Utilisation du *shell* `bash`

Guy Tremblay
Professeur

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

INF600A
4 septembre 2018

Unix is user-friendly

$\mathcal{O}(n)$

Computer Science (@CompSciFact)

2016-03-08 11:46

'Unix is user-friendly; it's just picky about who its friends are.'

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. Le *shell* `bash`

Qu'est-ce que `bash` ?

> `bash` *is the shell, or command language interpreter, for the GNU operating system. The name is an acronym for the «Bourne-Again SHell», a pun on Stephen Bourne, the author of the direct ancestor of the current Unix shell* `sh`*,*
>
> **Source***: Bash Reference Manual*

Rôle du `shell`

Rôle du *shell*

Le rôle du *shell* est d'exécuter des commandes.

> *[A] shell is simply a macro processor that executes commands. The term macro processor means functionality where text and symbols are expanded to create larger expressions.*
>
> *A Unix shell is both a command interpreter and a programming language.*
>
> **Source**: *Bash Reference Manual*

2. Contexte d’exécution des exemples

Contexte d'exécution des exemples

La machine utilisée = MacBook avec Mac OS X et `bash`

Les exemples qui suivent ont été exécutés (pour la plupart) sous Mac OS X, version 10.9.5, avec `bash` version 3.2.51 :

```
$ uname -a
Darwin localhost 13.4.0 Darwin Kernel Version 13.4.0:\
       Sun Aug 17 19:50:11 PDT 2014;\
       root:xnu-2422.115.4~1/RELEASE_X86_64 x86_64


$ bash --version
GNU bash, version 3.2.51(1)-release (x86_64-apple-darwin13)
Copyright (C) 2007 Free Software Foundation, Inc.
```

Contexte d'exécution des exemples

Un alias est utilisé pour `ls`

Dans les exemples qui suivent, `ls` est en fait un alias :

```
alias ls='ls -F'
```

La documentation de l'option «`-F`» est la suivante :

```
-F Display
   a slash (`/') immediately after each pathname
            that is a directory,
   an asterisk (`*') after each that is executable,
   an at sign (`@') after each symbolic link,
   an equals sign (`=') after each socket,
   a percent sign (`%') after each whiteout,
   and a vertical bar (`|') after each that is a FIFO.
```

Contexte d'exécution des exemples

L'invite (le *prompt*) est défini avec la variable `PS1`

La variable `PS1` détermine le *prompt*

```
<<~/INF600A@MacBook>> $ echo $PS1
<<\w@MacBook>> $

<<~/INF600A@MacBook>> $ export PS1='[\u@\h \W]$ '

[tremblay@localhost INF600A]$ export PS1='$ '

$ ls
AFAIRE              Divers/             Syllabus/
Varia/
ALIRE               Materiel/           TEMPS

$
```

Les exemples qui suivent utilisent parfois «`$ `» (pour ne pas alourdir la présentation), parfois une indication du répertoire courant.

Contexte d'exécution des exemples

Les lignes blanches (vides) ne sont pas significatives

Dans les exemples qui suivent, des lignes blanches (lignes vides) apparaissent simplement pour «aérer» les sorties.

```
$ cat f1.txt
abc

$ cat f2.txt
abc def
xxx
```

3. Format et exécution des commandes

Commandes et arguments

Format des commandes

Format général des commandes

```
commande options arguments
```

Commandes et arguments

Format des commandes

Exemple de format de description d'une commande

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

- Des `options` peuvent ou non être indiquées ([ ]).
- Un `motif` doit être indiqué.
- Zéro ([ ]), un ou plusieurs (...) `fichier`s peuvent être indiqués.
  Si aucun fichier n'est spécifié, `stdin` est utilisé.

Commandes et arguments

Format des commandes

Exemple de format de description d'une commande

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

- Des `options` peuvent ou non être indiquées ([ ]).
- Un `motif` doit être indiqué.
- Zéro ([ ]), un ou plusieurs (...) `fichier`s peuvent être indiqués.
  Si aucun fichier n'est spécifié, `stdin` est utilisé.

Commandes et arguments

Option de type *switch* vs. option de type *flag*


```
$ cat f1.txt
abc
```

```
$ cat f2.txt
abc
ab de
```

Quelle est la différence d'utilisation entre «-c» et «-f» ?

```
$ grep abc f2.txt
??

$ grep f1.txt f2.txt
??

$ grep -c f1.txt f2.txt
??

$ grep -f f1.txt f2.txt
??
```

Commandes et arguments

Option de type *switch* vs. option de type *flag*


```
$ cat f1.txt
abc
```

```
$ cat f2.txt
abc
ab de
```

Quelle est la différence d'utilisation entre «-c» et «-f» ?

```
$ grep abc f2.txt
abc

$ grep f1.txt f2.txt


$ grep -c f1.txt f2.txt
0

$ grep -f f1.txt f2.txt
abc
```

Commandes et arguments

Option de type *switch* vs. option de type *flag*


```
$ cat f1.txt
abc
```

```
$ cat f2.txt
abc
ab de
```

Quelle est la différence d'utilisation entre «`--count`» et «`--file`» ?

```
$ grep abc f2.txt
abc

$ grep f1.txt f2.txt

$ grep --count f1.txt f2.txt
??

$ grep --file=f1.txt f2.txt
??
```

Commandes et arguments

Option de type *switch* vs. option de type *flag*


```
$ cat f1.txt
abc
```

```
$ cat f2.txt
abc
ab de
```

Quelle est la différence d'utilisation entre «`--count`» et «`--file`» ?

```
$ grep abc f2.txt
abc

$ grep f1.txt f2.txt

$ grep --count f1.txt f2.txt
0

$ grep --file=f1.txt f2.txt
abc
```

Commandes et arguments

Les *switches*

Caractéristique d'une *switch*

N'a aucune valeur associée

Forme courte avec un tiret

```
$ grep -V
grep (BSD grep) 2.5.1-FreeBSD
```

Forme longue avec deux tirets et un identificateur

```
$ grep --version
grep (BSD grep) 2.5.1-FreeBSD
```

Commandes et arguments

Les *flags*

Caractéristique d'un *flag*

A une valeur associée — i.e., un argument associé.

