

Administration d'un système GNU / Linux

03 — Scripts bash

Hery Mampionona R.

INSI UNIVERSITY

Année : 2025



Commandes plus complexes

- Il arrive que l'on écrive des combinaisons de commandes que l'on a envie de réutiliser ;

Commandes plus complexes

- Il arrive que l'on écrive des combinaisons de commandes que l'on a envie de réutiliser ;
- On n'a pas envie de les apprendre par cœur, et elles sont parfois longues à écrire ;

Commandes plus complexes

- Il arrive que l'on écrive des combinaisons de commandes que l'on a envie de réutiliser ;
- On n'a pas envie de les apprendre par cœur, et elles sont parfois longues à écrire ;
- On peut définir des "synonymes" pour ces commandes ;

Commandes plus complexes

- Il arrive que l'on écrive des combinaisons de commandes que l'on a envie de réutiliser ;
- On n'a pas envie de les apprendre par cœur, et elles sont parfois longues à écrire ;
- On peut définir des "synonymes" pour ces commandes ;

```
$ mise_a_jour
```

au lieu de

```
$ sudo apt-get update && sudo apt-get upgrade
```

Commandes plus complexes

- Ces synonymes se définissent avec la commande **alias** ;

Commandes plus complexes

- Ces synonymes se définissent avec la commande **alias** ;
- La syntaxe est simple :
alias nouvelle_commande='autre commande'

Commandes plus complexes

- Ces synonymes se définissent avec la commande `alias` ;
- La syntaxe est simple :

`alias nouvelle_commande='autre commande'`

Pour l'exemple precedent :

`alias mise_a_jour='sudo apt-get update && sudo apt-get upgrade'`

Commandes plus complexes

- **alias** sans paramètre donne la liste des synonymes;

Commandes plus complexes

- **alias** sans paramètre donne la liste des synonymes ;
- **unalias** supprime des synonymes :

Commandes plus complexes

- **alias** sans paramètre donne la liste des synonymes ;
- **unalias** supprime des synonymes :
 - **unalias** commande supprime le synonyme donné en paramètre ;

Commandes plus complexes

- **alias** sans paramètre donne la liste des synonymes ;
- **unalias** supprime des synonymes :
 - **unalias** commande supprime le synonyme donné en paramètre ;
 - **unalias -a** les supprime tous ;

Commandes plus complexes

- **alias** sans paramètre donne la liste des synonymes ;
- **unalias** supprime des synonymes :
 - **unalias** commande supprime le synonyme donné en paramètre ;
 - **unalias** -a les supprime tous ;



Les changements ne sont valables que pour la session actuelle !

Commandes plus complexes

- On doit enregistrer les alias dans un fichier pour pouvoir les réutiliser ;

Commandes plus complexes

- On doit enregistrer les alias dans un fichier pour pouvoir les réutiliser ;
- Nous les placerons dans le fichier `~/.bashrc`, qui est chargé au début de chaque session ;

Commandes plus complexes

- On doit enregistrer les alias dans un fichier pour pouvoir les réutiliser ;
- Nous les placerons dans le fichier `~/.bashrc`, qui est chargé au début de chaque session ;
- Pour recharger `~/.bashrc` sans devoir se reconnecter, on utilise la commande **source** `~/.bashrc` ;

Commandes plus complexes

For almost every purpose, shell functions are preferred over aliases.

http://www.gnu.org/software/bash/manual/html_node/Aliases.html#Aliases

- Les **alias** sont utiles mais limités : on ne peut pas les "paramétrer" ;

Commandes plus complexes

For almost every purpose, shell functions are preferred over aliases.

http://www.gnu.org/software/bash/manual/html_node/Aliases.html#Aliases

- Les **alias** sont utiles mais limités : on ne peut pas les "paramétrer" ;
- Il est donc souvent utile de pouvoir écrire des fonctions, ou des programmes plus complexes qu'on réutilisera ;

Commandes plus complexes

For almost every purpose, shell functions are preferred over aliases.

http://www.gnu.org/software/bash/manual/html_node/Aliases.html#Aliases

- Les **alias** sont utiles mais limités : on ne peut pas les "paramétrer" ;
- Il est donc souvent utile de pouvoir écrire des fonctions, ou des programmes plus complexes qu'on réutilisera ;
- Pour y arriver, il faut connaître le langage du *shell* qu'on utilise ;

Références sur la programmation pour bash

La référence suivante est très complète et pédagogique :



“Advanced Bash-Scripting Guide : An in-depth exploration of the art of shell scripting”, de Mendel Cooper

<https://tldp.org/LDP/abs/html/>

En-tête : #!



