SHELL REFERENCE

100+ Bash Scripting Tricks

Practical tricks for faster, safer, and more readable Bash scripts — from the basics to the latest additions in Bash 5.3.

101 TRICKS

13 CATEGORIES

Compiled July 2026 · SplitZrate.com

Contents

1
Fundamentals & Script Structure

2
Variables & Parameter Expansion

3
Arrays

4
Conditionals & Tests

5
Loops & Flow Control

6
Functions

7
Error Handling & Debugging

8
Processes, Jobs & Subshells

9
Input/Output & Redirection

10
Performance & Efficiency

11
Script Security

12
Toolbox & Practical One-Liners

13
What's New in Bash 5.3

1 Fundamentals & Script Structure

1.1 Always use a proper shebang

Example: `#!/usr/bin/env bash` at the top of the file — more portable than `#!/bin/bash` because it finds bash via `PATH` no matter where it's installed.

1.2 Make the script executable once and for all

Example: `chmod +x my_script.sh` — skip typing `bash` in front of the command every time; just run `./my_script.sh`.

1.3 Exit early on error

Example: `set -euo pipefail` at the top of the script — `-e` aborts on error, `-u` catches uninitialized variables, `-o pipefail` makes a pipe fail if any stage of it fails, not just the last one.

1.4 Check the exit code explicitly when it matters

Example: `if ! command; then echo "Failed" >&2; exit 1; fi` — clearer than relying solely on `set -e` in complex scripts.

1.5 Document the script's purpose at the top

A short comment block with purpose, usage, and an example at the top of the file saves time for both you and your colleagues six months later.

1.6 Use the right exit code to signal the result

Example: `exit 0` on success, `exit 1` (or higher) on failure — other scripts and CI systems rely on this to determine whether your script succeeded.

1.7 Separate configuration from logic

Put constants and settings at the top of the script, or in a separate file sourced with `source config.sh` — makes the script easier to adjust without touching the logic.

1.8 Run a syntax check without executing

Example: `bash -n my_script.sh` — catches syntax errors before you actually run the script, handy in pre-commit hooks.

2 Variables & Parameter Expansion

2.1 Set default values elegantly

Example: `name="${1:-Anonymous}"` — uses the first argument if it exists, otherwise "Anonymous", without a separate if statement.

2.2 Require that a variable is set

Example: `: "${API_KEY:?The API_KEY environment variable must be set}"` — aborts the script with a clear error message if the variable is missing.

2.3 Extract substrings without external tools

Example: `${string:2:5}` — grab 5 characters starting at index 2, directly in Bash without calling `cut` or `awk`.

2.4 Replace text in a variable directly

Example: `${string//old/new}` replaces every occurrence; `${string/old/new}` replaces only the first.

2.5 Strip a prefix or suffix from a path

Example: `${path##*/}` gives you the filename (everything after the last `/`), while `${path%/*}` gives you the directory (everything before the last `/`) — faster than calling `basename` / `dirname` in tight loops.

2.6 Swap a file extension in one step

Example: `${file%.txt}.md` — strip `.txt` and add `.md` in a single step, handy for batch-converting files.

2.7 Uppercase or lowercase a variable's contents

Example: `${string^^}` for uppercase, `${string,,}` for lowercase — built into Bash since version 4, no `tr` needed.

2.8 Count the length of a string or array

Example: `${#string}` gives the character count, `${#array[@]}` gives the element count — same syntax for both, different meaning depending on context.

2.9 Measure time with microsecond precision

Example: `start=$EPOCHREALTIME ... end=$EPOCHREALTIME` — a built-in variable in modern Bash versions that gives microsecond-precision timestamps without repeatedly calling `date`.

2.10 Use indirect references for dynamic variable names

Example: `name="PATH"; echo "${!name}"` — fetch the value of the variable whose name is stored in another variable, useful for generic helper functions.

