Bash Style Guide

Jump to bottom

Jared Casner edited this page on Oct 9, 2015 · 3 revisions

Bash coding guide

Going forward, bash scripts should be used for only very limited tasks, cron jobs or init scripts. **Anything else with complex logic should be in Python.**

When to use Bash

- Category one: Small scripts that set a few environment variables (typically PYTHONPATH, PATH, AWS credentials), perhaps set up a working directory, then call a Python script to do real work. If the python script can also do any necessary clean up after itself, the last line of the bash script should be `exec /path/to/foo.py $@` so that the shell script wrapper inherits the python script's exit code.
- Category two: Scripts with very minimal logic and are essentially just "run these half dozen commands in order."
- Category three: Cron jobs and init scripts, since they should have their environment explicitly set instead of trusting that they will receive any custom setup. They should be in category one.

When to stop using Bash

- If you go to edit a script and there's already a TODO saying to rewrite it in Python, rewrite it in Python no matter how trivial the changes you want to make seem.

- If your script is longer than 50-75 lines, you should be using functions. Individual functions should rarely be more than 20-40 lines. If you have more than a half dozen or so functions, it is probably time to move to Python.
- Functions in the script are complex enough to need local variables.
- You're trying to create associative arrays (Bash's equivalent to dicts in Python)
- You need data structures.
- It takes more than 15 minutes Googling to figure out how to make Bash do what you want.

Before you write or edit any bash

- Read David Pashley's Writing Robust Shell Scripts article.
- Read Google's Shell Programming Style Guide.
- Read Bash Pitfalls.
- Install shellcheck on your laptop - Shellcheck is essentially pylint for bash. Always run it before creating a pull request for your bash scripts. There are instructions on github for creating a binary on your machine so you don't have to use the web form at http://www.shellcheck.net/. On OS X, shellcheck can be installed with `brew install shellcheck` .
- Read the Community Bash Style Guide

In addition to the above articles, keep the following rules in mind:

- Scripts **must** always have a working shebang line.
- Scripts should always be able to be called with `scriptname --help` so we can see useful usage instructions without having to dig through the code.
- Shebang lines must (with the exception of `/bin/bash`) always be `#!/usr/bin/env interpretername` . This allows you to change your $PATH to get a different version of the interpreter without having to edit every file you're working on. This is because `/usr/bin/env interpreter` is independent of the Linux flavor or even whether the script is running on Linux or OS X.
- `/usr/bin/env bash` is preferable to `/bin/sh` or `/bin/bash` . This allows us to easily test scripts against new versions of bash.
- Never use `/bin/sh` , especially when it is a symlink to `/bin/bash` because the syntax permitted is different when bash is invoked as `/bin/sh` . Also, different distributions use different shells for `/bin/sh` which each have different quirks. bash is everywhere, use it instead.
- **It is never ok to change a system's default version of an interpreter**. For example, if a script needs Python 2.7, use /usr/bin/env python2.7, never change /usr/bin/python to

point at python2.7 if python2.6 is the OS default.

- Always use `set -o optionFullName` format - `set -o errexit` instead of `set -x` for example.
- Always use `set -o pipefail` in your scripts to return a fail if any of the commands in a pipeline fail
- Always use `set -o nounset` to cause errors if you refer to a variable name before you assign it a value
- Scripts must have their X bit set so they can be called directly from the command line. **No one should ever have to do `interpreterName script`.**
- **Always run shellcheck to lint any scripts you write/edit.** If you edit a script, fix the problems shellcheck flags even if they were already in the file when you started so we can chip away at our technical debt.
- **Never use backticks to invoke other scripts**. Always use `$(scriptname)`. Fortunately, shellcheck will complain about this.
- Keep complexity in functions. If you're doing a loop over a bunch of items, put that loop into a function and then call the function. This makes the script a lot easier to read, and a lot easier to debug.
- xtrace is spammy. If you must use `set -o xtrace`, surround it in a if statement so it is only used if $VERBOSE_DEBUG is set in the environment so we don't default to spamming the console.
- If you are not explicitly checking exit codes and coping gracefully with errors (retrying, etc) `set -o errexit` so the script fails when the first command in it fails is strongly recommended.
- Avoid hardcoding paths within the script as much as possible. Better to set `NUMENTA='/opt/numenta'` and refer to `${NUMENTA}/htm.it/bin` and `${NUMENTA}/htm.it/conf/foo.conf`. Then when testing a new version, we only have to change one line to use a different directory tree and not collide with existing scripts usage of that tree. This bit us on rpmbuild during the switch from Python 2.6 to 2.7 and I don't want to deal with that hassle again.
- Commands referenced inside scripts should use long argument names when available. For example, `pip --build /path/to/tempdir` is clearer to read than `pip -b /path/to/tempdir` if you're not familiar with all of pip's options.
- It is never, ever, ok to call a script with `interpreter /path/to/script` from within any of our scripts or jenkins pipelines. It is ok when you're testing from the command line if you want to test something with a specific version of an interpreter, but **any script we use in automation (whether by Salt, packer or shell scripts), or that we install on our instances, must have a working shebang line**. If you need to change the interpreter version used, change your `$PATH`.