Il existe **beaucoup** de *shells* différents : nous ne parlerons ici que de `bash`, le plus répandu.

- On utilise le format `fichier.sh` pour nommer les scripts ;

En-tête : #!



Il existe **beaucoup** de *shells* différents : nous ne parlerons ici que de `bash`, le plus répandu.

- On utilise le format `fichier.sh` pour nommer les scripts ;
- `fichier.sh` doit être exécutable (utilisez `chmod` si nécessaire); on le lance avec la commande `./fichier.sh` ;

En-tête : #!



Il existe **beaucoup** de *shells* différents : nous ne parlerons ici que de `bash`, le plus répandu.

- On utilise le format `fichier.sh` pour nommer les scripts ;
- `fichier.sh` doit être exécutable (utilisez `chmod` si nécessaire); on le lance avec la commande `./fichier.sh` ;
- Pour que GNU sache qu'il faut utiliser `bash` pour exécuter votre script, on insère en première ligne ceci :

`#!/bin/bash`

Commentaires

- Les commentaires s'écrivent comme en Python : tout ce qui suit un # est ignoré ;

Commentaires

- Les commentaires s'écrivent comme en Python : tout ce qui suit un # est ignoré ;
- On ne peut pas commenter plusieurs lignes à la fois (pas d'équivalent de *'''Très \n longue \n chaîne.'''* ou des */* commentaires en C(++) */*) ;

Variables et affectations

- Pour déclarer une variable, on écrit : `nomvariable=valeur`



N'écrivez pas d'espaces autour du = ! Sinon, vous aurez l'erreur "command not found".

Variables et affectations

- Pour déclarer une variable, on écrit : **nomvariable**=valeur



N'écrivez pas d'espaces autour du = ! Sinon, vous aurez l'erreur "command not found".

- Pour obtenir la **valeur** d'une variable, on utilise l'opérateur \$ (exemple : **\$nomvariable**).

Variables et affectations

- Pour déclarer une variable, on écrit : **nomvariable**=valeur



N'écrivez pas d'espaces autour du = ! Sinon, vous aurez l'erreur "command not found".

- Pour obtenir la **valeur** d'une variable, on utilise l'opérateur \$ (exemple : **\$nomvariable**).
- On peut supprimer une variable avec la commande **unset** nomvariable ;

Variables et affectations

- L'opérateur \$ permet aussi d'évaluer le résultat d'une commande.

Variables et affectations

- L'opérateur \$ permet aussi d'évaluer le résultat d'une commande.

On peut mettre la date au format AAAA-MM-JJ dans la variable aujourd'hui comme ceci :

```
aujourd'hui=$(date -I)
```

Variables et affectations

- L'opérateur \$ permet aussi d'évaluer le résultat d'une commande.

On peut mettre la date au format AAAA-MM-JJ dans la variable aujourd'hui comme ceci :

```
aujourd'hui=$(date -I)
```

- Attention aux parenthèses !

Variables et affectations

- L'opérateur \$ permet aussi d'évaluer le résultat d'une commande.

On peut mettre la date au format AAAA-MM-JJ dans la variable aujourd'hui comme ceci :

```
aujourd'hui=$(date -I)
```

- Attention aux parenthèses !
 - \$date est la valeur de la variable date ;

Variables et affectations

- L'opérateur \$ permet aussi d'évaluer le résultat d'une commande.

On peut mettre la date au format AAAA-MM-JJ dans la variable aujourd'hui comme ceci :

```
aujourd'hui=$(date -I)
```

- Attention aux parenthèses !
 - \$date est la valeur de la variable date ;
 - \$(date) est le résultat de la commande date ;

Pas de types

- Les variables en bash n'ont pas de type;

Pas de types

- Les variables en bash n'ont pas de type ;
- Elles peuvent être vues comme des chaînes, sur lesquelles on peut parfois faire des calculs ;

Pas de types

- Les variables en bash n'ont pas de type ;
- Elles peuvent être vues comme des chaînes, sur lesquelles on peut parfois faire des calculs ;
- Que se passe-t-il quand on veut utiliser un calcul pour définir une variable ?

