Pure Bash Bible

Open source project book.

Fork by: jlopezmx | jazlopez at github.com | jaziel-lopez at github.com

2022

Table of Contents

STRINGS

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
```

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.

Example Function:

```
trim_string() {
    # Usage: trim_string "   example   string    "

    # Remove all leading white-space.
    # '${1%%[![:space:]]*}': Strip everything but leading white-space.
    # '${1#${XXX}}': Remove the white-space from the start of the string.
    trim=${1#${1%%[![:space:]]*}}

    # Remove all trailing white-space.
    # '${trim##*[![:space:]]}': Strip everything but trailing white-space.
    # '${trim%${XXX}}': Remove the white-space from the end of the string.
    trim=${trim%${trim##*[![:space:]]}}
```

```
    printf '%s\n' "$trim"
}
```

Example Usage:

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

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

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    "

    # Disable globbing to make the word-splitting below safe.
    set -f

    # Set the argument list to the word-splitted string.
    # This removes all leading/trailing white-space and reduces
    # all instances of multiple spaces to a single ("  " -> " ").
    set -- $*

    # Print the argument list as a string.
    printf '%s\n' "$*"

    # Re-enable globbing.
    set +f
}
```

Example Usage:

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

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

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

Check if string contains a sub-string

Using a case statement:

```
case $var in
    *sub_string1*)
        # Do stuff
    ;;

    *sub_string2*)
        # Do other stuff
    ;;

    *)
        # Else
    ;;
esac
```

Check if string starts with sub-string

Using a case statement:

```
case $var in
    sub_string1*)
        # Do stuff
    ;;

    sub_string2*)
        # Do other stuff
    ;;

    *)
        # Else
    ;;
esac
```

Check if string ends with sub-string

Using a case statement:

```
case $var in
    *sub_string1)
```

```
        # Do stuff
    ;;

    *sub_string2)
        # Do other stuff
    ;;

    *)
        # Else
    ;;
esac
```

Split a string on a delimiter

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

Example Function:

```
split() {
    # Disable globbing.
    # This ensures that the word-splitting is safe.
    set -f

    # Store the current value of 'IFS' so we
    # can restore it later.
    old_ifs=$IFS

    # Change the field separator to what we're
    # splitting on.
    IFS=$2

    # Create an argument list splitting at each
    # occurance of '$2'.
    #
    # This is safe to disable as it just warns against
    # word-splitting which is the behavior we expect.
    # shellcheck disable=2086
    set -- $1

    # Print each list value on its own line.
    printf '%s\n' "$@"

    # Restore the value of 'IFS'.
    IFS=$old_ifs

    # Re-enable globbing.
    set +f
}
```

Example Usage:

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

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

Trim quotes from a string

Example Function:

```
trim_quotes() {
    # Usage: trim_quotes "string"

    # Disable globbing.
    # This makes the word-splitting below safe.
    set -f

    # Store the current value of 'IFS' so we
    # can restore it later.
    old_ifs=$IFS

    # Set 'IFS' to ["'].
    IFS=\"\'

    # Create an argument list, splitting the
    # string at ["'].
    #
    # Disable this shellcheck error as it only
    # warns about word-splitting which we expect.
    # shellcheck disable=2086
    set -- $1

    # Set 'IFS' to blank to remove spaces left
    # by the removal of ["'].
    IFS=

    # Print the quote-less string.
    printf '%s\n' "$*"

    # Restore the value of 'IFS'.
```

```
    IFS=$old_ifs

    # Re-enable globbing.
    set +f
}
```

Example Usage:

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

FILES

Parsing a `key=val` file.

This could be used to parse a simple `key=value` configuration file.

```
# Setting 'IFS' tells 'read' where to split the string.
while IFS='=' read -r key val; do
    # Skip over lines containing comments.
    # (Lines starting with '#').
    [ "${key##\#*}" ] || continue

    # '$key' stores the key.
    # '$val' stores the value.
    printf '%s: %s\n' "$key" "$val"

    # Alternatively replacing 'printf' with the following
    # populates variables called '$key' with the value of '$val'.
    #
    # NOTE: I would extend this with a check to ensure 'key' is
    #       a valid variable name.
    # export "$key=$val"
    #
    # Example with error handling:
    # export "$key=$val" 2>/dev/null ||
    #     printf 'warning %s is not a valid variable name\n' "$key"
done < "file"
```

Get the first N lines of a file

Alternative to the `head` command.

Example Function:

```
head() {
    # Usage: head "n" "file"
    while IFS= read -r line; do
        printf '%s\n' "$line"
        i=$((i+1))
        [ "$i" = "$1" ] && return
    done < "$2"

    # 'read' used in a loop will skip over
    # the last line of a file if it does not contain
    # a newline and instead contains EOF.
    #
    # The final line iteration is skipped as 'read'
    # exits with '1' when it hits EOF. 'read' however,
    # still populates the variable.
    #
    # This ensures that the final line is always printed
    # if applicable.
    [ -n "$line" ] && printf %s "$line"
}
```

Example Usage:

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

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

Get the number of lines in a file

Alternative to `wc -l`.

