Creating a Bash script template

In the second article in this series, create a fairly simple template that you can use as a starting point for other Bash programs, then test it.

By David Both (Correspondent)

December 10, 2019 | 0 Comments | **11 min read** 131 readers like this

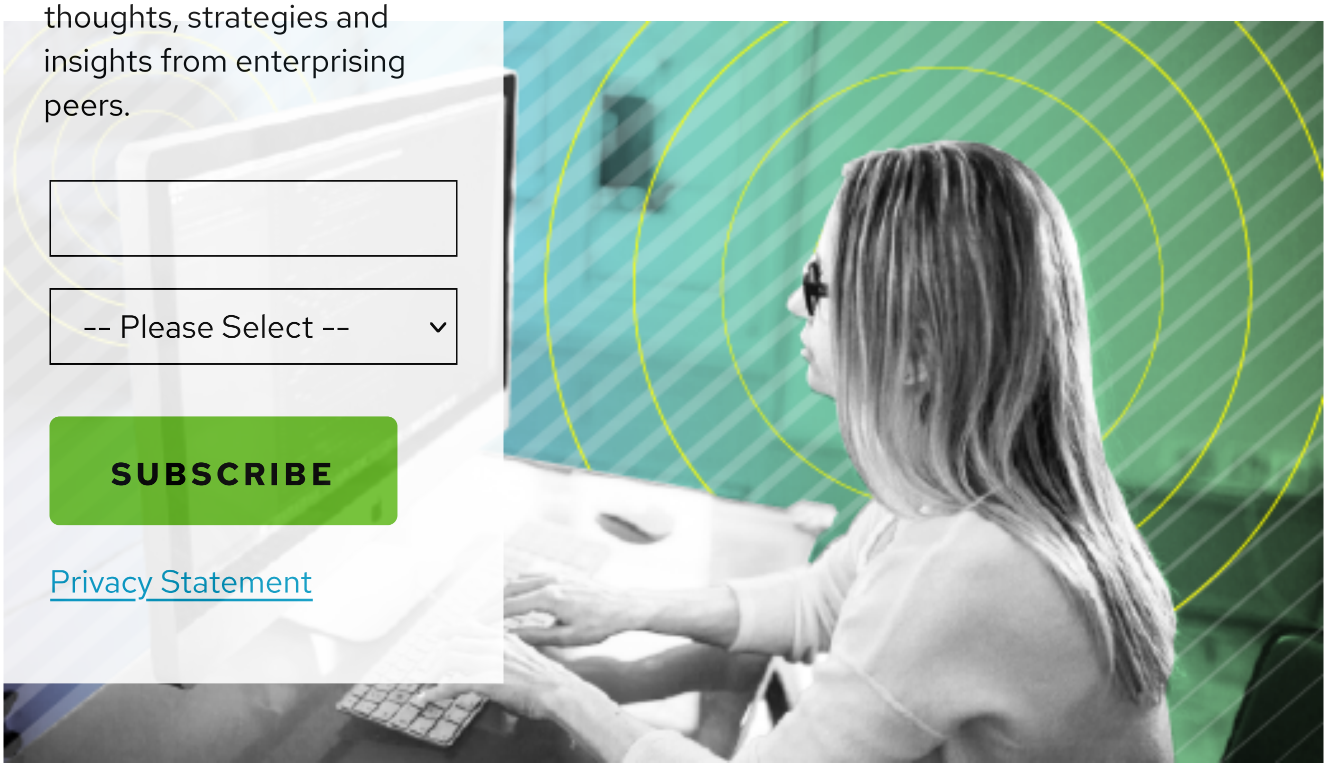


Image by: *Opensource.com*

In the first article in this series, you created a very small, one-line Bash script and explored the reasons for creating shell scripts and why they are the most efficient option for the system administrator, rather than compiled programs.

In this second article, you will begin creating a Bash script template that can be used as a starting point for other Bash scripts. The template will ultimately contain a Help facility, a licensing statement, a number of simple functions, and some logic to deal with those options and others that might be needed for the scripts that will be based on this template.

Why create a template?

More on sysadmins

Enable Sysadmin blog

The Automated Enterprise: a guide to managing IT with automation

eBook: Ansible Automation for SysAdmins

Tales from the field: A system administrator's guide to IT automation

eBook: A guide to Kubernetes for SREs and sysadmins

Latest sysadmin articles

Like automation in general, the idea behind creating a template is to be the "lazy sysadmin." A template contains the basic components that you want in all of your scripts. It saves time compared to adding those components to every new script and makes it easy to start a new script.

