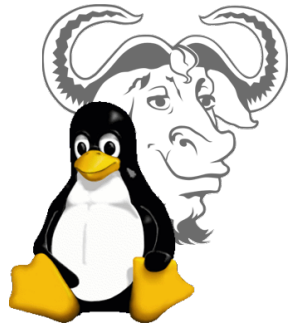
Administration d'un système GNU / Linux

07 — Scripts bash (2/2)

Anthony Labarre

上海师范大学

29 octobre 2024



Plan d'aujourd'hui

1. Fonctions
2. Chaînes
3. Tableaux
4. Fichiers

Fonctions

Fonctions

On peut créer des fonctions en `bash`:

Fonctions

On peut créer des fonctions en bash:

En Python

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


# appel de la fonction
ma_fonction()
```

Fonctions

On peut créer des fonctions en bash:

En Python

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


# appel de la fonction
ma_fonction()
```

En bash

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

# appel de la fonction
ma_fonction
```

Fonctions

On peut créer des fonctions en bash:

En Python

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


# appel de la fonction
ma_fonction()
```

En bash

```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!

Exemple

```
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;

Exemple: saluer un utilisateur existant (ou non)

Examinons un exemple simple:

./src/fonctions.sh

```
1  #!/usr/bin/env bash
2  saluer() {
3      user=$1
4      echo "Bonjour $user!"
5      if [[ $(grep ^$user /etc/passwd | cut -f1 -d:) == $user ]]
6      then
7          echo "Je vous ai trouvé dans /etc/passwd ."
8      else
9          echo "Je ne vous ai pas trouvé dans /etc/passwd ."
10     fi
11 }
12
13 saluer anthony  # existe sur ma machine
14 echo
15 saluer pingouin # n'existe pas sur ma machine
```

Paramètres de fonctions et paramètres de scripts

Attention, la syntaxe pour accéder aux paramètres des fonctions et ceux des scripts est la même!

Comment ne pas confondre les deux?

Paramètres de fonctions et paramètres de scripts

Attention, la syntaxe pour accéder aux paramètres des fonctions et ceux des scripts est la même!

Comment ne pas confondre les deux?

1. dans une fonction, `$i` est le i-ème paramètre de la fonction;
2. hors d'une fonction, `$i` est le i-ème paramètre du script;

On ne peut pas fixer le nombre d'arguments d'une fonction: si elle en reçoit "trop", elle ne saura pas qu'il s'agit de paramètres du script.

Chaînes

Opérations sur les chaînes

Syntaxe	Résultat

Opérations sur les chaînes

Syntaxe	Résultat
`${#chaine}`	la longueur de `$chaine`

Opérations sur les chaînes

Syntaxe	Résultat
`${#chaine}`	la longueur de `$chaine`
`${chaine:i}`	le suffixe de `$chaine` démarrant en `$i`

Opérations sur les chaînes

Syntaxe	Résultat
`${#chaine}`	la longueur de `$chaine`
`${chaine:i}`	le suffixe de `$chaine` démarrant en `$i`
`${chaine:i:k}`	le suffixe de longueur `k` de `$chaine` démarrant en `$i`

Opérations sur les chaînes

Syntaxe	Résultat
`${#chaine}`	la longueur de `$chaine`
`${chaine:i}`	le suffixe de `$chaine` démarrant en `$i`
`${chaine:i:k}`	le suffixe de longueur `k` de `$chaine` démarrant en `$i`
`${chaine#chaine2}`	retire `$chaine2` de `$chaine` si c'est un préfixe

Opérations sur les chaînes

Syntaxe	Résultat
`${#chaine}`	la longueur de `$chaine`
`${chaine:i}`	le suffixe de `$chaine` démarrant en `$i`
`${chaine:i:k}`	le suffixe de longueur `k` de `$chaine` démarrant en `$i`
`${chaine#chaine2}`	retire `$chaine2` de `$chaine` si c'est un préfixe
`${chaine%chaine2}`	retire `$chaine2` de `$chaine` si c'est un suffixe

