

04A : Bash, syntaxe et structures de contrôle

420-3A6-ST : Programmation système

Dany Gagnon

Cégep de Saint-Jean-sur-Richelieu

Lundi 14 septembre 2026

Le parcours de la séance

- ① Définir Bash et exécuter un script.
- ② Utiliser des variables et recevoir des arguments.
- ③ Choisir avec `if`, répéter avec `for` et `while`.
- ④ Définir et appeler une fonction.
- ⑤ Appliquer ces constructions aux fichiers et à l'environnement.

Qu'est-ce que Bash ?

Bash est un interpréteur de commandes, aussi appelé *shell*.

- Il lit une commande, interprète sa syntaxe et l'exécute.
- Il possède un langage : variables, conditions, boucles et fonctions.
- Il peut lire les commandes au clavier ou dans un fichier.

Le **terminal** permet de saisir et d'afficher du texte. Le **shell** interprète ce texte.

La forme d'une commande

commande options arguments

```
ls -l "mes documents"
```

- `ls` : nom de la commande.
- `-l` : option qui demande un affichage détaillé.
- `"mes documents"` : un seul argument grâce aux guillemets.

Les espaces séparent les mots. Les options possibles dépendent de la commande.

Un script est un fichier de commandes

Contenu de `bonjour.sh` :

```
#!/usr/bin/env bash
# Une ligne qui commence par # est un commentaire.
printf 'Bonjour !\n'
pwd
```

Les commandes s'exécutent dans l'ordre. La première ligne, appelée *shebang*, indique l'interpréteur lors d'une exécution directe.

Le shebang choisit l'interpréteur

La première ligne commence par `#!`, dès le premier caractère du fichier :

```
#!/usr/bin/env bash
```

- Lors de `./bonjour.sh`, le système utilise cette ligne pour lancer le bon programme.
- `/usr/bin/env` cherche ici `bash` dans les répertoires de `PATH`.
- Avec `#!/bin/bash`, le chemin de Bash est fixé directement.

Le suffixe `.sh` aide à reconnaître le fichier ; il ne choisit pas l'interpréteur.

Deux façons de lancer le script

Choisir explicitement Bash ; le fichier doit être lisible, sans exiger son droit `x` :

```
bash bonjour.sh
```

Utiliser le shebang ; le script doit être lisible et exécutable :

```
chmod u+x bonjour.sh  
./bonjour.sh
```

`./` désigne le répertoire courant. Pour nos scripts Bash, utilisez `bash` ou leur shebang, car `sh` peut désigner un autre shell.

Définir une variable et lire sa valeur

Une variable associe un nom à une valeur.

```
prenom=Alex  
message="Bonjour $prenom"  
printf '%s\n' "$message"
```

- Affectation : `nom=valeur`, sans espace autour de `=`.
- Lecture : `$nom` ou `${nom}`.
- `printf '%s\n'` affiche le texte reçu, puis un saut de ligne.

L'exemple affiche `Bonjour Alex`.

Guillemets simples et doubles

```
prenom=Alex  
printf '%s\n' 'Bonjour $prenom'  
printf '%s\n' "Bonjour $prenom"
```

Forme	Effet sur \$prenom	Texte affiché
Guillemets simples	texte conservé	Bonjour \$prenom
Guillemets doubles	valeur insérée	Bonjour Alex

Les guillemets doubles gardent une valeur comme `mes documents` dans un seul argument.

Délimiter un nom et récupérer une sortie

Les accolades indiquent où se termine le nom de la variable :

```
nom=rapport  
copie="${nom}_backup"  
printf '%s\n' "$copie"
```

`$(commande)` récupère la sortie standard d'une commande :

```
repertoire=$(pwd)  
printf 'Repertoire : %s\n' "$repertoire"
```

Lire une réponse au clavier

`read` lit une ligne et place la réponse dans une variable.

```
printf 'Votre prenom : '  
read -r prenom  
printf 'Bonjour %s\n' "$prenom"
```

- `prenom` est le nom de la variable à remplir : aucun `$` après `read`.
- `-r` conserve les antislashes tels qu'ils sont saisis.

Recevoir des arguments

Appel dans le terminal :

```
./saluer.sh Alex "Marie Claude"
```

Paramètre	Valeur dans ce script
\$0	./saluer.sh
\$1	Alex
\$2	Marie Claude
\$#	2, le nombre d'arguments
"\$@"	tous les arguments, chacun conservé séparément

Un argument est donné au lancement. Une réponse à `read` est lue pendant l'exécution.

Une commande produit un code de sortie

Le code 0 indique un succès. Un code non nul indique un échec ou un résultat particulier, selon la commande.

```
true
printf 'Code : %s\n' "$?"
false
printf 'Code : %s\n' "$?"
```

Les lignes affichent respectivement 0 et 1.

\$? contient le code de la dernière commande terminée ; une nouvelle commande le remplace.

if : choisir d'exécuter un bloc

Définition : `if` exécute un bloc si la commande de condition réussit.

```
if commande_de_condition; then
    commandes_si_succes
else
    commandes_sinon
fi
```

- `then` ouvre le bloc ; `fi` termine la structure.
- `else` est facultatif.
- Le `;` sépare la condition de `then` sur la même ligne.