Although it can be tempting to just throw a few command-line Bash statements together into a file and make it executable, that can be counterproductive in the long run. A well-written and well-commented Bash program with a Help facility and the capability to accept command-line

sysadmins who maintain the program,

and maintain.

The requirements

You should always create a set of requirements for every project you do. This includes scripts, even if it is a simple list with only two or three items on it. I have been involved in many projects that either failed completely or failed to meet the customer's needs, usually due to the lack of a requirements statement or a poorly written one.

The requirements for this Bash template are pretty simple:

1. Create a template that can be used as the starting point for future Bash programming projects.
2. The template should follow standard Bash programming practices.
3. It must include:
    - A heading section that can be used to describe the function of the program and a changelog
    - A licensing statement
    - A section for functions
    - A Help function
    - A function to test whether the program user is root
    - A method for evaluating command-line options

The basic structure

A basic Bash script has three sections. Bash has no way to delineate sections, but the boundaries between the sections are implicit.

- All scripts must begin with the shebang (**#!**), and this must be the first line in any Bash program.
- The functions section must begir the program. As part of my need

before each function with a short description of what it is intended to do. I also include comments inside the functions to elaborate further. Short, simple programs may not need functions.

The main part of the program comes after the function section. This can be a single Bash statement or thousands of lines of code. One of my programs has a little over 200 lines of code, not counting comments. That same program has more than 600 comment lines.

That is all there is—just three sections in the structure of any Bash program.

Leading comments

I always add more than this for various reasons. First, I add a couple of sections of comments immediately after the shebang. These comment sections are optional, but I find them very helpful.

The first comment section is the program name and description and a change history. I learned this format while working at IBM, and it provides a method of documenting the long-term development of the program and any fixes applied to it. This is an important start in documenting your program.

The second comment section is a copyright and license statement. I use GPLv2, and this seems to be a standard statement for programs licensed under GPLv2. If you use a different open source license, that is fine, but I suggest adding an explicit statement to the code to eliminate any possible confusion about licensing. Scott Peterson's article *The source code is the license* helps explain the reasoning behind this.

So now the script looks like this:

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

#
```

```
                           #
# Use this template as the beginning of a new
program. Place a short          #
# description of the script here.
                           #
#
                           #
# Change History
                          #
# 11/11/2019  David Both    Original code. This is a
template for creating     #
#                            new Bash shell scripts.
                           #
#                            Add new history entries
as needed.                 #
#
                           #
#
                           #
#####################################################
#####################################################
#####################################################
#
                           #
#  Copyright (C) 2007, 2019 David Both
                          #
#  LinuxGeek46@both.org
                           #
#
                           #
#  This program is free software; you can
redistribute it and/or modify        #
#  it under the terms of the GNU General Public
License as published by        #
#  the Free Software Foundation; either version 2 of
the License, or           #
#  (at your option) any later version.
                          #
#
#  This program is dist
```

```
will be useful,             #
#  but WITHOUT ANY WARRANTY; without even the
implied warranty of              #
#  MERCHANTABILITY or FITNESS FOR A PARTICULAR
PURPOSE.  See the                #
#  GNU General Public License for more details.
                           #
#
                           #
#  You should have received a copy of the GNU
General Public License           #
#  along with this program; if not, write to the
Free Software                #
#  Foundation, Inc., 59 Temple Place, Suite 330,
Boston, MA  02111-1307  USA    #
#
                           #
#####################################################
#####################################################
#####################################################

