# Contributing to bash-completion

Contributions to the bash completion project are more than

welcome. Fixes, clean-ups and improvements of existing code are much

appreciated, as are completion functions for new commands.

However, before submitting a completion to us, first consider submitting it to

the project that ships the commands your completion is for. Having the

completion shipped along with the command opens up some liberties we don't have

if the completion is included with bash-completion. For example, we generally

do not want to hardcode lists of available command options and their

completions, because they quite probably vary between versions of the completed

command, and therefore resort to scraping --help output and the like. While we

do fairly well there, depending on the command, this can be fragile or

expensive, or just not possible. If the completion is shipped alongside the

command, they can be kept in sync and use more hardcoding etc. They are also

more likely to be maintained and/or watched by people intimately familiar with

the completed commands. See instructions in README.md how to install completion

files from other projects so they are automatically enabled and dynamically

loaded by bash-completion.

On the other hand, we do have a pretty nice test suite and a bunch of helper

functions that you may find useful. And a whole slew of completions in one

package. Our functions can be used from "external" completions as well, just

make sure you test for their existence and/or fail gracefully if you intend

your completion to be usable without having bash-completion installed.

It's nowhere near clear cut always what is the best place for the completion,

upstream project or us. Even if it would seem to be upstream, not all upstreams

are interested in shipping completions, or their install systems might not

easily support installing completion files properly. But give it some thought,

and ask if unsure.

If you wish to contribute code to us, volunteering for long term maintainership

of your code within bash-completion is welcome. When exactly you will be asked

to do that depends on the case; don't be disappointed if it does or doesn't

happen instantly.

Also, please bare the following coding guidelines in mind:

- Do not use Perl, Ruby, Python etc. to do text processing unless the

command for which you are writing the completion code implies the

presence of one of those languages.

For example, if you were writing completion code for perldoc(1), the

use of Perl to achieve your goal would be acceptable. irb(1)

completion would similarly make the use of Ruby acceptable.

Even so, please consider alternatives to these large and slow to

start interpreters. Use lightweight programs such as grep(1), awk(1)

and sed(1).

- Use the full power of bash >= 4.2. We no longer support earlier bash

versions, so you may as well use all the features of that version of

bash to optimise your code. However, be careful when using features

added since bash 4.2, since not everyone will be able to use them.

For example, extended globs often enable you to avoid the use of

external programs, which are expensive to fork and execute, so do

make full use of those:

`?(pattern-list)` - match zero or one occurrences of patterns

`\*(pattern-list)` - match zero or more occurrences of patterns

`+(pattern-list)` - match one or more occurrences of patterns

`@(pattern-list)` - match exactly one of the given patterns

`!(pattern-list)` - match anything except one of the given patterns

- Following on from the last point, be sparing with the use of

external processes whenever you can. Completion functions need to be

fast, so sacrificing some code legibility for speed is acceptable.

For example, judicious use of sed(1) can save you from having to

call grep(1) and pipe the output to cut(1), which saves a fork(2)

and exec(3).

Sometimes you don't even need sed(1) or other external programs at

all, though. Use of constructs such as `${parameter#word}`,

`${parameter%word}` and `${parameter/pattern/string}` can provide

you a lot of power without having to leave the shell.

For example, if `$foo` contains the path to an executable,

`${foo##\*/}` will give you the basename of the program, without

having to call basename(1). Similarly, `${foo%/\*}` will give you the

dirname, without having to call dirname(1).

As another example,

```shell

bar=$(echo $foo | command sed -e 's/bar/baz/g')

```

can be replaced by:

```shell

bar=${foo//bar/baz}

```

These forms of parameter substitutions can also be used on arrays,

which makes them very powerful (if a little slow).

- We want our completions to work in `posix` and `nounset` modes.

Unfortunately due to a bash < 5.1 bug, toggling POSIX mode interferes

with keybindings and should not be done. This rules out use of

process substitution which causes syntax errors in POSIX mode.

Instead of toggling `nounset` mode, make sure to test whether

variables are set (e.g. with `[[ -v varname ]]`) or use default

expansion (e.g. `${varname-}`).

- Prefer `compgen -W '...' -- $cur` over embedding `$cur` in external

command arguments (often e.g. sed, grep etc) unless there's a good

reason to embed it. Embedding user input in command lines can result

in syntax errors and other undesired behavior, or messy quoting

requirements when the input contains unusual characters. Good

reasons for embedding include functionality (if the thing does not

sanely work otherwise) or performance (if it makes a big difference

in speed), but all embedding cases should be documented with

rationale in comments in the code.

- When completing available options, offer only the most descriptive

ones as completion results if there are multiple options that do the

same thing. Usually this means that long options should be preferred

over the corresponding short ones. This way the user is more likely

to find what she's looking for and there's not too much noise to

choose from, and there are less situations where user choice would

be needed in the first place. Note that this concerns only display

of available completions; argument processing/completion for options

that take an argument should be made to work with all known variants

for the functionality at hand. For example if `-s`, `-S`, and

`--something` do the same thing and require an argument, offer only

`--something` as a completion when completing option names starting

with a dash, but do implement required argument processing for all

`-s`, `-S`, and `--something`. Note that GNU versions of various

standard commands tend to have long options while other userland

implementations of the same commands may not have them, and it would

be good to have the completions work for as many userlands as

possible so things aren't always that simple.

- Do not write to the file-system under any circumstances. This can

create race conditions, is inefficient, violates the principle of

least surprise and lacks robustness.

- Use printf(1) instead of echo(1) for portability reasons, and be

sure to invoke commands that are often found aliased (such as `ls`

or `grep` etc) using the `command` (or `builtin`) command as

appropriate.

- Make small, incremental commits that do one thing. Don't cram

unrelated changes into a single commit.

- If your code was written for a particular platform, try to make it

portable to other platforms, so that everyone may enjoy it. If your

code works only with the version of a binary on a particular

platform, ensure that it will not be loaded on other platforms that

have a command with the same name.

In particular, do not use GNU extensions to commands like sed and

awk if you can write your code another way. If you really, REALLY must

use them, do so if there's no other sane way to do what you're doing.

The "Shell and Utilities" volume of the POSIX specification is a good

starting reference for portable use of various utilities, see

<https://pubs.opengroup.org/onlinepubs/9699919799/>.

- Use an editor that supports EditorConfig, see <https://editorconfig.org/>,

and format source code according to our settings.

- Read the existing source code for examples of how to solve

particular problems. Read the bash man page for details of all the

programming tools available to you within the shell.

- Please test your code thoroughly before sending it to us. We don't

have access to all the commands for which we are sent completion

functions, so we are unable to test them all personally. If your

code is accepted into the distribution, a lot of people will try it

out, so try to do a thorough job of eradicating all the bugs before

you send it to us. If at all practical, \*\*add test cases\*\* to our

test suite (in the test/ dir) that verify that the code does what it

is intended to do, fixes issues it intends to fix, etc.

- In addition to running the test suite, there are a few scripts in the test/

dir that catch some common issues, see and use for example runLint.

- File bugs, enhancement, and pull requests at GitHub,

<https://github.com/scop/bash-completion>.

Sending them to the developers might work too, but is really strongly

discouraged as bits are more likely to fall through the cracks that

way compared to the tracker. Just use GitHub. If that's not an

option for some reason and you want to use email to send patches,

send them as attachments formatted by `git format-patch` or directly

with `git send-email`.