Pas de types

- Les variables en bash n'ont pas de type;
- Elles peuvent être vues comme des chaînes, sur lesquelles on peut parfois faire des calculs;
- Que se passe-t-il quand on veut utiliser un calcul pour définir une variable?
- `x=1+1`; `echo $x` donne ... `1+1`;

Pas de types

- Les variables en bash n'ont pas de type;
- Elles peuvent être vues comme des chaînes, sur lesquelles on peut parfois faire des calculs;
- Que se passe-t-il quand on veut utiliser un calcul pour définir une variable?
- `x=1+1`; `echo $x` donne ... `1+1`;
- On doit utiliser `x=${1+1}` (anciennes versions de bash) ou `x=$((1+1))` (nouvelles versions de bash);

Pas de types

- Les variables en bash n'ont pas de type ;
- Elles peuvent être vues comme des chaînes, sur lesquelles on peut parfois faire des calculs ;
- Que se passe-t-il quand on veut utiliser un calcul pour définir une variable ?
- `x=1+1`; `echo $x` donne ... 1+1 ;
- On doit utiliser `x=${1+1}` (anciennes versions de bash) ou `x=$((1+1))` (nouvelles versions de bash) ;



bash ne gère pas les nombres réels :
`x=$((1+1.5))` provoque une erreur ! Il faut utiliser les programmes `bc` ou `dc`.

Pas de types ... mais ...

- On peut quand même tricher un peu pour les types ;

Pas de types ... mais ...

- On peut quand même tricher un peu pour les types ;
- La commande `declare` - i `var=42` force `var` à être entière ;

Pas de types ... mais ...

- On peut quand même tricher un peu pour les types ;
- La commande `declare` - i `var=42` force `var` à être entière ;
- Donc `var+=1` fait passer sa valeur à 43 ;

Pas de types ... mais ...

- On peut quand même tricher un peu pour les types ;
- La commande `declare -i var=42` force `var` à être entière ;
- Donc `var+=1` fait passer sa valeur à 43 ;
- On peut aussi créer des constantes : `declare -r pi=3.14`
(pour *read-only*)

Pas de types ... mais ...

- On peut quand même tricher un peu pour les types ;
- La commande `declare -i var=42` force `var` à être entière ;
- Donc `var+=1` fait passer sa valeur à 43 ;
- On peut aussi créer des constantes : `declare -r pi=3.14`
(pour *read-only*)
- `pi+=1` provoquera donc une erreur ;

Pas de types ... mais ...

- On peut quand même tricher un peu pour les types ;
- La commande `declare -i var=42` force `var` à être entière ;
- Donc `var+=1` fait passer sa valeur à 43 ;
- On peut aussi créer des constantes : `declare -r pi=3.14`
(pour *read-only*)
- `pi+=1` provoquera donc une erreur ;



La commande `typeset` est un synonyme de `declare` et fonctionne aussi sous d'autres *shells* (par exemple `ksh`).

Opérateurs et formes condensées

- Les opérations que vous connaissez sont disponibles (+, -, *, /, %...);

Opérateurs et formes condensées

- Les opérations que vous connaissez sont disponibles (+, -, *, /, %...);
- Les formes condensées aussi (+=, -=, *=, /=, ...);

Opérateurs et formes condensées

- Les opérations que vous connaissez sont disponibles (+, -, *, /, %...);
- Les formes condensées aussi (+=, -=, *=, /=, ...);
- Attention encore une fois aux espaces!

Opérateurs et formes condensées

- Les opérations que vous connaissez sont disponibles (+, -, *, /, % ...);
- Les formes condensées aussi (+=, -=, *=, /=, ...);
- Attention encore une fois aux espaces!
C x+=1 fonctionne (attention au résultat ...);

Opérateurs et formes condensées

- Les opérations que vous connaissez sont disponibles (+, -, *, /, %...);
- Les formes condensées aussi (+=, -=, *=, /=, ...);
- Attention encore une fois aux espaces!
 - **C** `x+=1` fonctionne (attention au résultat ...);
 - **×** `x += 1` déclenche une erreur;

Opérateurs et formes condensées

- Les opérations que vous connaissez sont disponibles (+, -, *, /, % ...);
- Les formes condensées aussi (+=, -=, *=, /=, ...);
- Attention encore une fois aux espaces!
 - **C** `x+=1` fonctionne (attention au résultat ...);
 - **X** `x += 1` déclenche une erreur;



Les formes condensées ne fonctionnent pas toujours . . . pour être sûr de ne pas avoir de problème, utilisez **let**. Par exemple :

let `x*=2`

Instructions conditionnelles (**if**, **elif**, **else**)

Très semblables à Python, mais :

- 1 les tests sont encadrés par des `[[]]` **avec des espaces** ;

Instructions conditionnelles (**if**, **elif**, **else**)

Très semblables à Python, mais :

- 1 les tests sont encadrés par des `[[]]` **avec des espaces** ;
- 2 les blocs **if** et **elif** commencent par **then** ;

Instructions conditionnelles (**if**, **elif**, **else**)