Forme courte avec un tiret et un argument

```
$ grep -f fichier.txt argument.txt
...
```

Forme longue avec deux tirets et une affectation à un identificateur

```
$ grep --file=fichier.txt argument.txt
...
```

Commandes et arguments

Mais il y a des exceptions...

Format des *flags* : forme longue avec un seul tiret

```
$ javac --version
javac: invalid flag: --version
Usage: javac <options> <source files>
use -help for a list of possible options

$ javac -v
javac: invalid flag: -v
Usage: javac <options> <source files>
use -help for a list of possible options

$ javac ??
javac 1.8.0_25
```

Commandes et arguments

Mais il y a des exceptions. . .

Format des *flags* : forme longue avec un seul tiret

```
$ javac --version
javac: invalid flag: --version
Usage: javac <options> <source files>
use -help for a list of possible options

$ javac -v
javac: invalid flag: -v
Usage: javac <options> <source files>
use -help for a list of possible options

$ javac -version
javac 1.8.0_25
```

Commandes et arguments

Mais il y a des exceptions. . .

Format des *flags* : forme longue avec un seul tiret

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

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

- Forme courte avec «−» vs. forme longue avec «−−» : style du standard Posix, plus récent.
- De nombreuses commandes plus anciennes ont des formes longues avec un seul «−».

On utilise « ;» pour exécuter plusieurs commandes de façon séquentielle

```
$ ls; ls
AFAIRE          Divers/         Syllabus/       Varia/
ALIRE           Materiel/       TEMPS
AFAIRE          Divers/         Syllabus/       Varia/
ALIRE           Materiel/       TEMPS
```

On utilise «&» pour exécuter plusieurs commandes de façon concurrente... en arrière-plan (*background*)

```
 $ ls & ls &
[1] 9033
[2] 9034
AFAIRE ALIRE Divers/ Materiel/ Syllabus/ TEMPS Varia/
AFAIRE ALIRE Divers/ Materiel/ Syllabus/ TEMPS Varia/
[1]-  Done                    ls -CF
[2]+  Done                    ls -CF
$
```

Commandes et arguments

Trois sortes de commandes

Les commandes *built-in*

Commandes prédéfinies internes au *shell*.

= Commandes primitives exécutées directement par le *shell*.

Les fonctions

Fonctions définies/programmées dans le langage du *shell*.

Les commandes externes

Programmes externes exécutés dans un processus indépendant.

Commandes et arguments

Trois sortes de commandes : Liste partielle des *builtins*

- alias
- bg
- cd
- dirs
- echo
- exec
- exit
- fg
- history
- jobs
- kill
- logout
- popd
- printf
- pushd
- pwd
- source
- test
- times
- type
- unalias
- ...

Commandes et arguments

Trois sortes de commandes : Un exemple d'une fonction pour lancer un terminal

Pour créer un terminal avec un nom spécifique

```
function terminal()
{
  nom=$1

  if [[ $nom == "1" ]]; then
     geom="80X50"
     nom="T1"
  elif [[ $nom == "2" ]]; then
     geom="70X50-70"
     nom="T2"
  else
     geom="70X50+300"
  fi

  $TERMINAL -name "$nom" -bg white -geometry "$geom" &
}
```

Commandes et arguments

Trois sortes de commandes : Un exemple d'une fonction pour lancer un terminal

Pour créer un terminal avec un nom spécifique

```
terminal()
{
  nom=$1

  if [[ $nom == "1" ]]; then
     geom="80X50"
     nom="T1"
  elif [[ $nom == "2" ]]; then
     geom="70X50-70"
     nom="T2"
  else
     geom="70X50+300"
  fi

  $TERMINAL -name "$nom" -bg white -geometry "$geom" &
}
```

Commandes et arguments

Trois sortes de commandes : Les commandes externes

Par exemple, un script *bash*

```
$ cat foo.sh
#!/usr/bin/env bash
...

$ ./foo.sh
```

Processus d'exécution d'une commande externe

1. Un nouveau processus (enfant) est créé par le *shell*.
2. Le nouveau processus exécute le programme associé à la commande externe — ici, `foo.sh`.
3. Le processus parent — le *shell* de départ — attend que le processus enfant se termine, [a] puis poursuit son exécution.

a. Sauf si la commande est lancée avec «&».

