styleguide

Shell Style Guide

Authored, revised and maintained by many Googlers.

Table of Contents

Background

Which Shell to Use

Bash is the only shell scripting language permitted for executables.

Executables must start with `#!/bin/bash` and minimal flags. Use `set` to set shell options so that calling your script as `bash script_name` does not break its functionality.

Restricting all executable shell scripts to *bash* gives us a consistent shell language that's installed on all our machines. In particular, this means there is generally no need to strive for POSIX-compatibility or otherwise avoid "bashisms".

The only exception to the above is where you're forced to by whatever you're coding for. For example some legacy operating systems or constrained execution environments may require plain Bourne shell for certain scripts.

When to use Shell

Shell should only be used for small utilities or simple wrapper scripts.

While shell scripting isn't a development language, it is used for writing various utility scripts throughout Google. This style guide is more a recognition of its use rather than a suggestion that it be used for widespread deployment.

Some guidelines:

- If you're mostly calling other utilities and are doing relatively little data manipulation, shell is an acceptable choice for the task.
- If performance matters, use something other than shell.
- If you are writing a script that is more than 100 lines long, or that uses non-straightforward control flow logic, you should rewrite it in a more structured language *now*. Bear in mind that scripts grow. Rewrite your script early to avoid a more time-consuming rewrite at a later date.
- When assessing the complexity of your code (e.g. to decide whether to switch languages) consider whether the code is easily maintainable by people other than its author.

Shell Files and Interpreter Invocation

File Extensions

Executables should have a `.sh` extension or no extension.

- If the executable will have a build rule that renames the source file then prefer to use a `.sh` extension. This enables you to use the recommended naming convention, with a source file like `foo.sh` and a build rule named `foo`.
- If the executable will be added directly to the user's `PATH`, then prefer to use no extension. It is not necessary to know what language a program is written in when executing it and shell doesn't require an extension so we prefer not to use one for executables that will be directly invoked by users. At the same time, consider whether it is preferable to deploy the output of a build rule rather than deploying the source file directly.
- If neither of the above apply, then either choice is acceptable.

Libraries must have a `.sh` extension and should not be executable.

SUID/SGID

SUID and SGID are *forbidden* on shell scripts.

There are too many security issues with shell that make it nearly impossible to secure sufficiently to allow SUID/SGID. While bash does make it difficult to run SUID, it's still possible on some platforms which is why we're being explicit about banning it.

Use `sudo` to provide elevated access if you need it.

Environment

STDOUT vs STDERR

All error messages should go to `STDERR`.

This makes it easier to separate normal status from actual issues.

A function to print out error messages along with other status information is recommended.

```
err() {
  echo "[$(date +'%Y-%m-%dT%H:%M:%S%z')]: $*" >&2
}

if ! do_something; then
  err "Unable to do_something"
  exit 1
fi
```

Comments

File Header

Start each file with a description of its contents.

Every file must have a top-level comment including a brief overview of its contents. A copyright notice and author information are optional.

Example:

```
#!/bin/bash
#
# Perform hot backups of Oracle databases.
```

Function Comments

Any function that is not both obvious and short must have a function header comment. Any function in a library must have a function header comment regardless of length or complexity.

It should be possible for someone else to learn how to use your program or to use a function in your library by reading the comments (and self-help, if provided) without reading the code.

All function header comments should describe the intended API behaviour using:

- Description of the function.
- Globals: List of global variables used and modified.
- Arguments: Arguments taken.
- Outputs: Output to STDOUT or STDERR.
- Returns: Returned values other than the default exit status of the last command run.

Example:

```
##########################################
# Cleanup files from the backup directory.
# Globals:
#   BACKUP_DIR
#   ORACLE_SID
# Arguments:
#   None
```

```
##########################################
function cleanup() {
  …
}

##########################################
# Get configuration directory.
# Globals:
#   SOMEDIR
# Arguments:
#   None
# Outputs:
#   Writes location to stdout
##########################################
function get_dir() {
  echo "${SOMEDIR}"
}

##########################################
# Delete a file in a sophisticated manner.
# Arguments:
#   File to delete, a path.
# Returns:
#   0 if thing was deleted, non-zero on error.
##########################################
function del_thing() {
  rm "$1"
}
```

Implementation Comments

Comment tricky, non-obvious, interesting or important parts of your code.