3 Arrays

3.1 Declare and loop over an indexed array

Example: `fruits=(apple banana pear)` and `for f in "${fruits[@]}"; do echo "$f"; done` — always quote `"${array[@]}"` to handle elements containing spaces correctly.

3.2 Use associative arrays for key-value data

Example: `declare -A config=([host]="localhost" [port]="8080")` and `echo "${config[host]}"` — a built-in alternative to parsing external config files for simple cases.

3.3 Add elements to an array dynamically

Example: `fruits+=("mango")` — extend an array without rewriting the whole declaration.

3.4 Loop with both index and value

Example: `for i in "${!fruits[@]}"; do echo "$i: ${fruits[$i]}"; done` — `"${!array[@]}"` gives you all the indices, handy when you need the position.

3.5 Split a string into an array

Example: `IFS=',' read -ra parts <<< "a,b,c"` — split a comma-separated string into an array without external tools like `cut`.

3.6 Join an array into a string

Example: `IFS=','; echo "${fruits[*]}"` — `"${array[*]}"` with `IFS` set joins the elements with the given separator character.

3.7 Remove an element from an array

Example: `unset 'fruits[1]'` — removes the element at index 1; remember the array's indexing can end up with gaps afterward.

3.8 Copy an array safely

Example: `copy=("${original[@]}")` — creates a real copy, not just a reference to the same data.

3.9 Copy an array to another variable without sharing a reference

Example: `mapfile -t copy < <(printf '%s\n' "${original[@]}")` — an alternative to direct array copying when you also want to filter or transform the contents during the copy.

4 Conditionals & Tests

4.1 Prefer [[ ]] over [ ] in Bash scripts

`[[ ]]` is a Bash built-in construct that handles string comparisons, pattern matching, and logical operators more safely and with fewer quoting pitfalls than the POSIX-compatible `[ ]`.

4.2 Test multiple conditions readably

Example: `if [[ -f "$file" && -r "$file" ]]; then ...` — combine conditions with `&&` / `||` directly inside `[[ ]]` instead of chaining separate test commands.

4.3 Match patterns without regex

Example: `if [[ "$filename" == *.log ]]; then ...` — glob patterns work directly inside `[[ ]]` without needing `case` or external tools.

4.4 Use regular expressions when patterns aren't enough

Example: `if [[ "$input" =~ ^[0-9]+$ ]]; then echo "Digits only"; fi` — `=~` inside `[[ ]]` gives you full POSIX ERE support right in the shell.

4.5 Check whether a variable is empty safely

Example: `if [[ -z "${name:-}" ]]; then echo "Empty or unset"; fi` — `-z` with default expansion avoids errors even with `set -u` enabled.

4.6 Test file types before acting on them

Example: `-e` (exists), `-f` (regular file), `-d` (directory), `-x` (executable), `-s` (non-empty) — combine these for robust pre-checks before file operations.

4.7 Use case for multi-way choices instead of long if chains

Example:

```
case "$1" in
    start) start_service ;;
    stop) stop_service ;;
    *) echo "Unknown command" ;;
esac
```

— clearer and faster than a long `if/elif` chain for multiple options.

4.8 Check whether a command exists before running it

Example: `if command -v docker &>/dev/null; then ...` — `command -v` is more portable and faster than `which` for checking availability.

4.9 Test multiple patterns in a single case branch

Example: `case "$file" in *.jpg|*.jpeg|*.png) process_image "$file" ;; esac` — separate multiple patterns with `|` instead of duplicating the branch.

5 Loops & Flow Control

5.1 Loop over files safely, even with spaces in the name

Example: `for file in *.txt; do echo "$file"; done` — always quote `"$file"` inside the loop; avoid parsing `ls` output line by line.

5.2 Read a file line by line correctly

Example: `while IFS= read -r line; do echo "$line"; done < file.txt` — `IFS=` preserves leading/trailing whitespace and `-r` stops backslashes from being interpreted as escape sequences.

