Bash scripting tutorial: A coding style guideline

Nov. 4, 2016 • 3 min read • original

Introduction

Using Bash scripts a Linux user can achieve many things, scripting is a good way to pipe programs together and automate certain tasks. **The main focus of this guideline is to assemble Bash scripts to address readability and script maintenance issues, not portability in particular (e.g: POSIX).** For this post it is assumed that you have some basic understanding of any UNIX-like shell, this tutorial also contains examples in order to put the guideline material into practice. Bash version used for this guideline: 4.3.11.

Shebang

Classic

```
#!/bin/bash
```

Env-style

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

- The classic shebang bets everything in one path
- The env shebang looks for the binary in more places, has more chances of running *something*

Comments

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

```
# This is a comment
```

- Write necessary comments (e.g tricky code, hacks, ..), increases readability
- Use guiding/instructive comments for long procedures and complex routines
- Add script information using comments

Example 1: Script header

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

# Name: script-example
# Brief: Guildeline example script
# Author: ___
```

Constants

```
# Example constant
readonly EXAMPLE_CONSTANT="Constant string"
```

- Use descriptive and unique name, if use all uppercase watch not to duplicate system environment variables
- Use underscore and partial abbreviations if needed when name starts to get big
- Place them at the top of the script
- Use the *readonly* keyword, protects against overwriting/reassignment

Variables

```
# CamelCase example variable
ExampleVar="Using CamelCase"

# Underscore-like example variable
example_var="Using underscore"
```

- Use descriptive name and partial abbreviations if needed
- CamelCase or underscore format
- Using the *local* keyword inside functions prevents problems with global variables

```
function some_function () {
    local some_var=2
    return 0
}
```

Example 2: Constants & variables

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

# Name: script-example
# Brief: Guildeline example script
# Author: Blue Penguin

# Constants
readonly SCRIPT_START="Starting script"
readonly SCRIPT_END="Script end. Exiting"

# First assignment
msg_string=$SCRIPT_START
```

```
echo $msg_string

# Script routine
echo "Working .."
# Second assignment
msg_string=$SCRIPT_END

echo $msg_string

exit 0
```

Conditional statements

```
# if-else statement example
if condition ; then
    operation
else
    operation
fi
```

```
# case statement example
case expression in
    case1)
        operation
    ;;
    case2)
        operation
    ;;
esac
```

- Indent properly to keep readability, specially when conditional statements start to get big

Commands & exit codes

```
#!/usr/bin/env bash
# Name: script-example
# Brief: Guildeline example script
# Author: Blue Penguin

readonly DIR=foo

ls $DIR

# Was ls capable of listing foo contents?
# Does foo actually exist?
```

- Use exit codes to control the script/program execution and to avoid unexpected behavior

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

# Name: script-example
# Brief: Guildeline example script
# Author: Blue Penguin

readonly DIR=foo
```

```
ls $DIR
ls_code=$?

if [ $ls_code != 0 ] ; then
    echo "List $DIR directory operation .. Error"
    exit 1
else
    echo "List $DIR directory operation .. Ok"
    # More ops ..
fi

exit 0
```

- Study the script commands and their respective exit codes to consider possible scenarios when the program is running
- 0 stands for *ok*, different than that (1, 2, ..) stands for *not ok*
- Correct script exiting helps other programs to use it better

Functions

Implicit declaration

```
some_function () {
    return 0
}
```

Explicit declaration

```
function some_function () {
    return 0
}
```

- The explicit alternative gives more readability, the implicit one is POSIX standard so is more portable
- Follow the same convention from the exit codes, 0 – ok, not 0 – not ok. Will make the program more consistent
- Check risky variables before doing the actual procedure, work safe
- Check error-prone commands before jumping to the next operation
- **return** something, this lets other program pieces to know if the function actually did what it was supposed to
- Use comments on top of the function to provide information: name, brief, arguments, return codes, notes, ..

Example 3: List function

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

# Name: script-example
# Brief: Guildeline example script
# Author: Blue Penguin

readonly DIR=foo

function list_directory () {
    if [ ! -z $1 ] ; then
        local dir=$1
```

```
        ls $dir
        if [ $? != 0 ] ; then
            echo "Error listing content"
            return 1
        fi
    else
        echo "Invalid argument"
        return 1
    fi
    return 0
}

# Program start
if ! list_directory "$DIR" ; then
    exit 1
fi

exit 0
```

Sourcing

Explicit

```
source some_file
```

Minimal

```
. some_file
```

- The source command has two formats: the dot '.' and the*source* word itself, the source word provides more readability, the minimal is POSIX standardso gives more portability

```
# Filename: user.defs

readonly SCRIPT_USERNAME="Someone"
#!/usr/bin/env bash

# Name: script-example
# Brief: Guildeline example script
# Author: Blue Penguin

if [ -f user.defs ] ; then
    source user.defs
    echo "This script was run by: $SCRIPT_USERNAME"
fi

exit 0
```

- Use sourcing for: loading common definitions, separating program code from user code
- Source definitions, e.g: variables, functions
- Be careful with overwrites!
- Don't source programs/implementations (for that use './'), you can end up running something the main program will not be in control of

Example 4: Sourcing

```
# Filename: log.defs

function log_error () {
    local user_log=$1
    echo -e "\e[91mERROR\e[0m:$LOG_SCRIPTNAME $user_log"
    return 0
}
```

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

# Name: script-example
# Brief: Guildeline example script
# Author: Blue Penguin

LOG_SCRIPTNAME="script-example:"

source log.defs

if [ 2 = 1 ] ; then
    echo "Really ? "
else
    log_error "2 doesn't equals 1 .."
    exit 1
fi

exit 0
```

Script naming

- Use lowercase, abbreviations if needed
- Dash(es) for longer naming. You can omit the '.bash' suffix
- Examples: scriptexample, scriptex, script-example

Conclusion

Following this document will help you to deal with Bash scripts/programs, from simple ones to big ones (like multiple scripts that interact with each other). The key to have readable, modular and maintainable Bash scripts is consistency, it is true that exercising consistency will bring up clues and hints for better script writing.

Original URL:
https://bluepenguinlist.com/2016/11/04/bash-scripting-tutorial/?fromTwitterID=nixCraft