Beginner Guide to Shell Scripting for ALX Tasks

Prepared for ALX System Engineering DevOps Course

June 13, 2025

Contents

1. Understanding the World of Shell Scripting

Let's start from scratch, like you've never used a computer. Your ALX course requires coding in Ubuntu 20.04, a Linux system. Since you're already on Linux, you can use your system directly to write and run scripts—no need for extra tools like Docker.

Shell scripting means typing commands into a "shell," a control panel for your Linux system. You're using Bash, a popular shell. A script is a file with commands, like a recipe, so you can run them all at once. Think of it as automating tasks in your Linux kitchen.

Scripting concepts include variables (sticky notes with data, like `NAME=Bob`), the `PATH` (a list of folders for programs, like `/usr/bin:/bin`), aliases (custom command shortcuts), and expansions (replacing `$NAME` with `Bob`). Arithmetic uses `$(( ))`, like `$((2+3))`.

Your tasks (0–17) involve writing 2-line Bash scripts (starting with `#!/bin/bash`, ending with a newline) without shortcuts like `&&` or tools like `bc`. Let's set up and solve them.

2. Setting Up Your Linux Environment

You're on Linux, so your system is ready for ALX tasks. Let's confirm and organize your workspace.

Check your Ubuntu version: open a terminal (Ctrl+Alt+T) and run `lsb_release -a`. It should show Ubuntu 20.04. If not, ask your instructor if your version is okay or consider upgrading to 20.04.

Install essential tools if missing:

```
sudo apt-get update
sudo apt-get install -y vim
```

Create a project folder:

```
mkdir -p ~/alx-system_engineering-devops/0x03-shell_variables_expansions
cd ~/alx-system_engineering-devops/0x03-shell_variables_expansions
```

Use `vim` to write scripts: `vim filename`, press `i`, type code, press Esc, type `:wq`, and Enter. Make scripts executable: `chmod +x filename`. Run them: `./filename` .

3. Doing Your ALX Tasks

Each task gets a script, explanation, testing steps, and debugging tips, explained like you're brand new.

3.1. Task 0: Create an alias `ls` with value `rm *`

This makes typing `ls` delete all files (`rm *`) instead of listing them.

```
#!/bin/bash
alias ls=’rm␣*’  # Makes the ls command run rm * to delete all files
```

The first line, `#!/bin/bash`, tells the system to use Bash. The second line creates an alias so `ls` runs `rm *`, deleting all files in the current folder.

Test: In your project folder, create files: `touch file1.txt file2.txt`. Check: `ls` (shows `file1.txt file2.txt`). Create the script: `vim 0-alias`, paste, save (`:wq`), and make executable: `chmod +x 0-alias`. Apply: `source ./0-alias`. Run `ls`—files should be gone. Verify with `\ls` (real `ls`).

Mistakes: No spaces around = in `alias ls=’rm␣*’`. Use single quotes, not double. If `ls` lists files, check `alias ls` (should show `alias ls=’rm␣*’`).

3.2. Task 1: Print `hello  user`, where user is the current user

Prints “hello” followed by your username (e.g., `hello Bob`).

```
#!/bin/bash
echo ”hello␣$USER”  # Prints hello followed by the current username
```

`#!/bin/bash` sets the shell. `echo ”hello␣$USER”` prints `hello` and the `USER` variable (your username). Double quotes allow `$USER` to expand.

Test: In your project folder, create `vim 1-hello_you`, paste, save, and `chmod +x 1-hello_you`. Run `./1-hello_you`—should print `hello` and your username (`echo $USER` to check).

Mistakes: Use double quotes, not single, or `$USER` won’t expand. If “permission denied,” run `chmod +x`. If output is `hello`, check `printenv USER`.

3.3. Task 2: Add `/action` to the PATH

Add the `/action` folder to the `PATH` variable.

```
#!/bin/bash
export PATH=$PATH:/action  # Adds /action to the end of the PATH
   list
```

`#!/bin/bash` sets Bash. `export PATH=$PATH:/action` appends `:/action` to `PATH` and exports it.

Test: Check `echo $PATH`. Create `vim 2-path`, paste, save, `chmod +x 2-path`. Run `source ./2-path`, then `echo $PATH`—should end with `:/action`.

Mistakes: Use `source`, not `./2-path`. No spaces around = or `:`. If `/action` missing, check script with `cat 2-path`.

3.4. Task 3: Count directories in PATH

Count the folders in `PATH` (e.g., `/usr/bin:/bin:/action` is 3).

```
#!/bin/bash
echo $PATH | tr ’:’ ’\n’ | wc -l  # Counts folders in PATH by turning colons into newlines
```

`#!/bin/bash` sets Bash. `echo $PATH | tr ’:’’\n’| wc -l` prints `PATH`, replaces colons with newlines (`\n`), and counts lines.

