

Join GitHub today

GitHub is home to over 28 million developers working together to host and review code, manage projects, and build software together.

Sign up

Dismiss

[community-driven] guide to writing useful and modern bash scripts. seriously.

108 commits

1 branch

0 releases

9 contributors

CC-BY-4.0

Branch: master

New pull request

Find file

Clone or download

azet Merge pull request #14 from kallies/array_count ...

Latest commit bd2a13e on Jan 10

CONTRIBUTING.md	Update CONTRIBUTING.md	2 years ago
LICENSE	add proper CC BY 4.0 plaintext legalcode	4 years ago
README.md	Arrays start index with 0, awk with 1	a year ago

README.md

Community Bash Style Guide

Formerly known as: *hitchhikers guide to writing useful and modern bash scripts*

Introduction


This is intended to be a community driven bash style and best practice guide. There are a lot of blog posts and articles out there, but they do not always agree on certain issues, and mostly lack hints and best practices to achieve a specific goal (e.g. which userland utilities to use, which built-ins can be used instead and which userland utilities you should avoid at all cost). It's not that difficult to figure out a common strategy. so here it is.

Please participate: fork this repo, add your thoughts and experiences and open a pull request!

Here's how you write bash code that somebody else will actually understand, is unit testable and will work in different environments no matter what. please read the mentioned articles, you will not regret it. Furthermore people that will have to work with or maintain your scripts will not hate you in the future.

Table of Contents

- 1. [When to use Bash and when to avoid Bash](#)
- 2. [Style conventions](#)
- 3. [Resources](#)
- 4. [Common mistakes and useful tricks](#)
- 5. [Trivia section](#)
- 6. [Final remarks](#)
- 7. [Licensing](#)



When to use bash and when to avoid bash

it's rather simple:

- does it need to glue userland utilities together? use bash.
- does it need to do complex tasks (e.g. database queries)? use something else.

Why? You can do a lot of complicated tasks with bash, and I've had some experience in trying them all out in bash. It consumes a lot of time and is often very difficult to debug in comparison to dynamic programming languages such as python, ruby or even perl. You are simply going to waste valuable time, performance and nerve you could have spent better otherwise.

Style conventions

This is based on most common practices and guides available. It is also what I've seen others recommend and use and seemed most consistent and/or logical.

This should be seen as an ongoing discussion, you might want to open an Issue in this GitHub repository if you disagree.

- use the `#!/usr/bin/env bash` shebang wherever possible
- memorize and utilize `set -eu -o pipefail` at the very beginning of your code:
 - never write a script without `set -e` at the very very beginning. This instructs bash to terminate in case a command or chain of command finishes with a non-zero exit status. The idea behind this is that a proper program should never have unhandled error conditions. Use constructs like `if myprogramm --parameter ; then ...` for calls that might fail and require specific error handling. Use a cleanup trap for everything else.
 - use `set -u` in your scripts. This will terminate your scripts in case an uninitialized variable is accessed. This is especially important when developing shell libraries, since library code accessing uninitialized variables will fail in case it's used in another script which sets the `-u` flag. Obviously this flag is relevant to the script's/code's security.
 - use `set -o pipefail` to get an exit status from a pipeline (last non-zero will be returned).
- never use TAB for indentation:
 - consistently use two (2) or four (4) character indentation.
- **always** put parameters in double-quotes: `util "--argument" "${variable}"` .
- do not put `if .. then`, `while .. do` or `for .. do`, `case .. in` et cetera on a new line. this is more a tradition than actual convention. Most Bash programmers will use that style - for the sake of simplicity, let's do as well:

```
if ${event}; then
    ...
fi

while ${event}; do
    ...
done

for v in ${list[@]}; do
    ...
done
```

- never forget that you cannot put a space/blank between a variable name and it's value during an assignment (e.g. `ret = false` will not work)
- always set local function variables `local`
- write clear code
 - **never** obfuscate what the script is trying to do

- **never** shorten unnecessarily with a lot of commands per LoC chained with a semicolon
- Bash does not have a concept of public and private functions, thus;
 - public functions get generic names, whereas
 - private functions are prepended by two underscores (RedHat convention)
- try to stick to the `pushd`, `popd`, and `dirs` builtins for [directory stack manipulation](#) where sensible
- every line must have a maximum of eighty (80) terminal columns
- like in other dynamic languages, switch/case blocks should be aligned:

```
case ${contenders}; in
teller)  x=4 ;;
ulam)    c=1 ;;
neumann) v=7 ;;
esac
```

