbitraire (mais pas la fin de ligne)
`*`	0, 1 ou plusieurs caractères arbitraires
`[...]`	un caractère dans l'ensemble indiqué
`[!...]`	un caractère pas dans l'ensemble indiqué
`{ch1,ch2,...,chk}`	un ensemble de chaines possibles

3.1 Expressions régulières simples

Les expressions régulières simples

Ces expr. rég. simples sont aussi des expr. rég. étendues

	Signification dans un patron
`\`	Supprime la signification du caractère spécial qui suit
`.`	Matche n'importe quel caractère
`*`	Matche 0, 1 ou plusieurs fois l'expression qui précède
`^`	Matche le début de ligne (ancre = *anchor*)
`$`	Matche la fin de ligne (ancre = *anchor*)
`[...]`	Matche l'un des caractères de la classe (ensemble de caractères)
`[^...]`	Matche si n'est pas un des caractères de la classe (ensemble complément)

Les expressions régulières simples

Ces expr. rég. simples ne font pas partie des expr. rég. étendues

	Signification dans un patron
`\{n\}`	Matche exactement `n` occurrences de l'expression qui précède
`\{n,\}`	Matche `n` occurrences ou plus de l'expression qui précède
`\{n,m\}`	Matche entre `n` et `m` occurrences de l'expression qui précède
`\(...\)`	Capture l'élément matché — définit un groupe avec une *backreference* appropriée
`\1`	Matche le 1er groupe matché (*backreference*)
`\2`	Matche le 2^{e} groupe matché
...	...
`\9`	Matche le 9^{e} groupe matché

Les expressions régulières simples

Ces expr. rég. simples sont aussi des expr. rég. étendues

	Signification dans un patron
`[:alpha:]`	Matche une lettre
`[:digit:]`	Matche un chiffre
`[:alnum:]`	Matche une lettre ou un chiffre
`[:punct:]`	Matche un caractère de ponctuation
`[:blank:]`	Matche un blanc (espace ou tab. = `[ \t]`)
`[:space:]`	Matche une espace — (= `[ \t\n\r\f\v]`)
`[:lower:]`	Matche une lettre minuscule
`[:upper:]`	Matche une lettre majuscule

Note : Doivent être utilisés dans une spécification de classe, par ex. :

```
$ echo foo | grep -o "[[:alpha:]]*"
foo
```

Note : Peuvent aussi être combinés, donc :
`[[:alpha:][:digit:]] = [[:alnum:]]`

Remarques sur la notation utilisée pour les exemples qui suivent (expr. rég. simples et étendues) :

- Ces expressions, telles que formulées, ne peuvent pas être évaluées par le *shell*.
  Cette notation est utilisée simplement pour alléger la présentation des exemples.
- L'opérateur «=~» dénote une opération de *pattern matching*, comme on la retrouve en Perl, Ruby, etc.
  L'opérande de gauche est le patron alors que l'opérande de droite est la chaine à matcher.
- La valeur indiquée est le code de statut retourné par `grep` (expr. rég. simple) ou `egrep` (expr. rég. étendue) :
  - `0` ⇒ Un match a été trouvé !
  - `1` ⇒ Aucun match n'a été trouvé !

Avec ou sans ancrage

```
$ echo aaadef | grep -o def
def

$ echo aaadef | grep -o ^def

$ echo aaadef | grep -o def$
def
```

Note : `grep -o` vérifie si les lignes reçues *matchent* une expr. rég. et, si oui, émet toutes les parties qui produisent un *match* !

Note : `grep -o = grep --only-matching`

Les expressions régulières simples

Quelques exemples

Avec classes de caractères

```
$ echo abc | grep -o [b-d]
b
c

$ echo abc | grep -o [0-9]

$ echo abc | grep -o [^b-d]
a

$ echo abc | grep -o [^0-9]
a
b
c
```

Les expressions régulières simples

Quelques exemples

Avec 0, 1 ou plusieurs... ??

```
$ echo aaadef | grep -o ".*"; echo $?
??

$ echo aaadef | grep -o "a*"; echo $?
??

$ echo aaadef | grep -o "x*"; echo $?
??

$ echo aaadef | grep -o "x*" | grep "^$" | wc -lc
??
```

Note : `echo $?` retourne le statut retourné par la commande précédente.
Pour `grep` : 0 = Un match a été trouvé ; 1 = Aucun match.

Les expressions régulières simples

Quelques exemples

Avec 0, 1 ou plusieurs... est parfois *tricky* — Mac OS X !

```
$ echo aaadef | grep -o ".*"; echo $?
aaadef
0

$ echo aaadef | grep -o "a*"; echo $?
aaa
0

$ echo aaadef | grep -o "x*"; echo $?

0

$ echo aaadef | grep -o "x*" | grep "^$" | wc -lc
       1       1
```

Note : `echo $?` retourne le statut retourné par la commande précédente.
Pour `grep` : 0 = Un match a été trouvé ; 1 = Aucun match.

Les expressions régulières simples

Quelques exemples

Avec 0, 1 ou plusieurs... est parfois *tricky* — CentOS !

```
$ echo aaadef | grep -o ".*"; echo $?
aaadef
0

$ echo aaadef | grep -o "a*"; echo $?
aaa
0

$ echo aaadef | grep -o "x*"; echo $?
0

$ echo aaadef | grep -o "x*" | grep "^$" | wc -lc
       0       0
```

Note : `echo $?` retourne le statut retourné par la commande précédente.
Pour `grep` : 0 = Un match a été trouvé ; 1 = Aucun match.