This follows general Google coding comment practice. Don't comment everything. If there's a complex algorithm or you're doing something out of the ordinary, put a short comment in.

TODO Comments

Use TODO comments for code that is temporary, a short-term solution, or good-enough but not perfect.

This matches the convention in the C++ Guide.

`TODO`s should include the string `TODO` in all caps, followed by the name, e-mail address, or other identifier of the person with the best context about the problem referenced by the `TODO`. The main purpose is to have a consistent `TODO` that can be searched to find out how to get more details upon request. A `TODO` is not a commitment that the person referenced

will fix the problem. Thus when you create a `TODO`, it is almost always your name that is given.

Examples:

```
# TODO(mrmonkey): Handle the unlikely edge cases (bug ####)
```

Formatting

While you should follow the style that's already there for files that you're modifying, the following are required for any new code.

Indentation

Indent 2 spaces. No tabs.

Use blank lines between blocks to improve readability. Indentation is two spaces. Whatever you do, don't use tabs. For existing files, stay faithful to the existing indentation.

Exception: The only exception for using tabs is for the body of `<<-` tab-indented here-document.

Line Length and Long Strings

Maximum line length is 80 characters.

If you have to write literal strings that are longer than 80 characters, this should be done with a here document or an embedded newline if possible.

Words that are longer than 80 chars and can't sensibly be split are ok, but where possible these items should be on a line of their own, or factored into a variable. Examples include file paths and URLs, particularly where string-matching them (such as `grep`) is valuable for maintenance.

```
# DO use 'here document's
cat <<END
I am an exceptionally long
string.
END

# Embedded newlines are ok too
long_string="I am an exceptionally
long string."
```

```
long_file="/i/am/an/exceptionally/looooooooooooooooooooooooooooooooooooooooooooo

long_string_with_long_file="i am including an exceptionally \
/very/long/file\
 in this long string."

# Long file converted into a shorter variable name with cleaner line breaking.
long_string_alt="i am an including an exceptionally ${long_file} in this long\
 string"



# Just because a line contains an exception doesn't mean the rest of the
# line shouldn't be wrapped like usual.

bad_long_string_with_long_file="i am including an exceptionally /very/long/file
```

Pipelines

Pipelines should be split one per line if they don't all fit on one line.

If a pipeline all fits on one line, it should be on one line.

If not, it should be split at one pipe segment per line with the pipe on the newline and a 2 space indent for the next section of the pipe. `\` should be consistently used to indicate line continuation. This applies to a chain of commands combined using `|` as well as to logical compounds using `||` and `&&` .

```
# All fits on one line
command1 | command2

# Long commands
command1 \
  | command2 \
  | command3 \
  | command4
```

This helps readability when distinguishing a pipeline from a regular long command continuation, particularly if the line is using both.

Comments will need to precede the whole pipeline. If the comment and pipeline are large and complex, then it is worth considering moving low level details of them aside by using a helper function.

Control Flow

Put `; then` and `; do` on the same line as the `if`, `for`, or `while`.

Control flow statements in shell are a bit different, but we follow the same principles as with braces when declaring functions. That is: `; then` and `; do` should be on the same line as the `if` / `for` / `while` / `until` / `select`. `else` should be on its own line and closing statements (`fi` and `done`) should be on their own line vertically aligned with the opening statement.