Opérations sur les chaînes

Syntaxe	Résultat
`${#chaine}`	la longueur de `$chaine`
`${chaine:i}`	le suffixe de `$chaine` démarrant en `$i`
`${chaine:i:k}`	le suffixe de longueur `k` de `$chaine` démarrant en `$i`
`${chaine#chaine2}`	retire `$chaine2` de `$chaine` si c'est un préfixe
`${chaine%chaine2}`	retire `$chaine2` de `$chaine` si c'est un suffixe

Exemple

```
$ x="bonjour"
$ echo "$x possède ${#x} lettres"
bonjour possède 7 lettres
```

Opérations sur les chaînes

Syntaxe	Résultat
`${#chaine}`	la longueur de `$chaine`
`${chaine:i}`	le suffixe de `$chaine` démarrant en `$i`
`${chaine:i:k}`	le suffixe de longueur `k` de `$chaine` démarrant en `$i`
`${chaine#chaine2}`	retire `$chaine2` de `$chaine` si c'est un préfixe
`${chaine%chaine2}`	retire `$chaine2` de `$chaine` si c'est un suffixe

Exemple

```
$ x="bonjour"
$ echo "$x possède ${#x} lettres"
bonjour possède 7 lettres

$ echo "Les lettres de $x à partir de la position 3 sont ${x:3}"
Les lettres de bonjour à partir de la position 3 sont jour
```

Opérations sur les chaînes

Syntaxe	Résultat
`${#chaine}`	la longueur de `$chaine`
`${chaine:i}`	le suffixe de `$chaine` démarrant en `$i`
`${chaine:i:k}`	le suffixe de longueur `k` de `$chaine` démarrant en `$i`
`${chaine#chaine2}`	retire `$chaine2` de `$chaine` si c'est un préfixe
`${chaine%chaine2}`	retire `$chaine2` de `$chaine` si c'est un suffixe

Exemple

```
$ x="bonjour"
$ echo "$x possède ${#x} lettres"
bonjour possède 7 lettres

$ echo "Les lettres de $x à partir de la position 3 sont ${x:3}"
Les lettres de bonjour à partir de la position 3 sont jour

$ echo "Les 3 premières lettres de $x sont ${x:0:3}"
Les 3 premières lettres de bonjour sont bon
```

Opérations sur les chaînes

Syntaxe	Résultat
`${#chaine}`	la longueur de `$chaine`
`${chaine:i}`	le suffixe de `$chaine` démarrant en `$i`
`${chaine:i:k}`	le suffixe de longueur `k` de `$chaine` démarrant en `$i`
`${chaine#chaine2}`	retire `$chaine2` de `$chaine` si c'est un préfixe
`${chaine%chaine2}`	retire `$chaine2` de `$chaine` si c'est un suffixe

Exemple

```
$ x="bonjour"
$ echo "$x possède ${#x} lettres"
bonjour possède 7 lettres

$ echo "Les lettres de $x à partir de la position 3 sont ${x:3}"
Les lettres de bonjour à partir de la position 3 sont jour

$ echo "Les 3 premières lettres de $x sont ${x:0:3}"
Les 3 premières lettres de bonjour sont bon

$ echo "\${$x#bon} = ${x#bon}"
${bonjour#bon} = jour
```

Opérations sur les chaînes

Syntaxe	Résultat
`${#chaine}`	la longueur de `$chaine`
`${chaine:i}`	le suffixe de `$chaine` démarrant en `$i`
`${chaine:i:k}`	le suffixe de longueur `k` de `$chaine` démarrant en `$i`
`${chaine#chaine2}`	retire `$chaine2` de `$chaine` si c'est un préfixe
`${chaine%chaine2}`	retire `$chaine2` de `$chaine` si c'est un suffixe