Exemple : vérifier un fichier

`[[ ... ]]` évalue une expression de test en Bash.

```
chemin="rapport.txt"
if [[ -f "$chemin" ]]; then
    printf 'Fichier present\n'
else
    printf 'Ce chemin ne designe pas un fichier regulier\n'
fi
```

`-f` teste le type fichier régulier. Gardez les espaces autour de `[[` et `]]` et entre les éléments du test.

elif : ajouter un autre cas

```
chemin="documents"
if [[ -f "$chemin" ]]; then
    printf 'Fichier\n'
elif [[ -d "$chemin" ]]; then
    printf 'Repertoire\n'
else
    printf 'Autre cas\n'
fi
```

Les conditions sont examinées dans l'ordre. Seul le premier bloc dont la condition réussit est exécuté; `else` sert si aucune ne réussit.

Tests sur les fichiers et répertoires

Ces expressions se placent entre `[[` et `]]`.

Expression	Question
<code>-e "\$chemin"</code>	la cible existe-t-elle ?
<code>-f "\$chemin"</code>	est-ce un fichier régulier ?
<code>-d "\$chemin"</code>	est-ce un répertoire ?
<code>-L "\$chemin"</code>	le chemin est-il un lien symbolique ?
<code>-r "\$chemin"</code>	est-il lisible pour le processus ?
<code>-w "\$chemin"</code>	est-il accessible en écriture ?

`-f` et `-d` suivent les liens symboliques. Un lien pendant peut satisfaire `-L` mais pas `-e`.

Comparer du texte ou des entiers

Texte dans un test

- `$nom == "Alex"` : égalité.
- `$nom != "Alex"` : différence.
- `-z "$nom"` : chaîne vide.
- `-n "$nom"` : chaîne non vide.

Entiers dans un test

- `$n -eq 3` : égal à 3.
- `$n -ne 3` : différent de 3.
- `$n -lt 3`, `-le` : inférieur, inférieur ou égal.
- `$n -gt 3`, `-ge` : supérieur, supérieur ou égal.

Exemple complet : `[[ $n -ge 3 ]]`. Utilisez les comparaisons numériques avec des valeurs entières.

Combiner des tests

Opérateur	Sens
!	non : inverser le test
&&	et : les deux tests doivent réussir
	ou : au moins un test doit réussir

```
if [[ -f "$chemin" && ! -L "$chemin" ]]; then
    printf 'Fichier regulier, hors lien symbolique\n'
fi
```

Tester une commande et terminer un script

La condition peut être une commande ordinaire :

```
if mkdir "nouveau-dossier"; then
    printf 'Dossier cree\n'
    exit 0
else
    printf 'Creation impossible\n' >&2
    exit 1
fi
```

`exit` termine le script avec le code indiqué. `>&2` envoie le message sur la sortie d'erreur.

for : parcourir une liste de valeurs

Définition : le bloc se répète une fois pour chaque valeur de la liste.

```
for variable in liste_de_valeurs; do  
    commandes  
done
```

```
for couleur in rouge vert bleu; do  
    printf '%s\n' "$couleur"  
done
```

La variable `couleur` reçoit successivement les trois mots. `do` ouvre le bloc; `done` le termine.

Parcourir les arguments avec for

Dans `afficher.sh` :

```
for nom in "$@"; do
    printf '<%s>\n' "$nom"
done
```

Appel et résultat :

```
./afficher.sh Alex "Marie Claude"
<Alex>
<Marie Claude>
```

`"$@"` garde chaque argument séparé. Sans argument, la boucle n'effectue aucun tour.

Parcourir des noms avec un motif

Un **motif de noms** (*glob*) est développé par Bash avant la boucle. ***** représente une suite de caractères dans un nom.

```
for fichier in ./*.txt; do
    if [[ -f "$fichier" && ! -L "$fichier" ]]; then
        printf '%s\n' "$fichier"
    fi
done
```

Avec les réglages par défaut, ***** ignore les noms cachés. Sans correspondance, le motif reste littéral : ici, le test empêche de le traiter comme un fichier.

Calculer avec des entiers

`$((expression))` calcule une valeur entière.

```
compteur=0
compteur=$((compteur + 1))
prix=7
quantite=3
total=$((prix * quantite))
printf 'Total : %s\n' "$total"
```

Les opérateurs usuels sont +, -, *, / et %. La division est entière : `$((7 / 2))` vaut 3.

while : répéter tant que le test réussit

```
while commande_de_condition; do
    commandes
done

i=1
while [[ "$i" -le 3 ]]; do
    printf 'Tour %s\n' "$i"
    i=$((i + 1))
done
```

Le test précède chaque tour. Si la condition échoue dès le début, le bloc n'est pas exécuté.

continue et break : agir sur la boucle

- `continue` passe au tour suivant.
- `break` quitte la boucle la plus proche.

```
for mot in debut ignorer fin apres; do
  if [[ "$mot" == "ignorer" ]]; then
    continue
  fi
  if [[ "$mot" == "fin" ]]; then
    break
  fi
  printf '%s\n' "$mot"
done
```

Seul `debut` est affiché.