5.3 Break out of nested loops with break N

Example: `break 2` exits two nested loop levels in one step instead of setting a flag that the outer loop checks.

5.4 Skip an iteration with continue

Example: `for f in *.log; do [[ -s "$f" ]] || continue; process "$f"; done` — cleanly skip past empty files without nesting the rest of the loop body in an if statement.

5.5 Use C-style for loops for numeric sequences

Example: `for ((i=0; i<10; i++)); do echo "$i"; done` — clearer than `seq` when you're already inside a Bash script.

5.6 Run a loop a fixed number of times without a counter variable

Example: `for _ in {1..5}; do echo "run"; done` — brace expansion generates the sequence directly, no `seq` process needed.

5.7 Create an infinite loop with a clear break condition

Example: `while true; do check_status || break; sleep 5; done` — a common pattern for monitoring or polling scripts.

6 Functions

6.1 Define functions with local variables

Example:

```
my_function() {
    local result="$1"
    echo "$result"
}
```

— `local` keeps a function's variables from leaking out and colliding with global variables.

6.2 Return data via stdout, not the return code

Bash's return code is limited to 0–255 and meant for status codes. Send actual data by writing to stdout and capturing it with `result=$(my_function)`.

6.3 Check the number of arguments before using them

Example: `if [[ $# -lt 2 ]]; then echo "Usage: $0 <arg1> <arg2>" >&2; exit 1; fi` — give a clear error message instead of cryptic failures further down the script.

6.4 Use named arguments for readability in complex functions

Example: assign `$1`, `$2` to descriptive local variables right at the start of the function — makes the rest of the function much easier to read.

6.5 Build reusable function libraries

Put shared functions in a separate file (e.g. `lib.sh`) and source it with `source lib.sh` across multiple scripts — avoid copying the same logic in several places.

6.6 Use fork-free execution for fast command substitutions

Example: `result=${ echo hi; }` — the new `${ command; }` syntax in Bash 5.3 runs the command in the same shell process instead of forking a subshell, which is faster in tight loops.

7 Error Handling & Debugging

7.1 Catch errors with trap

Example: `trap 'echo "Error on line $LINENO" >&2' ERR` — automatically run an action the moment a command fails, with the line number for fast debugging.

7.2 Clean up resources guaranteed, even on crash

Example: `trap 'rm -f "$tempfile"' EXIT` — the temp file is removed regardless of whether the script exits normally, with an error, or is interrupted with Ctrl+C.

7.3 Identify exactly which signal was caught

Example: in Bash 5.3, `$BASH_TRAPSIG` holds the numeric signal number inside a trap handler — handy when the same handler function is used for multiple signals.

7.4 Run the script in debug mode when needed

Example: `bash -x my_script.sh` or `set -x` inside the script — prints every command before it runs, invaluable for finding exactly where a script goes wrong.

7.5 Temporarily disable debug output around sensitive sections

Example: `set +x; secret_operation; set -x` — avoid passwords or tokens ending up in the debug log while troubleshooting a script that handles sensitive data.

7.6 Write error messages to stderr, not stdout

Example: `echo "Error: file missing" >&2` — keeps stdout clean for actual program data, which matters if the script's output gets piped further.

7.7 Validate input before you trust it

Example: check that numeric arguments are actually digits with `[[ "$1" =~ ^[0-9]+$ ]]` before using them in calculations or filenames.

7.8 Use a strict error-handling template for new scripts

The combination of `set -euo pipefail`, `trap ... ERR`, and clear error messages to stderr is a good default skeleton to start every new script from.

8 Processes, Jobs & Subshells

8.1 Run commands in parallel and wait for all of them

Example: `command1 & command2 & wait` — starts both in the background and waits until both finish before the script continues.

8.2 Wait for a specific background job

Example: `command & pid=$!; wait "$pid"` — save the PID of the job you actually care about instead of waiting for every background job.

