Published in Better Programming

Shinichi Okada Follow

Feb 10, 2021 · 10 min read ★ · Listen

Save

27 Simple Bash Scripting Tips for Beginners

Don’t be afraid of Bash scripting. It will be easier after reading these tips

[Updated on 2021–02-18. Codes changed to Gist and added links]

Table of Contents

Introduction

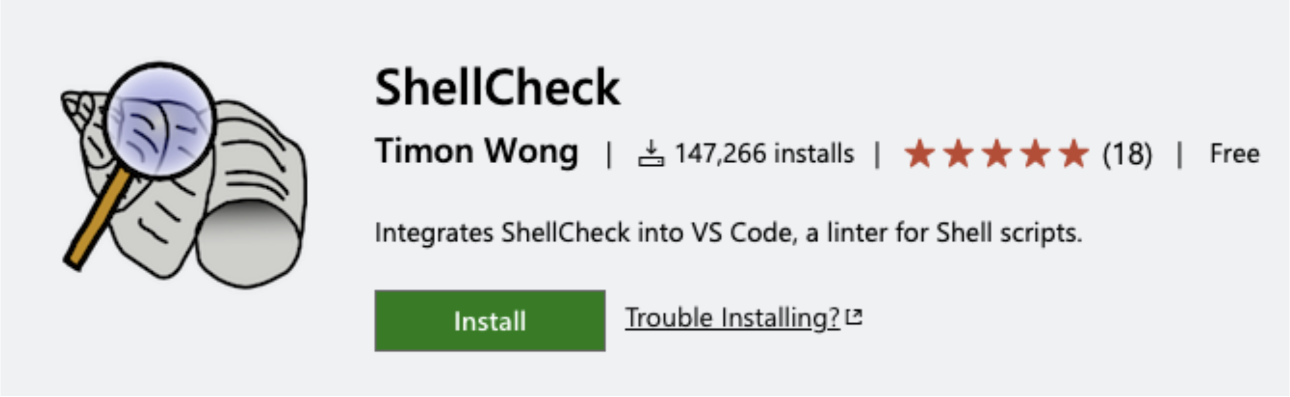
After learning the basics of Bash scripting, you can use these practices to make your Bash script look more professional. Here are 27 tips for Bash script beginners.

1. Clean Structure

Declare all of the global variables first and then declare all of the functions after the global variables. Use local variables in functions and write the main body after the functions. Use an explicit exit status code in your functions, the `if` statement, and at the end of the script.

2. Install ShellCheck on Your Editor

Or you can install it on VS Code.



ShellCheck

Timon Wong | 147,266 installs | ★★★★★ (18) | Free

Integrates ShellCheck into VS Code, a linter for Shell scripts.

VSCode ShellCheck extension

If you are interested in checking the ShellCheck error code list, check this gist.

3. Usage Function

If your script is using positional parameters, add a function explaining how and what users can use the parameters.

`$0` outputs the script name. In the above case, the `usage` function will be called when a user uses `h` or any letters except `f`, `d`, or `t`.

Try this online

The above script checks if the number of parameters is two. If it is not, it displays `usage` .

You can use Bash Heredoc:

4. Error Messages

Google's shell style guide recommends a function to print out messages along with other status information.

The guide suggests that all error messages should go to `STDERR` because this makes it easier to separate normal status from actual issues.

5. Function Comments

All functions should have comments mentioning description, global variables, arguments, outputs, and returned values, if applicable.

6. How To Concatenate String Variables

Photo by Alina Grubnyak on Unsplash

In a Bash script, you can concatenate strings in different ways.

Try this online

Please note that there is no space before and after an equal sign when you are assigning a value to a variable. The first one uses the `+=` sign to concatenate the second value to the first one. The second one uses a double quote with a space between variables. The last one is using a curly bracket since the variable `foo` is immediately followed by a character.

7. How To Set a Debug Mode

Try this online.

The output is a verbose trace of the script’s execution.

```
+ foo=Bash
+ '[' Bash == Bash ']'
+ echo Bash
Bash
```

`set -u` treats unset variables as an error when substituting.

`set -C` disallows existing regular files to be overwritten.

So you can use `set -Ceu` at the beginning of your script.

Find more about `set` using `help set` .

8. How To Slice Strings

You can use `cut -cstart-end` to slice a string.

Try this online

The first one slices the string from the first letter to the third. The second one slices from the second to the fourth. The last one slices the fifth letter.