Très semblables à Python, mais :

- 1 les tests sont encadrés par des `[[]]` **avec des espaces** ;
- 2 les blocs **if** et **elif** commencent par **then** ;
- 3 la fin d'un **if** doit être marquée par un **fi** ;

Instructions conditionnelles (**if**, **elif**, **else**)

Très semblables à Python, mais :

- 1 les tests sont encadrés par des `[[]]` **avec des espaces** ;
- 2 les blocs **if** et **elif** commencent par **then** ;
- 3 la fin d'un **if** doit être marquée par un **fi** ;

En Python

```
if test_1:  
    # bloc if  
elif test_2:  
    # bloc elif  
else:  
    # bloc else
```

Instructions conditionnelles (**if**, **elif**, **else**)

Très semblables à Python, mais :

- 1 les tests sont encadrés par des `[[]]` **avec des espaces** ;
- 2 les blocs **if** et **elif** commencent par **then** ;
- 3 la fin d'un **if** doit être marquée par un **fi** ;

En Python

```
if test_1:
    # bloc if
elif test_2:
    # bloc elif
else:
    # bloc else
```

En bash

```
if [[ test_1 ]]
then
    # bloc if
elif [[ test_2 ]]
then
    # bloc elif
else
    # bloc else
fi
```


Comparaisons

Les comparaisons renvoient 0 ou 1 (au lieu de **False** ou **True**) et se font à l'aide des opérateurs suivants :

Comparaison	Opérateur
égalité	\$a == \$b
différence	\$a != \$b
est inférieur	\$a < \$b
est inférieur ou égal	\$a <= \$b
est supérieur	\$a > \$b
est supérieur ou égal	\$a >= \$b

On peut combiner les conditions avec les opérateurs logiques :

- **and**, qu'on écrit &&;
- **or**, qu'on écrit || ;
- **not**, qu'on écrit ! ;



N'oubliez pas d'encadrer les espaces autour des opérateurs !

Boucles **for**

Les **for** fonctionnent comme en C et en Python, mais il faut préciser le début du bloc (et la fin) avec **do** (et **done**).

En Python

```
for x in iterable:  
    # bloc for  
    # suite
```

Boucles **for**

Les **for** fonctionnent comme en C et en Python, mais il faut préciser le début du bloc (et la fin) avec **do** (et **done**).

En Python

```
for x in iterable:  
    # bloc for  
    # suite
```

En bash

```
for x in iterable  
do  
    # bloc for  
done
```

Boucles **for**

Les **for** fonctionnent comme en C et en Python, mais il faut préciser le début du bloc (et la fin) avec **do** (et **done**).

En Python

```
for x in iterable:  
    # bloc for  
# suite
```

En Python

```
for i in range(1, n):  
    # bloc for  
# suite
```

En bash

```
for x in iterable  
do  
    # bloc for  
done
```

Boucles **for**

Les **for** fonctionnent comme en C et en Python, mais il faut préciser le début du bloc (et la fin) avec **do** (et **done**).

En Python

```
for x in iterable:  
    # bloc for  
# suite
```

En Python

```
for i in range(1, n):  
    # bloc for  
# suite
```

En bash

```
for x in iterable  
do  
    # bloc for  
done
```

En bash

```
for i in $(seq n-1)  
do  
    # bloc for  
done
```

Boucles **for**

Les **for** fonctionnent comme en C et en Python, mais il faut préciser le début du bloc (et la fin) avec **do** (et **done**).

En Python

```
for x in iterable:  
    # bloc for  
# suite
```

En Python

```
for i in range(1, n):  
    # bloc for  
# suite
```

En C

```
for (int i=0; i<n; i++) {  
    // bloc for  
}
```

En bash

```
for x in iterable  
do  
    # bloc for  
done
```

En bash

```
for i in $(seq n-1)  
do  
    # bloc for  
done
```

Boucles **for**

Les **for** fonctionnent comme en C et en Python, mais il faut préciser le début du bloc (et la fin) avec **do** (et **done**).

En Python

```
for x in iterable:  
    # bloc for  
# suite
```

En Python

```
for i in range(1, n):  
    # bloc for  
# suite
```

En C

```
for (int i=0; i<n; i++) {  
    // bloc for  
}
```

En bash

```
for x in iterable  
do  
    # bloc for  
done
```

En bash

```
for i in $(seqn-1)  
do  
    # bloc for  
done
```

En bash

```
for ((i=0; i<n; i++))  
do  
    # bloc for  
done
```

Boucles **while**

Les **while** fonctionnent comme en C et en Python, mais il faut préciser le début du bloc (et la fin) avec **do** (et **done**).

