[news.ycombinator.com](https://news.ycombinator.com/item?id=31250275)

**Bash-oneliner: A collection of handy Bash one-liners and terminal tricks**

26-32 minutes

|  |  |  |
| --- | --- | --- |
|  |  | I was surprised to see `$()` missing from this (otherwise quite extensive) list. There are a few commands listed which employ it, but it absolutely deserves its own entry.  That and `readlink -f` to get the absolute path of a file. (Doesn't work on MacOS; the only substitute I've found is to install `greadlink`.)  And `cp -a`, which is like `cp -r`, but it leaves permissions intact - meaning that you can prepend `sudo` without the hassle of changing the ownership back.  I never see `lndir` on these lists either. It makes a copy of a directory, but all of the non-directory files in the target are replaced with symlinks back to the source while directories are preserved as-is. Meaning that when you `cd` into it, you are actually landing in a copied structure of the source directory instead of the source directory itself, as would be the case if you just symlinked the source folder.  Once inside, any file you want to modify without affecting the original just needs you to create the symlink into a file, which you can do with `sed -i '' $symlink`. There you have it: effectively a copy of your original directory, with only the modified files actually taking up space (loosely speaking).  Looks like I have a few pull requests to submit. |

|  |  |  |
| --- | --- | --- |
|  |  | Whoa, realpath includes some flags for relative paths as well! That's bad ass.  I was surprised to see it was already installed with coreutils, so I've apparently had it for a while now. Thanks for the heads up. |

|  |  |  |
| --- | --- | --- |
|  |  | "And `cp -a`, which is like `cp -r` which leaves permissions intact - meaning you can prepend `sudo` without the hassle of changing ownership back."  I always use pax instead of cp  mkdir newdir  cd olddir  pax -rw -pp . ../newdir  preserves permissions and access times  cd olddir  pax -rw -pe . ../newdir  preserves ownership, permissions and access times. |

|  |  |  |
| --- | --- | --- |
|  |  | > `readlink -f`  Other languages have this also. In fact it's a C language system call in Linux, though it only goes one indirection level at a time.  This is useful if you run a script and want to use files contained in the directory where the script is located. This comes up if you have a symlink to the script in any /something/something-else/bin directory.  Use cases:  Script needs its own location for e.g. configuration files.  User of the script needs the location for e.g. another script that needs to be run that isn't in the search path. |

|  |  |  |
| --- | --- | --- |
|  |  | FWIW `realpath` on macOS should be functionally equivalent to `readlink -f` - particularly if you ignore all the other functionality `readlink` provides. |

|  |  |  |
| --- | --- | --- |
|  |  | Nice, thank you! I did not know about `realpath`. Piping absolute paths and copying them to the clipboard is a pattern I use a lot, so this tip will get substantial mileage out of me. |

|  |  |  |
| --- | --- | --- |
|  |  | For canonicalizing directories, you can use this portably:  dir=$(cd -P -- "$dir" && pwd)  with the usual gotcha that $() strips trailing newlines. |

|  |  |  |
| --- | --- | --- |
|  |  | *>a copy of your original directory, with only the modified files actually taking up space*  What's the use case for this sort of thing? |

|  |  |  |
| --- | --- | --- |
|  |  | When I need to shuffle around and/or rename my media files, for instance, it's risky to operate on the originals themselves. I've screwed up hundreds of mp3 files by issuing a bad `rename` command. I've lost the hierarchical structures of genres and artists and albums and such by accidentally moving them into the same directory together. And so on.  If you `lndir` your mp3 directory to a functional copy of it, a playground of sorts, you can move things around and rename them without having to worry about scenarios like having to listen to a bunch of mp3s in order to put them back to where they're supposed to be.  When you're satisfied with the re-organization of your files, you can replace your symlinks with the original files. Since none of the directories are symlinked, you never have to worry about `cd`ing into a place you didn't intend to. |

|  |  |  |
| --- | --- | --- |
|  |  | You could use hardlinks for this, though (as long as both places are in the same filesystem). Then when you're finishes you just delete the old folder. |

