Table of Contents

$ printf

Intermediate

Format and print data with precise control

printf formats and prints data according to a format string, similar to C\'s printf function. It provides precise control over output formatting including field widths, padding, number formatting, and escape sequences.

printf is more consistent than echo across different shells and POSIX systems...

Options & Flags

`%s`	String format specifier
`%d`	Integer format specifier
`%f`	Float format specifier
`%x`	Hexadecimal format specifier
`%-Ns`	Left-align in N-width field
`%0Nd`	Zero-padded number

Practical Examples

Example: Formatted string

```
$ printf "Name: %s, Age: %d\n" "Alice" 30
Name: Alice, Age: 30
```

Formats and prints with type-safe specifiers.

Example: Formatted table

```
$ printf "%-15s %-10s %s\n" "Name" "Role" "Dept"; printf "%-15s %-10s %s\n" "Alice" "Dev" "Engineering"
Name            Role       Dept
Alice           Dev        Engineering
```

Creates aligned table output with fixed-width columns.

Example: Zero-padded numbers

```
$ for i in $(seq 1 5); do printf "file_%03d.txt\n" $i; done
file_001.txt
file_002.txt
file_003.txt
```

Generates zero-padded filenames: file_001.txt through file_005.txt.

Example: Float formatting

```
$ printf "Price: $%.2f\n" 49.9
Price: $49.90
```

Formats a number with exactly 2 decimal places.

Example: Hex conversion

```
$ printf "Decimal: %d, Hex: 0x%x, Octal: %o\n" 255 255 255
Decimal: 255, Hex: 0xff, Octal: 377
```

Converts a number to decimal, hexadecimal, and octal.

Tips & Best Practices

Pro Tip: No newline by default: printf does not add \n automatically. Always include \n in your format string: printf "%s\n" "text".

Note: Reusable format: printf reuses the format string for extra arguments: printf "%s," a b c outputs a,b,c, - great for building CSV.

Warning: Quote the format string: Always quote the format string to prevent shell expansion: printf "%s\n" "$var". Without quotes, special characters may be interpreted.

$ read

Intermediate

Read a line of input from standard input

The read command reads a line of input from standard input (keyboard or pipe) and assigns it to one or more variables. It is the primary way to get user input in bash scripts.

read supports prompts, timeouts, silent mode (for passwords), character limits, and reading into arrays. It can read fro...

Options & Flags

`-p`	Display a prompt string
`-s`	Silent mode (for passwords)
`-t`	Timeout in seconds
`-n`	Read only N characters
`-r`	Raw mode (do not interpret backslashes)
`-a`	Read into an array
`-d`	Set delimiter instead of newline

Practical Examples

Example: Simple input

```
$ read -p "Enter your name: " name; echo "Hello, $name"
```

Prompts for input and stores it in the variable $name.

Example: Password input

```
$ read -sp "Password: " pass; echo
```

Reads input without displaying characters. The echo adds a newline after.

Example: Yes/No confirmation

```
$ read -n 1 -p "Continue? [y/n] " answer; [[ "$answer" == "y" ]] && echo " Yes!" || echo " No!"
```

Reads a single character for quick yes/no confirmation.

Example: Read with timeout

```
$ read -t 5 -p "Answer within 5 seconds: " answer || echo "Timeout!"
```

Waits 5 seconds for input, then continues with a timeout message.

Example: Read file line by line

```
$ while IFS= read -r line; do echo "Line: $line"; done < file.txt
```

Processes each line of a file - the standard file reading pattern in bash.

Tips & Best Practices

Pro Tip: Always use -r: Use read -r to prevent backslash interpretation. Without -r, backslashes are treated as escape characters, which can corrupt data.

Note: IFS for custom splitting: Set IFS to split by custom delimiter: IFS=':' read -r user pass uid gid info home shell < /etc/passwd

Warning: Pipe creates subshell: echo "data" | read var - the read runs in a subshell, so $var is empty after. Use: read var <<< "data" or process substitution.

$ seq

Beginner

Print a sequence of numbers

seq generates a sequence of numbers. It prints numbers from a start value to an end value, optionally with a specified increment. seq is commonly used for creating numbered lists, loop counters, and test data.

seq supports integer and floating-point numbers, custom separators, and format strings...

Options & Flags

`LAST`	Print 1 to LAST
`FIRST LAST`	Print FIRST to LAST
`FIRST INCR LAST`	Print with custom increment
`-s`	Set separator (default: newline)
`-w`	Equal width (zero-padded)
`-f`	printf-style format

Practical Examples

Example: Simple sequence

```
$ seq 5
1
2
3
4
```

Prints numbers 1 through 5.

Example: Custom range

```
$ seq 10 15
10
11
12
13
```

Prints numbers 10 through 15.