A parameter expansion performs substring extraction, `${s:off-set-index:length}` :

```
#!/usr/bin/env bash
```

```
# abc
# bcd
# e
```

The offset is zero-based.

9. How To Make Sure Users Use a Correct Bash Version

The following script ensures the users use Bash version 4.0 or higher.

```
if ((BASH_VERSINFO[0] < 4)); then
    printf '%s\n' "Error: This requires Bash v4.0 or higher. You have
version $BASH_VERSION." 1>&2
    exit 2
fi
```

10. How To Transform a String Case

When you read user input, the input can be in lowercase or uppercase. For Bash lower than version 4, you can change it to uppercase using the `tr` command with `[:lower:]` and `[:upper:]` .

Try this online

Please note that we set the option `-x` for debugging.

Line 4: Read user input.

Line 5: Pipes, `|`, let you use the output of a command as the input of another command. We `cut` the first letter, then transform it from lowercase to uppercase. You can interchange the position of lowercase and uppercase to transform from uppercase to lowercase.

Try this online

Using these case modification operators, you can create functions.

Try this online

11. Ternary Operator-Like Statement

Bash doesn’t have a ternary operator, but you can do a similar thing for simple variable assignments.

We can write an `if` statement:

Using `&&` and `||` together, we can create a statement similar to a ternary statement:

```
[ $foo -ge $bar ] && baz="Smile!" || baz="Sleep!"
```

Photo by Fleur on Unsplash

`${#string}` returns the string length and similarly `${#array[@]}` returns the array length.

Try this online

13. How To Unset Variables

Unsetting variables ensures you are not using predefined variables.

At the beginning of the script:

```
#!/usr/bin/env bash
```

14. How To Set a Default Value

`${foo-$DEFAULT}` and `${foo=$DEFAULT}` evaluate as `$DEFAULT` if `foo` is not set.

`${foo:-$DEFAULT}` and `${foo:=$DEFAULT}` evaluates as `$DEFAULT` if the `foo` is not set or is empty.

Try this online.

15. How To Determine Your Bash Script Name

By adding all your Bash scripts to the `~/bin` directory and adding its path to your terminal config file, `~/.bashrc` or `~/.zshrc` , you can run them from any directory. But before you

```
$ type my_script
my_script not found
$ which which
which: shell built-in command
```

There is no script name with `my_script` , but `which` is already taken.

16. How To Make a Variable Constant

`readonly` makes variables and functions read-only.

Try this online.

Line 3: Set a read-only variable.
Line 5: Trying to overwrite the read-only variable. But this returns an error, “line 5: MAX: readonly variable”.

Output:

```
  10
```

You can output the `readonly` variables:

Try this online.

The following example is rather trivial, but it finds the user's OS and applies the correct command.

Output:

18. How To Determine Which HTTP Get Tool the System Has Installed

Your Bash script may use an HTTP `GET` tool. Different systems use different tools. It can be `curl`, `wget`, `http`, `fetch`, or something else.

`&>/dev/null` redirects both the standard output stream and the standard error stream to

19. How To Determine Which Python the System Has Installed

Similarly, if your script uses Python, you can find the system Python:

20. Parameter expansion: How to Find the Script Name and Directory

Try this online.

This Tech.io page explains how to use `${parameter%word}` , `${parameter%%word}` , `${parameter#word}` , `${parameter##word}` .

You can use `basename "$0"` to get the file name as well.

```
c=$0
echo "${c%/*}"
```

Outputs:

```
/Users/shinokada/bin
```

21. How To Make Status Messages

You may want to inform users what your script is doing.

If you want to output status messages, you should start it with the program name. Use `$(basename "$0")` to get the program name. The message should be written on the standard error by using `echo ... 1>&2`.

```
#!/usr/bin/env bash

progname=$(basename "$0")
...
echo "$progname: Running this and that" 1>&2
```

22. How To Set Locale

You can find your locale environment:

Not all programmers use the same locale. Programmers from France may use `fr_CH.UTF-8`; geeks from England may be using `en_GB.UTF-8`. To ensure that the users use the correct locale, you can use `LC_ALL` to set the locale:

```
LC_ALL="en_US.UTF-8"
```

`declare`.

You can use `+` instead of `-` to turn off the attribute.

Try this online.

We declared `var1` as an integer in line 3. If you try to change it to a string, you get an error (line5). But you can change it to another integer (line7).

Try this online.

Please note that when you use `declare` in a function, the variable becomes a local variable which you can't access outside of the function.