Test: Create `vim 3-paths`, paste, save, `chmod +x 3-paths`. Run `./3-paths`. Set `export PATH=/a:/b::/c`, run again—should print 4.

Mistakes: Include `|` pipes. No single quotes around `$PATH`. If count’s off, run `echo $PATH | tr ’:’’\n’` to verify.

3.5. Task 4: List environment variables

Show all environment variables (e.g., `HOME=/home/bob`).

```
#!/bin/bash
printenv  # Shows all environment variables, like HOME and PATH
```

`#!/bin/bash` sets Bash. `printenv` lists all environment variables.

Test: Create `vim 4-global_variables`, paste, save, `chmod +x 4-global_variables`. Run `./4-global_variables`.

Mistakes: Don’t use `env` instead of `printenv`. If empty, try `printenv HOME`.

3.6. Task 5: List all variables and functions

Show all variables (local and global) and functions.

```
#!/bin/bash
set  # Shows all variables (local and global) and functions
```

`#!/bin/bash` sets Bash. `set` lists everything.

Test: Create `vim 5-local_variables`, paste, save, `chmod +x 5-local_variables`. Run `./5-local_variables | less`.

Mistakes: Output is long. If it fails, check `chmod +x`. Narrow with `./5-local_variables | grep BASH`.

3.7. Task 6: Create local variable `BEST=School`

Create a local variable `BEST` with value `School`.

```
#!/bin/bash
BEST=School  # Creates a local variable BEST with value School
```

`#!/bin/bash` sets Bash. `BEST=School` creates the local variable.

Test: Create `vim 6-create_local_variable`, paste, save, `chmod +x 6-create_local_variable`. Run `source ./6-create_local_variable`, check `echo $BEST` (should be `School`). Run `bash -c ’echo␣$BEST’`—should be empty.

Mistakes: No spaces around `=`. Use `source`. If empty, check `set | grep BEST`.

3.8. Task 7: Create global variable `BEST=School`

Create a global variable `BEST=School`.

```
#!/bin/bash
export BEST=School  # Creates a global variable BEST with value
   School
```

`#!/bin/bash` sets Bash. `export BEST=School` creates and exports `BEST`.

Test: Create `vim 7-create_global_variable`, paste, save, `chmod +x 7-create_global_variab` . Run `source ./7-create_global_variable`, check `echo $BEST` and `bash -c 'echo␣$BEST'`—both should show `School`.

Mistakes: Include `export`. Use `source`. If not global, check `printenv BEST`.

3.9. Task 8: Add 128 to `TRUEKNOWLEDGE`

Add 128 to the number in `TRUEKNOWLEDGE` (e.g., 1209 + 128 = 1337).

```
#!/bin/bash
echo $((TRUEKNOWLEDGE + 128))  # Adds 128 to TRUEKNOWLEDGE and
   prints result
```

`#!/bin/bash` sets Bash. `echo $((TRUEKNOWLEDGE + 128))` performs the addition.

Test: Create `vim 8-true_knowledge`, paste, save, `chmod +x 8-true_knowledge`. Set `export TRUEKNOWLEDGE=1209`, run `./8-true_knowledge`—should print 1337.

Mistakes: Set `TRUEKNOWLEDGE`. Don't use `$TRUEKNOWLEDGE` in `$(( ))`. If output is 128, check `printenv TRUEKNOWLEDGE`.

3.10. Task 9: Divide `POWER` by `DIVIDE`

Divide `POWER` by `DIVIDE` (e.g., 42784 / 32 = 1337).

```
#!/bin/bash
echo $((POWER / DIVIDE))  # Divides POWER by DIVIDE and prints
   result
```

`#!/bin/bash` sets Bash. `echo $((POWER / DIVIDE))` divides.

Test: Create `vim 9-divide_and_rule`, paste, save, `chmod +x 9-divide_and_rule`. Set `export POWER=42784`, `export DIVIDE=32`, run `./9-divide_and_rule`—should print 1337.

Mistakes: Set both variables. Avoid division by zero. Check `printenv POWER DIVIDE`.

3.11. Task 10: Raise `BREATH` to `LOVE`

Raise `BREATH` to the power of `LOVE` (e.g., $4^3 = 64$).

```
#!/bin/bash
echo $((BREATH ** LOVE))  # Raises BREATH to the power of LOVE
```

#!/bin/bash sets Bash. echo $((BREATH ** LOVE)) computes the power.

Test: Create vim 10-love_exponent_breath, paste, save, chmod +x 10-love_exponent_breath . Set export BREATH=4, export LOVE=3, run ./10-love_exponent_breath—should print 64.

Mistakes: Use **, not ^. Set variables. Check printenv BREATH LOVE.

3.12. Task 11: Convert BINARY from base 2 to base 10

Convert a binary number in BINARY (e.g., 10100111001) to decimal (1337).

```
#!/bin/bash
echo $((2#$BINARY))  # Converts binary number in BINARY to decimal
```

#!/bin/bash sets Bash. echo $((2#$BINARY)) converts base-2.