En Python

```
while condition:  
    # bloc while  
# suite
```

Note : il existe aussi des boucles **until**.

Boucles **while**

Les **while** fonctionnent comme en C et en Python, mais il faut préciser le début du bloc (et la fin) avec **do** (et **done**).

En Python

```
while condition:  
    # bloc while  
# suite
```

En bash

```
while [[ condition ]]  
do  
    # bloc while  
done
```

Note : il existe aussi des boucles **until**.

Fonctions

On peut aussi créer des fonctions :

Fonctions

On peut aussi créer des fonctions :

En Python

```
def ma_fonction():  
    # code fonction  
    return variable # facultatif
```

```
# appel de la fonction  
ma_fonction()
```

Fonctions

On peut aussi créer des fonctions :

En Python

```
def ma_fonction():  
    # code fonction  
    return variable # facultatif
```

```
# appel de la fonction  
ma_fonction()
```

En bash

```
ma_fonction() {  
    # code fonction  
    return $variable # facultatif  
}
```

```
# appel de la fonction  
ma_fonction
```

Fonctions

On peut aussi créer des fonctions :

En Python

```
def ma_fonction():  
    # code fonction  
    return variable # facultatif
```

```
# appel de la fonction  
ma_fonction()
```

En bash

```
ma_fonction() {  
    # code fonction  
    return $variable # facultatif  
}
```

```
# appel de la fonction  
ma_fonction
```



Pas de parenthèses en bash pour les appels de fonction !

Paramètres de fonction

- Les fonctions peuvent avoir des paramètres ...

Paramètres de fonction

- Les fonctions peuvent avoir des paramètres ...
- ... mais on ne les déclare pas !

Paramètres de fonction

- Les fonctions peuvent avoir des paramètres ...
- ... mais on ne les déclare pas !

```
saluer() {  
    echo "Bonjour $1 $2! Comment allez-vous?"  
}
```

```
# appel de la fonction  
ma_fonction monsieur Pingouin
```


Paramètres de fonction

- Les fonctions peuvent avoir des paramètres ...
- ... mais on ne les déclare pas !

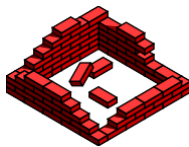
Exemple

```
saluer() {  
    echo "Bonjour $1 $2! Comment allez-vous?"  
}
```

```
# appel de la fonction  
ma_fonction monsieur Pingouin
```

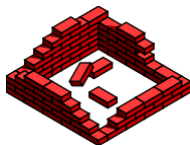
- Dans la fonction, `$i` est le i-ème paramètre de la fonction ;

Réutilisation de fonctions



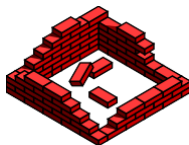
- La plupart des langages de programmation permettent de "recycler" du code) l'aide de modules :

Réutilisation de fonctions



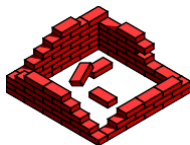
- La plupart des langages de programmation permettent de "recycler" du code) l'aide de modules :
 - C(++) : *#include "fichier.h" ;*

Réutilisation de fonctions



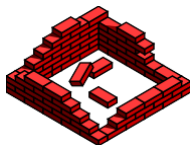
- La plupart des langages de programmation permettent de "recycler" du code) l'aide de modules :
 - C(++) : `#include "fichier.h" ;`
 - Python : `import fichier ;`

Réutilisation de fonctions



- La plupart des langages de programmation permettent de "recycler" du code) l'aide de modules :
 - C(++) : `#include "fichier.h" ;`
 - Python : `import fichier ;`
 - ...

Réutilisation de fonctions



- La plupart des langages de programmation permettent de "recycler" du code) l'aide de modules :
 - C(++) : `#include "fichier.h" ;`
 - Python : `import fichier ;`
 - ...
- bash ne propose pas ce mécanisme, mais on peut bricoler pour l'obtenir ;

Réutilisation de fonctions

On définit nos fonctions dans `fichier.sh` qu'on chargera dans `programme.sh` avec **source** :

fichier.sh

```
# définitions des fonctions
fonction1() {
    # ...
}
fonction2() {
    # ...
}

# rendre visibles les noms des
# fonctions
export -f fonction1
export -f fonction2
```

programme.sh

```
source fichier.sh

# fonction1 et fonction2 sont
# maintenant utilisables dans
# programme.sh

export est indispensable pour
que les fonctions soient "visibles"
dans programme.sh !
```