|  |  |  |
| --- | --- | --- |
|  |  | Huh. As a hardlink avoider, I never even thought of this. (I don't have a good reason for avoiding hardlinks - it's mainly just because I didn't grok them enough to predict their behavior. When you teach yourself, you gotta expect some gaps!) Thanks for the suggestion. |

|  |  |  |
| --- | --- | --- |
|  |  | At work, we have a codebase that is used by numerous projects. The solution for project specific changes is 'checkout the repo, and make your changes locally'.  What that has turned into is 'checkout the repo and make a new directory structure that symlinks everything back to the original'. The keeps the maintenance burden relatively small as you can easily update the base version, and only need to worry about the files you changed.  I certainly wouldn't recommend using this approach for anything; but it is not as terrible as it sounds. |

|  |  |  |
| --- | --- | --- |
|  |  | The bash customizations for fzf include a Ctrl-t interactive path completion. Works wonders on massive codebases. |

|  |  |  |
| --- | --- | --- |
|  |  | sed -i  Watch out, that's a Linux-ism and macOS's sed will cheerfully use the thing after it as the backup expression; as far as I know, the absolute safest choice is to always specify a backup extension "sed -i~" or "sed -i.bak" to make it portable, although there are plenty of work-arounds trying to detect which is which and "${SED\_I} -e whatever" type silliness  My contribution (and yeah, I know, PR it ...) is that I get a lot of mileage out of setting the terminal title from my scripts:  title() { printf '\033]0;%s\007' "$\*"; }  and there's one for tmux, too  printf '\033]2;%s\007' "$\*";  with the two infinitely handy resources:  \* <https://www.xfree86.org/current/ctlseqs.html>  \* <https://iterm2.com/documentation-escape-codes.html> |

|  |  |  |
| --- | --- | --- |
|  |  | Even better, use a real text editor like ed or ex. (Nowadays ex is more portable because many distros — against POSIX — omit all 55 kilobytes of GNU ed. Of course, smaller systems might not have ex/vi.)  Basic usage looks like this:  printf '%s\n' '" some commands...' 'wq' | ex -s file  Or:  ex -s file <<'EOF'  " some commands...  wq  EOF  By the way, these commands are the ones that you use in your vimrc or after a colon in vim — at least, the POSIX subset of that — so any ex commands you learn translate naturally to your normal editor. |

|  |  |  |
| --- | --- | --- |
|  |  | That's an interesting trick, I'll bear it in mind.  That said, the "lottery factor" is often a bigger contributor to the things that land in codebases than "optimality". Plus, I've actually seen somewhere that perl is the most common binary across every system, and it's likely a larger population who know perl than ed would be my guess |

|  |  |  |
| --- | --- | --- |
|  |  | How many major versions of Perl do you know about?  What is the distribution of these different versions of Perl across various OSes and OS versions?  Hint: Lots of backward-incompatible changes tend to get made around different major versions. Having Perl 4 is not like having Perl 5 which is not like Perl 6. |

|  |  |  |
| --- | --- | --- |
|  |  | I still don't know what your point is.  Are there any contemporary distros that have the "perl" exectuable that is not Perl 5? |

|  |  |  |
| --- | --- | --- |
|  |  | If you want to claim broad compatibility, you can't just look at the latest distributions. You have to look at OSes other than Linux. You have to look at older versions of OSes, too. And don't forget the billions of embedded and handheld devices, too. |

|  |  |  |
| --- | --- | --- |
|  |  | > You have to look at older versions of OSes, too.  Perl 5 was released in 1994. It is *incredibly* portable. If you have a system that legitimately has Perl 4, I would love to hear about it, but … |

|  |  |  |
| --- | --- | --- |
|  |  | Perl more common than vi? The study you’re remembering probably didn’t include POSIX utilities. |

|  |  |  |
| --- | --- | --- |
|  |  | With the full understanding that "docker images" are their own special little things:  $ docker run --rm ubuntu:22.04 bash -c 'command -v ex; command -v ed; command -v vi; command -v perl;'  /usr/bin/perl  and the same result for "debian:stable"  ---  *edit:* I just realized that's because apt is \_written in\_ perl, but tomato, tomahto, and it may very well be that they picked perl for that same universal-binary reason |