Définir une fonction et l'appeler

Une fonction donne un nom à un groupe de commandes réutilisable.

```
nom_fonction() {  
    commandes  
}  
  
saluer() {  
    printf 'Bonjour %s\n' "$1"  
}  
  
saluer Alex  
saluer "Marie Claude"
```

La définition n'exécute pas le corps. L'appel se fait comme une commande, avec des arguments séparés par des espaces.

Arguments et variables locales

Une fonction reçoit ses propres \$1, \$2 et "\$@".

```
nom=Alice
saluer() {
    local nom="$1"
    printf 'Bonjour %s\n' "$nom"
}
saluer Bob
printf 'Dans le script : %s\n' "$nom"
```

L'exemple affiche `Bonjour Bob`, puis `Dans le script : Alice`. `local` évite de remplacer la variable du script.

return : terminer la fonction avec un statut

```
est_fichier() {  
    if [[ -f "$1" ]]; then  
        return 0  
    fi  
    return 1  
}  
if est_fichier "rapport.txt"; then  
    printf 'Le test a reussi\n'  
fi
```

return quitte la fonction ; exit termine le script. Le statut est un nombre : pour produire du texte, utilisez printf.

Les dotfiles : un point au début du nom

Un **dotfile** est un fichier dont le nom commence par un point. Cette convention sert souvent à ranger la configuration des outils.

- `~/.bashrc` : commandes de configuration de Bash interactif.
- `~/.gitconfig` : préférences personnelles de Git.
- `~/.config/` : répertoire caché utilisé par plusieurs applications.

Le point ne change ni les permissions ni le contenu. Un fichier caché reste accessible par son chemin ; ce n'est pas une protection.

Voir les noms cachés et comprendre les motifs

```
ls "$HOME"           # noms visibles
ls -a "$HOME"        # inclut les caches, . et ..
ls -A "$HOME"        # inclut les caches, sans . et ..
cat "$HOME/.bashrc"
```

- `.` désigne le répertoire courant ; `..`, son parent.
- Par défaut, le motif Bash `*` ignore les noms commençant par un point.
- `ls -A` ne sélectionne pas seulement les dotfiles : les noms visibles restent affichés.

Afficher les fichiers cachés

Un nom commençant par un point est caché dans l'affichage habituel de `ls`.

```
ls -A "$HOME"
```

```
find "$HOME" -maxdepth 1 -type f -name '.*' -print
```

- `ls -A` affiche les noms visibles et cachés, sauf `.` et `...`
- `find` sélectionne ici seulement les fichiers réguliers cachés du premier niveau.
- Les guillemets de `'.*'` transmettent le motif à `find`.

export : transmettre une variable

L'**environnement** regroupe les variables transmises aux nouveaux processus.

```
COURS_DEMO=atelier  
export COURS_DEMO  
printenv COURS_DEMO
```

- Une affectation définit une variable du shell.
- **export** la marque pour transmission aux futures commandes externes.
- **printenv** affiche les variables de l'environnement reçu.

Enregistrer la sortie d'une commande

```
printenv > environnement.txt  
printenv COURS_DEMO > valeur.txt
```

- Sans nom, `printenv` affiche tout l'environnement.
- `>` envoie la sortie dans un fichier, créé ou remplacé.
- Le shell prépare la redirection avant d'exécuter la commande.

Les flux, les autres redirections et les pipelines seront détaillés en 04B.

Un script court : arguments, for et if

```
#!/usr/bin/env bash
for chemin in "$@"; do
    if [[ -f "$chemin" ]]; then
        printf 'FICHER %s\n' "$chemin"
    elif [[ -d "$chemin" ]]; then
        printf 'REPERTOIRE %s\n' "$chemin"
    else
        printf 'AUTRE %s\n' "$chemin"
    fi
done
bash classer.sh "rapport final.txt" documents absent
```

Atelier : modifier puis écrire

- ❶ Tester le script précédent avec un fichier, un répertoire et un chemin absent.
- ❷ Refuser un appel sans argument avec un usage et `exit 2`.
- ❸ Compter les fichiers réguliers ; afficher le total après la boucle.
- ❹ Placer la classification d'un chemin dans une fonction appelée par le `for`.
- ❺ Écrire un second script qui ajoute `.pret` aux fichiers passés en arguments, en refusant une destination occupée.

Les formes à reconnaître

Besoin	Construction
Nommer une valeur	<code>nom=valeur</code> , puis <code>\$nom</code>
Choisir un bloc	<code>if ...; then ...; else ...; fi</code>
Parcourir une liste	<code>for ... in ...; do ...; done</code>
Répéter selon un test	<code>while ...; do ...; done</code>
Nommer une opération	<code>fonction() { ...; }</code>
Terminer	<code>return</code> dans une fonction, <code>exit</code> dans un script

Pour retrouver la syntaxe localement : `help if`, `help for`, `help while`, `help read` et `man bash`.