Example: Custom increment

```
$ seq 0 5 50
```

Prints 0, 5, 10, ... 50 (increment by 5).

Example: Zero-padded

```
$ seq -w 1 100
001\n002\n...
```

Prints 001, 002, ... 100 with consistent width.

Example: Custom separator

```
$ seq -s", " 1 5
1, 2, 3, 4, 5
```

Prints comma-separated sequence.

Tips & Best Practices

Pro Tip: Bash brace expansion: {1..10} generates 1 to 10 without seq. But seq supports floats: seq 0.1 0.1 1.0 and custom formatting.

Note: Floating point: seq handles decimals: seq 0.0 0.5 5.0 generates 0.0, 0.5, 1.0, ... 5.0. Bash brace expansion cannot do this.

Warning: Large sequences: seq can generate millions of numbers. Be careful with: seq 1 1000000 | xargs command - may consume excessive memory.

$ sleep

Beginner

Delay for a specified amount of time

sleep pauses execution for a specified duration. It is one of the simplest yet most useful commands, essential for adding delays in scripts, rate limiting, polling loops, and countdown timers.

sleep accepts durations in seconds (default), minutes (m), hours (h), or days (d). It supports fraction...

Options & Flags

`N`	Sleep for N seconds
`Nm`	Sleep for N minutes
`Nh`	Sleep for N hours
`Nd`	Sleep for N days
`0.N`	Sleep for fractional seconds
`multiple`	Add multiple durations

Practical Examples

Example: Pause for 5 seconds

```
$ sleep 5
```

Pauses execution for 5 seconds.

Example: Retry loop

```
$ until curl -s http://localhost:8080/health; do echo "Waiting..."; sleep 2; done
```

Polls a health endpoint every 2 seconds until it responds.

Example: Rate limiting

```
$ for url in $(cat urls.txt); do curl -s "$url" > /dev/null; sleep 0.5; done
```

Adds a half-second delay between requests to avoid rate limiting.

Example: Countdown timer

```
$ for i in $(seq 10 -1 1); do echo "$i..."; sleep 1; done; echo "Go!"
```

Counts down from 10 to 1 with one-second intervals.

Example: Delayed command

```
$ sleep 5m && notify-send "Break time!"
```

Sends a notification after 5 minutes.

Tips & Best Practices

Pro Tip: Fractional seconds: sleep 0.1 pauses for 100ms. Useful for polling loops and rate limiting without being too slow.

Note: Multiple durations: sleep 1m 30s sleeps for 1 minute and 30 seconds. Durations are added together.

Warning: Not precise for timing: sleep is not a precision timer. System load can cause slight delays. For precise timing, use dedicated scheduling tools.

$ source

Beginner

Execute commands from a file in the current shell

source (or the dot command .) reads and executes commands from a file in the current shell environment. Unlike running a script with bash script.sh, source does not create a subshell - variables and functions defined in the file become part of your current session.

source is essential for loadin...

Options & Flags

`source file`	Execute file in current shell
`. file`	POSIX shorthand for source
`with args`	Pass arguments to sourced file

Practical Examples

Example: Reload shell configuration

```
$ source ~/.bashrc
```

Applies changes to .bashrc without logging out and back in.

Example: Load environment variables

```
$ source .env
```

Loads all variable assignments from .env into the current shell.

Example: Load function library

```
$ source /usr/lib/bash/functions.sh
```

Imports shell functions from a library file for use in the current session.

Example: POSIX dot syntax

```
$ . ~/.profile
```

The dot command is the POSIX-standard equivalent of source.

Example: Apply Python virtualenv

```
$ source venv/bin/activate
```

Activates a Python virtual environment in the current shell.

Tips & Best Practices

Pro Tip: Auto-export with set -a: set -a before source makes all loaded variables automatically exported. set +a turns it off: set -a; source .env; set +a

Warning: source vs bash execution: source modifies YOUR shell. bash script.sh runs in a subshell. If a script sets variables, source it; otherwise the variables are lost.

Note: . is POSIX, source is bash: The . (dot) command works in all POSIX shells. source is a bash extension. For maximum portability, use dot.

$ test

Intermediate

Evaluate conditional expressions for scripting

test evaluates conditional expressions and returns an exit status of 0 (true) or 1 (false). It is the foundation of if statements and conditional logic in shell scripts.

test can be written as test expression or [expression] (the bracket syntax). Bash also provides [[expression]] which is mo...

Options & Flags

`-f`	True if file exists and is a regular file
`-d`	True if directory exists
`-e`	True if file exists (any type)
`-r/-w/-x`	True if file is readable/writable/executable
`-z`	True if string is empty
`-n`	True if string is non-empty
`-eq/-ne/-lt/-gt`	Numeric comparisons
`=`	String equality

Practical Examples

Example: Check if file exists

```
$ [ -f config.yml ] && echo "Config found" || echo "Config missing"
```

Tests if config.yml exists and is a regular file.