|  |  |  |
| --- | --- | --- |
|  |  | > *Watch out, that's a Linux-ism and macOS's sed*  It's GNU sed vs (Free)BSD sed, which are different enhancements of the POSIX standards for sed that went in different design directions. One could Homebrew/macports install gnu-sed on macOS to get a GNU version to write Linux-portable scripts as-needed. |

|  |  |  |
| --- | --- | --- |
|  |  | Plan9 sed has no -i option. Older versions of NetBSD will not have it either.  I never understood the point of the -i option other than to conserve keystrokes. A temporary file is still created then removed; the -i option only saves the user from having to specify it. Maybe the intent is it is only for "one-off" use, not for use in scripts.  This will work for GNU, BSD and Plan9:  sed -n 's/old/new/wfile.tmp' file  mv file.tmp file  Or just use redirection.  Given the choice between avoiding some keypresses and more portable scripts, I will keep choosing the later.  NetBSD sed may have the -i option now but I do not see anyone using it in scripts meant to be portable, like build.sh^1  1. [https://ftp.netbsd.org/pub/NetBSD/NetBSD-release-9/src/build...](https://ftp.netbsd.org/pub/NetBSD/NetBSD-release-9/src/build.sh) |

|  |  |  |
| --- | --- | --- |
|  |  | > sed -n 's/old/new/wfile.tmp' file  This saves only lines which underwent substitution, which was probably not what you wanted. |

|  |  |  |
| --- | --- | --- |
|  |  | Missing semicolon.  sed -n 's/old/new/;w file.tmp' file  Unlike BSD and GNU sed, it appears that Plan9 sed will append to instead of overwite file.tmp  It also requires a space after the w command. |

|  |  |  |
| --- | --- | --- |
|  |  | -i creates a temporary file and then replaces the original with the temporary. The BSD man page advises not to use -i alone because if there is insufficient space to store the temporary, data may be lost. |

|  |  |  |
| --- | --- | --- |
|  |  | Using raw escape codes is ugly and device-dependent. People learned this in the 1970’s, and created libraries to get *away* from having to hard-code escape codes.  Here’s a device-independent variant:  title(){ tput tsl || tput -T xterm+sl tsl; printf %s "$\*"; tput fsl || tput -T xterm+sl fsl; }  Note: if the terminal does not advertise support of the necessary capabilities, it falls back to using the XTerm escape sequences. |

|  |  |  |
| --- | --- | --- |
|  |  | Having arrived back at my Mac, running iTerm2, I wanted to share another fun fact about using those executables: running that function while the outer shell is in "set -x" causes the title of the window to be  + printf %s 'hello world'^M^Jhello world+ tput fsl^M^J+tput -T xterm  and that's where it cuts off but I presume ends with "^M^J" at the end :-D |

|  |  |  |
| --- | --- | --- |
|  |  | As a point of comparison, I ssh-ed into my mac and ran "tput -T xterm+sl tsl" in order to see what it would output, and it hung my connection  So, I'll stick to my printf thanks |

|  |  |  |
| --- | --- | --- |
|  |  | You can’t run only one of the tput commands! You need to run both of them, as in the shell function; i.e. both tsl and fsl needs to be sent to the terminal!  If you want to see what *bytes* it would output, use “od”:  tput -T xterm+sl tsl | od -t c |

|  |  |  |
| --- | --- | --- |
|  |  | Yeah, I actually thought about that afterward, however, in that same "I wonder what happened", I also wondered if tput is bright enough to know the difference between the *local* termcap and the *connected* one  As a concrete example, my printf version works even when run inside docker, but  $ docker run --rm ubuntu:22.04 bash -c '{ tput tsl || tput -T xterm+sl tsl; } | od -c'  tput: No value for $TERM and no -T specified  tput: unknown terminal "xterm+sl"  0000000 |