Example:

```
# If inside a function remember to declare the loop variable as
# a local to avoid it leaking into the global environment:
local dir
for dir in "${dirs_to_cleanup[@]}"; do
  if [[ -d "${dir}/${SESSION_ID}" ]]; then
    log_date "Cleaning up old files in ${dir}/${SESSION_ID}"
    rm "${dir}/${SESSION_ID}/"* || error_message
  else
    mkdir -p "${dir}/${SESSION_ID}" || error_message
  fi
done
```

Although it is possible to omit `in "$@"` in for loops we recommend consistently including it for clarity.

```
for arg in "$@"; do
  echo "argument: ${arg}"
done
```

Case statement

- Indent alternatives by 2 spaces.
- A one-line alternative needs a space after the close parenthesis of the pattern and before the `;;`.
- Long or multi-command alternatives should be split over multiple lines with the pattern, actions, and `;;` on separate lines.

The matching expressions are indented one level from the `case` and `esac`. Multiline actions are indented another level. In general, there is no need to quote match expressions. Pattern expressions should not be preceded by an open parenthesis. Avoid the `;&` and `;;&` notations.

```
case "${expression}" in
  a)
    variable="…"
    some_command "${variable}" "${other_expr}" …
    ;;
  absolute)
    actions="relative"
    another_command "${actions}" "${other_expr}" …
    ;;
  *)
    error "Unexpected expression '${expression}'"
    ;;
esac
```

Simple commands may be put on the same line as the pattern *and* `;;` as long as the expression remains readable. This is often appropriate for single-letter option processing. When the actions don't fit on a single line, put the pattern on a line on its own, then the actions, then `;;` also on a line of its own. When on the same line as the actions, use a space after the close parenthesis of the pattern and another before the `;;` .

```
verbose='false'
aflag=''
bflag=''
files=''
while getopts 'abf:v' flag; do
  case "${flag}" in
    a) aflag='true' ;;
    b) bflag='true' ;;
    f) files="${OPTARG}" ;;
    v) verbose='true' ;;
    *) error "Unexpected option ${flag}" ;;
  esac
done
```

Variable expansion

In order of precedence: Stay consistent with what you find; quote your variables; prefer `"${var}"` over `"$var"` .

These are strongly recommended guidelines but not mandatory regulation. Nonetheless, the fact that it's a recommendation and not mandatory doesn't mean it should be taken lightly or downplayed.

They are listed in order of precedence.

- Stay consistent with what you find for existing code.
- Quote variables, see Quoting section below.
- Don't brace-delimit single character shell specials / positional parameters, unless strictly necessary or avoiding deep confusion.

Prefer brace-delimiting all other variables.

```
# Section of *recommended* cases.

# Preferred style for 'special' variables:
echo "Positional: $1" "$5" "$3"
echo "Specials: !=$!, -=$-, _=$_. ?=$?, #=$# *=$* @=$@ \$=$$ …"

# Braces necessary:
echo "many parameters: ${10}"

# Braces avoiding confusion:
# Output is "a0b0c0"
set -- a b c
echo "${1}0${2}0${3}0"

# Preferred style for other variables:
echo "PATH=${PATH}, PWD=${PWD}, mine=${some_var}"
while read -r f; do
  echo "file=${f}"
done < <(find /tmp)


# Section of *discouraged* cases

# Unquoted vars, unbraced vars, brace-delimited single letter
# shell specials.
echo a=$avar "b=$bvar" "PID=${$}" "${1}"

# Confusing use: this is expanded as "${1}0${2}0${3}0",
# not "${10}${20}${30}
set -- a b c
echo "$10$20$30"
```

NOTE: Using braces in `${var}` is *not* a form of quoting. "Double quotes" must be used *as well*.

Quoting

- Always quote strings containing variables, command substitutions, spaces or shell meta characters, unless careful unquoted expansion is required or it's a shell-internal

integer (see next point).