Opérations sur les chaînes

Bash	Résultat
<code>\${#chaine}</code>	la longueur de <code>\$chaine</code>
<code>\${chaine:i}</code>	le suffixe de <code>\$chaine</code> démarrant en <code>\$i</code>
<code>\${chaine:i:k}</code>	le suffixe de longueur <code>k</code> de <code>\$chaine</code> démarrant en <code>\$i</code>
<code>\${chaine#chaine2}</code>	retire <code>\$chaine2</code> de <code>\$chaine</code> si c'est un préfixe
<code>\${chaine%chaine2}</code>	retire <code>\$chaine2</code> de <code>\$chaine</code> si c'est un suffixe

Les opérations suivantes remplacent dans chaîne :

- `${chaine/avant/après}` : la première occurrence de `$avant` par `$après` ;

Opérations sur les chaînes

Bash	Résultat
<code>\${#chaine}</code>	la longueur de <code>\$chaine</code>
<code>\${chaine:i}</code>	le suffixe de <code>\$chaine</code> démarrant en <code>\$i</code>
<code>\${chaine:i:k}</code>	le suffixe de longueur k de <code>\$chaine</code> démarrant en <code>\$i</code>
<code>\${chaine#chaine2}</code>	retire <code>\$chaine2</code> de <code>\$chaine</code> si c'est un préfixe
<code>\${chaine%chaine2}</code>	retire <code>\$chaine2</code> de <code>\$chaine</code> si c'est un suffixe

Les opérations suivantes remplacent dans chaîne :

- `${chaine/avant/après}` : la première occurrence de `$avant` par `$après` ;
- `${chaine//avant/après}` : **toutes** les occurrences de `$avant` par `$après` ;

Opérations sur les chaînes

Bash	Résultat
<code>\${#chaine}</code>	la longueur de <code>\$chaine</code>
<code>\${chaine:i}</code>	le suffixe de <code>\$chaine</code> démarrant en <code>\$i</code>
<code>\${chaine:i:k}</code>	le suffixe de longueur k de <code>\$chaine</code> démarrant en <code>\$i</code>
<code>\${chaine#chaine2}</code>	retire <code>\$chaine2</code> de <code>\$chaine</code> si c'est un préfixe
<code>\${chaine%chaine2}</code>	retire <code>\$chaine2</code> de <code>\$chaine</code> si c'est un suffixe

Les opérations suivantes remplacent dans chaîne :

- `${chaine/avant/après}` : la première occurrence de `$avant` par `$après` ;
- `${chaine//avant/après}` : **toutes** les occurrences de `$avant` par `$après` ;
- `${chaine/#avant/après}` : `$avant` par `$après` si `$avant` est un préfixe de chaîne ;

Opérations sur les chaînes

Bash	Résultat
<code>\${#chaine}</code>	la longueur de <code>\$chaine</code>
<code>\${chaine:i}</code>	le suffixe de <code>\$chaine</code> démarrant en <code>\$i</code>
<code>\${chaine:i:k}</code>	le suffixe de longueur <code>k</code> de <code>\$chaine</code> démarrant en <code>\$i</code>
<code>\${chaine#chaine2}</code>	retire <code>\$chaine2</code> de <code>\$chaine</code> si c'est un préfixe
<code>\${chaine%chaine2}</code>	retire <code>\$chaine2</code> de <code>\$chaine</code> si c'est un suffixe

Les opérations suivantes remplacent dans chaîne :

- `${chaine/avant/après}` : la première occurrence de `$avant` par `$après` ;
- `${chaine//avant/après}` : **toutes** les occurrences de `$avant` par `$après` ;
- `${chaine/#avant/après}` : `$avant` par `$après` si `$avant` est un préfixe de chaîne ;
- `${chaine/%avant/après}` : `$avant` par `$après` si `$avant` est un suffixe de chaîne ;

Concaténation et addition

- Comme en Python, l'effet de + dépend du type des variables
...

Concaténation et addition

- Comme en Python, l'effet de + dépend du type des variables
...
- ... mais les variables en bash ne sont pas typées;

Concaténation et addition

- Comme en Python, l'effet de + dépend du type des variables
...
- ... mais les variables en bash ne sont pas typées;
- C'est donc à nous de préciser de quel + on parle;

Concaténation et addition

- Comme en Python, l'effet de + dépend du type des variables
...
- ... mais les variables en bash ne sont pas typées;
- C'est donc à nous de préciser de quel + on parle;

Exemple

```
$ x=1; y=3; x+= $y; echo $x
```

```
13
```

```
$ x=1; y=3; x=$x+$y; echo $x
```

```
1+3
```

```
$ x=1; y=3; x=$((x+y)); echo $x
```

```
4
```

Structures de données : tableaux

- bash permet de stocker des tableaux (*arrays*) comme en C ; ...
mais en plus bizarre : les indices ne doivent pas être consécutifs !

