

NEW: [pure sh bible \(A collection of pure POSIX sh alternatives to external processes\)](#).



pure bash bible

A collection of pure bash alternatives to external processes.



The goal of this book is to document commonly-known and lesser-known methods of doing various tasks using only built-in bash features. Using the snippets from this bible can help remove unneeded dependencies from scripts and in most cases make them faster. I came across these tips and discovered a few while developing [neofetch](#), [pxltrim](#) and other smaller projects.

The snippets below are linted using `shellcheck` and tests have been written where applicable. Want to contribute? Read the [CONTRIBUTING.md](#). It outlines how the unit tests work and what is required when adding snippets to the bible.

See something incorrectly described, buggy or outright wrong? Open an issue or send a pull request. If the bible is missing something, open an issue and a solution will be found.

This book is also available to purchase on leanpub. <https://leanpub.com/bash>

Or you can buy me a coffee.

Table of Contents

- [FOREWORD](#)
- [STRINGS](#)
 - [Trim leading and trailing white-space from string](#)
 - [Trim all white-space from string and truncate spaces](#)
 - [Use regex on a string](#)
 - [Split a string on a delimiter](#)
 - [Change a string to lowercase](#)
 - [Change a string to uppercase](#)
 - [Reverse a string case](#)
 - [Trim quotes from a string](#)
 - [Strip all instances of pattern from string](#)
 - [Strip first occurrence of pattern from string](#)
 - [Strip pattern from start of string](#)
 - [Strip pattern from end of string](#)
 - [Percent-encode a string](#)
 - [Decode a percent-encoded string](#)
 - [Check if string contains a sub-string](#)
 - [Check if string starts with sub-string](#)
 - [Check if string ends with sub-string](#)
- [ARRAYS](#)
 - [Reverse an array](#)
 - [Remove duplicate array elements](#)
 - [Random array element](#)
 - [Cycle through an array](#)
 - [Toggle between two values](#)
- [LOOPS](#)
 - [Loop over a range of numbers](#)
 - [Loop over a variable range of numbers](#)
 - [Loop over an array](#)

- [Loop over an array with an index](#)
- [Loop over the contents of a file](#)
- [Loop over files and directories](#)
- [FILE HANDLING](#)
 - [Read a file to a string](#)
 - [Read a file to an array \(*by line*\)](#)
 - [Get the first N lines of a file](#)
 - [Get the last N lines of a file](#)
 - [Get the number of lines in a file](#)
 - [Count files or directories in directory](#)
 - [Create an empty file](#)
 - [Extract lines between two markers](#)
- [FILE PATHS](#)
 - [Get the directory name of a file path](#)
 - [Get the base-name of a file path](#)
- [VARIABLES](#)
 - [Assign and access a variable using a variable](#)
 - [Name a variable based on another variable](#)
- [ESCAPE SEQUENCES](#)
 - [Text Colors](#)
 - [Text Attributes](#)
 - [Cursor Movement](#)
 - [Erasing Text](#)
- [PARAMETER EXPANSION](#)
 - [Indirection](#)
 - [Replacement](#)
 - [Length](#)
 - [Expansion](#)
 - [Case Modification](#)
 - [Default Value](#)
- [BRACE EXPANSION](#)
 - [Ranges](#)
 - [String Lists](#)
- [CONDITIONAL EXPRESSIONS](#)
 - [File Conditionals](#)
 - [File Comparisons](#)
 - [Variable Conditionals](#)
 - [Variable Comparisons](#)
- [ARITHMETIC OPERATORS](#)
 - [Assignment](#)
 - [Arithmetic](#)
 - [Bitwise](#)
 - [Logical](#)
 - [Miscellaneous](#)
- [ARITHMETIC](#)
 - [Simpler syntax to set variables](#)
 - [Ternary Tests](#)
- [TRAPS](#)
 - [Do something on script exit](#)
 - [Ignore terminal interrupt \(CTRL+C, SIGINT\)](#)
 - [React to window resize](#)
 - [Do something before every command](#)
 - [Do something when a shell function or a sourced file finishes executing](#)
- [PERFORMANCE](#)
 - [Disable Unicode](#)
- [OBSOLETE SYNTAX](#)
 - [Shebang](#)
 - [Command Substitution](#)
 - [Function Declaration](#)
- [INTERNAL VARIABLES](#)
 - [Get the location to the bash binary](#)
 - [Get the version of the current running bash process](#)
 - [Open the user's preferred text editor](#)
 - [Get the name of the current function](#)
 - [Get the host-name of the system](#)
 - [Get the architecture of the Operating System](#)
 - [Get the name of the Operating System / Kernel](#)

- [Get the current working directory](#)
- [Get the number of seconds the script has been running](#)
- [Get a pseudorandom integer](#)
- [INFORMATION ABOUT THE TERMINAL](#)
 - [Get the terminal size in lines and columns \(from a script\)](#)
 - [Get the terminal size in pixels](#)
 - [Get the current cursor position](#)
- [CONVERSION](#)
 - [Convert a hex color to RGB](#)
 - [Convert an RGB color to hex](#)
- [CODE GOLF](#)
 - [Shorter for loop syntax](#)
 - [Shorter infinite loops](#)
 - [Shorter function declaration](#)
 - [Shorter if syntax](#)
 - [Simpler case statement to set variable](#)
- [OTHER](#)
 - [Use read as an alternative to the sleep command](#)
 - [Check if a program is in the user's PATH](#)
 - [Get the current date using strftime](#)
 - [Get the username of the current user](#)
 - [Generate a UUID V4](#)
 - [Progress bars](#)
 - [Get the list of functions in a script](#)
 - [Bypass shell aliases](#)
 - [Bypass shell functions](#)
 - [Run a command in the background](#)
 - [Capture function return without command substitution](#)
- [AFTERWORD](#)

FOREWORD

A collection of pure bash alternatives to external processes and programs. The bash scripting language is more powerful than people realise and most tasks can be accomplished without depending on external programs.