4. Flux d’entrée et flux de sortie

Les flux `stdout` et `stderr`

Par défaut, les deux flux de sortie vont «à l'écran»

```
$ cat foo.rb
#!/usr/bin/env ruby

puts "+++ Foo!"
fail "*** Erreur"


$ ./foo.rb
+++ Foo!
./foo.rb:4:in '<main>': *** Erreur (RuntimeError)
```

Les flux `stdout` et `stderr`

??

```
$ cat foo.rb
#!/usr/bin/env ruby

puts "+++ Foo!"
fail "*** Erreur"


$ ./foo.rb >res.txt
??

$ cat res.txt
??
```

Les flux `stdout` et `stderr`

Redirection de `stdout` dans un fichier


```
$ cat foo.rb
#!/usr/bin/env ruby

puts "+++ Foo!"
fail "*** Erreur"


$ ./foo.rb >res.txt
./foo.rb:4:in '<main>': *** Erreur (RuntimeError)

$ cat res.txt
+++ Foo!
```

Les flux `stdout` et `stderr`

Redirection de `stdout` avec '>' (écrase l'ancien contenu) vs. '>>' (*append*)

```
$ cat foo.rb
#!/usr/bin/env ruby

puts "+++ Foo!"
fail "*** Erreur"


$ ./foo.rb >res.txt
./foo.rb:4:in '<main>': *** Erreur (RuntimeError)
$ ./foo.rb >>res.txt
./foo.rb:4:in '<main>': *** Erreur (RuntimeError)

$ cat res.txt
+++ Foo!
+++ Foo!
```

Les flux `stdout` et `stderr`

??

```
$ cat foo.rb
#!/usr/bin/env ruby

puts "+++ Foo!"
fail "*** Erreur"


$ ./foo.rb 2>res.txt
```

??

```
$ cat res.txt
```

??

Les flux `stdout` et `stderr`

Redirection de `stderr` dans un fichier


```
$ cat foo.rb
#!/usr/bin/env ruby

puts "+++ Foo!"
fail "*** Erreur"


$ ./foo.rb 2>res.txt
+++ Foo!

$ cat res.txt
./foo.rb:4:in '<main>': *** Erreur (RuntimeError)
```

Les flux `stdout` et `stderr`

Redirection de `stdout` et `stderr` vers des fichiers distincts


```
$ cat foo.rb
#!/usr/bin/env ruby

puts "+++ Foo!"
fail "*** Erreur"


$ ./foo.rb 1>out.txt 2>errs.txt

$ cat out.txt
+++ Foo!

$ cat errs.txt
./foo.rb:4:in '<main>': *** Erreur (RuntimeError)
```

Les flux `stdout` et `stderr`

??

```
$ cat foo.rb
#!/usr/bin/env ruby

puts "+++ Foo!"
fail "*** Erreur"


$ ./foo.rb >res.txt 2>&1
??
$ cat res.txt
??
```

Les flux `stdout` et `stderr`

Redirection de `stdout` et `stderr` vers un même fichier


```
$ cat foo.rb
#!/usr/bin/env ruby

puts "+++ Foo!"
fail "*** Erreur"


$ ./foo.rb >res.txt 2>&1

$ cat res.txt
./foo.rb:4:in '<main>': *** Erreur (RuntimeError)
+++ Foo!
```

Les flux `stdout` et `stderr`

Redirection de `stdout` et `stderr` vers un même fichier


```
$ cat foo.rb
#!/usr/bin/env ruby

puts "+++ Foo!"
fail "*** Erreur"


$ ./foo.rb &>res.txt

$ cat res.txt
./foo.rb:4:in '<main>': *** Erreur (RuntimeError)
+++ Foo!
```

Les flux `stdout` et `stderr`

Redirection de `stderr` pour ignorer les erreurs, sans créer un fichier


```
$ cat foo.rb
#!/usr/bin/env ruby

puts "+++ Foo!"
fail "*** Erreur"


$ ./foo.rb ??
+++ Foo!
```

??

Les flux `stdout` et `stderr`

Redirection de `stderr` pour ignorer les erreurs, sans créer un fichier


```
$ cat foo.rb
#!/usr/bin/env ruby

puts "+++ Foo!"
fail "*** Erreur"


$ ./foo.rb 2>/dev/null
+++ Foo!
```

Note : `/dev/null` est parfois appelé le *bit bucket* (la poubelle à bits)

Le flux `stdin`

Par défaut, `stdin` provient du clavier


```
$ cat >res.txt
abc
def
^D

$ cat res.txt
abc
def

$ cat <res.txt
abc
def
```

Le flux `stdin`

Question : On a vu qu'on pouvait utiliser «`>>`» pour `stdout`.
Est-ce qu'on peut utiliser «`<<`» pour `stdin` ?

Le flux `stdin`

La lecture de `stdin` peut se faire à partir d'un *here-document*

Utilisation d'un *here-document* au niveau du *shell*

```
$ cat <<EOF >res.txt
> abc
> def
> EOF

$ cat res.txt
abc
def

$ cat <<FOO
> abc
>   foo
>   FOO
> FOO
abc
  foo
  FOO
```

Le flux `stdin`

La lecture de `stdin` peut se faire à partir d'un *here document*

Utilisation d'un *here-document* dans un script

```
$ cat foo.sh
#!
cat <<FOO
  $0 $1
  FOO
  def
FOO


$ ./foo.sh bar
  ./foo.sh bar
  FOO
  def
```

Redirection avec un pipeline

Un pipeline connecte le `stdout` du premier au `stdin` du deuxième


```
$ cat bar.rb
#!/usr/bin/env ruby
puts STDIN.read
puts "*** FIN ***"


$ ./bar.rb <bar.rb
#!/usr/bin/env ruby
puts STDIN.read
puts "*** FIN ***"
*** FIN ***
```

```
$ ./bar.rb <bar.rb | ./bar.rb
#!/usr/bin/env ruby
puts STDIN.read
puts "*** FIN ***"
*** FIN ***
*** FIN ***


$ ./bar.rb <bar.rb | ./bar.rb | ./bar.rb
#!/usr/bin/env ruby
puts STDIN.read
puts "*** FIN ***"
*** FIN ***
*** FIN ***
*** FIN ***
```

5. Arborescence des fichiers

5.1 Listage des fichiers

La commande `ls`

Avec l'option «`-F`», utilisée dans l'alias mentionné plus haut

Donne la liste des fichiers du répertoire courant et leur «type» :

```
<<~@MacBook>> $ cd ~/INF600A

<<~/INF600A@MacBook>> $ ls
AFAIRE          Divers/         Syllabus/       Varia/
ALIRE           Materiel/       TEMPS

<<~/INF600A@MacBook>> $ ls -F
AFAIRE          Divers/         Syllabus/       Varia/
ALIRE           Materiel/       TEMPS

<<~/INF600A@MacBook>> $ \ls
AFAIRE          Divers          Syllabus        Varia
ALIRE           Materiel        TEMPS
```

Note : «`\ls`» dénote la «vraie» commande `ls`, i.e., ignore l'alias mentionné plus haut, d'où l'absence des «`/`» après les répertoires.