echo "hello world!"
```

Run the revised program to verify that it still works as expected.

About testing

Now is a good time to talk about testing.

> "*There is always one more bug.*"
>
> – Lubarsky's Law of Cybernetic Entomology

Lubarsky—whoever that might be—is correct. You can never find all the bugs in your code. For every bug I find, there always seems to be another that crops up, usually at a very inopportune time.

Testing is not just about programs. It is

—whether caused by hardware, software, or the seemingly endless ways users can find to break things—that are supposed to be resolved actually are. Just as important, testing is also about ensuring that the code is easy to use and the interface makes sense to the user.

Following a well-defined process when writing and testing shell scripts can contribute to consistent and high-quality results. My process is simple:

1. Create a simple test plan.
2. Start testing right at the beginning of development.
3. Perform a final test when the code is complete.
4. Move to production and test more.

The test plan

There are lots of different formats for test plans. I have worked with the full range—from having it all in my head; to a few notes jotted down on a sheet of paper; and all the way to a complex set of forms that require a full description of each test, which functional code it would test, what the test would accomplish, and what the inputs and results should be.

Speaking as a sysadmin who has been (but is not now) a tester, I try to take the middle ground. Having at least a short written test plan will ensure consistency from one test run to the next. How much detail you need depends upon how formal your development and test functions are.

The sample test plan documents I found using Google were complex and intended for large organizations with very formal development and test processes. Although those test plans would be good for people with "test" in their job title, they do not apply well to sysadmins' more chaotic and time-dependent working conditions. As in most other aspects of the job, sysadmins need to be creative. So here is a short list of things to consider including in your test plan. Modify it to suit your needs:

The name and a short descriptior

- A description of the software features to be tested
- The starting conditions for each test
- The functions to follow for each test
- A description of the desired outcome for each test
- Specific tests designed to test for negative outcomes
- Tests for how the program handles unexpected inputs
- A clear description of what constitutes pass or fail for each test
- Fuzzy testing, which is described below

This list should give you some ideas for creating your test plans. Most sysadmins should keep it simple and fairly informal.

Test early—test often

I always start testing my shell scripts as soon as I complete the first portion that is executable. This is true whether I am writing a short command-line program or a script that is an executable file.

I usually start creating new programs with the shell script template. I write the code for the Help function and test it. This is usually a trivial part of the process, but it helps me get started and ensures that things in the template are working properly at the outset. At this point, it is easy to fix problems with the template portions of the script or to modify it to meet needs that the standard template does not.