Le saut de ligne n'est pas matché par `grep`

By default, a pattern matches an input line if the regular expression (RE) in the pattern matches the input line without its trailing newline. An empty expression matches every line. Each input line that matches at least one of the patterns is written to the standard output.

Extrait de «`man grep`»

Les expressions régulières simples

Quelques exemples

Avec *back references*

```
$ echo aaadef | grep -o "\(a\)\1\1"
aaa

$ echo ' aa--aa--aa!' | grep -o "\(a\{2\}\).*\1.*\1"
aa--aa--aa
```

Les expressions régulières simples

Quelques exemples

Le symbole «^» n'est une ancre qu'en début du patron

```
$ echo "abc^def" | grep -o "^def"; echo $?
1

$ echo "abc^def" | grep -o ".^def"
c^def

$ echo "abc^def" | grep -o "\^def"
^def
```

Le symbole «$» n'est une ancre qu'en fin du patron

```
$ echo 'abc$def' | grep -o "abc$"; echo $?
??

$ echo 'abc$def' | grep -o "abc$."; echo $?
??

$ echo "abc$def" | grep -o "abc$"; echo $?
??
```

Note : Avec les guillemets simples (apostrophes), les caractères spéciaux au niveau du *shell* sont ignorés, donc pas d'expansion (interpolation) des variables.
Avec les guillemets doubles (guillemets), on procède à l'expansion des variables. Dans l'exemple, on fait donc l'expansion de la variable `def`, qui n'est pas définie, donc sa valeur (par défaut) est la chaine vide !

Les expressions régulières simples

Quelques exemples

Le symbole «$» n'est une ancre qu'en fin du patron

```
$ echo 'abc$def' | grep -o "abc$"; echo $?
1

$ echo 'abc$def' | grep -o "abc$."; echo $?
abc$d
0

$ echo "abc$def" | grep -o "abc$"; echo $?
abc
0
```

Note : Avec les guillemets simples (apostrophes), les caractères spéciaux au niveau du *shell* sont ignorés, donc pas d'expansion (interpolation) des variables.
Avec les guillemets doubles (guillemets), on procède à l'expansion des variables. Dans l'exemple, on fait donc l'expansion de la variable `def`, qui n'est pas définie, donc sa valeur (par défaut) est la chaine vide !

3.2 Expressions régulières étendues

Les expressions régulières étendues

Autres expressions possibles, en plus de certaines des expressions simples vues précédemment

	Signification dans un patron
`?`	Matche 0 ou 1 occurrence
`+`	Matche une ou plusieurs occurrences
`{n}`	Matche `n` occurrences
`{n,}`	Matche `n` occurrences ou +
`{n,m}`	Matche entre `n` et `m` occurrences
`|`	Matche l'expression gauche ou droite (choix)
`(...)`	Groupe une expression pour changer la priorité des opérateurs — mais sans *backreference* avec `awk` !

On inclue aussi les symboles des expr. rég. simples suivants :

`.`, `*`, `^`, `$`, `[...]`, `[^...]`

Avec une ou plusieurs occurrences

```
$ echo 123 | grep -E -o [[:digit:]]+
123

$ echo 123 | grep -E -o [[:alnum:]]+
123

$ echo foo1 | grep -E -o [[:alnum:]]+
foo1
```

Les expressions régulières étendues

Quelques exemples

Avec choix (alternation), l'opérateur de priorité la plus faible

```
$ echo abcx | grep -E -o "^abc|def$"
??

$ echo xdef | grep -E -o "^abc|def$"
??

$ echo abcx | grep -E -o "^(abc|def)$"; echo $?
??

$ echo abc | grep -E -o "^(abc|def)$"; echo $?
??
```

Les expressions régulières étendues

Quelques exemples

Avec choix (alternation), l'opérateur de priorité la plus faible

```
$ echo abcx | grep -E -o "^abc|def$"
abc

$ echo xdef | grep -E -o "^abc|def$"
def

$ echo abcx | grep -E -o "^(abc|def)$"; echo $?
1

$ echo abc | grep -E -o "^(abc|def)$"; echo $?
abc
0
```

3.3 Remarques sur les expressions régulières simples vs. étendues

Ne pas confondre «expressions régulières» et «*file globbing*» (bis)

Caractère spécial de *file globbing*	**Expr. rég. étendue** *équivalente*
`?`	??
`*`	??
`[...]`	??
`[!...]`	??
`{ch1,ch2,...,chk}`	??

Ne pas confondre «expressions régulières» et «*file globbing*» (bis)

Caractère spécial de *file globbing*	**Expr. rég. étendue** *équivalente*
`?`	`.`
`*`	`.*`
`[...]`	`[...]`
`[!...]`	`[^...]`
`{ch1,ch2,...,chk}`	`ch1|ch2|...|chk`

Les expressions régulières simples et étendues ne sont plus si différentes

Dans plusieurs mises en œuvre modernes des expressions régulières simples, les caractères «+», «?» et «|» sont interprétés comme dans les expressions étendues s'ils sont précédés d'un «\».

Exemple

```
$ echo xxaaayy | grep -o "a+b?"
$ echo xxaaayy | grep -o "a\+b\?"
aaa
```

Extrait de `man re_format` (sur Mac OS X)

For basic REs, '+', '?' and '|' remain regular characters, but '\+', '\ ?' and '\|' have the same special meaning as the unescaped characters do for extended REs, i.e., one or more matches, zero or one matches and alteration, respectively.

[. . .]