Calling an external process in bash is expensive and excessive use will cause a noticeable slowdown. Scripts and programs written using built-in methods (*where applicable*) will be faster, require fewer dependencies and afford a better understanding of the language itself.

The contents of this book provide a reference for solving problems encountered when writing programs and scripts in bash. Examples are in function formats showcasing how to incorporate these solutions into code.

STRINGS

Trim leading and trailing white-space from string

This is an alternative to sed, awk, perl and other tools. The function below works by finding all leading and trailing white-space and removing it from the start and end of the string. The : built-in is used in place of a temporary variable.

Example Function:

```
trim_string() {
    # Usage: trim_string " example string "
    : "${1#"${1%[![:space:]]*}"}"
    : "${_%"${_##*![:space:]}"}"
    printf '%s\n' "$_"
}
```

Example Usage:

```
$ trim_string " Hello, World "
Hello, World
```

```
$ name=" John Black "
$ trim_string "$name"
```

Trim all white-space from string and truncate spaces

This is an alternative to sed, awk, perl and other tools. The function below works by abusing word splitting to create a new string without leading/trailing white-space and with truncated spaces.

Example Function:

```
# shellcheck disable=SC2086,SC2048
trim_all() {
    # Usage: trim_all " example string "
    set -f
    set -- $*
    printf '%s\n' "$*"
    set +f
}
```

Example Usage:

```
$ trim_all " Hello, World "
Hello, World

$ name=" John Black is my name. "
$ trim_all "$name"
John Black is my name.
```

Use regex on a string

The result of bash's regex matching can be used to replace sed for a large number of use-cases.

CAVEAT: This is one of the few platform dependent bash features. bash will use whatever regex engine is installed on the user's system. Stick to POSIX regex features if aiming for compatibility.

CAVEAT: This example only prints the first matching group. When using multiple capture groups some modification is needed.

Example Function:

```
regex() {
    # Usage: regex "string" "regex"
    [[ $1 =~ $2 ]] && printf '%s\n' "${BASH_REMATCH[1]}"
}
```

Example Usage:

```
$ # Trim leading white-space.
$ regex ' hello' '^\s*(.*)'
hello

$ # Validate a hex color.
$ regex "#FFFFFF" '^(#?([a-fA-F0-9]{6}|[a-fA-F0-9]{3}))$'
#FFFFFF

$ # Validate a hex color (invalid).
$ regex "red" '^(#?([a-fA-F0-9]{6}|[a-fA-F0-9]{3}))$'
# no output (invalid)
```

Example Usage in script:

```
is_hex_color() {
    if [[ $1 =~ ^(#?([a-fA-F0-9]{6}|[a-fA-F0-9]{3}))$ ]]; then
        printf '%s\n' "${BASH_REMATCH[1]}"
    else
        printf '%s\n' "error: $1 is an invalid color."
        return 1
    fi
}
```

```
}  
  
read -r color  
is_hex_color "$color" || color="#FFFFFF"  
  
# Do stuff.
```

Split a string on a delimiter

CAVEAT: Requires bash 4+

This is an alternative to cut, awk and other tools.

Example Function:

```
split() {  
    # Usage: split "string" "delimiter"  
    IFS=$'\n' read -d "" -ra arr <<< "${1//$2/$'\n'}"  
    printf '%s\n' "${arr[@]}"  
}
```

Example Usage:

```
$ split "apples,oranges,pears,grapes" ","  
apples  
oranges  
pears  
grapes
```

```
$ split "1, 2, 3, 4, 5" ", "  
1  
2  
3  
4  
5
```

```
# Multi char delimiters work too!  
$ split "hello---world---my---name---is---john" "---"  
hello  
world  
my  
name  
is  
john
```

Change a string to lowercase

CAVEAT: Requires bash 4+

Example Function:

```
lower() {  
    # Usage: lower "string"  
    printf '%s\n' "${1,,}"  
}
```

Example Usage:

```
$ lower "HELLO"  
hello
```

```
$ lower "HeLlO"  
hello
```

```
$ lower "hello"  
hello
```

Change a string to uppercase

CAVEAT: Requires bash 4+

Example Function:

```
upper() {  
    # Usage: upper "string"  
    printf '%s\n' "${1^^}"  
}
```

Example Usage:

```
$ upper "hello"  
HELLO
```

```
$ upper "HeLlO"  
HELLO
```

```
$ upper "HELLO"  
HELLO
```

Reverse a string case

CAVEAT: Requires bash 4+

Example Function:

```
reverse_case() {  
    # Usage: reverse_case "string"  
    printf '%s\n' "${1~~}"  
}
```

Example Usage:

```
$ reverse_case "hello"  
HELLO
```

```
$ reverse_case "HeLlO"  
hElLo
```

```
$ reverse_case "HELLO"  
hello
```

Trim quotes from a string

Example Function:

```
trim_quotes() {  
    # Usage: trim_quotes "string"  
    : "${1//\'}"  
    printf '%s\n' "${_//\'}"  
}
```

Example Usage:

```
$ var="'Hello', \"World\""  
$ trim_quotes "$var"  
Hello, World
```

Strip all instances of pattern from string

Example Function:

```
strip_all() {
    # Usage: strip_all "string" "pattern"
    printf '%s\n' "${1//$2}"
}
```

Example Usage:

```
$ strip_all "The Quick Brown Fox" "[aeiou]"
Th Qck Brwn Fx
```

```
$ strip_all "The Quick Brown Fox" "[[:space:]]"
TheQuickBrownFox
```

```
$ strip_all "The Quick Brown Fox" "Quick "
The Brown Fox
```

Strip first occurrence of pattern from string

Example Function:

```
strip() {
    # Usage: strip "string" "pattern"
    printf '%s\n' "${1/$2}"
}
```