Structures de données : tableaux

- bash permet de stocker des tableaux (*arrays*) comme en C ; ...
mais en plus bizarre : les indices ne doivent pas être consécutifs !
- L'accès aux éléments se fait avec `tableau[position]` ;

Structures de données : tableaux

- bash permet de stocker des tableaux (*arrays*) comme en C ; ...
mais en plus bizarre : les indices ne doivent pas être consécutifs !
- L'accès aux éléments se fait avec `tableau[position]` ;

Exemple (initialisation d'un tableau)

On peut déclarer les éléments avec leur position :

```
tableau[0]="bonjour"
```

```
tableau[1]="tout"
```

```
tableau[2]="le"
```

```
tableau[3]="monde"
```

Ou plus simplement (attention aux parenthèses) :

```
tableau=(bonjour tout le monde)
```

Structures de données : tableaux

- bash permet de stocker des tableaux (*arrays*) comme en C ; ... mais en plus bizarre : les indices ne doivent pas être consécutifs !
- L'accès aux éléments se fait avec `tableau[position]` ;

Exemple (initialisation d'un tableau)

On peut déclarer les éléments avec leur position :

```
tableau[0]="bonjour"
```

```
tableau[1]="tout"
```

```
tableau[2]="le"
```

```
tableau[3]="monde"
```

Ou plus simplement (attention aux parenthèses) :

```
tableau=(bonjour tout le monde)
```

- Attention, consulter une valeur nécessite des accolades (voir la suite) !

Structures de données : tableaux

- On peut aussi initialiser le tableau avec le résultat d'une commande ;

Structures de données : tableaux

- On peut aussi initialiser le tableau avec le résultat d'une commande ;

Exemple

```
$ aujourd'hui=$(date -I | sed s/-/' /g))
```

```
$ echo ${myarray[0]}
```

```
2021
```

```
$ echo ${myarray[1]}
```

```
10
```

```
$ echo ${myarray[2]}
```

```
22
```

Structures de données : tableaux

- On peut aussi initialiser le tableau avec le résultat d'une commande ;

Exemple

```
$ aujourd'hui=$(date -I | sed s/-/' '/g))
```

```
$ echo ${myarray[0]}
```

```
2021
```

```
$ echo ${myarray[1]}
```

```
10
```

```
$ echo ${myarray[2]}
```

```
22
```

- Ceci met l'année, le mois et puis le jour dans le tableau aujourd'hui ;

Structures de données : tableaux

- Très utile : on peut facilement mettre chaque "mot" d'un fichier dans un tableau !

```
$ mots=$($(< fichier.txt))
```

Structures de données : tableaux

- Très utile : on peut facilement mettre chaque "mot" d'un fichier dans un tableau !

```
$ mots=$(< fichier.txt)
```

- Les espaces sont considérés comme les séparateurs des mots et disparaissent ;

Structures de données : tableaux

- Très utile : on peut facilement mettre chaque "mot" d'un fichier dans un tableau !

```
$ mots=$($(< fichier.txt))
```

- Les espaces sont considérés comme les séparateurs des mots et disparaissent ;
- Si l'on veut les **lignes** du fichier :

```
$ readarray -t lignes < fichier.txt
```

Les opérateurs @ et !

- Pour accéder à l'entièreté du tableau, on utilise l'opérateur @;

Les opérateurs @ et !

- Pour accéder à l'entièreté du tableau, on utilise l'opérateur @;
× `echo ${myarray}` n'affiche que `myarray[0]` (s'il existe !)

Les opérateurs @ et !

- Pour accéder à l'entièreté du tableau, on utilise l'opérateur @;
 - × `echo ${myarray}` n'affiche que `myarray[0]` (s'il existe !)
 - C `echo ${myarray[@]}` affiche tout le tableau ;

Les opérateurs @ et !

- Pour accéder à l'entièreté du tableau, on utilise l'opérateur @;
 - × `echo ${myarray}` n'affiche que `myarray[0]` (s'il existe !)
 - C `echo ${myarray[@]}` affiche tout le tableau ;
- Si l'on veut connaître l'ensemble des **positions** valides du tableau, on rajoute l'opérateur ! devant son nom ;
Exemple : `echo ${!myarray[@]}`

Les opérateurs @ et !