Once the template and Help function are working, I move on to creating the body of the program by adding comments to document the programming steps required to meet the program specifications. Now I start adding code to meet the requirements stated in each comment. This code will probably require adding variables that are initialized in that section of the template—which is now becoming a shell script.

This is where testing is more than just entering data and verifying the results. It takes a bit of extra work. Sometimes I intermediate result of the code I just wrote and verify that. For more complex scripts, I add a **-t** option for "test mode." In this case, the internal test code executes only when the **-t** option is entered on the command line.

Final testing

After the code is complete, I go back to do a complete test of all the features and functions using known inputs to produce specific outputs. I also test some random inputs to see if the program can handle unexpected input.

Final testing is intended to verify that the program is functioning essentially as intended. A large part of the final test is to ensure that functions that worked earlier in the development cycle have not been broken by code that was added or changed later in the cycle.

If you have been testing the script as you add new code to it, you may think there should not be any surprises during the final test. Wrong! There are always surprises during final testing. Always. Expect those surprises, and be ready to spend time fixing them. If there were never any bugs discovered during final testing, there would be no point in doing a final test, would there?

Testing in production

Huh—what?

> "Not until a program has been in production for at least six months will the most harmful error be discovered."
>
> — Troutman's Programming Postulates

Yes, testing in production is now considered normal and desirable. Having been a tester myself, this seems reasonable. "But wait! That's dangerous," you say. My experience is that it is no more dangerous than extensive and rigorous testing in a dedicated test environment. In some cases, there is no choice because there is no test environment—only production.

Sysadmins are no strangers to the need to test new or revised scripts in production. Anytime a script is moved into production, that becomes the ultimate test. The production environment constitutes the most critical part of that test. Nothing that testers can dream up in a test environment can fully replicate the true production environment.

The allegedly new practice of testing in production is just the recognition of what sysadmins have known all along. The best test is production—so long as it is not the only test.

Fuzzy testing

This is another of those buzzwords that initially caused me to roll my eyes. Its essential meaning is simple: have someone bang on the keys until something happens, and see how well the program handles it. But there really is more to it than that.

Fuzzy testing is a bit like the time my son broke the code for a game in less than a minute with random input. That pretty much ended my attempts to write games for him.

Most test plans utilize very specific input that generates a specific result or output. Regardless of whether the test defines a positive or negative outcome as a success, it is still controlled, and the inputs and results are specified and expected, such as a specific error message for a specific failure mode.

Fuzzy testing is about dealing with randomness in all aspects of the test, such as starting conditions, very random and unexpected input, random combinations of options selected, low memory, high levels of CPU contending with other programs, multiple instances of the program under test, and any other random conditions that you can think of to apply to the tests.