Example Usage:

```
$ strip "The Quick Brown Fox" "[aeiou]"
Th Quick Brown Fox
```

```
$ strip "The Quick Brown Fox" "[[:space:]]"
TheQuick Brown Fox
```

Strip pattern from start of string

Example Function:

```
lstrip() {
    # Usage: lstrip "string" "pattern"
    printf '%s\n' "${1##$2}"
}
```

Example Usage:

```
$ lstrip "The Quick Brown Fox" "The "
Quick Brown Fox
```

Strip pattern from end of string

Example Function:

```
rstrip() {
    # Usage: rstrip "string" "pattern"
    printf '%s\n' "${1%%$2}"
}
```

Example Usage:

```
$ rstrip "The Quick Brown Fox" " Fox"
The Quick Brown
```

Percent-encode a string

Example Function:

```
urlencode() {
    # Usage: urlencode "string"
    local LC_ALL=C
    for (( i = 0; i < ${#1}; i++ )); do
        : "${1:i:1}"
        case "$_" in
            [a-zA-Z0-9.~_-])
                printf '%s' "$_"
                ;;
            *)
                printf '%%%02X' "'$_"
                ;;
        esac
    done
    printf '\n'
}
```

Example Usage:

```
$ urlencode "https://github.com/dylanaraps/pure-bash-bible"
https%3A%2F%2Fgithub.com%2Fdylanaraps%2Fpure-bash-bible
```

Decode a percent-encoded string

Example Function:

```
urldecode() {
    # Usage: urldecode "string"
    : "${1//+/ }"
    printf '%b\n' "${_//%/\\x}"
}
```

Example Usage:

```
$ urldecode "https%3A%2F%2Fgithub.com%2Fdylanaraps%2Fpure-bash-bible"
https://github.com/dylanaraps/pure-bash-bible
```

Check if string contains a sub-string

Using a test:

```
if [[ $var == *sub_string* ]]; then
    printf '%s\n' "sub_string is in var."
fi

# Inverse (substring not in string).
if [[ $var != *sub_string* ]]; then
    printf '%s\n' "sub_string is not in var."
fi

# This works for arrays too!
if [[ ${arr[*]} == *sub_string* ]]; then
    printf '%s\n' "sub_string is in array."
fi
```

Using a case statement:

```
case "$var" in
    *sub_string*)
        # Do stuff
        ;;
```

```

    *sub_string2*)
        # Do more stuff
    ;;

    *)
        # Else
    ;;
esac

```

Check if string starts with sub-string

```

if [[ $var == sub_string* ]]; then
    printf '%s\n' "var starts with sub_string."
fi

# Inverse (var does not start with sub_string).
if [[ $var != sub_string* ]]; then
    printf '%s\n' "var does not start with sub_string."
fi

```

Check if string ends with sub-string

```

if [[ $var == *sub_string ]]; then
    printf '%s\n' "var ends with sub_string."
fi

# Inverse (var does not end with sub_string).
if [[ $var != *sub_string ]]; then
    printf '%s\n' "var does not end with sub_string."
fi

```

ARRAYS

Reverse an array

Enabling extdebug allows access to the BASH_ARGV array which stores the current function's arguments in reverse.

CAVEAT: Requires shopt -s compat44 in bash 5.0+.

Example Function:

```

reverse_array() {
    # Usage: reverse_array "array"
    shopt -s extdebug
    f()(printf '%s\n' "${BASH_ARGV[@}"); f "$@"
    shopt -u extdebug
}

```

Example Usage:

```

$ reverse_array 1 2 3 4 5
5
4
3
2
1

$ arr=(red blue green)
$ reverse_array "${arr[@]}"
green
blue
red

```

Remove duplicate array elements

Create a temporary associative array. When setting associative array values and a duplicate assignment occurs, bash overwrites the key. This allows us to effectively remove array duplicates.

CAVEAT: Requires bash 4+

CAVEAT: List order may not stay the same.

Example Function:

```
remove_array_dups() {
    # Usage: remove_array_dups "array"
    declare -A tmp_array

    for i in "$@"; do
        [[ $i ]] && IFS=" " tmp_array["${i:- }"]=1
    done

    printf '%s\n' "${!tmp_array[@]}"
}
```

Example Usage:

```
$ remove_array_dups 1 1 2 2 3 3 3 3 4 4 4 4 5 5 5 5 5
1
2
3
4
5
```

```
$ arr=(red red green blue blue)
$ remove_array_dups "${arr[@]}"
red
green
blue
```

Random array element

Example Function:

```
random_array_element() {
    # Usage: random_array_element "array"
    local arr=("$@")
    printf '%s\n' "${arr[RANDOM % $#]}"
}
```

Example Usage:

```
$ array=(red green blue yellow brown)
$ random_array_element "${array[@]}"
yellow
```

```
# Multiple arguments can also be passed.
$ random_array_element 1 2 3 4 5 6 7
3
```

Cycle through an array

Each time the `printf` is called, the next array element is printed. When the print hits the last array element it starts from the first element again.

```
arr=(a b c d)
```

```
cycle() {
    printf '%s ' "${arr[${i:=0}]}"
    ((i>=${#arr[@]}-1?0:++i))
}
```

```
}
```

Toggle between two values

This works the same as above, this is just a different use case.

```
arr=(true false)

cycle() {
    printf '%s ' "${arr[${i:=0}]}"
    ((i>=${#arr[@]}-1?0:++i))
}
```

LOOPS

Loop over a range of numbers

Alternative to seq.

```
# Loop from 0-100 (no variable support).
for i in {0..100}; do
    printf '%s\n' "$i"
done
```

Loop over a variable range of numbers

Alternative to seq.

```
# Loop from 0-VAR.
VAR=50
for ((i=0;i<=VAR;i++)); do
    printf '%s\n' "$i"
done
```

Loop over an array

```
arr=(apples oranges tomatoes)

# Just elements.
for element in "${arr[@]}; do
    printf '%s\n' "$element"
done
```

