How can I see what command was actually run in the shell, through an alias or function

Asked 6 years, 3 months ago Modified 10 months ago Viewed 5k times

27

I have a bash function (or alias), for instance `function install() {sudo apt-get install $@}`. When running the command `install dicelab`, what I expect will actually be run is `sudo apt-get install dicelab`. Where can I see what was actually run by the shell? I would like to make sure that my more complicated aliases are working as expected.

bash shell debugging

Share Follow

edited May 4, 2017 at 21:59

Gilles 'SO- stop being evil'
806k 194 1669 2171

asked May 4, 2017 at 7:30

Alon Aviv
279 3 4

Is that `$@` part of your alias? Remember that aliases don't really support arguments, that will expand to the positional parameters (if any) of the context calling the alias. The usual way of running `somealias some args` works just by expanding the alias and leaving the arguments to follow it. If you actually want to be able to access the arguments, use a function, and quote the `"$@"` – ilkkachu May 4, 2017 at 7:54

My bad, I meant a bash function. I'll change it. – Alon Aviv May 4, 2017 at 8:06

1 @AlonAviv, Good. :) Still, best get in the habit of quoting that `"$@"`, arguments with whitespace or glob characters will burn otherwise. – ilkkachu May 4, 2017 at 8:53

4 Answers

Sorted by: Highest score (default)

35

Use `set -x` in the shell.

```
$ alias hello='echo hello world!'
$ hello
```

```
hello world!
```

```
$ set -x
$ hello
+ echo hello world!
hello world!
```

Using `set -x` turns on the `xtrace` shell option (`set +x` turns it off) and should work in all Bourne-like shells, like `bash`, `dash` `ksh93`, `pdksh` and `zsh`. This prompts the shell to display the command that gets executed after alias expansions and variable expansions etc. has been performed.

The output will be on the standard error stream of the shell (just like the ordinary prompt) so it will not interfere with redirections of standard output, and it will be preceded by a prompt as defined by the `PS4` shell variable (often `+␣` by default).

Example with a few functions:

```
$ world () { echo "world"; }
$ hello () { echo "hello"; }
$ helloworld () { printf '%s %s!\n' "$(hello)" "$(world)"; }
```

```
$ helloworld
hello world!
```

```
$ set -x
$ helloworld
+ helloworld
++ hello
++ echo hello
++ world
++ echo world
+ printf '%s %s!\n' hello world
hello world!
```

With your specific example (with syntax corrected and added quotes):

```
$ install () { sudo apt-get install "$@"; }
$ set -x
$ install dicelab
+ install dicelab
```

```
+ sudo apt-get install dicelab
bash: sudo: command not found
```

(I don't use or have `sudo` on my system, so that error is expected.)

Note that there is a common utility already called `install`, so naming your function something else (`aptin`?) may be needed if you at some point want to use that utility.

Note that the trace output is *debugging output*. It is a *representation* of what the shell is doing while executing your command. The output that you see on the screen may not be suitable for shell input.

Also note that I was using `bash` above. Other shells may have another default trace prompt (`zsh` incorporates the string `zsh` and the current command's history number, for example) or may not "stack" the prompts up like `bash` does for nested calls.

~~I'm running~~ I used to run with `set -x` in all my interactive shells by default. It's nice to see what actually got executed... but I've noticed that programmable tab completion etc. may cause unwanted trace output in some shells, and some shells are a bit verbose in their default trace output (e.g. `zsh`).

edited Apr 16, 2020 at 11:24

answered May 4, 2017 at 7:36

Kusalananda ♦

319k 36 629 933