Table of Contents

$ alias

Beginner

Create command shortcuts

alias creates shortcuts for commands or command sequences. An alias replaces a word with a string when used as the first word of a command. Aliases save typing and reduce errors for frequently used commands.

alias is a shell built-in that defines command shortcuts for the current session. For pe...

Options & Flags

`(no args)`	List all current aliases
`name=value`	Create an alias
`unalias`	Remove an alias

Practical Examples

Example: List all aliases

```
$ alias
alias ll='ls -la'
alias la='ls -A'
```

Shows all currently defined aliases.

Example: Create simple alias

```
$ alias update='sudo apt update && sudo apt upgrade -y'
```

Creates a shortcut for system updates.

Example: Create ls shortcut

```
$ alias ll='ls -lah --color=auto'
```

Creates a detailed listing shortcut with human-readable sizes.

Example: Safety aliases

```
$ alias rm='rm -i' alias cp='cp -i' alias mv='mv -i'
```

Adds confirmation prompts to destructive commands.

Example: Make alias permanent

```
$ echo "alias ll='ls -la'" >> ~/.bashrc && source ~/.bashrc
```

Adds the alias to .bashrc and reloads it.

Tips & Best Practices

Pro Tip: Bypass an alias: Prefix with backslash to skip alias expansion: \rm file.txt runs /bin/rm even if rm is aliased to rm -i.

Note: Permanent aliases: Add aliases to ~/.bashrc or create ~/.bash_aliases (sourced by many default .bashrc files). Run source ~/.bashrc to apply.

Warning: Aliases do not work in scripts: Bash disables alias expansion in non-interactive mode (scripts). Use functions instead of aliases in scripts.

$ basename

Beginner

Strip directory and suffix from filenames

basename strips directory paths and optionally file extensions from filenames. It extracts just the filename component from a full path, making it essential for file processing scripts.

basename takes a path and returns only the final component. With a second argument, it also removes a specifie...

Options & Flags

`PATH`	Strip directory from path
`PATH SUFFIX`	Strip directory and suffix
`-s`	Remove suffix (alternative syntax)
`-a`	Process multiple arguments

Practical Examples

Example: Get filename from path

```
$ basename /var/log/nginx/access.log
access.log
```

Extracts just the filename from the full path.

Example: Remove extension

```
$ basename report.pdf .pdf
report
```

Strips the .pdf extension from the filename.

Example: In a script

```
$ for f in /data/*.csv; do echo "Processing $(basename "$f" .csv)"; done
Processing users\nProcessing orders
```

Gets base name without extension for each CSV file.

Example: Remove compound extension

```
$ basename -s .tar.gz backup.tar.gz
backup
```

Strips a multi-part extension.

Example: Multiple paths

```
$ basename -a /usr/bin/python3 /usr/bin/node /usr/bin/php
python3
node
php
```

Extracts filenames from multiple paths.

Tips & Best Practices

Pro Tip: Use in conversion scripts: base=$(basename "$file" .jpg); convert "$file" "${base}.png" - converts image format using the original base name.

Note: dirname is the complement: basename gets the filename; dirname gets the directory. Together they split a path: dirname /a/b/c ? /a/b, basename /a/b/c ? c.

Warning: Quote your variables: Always quote: basename "$filepath". Filenames with spaces will break without quotes.

$ dirname

Beginner

Strip last component from a file path

dirname strips the last component from a file path, returning the directory portion. It is the complement to basename - dirname gets the directory, basename gets the filename.

dirname is essential in scripts for finding the directory containing a file, navigating relative to a script\'s location...

Options & Flags

`PATH`	Extract directory from path
`-z`	End output with NUL instead of newline
`multiple`	Process multiple paths

Practical Examples

Example: Get directory from path

```
$ dirname /var/log/nginx/access.log
/var/log/nginx
```

Returns the directory containing the file.

Example: Find script directory

```
$ SCRIPT_DIR=$(dirname "$0")
```

Gets the directory where the current script is located.

Example: Navigate relative to script

```
$ source "$(dirname "$0")/config.sh"
```

Sources a config file from the same directory as the script.

Example: Process file in same directory

```
$ OUTPUT_DIR=$(dirname "$INPUT_FILE"); cp "$INPUT_FILE" "$OUTPUT_DIR/backup/"
```

Uses dirname to construct output path relative to input file.