I try to do some fuzzy testing from the beginning. If the Bash script cannot deal with significant randomness in its very early stages, then it is unlikely to get better as you add more code. This is a good t

while the code is relatively simple. A bi in locating problems before they get masked by even more code.

After the code is completed, I like to do some more extensive fuzzy testing. Always do some fuzzy testing. I have certainly been surprised by some of the results. It is easy to test for the expected things, but users do not usually do the expected things with a script.

Previews of coming attractions

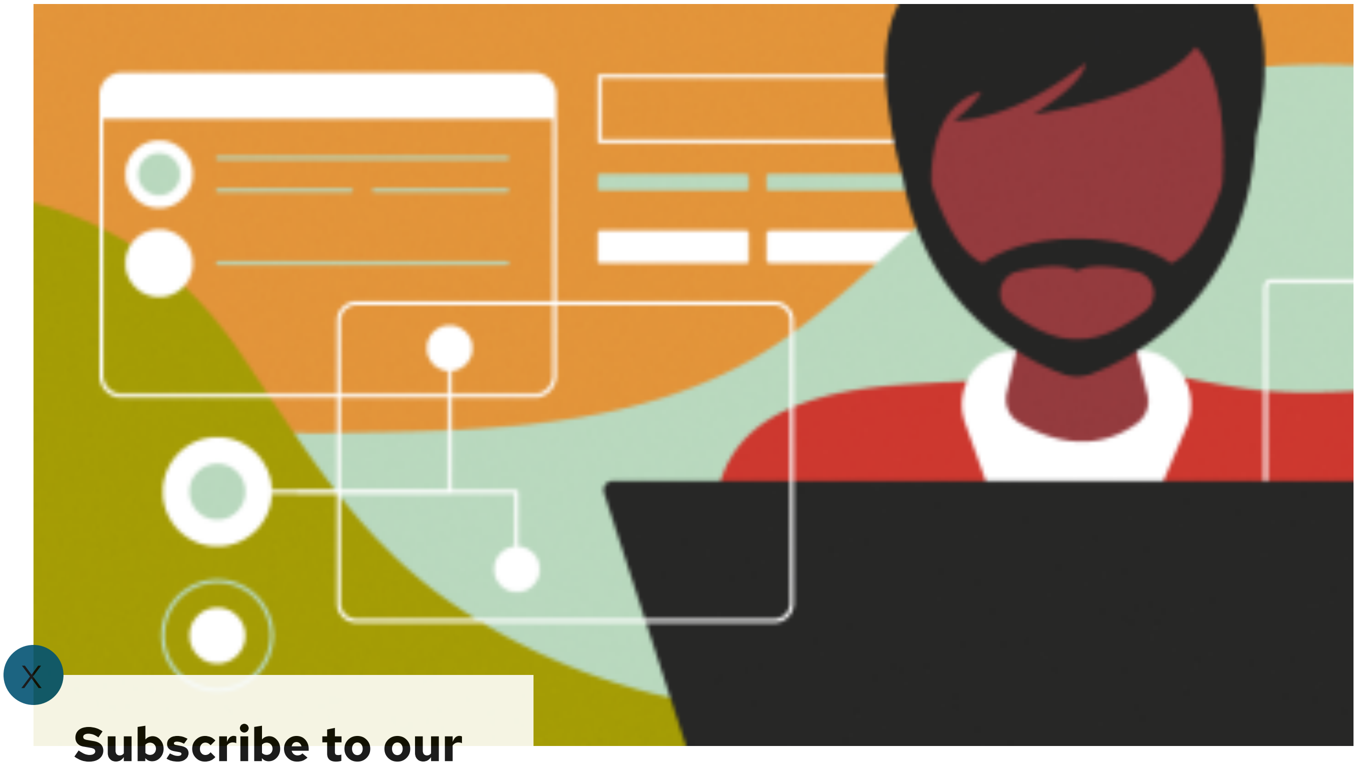
This article accomplished a little in the way of creating a template, but it mostly talked about testing. This is because testing is a critical part of creating any kind of program. In the next article in this series, you will add a basic Help function along with some code to detect and act on options, such as **-h**, to your Bash script template.

Resources

- How to program with Bash: Syntax and tools
- How to program with Bash: Logical operators and shell expansions
- How to program with Bash: Loops

This series of articles is partially based on Volume 2, Chapter 10 of David Both's three-part Linux self-study course, Using and Administering Linux—Zero to SysAdmin.

What to read next



Introduction to automation with Bash scripts

In the first article in this four-part series, learn how to create a simple shell script and why they are the best way to automate tasks.

David Both (Correspondent)

X

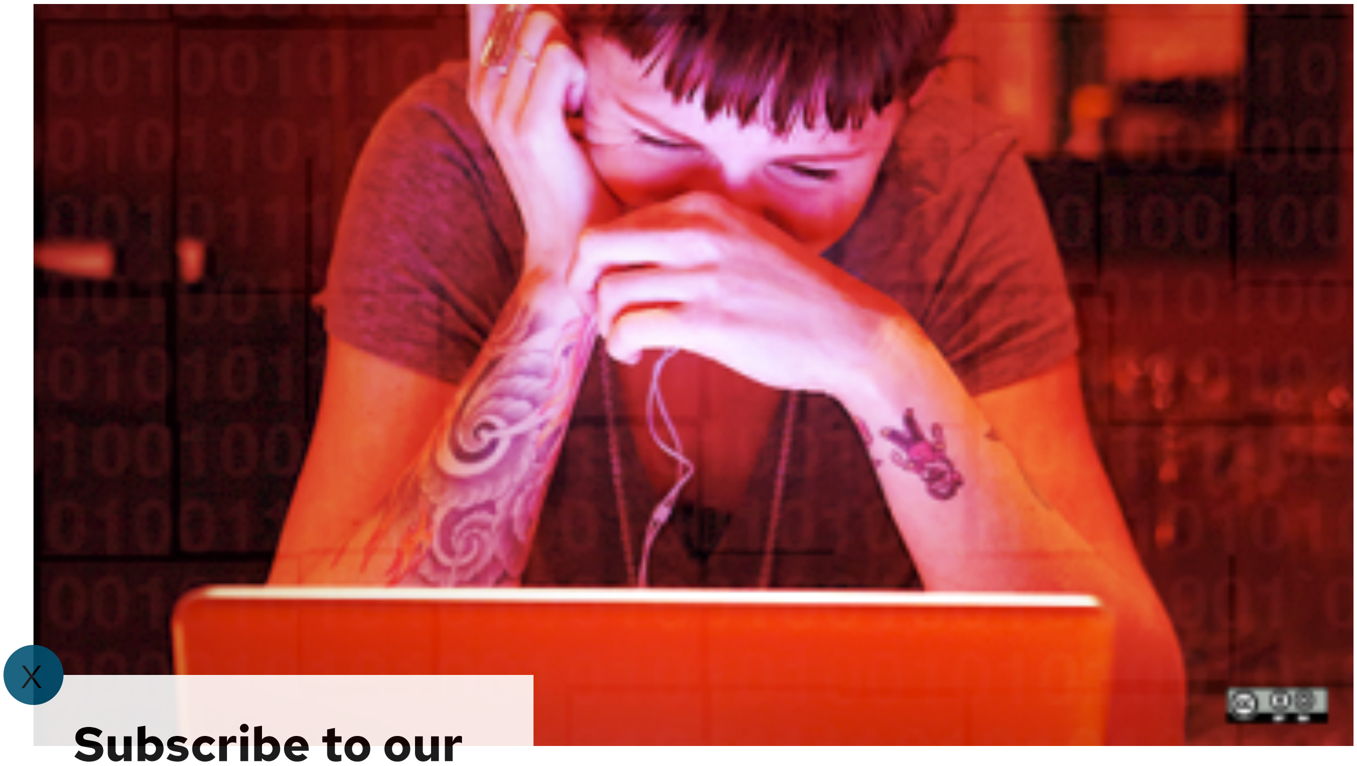
Subscribe to our newsletter.

Stay on top of the latest thoughts, strategies and insights from enterprising peers.

Bash cheat sheet: Key combos and special syntax

Download our new cheat sheet for Bash commands and shortcuts you need to talk to your computer.

Seth Kenlon (Team, Red Hat)



How to program with Bash: Logical operators and shell expansions

Learn about logical operators and shell expansions, in the second article in this three-part series on programming with Bash.

David Both (Correspondent)

Tags: BASH

David Both

David Both is an Open Source Software and GNU/Linux advocate, trainer, writer, and speaker who lives in Raleigh North Carolina. He is a strong proponent of and evangelist fo

More about me

Comments are closed.

These comments are closed, however you can Register or Login to post a comment on another article.

Related Content

How I configure Vim as my default editor on Linux

How I dynamically generate Jekyll config files

Automate image processing with this Bash script

Subscribe to our weekly newsletter

Privacy Statement

ABOUT THIS SITE

The opinions expressed on this website are those of each author, not of the author's employer or of Red Hat.

Opensource.com aspires to publish all content under a **Creative Commons license** but may not be able to do so in all cases. You are responsible for ensuring that you have the necessary permission to reuse any work on this site. Red Hat and the Red Hat logo are trademarks of Red Hat, Inc., registered in the United States and other countries.

A note on advertising: Opensource.com does not sell advertising on the site or in any of its newsletters.

CONTACT

- Follow us @opensource.com on Twitter
- Like Opensource.com on Facebook
- Watch us at Opensource.com
- Follow us on Mastodon

RSS Feed

Privacy Policy

Terms of use

Contact

X

Subscribe to our newsletter.

Stay on top of the latest thoughts, strategies and insights from enterprising peers.