BUGS

Having two kinds of REs is a botch.

4. Traitement de fichiers de texte

4.1 Analyse et recherche

La commande `wc`

Rôle = Compter le nombre de caractères, mots et/ou lignes d'un fichier texte

```
$ cat fich.txt
xxx yyy zzz
d/e/f
```

Note : Si des noms de fichiers sont indiqués, ces noms sont émis en sortie, mais pas si on traite `stdin`

(sauf si spécifié avec «–»).

```
$ wc -l fich.txt
2 fich.txt

$ wc -w fich.txt
4 fich.txt

$ wc -c fich.txt
??

$ wc fich.txt
 2   4 ?? fich.txt

$ cat fich.txt | wc
      2       4      ??
```

La commande `wc`

Rôle = Compter le nombre de caractères, mots et/ou lignes d'un fichier texte

```
$ cat fich.txt
xxx yyy zzz
d/e/f
```

Note : Si des noms de fichiers sont indiqués, ces noms sont émis en sortie, mais pas si on traite `stdin`

(sauf si spécifié avec «–»).

```
$ wc -l fich.txt
2 fich.txt

$ wc -w fich.txt
4 fich.txt

$ wc -c fich.txt
18 fich.txt

$ wc fich.txt
 2  4 18 fich.txt

$ cat fich.txt | wc
      2       4      18
```

La commande `grep`

`grep` = *global regular expression print*

`man grep`

SYNOPSIS

```
grep [options] motif [fichier...]
```

DESCRIPTION

`grep` recherche dans les `fichier`s indiqués (ou depuis l'entrée standard si aucun fichier n'est fourni, ou si un simple trait d'union - est fourni en tant que nom de fichier) les lignes correspondant à un certain `motif`. Par défaut, `grep` affiche les lignes qui correspondent au `motif`.

La commande `grep`

Les principales options

Option	Signification
`-E`	Utilise une expression régulière étendue
`-i`	Ignore la casse (minuscule/MAJUSCULE)
`-l`	Émet seulement le nom du fichier s'il y a un match, pas les lignes matchées
`-q`	Retourne uniquement un code d'erreur, sans émettre les lignes matchées
`-v`	Émet les lignes qui ne matchent pas

La commande `grep`

Quelques exemples


```
$ cat f0.txt
bc
def
ghi
jkl

$ cat f1.txt
abc
def
ghi

$ grep b f0.txt
bc

$ grep b f1.txt
abc
```

```
$ grep b f0.txt f1.txt
f0.txt:bc
f1.txt:abc

$ grep -v b f0.txt f1.txt
f0.txt:def
f0.txt:ghi
f0.txt:jkl
f1.txt:def
f1.txt:ghi

$ grep -l b f0.txt f1.txt
f0.txt
f1.txt
```

Note : Si plusieurs fichiers sont indiqués, alors les noms sont émis.

La commande `grep`
Quelques exemples


```
$ cat f0.txt
bc
def
ghi
jkl

$ cat f1.txt
abc
def
ghi
```

Avec code de statut seulement

```
$ grep -q b f0.txt f1.txt
$ echo $?
0

$ grep -q bfdfdx f0.txt f1.txt
$ echo $?
1
```

La commande `grep`

Quelques autres options

Option	Signification
`-F`	Utilise une chaine fixe
`-h`	N'indique pas le nom du fichier, même quand plusieurs fichiers sont traités
`-n`	Préfixe chaque ligne matchée par son numéro de ligne
`-o`	Affiche **la partie matchée**, et non pas la ligne au complet
`-r`	Fouille récursivement tous les fichiers e.g., `grep -r motif repertoire`
`-w`	Matche le motif seulement si c'est un mot complet

Note : L'option «`-o`» est utile pour voir (comprendre ! ?) ce qui est vraiment matché. De plus, tous les matchs sont indiqués.

La commande `grep`

Quelques exemples : `grep -F` = `fgrep`


```
$ cat bar.txt
xyz
abc.def
```

Expression régulière vs. chaine fixe

```
$ grep . bar.txt
??

$ grep -F . bar.txt
??
```

Note : «`fgrep`» == «`grep -F`»

La commande `grep`

Quelques exemples : `grep -F` = `fgrep`


```
$ cat bar.txt
xyz
abc.def
```

Expression régulière vs. chaine fixe

```
$ grep . bar.txt
xyz
abc.def

$ grep -F . bar.txt
abc.def
```

Note : «`fgrep`» == «`grep -F`»

La commande «grep -o» avec expr. rég. simple

Quelques exemples

```
$ echo "aadef" | grep -o ".*"; echo $?
??

$ echo "aadef" | grep -o "a*"; echo $?
??

$ echo "aadef" | grep -o "ZZ"; echo $?
??

$ echo "aadef" | grep -o "Z*"; echo $?
??
```

Question : Comment explique-t-on le dernier résultat ?

??

La commande «grep -o» avec expr. rég. simple

Quelques exemples

```
$ echo "aadef" | grep -o ".*"; echo $?
aadef
0

$ echo "aadef" | grep -o "a*"; echo $?
aa
0

$ echo "aadef" | grep -o "ZZ"; echo $?
1

$ echo "aadef" | grep -o "Z*"; echo $?
0
```

Question : Comment explique-t-on le dernier résultat ?
Matche la chaine vide !

La commande «grep -o» avec expr. rég. simple

Quelques exemples

```
echo "123-123-123" | grep -o "^.*-"
```

??

```
echo "123-123-123" | grep -o "^\(.*\)-\1"
```

??