|  |  |  |
| --- | --- | --- |
|  |  | Your example suffers from being a toy example; it makes no sense to run a noninteractive command in a docker container merely to output terminal escape codes. If this were the norm, "docker run” would probably by default make sure to copy the TERM setting to the inner command.  I would assume that if you run an *interactive* shell inside docker, TERM would actually be set correctly. It’s the same when you ssh somewhere else – the TERM environment variable is sent along, so that the remote program can see it and output the correct codes for your local terminal. Also, the docker image needs the terminfo database installed for “tput” to work. |

|  |  |  |
| --- | --- | --- |
|  |  | Thank you so much for the navigation shortcut link! I love these and picked them up from mentors at jobs but never found a definitive guide to all of the ones I could learn. |

|  |  |  |
| --- | --- | --- |
|  |  | Learnt a neat trick from an old sysadmin colleague.  If you’ve written a command but realize you don’t want to run it right now but want to save it in your history you can just put a `#` in front of it (ctrl-a #) making it a comment and allowing you to save it in your history without running it.  When you’re ready to run it you find it and remove the preceding `#` |

|  |  |  |
| --- | --- | --- |
|  |  | The opposite is adding space before the command. The command will run but it will not be saved in history.  EDIT: This apparently needs to be configured - setting HISTCONTROL=ignorespace |

|  |  |  |
| --- | --- | --- |
|  |  | I had been in the habit of symlinking ~/.bash\_history to /dev/null to avoid AFS/NFS writes on every local command execution. When I moved over to the financial industry, it didn't occur to me that such a symlink might look like an attempt to evade monitoring. A year or two in, I realized it didn't look good, but it had clearly been made my first week on the job, so I just left it in place for over 10 years rather than risk looking like I was again monkeying with my history.  I hope and presume they had much better monitoring than scanning bash history, but I'm not bet-my-career confident of that. |

|  |  |  |
| --- | --- | --- |
|  |  | It's simpler to use a tmpfs for this purpose. $XDG\_RUNTIME\_DIR is already available, on modern Linux versions. |

|  |  |  |
| --- | --- | --- |
|  |  | systemd-tmpfiles can be configured to delete a path upon ‘systemd-tmpfiles —-user clean’ |

|  |  |  |
| --- | --- | --- |
|  |  | Lol that makes sense, never thought of commenting out the command but I guess I do something similar. If i realize i dont want a command yet I enter it with a trailing `\`, then `CTRL+C` to get back to an empty prompt. |

|  |  |  |
| --- | --- | --- |
|  |  | This is useful when saving command lines to files (scripts) using the POSIX-required fc builtin. Command line histories are relatively cumbersome to save with Ash, Bash saves them but truncates them to 500 entries, whereas scripts can easily be saved indefinitely. Amongst Bash and other feature-heavy shell users, there are Rube Goldberg-like workarounds for command line history saving. OTOH, all shells aiming for POSIX compliance, including the fastest, lightest weight ones I prefer, will implement fc. It's already there; I make use of it.  I will type fc, save to a file (script) and then delete all lines before exiting the default EDITOR, e.g., %d in vi. This prevents the commands from being re-executed when I exit vi.  Also I sometimes use # combined with a semicolon to disable portions of command lines, e.g., early commands ;# late commands. I might cut and paste from one entry in the history into another one. Or I might fc -l 1 > file and edit the file down the the entries that form the starting point for a new script. By far, the shell is the most useful REPL for me.  There is no shortage of comments online praising the utility of the REPL concept but the only comment I have ever seen about fc was from a shell implementor/maintainer; it was negative. I use fc all the time. It has become essential for me to use the shell effectively as a REPL. |

|  |  |  |
| --- | --- | --- |
|  |  | > Bash saves them but truncates them to 500 entries  That behavior can be modified with the HISTSIZE environment variable.  > The maximum number of commands to remember on the history list. If the value is 0, commands are not saved in the history list. Numeric values less than zero result in every command being saved on the history list (there is no limit). The shell sets the default value to 500 after reading any startup files.  [https://www.gnu.org/software/bash/manual/html\_node/Bash-Vari...](https://www.gnu.org/software/bash/manual/html_node/Bash-Variables.html#index-HISTSIZE) |