- only `trap` / handle signals you actually do care about
- use the builtin `readonly` when declaring constants and immutable variable
- assign integer variables, arrays, etc. with `typeset` / `declare` ([see also](#))
- always work with return values instead of strings passed from a function or userland utility (where applicable)
- write generic small check functions instead of large init and clean-up code:

```
# both functions return non-zero on error
function is_valid_string?() {
  [[ $@ =~ ^[A-Za-z0-9]*$ ]]
}
function is_integer?() {
  [[ $@ =~ ^-[0-9]+$ ]]
}
```

- be as modular and pluggable as possible and;
- if a project gets bigger, split it up into smaller files with clear and obvious naming scheme
- clearly document code parts that are not easily understood (long chains of piped commands for example)
- try to stick to [restricted mode](#) where sensible and possible to use: `set -r` (not supported in old versions of Bash). **Use with caution.** While this flag is *very useful for security* sensitive environments, scripts have to be written with the flag in mind. Adding restricted mode to an existing script will most likely break it.
- Thus, scripts should somewhat reflect the following general layout:

```
#!/usr/bin/env bash
#
# AUTHORS, LICENSE and DOCUMENTATION
#
set -eu -o pipefail

Readonly Variables
Global Variables

Import ("source scriptname") of external source code

Functions
`-. function local variables
`-. clearly describe interfaces: return either a code or string
```

```

Main
`-. option parsing
`-. log file and syslog handling
`-. temp. file and named pipe handling
`-. signal traps

```

To keep in mind:

- quoting of all variables passed when executing sub-shells or cli tools
- testing of functions, conditionals and flow (see style guide)
- makes restricted mode ("set -r") for security sense here?

- Silence is golden - like in any UNIX programm, avoid cluttering the terminal with useless output. [Read this.](#)

Resources

General documentation, style guides, tutorials and articles:

- <https://www.gnu.org/software/bash/manual/bashref.html>
- <http://wiki.bash-hackers.org/doku.php>
- <http://mywiki.woledge.org/BashFAQ>
- <https://google-styleguide.googlecode.com/svn/trunk/shell.xml>
- <http://www.kfirlavi.com/blog/2012/11/14/defensive-bash-programming/>
- <http://mywiki.woledge.org/BashWeaknesses>
- <https://github.com/docopt/docopts> (see: <http://docopt.org>)
- <http://isquared.nl/blog/2012/11/19/bash-lambda-expressions>
- <http://www.davidpashley.com/articles/writing-robust-shell-scripts/>

Linting and static analysis:

- <http://www.shellcheck.net> (<https://github.com/koalaman/shellcheck>)

Portability

- <https://github.com/duggan/shlint>
- <http://manpages.ubuntu.com/manpages/natty/man1/checkbashisms.1.html>

Test driven development and Unit testing:

- <https://github.com/sstephenson/bats>
- <https://github.com/mlafeldt/sharness>
- <https://bitheap.org/cram/>
- <https://github.com/rylnd/shpec>
- <https://github.com/roman-neuhauser/rnt>
- <https://code.google.com/p/shunit2/>
- <https://github.com/thinkerbot/ts>

Profiling:

- <https://github.com/sstephenson/bashprof>

Debugging:

- set -evx and bash -evx script.sh
- <http://bashdb.sourceforge.net/>

Presentations on this Document:

- 17/12/14: *Beautiful Bash: A community driven effort* by Aaron Zauner @ Vienna System Architects & DevOps/Security Meetup Vienna
 - http://www.slideshare.net/a_z_e_t/inpresentation
 - https://github.com/azet/talks/tree/master/2014/DevOpsSec-Meetup_Vienna/beautiful_bash-17_12_2014

Common mistakes and useful tricks

Never use backticks

wrong:

```
`call_command_in_subshell`
```

correct:

```
$(call_command_in_subshell)
```

Backticks are POSIX compliant but not 100% portable (doesn't work on Solaris 10 /bin/sh for example). Backticks also cannot be nested without being escaped (which looks just insane):

```
$(call_command_in_subshell $(different_command $(yetanother_as_parameter)))
```

Multiline pipe

instead of:

```
ls ${long_list_of_parameters} | grep ${foo} | grep -v grep | pgrep | wc -l | sort | uniq
```

do:

```
ls ${long_list_of_parameters} \
| grep ${foo} \
| grep -v grep \
| pgrep \
| wc -l \
| sort \
| uniq
```

..far more readable, isn't it?