- **Command line tools must never have a name ending in .sh, .py or .rb**! If we rewrite a script in another language (to move from a simple bash script to a more functional python script for example), we don't want to have to track down every reference to that script and rename it, or worse, have a python script with a .sh extension.
- If a script is making temporary files/directories, there must be a cleanup function and we should trap (at least) SIGINT and SIGTERM to call it so we don't leave kruft around burning disk space if the script is killed.
- Always use `mktemp` when you need a temporary file or directory instead of rolling your own way of making a temporary file.
- It is always ok (and preferred) to spin up a temporary box for testing instead of trying to test directly on rpmbuild. We want rpmbuild to always be able to run our production pipeline helper scripts.
- Use lock directories instead of lock files. `mkdir` is atomic and two simultaneous attempts to create the same directory will have only one succeed.

Formatting:

Unless explicitly overruled below, follow Google's Shell Programming Style Guide.

Indentation

We do not use tabs for indentation in bash scripts at Numenta. As with our Python code, we indent two spaces for each level of indentation.

No:

```
if [ ${ABC} = "def" ]; then
blah foo bar
fi

for x in a b c
do
blah $x
done
```

Yes:

```
if [ "${ABC}" = "def" ]; then
  blah foo bar
fi
```

```
for x in a b c
do
  blah $x
done
```

Using whitespace for clarification

As in our Python style guide, use no more than one blank line within a function to offset logical chunks.

Line length and Long Strings

The maximum line length is 80 characters, just like in Python. If you need a string that is longer than 80 characters, either use embedded newlines or here documents. If the string has to be longer than 80 characters and can't sensibly be split, ok, but 80 character width is strongly preferred.

```
# DO use 'here document's
cat <<END;
I am an exceptionally long
string.
END

# Embedded newlines are ok too
long_string="I am an exceptionally
  long string."
```

Variables

Bash scripts will typically have a section at the beginning where you set up variables that will be used later in the scrip. Please list them in alphabetical order, unless they use another variable. Variables that are used in other variables should be in a separate block before the dependent variables.

No:

```
XYZ="123"
DEF="456"
PROD_D="/path/to/something"
CONF_D="${PROD_D}/conf"
ABC="${CONF_D}/foo.json"
```

Yes:

```
PROD_D="/path/to/something"

CONF_D="${PROD_D}/conf"

ABC="${CONF_D}/foo.json"
DEF="789"
XYZ="123"
```

Always quote strings containing variables or shell metacharacters when passing them to other commands or functions.

No:

```
echo $foo
```

Yes:

```
echo "${foo}"
```

Exceptions:

Bash system variables like `$?` , `$#` , `$@` , `$*` and `$0` - `$9` should not be surrounded in {} characters. They should still be double-quoted if passed as an argument to another function or script.

Pipelines

If a pipeline is too long to fit on one line, put one component on each line after the first, don't do random chunking. Indent each of the subsequent lines by two spaces. Also, remember to `set -o pipefail` in the beginning of your script.

