# Writing the Bible

<!-- vim-markdown-toc GFM -->

\* [Adding Code to the Bible.](#adding-code-to-the-bible)

\* [Special meanings for code blocks.](#special-meanings-for-code-blocks)

\* [Writing tests](#writing-tests)

\* [Running tests](#running-the-tests)

<!-- vim-markdown-toc -->

## Adding Code to the Bible.

- The code must use only `bash` built-ins.

- A fallback to an external program is allowed if the code doesn't

always work.

- Example Fallback: `${HOSTNAME:-$(hostname)}`

- If possible, wrap the code in a function.

- This allows tests to be written.

- It also allows `shellcheck` to properly lint it.

- An added bonus is showing a working use-case.

- Write some examples.

- Show some input and the modified output.

## Special meanings for code blocks.

Use `sh` for functions that should be linted and unit tested.

```sh

# Shellcheck will lint this and the test script will source this.

func() {

# Usage: func "arg"

:

}

```

Use `shell` for code that should be ignored.

```shell

# Shorter file creation syntax.

:>file

```

## Writing tests

The test file is viewable here: https://github.com/dylanaraps/pure-bash-bible/blob/master/test.sh

Example test:

```sh

test\_upper() {

result="$(upper "HeLlO")"

assert\_equals "$result" "HELLO"

}

```

Steps:

1. Write the test.

- Naming is `test\_func\_name`

- Store the function output in a variable (`$result` or `${result[@]}`).

- Use `assert\_equals` to test equality between the variable and the

expected output.

2. The test script will automatically execute it. :+1:

## Running the tests

Running `test.sh` also runs `shellcheck` on the code.

```sh

cd pure-bash-bible

./test.sh

```