La commande «grep -o» avec expr. rég. simple

Quelques exemples

```
echo "123-123-123" | grep -o "^.*-"
123-123-

echo "123-123-123" | grep -o "^\(.*\)-\1"
123-123
```

La commande «grep -o» avec expr. rég. simple

Quelques exemples

Soit un fichier de type CSV = *Comma-separated values*

$champ_{11}$,$champ_{21}$,$champ_{31}$,...,$champ_{k1}$
$champ_{12}$,$champ_{22}$,$champ_{32}$,...,$champ_{k2}$
...

On veut lister le premier champ de chacune des lignes

```
$ cat <<FIN | grep -o motif
champ%11,c21,c31
c-12,c22,c32
chmp.13,c23,c33
FIN
champ%11
c-12
chmp.13
```

Quel *motif* faudrait-il utiliser ?

La commande «grep -o» avec expr. rég. simple

Quelques exemples

Soit un fichier de type CSV = *Comma-separated values*

champ$_{11}$,champ$_{21}$,champ$_{31}$,...,champ$_{k1}$
champ$_{12}$,champ$_{22}$,champ$_{32}$,...,champ$_{k2}$
...

On veut lister le premier champ de chacune des lignes

```
$ cat <<FIN | grep -o motif
champ%11,c21,c31
c-12,c22,c32
chmp.13,c23,c33
FIN
champ%11
c-12
chmp.13
```

Quel *motif* faudrait-il utiliser ?

Réponse : *motif* = `"^[^,]*"`

4.2 Transformation et substitution

La commande `tr`
Transposer ou éliminer des caractères

`man tr` (description simplifiée)

SYNOPSIS

`tr [options]` *`cars-source cars-de-remplacement`*

DESCRIPTION

`tr` copie son entrée standard sur sa sortie standard en effectuant l'une des manipulations suivantes :

- transpose, et éventuellement réunit les caractères dupliqués de la chaîne résultante
- réunit les caractères dupliqués
- supprime des caractères
- supprime des caractères, et éventuellement réunit les caractères dupliqués de la chaîne résultante

La commande `tr`

Les principales options

Option	**Signification**
`-d`	Supprime les caractères de *`cars-source`*
`-s`	Élimine les répétitions de caractères pour ne conserver qu'une seule occurrence

La commande `tr`

Quelques exemples

Transpositions simples

```
$ echo abcdef | tr abc DEF
DEFdef

$ echo abcdef | tr a-z A-Z
ABCDEF

$ echo '(.)(.)()()(())' | tr '()' '<>'
<.><.><><><<>>


$ echo abcDEF | tr '[:lower:]' '[:upper:]'
ABCDEF
```

Transpositions simples avec ensembles de tailles différentes

```
$ echo abcdef | tr bcde DE
aDEEEf

$ echo abcdef | tr a A-Z
Abcdef
```

La commande `tr`

Quelques exemples

Suppression et compaction de caractères

```
$ echo 'un  mot     et un autre' | tr -d ' '
unmotetunautre

$ echo 'un  mot     et un autre' | tr -s ' '
un mot et un autre

$ cat f1.txt
abc

def
ghi

$ cat f1.txt | tr -s '\n' '.'
??
```

La commande `tr`

Quelques exemples

Suppression et compaction de caractères

```
$ echo 'un  mot     et un autre' | tr -d ' '
unmotetunautre

$ echo 'un  mot     et un autre' | tr -s ' '
un mot et un autre

$ cat f1.txt
abc

def
ghi

$ cat f1.txt | tr -s '\n' '.'
abc.def.ghi.$
```

La commande `sed`

`sed` = *stream editor* = éditeur de flux pour le filtrage et la transformation de texte

`man sed`

SYNOPSIS

```
sed [options]... {script-seulement-si-pas-d-autre-script
                 [fichier-d-entrée]...
```

DESCRIPTION

`sed` est un éditeur de flux. Un éditeur de flux est utilisé pour effectuer des transformations de texte basiques sur un flux d'entrée (un fichier ou l'entrée d'un tube). Alors que d'une certaine manière il est similaire à un éditeur qui permet des éditions scriptées (comme `ed`), `sed` fonctionne en seulement une passe sur l'entrée(s) et est, par conséquent, plus efficace. Mais c'est sa capacité à filtrer du texte dans un tube qui le distingue des autres éditeurs.

Note : Dans ce qui suit, nous examinons quelques-unes des (très !) nombreuses expressions d'édition et options.

La commande `sed`

Les principales expressions (scripts) d'édition

`/patron/d`	Supprime la ligne si elle matche *patron*
`/patron/p`	Imprime la ligne si elle matche *patron*
`s/patron/chaine/`	Substitue la première occurrence de *patron* par *chaine*
`s/patron/chaine/g`	Substitue toutes les occurrences de *patron* par *chaine*
`/patron/i\chaine`	Insère une ligne avec *chaine* devant la ligne si elle matche *patron*⋆

Notes :

- Chaque expression d'édition s'applique sur chacune des lignes du flux d'entrée qui matche le motif et la ligne modifiée est émise en sortie sauf évidemment si l'action est `d`.
- Le caractère utilisé pour délimiter la commande, le patron et la chaine peut être n'importe quel caractère répété — pas nécessairement «/»

Option	Signification
`-e script` `--expression=script`	Ajoute le `script` aux commandes à exécuter
`-f fich` `--script-file=fich`	Ajoute le contenu de `fich` aux commandes à exécuter
`-i[EXT]` `--in-place[=EXT]`	Édite le fichier en ligne Crée un fichier de sauvegarde si une `EXT`ension est spécifiée
`-E`	Utilise des expressions régulières étendues plutôt que des expr. rég. simples