- Pour accéder à l'entièreté du tableau, on utilise l'opérateur @;
 - × `echo ${myarray}` n'affiche que `myarray[0]` (s'il existe !)
 - C `echo ${myarray[@]}` affiche tout le tableau ;
- Si l'on veut connaître l'ensemble des **positions** valides du tableau, on rajoute l'opérateur ! devant son nom ;
Exemple : `echo ${!myarray[@]}`
- Enfin, le nombre d'éléments d'un tableau s'obtient à l'aide de `${#tableau[@]}` ;

Itérer sur des tableaux

Ces opérateurs nous permettent de simuler les boucles vues en Python :

En Python

```
for elem in iterable:  
    print(elem)
```

Itérer sur des tableaux

Ces opérateurs nous permettent de simuler les boucles vues en Python :

En Python

```
for elem in iterable:  
    print(elem)
```

En bash

```
for elem in ${iterable[@]}  
do  
    echo $elem  
done
```


Itérer sur des tableaux

Ces opérateurs nous permettent de simuler les boucles vues en Python :

En Python

```
for elem in iterable:
    print(elem)
```

En Python

```
for i in range(len(iterable)):
    print(i, iterable[i])
```

En bash

```
for elem in ${iterable[@]}
do
    echo $elem
done
```

Itérer sur des tableaux

Ces opérateurs nous permettent de simuler les boucles vues en Python :

En Python

```
for elem in iterable:
    print(elem)
```

En bash

```
for elem in ${iterable[@]}
do
    echo $elem
done
```

En Python

```
for i in range(len(iterable)):
    print(i, iterable[i])
```

En bash

```
for i in ${!iterable[@]}
do
    echo $i ${iterable[$i]}
done
```

Manipulation basique de fichiers

On devra fréquemment vérifier des propriétés de nos fichiers. Voici les tests les plus fréquents :

Expression	Signification
-e nom	le fichier (normal, répertoire, ...) nom existe
-d nom	le répertoire nom existe
-f nom	le fichier "normal" existe
-r nom	le fichier nom est lisible
-w filename	le fichier nom est modifiable
-x filename	le fichier nom est exécutable

Utilisation typique

```
if [[ -lettre nom ]]
then
    # du code
fi
```

Paramètres

- Les scripts peuvent prendre des paramètres, accessibles par leur position et \$:

Paramètres

- Les scripts peuvent prendre des paramètres, accessibles par leur position et \$:
 - \$0 est le nom du script ;

Paramètres

- Les scripts peuvent prendre des paramètres, accessibles par leur position et \$:
 - \$0 est le nom du script ;
 - \$1 est le premier paramètre ;

Paramètres

- Les scripts peuvent prendre des paramètres, accessibles par leur position et \$:
 - \$0 est le nom du script ;
 - \$1 est le premier paramètre ;
 - \$2 est le deuxième paramètre ;

Paramètres

- Les scripts peuvent prendre des paramètres, accessibles par leur position et \$:
 - \$0 est le nom du script ;
 - \$1 est le premier paramètre ;
 - \$2 est le deuxième paramètre ;
 - ...

Paramètres

- Les scripts peuvent prendre des paramètres, accessibles par leur position et \$:
 - \$0 est le nom du script ;
 - \$1 est le premier paramètre ;
 - \$2 est le deuxième paramètre ;
 - ...
- Attention : après \$9, on doit utiliser des accolades (donc \${10}, \${11}, etc.)

Paramètres

- Les scripts peuvent prendre des paramètres, accessibles par leur position et \$:
 - \$0 est le nom du script ;
 - \$1 est le premier paramètre ;
 - \$2 est le deuxième paramètre ;
 - ...
- Attention : après \$9, on doit utiliser des accolades (donc \${10}, \${11}, etc.)
- Il y a aussi quelques variables spéciales :

Paramètres

- Les scripts peuvent prendre des paramètres, accessibles par leur position et \$:
 - \$0 est le nom du script ;
 - \$1 est le premier paramètre ;
 - \$2 est le deuxième paramètre ;
 - ...
- Attention : après \$9, on doit utiliser des accolades (donc \${10}, \${11}, etc.)
- Il y a aussi quelques variables spéciales :
 - \$# est le nombre de paramètres du script ;

Paramètres

- Les scripts peuvent prendre des paramètres, accessibles par leur position et \$:
 - \$0 est le nom du script ;
 - \$1 est le premier paramètre ;
 - \$2 est le deuxième paramètre ;
 - ...
- Attention : après \$9, on doit utiliser des accolades (donc \${10}, \${11}, etc.)
- Il y a aussi quelques variables spéciales :
 - \$# est le nombre de paramètres du script ;
 - \$* et @\$ contiennent tous les paramètres du script ;