Example: Multiple paths

```
$ dirname /usr/bin/python3 /etc/nginx/nginx.conf
/usr/bin
/etc/nginx
```

Returns directory for each path.

Tips & Best Practices

Pro Tip: Script location pattern: SCRIPT_DIR=$(cd "$(dirname "$0")" && pwd) gives the absolute directory of the running script. Very common pattern.

Note: dirname is string manipulation: dirname does not check if paths exist. dirname /nonexistent/path/file returns /nonexistent/path.

Warning: Trailing slashes: dirname handles trailing slashes: dirname /a/b/ returns /a. But edge cases exist - test your specific usage.

$ echo

Beginner

Display a line of text to the terminal

echo displays a line of text or variable values to standard output. It is one of the most basic and frequently used commands in Linux, essential for scripts, debugging, and generating output.

echo supports escape sequences (with -e flag) for formatting including newlines, tabs, and colors. It is...

Options & Flags

`-n`	Do not output trailing newline
`-e`	Enable interpretation of escape sequences
`-E`	Disable escape sequences (default)
`\n`	Newline (with -e)
`\t`	Tab (with -e)
`\033[`	ANSI color codes (with -e)

Practical Examples

Example: Print text

```
$ echo "Hello World"
Hello World
```

Prints text to standard output.

Example: Print variable

```
$ echo "Home is: $HOME"
Home is: /home/user
```

Prints the value of the HOME environment variable.

Example: No trailing newline

```
$ echo -n "Enter your name: "
```

Prints without newline - useful for inline prompts.

Example: Write to file

```
$ echo "server=192.168.1.1" > config.txt
```

Creates (or overwrites) a file with the text.

Example: Append to file

```
$ echo "$(date): Deploy complete" >> deploy.log
```

Appends a timestamped message to a log file.

Tips & Best Practices

Pro Tip: Use printf for consistency: echo behavior varies between shells. printf behaves consistently: printf "%s\n" "Hello" works the same everywhere.

Note: Single vs double quotes: Double quotes expand variables: echo "". Single quotes are literal: echo '$HOME' prints $HOME literally.

Warning: -e is not universal: Not all echo implementations support -e. In scripts targeting multiple systems, use printf instead for escape sequences.

$ env

Beginner

Display or set environment variables

env displays environment variables or runs a command with a modified environment. Without arguments, it prints all current environment variables. With arguments, it sets variables for a single command execution.

env is useful for running commands with specific environment settings without modify...

Options & Flags

`-i`	Start with empty environment
`-u`	Unset a variable
`VAR=val`	Set variable for the command
`(no args)`	Print all environment variables
`-0`	End each line with NUL instead of newline
`#!/usr/bin/env`	Find interpreter in PATH (shebang)

Practical Examples

Example: Show all variables

```
$ env
HOME=/home/user\nPATH=/usr/bin:/bin\nSHELL=/bin/bash
```

Displays all current environment variables.

Example: Run with custom environment

```
$ env NODE_ENV=production node server.js
```

Runs node with NODE_ENV set to production, without affecting the current shell.

Example: Run with empty environment

```
$ env -i HOME=$HOME PATH=/usr/bin bash -c "echo clean shell"
```

Starts a command with minimal environment variables.

Example: Unset variable for command

```
$ env -u http_proxy curl https://api.example.com
```

Runs curl without the http_proxy variable (bypasses proxy).

Example: Sort environment variables

```
$ env | sort
```

Displays all variables in alphabetical order.

Tips & Best Practices

Pro Tip: #!/usr/bin/env for scripts: Use #!/usr/bin/env python3 instead of #!/usr/bin/python3 in scripts. env finds the right interpreter regardless of where it is installed.

Note: env vs printenv: env without args and printenv both show environment variables. env can also run commands; printenv can show a single variable: printenv HOME.

Warning: env -i is very minimal: env -i strips ALL variables. Most programs need at least HOME, PATH, and TERM to function properly.

$ export

Beginner

Set environment variables for child processes

export marks a shell variable so it is passed to child processes as an environment variable. Without export, variables are local to the current shell and invisible to any commands or scripts you run.

export is essential for setting up environment variables that programs need to function - like P...