La commande `ls`

Avec l'option «`-1`»

L'option «`-1`» permet d'afficher un fichier par ligne — utile pour *piper* le résultat à une autre commande :

```
<<~/INF600A@MacBook>> $ ls -1
AFAIRE
ALIRE
Divers/
Materiel/
Syllabus/
TEMPS
Varia/
```

Remarque : Règle générale, Unix/Linux est sensible à la casse (*case-sensitive*), donc `fich` et `Fich` sont deux identificateurs différents. Une exception à cette règle = Mac OS X !?

La commande `ls`

Avec l'option «`-l`»

L'option «`-l`» permet d'afficher les méta-données associées à chaque fichier :

```
<<~/INF600A@MacBook>> $ ls -l
total 24
-rw-r--r--   1 tremblay  staff    90 Dec 14 18:54 AFAIRE
-rw-r--r--   1 tremblay  staff    76 Dec 11 07:29 ALIRE
drwxr-xr-x   3 tremblay  staff
102 Dec 23 06:31 Divers/
drwxr-xr-x  36 tremblay  staff  1224 Jan
7 11:29 Materiel/
drwxr-xr-x  11 tremblay  staff   374 Jan
6 14:05 Syllabus/
-rw-r--r--   1 tremblay  staff   302 Dec 23 06:31 TEMPS
drwxr-xr-x   8 tremblay  staff   272 Jan  5 16:02 Varia/
```

La commande `ls`

Avec l'option «`-a`»

L'option «`-a`» permet d'afficher les fichiers invisibles — i.e., ceux dont le nom débute par «.» :

```
<<~/INF600A@MacBook>> $ ls -a -1
./
../
.git/
.gitignore
AFAIRE
ALIRE
Divers/
Materiel/
Syllabus/
TEMPS
Varia/
```

La commande `ls`

Avec l'option «`-R`»

L'option «`-R`» permet d'afficher récursivement le répertoire courant et ses sous-répertoires :

```
<<~/INF600A/Materiel@MacBook>> $ ls -R
Figures/   bash.tex  intro.tex  performances.txt
TEMPS      biblio@   latex/     preamble.tex
avec-notes.tex build.tex macros.tex sans-notes.tex
avec-sans-notes.tex intro-unix.txt makefile

./Figures:
duck-typing-1.png    geek-poke-build2.png
graphe-dependances.png ruby-history.png
geek-poke-build1.png graphe-dependances.fig nails.png

./latex:
ruby.tex         sectsty.sty
```

L'utilitaire `tree`

L'utilitaire `tree` permet d'afficher récursivement le répertoire courant et ses sous-répertoires, mais sous la forme d'un arbre — voir diapositive suivante.

Remarque : N'est pas toujours installé par défaut.

Voir : `http://mama.indstate.edu/users/ice/tree/`

L'utilitaire `tree`

Un exemple d'utilisation — où «->» indique un lien symbolique


```
<<~/INF600A/Materiel@MacBook>> $ tree
.
|-- Figures
|   |-- duck-typing-1.png
|   |-- geek-poke-build1.png
|   |-- geek-poke-build2.png
|   |-- graphe-dependances.fig
|   |-- graphe-dependances.png
|   |-- nails.png
|   `-- ruby-history.png
|-- TEMPS
|-- avec-notes.tex
|-- avec-sans-notes.tex
|-- bash.tex
|-- biblio -> /Users/tremblay/biblio
|-- build.tex
|-- intro-unix.txt
|-- intro.tex
|-- latex
|   |-- ruby.tex
|   `-- sectsty.sty
|-- macros.tex
|-- makefile
|-- performances.txt
|-- preamble.tex
`-- sans-notes.tex

3 directories, 21 files
```

5.2 Caractères spéciaux et *file globbing*

Caractères spéciaux et *file globbing*

- *File globbing* : Processus qui consiste à remplacer les caractères spéciaux dans des patrons de noms de fichiers pour générer un ou plusieurs noms de fichiers.

- Les caractères spéciaux qui peuvent être utilisés dans des patrons de noms de fichiers :

```
?
*
[...]
[!...]
{...}
```

Caractères spéciaux

Caractère « ? » : Un (1) caractère quelconque

```
$ ls
a1   b2   c3   f1.txt   f2.txt   f3.txt

$ ls f?.txt
f1.txt   f2.txt   f3.txt

$ ls ??
a1   b2   c3

$ ls ?????
??

$ ls ??????
??
```

Caractères spéciaux

Caractère « ? » : Un (1) caractère quelconque

```
$ ls
a1  b2  c3  f1.txt  f2.txt  f3.txt

$ ls f?.txt
f1.txt  f2.txt  f3.txt

$ ls ??
a1  b2  c3

$ ls ?????
ls: impossible d'accéder à ?????: Aucun fichier ou
dossier de ce type

$ ls ??????
f1.txt f2.txt f3.txt
```

Caractères spéciaux

Caractère «⋆» : Zéro, un ou plusieurs caractères arbitraires

```
$ ls
a1   b2   c3   f1.txt   f2.txt   f3.txt

$ ls *.txt
f1.txt   f2.txt   f3.txt

$ ls f*.*
f1.txt   f2.txt   f3.txt

$ ls ??*
??
```

Remarque : Il ne faut pas confondre le «⋆» du *file globbing* (0, 1 ou plusieurs caractères arbitraires) avec le «⋆» des expressions régulières (0, 1 ou plusieurs occurrences du patron qui précède).