Exemple

```
$ x="bonjour"
$ echo "$x possède ${#x} lettres"
bonjour possède 7 lettres

$ echo "Les lettres de $x à partir de la position 3 sont ${x:3}"
Les lettres de bonjour à partir de la position 3 sont jour

$ echo "Les 3 premières lettres de $x sont ${x:0:3}"
Les 3 premières lettres de bonjour sont bon

$ echo "\${$x#bon} = ${x#bon}"
${bonjour#bon} = jour

$ echo "\${$x%jour} = ${x%jour}"
${bonjour%jour} = bon
```

Remplacement de texte

Syntaxe	Remplace `avant` par `après` ...

Remplacement de texte

Syntaxe	Remplace `avant` par `après` ...
`${chaine/avant/après}`	première occurrence uniquement

Remplacement de texte

Syntaxe	Remplace `avant` par `après` ...
`${chaine/avant/après}`	première occurrence uniquement
`${chaine//avant/après}`	**toutes** les occurrences

Remplacement de texte

Syntaxe	Remplace `avant` par `après` ...
`${chaine/avant/après}`	première occurrence uniquement
`${chaine//avant/après}`	**toutes** les occurrences
`${chaine/#avant/après}`	si `$avant` est un préfixe de `chaine`

Remplacement de texte

Syntaxe	Remplace `avant` par `après` ...
`${chaine/avant/après}`	première occurrence uniquement
`${chaine//avant/après}`	**toutes** les occurrences
`${chaine/#avant/après}`	si `$avant` est un préfixe de `chaine`
`${chaine/%avant/après}`	si `$avant` est un suffixe de `chaine`

Remplacement de texte

Syntaxe	Remplace `avant` par `après` ...
`${chaine/avant/après}`	première occurrence uniquement
`${chaine//avant/après}`	**toutes** les occurrences
`${chaine/#avant/après}`	si `$avant` est un préfixe de `chaine`
`${chaine/%avant/après}`	si `$avant` est un suffixe de `chaine`

Exemple

```
$ x="bonbon"
$ echo ${x/bon/mal}
malbon
```

Remplacement de texte

Syntaxe	Remplace `avant` par `après` ...
`${chaine/avant/après}`	première occurrence uniquement
`${chaine//avant/après}`	**toutes** les occurrences
`${chaine/#avant/après}`	si `$avant` est un préfixe de `chaine`
`${chaine/%avant/après}`	si `$avant` est un suffixe de `chaine`

Exemple

```
$ x="bonbon"
$ echo ${x/bon/mal}
malbon
$ echo ${x//bon/mal}
malmal
```

Remplacement de texte

Syntaxe	Remplace `avant` par `après` ...
`${chaine/avant/après}`	première occurrence uniquement
`${chaine//avant/après}`	**toutes** les occurrences
`${chaine/#avant/après}`	si `$avant` est un préfixe de `chaine`
`${chaine/%avant/après}`	si `$avant` est un suffixe de `chaine`

Exemple

```
$ x="bonbon"
$ echo ${x/bon/mal}
malbon
$ echo ${x//bon/mal}
malmal
$ echo ${x/#bon/jam}
jambon
```

Remplacement de texte

Syntaxe	Remplace avant par après ...
`${chaine/avant/après}`	première occurrence uniquement
`${chaine//avant/après}`	**toutes** les occurrences
`${chaine/#avant/après}`	si `$avant` est un préfixe de `chaine`
`${chaine/%avant/après}`	si `$avant` est un suffixe de `chaine`

Exemple

```
$ x="bonbon"
$ echo ${x/bon/mal}
malbon
$ echo ${x//bon/mal}
malmal
$ echo ${x/#bon/jam}
jambon
$ echo ${x/%bon/heur}
bonheur
```

Comparaisons de chaînes

On peut également utiliser les tests suivants:

- `[[ $x == y* ]]`: est-ce que la chaîne `x` commence par “y”?
- `[[ $x == *y ]]`: est-ce que la chaîne `x` se termine par “y”?

