Bash Programming Cheat Sheet

🄯 Erik E. Lorenz, May 26, 2014

Cheat Sheet Color Coding

🟩 `cmd`	Most frequent commands
▨ `cmd`	Usually not harmful
🟥 `cmd`	deletes data, requires root or is just bad programming

Internal Files and Directories

🟩 `~/.bashrc`	user-specific global functions and aliases
🟥 `~/.bash_profile`	similar to `~/.bashrc`
▨ `~/.bash_history`	list of previous bash commands
🟥 `~/.bash_logout`	runs on bash logout
▨ `/bin/bash`	location of the `bash` executable

Terms

term	description	examples
user	a user account of the system	root e.lorenz
file	regular file	~/file.txt code/asd/src/main.cpp
dir	regular directory	~/directory /etc

cmd	any command	`echo`
		`date +%F`

host	name or ip of a remote machine	enssim.etit.tu-chemnitz.de 134.109.52.89
port	a network port for communication with a program	22 31159
url	uniform resource locator	http://*host*:port/*dir*/*file*
pid	process id	18738
alias	command alias	`alias ssk='ssh enssim'`
export	define an environment variable	`export PATH=~/bin:$PATH`

source	run a script that sets environment variables/aliases	`. ~/.bashrc`
		`source ~/.bashrc`

Useful Environment Variables

`$HOME`	home directory. Usually `/home/`*`user`*
`~`	same as `$HOME`
`$USER`	name of the current user
`$UID, $EUID`	user id, effective user id
`$PATH`	colon-separated list of search directories for binaries
`$LIBRARY_PATH`	search paths for .so and .a files at compile time
`$LD_LIBRARY_PATH`	search paths for .so and .a files at run time
`$PWD`	current working directory
`$EDITOR`	preferred command line text editor, e.g. `vim`
`$IFS`	internal field separator, e.g. for `for...in` constructs
`$LINENO`	current line number in a script, e.g. for debugging
`$COLUMNS`	width of the terminal
`$LINES`	height of the terminal
`$LANG`	preferred language of the user
`$SHELL`	path of the shell-executable. Should be `/bin/bash`
`$SHLVL`	shell nesting level on the current machine
`$$`	*pid* of the current script or bash instance
`$PPID`	*pid* of the parent process
`$!`	*pid* of the last child process (see `Forking`)
`$0`	command used to run this script or bash instance
`$@`	array of arguments of a script or function
`$1, $2, ... $9`	first, second, ... ninth argument
`!!`	previous command
`!$`	last argument of the previous command
`!^`	first arguments of the previous command
`!:1, !:2, ...`	arguments of the last command
`!:1-`	all arguments of the last command

Debugging

▨ `set -x`	print every command before execution
▨ `trap read debug`	confirm every command with [Enter]

Hotkeys

🟩 Tab	autocomplete the current command or path
▨ Ctrl+I	same as Tab
▨ Alt+*	insert all possible completions
🟩 Ctrl+C	kill the current command
🟩 Ctrl+D	exit the current shell (write end-of-file character)
▨ Ctrl+X Ctrl+E	write the next command in your $EDITOR
🟩 Ctrl+R	reverse-search your history for a command
▨ Ctrl+Z	suspend the process. Resume with `%`
🟥 Ctrl+S	suspend the current terminal
🟩 Ctrl+Q	resume a suspended terminal
▨ Ctrl+L	clear the terminal. Similar to `clear`
▨ Ctrl+U	clear the line before the cursor
▨ Ctrl+K	clear the line after the cursor
▨ Alt+F	move forward one word
▨ Alt+B	move backward one word
▨ Alt+D	delete next word
▨ Alt+Backspace	delete previous word

Redirecting Standard I/O

🟩 *`cmd`* `>` *`file`*	write output to a new *file* (overwrites)
🟩 *`cmd`* `>>` *`file`*	append output to *file*
▨ *`cmd`* `| tee` *`file`*	both print and write to a file (add `-a` to append)
▨ *`cmd`* `2>` *`file`*	write errors to *file*
▨ *`cmd`* `2>&1`	redirect errors to standard output
🟩 *`cmd`* `&>/dev/null`	discard all output
▨ *`cmd`* `<` *`file`*	read input from *file*
▨ *`cmd`* `<< EOF`	read input from command line until the line "`EOF`"
▨ *`cmd`* `<<<` *`cmd`*	read input from the rest of the line
▨ *`cmd`* `|` *`cmd`*	pipe output from the first *cmd* to the second

Process Control (Forking and Killing)

🟩 *`cmd`* `&`	Send *cmd* to background, return to command line
▨ `wait`	wait for forked processes to finish
▨ `(` *`cmd`* `& );exit`	fork a command within a one-liner (example)x
▨ `jobs`	show all child processes
▨ `-r`	show only running processes
▨ `-p`	show *pids* only
🟥 `killall` *`cmd`*	stop all processes with the name *cmd*
🟥 `kill` *`pid`*	ask a process to stop
🟥 ``kill `jobs -rp` ``	kill all child processes
🟥 `kill -KILL` *`pid`*	forcefully stop a process