Options & Flags

`-n`	Un-export a variable (keep as local)
`-p`	List all exported variables
`-f`	Export a function
`VAR=value`	Set and export in one step

Practical Examples

Example: Set environment variable

```
$ export DB_HOST="localhost"
```

Sets and exports DB_HOST so child processes can read it.

Example: Extend PATH

```
$ export PATH="$HOME/.local/bin:$PATH"
```

Adds a custom directory to the beginning of PATH.

Example: Set multiple variables

```
$ export NODE_ENV="production" PORT=3000
```

Exports multiple variables in one command.

Example: Check exported variables

```
$ export -p | grep DB_
declare -x DB_HOST="localhost"
```

Lists all exported variables matching DB_.

Example: Un-export a variable

```
$ export -n SECRET_KEY
```

Removes export attribute - variable remains in shell but is not passed to children.

Tips & Best Practices

Pro Tip: Difference: VAR=x vs export VAR=x: VAR=x sets a local shell variable. export VAR=x makes it available to child processes. Use export when programs need to read it.

Note: Inline environment: Set a variable for a single command: DB_HOST=remote ./script.sh. This exports DB_HOST only for that command.

Warning: Not persistent by default: export only lasts for the current session. For persistence, add it to ~/.bashrc or ~/.profile.

$ expr

Intermediate

Evaluate expressions (arithmetic, string, comparison)

expr evaluates expressions and outputs the result. It performs integer arithmetic, string operations, and pattern matching. expr is a legacy tool - most of its functionality is now handled better by bash built-in arithmetic.

expr supports addition, subtraction, multiplication, division, modulo, ...

Options & Flags

`+,-,*,/,%`	Integer arithmetic
`length`	String length
`substr`	Substring extraction
`:`	Pattern matching (regex)
`=,!=`	String comparison
`<,<=,>,>=`	Numeric comparison

Practical Examples

Example: Integer arithmetic

```
$ expr 10 + 5
15
```

Adds two numbers.

Example: Multiplication

```
$ expr 6 \* 7
42
```

Multiplies - note the backslash to escape * from shell expansion.

Example: String length

```
$ expr length "Hello World"
11
```

Returns the number of characters in the string.

Example: Substring

```
$ expr substr "Hello World" 7 5
World
```

Extracts 5 characters starting at position 7.

Example: Pattern matching

```
$ expr "report_2024.pdf" : ".*_\([0-9]*\)"
2024
```

Extracts the year using regex.

Tips & Best Practices

Pro Tip: Use $(( )) instead: Bash arithmetic is simpler and faster: echo $((10 + 5)) instead of expr 10 + 5. Use expr only for POSIX sh scripts.

Warning: Escape * for multiplication: expr 6 * 7 fails because * is expanded by the shell. Use expr 6 * 7 with backslash escape.

Note: Spaces are required: Every operator and operand must be a separate argument: expr 10 + 5 works, expr 10+5 fails.

$ getopts

Intermediate

Parse command-line options in shell scripts

The getopts command is a built-in shell utility for parsing command-line options (flags) in bash and POSIX shell scripts. It processes single-character options like -v, -f filename, and -h, following Unix conventions for command-line argument handling.

getopts is the standard way to make shell s...

Options & Flags

`OPTSTRING`	String of valid option characters (colon after = requires...
`VARIABLE`	Variable to store the current option character
`$OPTARG`	Contains the argument for the current option (if colon in...
`$OPTIND`	Index of the next argument to process
`Leading :`	Silent error mode - suppress error messages
`?`	Matches unknown/invalid options

Practical Examples

Example: Basic option parsing

```
$ #!/bin/bash while getopts "vf:o:h" opt; do   case $opt in     v) verbose=true ;;     f) input_file=$OPTARG ;;     o) output_file=
```

Complete option parsing: -v (flag), -f file (with argument), -o output (with argument), -h (help). Shift removes parsed options leaving positional args.

Example: Script with verbose mode

```
$ #!/bin/bash verbose=false while getopts "v" opt; do   case $opt in v) verbose=true ;; esac done shift $((OPTIND - 1)) [ "$verbos
```

Simple verbose flag. After getopts, $1 contains the first non-option argument.

Example: Required argument validation

```
$ #!/bin/bash while getopts ":f:" opt; do   case $opt in     f) file=$OPTARG ;;     :) echo "Option -$OPTARG requires an argument"
```

Handle missing arguments (leading colon enables silent mode). Check required options after parsing.