24. Store Exit Status in a Variable

Photo by Artem Beliaikin on Unsplash

Rather than using `$?` in `if` statements:

```
mkdir /tmp/tmp_dir
if [ $? = 0 ]; then
  # do something
fi
```

Store the exit status in a variable:

The following stores the exit status for checking if a file exists and is a directory.

```
test -d /tmp/tmp_dir
test_es=$?
if [[ ${test_es} -ne 0 ]]; then
    echo "No dir found. Exiting script!" >&2
    exit 1
fi
```

modifying a directory or a file, you need to bring it back to the original state. The command `trap` is for this situation.

When a user uses `Ctrl-c` , it generates a signal `SIGINT` . When a user kills a process, it generates a signal `SIGTERM` . You can list all the signals using:

```
# Linux, from terminal
$ trap --list
# macOS, from a script
trap --list
```

You can find the standard signals on man7.org.

The `trap` command has the form of `trap "do something" signal_to_trap` . If your cleanup code is long, you can create a function.

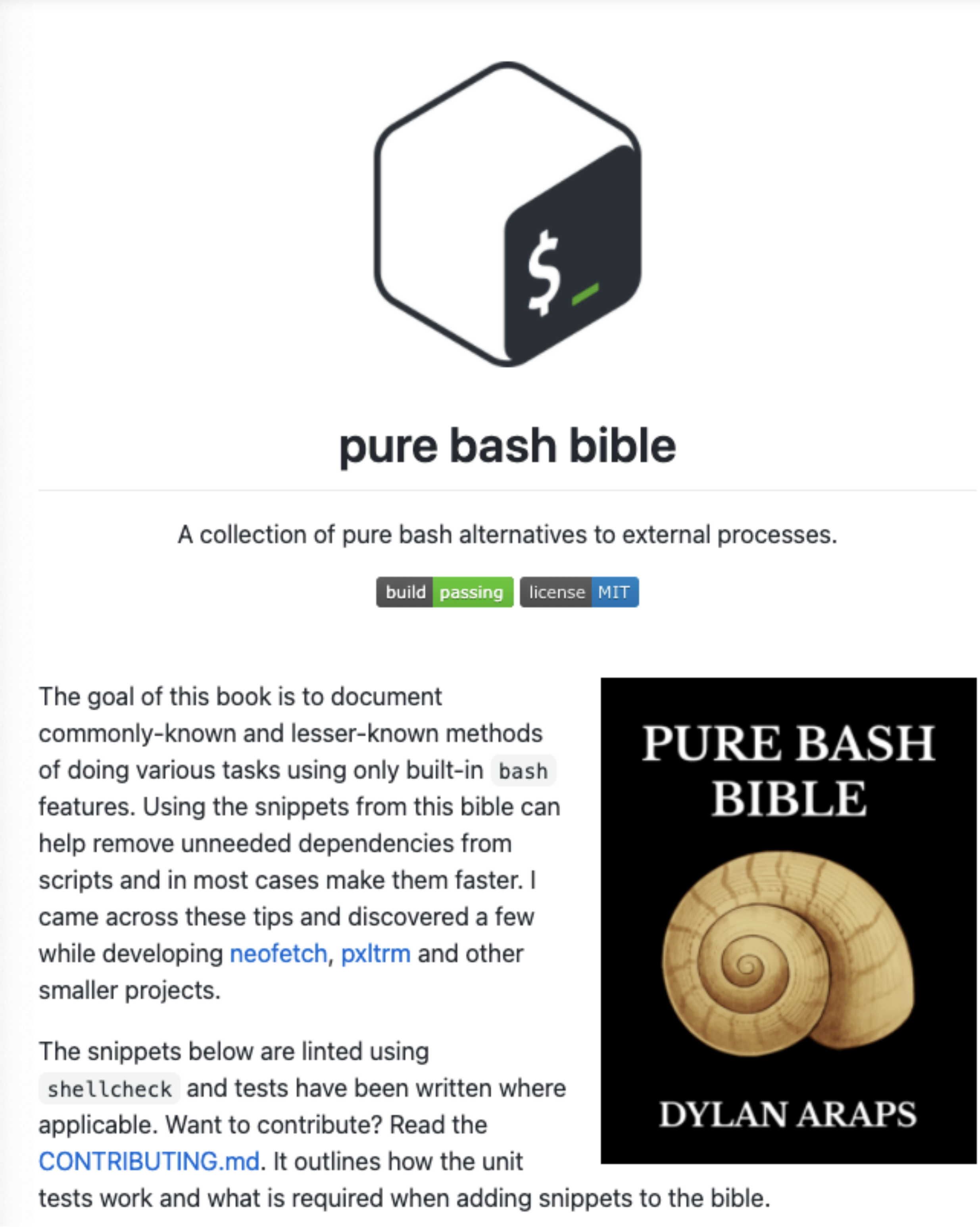
Let's trap the `Ctrl-c` signal:

```
tempfile=/tmp/myfile
cleanup(){
    rm -f $tempfile
}
trap cleanup SIGINT
```

Read more about the trap command.