Loop over an array with an index

```
arr=(apples oranges tomatoes)

# Elements and index.
for i in "${!arr[@]}; do
    printf '%s\n' "${arr[i]}"
done

# Alternative method.
for ((i=0;i<${#arr[@]};i++)); do
    printf '%s\n' "${arr[i]}"
done
```

Loop over the contents of a file

```
while read -r line; do
    printf '%s\n' "$line"
done
```

```
done < "file"
```

Loop over files and directories

Don't use ls.

```
# Greedy example.
for file in *; do
    printf '%s\n' "$file"
done

# PNG files in dir.
for file in ~/Pictures/*.png; do
    printf '%s\n' "$file"
done

# Iterate over directories.
for dir in ~/Downloads/*; do
    printf '%s\n' "$dir"
done

# Brace Expansion.
for file in /path/to/parentdir/{file1,file2,subdir/file3}; do
    printf '%s\n' "$file"
done

# Iterate recursively.
shopt -s globstar
for file in ~/Pictures/**/*; do
    printf '%s\n' "$file"
done
shopt -u globstar
```

FILE HANDLING

CAVEAT: bash does not handle binary data properly in versions < 4.4.

Read a file to a string

Alternative to the cat command.

```
file_data="$(<"file")"
```

Read a file to an array (*by line*)

Alternative to the cat command.

```
# Bash <4 (discarding empty lines).
IFS=$'\n' read -d "" -ra file_data < "file"

# Bash <4 (preserving empty lines).
while read -r line; do
    file_data+=("$line")
done < "file"

# Bash 4+
mapfile -t file_data < "file"
```

Get the first N lines of a file

Alternative to the head command.

CAVEAT: Requires bash 4+

Example Function:

```
head() {
    # Usage: head "n" "file"
    mapfile -tn "$1" line < "$2"
    printf '%s\n' "${line[@]}"
}
```

Example Usage:

```
$ head 2 ~/.bashrc
# Prompt
PS1='> '

$ head 1 ~/.bashrc
# Prompt
```

Get the last N lines of a file

Alternative to the tail command.

CAVEAT: Requires bash 4+

Example Function:

```
tail() {
    # Usage: tail "n" "file"
    mapfile -tn 0 line < "$2"
    printf '%s\n' "${line[@]: -$1}"
}
```

Example Usage:

```
$ tail 2 ~/.bashrc
# Enable tmux.
# [[ -z "$TMUX" ]] && exec tmux

$ tail 1 ~/.bashrc
# [[ -z "$TMUX" ]] && exec tmux
```

Get the number of lines in a file

Alternative to wc -l.

Example Function (bash 4):

```
lines() {
    # Usage: lines "file"
    mapfile -tn 0 lines < "$1"
    printf '%s\n' "${#lines[@]}"
}
```

Example Function (bash 3):

This method uses less memory than the mapfile method and works in bash 3 but it is slower for bigger files.

```
lines_loop() {
    # Usage: lines_loop "file"
    count=0
    while IFS= read -r _; do
        ((count++))
    done < "$1"
    printf '%s\n' "$count"
}
```

Example Usage:

```
$ lines ~/.bashrc
48
```

```
$ lines_loop ~/.bashrc
48
```

Count files or directories in directory

This works by passing the output of the glob to the function and then counting the number of arguments.

Example Function:

```
count() {
    # Usage: count /path/to/dir/*
    #         count /path/to/dir/*/
    printf '%s\n' "$#"
}
```

Example Usage:

```
# Count all files in dir.
$ count ~/Downloads/*
232
```

```
# Count all dirs in dir.
$ count ~/Downloads/*/
45
```

```
# Count all jpg files in dir.
$ count ~/Pictures/*.jpg
64
```

Create an empty file

Alternative to touch.

```
# Shortest.
>file
```

```
# Longer alternatives:
:>file
echo -n >file
printf '' >file
```

Extract lines between two markers

Example Function:

```
extract() {
    # Usage: extract file "opening marker" "closing marker"
    while IFS=$'\n' read -r line; do
        [[ $extract && $line != "$3" ]] &&
            printf '%s\n' "$line"

        [[ $line == "$2" ]] && extract=1
        [[ $line == "$3" ]] && extract=
    done < "$1"
}
```

Example Usage:

```
# Extract code blocks from Markdown file.
$ extract ~/projects/pure-bash/README.md '`sh`'
# Output here...
```

FILE PATHS

Get the directory name of a file path

Alternative to the `dirname` command.

Example Function:

```
dirname() {
    # Usage: dirname "path"
    local tmp=${1:-.}

    [[ $tmp != *[/]* ]] && {
        printf '/\n'
        return
    }

    tmp=${tmp%*"${tmp##*[!/]}" }

    [[ $tmp != */* ]] && {
        printf '\n'
        return
    }

    tmp=${tmp%/*}
    tmp=${tmp%*"${tmp##*[!/]}" }

    printf '%s\n' "${tmp:-/}"
}
```

Example Usage:

```
$ dirname ~/Pictures/Wallpapers/1.jpg
/home/black/Pictures/Wallpapers
```

```
$ dirname ~/Pictures/Downloads/
/home/black/Pictures
```

Get the base-name of a file path

Alternative to the `basename` command.