Example Function:

```
lines() {
    # Usage: lines "file"

    # '|| [ -n "$line" ]': This ensures that lines
    # ending with EOL instead of a newline are still
    # operated on in the loop.
    #
    # 'read' exits with '1' when it sees EOL and
    # without the added test, the line isn't sent
    # to the loop.
    while IFS= read -r line || [ -n "$line" ]; do
        lines=$((lines+1))
```

```
    done < "$1"

    printf '%s\n' "$lines"
}
```

Example Usage:

```
$ lines ~/.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/*/
    [ -e "$1" ] \
        && printf '%s\n' "$#" \
        || printf '%s\n' 0
}
```

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`.

```
:>file

# OR (shellcheck warns for this)
>file
```

FILE PATHS

Get the directory name of a file path

Alternative to the `dirname` command.

Example Function:

```
dirname() {
    # Usage: dirname "path"

    # If '$1' is empty set 'dir' to '.', else '$1'.
    dir=${1:-.}

    # Strip all trailing forward-slashes '/' from
    # the end of the string.
    #
    # "${dir##*[!/]}": Remove all non-forward-slashes
    # from the start of the string, leaving us with only
    # the trailing slashes.
    # "${dir%%"${}"}": Remove the result of the above
    # substitution (a string of forward slashes) from the
    # end of the original string.
    dir=${dir%%"${dir##*[!/]}"}

    # If the variable *does not* contain any forward slashes
    # set its value to '.'.
    [ "${dir##*/*}" ] && dir=.

    # Remove everything *after* the last forward-slash '/'.
    dir=${dir%/*}

    # Again, strip all trailing forward-slashes '/' from
    # the end of the string (see above).
    dir=${dir%%"${dir##*[!/]}"}

    # Print the resulting string and if it is empty,
    # print '/'.
    printf '%s\n' "${dir:-/}"
}
```

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"]

    # Strip all trailing forward-slashes '/' from
    # the end of the string.
    #
    # "${1##*[!/]}": Remove all non-forward-slashes
    # from the start of the string, leaving us with only
    # the trailing slashes.
    # "${1%%"${}"}:  Remove the result of the above
    # substitution (a string of forward slashes) from the
    # end of the original string.
    dir=${1%${1##*[!/]}}

    # Remove everything before the final forward-slash '/'.
    dir=${dir##*/}

    # If a suffix was passed to the function, remove it from
    # the end of the resulting string.
    dir=${dir%"$2"}

    # Print the resulting string and if it is empty,
    # print '/'.
    printf '%s\n' "${dir:-/}"
}
```

Example Usage:

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

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

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

LOOPS

Loop over a (*small*) range of numbers

Alternative to `seq` and only suitable for small and static number ranges. The number list can also be replaced with a list of words, variables etc.

```
# Loop from 0-10.
for i in 0 1 2 3 4 5 6 7 8 9 10; do
    printf '%s\n' "$i"
done
```

Loop over a variable range of numbers

Alternative to `seq`.

```
# Loop from var-var.
start=0
end=50

while [ "$start" -le "$end" ]; do
    printf '%s\n' "$start"
    start=$((start+1))
done
```

Loop over the contents of a file

```
while IFS= read -r line || [ -n "$line" ]; do
    printf '%s\n' "$line"
done < "file"
```

Loop over files and directories

Don't use `ls`.

CAVEAT: When the glob does not match anything (empty directory or no matching files) the variable will contain the unexpanded glob. To avoid working on unexpanded globs check the existence of the file contained in the variable using the appropriate file conditional. Be aware that symbolic links are resolved.

```
# Greedy example.
for file in *; do
    [ -e "$file" ] || [ -L "$file" ] || continue
    printf '%s\n' "$file"
done

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

# Iterate over directories.
for dir in ~/Downloads/*/; do
    [ -d "$dir" ] || continue
    printf '%s\n' "$dir"
done
```

VARIABLES

Name and access a variable based on another variable

```
$ var="world"
$ eval "hello_$var=value"
$ eval printf '%s\n' "\$hello_$var"
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
`\033[38;5;<NUM>m`	Set text foreground color.	`0-255`

Sequence	What does it do?	Value
`\033[48;5;<NUM>m`	Set text background color.	`0-255`
`\033[38;2;<R>;<G>;<B>m`	Set text foreground color to RGB color.	`R`, `G`, `B`
`\033[48;2;<R>;<G>;<B>m`	Set text background color to RGB color.	`R`, `G`, `B`

Text Attributes

Sequence	What does it do?
`\033[m`	Reset text formatting and colors.
`\033[1m`	Bold text.
`\033[2m`	Faint text.
`\033[3m`	Italic text.
`\033[4m`	Underline text.
`\033[5m`	Slow blink.
`\033[7m`	Swap foreground and background colors.
`\033[8m`	Hidden text.
`\033[9m`	Strike-through text.