Overusing grep and grep -v

please never do that. there's almost certainly a better way to express this.

for example:

```
ps ax | grep ${processname} | grep -v grep
```

versus using appropriate userland utilities:

```
pgrep ${processname}
```

Using `awk(1)` to print an element

stackoverflow is full of this behavior:

```
${listofthings} | awk '{ print $3 }' # get the third item
```

you may use bashisms instead:

```
listofthings=(${listofthings}) # convert to array
${listofthings[2]}             # get the third item (start counting from 0)
```

Use built in variable expansion instead of `sed/awk`

instead of this

```
VAR=FOO
printf ${VAR} | awk '{print tolower($0)}' # foo
```

use built in expansion like this

```
# ${VAR^} # upper single
# ${VAR^^} # upper all
# ${VAR,} # lower single
# ${VAR,,} # lower all
# ${VAR~} # swap case single
# ${VAR~~} # swap case all
```

```
VAR=BAR
printf ${VAR,,} # bar
```

same thing with string replacement.

```
# ${VAR/PATTERN/STRING} # single replacement
# ${VAR//PATTERN/STRING} # all match replacement
# Use ${VAR#PATTERN} ${VAR%PATTERN} ${VAR/PATTERN} for string removal

VAR=foofoobar
${VAR/foo/bar} # barfoobar
${VAR//foo/bar} # barbarbar
${VAR//foo} # bar
```

More examples and uses here: <http://wiki.bash-hackers.org/syntax/pe>

Do not use `seq` for ranges

use `{x..y}` instead!

e.g.:

```
for k in {1..100}; do
    $(do_awesome_stuff_with_input ${k})
done
```

the built-in range expression can do much more, see: <http://wiki.bash-hackers.org/syntax/expansion/brace#ranges>

Timeouts

The GNU coreutils program `timeout(1)` should be used to timeout processes:

https://www.gnu.org/software/coreutils/manual/html_node/timeout-invocation.html

caveat: `timeout(1)` might not be available on BSD, Mac OS X and UNIX systems.

Please use `printf` instead of `echo`

the bash builtin `printf` should be preferred to `echo` where possible. it does work like `printf` in C or any other high-level language, for reference see: <http://wiki.bash-hackers.org/commands/builtin/printf>

Bash arithmetic instead of `expr`

bash offers the whole nine yards of arithmetic expressions directly as built-in bashisms.

DO NOT USE `expr`

for reference see:

- http://wiki.bash-hackers.org/syntax/arith_expr
- http://www.softpanorama.org/Scripting/Shellorama/arithmetic_expressions.shtml

Never use `bc(1)` for modulo operations

it will come to hurt you, trust me.

`bc(1)` does not properly handle modulo operations most of the time: <https://superuser.com/questions/31445/gnu-bc-modulo-with-scale-other-than-0>

FIFO/named pipes

if you do not know what a named pipe is, please read this: http://wiki.bash-hackers.org/howto/redirection_tutorial

disown

`disown` is a bash built-in that can be used to remove a job from the job table of a bash script. for example, if you spawn a lot of sub processes, you can remove one or multiple of these processes with `disown` and the script will not care about it anymore.

see: <https://www.gnu.org/software/bash/manual/bashref.html#index-disown>

Basic parallelism

usually people use `&` to send a process to the background and `wait` to wait for the process to finish. people then often use named pipes, files and global variables to communicate between the parent and sub programs.

xargs

for file-based in-node parallelization, `xargs` is the easiest way to parallelize the processing of list elements.

```
# simple example: replace all occurrences of "foo" with "bar" in ".txt" files
# will process each file individually and up to 16 processes in parallel
find . -name "*.txt" | xargs -n1 -P16 -I{} sed -i 's/foo/bar/g' {}
```

```
# complex example: HDF5 repack for transparent compression of files
# find all ".h5" files in "${dirName}" and use up to 64 processes in parallel to independently compress them
find ${dirName} -name "*.h5" | xargs -n1 -P64 -I{} \
  sh -c 'echo "compress $1 ..." && \
  h5repack -i $1 -o $1.gz -f GZIP=1 && mv $1.gz $1' _ {}
```

coproc and GNU parallel

`coproc` can be used instead to have parallel jobs that can easily communicate with each other: <http://wiki.bash-hackers.org/syntax/keywords/coproc>

another excellent way to parallelize things in bash, especially for easy distribution over multiple hosts via SSH, is by using GNU parallel: https://www.gnu.org/software/parallel/parallel_tutorial.html