- Use arrays for safe quoting of lists of elements, especially command-line flags. See Arrays below.
- Optionally quote shell-internal, readonly special variables that are defined to be integers: `$?`, `$#`, `$$`, `$!`. Prefer quoting of "named" internal integer variables, e.g. PPID etc for consistency.
- Prefer quoting strings that are "words" (as opposed to command options or path names).
- Be aware of the quoting rules for pattern matches in `[[ … ]]`. See the Test, `[ … ]`, and `[[ … ]]` section below.
- Use `"$@"` unless you have a specific reason to use `$*`, such as simply appending the arguments to a string in a message or log.

```
# 'Single' quotes indicate that no substitution is desired.
# "Double" quotes indicate that substitution is required/tolerated.

# Simple examples

# "quote command substitutions"
# Note that quotes nested inside "$()" don't need escaping.
flag="$(some_command and its args "$@" 'quoted separately')"

# "quote variables"
echo "${flag}"

# Use arrays with quoted expansion for lists.
declare -a FLAGS
FLAGS=( --foo --bar='baz' )
readonly FLAGS
mybinary "${FLAGS[@]}"

# It's ok to not quote internal integer variables.
if (( $# > 3 )); then
  echo "ppid=${PPID}"
fi

# "never quote literal integers"
value=32
# "quote command substitutions", even when you expect integers
number="$(generate_number)"

# "prefer quoting words", not compulsory
readonly USE_INTEGER='true'

# "quote shell meta characters"
echo 'Hello stranger, and well met. Earn lots of $$$'
```

```
echo "Process $$: Done making \$\$\$."

# "command options or path names"
# ($1 is assumed to contain a value here)
grep -li Hugo /dev/null "$1"

# Less simple examples
# "quote variables, unless proven false": ccs might be empty
git send-email --to "${reviewers}" ${ccs:+"--cc" "${ccs}"}

# Positional parameter precautions: $1 might be unset
# Single quotes leave regex as-is.
grep -cP '([Ss]pecial|\|?characters*)$' ${1:+"$1"}

# For passing on arguments,
# "$@" is right almost every time, and
# $* is wrong almost every time:
#
# * $* and $@ will split on spaces, clobbering up arguments
#   that contain spaces and dropping empty strings;
# * "$@" will retain arguments as-is, so no args
#   provided will result in no args being passed on;
#   This is in most cases what you want to use for passing
#   on arguments.
# * "$*" expands to one argument, with all args joined
#   by (usually) spaces,
#   so no args provided will result in one empty string
#   being passed on.
#
# Consult
# https://www.gnu.org/software/bash/manual/html_node/Special-Parameters.html and
# https://mywiki.wooledge.org/BashGuide/Arrays for more

(set -- 1 "2 two" "3 three tres"; echo $#; set -- "$*"; echo "$#, $@")
(set -- 1 "2 two" "3 three tres"; echo $#; set -- "$@"; echo "$#, $@")
```

Features and Bugs

ShellCheck

The ShellCheck project identifies common bugs and warnings for your shell scripts. It is recommended for all scripts, large or small.

Command Substitution

Use `$(command)` instead of backticks.

Nested backticks require escaping the inner ones with `\` . The `$(command)` format doesn't change when nested and is easier to read.

Example:

```
# This is preferred:
var="$(command "$(command1)")"


# This is not:
var="`command \`command1\``"
```

Test, `[ … ]` , and `[[ … ]]`

`[[ … ]]` is preferred over `[ … ]` , `test` and `/usr/bin/[` .

`[[ … ]]` reduces errors as no pathname expansion or word splitting takes place between `[[` and `]]` . In addition, `[[ … ]]` allows for pattern and regular expression matching, while `[ … ]` does not.

```
# This ensures the string on the left is made up of characters in
# the alnum character class followed by the string name.
# Note that the RHS should not be quoted here.
if [[ "filename" =~ ^[[:alnum:]]+name ]]; then
  echo "Match"
fi

# This matches the exact pattern "f*" (Does not match in this case)
if [[ "filename" == "f*" ]]; then
  echo "Match"
fi


# This gives a "too many arguments" error as f* is expanded to the
# contents of the current directory. It might also trigger the
# "unexpected operator" error because `[` does not support `==`, only `=`.
if [ "filename" == f* ]; then
  echo "Match"
fi
```

For the gory details, see E14 in the Bash FAQ

Testing Strings

Use quotes rather than filler characters where possible.