Caractères spéciaux

Caractère « ⋆ » : Zéro, un ou plusieurs caractères arbitraires

```
$ ls
a1  b2  c3  f1.txt  f2.txt  f3.txt

$ ls *.txt
f1.txt  f2.txt  f3.txt

$ ls f*.*
f1.txt  f2.txt  f3.txt

$ ls ??*
a1 b2 c3 f1.txt f2.txt f3.txt
```

Remarque : Il ne faut pas confondre le « ⋆ » du *file globbing* (0, 1 ou plusieurs caractères arbitraires) avec le « ⋆ » des expressions régulières (0, 1 ou plusieurs occurrences du patron qui précède).

Caractères « [. . .] » : Une classe de caractères

```
$ ls
a1   b2   c3   f1.txt   f2.txt   f3.txt

$ ls [abc]*
a1   b2   c3

$ ls [a-c]*
a1   b2   c3

$ ls [a-cf]*
a1   b2   c3   f1.txt   f2.txt   f3.txt

$ ls [af][12]*
??
```

Caractères « [. . .] » : Une classe de caractères

```
$ ls
a1  b2  c3  f1.txt  f2.txt  f3.txt

$ ls [abc]*
a1  b2  c3

$ ls [a-c]*
a1  b2  c3

$ ls [a-cf]*
a1  b2  c3  f1.txt  f2.txt  f3.txt

$ ls [af][12]*
a1 f1.txt f2.txt
```

Caractères « [! . . .] » : Exclusion d'une classe de caractères

```
$ ls
a1   b2   c3   f1.txt   f2.txt   f3.txt

$ ls [!f]*
a1   b2   c3

$ ls [!a-c]*
??
```

Caractères « [! . . .] » : Exclusion d'une classe de caractères

```
$ ls
a1  b2  c3  f1.txt  f2.txt  f3.txt

$ ls [!f]*
a1  b2  c3

$ ls [!a-c]*
f1.txt f2.txt f3.txt
```

Caractères spéciaux

« ? » et « ∗ » ne matchent pas le « . » des fichiers cachés

```
$ ls -a
./  ../  a1  b2  c3  f1.txt  f2.txt  f3.txt  .Transferes

$ ls -a *
a1  b2  c3  f1.txt  f2.txt  f3.txt

$ ls -a ?*
a1  b2  c3  f1.txt  f2.txt  f3.txt

$ ls .*
.Transferes

.:
a1  b2  c3  f1.txt  f2.txt  f3.txt

..:
aTmp@   emacs-dir/        INF600A/ pruby/  Tests-Oto2/
biblio/ ExemplesCucumber/ INF7235/ Public/
...
```

Caractères spéciaux

On utilise «\» pour ignorer la signification spéciale du caractère

```
$ ls
a1   b2   c3   f1.txt   f2.txt   f3.txt

$ touch *
$ ls

??

$ touch \*
$ ls
??

$ touch \?
$ ls
??
```

Caractères spéciaux

On utilise «\» pour ignorer la signification spéciale du caractère

```
$ ls
a1  b2  c3  f1.txt  f2.txt  f3.txt

$ touch *
$ ls

a1 b2 c3 f1.txt f2.txt f3.txt

$ touch \*
$ ls
* a1 b2 c3 f1.txt f2.txt f3.txt

$ touch \?
$ ls
? * a1 b2 c3 f1.txt f2.txt f3.txt
```

Caractères spéciaux

Caractères blancs

```
$ touch un fichier
$ ls
??
$ ls u*
??

$ touch un\ autre\ fichier
$ ls -1 u*
??

$ touch "un autre fichier"
$ ls -1 u*
??
```

Caractères spéciaux

Caractères blancs

```
$ touch un fichier
$ ls
? * a1 b2 c3 f1.txt f2.txt f3.txt fichier un
$ ls u*
un

$ touch un\ autre\ fichier
$ ls -1 u*
un
un autre fichier

$ touch "un autre fichier"
$ ls -1 u*
un
un autre fichier
```

Accolades {...}

```
$ ls
a1  b2  c3  f1.txt  f2.txt  f3.txt

$ ls f{1,2}.txt
f1.txt    f2.txt

$ ls f{1,XX}.txt
ls: impossible d'accéder à fXX.txt: Aucun fichier ou dossier d
f1.txt


$ mv f{1,XX}.txt; ls
a1  b2  c3  ??


$ mv f{XX,}.txt; ls
a1  b2  c3  ??
```

Caractères spéciaux

Accolades {...}

```
$ ls
a1  b2  c3  f1.txt  f2.txt  f3.txt

$ ls f{1,2}.txt
f1.txt    f2.txt

$ ls f{1,XX}.txt
ls: impossible d'accéder à fXX.txt: Aucun fichier ou dossier d
f1.txt


$ mv f{1,XX}.txt; ls
a1  b2  c3  f2.txt f3.txt fXX.txt


$ mv f{XX,}.txt; ls
a1  b2  c3  f2.txt f3.txt f.txt
```

5.3 Navigation

Le répertoire courant

La commande `pwd` (*print working directory*) retourne le répertoire courant.

```
<<~/INF600A@MacBook>> $ cd ~/INF600A

<<~/INF600A@MacBook>> $ pwd
/Users/tremblay/INF600A
```

Changement de répertoire avec `cd`

La commande `cd` change le répertoire courant, i.e., on se déplace dans un autre répertoire.