8.3 Capture the exit code from a background job

In Bash 5.3, `wait` can return the status code of a specific job more reliably — combine with `$?` right after `wait "$pid"` to check whether the background job succeeded.

8.4 Use process substitution to avoid temp files

Example: `diff <(sort file1) <(sort file2)` — treat the output of two commands as files without creating actual temporary files.

8.5 Run a subshell to isolate side effects

Example: `(cd /tmp && command)` — parentheses create a subshell where `cd` only affects the inner environment; the main script's working directory stays unchanged.

8.6 List and manage active background jobs

Example: `jobs -l` — show the PID and status of every background job started in the current shell session.

8.7 Send a process to the background and detach it

Example: `nohup ./long_job.sh > log.txt 2>&1 &` — let the process keep running even if the terminal session is closed.

9 Input/Output & Redirection

9.1 Write multi-line text with a heredoc

Example:

```
cat <<EOF > config.txt
host=localhost
port=8080
EOF
```

— clearer than multiple `echo` lines for generating text files in scripts.

9.2 Avoid variable expansion in a heredoc when needed

Example: `cat <<'EOF'` (with a quoted delimiter) — prevents `$var` and command substitution from being interpreted inside the heredoc block.

9.3 Discard unwanted output cleanly

Example: `command > /dev/null 2>&1` — hide both standard output and error output when you only care about the exit code.

9.4 Read input interactively with readline support

Example: `read -e -p "Enter name: " name` — gives line editing and history in the prompt. The new `-E` flag in Bash 5.3 also enables Bash's standard completions during input.

9.5 Send multiple commands' output to the same file

Example: `{ command1; command2; } > log.txt` — group commands with braces to share a single redirection.

9.6 Read an entire file into a variable

Example: `content=$(<file.txt)` — faster than `cat file.txt` since it avoids starting an extra process.

9.7 Write to multiple destinations at once

Example: `command | tee log.txt` — see the output on screen while it's also saved to a file, handy when debugging long-running scripts.

10 Performance & Efficiency

10.1 Prefer built-in commands over external processes

Use Bash's own `${variable//search/replace}` instead of calling `sed`, or arithmetic with `(( ))` instead of `expr` — every avoided process start saves time, especially in loops that run thousands of times.

10.2 Avoid spawning a subshell unnecessarily

Example: `((counter++))` instead of `counter=$(( counter + 1 ))` when you just need to increment a value — the former is faster and clearer.

10.3 Read files once instead of opening them repeatedly in loops

Load the whole file into an array with `mapfile -t lines < file.txt` and then loop over the array, instead of calling `sed -n` or `head` per iteration.

10.4 Use printf instead of echo for formatted output

Example: `printf "%-10s %5d\n" "$name" "$count"` — more reliable and portable than `echo` for column-aligned or formatted text.

10.5 Sort pathname expansion without an extra process

In Bash 5.3, the `GLOBSORT` variable controls how glob patterns like `*.txt` are sorted (by name, size, modification time, etc.) directly in the shell, without piping the result to `sort`.

10.6 Measure a script's performance easily

Example: `time ./my_script.sh` — see real, user, and sys time for the entire run without instrumenting the code by hand.

10.7 Avoid unnecessary calls to external tools for simple checks

Example: check a file's size with `[[ -s "$file" ]]` instead of parsing the output of `ls -l` or `du`.

11 Script Security

11.1 Always quote variables that might contain spaces

Example: `rm "$file"` instead of `rm $file` — an unquoted variable can be interpreted as several separate arguments if the value contains spaces or special characters.

11.2 Avoid eval unless absolutely necessary

`eval` runs arbitrary text as code, a classic source of command injection if the text comes from external input. Look for a safer alternative first, like arrays or `${!variable}`.

11.3 Set a safe IFS before parsing external data

Example: `IFS=$'\n\t'` — restrict the field separator to newline and tab instead of Bash's default (space, tab, newline), reducing the risk of unexpected word splitting.