Example Function:

```
basename() {
    # Usage: basename "path" ["suffix"]
    local tmp

    tmp=${1%"${1##*[!/]}" }
    tmp=${tmp##*/}
    tmp=${tmp%"${2/"$tmp"}" }

    printf '%s\n' "${tmp:-/}"
}
```

Example Usage:

```
$ basename ~/Pictures/Wallpapers/1.jpg
1.jpg
```

```
$ basename ~/Pictures/Wallpapers/1.jpg .jpg
1
```

```
$ basename ~/Pictures/Downloads/
Downloads
```

VARIABLES

Assign and access a variable using a variable

```
$ hello_world="value"

# Create the variable name.
$ var="world"
$ ref="hello_${var}"

# Print the value of the variable name stored in 'hello_${var}'.
$ printf '%s\n' "${!ref}"
value
```

Alternatively, on bash 4.3+:

```
$ hello_world="value"
$ var="world"

# Declare a nameref.
$ declare -n ref=hello_${var}

$ printf '%s\n' "${ref}"
value
```

Name a variable based on another variable

```
$ var="world"
$ declare "hello_${var}=value"
$ printf '%s\n' "${hello_world}"
value
```

ESCAPE SEQUENCES

Contrary to popular belief, there is no issue in utilizing raw escape sequences. Using `tput` abstracts the same ANSI sequences as if printed manually. Worse still, `tput` is not actually portable. There are a number of `tput` variants each with different commands and syntaxes (*try `tput setaf 3 on a FreeBSD system`*). Raw sequences are fine.

Text Colors

NOTE: Sequences requiring RGB values only work in True-Color Terminal Emulators.

Sequence	What does it do?	Value
<code>\e[38;5;<NUM>m</code>	Set text foreground color.	0-255
<code>\e[48;5;<NUM>m</code>	Set text background color.	0-255
<code>\e[38;2;<R>;<G>;m</code>	Set text foreground color to RGB color.	R, G, B
<code>\e[48;2;<R>;<G>;m</code>	Set text background color to RGB color.	R, G, B

Text Attributes

NOTE: Prepend 2 to any code below to turn its effect off (examples: 21=bold text off, 22=faint text off, 23=italic text off).

Sequence	What does it do?
<code>\e[m</code>	Reset text formatting and colors.
<code>\e[1m</code>	Bold text.
<code>\e[2m</code>	Faint text.
<code>\e[3m</code>	Italic text.
<code>\e[4m</code>	Underline text.
<code>\e[5m</code>	Blinking text.
<code>\e[7m</code>	Highlighted text.
<code>\e[8m</code>	Hidden text.
<code>\e[9m</code>	Strike-through text.

Cursor Movement

Sequence	What does it do?	Value
<code>\e[<LINE>;<COLUMN>H</code>	Move cursor to absolute position.	line, column
<code>\e[H</code>	Move cursor to home position (0, 0).	
<code>\e[<NUM>A</code>	Move cursor up N lines.	num
<code>\e[<NUM>B</code>	Move cursor down N lines.	num
<code>\e[<NUM>C</code>	Move cursor right N columns.	num
<code>\e[<NUM>D</code>	Move cursor left N columns.	num
<code>\e[s</code>	Save cursor position.	
<code>\e[u</code>	Restore cursor position.	

Erasing Text

Sequence	What does it do?
<code>\e[K</code>	Erase from cursor position to end of line.
<code>\e[1K</code>	Erase from cursor position to start of line.
<code>\e[2K</code>	Erase the entire current line.
<code>\e[J</code>	Erase from the current line to the bottom of the screen.
<code>\e[1J</code>	Erase from the current line to the top of the screen.
<code>\e[2J</code>	Clear the screen.
<code>\e[2J\e[H</code>	Clear the screen and move cursor to 0, 0.

PARAMETER EXPANSION

Indirection

Parameter	What does it do?
<code>\${!VAR}</code>	Access a variable based on the value of VAR.
<code>\${!VAR*}</code>	Expand to IFS separated list of variable names starting with VAR.
<code>\${!VAR@}</code>	Expand to IFS separated list of variable names starting with VAR. If double-quoted, each variable name expands to a separate word.

Replacement

Parameter	What does it do?
<code>\${VAR#PATTERN}</code>	Remove shortest match of pattern from start of string.
<code>\${VAR##PATTERN}</code>	Remove longest match of pattern from start of string.
<code>\${VAR%PATTERN}</code>	Remove shortest match of pattern from end of string.
<code>\${VAR%%PATTERN}</code>	Remove longest match of pattern from end of string.
<code>\${VAR/PATTERN/REPLACE}</code>	Replace first match with string.
<code>\${VAR//PATTERN/REPLACE}</code>	Replace all matches with string.
<code>\${VAR/PATTERN}</code>	Remove first match.
<code>\${VAR//PATTERN}</code>	Remove all matches.

Length

Parameter	What does it do?
<code>\${#VAR}</code>	Length of var in characters.
<code>\${#ARR[@]}</code>	Length of array in elements.

Expansion

Parameter	What does it do?
<code>\${VAR:OFFSET}</code>	Remove first N chars from variable.
<code>\${VAR:OFFSET:LENGTH}</code>	Get substring from N character to N character. (<code>\${VAR:10:10}</code>): Get sub-string from char 10 to char 20)
<code>\${VAR::OFFSET}</code>	Get first N chars from variable.
<code>\${VAR::-OFFSET}</code>	Remove last N chars from variable.
<code>\${VAR:-OFFSET}</code>	Get last N chars from variable.
<code>\${VAR:OFFSET:-OFFSET}</code>	Cut first N chars and last N chars. bash 4.2+

Case Modification

Parameter	What does it do?	CAVEAT
<code>\${VAR^}</code>	Uppercase first character.	bash 4+
<code>\${VAR^^}</code>	Uppercase all characters.	bash 4+

Parameter	What does it do?	CAVEAT
<code>\${VAR,}</code>	Lowercase first character.	bash 4+
<code>\${VAR,,}</code>	Lowercase all characters.	bash 4+
<code>\${VAR~}</code>	Reverse case of first character.	bash 4+
<code>\${VAR~~}</code>	Reverse case of all characters.	bash 4+

Default Value

Parameter	What does it do?
<code>\${VAR:-STRING}</code>	If VAR is empty or unset, use STRING as its value.
<code>\${VAR-STRING}</code>	If VAR is unset, use STRING as its value.
<code>\${VAR:=STRING}</code>	If VAR is empty or unset, set the value of VAR to STRING.
<code>\${VAR=STRING}</code>	If VAR is unset, set the value of VAR to STRING.
<code>\${VAR:+STRING}</code>	If VAR is not empty, use STRING as its value.
<code>\${VAR+STRING}</code>	If VAR is set, use STRING as its value.
<code>\${VAR:?STRING}</code>	Display an error if empty or unset.
<code>\${VAR?STRING}</code>	Display an error if unset.