Cursor Movement

Sequence	What does it do?	Value
`\033[<LINE>;<COLUMN>H`	Move cursor to absolute position.	`line`, `column`
`\033[H`	Move cursor to home position (`0,0`).	
`\033[<NUM>A`	Move cursor up N lines.	`num`
`\033[<NUM>B`	Move cursor down N lines.	`num`
`\033[<NUM>C`	Move cursor right N columns.	`num`
`\033[<NUM>D`	Move cursor left N columns.	`num`
`\033[s`	Save cursor position.	
`\033[u`	Restore cursor position.	

Erasing Text

Sequence	What does it do?
`\033[K`	Erase from cursor position to end of line.
`\033[1K`	Erase from cursor position to start of line.
`\033[2K`	Erase the entire current line.
`\033[J`	Erase from the current line to the bottom of the screen.
`\033[1J`	Erase from the current line to the top of the screen.
`\033[2J`	Clear the screen.
`\033[2J\033[H`	Clear the screen and move cursor to `0,0`.

PARAMETER EXPANSION

Prefix and Suffix Deletion

Parameter	What does it do?
`${VAR#PATTERN}`	Remove shortest match of pattern from start of string.
`${VAR##PATTERN}`	Remove longest match of pattern from start of string.
`${VAR%PATTERN}`	Remove shortest match of pattern from end of string.
`${VAR%%PATTERN}`	Remove longest match of pattern from end of string.

Length

Parameter	What does it do?
`${#VAR}`	Length of var in characters.

Default Value

Parameter	What does it do?
`${VAR:-STRING}`	If `VAR` is empty or unset, use `STRING` as its value.
`${VAR-STRING}`	If `VAR` is unset, use `STRING` as its value.

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

CONDITIONAL EXPRESSIONS

For use in `[ ]` `if [ ]; then` and `test`.

File Conditionals

Expression	Value	What does it do?
`-b`	`file`	If file exists and is a block special file.
`-c`	`file`	If file exists and is a character special file.
`-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.
`-p`	`file`	If file exists and is a named pipe (*FIFO*).
`-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.

Expression	Value	What does it do?
`-L`	`file`	If file exists and is a symbolic link.
`-S`	`file`	If file exists and is a socket.

Variable Conditionals

Expression	Value	What does it do?
`-z`	`var`	If the length of string is zero.
`-n`	`var`	If the length of string is non-zero.

Variable Comparisons

Expression	What does it do?
`var = var2`	Equal to.
`var != var2`	Not equal to.
`var -eq var2`	Equal to (*algebraically*).
`var -ne var2`	Not equal to (*algebraically*).
`var -gt var2`	Greater than (*algebraically*).
`var -ge var2`	Greater than or equal to (*algebraically*).
`var -lt var2`	Less than (*algebraically*).
`var -le var2`	Less than or equal to (*algebraically*).

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
`%`	Modulo
`+=`	Plus-Equal (*Increment a variable.*)
`-=`	Minus-Equal (*Decrement a variable.*)
`*=`	Times-Equal (*Multiply a variable.*)
`/=`	Slash-Equal (*Divide a variable.*)
`%=`	Mod-Equal (*Remainder of dividing a variable.*)

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

Ternary Tests

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

Check if a number is a float

Example Function:

```
is_float() {
    # Usage: is_float "number"

    # The test checks to see that the input contains
    # a '.'. This filters out whole numbers.
    [ -z "${1##*.*}" ] &&
        printf %f "$1" >/dev/null 2>&1
}
```

Example Usage:

```
$ is_float 1 && echo true
$
```

```
$ is_float 1.1 && echo true
$ true
```

Check if a number is an integer

Example Function:

```
is_int() {
    # usage: is_int "number"
    printf %d "$1" >/dev/null 2>&1
}
```

Example Usage:

```
$ is_int 1 && echo true
$ true

$ is_int 1.1 && echo true
$
```

TRAPS

Traps allow a script to execute code on various signals. In pxltrm (*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.

Do something on script exit

```
# Clear screen on script exit.
trap 'printf \\033[2J\\033[H\\033[m' EXIT

# Run a function on script exit.
# 'clean_up' is the name of a function.
trap clean_up EXIT
```

Ignore terminal interrupt (CTRL+C, SIGINT)

```
trap '' INT
```

OBSOLETE SYNTAX

Command Substitution

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

```
# Right.
var="$(command)"

# Wrong.
var=`command`

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

INTERNAL AND ENVIRONMENT VARIABLES

Open the user's preferred text editor

```
"$EDITOR" "$file"

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

Get the current working directory

This is an alternative to the `pwd` built-in.

```
"$PWD"
```

Get the PID of the current shell

```
"$$"
```

Get the current shell options

```
"$-"
```

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. 🤘