Bash is smart enough to deal with an empty string in a test. So, given that the code is much easier to read, use tests for empty/non-empty strings or empty strings rather than filler characters.

```
# Do this:
if [[ "${my_var}" == "some_string" ]]; then
  do_something
fi

# -z (string length is zero) and -n (string length is not zero) are
# preferred over testing for an empty string
if [[ -z "${my_var}" ]]; then
  do_something
fi

# This is OK (ensure quotes on the empty side), but not preferred:
if [[ "${my_var}" == "" ]]; then
  do_something
fi



# Not this:
if [[ "${my_var}X" == "some_stringX" ]]; then
  do_something
fi
```

To avoid confusion about what you're testing for, explicitly use `-z` or `-n`.

```
# Use this
if [[ -n "${my_var}" ]]; then
  do_something
fi



# Instead of this
if [[ "${my_var}" ]]; then
  do_something
fi
```

For clarity, use `==` for equality rather than `=` even though both work. The former encourages the use of `[[` and the latter can be confused with an assignment. However, be careful when using `<` and `>` in `[[ … ]]` which performs a lexicographical comparison. Use `(( … ))` or `-lt` and `-gt` for numerical comparison.

```
# Use this
if [[ "${my_var}" == "val" ]]; then
  do_something
fi

if (( my_var > 3 )); then
  do_something
fi

if [[ "${my_var}" -gt 3 ]]; then
  do_something
fi



# Instead of this
if [[ "${my_var}" = "val" ]]; then
  do_something
fi

# Probably unintended lexicographical comparison.
if [[ "${my_var}" > 3 ]]; then
  # True for 4, false for 22.
  do_something
fi
```

Wildcard Expansion of Filenames

Use an explicit path when doing wildcard expansion of filenames.

As filenames can begin with a `-`, it's a lot safer to expand wildcards with `./*` instead of `*`.

```
# Here's the contents of the directory:
# -f  -r  somedir  somefile

# Incorrectly deletes almost everything in the directory by force
psa@bilby$ rm -v *
removed directory: `somedir'
removed `somefile'



# As opposed to:
psa@bilby$ rm -v ./*
removed `./-f'
removed `./-r'
rm: cannot remove `./somedir': Is a directory
```

```
removed `./somefile'
```

Eval

`eval` should be avoided.

Eval munges the input when used for assignment to variables and can set variables without making it possible to check what those variables were.

```
# What does this set?
# Did it succeed? In part or whole?
eval $(set_my_variables)

# What happens if one of the returned values has a space in it?
variable="$(eval some_function)"
```

Arrays

Bash arrays should be used to store lists of elements, to avoid quoting complications. This particularly applies to argument lists. Arrays should not be used to facilitate more complex data structures (see When to use Shell above).

Arrays store an ordered collection of strings, and can be safely expanded into individual elements for a command or loop.

Using a single string for multiple command arguments should be avoided, as it inevitably leads to authors using `eval` or trying to nest quotes inside the string, which does not give reliable or readable results and leads to needless complexity.

```
# An array is assigned using parentheses, and can be appended to
# with +=( … ).
declare -a flags
flags=(--foo --bar='baz')
flags+=(--greeting="Hello ${name}")
mybinary "${flags[@]}"


# Don't use strings for sequences.
flags='--foo --bar=baz'
flags+=' --greeting="Hello world"'  # This won't work as intended.
mybinary ${flags}


# Command expansions return single strings, not arrays. Avoid
```

```
# unquoted expansion in array assignments because it won't
# work correctly if the command output contains special
# characters or whitespace.

# This expands the listing output into a string, then does special keyword
# expansion, and then whitespace splitting.  Only then is it turned into a
# list of words.  The ls command may also change behavior based on the user's
# active environment!
declare -a files=($(ls /directory))

# The get_arguments writes everything to STDOUT, but then goes through the
# same expansion process above before turning into a list of arguments.
mybinary $(get_arguments)
```

Arrays Pros

- Using Arrays allows lists of things without confusing quoting semantics. Conversely, not using arrays leads to misguided attempts to nest quoting inside a string.
- Arrays make it possible to safely store sequences/lists of arbitrary strings, including strings containing whitespace.