La commande sed

Quelques exemples de suppression


```
$ cat fich.txt
1 2 3
xxx yyy zzz
d/e/f
```

Suppression

```
$ sed '/xx/d' fich.txt
1 2 3
d/e/f

$ sed '/a/d' fich.txt
1 2 3
xxx yyy zzz
d/e/f

$ sed 'd' fich.txt
```

La commande sed

Quelques exemples de substitution

```
$ cat fich.txt
1 2 3
xxx yyy zzz
d/e/f
```

Substitutions simples

```
$ sed 's/x/ABC/' fich.txt
1 2 3
ABCxx yyy zzz
d/e/f

$ sed 's/x/ABC/g' fich.txt
1 2 3
ABCABCABC yyy zzz
d/e/f

$ sed 's/\(.\)\1\1/\1/g' fich.txt
??
```

La commande sed

Quelques exemples de substitution

```
$ cat fich.txt
1 2 3
xxx yyy zzz
d/e/f
```

Substitutions simples

```
$ sed 's/x/ABC/' fich.txt
1 2 3
ABCxx yyy zzz
d/e/f

$ sed 's/x/ABC/g' fich.txt
1 2 3
ABCABCABC yyy zzz
d/e/f

$ sed 's/\(.\)\1\1/\1/g' fich.txt
1 2 3
x y z
d/e/f
```

La commande sed

Quelques exemples avec impression explicite


```
$ cat fich.txt
1 2 3
xxx yyy zzz
d/e/f
```

Impression explicite

```
$ sed '/xx/p' fich.txt
1 2 3
xxx yyy zzz
xxx yyy zzz
d/e/f


$ sed '/a/p' fich.txt
1 2 3
xxx yyy zzz
d/e/f
```

La commande `sed`

Pseudocode **simplifié** (! !), pour comprendre le fonctionnement de base de `sed`

```
while !STDIN.eof? do
   ligne ← STDIN.readline
   emettre_ligne ← true  # On l'emet... sauf si action = 'd'

   if ligne matche le motif then
     case action
       when 'd':
         emettre_ligne ← false
       when 'p':
         STDOUT.print ligne
       when 's':
         modifier ligne selon la substitution indiquée
     end
   end

   STDOUT.print ligne if emettre_ligne
end
```

```
$ cat fich.txt
xxx yyy zzz
d/e/f
```

Substitutions simples

```
$ sed 's/x/abc/' fich.txt
abcxx yyy zzz
d/e/f

$ sed 's/x/abc/g' fich.txt
abcabcabc yyy zzz
d/e/f

$ sed 's/\(.\)\1\1/\1/g' fich.txt
x y z
d/e/f
```

```
$ cat fich.txt
xxx yyy zzz
d/e/f
```

Note : Dans plusieurs cas, les guillemets ne sont pas nécessaires, mais il est généralement plus simple et plus sûr de les utiliser...

Quelques exemples, avec/sans guillemets

```
$ sed s/x/abc/g fich.txt
abcabcabc yyy zzz
d/e/f

$ sed 's/\(.\)\1\1/\1/g' fich.txt
x y z
d/e/f

$ sed s/\(.\)\1\1/\1/g fich.txt
xxx yyy zzz
d/e/f
```

Sans `'...'`, la dernière commande est interprétée comme suit ☹ :

```
sed 's/(.)11/1/g' fich.txt
```

La commande sed

Quelques exemples de substitution avec des délimiteurs autres que «/»


```
$ cat fich.txt
xxx yyy zzz
d/e/f
```

Substitutions avec d'autres délimiteurs

```
$ sed 's/\/e\//XX/' fich.txt
??

$ sed 's;/e/;XX;' fich.txt
??

$ sed 's;/e/;;' fich.txt
??
```

La commande sed

Quelques exemples de substitution avec des délimiteurs autres que «/»

```
$ cat fich.txt
xxx yyy zzz
d/e/f
```

Substitutions avec d'autres délimiteurs

```
$ sed 's/\/e\//XX/' fich.txt
xxx yyy zzz
dXXf

$ sed 's;/e/;XX;' fich.txt
xxx yyy zzz
dXXf

$ sed 's;/e/;;' fich.txt
xxx yyy zzz
df
```

La commande sed

Quelques exemples de substitution avec le patron spécial «&»


```
$ cat fich.txt
1 2 3
xxx yyy zzz
d/e/f
```

Substitution avec l'item matché

```
$ sed 's/x/|&|/g' fich.txt
1 2 3
|x||x||x| yyy zzz
d/e/f

$ sed 's/[0-9]/&+&/g' fich.txt
1+1 2+2 3+3
xxx yyy zzz
d/e/f
```

```
$ cat fich.txt
xxx yyy zzz
d/e/f
```

Impression explicite, généralement utilisée avec l'option «-n»

```
$ sed '/xx/p' fich.txt
xxx yyy zzz
xxx yyy zzz
d/e/f

$ sed -n '/xx/p' fich.txt
xxx yyy zzz
```

La commande sed

Un exemple d'édition «en ligne»


```
$ cat fich.txt
xxx yyy zzz
d/e/f
```

Modification directe du fichier et *backup*

```
$ ls fich*
fich.txt

$ sed -i.bak 's/xxx/123/' fich.txt

$ ls fich*
fich.txt  fich.txt.bak

$ cat fich.txt
123 yyy zzz
d/e/f

$ cat fich.txt.bak
xxx yyy zzz
d/e/f
```

```
$ cat fich.txt
xxx yyy zzz
d/e/f
```

Avec plusieurs commandes/expressions

```
$ sed -e 's/x/A/g' -e 's/A/W/g' fich.txt
WWW yyy zzz
d/e/f


$ cat actions.txt
s/^xxx \(.*\) \(.*\)$/\2:\1/g
s;/\(e\)/;\1;

$ sed -f actions.txt fich.txt
zzz:yyy
def
```

4.3 Découpage et fusion de champs

La commande `cut`

Rôle = Découper des lignes en une série de champs et émettre certains champs.