Test: Create vim 11-binary_to_decimal, paste, save, chmod +x 11-binary_to_decimal . Set export BINARY=10100111001, run ./11-binary_to_decimal—should print 1337.

Mistakes: Set BINARY with 0s and 1s. Check printenv BINARY.

3.13. Task 12: Print all two-letter combinations except oo

Print all lowercase letter pairs (aa to zz) except oo, one per line (675 pairs).

```
#!/bin/bash
printf "%s\n" {a..z}{a..z} | grep -v oo  # Prints all letter pairs
   except oo
```

#!/bin/bash sets Bash. printf "%s\n"{a..za..z | grep -v oo generates pairs, skips oo.

Test: Create vim 12-combinations, paste, save, chmod +x 12-combinations. Run ./12-combinations | wc -l—should print 675. Check ./12-combinations | grep oo (empty).

Mistakes: Second line must be □64 characters. Ensure oo is skipped. Verify with printf "%s\n"{a..za..z | grep oo.

3.14. Task 13: Print NUM with two decimal places

Print a number in NUM (e.g., 3.14159) with two decimal places (3.14).

```
#!/bin/bash
printf "%.2f\n" $NUM  # Prints NUM with two decimal places
```

#!/bin/bash sets Bash. printf "%.2f\n"$NUM formats to two decimals.

Test: Create vim 13-print_float, paste, save, chmod +x 13-print_float. Set export NUM=3.14159265359, run ./13-print_float—should print 3.14.

Mistakes: Use printf, not echo. Set NUM. Check printenv NUM.

3.15. Task 14: Convert `DECIMAL` from base 10 to base 16

Convert a decimal number in `DECIMAL` (e.g., 1337) to hexadecimal (539).

```
#!/bin/bash
printf ”%x\n” $DECIMAL  # Converts DECIMAL to hexadecimal
```

`#!/bin/bash` sets Bash. `printf ”%x\n”$DECIMAL` outputs lowercase hex.

Test: Create `vim 100-decimal_to_hexadecimal`, paste, save, `chmod +x 100-decimal_to_hexac` `.` Set `export DECIMAL=1337`, run `./100-decimal_to_hexadecimal`—should print 539.

Mistakes: Use `%x`, not `%X`. Set `DECIMAL`. Check `printenv DECIMAL`.

3.16. Task 15: Encode/decode with ROT13

Apply ROT13 to text, shifting letters by 13 positions.

```
#!/bin/bash
tr ’A-Za-z’ ’N-ZA-Mn-za-m’  # Shifts each letter by 13 positions
```

`#!/bin/bash` sets Bash. `tr ’A-Za-z’’N-ZA-Mn-za-m’` applies ROT13.

Test: Create `vim 101-rot13`, paste, save, `chmod +x 101-rot13`. Run `echo ”Hello ”| ./101-rot13`—should print `Uryyb`. Repeat to unscramble.

Mistakes: `tr` needs input. Verify letter ranges. Test with `echo ”ABC”| tr ’A-Za-z’’N-ZA-Mn-za-m’`.

3.17. Task 16: Print every other line (odd-numbered)

Print odd-numbered lines (1, 3, 5, ...) from input.

```
#!/bin/bash
nl -ba | grep ’^[[:space:]]*[13579]\>’ | cut -f2-  # Prints odd-numbered lines
```

`#!/bin/bash` sets Bash. `nl -ba` numbers lines, `grep` keeps odd numbers, `cut` removes numbers.

Test: Create `vim 102-odd`, paste, save, `chmod +x 102-odd`. Run `ls -1 | ./102-odd`.

Mistakes: Needs input. Check `grep` pattern. Test with `ls -1 | nl -ba`.

3.18. Task 17: Add `WATER` and `STIR` in custom bases

Add `WATER` (base `w=0, a=1, t=2, e=3, r=4`) and `STIR` (base `s=0, t=1, i=2, r=3`), output in `bestchol` (base `b=0, e=1, s=2, t=3, c=4, h=5, o=6, l=7`). Uses octal due to 2-line limit.

```
#!/bin/bash
printf ”%o\n” $(( 5#$(tr ’water’ ’0-4’ <<<”$WATER”) + 4#$(tr ’stir’
   ’0-3’ <<<”$STIR”) ))  # Adds WATER and STIR, prints in octal
```

`#!/bin/bash` sets Bash. Converts `WATER` and `STIR` to decimal, adds, prints in octal.

Test: Create `vim 103-water_and_stir`, paste, save, `chmod +x 103-water_and_stir` . Set `export WATER=ewwatratewa`, `export STIR=ti.itirtrtr`, run `./103-water_and_stir` . Map octal to `bestchol` manually.

Mistakes: Set variables. Map output manually. Ask instructor if more lines are allowed for exact `bestchol`.