Arrays Cons

Using arrays can risk a script's complexity growing.

Arrays Decision

Arrays should be used to safely create and pass around lists. In particular, when building a set of command arguments, use arrays to avoid confusing quoting issues. Use quoted expansion – `"${array[@]}"` – to access arrays. However, if more advanced data manipulation is required, shell scripting should be avoided altogether; see above.

Pipes to While

Use process substitution or the `readarray` builtin (bash4+) in preference to piping to `while`. Pipes create a subshell, so any variables modified within a pipeline do not propagate to the parent shell.

The implicit subshell in a pipe to `while` can introduce subtle bugs that are hard to track down.

```
last_line='NULL'
your_command | while read -r line; do
  if [[ -n "${line}" ]]; then
    last_line="${line}"
```

```
  fi
done

# This will always output 'NULL'!
echo "${last_line}"
```

Using process substitution also creates a subshell. However, it allows redirecting from a subshell to a `while` without putting the `while` (or any other command) in a subshell.

```
last_line='NULL'
while read line; do
  if [[ -n "${line}" ]]; then
    last_line="${line}"
  fi
done < <(your_command)

# This will output the last non-empty line from your_command
echo "${last_line}"
```

Alternatively, use the `readarray` builtin to read the file into an array, then loop over the array's contents. Notice that (for the same reason as above) you need to use a process substitution with `readarray` rather than a pipe, but with the advantage that the input generation for the loop is located before it, rather than after.

```
last_line='NULL'
readarray -t lines < <(your_command)
for line in "${lines[@]}"; do
  if [[ -n "${line}" ]]; then
    last_line="${line}"
  fi
done
echo "${last_line}"
```

> Note: Be cautious using a for-loop to iterate over output, as in `for var in $(...)`, as the output is split by whitespace, not by line. Sometimes you will know this is safe because the output can't contain any unexpected whitespace, but where this isn't obvious or doesn't improve readability (such as a long command inside `$(...)`), a `while read` loop or `readarray` is often safer and clearer.

Arithmetic

Always use `(( … ))` or `$(( … ))` rather than `let` or `$[ … ]` or `expr`.

Never use the `$[ … ]` syntax, the `expr` command, or the `let` built-in.

`<` and `>` don't perform numerical comparison inside `[[ … ]]` expressions (they perform lexicographical comparisons instead; see Testing Strings). For preference, don't use `[[ … ]]` *at all* for numeric comparisons, use `(( … ))` instead.

It is recommended to avoid using `(( … ))` as a standalone statement, and otherwise be wary of its expression evaluating to zero

- particularly with `set -e` enabled. For example, `set -e; i=0; (( i++ ))` will cause the shell to exit.

```
# Simple calculation used as text - note the use of $(( … )) within
# a string.
echo "$(( 2 + 2 )) is 4"

# When performing arithmetic comparisons for testing
if (( a < b )); then
  …
fi

# Some calculation assigned to a variable.
(( i = 10 * j + 400 ))



# This form is non-portable and deprecated
i=$[2 * 10]

# Despite appearances, 'let' isn't one of the declarative keywords,
# so unquoted assignments are subject to globbing wordsplitting.
# For the sake of simplicity, avoid 'let' and use (( … ))
let i="2 + 2"

# The expr utility is an external program and not a shell builtin.
i=$( expr 4 + 4 )

# Quoting can be error prone when using expr too.
i=$( expr 4 '*' 4 )
```

Stylistic considerations aside, the shell's built-in arithmetic is many times faster than `expr`.