Forme la plus souvent utilisée

```
cut [-d car_separateur] -f liste_de_champs [fichier...]
```

- Chaque ligne est découpée en champs, en fonction du séparateur (caractère unique) spécifié.
- Les champs spécifiés dans `liste_de_champs` sont émis, séparés par `car_separateur`.

- Note : Si `car_separateur` est omis, alors c'est le caractères de tabulation qui est utilisé.

La commande `cut`

Exemples

```
$ echo "abc,def,ghi,jkl" | cut -d, -f2
def

$ echo "abc,def,ghi,jkl" | cut -d, -f3
ghi

$ echo "abc def ghi jkl" | cut -d' ' -f2,4
def jkl
```

La commande `cut`

Autres exemples

```
$ echo "a,b,c,d" | cut -d, -f2,3
b,c

$ echo "a,b,c,d" | cut -d, -f3,2
b,c


$ echo "a  b  c   d" | cut -d' ' -f1,4
a

$ echo "a  b  c   d" | tr -s ' ' | cut -d' ' -f1,4
a d

$ echo "a:::b"  | cut -d: -f1,4
a:b
```

La commande `cut`

Contenu du fichier `/etc/passwd`

```
$ cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
...
tcpdump:x:72:72::/:/sbin/nologin
tremblay_gu:x:1000:1000:tremblay_gu:/home/tremblay_gu:/bin/bas
```

Pour obtenir la liste, triée, des usagers

```
$ cut -d: -f1 /etc/passwd | sort
abrt
adm
...
tremblay_gu
tss
unbound
usbmuxd
```

La commande `paste`

Rôle = Regrouper les lignes de différents fichiers

```
$ cat prenoms.txt
Guy
Nellie
Anne-Marie

$ cat noms.txt
Tremblay
Durocher
David
```

Regroupement de deux fichiers

```
$ paste prenoms.txt noms.txt
Guy         Tremblay
Nellie      Durocher
Anne-Marie  David

$ paste -s prenoms.txt noms.txt
Guy         Nellie      Anne-Marie
Tremblay    Durocher    David
```

5. Recherche de fichiers

La commande `type`

Rôle = Trouver le «type» d'une commande

```
$ type javac
javac est /usr/bin/javac

$ type type
type est une primitive du shell

$ type ls
ls est un alias vers <<ls -CF>>

$ type +materiel
+materiel est une fonction
+materiel ()
{
    dest=${1:-$(coursParDefaut)};
    pushd+ ~/$dest/Materiel
}
```

La commande `type`

Rôle = Trouver tous les types d'une commande

```
$ type ls
ls est un alias vers << ls -CF >>

$ type --all ls
ls est un alias vers << ls -CF >>
ls est /usr/bin/ls
ls est /bin/ls
```

Si «foo» est un alias, alors «\foo ignore l'alias


```
$ ls
f1.txt  f2.txt

$ type rm
rm est un alias vers << rm -i >>

$ rm f1.txt
rm : supprimer fichier << f1.txt >> ? oui

$ ls
f2.txt

$ \rm f2.txt

$ ls

$
```

La commande `which`

Rôle = Trouver le chemin complet d'une commande

```
$ which javac
/usr/bin/javac

$ which type
which: no type in
(...:/usr/local/bin)

$ which ls
/bin/ls

$ which +materiel
which: no +materiel in
(...:/usr/local/bin)
```

On veut trouver tous les fichiers avec une extension « `.bak` », y compris dans les sous-répertoires.

Comment peut-on faire avec ce qu'on a vu ?

Trouver tous les fichiers avec une extension `.bak`

Solution avec `tree` ☹

```
$ tree .
.
|-- bar.bak
|-- baz.kabak
`-- Projet1
    `-- foo.bak

1 directory, 3 files
```

Trouver tous les fichiers avec une extension `.bak`

Solution avec `ls -1`/`ls -R` ☹

```
$ ls -1 *.bak
bar.bak

$ ls -1 Projet1/*.bak
Projet1/foo.bak

$ ls -R -1 *.bak
bar.bak
```

Trouver tous les fichiers avec une extension `.bak`

Solution avec `ls -1`/`ls -R` et `grep` ☹

```
$ ls -R -1 | grep .bak
bar.bak
baz.kabak
foo.bak

$ ls -R -1 | grep \.bak     # \. est comme . ☹
bar.bak
baz.kabak
foo.bak

$ ls -R -1 | grep [.]bak
bar.bak
foo.bak

$
```

Sauf que...