On peut spécifier un chemin absolu ou un chemin relatif au répertoire courant.

Déplacement relatif

```
<<~/INF600A@MacBook>> $ pwd
/Users/tremblay/INF600A

<<~/INF600A@MacBook>> $ ls
AFAIRE          Divers/         Syllabus/       Varia/
ALIRE           Materiel/       TEMPS

<<~/INF600A@MacBook>> $ cd Materiel

<<~/INF600A/Materiel@MacBook>> $ pwd
/Users/tremblay/INF600A/Materiel
```

Déplacement absolu

```
<<~/INF600A/Materiel@MacBook>> $ pwd
/Users/tremblay/INF600A/Materiel

<<~/INF600A/Materiel@MacBook>> $ cd /Users/tremblay/INF600A

<<~/INF600A@MacBook>> $ pwd
/Users/tremblay/INF600A
```

Changement de répertoire avec `cd`

Déplacement avec «~»

Déplacement avec «~» = répertoire de base (`$HOME`)

```
<<~/INF600A/Materiel@MacBook>> $ cd ~/INF600A

<<~/INF600A@MacBook>> $ pwd
/Users/tremblay/INF600A

<<~/INF600A@MacBook>> $ cd ~

<<~@MacBook>> $ cd $HOME

<<~@MacBook>> $ cd

<<~@MacBook>> $
```

Note : `cd` sans argument retourne aussi à `$HOME`.

Changement de répertoire avec `cd`

Déplacement relatif à «..»

Déplacement avec « . . » = répertoire parent

```
<<~/INF600A@MacBook>> $ ls
AFAIRE      Divers/      Syllabus/    Varia/
ALIRE       Materiel/    TEMPS

<<~/INF600A@MacBook>> $ cd Materiel

<<~/INF600A/Materiel@MacBook>> $ cd ..

<<~/INF600A@MacBook>> $ pwd
/Users/tremblay/INF600A
```

Changement de répertoire avec `pushd`

La pile des répertoires visités

- La commande `pushd` change de répertoire courant, comme `cd` ET le répertoire destination est empilé au sommet de la pile des répertoires.
- La commande `popd` désempile le répertoire au sommmet de la pile et va dans le répertoire indiqué par le nouveau sommet.
- Donc, le sommet de la pile indique toujours le répertoire courant.
- Le contenu de la pile peut être examiné avec `dirs`.

Changement de répertoire avec `pushd`

Exemples

```
<<~@MacBook>> $ pwd
/Users/tremblay
<<~@MacBook>> $ dirs
~
<<~@MacBook>> $ pushd INF600A
~/INF600A ~
<<~/INF600A@MacBook>> $ dirs
~/INF600A ~
<<~/INF600A@MacBook>> $ pushd ~
~ ~/INF600A ~
<<~@MacBook>> $ popd
~/INF600A ~
<<~/INF600A@MacBook>> $ dirs
~/INF600A ~
```

6. Manipulation de fichiers et répertoires

Manipulation de fichiers

Copie d'un fichier

```
$ cp source destination
```

Déplacement/changement de nom d'un fichier

```
$ mv source destination
```

Suppression d'un fichier

```
$ rm fichier
```

Manipulation de répertoires

Création d'un répertoire

```
$ mkdir repertoire
```

Copie d'un répertoire

```
$ cp -r source destination
```

Suppression d'un répertoire

```
$ rmdir repertoire
```

```
$ rm -r repertoire
```

7. Complétion et historique de commandes

Complétion automatique des commandes

Lorsqu'on tape un nom de commande ou un nom de fichier, on peut ne taper qu'une partie du nom et faire compléter la partie restante par le *shell*.

Notons par `<TAB>` le fait de presser le caractère de tabulation.

Complétion automatique des commandes


```
<<~/INF600A@MacBook>> $ mo<TAB>
mogrify             more-md      mount_exfat mount_smbfs
moi                 mount        mount_fdesc mount_udf
montage             mount_acfs   mount_ftp   mount_webdav
moose-outdated      mount_afp    mount_hfs   moyenne
moose-outdated5.12 mount_cd9660 mount_msdos moyenne+
moose-outdated5.16 mount_cddafs mount_nfs
more                mount_devfs  mount_ntfs

<<~/INF600A@MacBook>> $ more<TAB>
??

<<~/INF600A@MacBook>> $ more A<TAB>
AFAIRE  ALIRE
```

Complétion automatique des commandes


```
<<~/INF600A@MacBook>> $ mo<TAB>
mogrify             more-md      mount_exfat mount_smbfs
moi                 mount        mount_fdesc mount_udf
montage             mount_acfs   mount_ftp   mount_webdav
moose-outdated      mount_afp    mount_hfs   moyenne
moose-outdated5.12 mount_cd9660 mount_msdos moyenne+
moose-outdated5.16 mount_cddafs mount_nfs
more                mount_devfs  mount_ntfs

<<~/INF600A@MacBook>> $ more<TAB>
more more-md

<<~/INF600A@MacBook>> $ more A<TAB>
AFAIRE  ALIRE
```

Historique des commandes exécutées

Les commandes exécutées sont conservées dans un historique — par défaut, les 500 dernières commandes sont conservées :

```
<<~@MacBook>> $ history 5
  502  cd INF600A
  503  pwd
  504  ls
  505  dirs
  506  history 5

<<~@MacBook>> $ history
    8  make
  ...
  504  ls
  505  dirs
  506  history 5
  507  history
```