Example: Check directory

```
$ [ -d /var/www ] && echo "Directory exists"
```

Tests if the directory exists.

Example: Numeric comparison

```
$ count=15; [ $count -gt 10 ] && echo "More than 10"
More than 10
```

Compares numbers using -gt (greater than).

Example: String comparison

```
$ [ "$USER" = "root" ] && echo "Running as root"
```

Checks if the current user is root.

Example: Empty string check

```
$ [ -z "$API_KEY" ] && echo "API_KEY not set!"
```

Tests if a variable is empty or unset.

Tips & Best Practices

Pro Tip: Always quote variables: Use ["$var" = "value"] with quotes. Without quotes, empty variables cause syntax errors: [= value] fails.

Note: [[vs [in bash: [[is bash-specific but safer - it handles empty variables and supports regex (=~) and glob (*) matching. Prefer [[in bash scripts.

Warning: Spaces are required: [and] need spaces: [-f file] is correct. [-f file] or [-f file] will fail. The brackets are actually commands.

$ trap

Intermediate

Handle signals and cleanup in shell scripts

The trap command sets up signal handlers in shell scripts - executing specified commands when the script receives a signal (like SIGINT from Ctrl+C), exits (EXIT), or encounters an error (ERR). It is one of the most important commands for writing robust, production-quality shell scripts.

The mos...

Options & Flags

`trap COMMAND EXIT`	Execute command when script exits (any reason)
`trap COMMAND SIGINT`	Handle Ctrl+C interrupt
`trap COMMAND SIGTERM`	Handle kill/terminate signal
`trap COMMAND ERR`	Execute on any command error (with set -e)
`trap '' SIGNAL`	Ignore a signal (empty string handler)
`trap - SIGNAL`	Reset signal to default handler
`trap -p`	Print current trap settings

Practical Examples

Example: Cleanup temporary files on exit

```
$ #!/bin/bash\ntmpfile=$(mktemp)\ntrap "rm -f $tmpfile" EXIT\n# script work here...\necho "data" > "$tmpfile"
```

Remove temp file when script exits - whether normally, by Ctrl+C, or on error. The most common and important trap pattern.

Example: Handle Ctrl+C gracefully

```
$ #!/bin/bash\ntrap "echo; echo 'Interrupted. Cleaning up...'; cleanup; exit 130" SIGINT\nfunction cleanup() { rm -f /tmp/lockfile
```

Catch Ctrl+C, run cleanup function, then exit with code 130 (128+SIGINT=2).

Example: Error handling with line number

```
$ #!/bin/bash\nset -euo pipefail\ntrap 'echo "ERROR at line $LINENO: command \\"$BASH_COMMAND\\" failed with exit code $?"' ERR\n#
```

On any error, print the line number and failed command. Combined with set -e for fail-fast behavior.

Example: Release lock file on exit

```
$ #!/bin/bash\nLOCKFILE="/var/run/myscript.lock"\nif [ -f "$LOCKFILE" ]; then echo "Already running"; exit 1; fi\ntouch "$LOCKFILE
```

Create a lock file and ensure it is always removed, even if the script crashes.

Example: Multiple signal handlers

```
$ #!/bin/bash\ntrap "echo Normal exit" EXIT\ntrap "echo Ctrl+C pressed; exit 130" SIGINT\ntrap "echo Terminated; exit 143" SIGTERM
```

Set different handlers for different signals. EXIT trap runs AFTER SIGINT/SIGTERM handlers.

Tips & Best Practices

Pro Tip: Always use trap EXIT for cleanup: trap EXIT runs on ANY exit - normal, error, Ctrl+C, or kill. It is the most reliable way to ensure cleanup. Use it for temp files, lock files, and resource release.

Warning: EXIT trap runs after signal traps: If you have both SIGINT and EXIT traps, on Ctrl+C: SIGINT handler runs first, then EXIT handler. Avoid duplicating cleanup in both handlers.

Note: SIGKILL cannot be trapped: kill -9 (SIGKILL) cannot be trapped or caught by any process. Avoid kill -9 in scripts - use kill (SIGTERM) first to allow clean shutdown.

Pro Tip: Use with set -e: Combine trap ERR with set -e (exit on error) for robust error handling: set -euo pipefail; trap 'echo "Error line $LINENO"' ERR

$ type

Beginner

Show how a command name is interpreted by the shell

type identifies how a command name would be interpreted by the shell. It reveals whether a name is a shell built-in, an alias, a function, an external executable, or a keyword.

type is more comprehensive than which - it knows about shell built-ins (cd, echo), aliases, and functions, while which ...