When using variables, the `${var}` (and `$var`) forms are not required within `$(( … ))`. The shell knows to look up `var` for you, and omitting the `${…}` leads to cleaner code. This is slightly contrary to the previous rule about always using braces, so this is a recommendation only.

```
# N.B.: Remember to declare your variables as integers when
# possible, and to prefer local variables over globals.
```

```
local -i hundred="$(( 10 * 10 ))"
declare -i five="$(( 10 / 2 ))"

# Increment the variable "i" by three.
# Note that:
#  - We do not write ${i} or $i.
#  - We put a space after the (( and before the )).
(( i += 3 ))

# To decrement the variable "i" by five:
(( i -= 5 ))

# Do some complicated computations.
# Note that normal arithmetic operator precedence is observed.
hr=2
min=5
sec=30
echo "$(( hr * 3600 + min * 60 + sec ))" # prints 7530 as expected
```

Aliases

Although commonly seen in `.bashrc` files, aliases should be avoided in scripts. As the Bash manual notes:

> For almost every purpose, shell functions are preferred over aliases.

Aliases are cumbersome to work with because they require carefully quoting and escaping their contents, and mistakes can be hard to notice.

```
# this evaluates $RANDOM once when the alias is defined,
# so the echo'ed string will be the same on each invocation
alias random_name="echo some_prefix_${RANDOM}"
```

Functions provide a superset of alias' functionality and should always be preferred. .

```
random_name() {
  echo "some_prefix_${RANDOM}"
}

# Note that unlike aliases function's arguments are accessed via $@
fancy_ls() {
  ls -lh "$@"
}
```

Naming Conventions

Function Names

Lower-case, with underscores to separate words. Separate libraries with `::`. Parentheses are required after the function name. The keyword `function` is optional, but must be used consistently throughout a project.

If you're writing single functions, use lowercase and separate words with underscore. If you're writing a package, separate package names with `::`. However, functions intended for interactive use may choose to avoid colons as it can confuse bash auto-completion.

Braces must be on the same line as the function name (as with other languages at Google) and no space between the function name and the parenthesis.

```
# Single function
my_func() {
  …
}

# Part of a package
mypackage::my_func() {
  …
}
```

The `function` keyword is extraneous when "()" is present after the function name, but enhances quick identification of functions.

Variable Names

Same as for function names.

Variables names for loops should be similarly named for any variable you're looping through.

```
for zone in "${zones[@]}"; do
  something_with "${zone}"
done
```

Constants, Environment Variables, and readonly Variables

Constants and anything exported to the environment should be capitalized, separated with underscores, and declared at the top of the file.

```
# Constant
readonly PATH_TO_FILES='/some/path'

# Both constant and exported to the environment
declare -xr ORACLE_SID='PROD'
```

For the sake of clarity `readonly` or `export` is recommended vs. the equivalent `declare` commands. You can do one after the other, like:

```
# Constant
readonly PATH_TO_FILES='/some/path'
export PATH_TO_FILES
```

It's OK to set a constant at runtime or in a conditional, but it should be made readonly immediately afterwards.

```
ZIP_VERSION="$(dpkg --status zip | sed -n 's/^Version: //p')"
if [[ -z "${ZIP_VERSION}" ]]; then
  ZIP_VERSION="$(pacman -Q --info zip | sed -n 's/^Version *: //p')"
fi
if [[ -z "${ZIP_VERSION}" ]]; then
  handle_error_and_quit
fi
readonly ZIP_VERSION
```

Source Filenames

Lowercase, with underscores to separate words if desired.

This is for consistency with other code styles in Google: `maketemplate` or `make_template` but not `make-template`.

Use Local Variables

Declare function-specific variables with `local`.

Ensure that local variables are only seen inside a function and its children by using `local` when declaring them. This avoids polluting the global namespace and inadvertently setting variables that may have significance outside the function.