Trouver tous les fichiers avec une extension `.bak`

Solution avec `ls -1`/`ls -R` et `grep` ☹

```
$ ls -R -1 | grep .bak
bar.bak
baz.kabak
foo.bak

$ ls -R -1 | grep \.bak     # \. est comme . ☹
bar.bak
baz.kabak
foo.bak

$ ls -R -1 | grep [.]bak
bar.bak
foo.bak

$
```

Sauf que... on ne sait pas que `foo.bak` provient du sous-répertoire `Projet1` ☹

La commande `find`

Rôle = Rechercher des fichiers dans des répertoires

`man find`

SYNOPSIS

```
find [chemin...] [expression]
```

DESCRIPTION

`find` parcourt les arborescences de répertoires commençant en chacun des chemins mentionnés, en évaluant les expressions fournies pour chaque fichier rencontré.

[...]

`find` applique l'action `-print` par défaut sur tous les fichiers pour lesquels l'expression est vraie.

La commande `find`

Les principales options

Option	Test ou action
`-name motif`	Si le nom de base (sans les répertoires du chemin) correspond au `motif`...
`-print`	Émet le nom du fichier sur la sortie standard (action par défaut)
`-type t`	Si le fichier est du type indiqué par `t` : `d`=répertoire ; `f`=fichier régulier ; `l`=lien symbolique, etc.
`-empty`	Si le fichier est vide...
`-path motif`	Si le nom complet correspond au `motif`...
`-user utilisateur`	Si le fichier appartient à `utilisateur`...
...	...

Note : `motif` est une expression de *file globbing*, et non une expr. régulière !

La commande `find`

Quelques exemples

Trouver les répertoires, sauf ceux de `.git` (trop nombreux ☺)

```
$ find -type d | grep -v .git
.
./Divers
./Materiel
./Materiel/Figures
./Materiel/latex
./Syllabus
./Programmes
./Programmes/Bash

$ find Materiel -type d | grep -v .git
Materiel
Materiel/Figures
Materiel/latex
```

La commande `find`

Quelques exemples

Trouver les fichiers avec extension `.sh`

```
$ find . -name ".sh"

$ find . -name "*.sh"
./Programmes/Bash/ex-sed.sh
./Programmes/Bash/regex.sh
```

La commande `find`

Quelques exemples

Trouver tous mes répertoires qui sont des dépôts `git`

```
$ find ~ -name ".git"
/home/tremblay_gu/Attribution-taches-enseignement/.git
/home/tremblay_gu/BiblioRailsRestGli/.git
.
.
.
/home/tremblay_gu/INF600A/.git
/home/tremblay_gu/RapportLatece/.git
```

On veut supprimer tous les fichiers avec une extension « `.bak` », y compris dans les sous-répertoires.

Comment peut-on faire avec ce qu'on a vu ?

Supprimer tous les fichiers avec une extension .bak

Solution avec pipeline et commande directe ☹

```
$ find . -name "*.bak"
./Projet1/foo.bak
./bar.bak

$ find . -name "*.bak" | rm
rm: operande manquant
Saisissez "rm --help" pour plus d'informations.
```

Sauf que...

Supprimer tous les fichiers avec une extension `.bak`

Solution avec pipeline et commande directe ☹

```
$ find . -name "*.bak"
./Projet1/foo.bak
./bar.bak

$ find . -name "*.bak" | rm
rm: operande manquant
Saisissez "rm --help" pour plus d'informations.
```

Sauf que… on ne peut pas exécuter `rm` sur un flux de valeurs — il faut plutôt pouvoir utiliser chacune de ces valeurs comme argument de `rm` ☹

6. Exécution d’une commande sur les éléments d’un flux

La commande `xargs`

Rôle = Construire et exécuter des lignes de commandes à partir de l'entrée standard

`man xargs`

SYNOPSIS

```
xargs [options...] [commande [arguments]]
```

DESCRIPTION

`xargs` lit des arguments délimités par des blancs (pouvant être protégés par des apostrophes, des guillemets ou un backslash) ou par des sauts de ligne depuis l'entrée standard, et exécute une ou plusieurs fois la `commande` (`/bin/echo` par défaut) en utilisant les `arguments` suivis des arguments lus depuis l'entrée standard. Les lignes blanches en entrée sont ignorées.

Note : Commande souvent (!) utilisée avec `find`.

La commande `xargs`

`man xargs` (sur `zeta` : Solaris)

SYNOPSIS

```
xargs [options...] [utility [argument...]]
```

DESCRIPTION

The `xargs` xargs utility constructs a command line consisting of the `utility` and `argument` operands specified followed by as many arguments read in sequence from standard input as will fit in length and number constraints specified by the options. The `xargs` utility then invokes the constructed command line and waits for its completion. This sequence is repeated until an end-of-file condition is detected on standard input or an invocation of a constructed command line returns an exit status of 255.

La commande `xargs` consomme autant d'arguments que requis par la commande qui doit être exécutée


```
$ cat foo.txt
abc
def
123 32131
xxx ddd eee
0000
```

```
$ cat foo.txt | xargs printf '%s\n'
abc
def
123
32131
xxx
ddd
eee
0000
```

La commande `xargs` consomme autant d'arguments que requis par la commande qui doit être exécutée


```
$ cat foo.txt
abc
def
123 32131
xxx ddd eee
0000
```

```
$ cat foo.txt | xargs printf '%s %s\n'
abc def
123 32131
xxx ddd
eee 0000
```

La commande `xargs` consomme autant d'arguments que requis par la commande qui doit être exécutée


```
$ cat foo.txt
abc
def
123 32131
xxx ddd eee
0000
```

```
$ cat foo.txt | xargs printf '%s %s %s\n'
abc def 123
32131 xxx ddd
eee 0000
```