BRACE EXPANSION

Ranges

```
# Syntax: {<START>..<END>}

# Print numbers 1-100.
echo {1..100}

# Print range of floats.
echo 1.{1..9}

# Print chars a-z.
echo {a..z}
echo {A..Z}

# Nesting.
echo {A..Z}{0..9}

# Print zero-padded numbers.
# CAVEAT: bash 4+
echo {01..100}

# Change increment amount.
# Syntax: {<START>..<END>..<INCREMENT>}
# CAVEAT: bash 4+
echo {1..10..2} # Increment by 2.
```

String Lists

```
echo {apples,oranges,pears,grapes}

# Example Usage:
# Remove dirs Movies, Music and ISOS from ~/Downloads/.
rm -rf ~/Downloads/{Movies,Music,ISOS}
```

CONDITIONAL EXPRESSIONS

File Conditionals

Expression	Value	What does it do?
-a	file	If file exists.
-b	file	If file exists and is a block special file.
-c	file	If file exists and is a character special file.

Expression	Value	What does it do?
-d	file	If file exists and is a directory.
-e	file	If file exists.
-f	file	If file exists and is a regular file.
-g	file	If file exists and its set-group-id bit is set.
-h	file	If file exists and is a symbolic link.
-k	file	If file exists and its sticky-bit is set
-p	file	If file exists and is a named pipe (<i>FIFO</i>).
-r	file	If file exists and is readable.
-s	file	If file exists and its size is greater than zero.
-t	fd	If file descriptor is open and refers to a terminal.
-u	file	If file exists and its set-user-id bit is set.
-w	file	If file exists and is writable.
-x	file	If file exists and is executable.
-G	file	If file exists and is owned by the effective group ID.
-L	file	If file exists and is a symbolic link.
-N	file	If file exists and has been modified since last read.
-O	file	If file exists and is owned by the effective user ID.
-S	file	If file exists and is a socket.

File Comparisons

Expression	Value	What does it do?
file -ef file2		If both files refer to the same inode and device numbers.
file -nt file2		If file is newer than file2 (<i>uses modification time</i>) or file exists and file2 does not.
file -ot file2		If file is older than file2 (<i>uses modification time</i>) or file2 exists and file does not.

Variable Conditionals

Expression	Value	What does it do?
-o	opt	If shell option is enabled.
-v	var	If variable has a value assigned.
-R	var	If variable is a name reference.
-z	var	If the length of string is zero.
-n	var	If the length of string is non-zero.

Variable Comparisons

Expression	Value	What does it do?
var = var2		Equal to.
var == var2		Equal to (<i>synonym for =</i>).
var != var2		Not equal to.
var < var2		Less than (<i>in ASCII alphabetical order</i>).
var > var2		Greater than (<i>in ASCII alphabetical order</i>).

ARITHMETIC OPERATORS

Assignment

Operators	What does it do?
=	Initialize or change the value of a variable.

Arithmetic

Operators	What does it do?
+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Exponentiation
%	Modulo
+=	Plus-Equal (<i>Increment a variable.</i>)
-=	Minus-Equal (<i>Decrement a variable.</i>)

Operators	What does it do?
*=	Times-Equal (<i>Multiply a variable.</i>)
/=	Slash-Equal (<i>Divide a variable.</i>)
%=	Mod-Equal (<i>Remainder of dividing a variable.</i>)

Bitwise

Operators What does it do?

<<	Bitwise Left Shift
<<=	Left-Shift-Equal
>>	Bitwise Right Shift
>>=	Right-Shift-Equal
&	Bitwise AND
&=	Bitwise AND-Equal
	Bitwise OR
=	Bitwise OR-Equal
~	Bitwise NOT
^	Bitwise XOR
^=	Bitwise XOR-Equal

Logical

Operators What does it do?

!	NOT
&&	AND
	OR

Miscellaneous

Operators	What does it do?	Example
,	Comma Separator	((a=1 , b=2 , c=3))

ARITHMETIC

Simpler syntax to set variables

```
# Simple math
((var=1+2))

# Decrement/Increment variable
((var++))
((var--))
((var+=1))
((var-=1))

# Using variables
((var=var2*arr[2]))
```

Ternary Tests

```
# Set the value of var to var2 if var2 is greater than var.
# var: variable to set.
# var2>var: Condition to test.
# ?var2: If the test succeeds.
# :var: If the test fails.
((var=var2>var?var2:var))
```

TRAPS

Traps allow a script to execute code on various signals. In [pxlterm](#) (a pixel art editor written in bash) traps are used to redraw the user interface on window resize. Another use case is cleaning up temporary files on script exit.

Traps should be added near the start of scripts so any early errors are also caught.

NOTE: For a full list of signals, see `trap -l`.

Do something on script exit

```
# Clear screen on script exit.
trap 'printf \e[2J\e[H\e[m' EXIT
```

Ignore terminal interrupt (CTRL+C, SIGINT)

```
trap '' INT
```

React to window resize

```
# Call a function on window resize.
trap 'code_here' SIGWINCH
```

Do something before every command

```
trap 'code_here' DEBUG
```

Do something when a shell function or a sourced file finishes executing

```
trap 'code_here' RETURN
```

PERFORMANCE

Disable Unicode

If unicode is not required, it can be disabled for a performance increase. Results may vary however there have been noticeable improvements in [neofetch](#) and other programs.

```
# Disable unicode.
LC_ALL=C
LANG=C
```

OBSOLETE SYNTAX

Shebang

Use `#!/usr/bin/env bash` instead of `#!/bin/bash`.