11.4 Validate file paths before destructive operations

Example: check that a variable actually points to the expected path before running `rm -rf "$path"` — an empty or wrong variable can otherwise lead to catastrophic results.

11.5 Use mktemp for temp files instead of hardcoded names

Example: `tempfile=$(mktemp)` — avoids race conditions and collisions that can occur with predictable filenames in `/tmp`.

11.6 Limit what a script can do with least privilege

Run scripts as a restricted user wherever possible, and avoid requiring `sudo` for the entire script when only a handful of lines actually need elevated rights.

11.7 Clear sensitive variables after use

Example: `unset password` as soon as a sensitive variable is no longer needed — reduces the risk of it accidentally getting logged or leaking via a debug print later in the script.

12 Toolbox & Practical One-Liners

12.1 **Find and act on files in one sweep**

Example: `find . -name "*.tmp" -mtime +7 -delete` — find files older than 7 days and delete them directly, no intermediate step.

12.2 **Run a command for each line from another process**

Example: `find . -name "*.log" | xargs -I{} gzip {}` — compress every log file in a single piped command instead of a manual loop.

12.3 **Count occurrences quickly**

Example: `grep -c "ERROR" log.txt` — count matching lines directly without piping to `wc -l`.

12.4 **Generate random values directly in the shell**

Example: `echo $RANDOM` for an integer, or `head -c 16 /dev/urandom | base64` for a stronger random string for things like temporary passwords.

12.5 **Create multiple directories in a tree structure at once**

Example: `mkdir -p project/{src,test,docs}` — brace expansion creates three directories in a single command.

12.6 **Rename multiple files according to a pattern**

Example: `for f in *.jpeg; do mv "$f" "${f%.jpeg}.jpg"; done` — combine a loop and parameter expansion for batch renaming without external tools.

12.7 **Chain output from multiple commands conditionally**

Example: `command1 && command2 || command3` — run `command2` only if `command1` succeeds, otherwise `command3`; a common pattern for simple fallback chains.

12.8 Build a simple menu with select

Example:

```
select choice in "Start" "Stop" "Exit"; do
    case $choice in
        Start) start ;;
        Stop) stop ;;
        Exit) break ;;
    esac
done
```

— a built-in way to build interactive text menus without external libraries.

13 What's New in Bash 5.3

13.1 Run commands in the current shell process with ${ ;}

The new `${ command; }` syntax executes a command in the same shell process instead of creating a subshell, making command substitution noticeably faster in performance-sensitive scripts.

13.2 Get better error messages for invalid variable names

Bash 5.3 gives clearer diagnostics when a script tries to create or use an invalid variable name (one containing a hyphen, for example), making debugging faster.

13.3 Take advantage of monotonic clock support

Example: `$BASH_MONOSECONDS` gives you access to a monotonic clock unaffected by system clock adjustments — more reliable than `$SECONDS` for measuring elapsed time in long-running scripts.

13.4 Check POSIX-compliant umask handling

The `umask` builtin gained more flags for full POSIX compliance in Bash 5.3 — relevant if your scripts need to run identically across several different POSIX-compatible shells.

13.5 Force quoting in printf with %q and %Q

Example: `printf "%q\n" "$path"` now uses an "alternate form" that forces single quotes — makes it easier to generate shell-safe output that can be reused as input to another command.

13.6 Sort file matches without external sorting

Set `GLOBSORT=size` or `GLOBSORT=mtime` before a glob expansion to get files in size or time order directly, without piping the result through `sort`.

13.7 Use new loadable builtins for specialized tasks

Bash 5.3 introduces new loadable builtins like `strptime` for date parsing and `fltexpr` for floating-point calculations — worth exploring if your script would otherwise need to repeatedly call `date` or `bc`.

13.8 Save completion suggestions to a variable

Example: `compgen -V suggestions -A file` — the new `-V` flag in `compgen` saves the result directly into a named array variable instead of printing it to stdout, handy for custom completion scripts.