Historique des commandes exécutées

Indication explicite d'une commande précédente

On peut réexécuter une commande en indiquant son numéro, le début de la commande, ou un «extrait» de la commande :

```
<<~@MacBook>> $ !482
dirs
~

<<~@MacBook>> $ !di
dirs
~

<<~@MacBook>> $ !?ir?
dirs
~
```

Historique des commandes exécutées

Recherche avec `grep`

Recherche du bon numéro avec `grep`

```
<<~/Tmp@MacBook>>  $ history | grep cd | tail
 1284  cd Materiel/
 1286  cd ../Transparents/
 1303  cd Materiel/
 1633  history | grep cd
 1634  history | grep cd | tail
 1635  history | grep cd | tail
 1636  cd Figures/
 1637  cd ~/Figures
 1638  cd ~/INF600A
 1639  history | grep cd | tail
<<~/Tmp@MacBook>>  $ !1638
cd ~/INF600A
<<~/INF600A@MacBook>> $
```

Historique des commandes exécutées

Réexécution de la commande précédente, telle quelle ou modifiée

On peut aussi réexécuter la commande précédente, telle quelle ou modifiée :

```
<<~@MacBook>> $ ls agent
Gemfile      README.md   agent.gemspec benchmark/ lib/
Gemfile.lock Rakefile    autotest/     examples/  spec/

<<~@MacBook>> $ !!
ls agent
Gemfile      README.md   agent.gemspec benchmark/ lib/
Gemfile.lock Rakefile    autotest/     examples/  spec/

<<~@MacBook>> $ !!:s/gent/rchive/
ls archive
Backup-wikidot/    Pendu/    VieuxWindows/ wifi backups/
Choix cours Hiver/ STUFF-VIEUX-MAC/ iPad/
```

Historique des commandes exécutées

Réexécution de la commande précédente, mais modifiée

Utile pour répéter une commande sur quelques fichiers :

```
<~/Tmp@MacBook>> $ ls UnDir

<<~/Tmp@MacBook>> $ cp f1.txt UnDir

<<~/Tmp@MacBook>> $ ^1^2
??

<<~/Tmp@MacBook>> $ ls UnDir
??
```

Historique des commandes exécutées

Réexécution de la commande précédente, mais modifiée

Utile pour répéter une commande sur quelques fichiers :

```
<~/Tmp@MacBook>> $ ls UnDir

<<~/Tmp@MacBook>> $ cp f1.txt UnDir

<<~/Tmp@MacBook>> $ ^1^2
cp f2.txt UnDir

<<~/Tmp@MacBook>> $ ls UnDir
f1.txt f2.txt
```

Historique des commandes exécutées

Réexécution de la commande précédente, mais modifiée (bis)

Utile pour répéter une commande sur quelques fichiers :

```
<<~/Tmp@MacBook>> $ ls
f1.txt            f2.txt            journal.txt

<<~/Tmp@MacBook>> $ mv f1.txt fich1.txt

<<~/Tmp@MacBook>> $ !!:gs/1/2/
??

<<~/Tmp@MacBook>> $ ls
??
```

Historique des commandes exécutées

Réexécution de la commande précédente, mais modifiée (bis)

Utile pour répéter une commande sur quelques fichiers :

```
<<~/Tmp@MacBook>> $ ls
f1.txt            f2.txt            journal.txt

<<~/Tmp@MacBook>> $ mv f1.txt fich1.txt

<<~/Tmp@MacBook>> $ !!:gs/1/2/
mv f2.txt fich2.txt

<<~/Tmp@MacBook>> $ ls
fich1.txt fich2.txt journal.txt
```

Historique des commandes exécutées

Extraction des arguments de la commande précédente

Dernier argument = « ! $ » ; tous les arguments = « ! ⋆ »

```
$ ls

$ touch f1 f2

$ \cp !$ f3
\cp f2 f3

$ ls !*
ls f2 f3
f2  f3

$ ls
f1  f2  f3
```

Historique des commandes exécutées

Navigation avec les flèches

On peut aussi naviguer dans l'historique avec les flèches :

```
<<~/Tmp@MacBook>> $ history 5
  531  history
  532  history 5
  533  pwd
  534  cd ~/Tmp
  535  history 5
<<~/Tmp@MacBook>> $ ↑
<<~/Tmp@MacBook>> $ history 5
<<~/Tmp@MacBook>> $ ↑
<<~/Tmp@MacBook>> $ cd ~/Tmp
<<~/Tmp@MacBook>> $ ↑
<<~/Tmp@MacBook>> $ pwd
<<~/Tmp@MacBook>> $ ↓
<<~/Tmp@MacBook>> $ cd ~/Tmp
```

Historique des commandes exécutées

Recherche dans l'historique

Et on peut aussi naviguer en fouillant dans l'historique.

Notons <Ctrl-r> = presser les touches «Control-r» suivies d'autres caractères en bleu :

```
<<~/Tmp@MacBook>> $ <Ctrl-r>
(reverse-i-search)'': c
(reverse-i-search)'c': mv f2.txt fich2.txtd
(reverse-i-search)'cd': cd<Ctrl-r>
(reverse-i-search)'cd': cd ..
(reverse-i-search)'cd': cd<Ctrl-r>
(reverse-i-search)'cd': cd ~/aTmp
<<~@MacBook>> $
```

Autres commandes pour naviguer dans l'historique

Mode `emacs`

Les commandes ci-bas permettent de naviguer dans l'historique.