- The former searches the user's PATH to find the bash binary.
- The latter assumes it is always installed to `/bin/` which can cause issues.

NOTE: There are times when one may have a good reason for using `#!/bin/bash` or another direct path to the binary.

Right:

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

Less right:

```
#!/bin/bash
```

Command Substitution

Use `$()` instead of `` ``.

Right.

```
var="$(command)"

# Wrong.
var=`command`

# $() can easily be nested whereas `` cannot.
var="$(command "$(command))"
```

Function Declaration

Do not use the function keyword, it reduces compatibility with older versions of bash.

```
# Right.
do_something() {
    # ...
}

# Wrong.
function do_something() {
    # ...
}
```

INTERNAL VARIABLES

Get the location to the bash binary

```
"$BASH"
```

Get the version of the current running bash process

```
# As a string.
"$BASH_VERSION"

# As an array.
"${BASH_VERSINFO[@]}"
```

Open the user's preferred text editor

```
"$EDITOR" "$file"

# NOTE: This variable may be empty, set a fallback value.
"${EDITOR:-vi}" "$file"
```

Get the name of the current function

```
# Current function.
"${FUNCNAME[0]}"

# Parent function.
"${FUNCNAME[1]}"

# So on and so forth.
"${FUNCNAME[2]}"
"${FUNCNAME[3]}"

# All functions including parents.
"${FUNCNAME[@]}"
```

Get the host-name of the system

```
"$HOSTNAME"
```

```
# NOTE: This variable may be empty.  
# Optionally set a fallback to the hostname command.  
"${HOSTNAME:-$(hostname)}"
```

Get the architecture of the Operating System

```
"$HOSTTYPE"
```

Get the name of the Operating System / Kernel

This can be used to add conditional support for different Operating Systems without needing to call uname.

```
"$OSTYPE"
```

Get the current working directory

This is an alternative to the pwd built-in.

```
"$PWD"
```

Get the number of seconds the script has been running

```
"$SECONDS"
```

Get a pseudorandom integer

Each time \$RANDOM is used, a different integer between 0 and 32767 is returned. This variable should not be used for anything related to security (*this includes encryption keys etc*).

```
"$RANDOM"
```

INFORMATION ABOUT THE TERMINAL

Get the terminal size in lines and columns (*from a script*)

This is handy when writing scripts in pure bash and stty/tput can't be called.

Example Function:

```
get_term_size() {  
    # Usage: get_term_size  
  
    # (;;;) is a micro sleep to ensure the variables are  
    # exported immediately.  
    shopt -s checkwinsize; (;;;)  
    printf '%s\n' "$LINES $COLUMNS"  
}
```

Example Usage:

```
# Output: LINES COLUMNS  
$ get_term_size  
15 55
```

Get the terminal size in pixels

CAVEAT: This does not work in some terminal emulators.

Example Function:

```
get_window_size() {
    # Usage: get_window_size
    printf '%b' "${TMUX:+\\ePtmux;\\e}\\e[14t${TMUX:+\\e\\}\\}"
    IFS=';t' read -d t -t 0.05 -sra term_size
    printf '%s\n' "${term_size[1]}x${term_size[2]}"
}
```

Example Usage:

```
# Output: WIDTHxHEIGHT
$ get_window_size
1200x800
```

```
# Output (fail):
$ get_window_size
x
```

Get the current cursor position

This is useful when creating a TUI in pure bash.

Example Function:

```
get_cursor_pos() {
    # Usage: get_cursor_pos
    IFS=';'; read -p $'\e[6n' -d R -rs _ y x _
    printf '%s\n' "$x $y"
}
```

Example Usage:

```
# Output: X Y
$ get_cursor_pos
1 8
```

CONVERSION

Convert a hex color to RGB

Example Function:

```
hex_to_rgb() {
    # Usage: hex_to_rgb "#FFFFFF"
    #           hex_to_rgb "000000"
    : "${1/#/}"
    ((r=16#${_}:0:2},g=16#${_}:2:2},b=16#${_}:4:2}))
    printf '%s\n' "$r $g $b"
}
```

Example Usage:

```
$ hex_to_rgb "#FFFFFF"
255 255 255
```

Convert an RGB color to hex

Example Function:

```
rgb_to_hex() {
    # Usage: rgb_to_hex "r" "g" "b"
    printf '#%02x%02x%02x\n' "$1" "$2" "$3"
}
```

Example Usage:

```
$ rgb_to_hex "255" "255" "255"  
#FFFFFF
```

CODE GOLF

Shorter for loop syntax

```
# Tiny C Style.  
for(;;i++<10;){ echo "$i";}  
  
# Undocumented method.  
for i in {1..10};{ echo "$i";}  
  
# Expansion.  
for i in {1..10}; do echo "$i"; done  
  
# C Style.  
for((i=0;i<=10;i++)); do echo "$i"; done
```

Shorter infinite loops

```
# Normal method  
while :; do echo hi; done  
  
# Shorter  
for(;;){ echo hi;}
```

Shorter function declaration

```
# Normal method  
f(){ echo hi;}  
  
# Using a subshell  
f()(echo hi)  
  
# Using arithmetic  
# This can be used to assign integer values.  
# Example: f a=1  
#           f a++  
f()(($1))  
  
# Using tests, loops etc.  
# NOTE: 'while', 'until', 'case', '(()', '[[[]]' can also be used.  
f()if true; then echo "$1"; fi  
f()for i in "$@"; do echo "$i"; done
```

Shorter if syntax

```
# One line  
# Note: The 3rd statement may run when the 1st is true  
[[ $var == hello ]] && echo hi || echo bye  
[[ $var == hello ]] && { echo hi; echo there; } || echo bye  
  
# Multi line (no else, single statement)  
# Note: The exit status may not be the same as with an if statement  
[[ $var == hello ]] &&  
    echo hi  
  
# Multi line (no else)
```

```
[[ $var == hello ]] && {
    echo hi
    # ...
}
```