Declaration and assignment must be separate statements when the assignment value is provided by a command substitution; as the `local` builtin does not propagate the exit code from the command substitution.

```
my_func2() {
  local name="$1"

  # Separate lines for declaration and assignment:
  local my_var
  my_var="$(my_func)"
  (( $? == 0 )) || return

  …
}
```

```
my_func2() {
  # DO NOT do this:
  # $? will always be zero, as it contains the exit code of 'local', not my_func
  local my_var="$(my_func)"
  (( $? == 0 )) || return

  …
}
```

Function Location

Put all functions together in the file just below constants. Don't hide executable code between functions. Doing so makes the code difficult to follow and results in nasty surprises when debugging.

If you've got functions, put them all together near the top of the file. Only includes, `set` statements and setting constants may be done before declaring functions.

main

A function called `main` is required for scripts long enough to contain at least one other function.

In order to easily find the start of the program, put the main program in a function called `main` as the bottom-most function. This provides consistency with the rest of the code base as well as allowing you to define more variables as `local` (which can't be done if the main code is not a function). The last non-comment line in the file should be a call to `main`:

```
main "$@"
```

Obviously, for short scripts where it's just a linear flow, `main` is overkill and so is not

required.

Calling Commands

Checking Return Values

Always check return values and give informative return values.

For unpiped commands, use `$?` or check directly via an `if` statement to keep it simple.

Example:

```
if ! mv "${file_list[@]}" "${dest_dir}/"; then
  echo "Unable to move ${file_list[*]} to ${dest_dir}" >&2
  exit 1
fi

# Or
mv "${file_list[@]}" "${dest_dir}/"
if (( $? != 0 )); then
  echo "Unable to move ${file_list[*]} to ${dest_dir}" >&2
  exit 1
fi
```

Bash also has the `PIPESTATUS` variable that allows checking of the return code from all parts of a pipe. If it's only necessary to check success or failure of the whole pipe, then the following is acceptable:

```
tar -cf - ./* | ( cd "${dir}" && tar -xf - )
if (( PIPESTATUS[0] != 0 || PIPESTATUS[1] != 0 )); then
  echo "Unable to tar files to ${dir}" >&2
fi
```

However, as `PIPESTATUS` will be overwritten as soon as you do any other command, if you need to act differently on errors based on where it happened in the pipe, you'll need to assign `PIPESTATUS` to another variable immediately after running the command (don't forget that `[` is a command and will wipe out `PIPESTATUS`).

```
tar -cf - ./* | ( cd "${DIR}" && tar -xf - )
return_codes=( "${PIPESTATUS[@]}" )
if (( return_codes[0] != 0 )); then
  do_something
fi
```

```
if (( return_codes[1] != 0 )); then
  do_something_else
fi
```

Builtin Commands vs. External Commands

Given the choice between invoking a shell builtin and invoking a separate process, choose the builtin.

We prefer the use of builtins such as the *Parameter Expansion* functionality provided by `bash` as it's more efficient, robust, and portable (especially when compared to things like `sed`). See also the `=~` operator.

Examples:

```
# Prefer this:
addition="$(( X + Y ))"
substitution="${string/#foo/bar}"
if [[ "${string}" =~ foo:(\d+) ]]; then
  extraction="${BASH_REMATCH[1]}"
fi


# Instead of this:
addition="$(expr "${X}" + "${Y}")"
substitution="$(echo "${string}" | sed -e 's/^foo/bar/')"
extraction="$(echo "${string}" | sed -e 's/foo:\([0-9]\)/\1/')"
```

When in Doubt: Be Consistent

Using one style consistently through our codebase lets us focus on other (more important) issues. Consistency also allows for automation. In many cases, rules that are attributed to "Be Consistent" boil down to "Just pick one and stop worrying about it"; the potential value of allowing flexibility on these points is outweighed by the cost of having people argue over them.

However, there are limits to consistency. It is a good tie breaker when there is no clear technical argument, nor a long-term direction. Consistency should not generally be used as a justification to do things in an old style without considering the benefits of the new style, or the tendency of the codebase to converge on newer styles over time.