Example: Backup script with options

```
$ #!/bin/bash usage() { echo "Usage: $0 -s source -d dest [-c] [-v]"; exit 1; } while getopts "s:d:cvh" opt; do   case $opt in
```

A practical backup script accepting source, destination, compression flag, and verbose mode.

Example: Option bundling works

```
$ ./script.sh -cv -f input.txt
```

getopts handles bundled options: -cv is the same as -c -v. Options with arguments (-f) can be bundled too: -cvf input.txt.

Tips & Best Practices

Pro Tip: Always shift after getopts: After the getopts loop, run: shift $((OPTIND - 1)). This removes parsed options, leaving only positional arguments in $1, $2, etc.

Note: getopts vs getopt: getopts (built-in, POSIX) handles short options only. getopt (external, GNU) supports long options (--verbose). getopts is more portable; use getopt when you need long options.

Warning: Reset OPTIND for functions: OPTIND is global. If calling getopts in a function, set local OPTIND=1 first, or pass arguments explicitly to avoid conflicts with the main script.

Pro Tip: Use leading colon for custom errors: Start optstring with : (e.g., ":vf:h") to suppress default error messages. Handle errors yourself in the ? and : cases for cleaner output.

$ history

Beginner

Display or manipulate command history

history displays the command history list, showing previously executed commands with their line numbers. It allows you to search, recall, and re-execute past commands, dramatically increasing shell productivity.

Bash stores command history in memory during a session and saves it to ~/.bash_histo...

Options & Flags

`N`	Show last N commands
`-c`	Clear the history list
`-d N`	Delete entry number N
`-w`	Write history to file
`-r`	Read history from file
`-a`	Append new entries to history file

Practical Examples

Example: Show recent commands

```
$ history 20
  151  git status\n  152  git add .\n  153  git commit -m "fix"
```

Shows the last 20 commands you executed.

Example: Search history

```
$ history | grep ssh
  45  ssh user@server1\n  89  ssh admin@server2
```

Finds all SSH commands you have previously run.

Example: Re-execute last command

```
$ !!
```

Runs the previous command again. Useful for: sudo !!

Example: Re-execute by number

```
$ !151
```

Runs the command at history line 151.

Example: Re-execute by prefix

```
$ !ssh
```

Runs the most recent command starting with "ssh".

Tips & Best Practices

Pro Tip: Ctrl+R reverse search: Press Ctrl+R and type to search backwards through history. Press Ctrl+R again to find older matches. Enter to execute, Ctrl+G to cancel.

Note: sudo !! pattern: Forgot sudo? Type sudo !! to re-run the last command with sudo. !! is replaced with the previous command.

Warning: Sensitive data in history: Commands with passwords appear in history. Use HISTCONTROL=ignorespace and prefix sensitive commands with a space to exclude them.

$ printenv

Beginner

Print all or specific environment variables

printenv prints the values of environment variables. Without arguments, it prints all environment variables. With a variable name as argument, it prints just that variable\'s value.

printenv is simpler than env for displaying variables. It is useful in scripts for checking if a variable is set a...

Options & Flags

`VARNAME`	Print value of specific variable
`(no args)`	Print all environment variables
`-0`	End each line with NUL instead of newline
`multiple`	Print multiple specific variables

Practical Examples

Example: Print specific variable

```
$ printenv HOME
/home/user
```

Shows the value of HOME.

Example: Print all variables

```
$ printenv | sort
```

Lists all environment variables sorted alphabetically.

Example: Check if variable exists

```
$ printenv API_KEY > /dev/null 2>&1 && echo "Set" || echo "Not set"
```

Uses exit code to check if a variable is defined.

Example: Print multiple variables

```
$ printenv HOME USER SHELL
/home/user
user
/bin/bash
```

Displays the values of HOME, USER, and SHELL.

Example: Search for variables

```
$ printenv | grep -i proxy
```

Finds all proxy-related environment variables.

Tips & Best Practices

Pro Tip: Check variable existence: printenv VARNAME returns exit code 0 if set, 1 if not. More reliable than testing echo $VAR which shows empty string for unset vars.

Note: printenv vs echo $VAR: printenv VAR shows only exported variables. echo $VAR shows both local and exported. printenv has useful exit codes.

Warning: Only environment variables: printenv shows only exported environment variables, not local shell variables. Use set or declare to see all variables.