|  |  |  |
| --- | --- | --- |
|  |  | Bash is too complicated. Too many options. I am not smart enough to use it. I prefer the Almquist shell. No shellshock.  echo "export PROMPT\_COMMAND='history -a;history -r;export HISTFILE=-1;export HISTFILESIZE=-1;shopt -s histappend'" > .bashrc  cp .bashrc /etc/profile |

|  |  |  |
| --- | --- | --- |
|  |  | You can achieve the same thing by typing a command, then hitting Escape followed by '#'.  It will prefix your current commandline with a '#' and "run" it. |

|  |  |  |
| --- | --- | --- |
|  |  | I wonder if it isn't possible to get it to save your command to history when you do Ctrl-C.  I tried a naive way by trapping sigint but couldn't get it to work. |

|  |  |  |
| --- | --- | --- |
|  |  | My favourite example in this page is used four times but not pointed out specifically. It's the use of <(some stuff) to create a temporary file descriptor. Since it's not explained I'll give it a quick go.  If you ever found yourself doing something like this:  sort file1 > file1\_sorted  sort file2 > file2\_sorted  diff file1\_sorted file2\_sorted  You can instead skip making two new files, and do:  diff <(sort file1) <(sort file2)  And voila - you've got two 'files' you are comparing, but without having to save them to disk. The examples on the page use this with `curl` and `head` to good effect, but it wouldn't necessarily be obvious what's going on. |

|  |  |  |
| --- | --- | --- |
|  |  | > *$SHELL current shell*  No, $SHELL is the user’s *default* shell; i.e. the shell started in a new terminal or when logging in on a console or remotely. If another shell program is started, $SHELL will still refer to the default shell, not the running shell program. |

|  |  |  |
| --- | --- | --- |
|  |  | This is a rather long list so i'll just mention my latest favorite which I recently learned on HN and it has made my life very easy with bash  Whenever you need to use a single-quote on command line add a $ sign before it. It makes escaping everything super easy  su user -c $'cd \'$dir\' && ...'  Before this it used to confuse the hell out of me.  More details are here: <https://stackoverflow.com/a/16605140/1031454> |

|  |  |  |
| --- | --- | --- |
|  |  | I know that may be convenient, but you'll want to exercise caution because $'' turns on inner escaping that wouldn't otherwise happen inside single-quoted strings  $ echo 'hello \"world'  hello \"world  $ echo $'hello \"world'  hello "world |

|  |  |  |
| --- | --- | --- |
|  |  | New users on my systems commonly ask me "what implements your pps process search?"  When the shell itself filters the output of ps, then removing a grep is unnecessary. Note this uses POSIX shell patterns, not regular expressions.  On a truly POSIX shell that does not support "local," remove the keyword, and change the braces to parentheses to force the function into a subshell.  pps () { local a= b= c= IFS='\0'; ps ax | while read a  do [ "$b" ] || c=1; for b; do case "$a" in \*"$b"\*) c=1;;  esac; done; [ "$c" ] && printf '%s\n' "$a" && c=; done; }  $ pps systemd  PID TTY STAT TIME COMMAND  1 ? Ss 5:11 /usr/lib/systemd/systemd --switched-root --system --deserialize 22  557 ? Ss 0:19 /usr/lib/systemd/systemd-journald  ... |

|  |  |  |
| --- | --- | --- |
|  |  | Thanks for the critique.  pps () { local a= b= c= IFS=$'\0'; ps ax | while read -r a  do [ "$b" ] || c=1; for b; do case "$a" in \*"$b"\*) c=1;;  esac; done; [ "$c" ] && printf '%s\n' "$a" && c=; done; } |

|  |  |  |
| --- | --- | --- |
|  |  | > Ctrl + x + Ctrl + e : launch editor defined by $EDITOR to input your command. Useful for multi-line commands. --  Great, I was just trying to remember that key combination the other day. Just got back to work after being for awhile for a child bonding leave. |

|  |  |  |
| --- | --- | --- |
|  |  | If your $VISUAL is overwrought with plugins (ahem) you can set $FCEDIT to something lighter. |

|  |  |  |
| --- | --- | --- |
|  |  | all these work for me  - C-x, C-e (separately)  - C-(x,e) (hold ctrl, x and e separately)  - C-(x+e) (hold ctrl-x and press e) |