Automatic String Expansion (Examples)

🟩 `echo *.txt`	asd.txt dsa.txt longfilename.txt s.txt
🟩 `echo ?s?.t?t`	asd.txt dsa.txt bse.tot
▨ `echo {7..11}`	7 8 9 10 11
▨ `echo {07..11}`	07 08 09 10 11
▨ `echo {a..g}`	a b c d e f g
▨ `echo sim{08..10}`	sim08 sim09 sim10
▨ `echo foo.{txt,pdf,png}`	foo.txt foo.pdf foo.png

Flow Control

🟩 `if` *`expression`*`; then`
 `do something`
`else`
 `do something else`
`fi`
	Expressions can be commands and functions (return 0 → true) or built-in conditionals

▨ *`expression`* `&&` *`cmd`*	run *cmd* if *expression* is true
▨ *`expression`* `||` *`cmd`*	run *cmd* if *expression* is false

Aborting and Exiting

▨ `continue`	next loop iteration	▨ `break`	exit loop
▨ `return`	exit function	▨ `exit`	exit script / terminal

Unary Conditionals

🟩 `[ -z` *`str`* `]`	*str* is empty	🟩 `[ -n` *`str`* `]`	*str* is not empty
▨ `[ -e` *`file`* `]`	*file* exists	▨ `[ -s` *`file`* `]`	*file* is not empty
▨ `[ -f` *`file`* `]`	*file* is a regular file	▨ `[ -d` *`dir`* `]`	dir is a directory
▨ `[ -L` *`file`* `]`	*file* is a symlink	▨ `[ -x` *`file`* `]`	*file* is executable
▨ `[ -r` *`file`* `]`	*file* is readable	▨ `[ -w` *`file`* `]`	*file* is writable
▨ `[ -v` *`str`* `]`	*str* is a variable	▨ `[ -O` *`file`* `]`	$USER owns *file*

Binary Conditionals

🟩 `[` *`arg1`* `<` *`arg2`* `]`	strings	▨ `[[` *`arg1`* `<` *`arg2`* `]]`	raw strings (no string expansion)
🟩 `[` *`arg1`* `>` *`arg2`* `]`		▨ `[[` *`arg1`* `>` *`arg2`* `]]`	
🟩 `[` *`arg1`* `==` *`arg2`* `]`		▨ `[[` *`arg1`* `==` *`arg2`* `]]`	
🟩 `[` *`arg1`* `!=` *`arg2`* `]`		▨ `[[` *`arg1`* `!=` *`arg2`* `]]`	
▨ `[` *`arg1`* `-lt` *`arg2`* `]`	integers	▨ `((` *`arg1`* `<` *`arg2`* `))`	integers
▨ `[` *`arg1`* `-gt` *`arg2`* `]`		▨ `((` *`arg1`* `>` *`arg2`* `))`	
▨ `[` *`arg1`* `-eq` *`arg2`* `]`		▨ `((` *`arg1`* `==` *`arg2`* `))`	
▨ `[` *`arg1`* `-ne` *`arg2`* `]`		▨ `((` *`arg1`* `!=` *`arg2`* `))`	

Loops

🟩 `for word in $words; do`
 `echo $word`
`done`
	print every word in $words

▨ `while` *`expression`*`; do`
 `do something`
`done`
	traditional while loop

different methods of iterating over an integer range:

🟩
```
for i in {0..9}; do echo $i; done
```
▨
```
for i in `seq $start $num $step`; do echo $i; done
```
🟥
```
i=0; while (( num < 10 )); do echo $i; let i++; done
```

Iterate over every line in $var:

▨
```
IFS=$'\r\n' ; for line in $var; do echo $line; done
```

Parallel Workers

▨
```
num=100
ncpus=`grep proc /proc/cpuinfo|wc -l`
for i in `seq 1 $num`; do
 cmd &
 (( $ncpus > `jobs -r|wc -l` )) || wait -n
done
wait
```

Run *cmd* 100 times in a parallel queue, one process per cpu core.

I/O Processing

🟩 *`cmd`* `$@`	process all arguments at once

▨ `while true; do`
 `cmd` `"$1"`
 `shift || break`
`done`
	process arguments separately. To be used in a script or function.

🟩 *`cmd`* `| xargs`	merge output to a single line
🟩 `echo "foo bar" | xargs` *`cmd`*	set arguments of *cmd* to `foo bar`
🟩 `echo "foo bar" | xargs -n1`	split to one word per line
🟩 `echo "foo bar" | xargs -n1` *`cmd`*	run *cmd* on every single word
▨ `read myvar`	read a line from stdin into $myvar

Bash Invocation

▨ `bash -c "`*`cmd`*`"`	run *cmd* in a fresh bash instance
▨ `su` *`user`* `-c "`*`cmd`*`"`	run *cmd* as another user
🟥 `sudo "`*`cmd`*`"`	run *cmd* as root
▨ `ssh` *`user`*`@`*`host cmd`*	run *cmd* as *user* on *host*