Hypothèse : On suppose que le mode d'édition par défaut n'a pas été modifié — «`set -o emacs`».

Commande	**Description**
Ctrl-p	Aller à la ligne précédente
Ctrl-n	Aller à la ligne suivante
Ctrl-r	Rechercher vers l'arrière
Esc-<	Aller à la première ligne
Esc->	Aller à la dernière ligne

Édition de la commande courante

Mode `emacs`

Une fois une commande obtenue, on peut l'éditer avec les commandes `emacs` habituelles :

Commande	**Description**
Ctrl-b	Reculer d'un caractère
Ctrl-f	Avancer d'un caractère
Esc-b	Reculer d'un mot
Esc-f	Avancer d'un mot
Del	Effacer le caractère qui précède
Ctrl-d	Effacer le caractère courant
...	...

Autres commandes pour naviguer dans l'historique et éditer la commande courante

Mode `vi`

Si la commande suivante a été exécutée au préalable (par ex., fichier de configuration), alors les commandes de navigation et d'édition sont celles de `vi` :

```
set -o vi
```

À vous de voir si vous préférez ce dernier mode. . .

Remarque : C'est le mode `emacs` qui est celui par défaut !

8. Configuration de l’environnement

Les variables d'environnement

??

Définition et utilisation d'UNE_VAR

```
<<~@MacBook>> $ UNE_VAR=foo

<<~@MacBook>> $ echo $UNE_VAR
foo

<<~@MacBook>> $ UNE_VAR = foo
```

??

- ??

Les variables d'environnement

Il ne faut pas mettre d'espace autour de «=» !

Définition et utilisation d'UNE_VAR

```
<<~@MacBook>> $ UNE_VAR=foo

<<~@MacBook>> $ echo $UNE_VAR
foo

<<~@MacBook>> $ UNE_VAR = foo
bash: UNE_VAR: command not found
```

- Il ne faut pas mettre d'espace autour de «=» ! Les espaces séparent les arguments fournis à la commande (1^{er} symbole).

Les variables d'environnement

`export` rend la variable visible dans l'environnement et dans les processus enfants

Définition et utilisation d'`UNE_VAR`

```
<<~@MacBook>> $ UNE_VAR=foo

<<~@MacBook>> $ env | grep UNE_VAR

<<~@MacBook>> $ export UNE_VAR

<<~@MacBook>> $ env | grep UNE_VAR
UNE_VAR=foo
```

Les variables d'environnement

`export` rend la variable visible dans l'environnement et dans les processus enfants

Définition et utilisation d'`UNE_VAR`

```
<<~@MacBook>> $ UNE_VAR=foo

<<~@MacBook>> $ env | grep UNE_VAR

<<~@MacBook>> $ export UNE_VAR

<<~@MacBook>> $ env | grep UNE_VAR
UNE_VAR=foo
```

Note : On peut aussi définir et exporter en une seule instruction :

```
$ export UNE_VAR=foo
```

Les variables d'environnement

`export` rend la variable visible dans l'environnement et dans les processus enfants

Définition et utilisation d'`UNE_VAR`

```
$ cat baz.sh
echo $UNE_VAR

$ UNE_VAR=foo

$ ./baz.sh

$ export UNE_VAR=foo

$ ./baz.sh
foo

$ UNE_VAR=baz ./baz.sh
baz

$ echo $UNE_VAR
foo
```

Les fichiers `.bash_profile` et `.bashrc`

Fichier `.bash_profile`

Les commandes de ce fichier sont exécutées lorsque vous vous connectez à votre compte — donc généralement exécutées une fois par session, par le *login shell*.

Fichier `.bashrc`

Les commandes de ce fichier sont exécutées à chaque fois qu'un nouveau *shell* est lancé.

Contenu «typique» du fichier `.bash_profile`

Lorsqu'on a rien de spécial à faire lors d'une connection initiale

```
$ cat ~/.bash_profile
# .bash_profile

# Get the aliases and functions
if [ -f ~/.bashrc ]; then
        . ~/.bashrc       # Note: "." == "source"
fi

# User specific environment and startup programs
export PATH=$PATH:$HOME/bin
```

Contenu typique du fichier `.bashrc`


```
$ cat ~/.bashrc
# Definition de diverses variables globales.
export EDITOR=emacs
export OPEN_OFFICE=/usr/bin/soffice
...
export PS1='<<\w@MacBook>> $ '
...

# Pour corriger automatiquement les noms de repertoire.
shopt -s cdspell

# Definition de divers alias.
source ~/.alias

# Ajout de repertoires additionnels au PATH.
export PATH=$PATH:$HOME/bin
```

Des raccourcis : les alias

Quelques-un de mes alias personnels — parmi presque 300 :

```
$ cat ~/.alias
alias ll='ls -l'
alias ls='ls -F'

alias p='more'
alias cp='cp -i -p'
alias rm='rm -i'
alias mv='mv -i'

alias +='pushd'
alias pp='popd'
...
alias +600a='+ ~/INF600A'
...
```

```
...
alias emac='emacs'
alias emcas='emacs'
alias emcsa='emacs'
...
alias push='git push'
alias pull='git pull'
alias upd='git pull'
alias ci="git commit"
alias st="git status"
alias add="git add"
alias lo='git lo'
...
```

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, 2005.

. . . et de nombreux sites Web !

Questions pour le prochain cours

Qui connait les base de `git` ?

Questions pour le prochain cours

Qui connait les base de `git` ?

Qui utilise régulièrement `git` ?

Questions pour le prochain cours

Qui connait les base de `make` ?

Questions pour le prochain cours

Qui connait les base de `make` ?

Qui utilise régulièrement `make` ?