Simpler case statement to set variable

The `:` built-in can be used to avoid repeating `variable=` in a case statement. The `$_` variable stores the last argument of the last command. `:` always succeeds so it can be used to store the variable value.

```
# Modified snippet from Neofetch.
case "$OSTYPE" in
    "darwin"*)
        : "MacOS"
        ;;

    "linux"*)
        : "Linux"
        ;;

    *"bsd"* | "dragonfly" | "bitrig")
        : "BSD"
        ;;

    "cygwin" | "msys" | "win32")
        : "Windows"
        ;;

    *)
        printf '%s\n' "Unknown OS detected, aborting..." >&2
        exit 1
        ;;
esac

# Finally, set the variable.
os="$_"
```

OTHER

Use read as an alternative to the sleep command

Surprisingly, `sleep` is an external command and not a bash built-in.

CAVEAT: Requires bash 4+

Example Function:

```
read_sleep() {
    # Usage: read_sleep 1
    #         read_sleep 0.2
    read -rt "$1" <> <(:) || :
}
```

Example Usage:

```
read_sleep 1
read_sleep 0.1
read_sleep 30
```

For performance-critical situations, where it is not economic to open and close an excessive number of file descriptors, the allocation of a file descriptor may be done only once for all invocations of `read`:

(See the generic original implementation at <https://blog.dhampir.no/content/sleeping-without-a-subprocess-in-bash-and-how-to-sleep-forever>)

```
exec {sleep_fd}<> <(:)
```

```
while some_quick_test; do
    # equivalent of sleep 0.001
    read -t 0.001 -u $sleep_fd
done
```

Check if a program is in the user's PATH

```
# There are 3 ways to do this and either one can be used.
type -p executable_name &>/dev/null
hash executable_name &>/dev/null
command -v executable_name &>/dev/null

# As a test.
if type -p executable_name &>/dev/null; then
    # Program is in PATH.
fi

# Inverse.
if ! type -p executable_name &>/dev/null; then
    # Program is not in PATH.
fi

# Example (Exit early if program is not installed).
if ! type -p convert &>/dev/null; then
    printf '%s\n' "error: convert is not installed, exiting..."
    exit 1
fi
```

Get the current date using strftime

Bash's `printf` has a built-in method of getting the date which can be used in place of the `date` command.

CAVEAT: Requires bash 4+

Example Function:

```
date() {
    # Usage: date "format"
    # See: 'man strftime' for format.
    printf "%($1)T\n" "-1"
}
```

Example Usage:

```
# Using above function.
$ date "%a %d %b - %l:%M %p"
Fri 15 Jun - 10:00 AM

# Using printf directly.
$ printf '%(%a %d %b - %l:%M %p)T\n' "-1"
Fri 15 Jun - 10:00 AM

# Assigning a variable using printf.
$ printf -v date '%(%a %d %b - %l:%M %p)T\n' '-1'
$ printf '%s\n' "$date"
Fri 15 Jun - 10:00 AM
```

Get the username of the current user

CAVEAT: Requires bash 4.4+

```
$ : \u
# Expand the parameter as if it were a prompt string.
$ printf '%s\n' "${_P}"
black
```

Generate a UUID V4

CAVEAT: The generated value is not cryptographically secure.

Example Function:

```
uuid() {
  # Usage: uuid
  C="89ab"

  for ((N=0;N<16;++N)); do
    B="$((RANDOM%256))"

    case "$N" in
      6) printf '4%x' "$((B%16))" ;;
      8) printf 'c%x' "${C:$RANDOM%${#C}:1}" "$((B%16))" ;;

      3|5|7|9)
        printf '%02x-' "$B"
        ;;

      *)
        printf '%02x' "$B"
        ;;
    esac
  done

  printf '\n'
}
```

Example Usage:

```
$ uuid
d5b6c731-1310-4c24-9fe3-55d556d44374
```

Progress bars

This is a simple way of drawing progress bars without needing a for loop in the function itself.

Example Function:

```
bar() {
  # Usage: bar 1 10
  #           ^----- Elapsed Percentage (0-100).
  #           ^-- Total length in chars.
  ((elapsed=$1*$2/100))

  # Create the bar with spaces.
  printf -v prog "%${elapsed}s"
  printf -v total "%${($2-elapsed)}s"

  printf '%s\r' "[${prog// /-}${total}]"
}
```

Example Usage:

```
for ((i=0;i<=100;i++)); do
  # Pure bash micro sleeps (for the example).
  ( ;; ) && ( ;; ) && ( ;; ) && ( ;; ) && ( ;; )

  # Print the bar.
  bar "$i" "10"
done

printf '\n'
```

Get the list of functions in a script

```
get_functions() {
    # Usage: get_functions
    IFS=$'\n' read -d "" -ra functions <<(declare -F)
    printf '%s\n' "${functions[@]//declare -f}"
}
```

Bypass shell aliases

```
# alias
ls

# command
# shellcheck disable=SC1001
\ls
```

Bypass shell functions

```
# function
ls

# command
command ls
```

Run a command in the background

This will run the given command and keep it running, even after the terminal or SSH connection is terminated. All output is ignored.

```
bkr() {
    (nohup "$@" &>/dev/null &)
}
```

bkr ./some_script.sh # some_script.sh is now running in the background

Capture the return value of a function without command substitution

CAVEAT: Requires bash 4+

This uses local namerefs to avoid using `var=$(some_func)` style command substitution for function output capture.

```
to_upper() {
    local -n ptr=${1}

    ptr=${ptr^^}
}

foo="bar"
to_upper foo
printf "%s\n" "${foo}" # BAR
```

AFTERWORD

Thanks for reading! If this bible helped you in any way and you'd like to give back, consider donating. Donations give me the time to make this the best resource possible. Can't donate? That's OK, star the repo and share it with your friends!



Rock on.