Trapping, exception handling and failing gracefully

`trap` is used for signal handling in bash, a generic error handling function may be used like this:

```
readonly banner="my first bash project >>"
function fail() {
    # generic fail function for bash scripts
    # arg: 1 - custom error message
    # arg: 2 - file
    # arg: 3 - line number
    # arg: 4 - exit status
    echo "${banner} ERROR: ${1}." >&2
    [[ ${2+defined} && ${3+defined} && ${4+defined} ]] && \
    echo "${banner} file: ${2}, line number: ${3}, exit code: ${4}. exiting!"

    # generic clean up code goes here (tempfiles, forked processes,..)

    exit 1
} ; trap 'fail "caught signal"' HUP KILL QUIT

do_stuff ${withinput} || fail "did not do stuff correctly" ${FILENAME} ${LINENO} $?
```

Trapping on `EXIT` instead of a specific signal is particularly useful for cleanup handlers since this executes the handler regardless of the reason for the script's termination. This also includes reaching the end of your script and aborts due to `set -e`.

You don't need cat

sometimes `cat` is not available, but with bash you can read files anyhow.

```
batterystatus=$(< /sys/class/power_supply/BAT0/status)
printf "%s\n" ${batterystatus}
```

Also avoid `cat` where reading a file can be achieved through passing the file name as a parameter. Never do `cat ${FILENAME}` | `grep -v ...`, instead use `grep -v ... ${FILENAME}`.

locking (file based)

`flock(1)` is an userland utility for managing file based locking from within shell scripts. It supports exclusive and shared locks.

Mutex (Mutual Exclusion)

mutex, although rather complex, is possible, too: <http://wiki.bash-hackers.org/howto/mutex>

Use the `getopt` builtin for command line parameters

```
printf "This script is: %s\n" ${0##*/}

[[ "${#}" == 0 ]] && {
    # no arguments
    printf "No options given: %s\n" ${OPTIND}
    exit 1
}
```



```

log=""      # numeric, log
table=""    # single fill
stores=( )  # array

# : after a letter is for string into parameter
while getopts ":dhls:t:" opt; do
  case "${opt}" in
    d) set -x ;;
    h) printf "Help page\n" ; exit ;;
    s) stores[${#stores[*]}]="${OPTARG}" ;;
    t)
      if [ -z "${table}" ]; then
        table="${OPTARG}"
      fi
    ;;
    l) (( log++ )) ;;
    *)
      printf "\n Option does not exist: %s\nOne option\n" "${OPTARG}"
      exit 1
    ;;
  esac
done

# set debug if log is more than two
[[ "${log}" >= 2 ]] && {
  set -x ; log=""
}
[[ "${log}" == "" ]] && unset log

```

Trivia section

This section outlines stuff that can be done in Bash but is not necessarily a good idea to do in Bash - might still come in handy for some corner cases or for curious Bash hackers, I've chosen to include that information.

Anonymous Functions (Lambdas)

Yup, it's possible. But you'll probably never need them, in case you really do, here's how:

```

function lambda() {
  _f=${1} ; shift
  function _l {
    eval ${_f};
  }
  _l ${*} ; unset _l
}

```

Using sockets with bash

although i do not really recommend it, it's possible to do simple (or even complex) socket operations in bash using the /dev/tcp and /dev/udp pseudo-devices: <http://wiki.bash-hackers.org/syntax/redirection>

example:

```

function recv() {
  local proto=${1} # tcp or udp
  local host=${2}  # hostname
  local port=${3}  # port number
  exec 3<>/dev/${proto}/${host}/${port}
  cat <&3
}

function send() {
  local msg=${1}

```

```
    echo -e "${msg}" >&3  
}  
  
[...]
```

you may consider using `nc` (netcat) or even the far more advanced program `socat` :

- <http://www.dest-unreach.org/socat/doc/socat.html>
- <http://stuff.mit.edu/afs/sipb/machine/penguin-lust/src/socat-1.7.1.2/EXAMPLES>

Foreign Function Interface

Tavis Ormandy wrote a FFI for Bash. You can directly access function from shared libraries in bash using `ctypes.sh` . It's a nice hack, but use is somewhat discouraged. Use userland utilities.

[ctypes.sh](#)

Final remarks

Every contribution is valuable to this effort. I'll do my best to incorporate all positive and negative feedback, criticism and am, of course, looking forward to people opening issues and pull requests for this project.

Please spread the word!

Licensing

This project is licensed under a [Creative Commons Attribution 4.0 International License](#).

The full legal code is contained in the `LICENSE` file distributed with this repository.