26. Don't Reinvent the Wheel

If your goal is learning, then you can reinvent the wheel, but if not, use snippets from pure-bash-bible. The pure-bash-bible is a great learning place as well. There are many functions you can use in your scripts. The scripts include strings, arrays, loops, file handling, file paths, and more.



pure bash bible

A collection of pure bash alternatives to external processes.

The goal of this book is to document commonly-known and lesser-known methods of doing various tasks using only built-in `bash` features. Using the snippets from this bible can help remove unneeded dependencies from scripts and in most cases make them faster. I came across these tips and discovered a few while developing neofetch, pxltrm and other smaller projects.

The snippets below are linted using `shellcheck` and tests have been written where applicable. Want to contribute? Read the CONTRIBUTING.md. It outlines how the unit tests work and what is required when adding snippets to the bible.

Image from pure bash bible

27. Subshell and Exit Status

The first one returns:

```
/Users/myname/bin/ex5: line 34: no_func1: command not found
there is nothing
1
```

The difference is the parentheses and braces. Parentheses cause the commands to be run in a subshell, and braces cause the commands to be grouped together but not in a subshell.

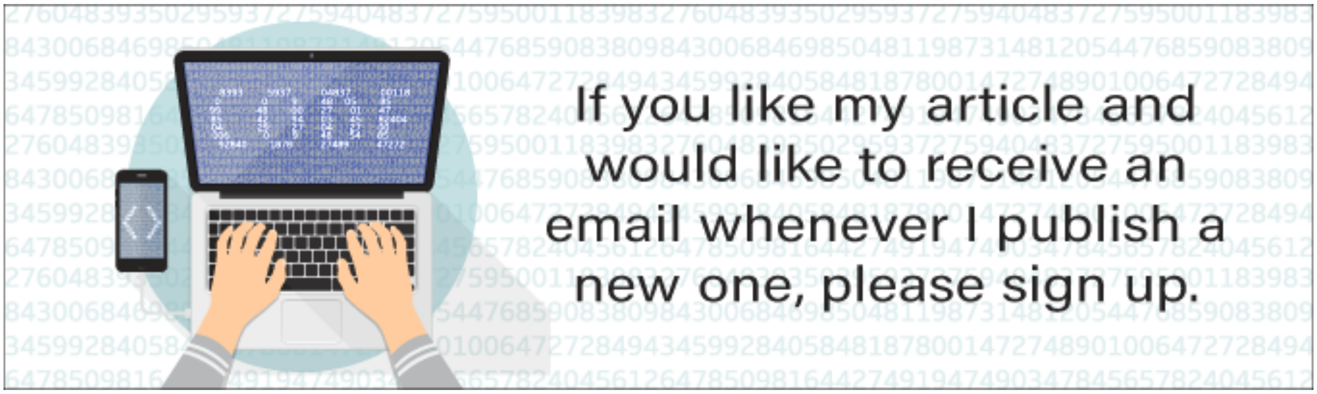
Since braces don't create a subshell, `exit` exits the main shell process, so it never reaches the point where it would run `echo $?`.

Conclusion

These are 27 Bash scripting tips you can use in your next Bash script project. Many of them are easy to adopt and they make your Bash script look more professional.

Happy coding!

Get full access to every story on Medium by becoming a member.


If you like my article and would like to receive an email whenever I publish a new one, please sign up.

https://blog.codewithshin.com/subscribe

References

- https://tldp.org/LDP/abs/html/special-chars.html

The Ultimate Programmer’s Guide to Bash Scripting

A deep dive into Bash scripts to help you automate tasks

medium.com

13 Fantastic Learning Tools and Resources for Bash Scripting

Alternative Bash scripting tools you will use daily

medium.com

Comparing SH, BASH, KSH, and ZSH Speed

The winner goes to ... Drumroll

towardsdatascience.com

Thanks to Emile Okada and Anupam Chugh