|  |  |  |
| --- | --- | --- |
|  |  | A really nice list, thanks a lot for sharing! :)  I'd add this when a filesystem gets almost full (but not overfilled, see below). This shows where most of the space goes:  # du -axm / | sort -n | tail # takes a while on large filesystems, or ones with lots of files  Then narrow down for each of the most filled directories:  # du -axm /some/dir | sort -n | tail # subsequent searches are fast, now that metadata is cached.  In case there is no space at all, *sort* will complain if the /tmp directory is on the same fs, then the only option is to search any suspect directories with du -sm $dir  And about this one: <https://github.com/onceupon/Bash-Oneliner#using-ctrl-keys>  A bit surprised that the Ctrl+b(ack one...) and Ctrl+f(orward one char) shortcuts are not included.  As well as their Alt+b/f for a word back/forward too. Very convenient for going through a long command by getting in the beginning or the end of the line, then move words back/forth to update it. |

|  |  |  |
| --- | --- | --- |
|  |  | Ctrl + s : to stop output to terminal.  Ctrl + q : to resume output to terminal after Ctrl + s.  Who uses these? These are newb as well as advanced user killers that doesn't seem to serve a purpose.  It just makes you think your shell is locked up making you think either the server is out of memory, process is hung to the point no keys would react or the network is hosed. |

|  |  |  |
| --- | --- | --- |
|  |  | I /remember/ them from when I was logging in to "the computer lab" at my Uni from literally decades ago, but I haven't thought about them nor used them in at least that time. |

|  |  |  |
| --- | --- | --- |
|  |  | Something I’ve been wanting to do but haven’t found a perfect solution for yet: storing the output of previous command in some variable by default.  I use Terminal.app’s Select Between Marks or tmux, but I wish this was a thing. |

|  |  |  |
| --- | --- | --- |
|  |  | You can use:  VAR=$(!!)  to accomplish something similar. Not by default of course.  It re-runs the command, so if not idempotent/etc it will not return expected results. Also when re-run, the command will not be in a tty context, so if the executable is sensitive to such things (e.g. `ls`), the output format might be different. |

|  |  |  |
| --- | --- | --- |
|  |  | might be kludgey but i wonder if bash coproc would help, it basically can send stdout to a named pipe that makes it avail as a environment variable fike descriptor. |

|  |  |  |
| --- | --- | --- |
|  |  | Since bash can only handle one coproc at a time, using it behind the scenes blocks that function interactively. |

|  |  |  |
| --- | --- | --- |
|  |  | I've used `look` for years when I'm not sure how to spell some obscure word and I'm in a context where there isn't a built-in spellchecker (e.g. editing source code). I was today years old when I learned that looking up words from the system dictionary is just the convenient default and you can use it to search lines from any file. |

|  |  |  |
| --- | --- | --- |
|  |  | I don't recall where I heard it, but my understanding is that socat is the sort of successor to good ol' netcat. (Of course, don't ask me to compare each, nor know what socat brings that netcat lacks, etc.) |

|  |  |  |
| --- | --- | --- |
|  |  | Beware that look(1) does binary search, so it won't be reliable if the input file is not sorted in the exact way look(1) expects it.  On my system:  $ look asce  Ascella  Ascella's  $ grep -i ^asce /usr/share/dict/words  Ascella  Ascella's  ascend  ascendancy  ascendancy's  … |

|  |  |  |
| --- | --- | --- |
|  |  | And, fun fact, "socat" is how "kubectl port-forward" works so it'll likely be present on any machine behaving as a kubernetes Node |

|  |  |  |
| --- | --- | --- |
|  |  | Related: the Warp terminal has the concept of "workflows" - which are a list of snippets you can pull and use with auto-complete. I found that to be a good way to remember those one-liners that I only use once every two months. |

|  |  |  |
| --- | --- | --- |
|  |  | this is awesome, great work. this is really the first time i've opened up one of these "awesome list of X" repos and immediately learned a ton.  required reading for the bash newbie and mage alike. 10/10 will reference again and again. |