Un pipeline avec `find` et `grep`, avec/sans `xargs`


```
$ ls fich*
fich0.txt fich1.txt

$ cat fich0.txt
abc def ghi
xx yy

$ cat fich1.txt
a  BCDEF ghi
x x yy
```

```
$ find . -name 'fich?.txt' | grep f
??

$ find . -name 'fich?.txt' | xargs grep f
??
```

Un pipeline avec `find` et `grep`, avec/sans `xargs`


```
$ ls fich*
fich0.txt fich1.txt

$ cat fich0.txt
abc def ghi
xx yy

$ cat fich1.txt
a  BCDEF ghi
x x yy
```

```
$ find . -name 'fich?.txt' | grep f
./fich0.txt
./fich1.txt

$ find . -name 'fich?.txt' | xargs grep f
./fich0.txt:abc def ghi
```

Tous les éléments du flux d'entrée sont mis en argument de la commande lorsqu'approprié

Un `grep` sur un seul fichier n'indique pas le nom de fichier

```
$ grep f fich0.txt; grep f fich1.txt
abc def ghi
```

Un `grep` sur plusieurs fichiers ajoute le nom du fichier en préfixe, même s'il n'y a qu'un seul fichier qui matche

```
$ grep f fich0.txt fich1.txt
fich0.txt:abc def ghi

$ find . -name 'fich?.txt' | xargs grep f
./fich0.txt:abc def ghi
```

Suppression d'une série de fichiers


```
$ ls fich?.txt
fich0.txt  fich1.txt

$ find . -name 'fich?.txt' | xargs rm -f

$ ls fich?.txt
ls: impossible d'acceder a fich?.txt:
    Aucun fichier ou dossier de ce type
```

La commande `xargs`

Quelques options

Option	Signification
`-I chaine`	Remplace les occurrences de `chaine` dans les `arguments` par les noms lus depuis l'entrée standard.
`-i`	Équivalent à «`-I {}`».
`--max-lines=k`	Utilise au plus `k` lignes d'entrée pour former les arguments fournis à la commande. Forme courte = `-l k`.

Note : Si l'option «`-i`» ou «`-I`» est utilisée, alors on a implicitement `--max-lines=1`.

Déplacement d'une série de fichiers dans un sous-répertoire

Utilisation de «-I»

```
$ ls Backup

$ find . -name "*.bak"
./f1.bak
./f2.bak
./f3.bak

$ find . -name "*.bak" | xargs -I {} mv {} Backup

$ find . -name "*.bak"
./Backup/f1.bak
./Backup/f2.bak
./Backup/f3.bak
```

Renommage d'une série de fichiers

Utilisation de «-I»

```
$ ls Backup
f1.bak  f2.bak  f3.bak

$ find . -name "*.bak"
./Backup/f1.bak
./Backup/f2.bak
./Backup/f3.bak

$ find . -name "*.bak" | xargs -I F mv F F.old

$ ls Backup
f1.bak.old  f2.bak.old  f3.bak.old
```

La commande `xargs` consomme autant d'arguments que requis par la commande qui doit être exécutée ★130

Mais sans utiliser plus que le nombre de lignes spécifiées par `max-lines`

```
$ cat foo.txt
abc
def
123 32131
xxx ddd eee
0000
```

```
$ cat foo.txt |\
  xargs --max-lines=1 printf '%s %s\n'
abc
def
123 32131
xxx ddd
eee
0000
```

Attention si on traite des noms de fichiers contenant des espaces !

Les noms sont séparés en mots ☹

```
$ ls -1
un   autre    fich.txt
un fich.txt

$ find . -name "*.txt" | xargs ls
ls: ./un: No such file or directory
ls: ./un: No such file or directory
ls: autre: No such file or directory
ls: fich.txt: No such file or directory
ls: fich.txt: No such file or directory

$ find . -name "*.txt" | xargs echo
./un autre fich.txt ./un fich.txt
```

Attention si on traite des noms de fichiers contenant des espaces !

Solution avec spécification d'un délimiteur

```
$ ls -1
un   autre    fich.txt
un fich.txt

$ find . -name "*.txt" | xargs -d '\n' ls
./un   autre    fich.txt  ./un fich.txt

$ find . -name "*.txt" | xargs -d '\n' -I {} ls {}
./un   autre    fich.txt
./un fich.txt

$ find . -name "*.txt" | xargs -d '\n' -I {} echo {}
./un   autre    fich.txt
./un fich.txt
```

Note : Option «`-d`» non reconnue sous Mac OS (BSD). Solution alternative :

```
$ find . -name "*.txt" | tr '\n' '\0' | xargs -0 ls
```

L'option «-exec» de find peut souvent être utilisée à la place de xargs


```
$ cat fich0.txt
abc def ghi
xx yy
```

```
$ cat fich1.txt
a  BCDEF ghi
x x yy
```

```
$ find . -name 'fich?.txt' | xargs grep f
./fich0.txt:abc def ghi

$ find . -name 'fich?.txt' -exec grep f {} \;
abc def ghi

$ find . -name 'fich?.txt' -exec grep f {} +
./fich0.txt:abc def ghi
```

- Avec «\;» : commande exécutée avec un seul fichier
- Avec «+» : commande exécutée avec autant de fichiers que possible

Pour en savoir plus
Quelques références

C. Newham and B. Rosenblatt.
Learning the bash shell.
O'Reilly, 2005.

A. Robbins and N.H.F. Beebe.
Classic Shell Scripting.
O'Reilly, 2009.

`man` *`cmd`*

De nombreux sites Web !