Here's an example we want to format correctly

```
#!/bin/bash

# Assume the next line is longer than 80 chars for sake of example:
abc | def | grep -e foo -e bar | jkl | mno | pqr
```

No:

```
#!/bin/bash

abc | def | grep -e foo -e bar | \
jkl | mno | pqr
```

Yes:

```
#!/bin/bash

set -o pipefail

abc | \
  def | \
  grep -e foo -e bar | \
  jkl | \
  mno | \
  pqr
```

Comments

As with Python, comments should be preceded by a blank line, and there should not be blank lines between them and the code they are related to.

No:

```
blah blah blah
blah
blah
# Now we foo the bar

foo bar
```

Yes:

```
blah blah blah
blah
blah

# Now we foo the bar
foo bar
```

Examples:

Setting multipart environment variables:

If you're dealing with a multipart environment variable like `$PATH` or `$PYTHONPATH`, make changes easy to read in the diffs. E.g:

```
PYTHONPATH=/opt/numenta/htm.it
PYTHONPATH=$PYTHONPATH:/opt/numenta/nupic
PYTHONPATH=$PYTHONPATH:/opt/numenta/anaconda/lib/python2.7/site-packages
export PYTHONPATH
```

is preferred to

```
export PYTHONPATH=/opt/numenta/htm.it:/opt/numenta/nupic:/opt/numenta/anaconda/lib/pyt
```

since diffs will now highlight just the altered portion instead of highlighting the entire line.

Help text display

If a script requires arguments before it can run, it must have code that prints an error message that includes usage if it doesn't get them. If it receives a bad argument, it should display a list of valid arguments. It must also print the usage when it is passed the wrong number of arguments, or is passed `--help` as an argument. **If `--help` is given as the argument, the script must not do anything other than display its usage information and explicitly exit with a 0 error code.**

Here's an example for a script that requires two arguments:

```
#!/bin/bash
#

MY_NAME=$(basename $0)

print_help() {
  echo "${MY_NAME} usage:"
  echo "${MY_NAME} foo bar"
  echo
  echo "This script will blah two branches where foo is the original branch and bar is
  echo
}

if [ "$1" == "--help" ]; then
  print_help
  exit 0
fi

if [ $# != 2 ]; then
  print_help
  exit 1
fi
```

or, in the case of a script that doesn't take arguments

```
#!/bin/bash
#

MY_NAME=$(basename $0)

if [ "$1" == "--help" ]; then
  echo "${MY_NAME} usage:"
  echo "${MY_NAME}"
  echo
  echo "This script will blah blah blah and requires no arguments"
  echo
  exit 0
fi
```

Command line argument parsing.

Here's an example of good command line parsing:

```
interactive=0
filename=~/system_page.html

print_usage() {
  echo "${MY_NAME} usage:"
  echo "${MY_NAME} -f|--file FILENAME -i|--interactive"
  echo
  echo "This script will munge a file and can be run in interactive mode."
  echo
}

# If this command doesn't have default settings to run with, we should
# print help info if called with no args
if [ $# -eq 0 ]; then
```

```
    print_usage
    exit 0
  fi

  while [ "$1" != "" ]; do
      case "$1" in
          -f | --file )           shift
                                  filename="$1"
                                  ;;
          -i | --interactive )    interactive=1
                                  ;;
          -h | --help )           print_usage
                                  exit
                                  ;;
          * )                     print_usage
                                  exit 1

      esac
      shift
  done
```

How it works (cribbed from linuxcommand.org/wss0130.php):

This code is a little tricky, so bear with me as I attempt to explain it.

The first two lines are pretty easy. We set the variable `interactive` to zero. This will indicate that the interactive mode has not been requested. Then we set the variable `filename` to contain a default file name. If nothing else is specified on the command line, this file name will be used.

After these two variables are set, we have default settings, in case the user does not specify any options.

Next, we construct a `while` loop that will cycle through all the items on the command line and process each one with `case` . The `case` will detect each possible option and process it accordingly.

Now the tricky part. How does that loop work? It relies on the magic of `shift` .

`shift` is a shell builtin that operates on the positional parameters. Each time you invoke `shift` , it "shifts" all the positional parameters down by one.

Pages 19

Find a Page...

Home