Options & Flags

`-t`	Show type only (alias, function, builtin, file, keyword)
`-a`	Show all matches (all aliases, functions, and executables)
`-p`	Show executable path (skip aliases/functions)
`-f`	Do not look up functions

Practical Examples

Example: Identify a command

```
$ type ls
ls is aliased to 'ls --color=auto'
```

Shows what 'ls' resolves to.

Example: Show command type only

```
$ type -t cd
builtin
```

Returns just the type category.

Example: Show all definitions

```
$ type -a echo
echo is a shell builtin
echo is /usr/bin/echo
```

Shows all versions of echo - built-in and external.

Example: Check if command exists

```
$ type -t docker > /dev/null 2>&1 && echo "Found" || echo "Not found"
```

Checks if a command is available (any type).

Example: Find executable path

```
$ type -p python3
/usr/bin/python3
```

Shows only the file path, ignoring aliases and functions.

Tips & Best Practices

Pro Tip: type vs which: type knows about aliases, functions, and built-ins. which only finds executables in PATH. type gives you the complete picture.

Note: Debug command resolution: Use type -a command to see everything: aliases, functions, built-ins, AND executables. The first match is what runs.

Warning: Shell-specific: type is a shell built-in and its output varies between bash, zsh, and other shells. The -t flag is the most portable.

$ wait

Intermediate

Wait for background processes to finish

wait pauses the shell until specified background processes complete. Without arguments, it waits for all background jobs. With a PID or job ID, it waits for that specific process.

wait returns the exit status of the waited-for process, making it essential for parallel processing scripts that nee...

Options & Flags

`PID`	Wait for specific process
`%N`	Wait for job number N
`-n`	Wait for any single job to complete (bash 4.3+)
`-p VAR`	Store the PID that completed (bash 5.1+)
`(no args)`	Wait for all background jobs

Practical Examples

Example: Wait for all background jobs

```
$ cmd1 & cmd2 & cmd3 & wait; echo "All done"
```

Runs 3 commands in parallel and waits for all to finish.

Example: Wait for specific process

```
$ long_task & PID=$!; echo "Started $PID"; wait $PID; echo "Exit code: $?"
```

Starts a task, captures its PID, waits for it, and checks its exit status.

Example: Parallel with error checking

```
$ task1 & P1=$!; task2 & P2=$!; wait $P1 || echo "task1 failed"; wait $P2 || echo "task2 failed"
```

Runs tasks in parallel and checks each exit code independently.

Example: Wait for any job

```
$ for i in 1 2 3 4 5; do slow_task $i & done; wait -n; echo "First one done"
```

Starts multiple jobs and continues as soon as any one finishes.

Example: Parallel file processing

```
$ for f in *.csv; do process "$f" & done; wait; echo "All files processed"
```

Processes all CSV files in parallel, then continues.

Tips & Best Practices

Pro Tip: $! captures last background PID: After command &, $! contains its PID. Save it immediately: cmd & PID=$! for use with wait later.

Note: Exit code propagation: wait returns the exit code of the process it waited for. Use $? after wait to check if the background task succeeded.

Warning: Only waits for children: wait can only wait for processes started by the current shell. It cannot wait for arbitrary PIDs from other sessions.

$ yes

Beginner

Output a string repeatedly until killed

yes outputs a string repeatedly until killed. Without arguments, it outputs "y" endlessly. With an argument, it outputs that string repeatedly. yes is primarily used to automatically answer confirmation prompts in scripts.

yes generates output as fast as possible, making it also useful for stres...

Options & Flags

`(no args)`	Output "y" repeatedly
`STRING`	Output STRING repeatedly
`piped`	Auto-answer prompts

Practical Examples

Example: Auto-confirm prompts

```
$ yes | rm -i *.tmp
```

Automatically answers "y" to each rm confirmation prompt.

Example: Custom string

```
$ yes "I accept" | head -3
I accept
I accept
I accept
```

Outputs "I accept" three times.

Example: Auto-accept license

```
$ yes | ./installer.sh
```

Automatically accepts all prompts during installation.

Example: Generate test lines

```
$ yes "test data" | head -1000 > testfile.txt
```

Creates a 1000-line test file.

Example: Answer no

```
$ yes n | dangerous-command
```

Automatically answers "n" to all prompts.

Tips & Best Practices

Pro Tip: Use command flags instead: Many commands have -y or --yes flags: apt install -y, rm -f. These are cleaner than piping yes.

Note: yes is very fast: yes outputs gigabytes per second. Always pipe to a command or limit with head to avoid filling disk/memory.

Warning: Can cause damage: yes | dangerous_command will answer y to ALL prompts without discrimination. Make sure you want to confirm everything.