Tableaux

Structures de données: tableaux

- `bash` permet de stocker des tableaux (*arrays*);

Structures de données: tableaux

- `bash` permet de stocker des tableaux (*arrays*);

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)
```

Tableaux: initialisations alternatives

Il est souvent utile d'initialiser les tableaux différemment:

À partir d'une commande

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

Tableaux: initialisations alternatives

Il est souvent utile d'initialiser les tableaux différemment:

À partir d'une commande

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

À partir d'un fichier (1 mot $\mapsto$ 1 élément))

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

Tableaux: initialisations alternatives

Il est souvent utile d'initialiser les tableaux différemment:

À partir d'une commande

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

À partir d'un fichier (1 mot $\mapsto$ 1 élément))

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

À partir d'un fichier (1 ligne $\mapsto$ 1 élément))

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

Tableaux: accès aux éléments

Accéder aux éléments se fait comme dans les autres langages, mais on doit encadrer la variable avec `{nom}`:

Exemple

```
$ aujourdhui=($(date -I | sed s/-/' '/g))
$ echo ${aujourdhui[0]}
2021
$ echo ${aujourdhui[1]}
10
$ echo ${aujourdhui[2]}
22
```

L'opérateur @

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

- × `echo ${aujourdhui}` affiche seulement le premier élément;
- ✓ `echo ${aujourdhui[@]}` affiche tout le tableau;

Le nombre d'éléments d'un tableau s'obtient à l'aide de `${#tableau[@]}`.

L'opérateur @

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

- × `echo ${aujourdhui}` affiche seulement le premier élément;
- ✓ `echo ${aujourdhui[@]}` affiche tout le tableau;

Le nombre d'éléments d'un tableau s'obtient à l'aide de `${#tableau[@]}`.

Exemple

```
$ aujourdhui=($(date -I | sed s/-/' '/g))
$ echo $aujourdhui
2021
$ echo ${aujourdhui}
2021
$ echo ${aujourdhui[@]}
2021 10 22
$ echo ${#aujourdhui[@]}
3
```

L'opérateur !

`bash` autorise les indices non consécutifs!

L'opérateur !

`bash` autorise les indices non consécutifs!

Il n'y a aucune garantie qu'un tableau de taille `n` soit indicé de `0` à `n-1`.

L'opérateur !

`bash` autorise les indices non consécutifs!

Il n'y a aucune garantie qu'un tableau de taille `n` soit indicé de `0` à `n-1`. Pour connaître les indices valides, on utilise `${!tableau[@]}`.

L'opérateur !

`bash` autorise les indices non consécutifs!

Il n'y a aucune garantie qu'un tableau de taille `n` soit indicé de `0` à `n-1`. Pour connaître les indices valides, on utilise `${!tableau[@]}`.

Exemple

```
$ x[2]="bonjour"
$ x[7]="tout"
$ x[1]="le"
$ x[8]="monde"
$ echo ${!x[@]}
1 2 7 8
$ unset x[2]
$ echo ${!x[@]}
1 7 8
```

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 Python

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

En bash

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

En bash

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

Copie de tableaux

On ne peut pas copier les tableaux avec une simple affectation!

Il faut utiliser la syntaxe `y=${x[@]}`.

Exemple

```
$ x=(1 2 3)
$ y=$x          # non
$ echo ${y[@]}
1
$ y=${x[@]}   # oui
$ echo ${y[@]}
1 2 3
```

Fichiers

Manipulation basique de fichiers

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

Expression	Signification

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

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

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

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

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 nom`	le fichier `nom` est modifiable

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 nom`	le fichier `nom` est modifiable
`-x nom`	le fichier `nom` est exécutable

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 nom`	le fichier `nom` est modifiable
`-x nom`	le fichier `nom` est exécutable

Utilisation typique

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