Shell Scripting Examples with Explanations

Contents

1 zip-dir

We start with an extremely simple Bash shell script. This script accepts no options, and must have exactly one (directory) argument. Its purpose is to create a zip file of the files in a directory, in such a way that unzipping recreates the directory and its files. Doing this is of course a single command, the problem is that the syntax can be hard to recall when it is done only occasionally. Defining a script named "zip-dir" makes it easy to zip up a directory without having to reread the man page for the `zip` command every time you do this (a few times per year).

First up is some basic syntax checking of how the script was called:

```
if [[ $# != 1 ]]; then
  echo "Usage: zip-dir DIRECTORY" >&2
  exit 1
fi
```

It is standard practice in the Linux/UNIX world to have programs do some basic syntax checking of how they were called, and print out a "usage" message with the program's correct syntax if they were not called correctly. Here, we simply check that script was called with a single argument. We do not bother checking if the argument is indeed a directory, because `zip` will do that for us and print its own error message. However, adding that checking would be reasonable.

Note that usage messages are really error messages, so should be printed to *standard error* instead of *standard output*. That is the purpose of the *redirection* at the end of the `echo` line (we could have written it as "`> &2`" instead). The notation "`&2`" refers to *file descriptor* 2, file descriptors being Linux/UNIX OS identifiers for open files. FD 2 is by default standard error.

The "`exit 1`" call terminates the program and returns an *exit status* of 1, which means failure.

Here is the desired `zip` command call:

```
zip -r "${1%/}" "$1"
```

To zip a directory with its files, `zip` wants the `-r` or `--recurse-paths` option. The first argument is then the name of the zip file (`.zip` is automatically added if not supplied), and the second argument is the directory to zip up. Since the intention is to create a zip file named `foo.zip` when zipping directory `foo`, both `zip` arguments are basically `$1`.

There are two subtleties here. First, since the directory name could contain spaces, we must double quote `$1`, so use `"$1"`. The second subtlety is that directories can often end up with a "`/`" (forward slash) being appended. This is what happens, for example, if you use *tab completion* on a directory. We do not want to pass `$1` as the zip filename with a slash on the end, because in fact, `zip` fails to create a zip file then! Here, we use the Bash *parameter expansion operator* `%` to remove a forward slash from the RHS of the `$1` value if there is a forward slash (otherwise it has no effect).

The *exit status* of the script will by default here be the exit status of the final command that was run, i.e., `zip`. If a different exit status is desired, an explicit `exit` call must be placed

as the final line in the script.

Entire script:

```
if [[ $# != 1 ]]; then
  echo "Usage: zip-dir DIRECTORY" >&2
  exit 1
fi

zip -r "${1%/}" "$1"
```

2 mid

Next we will look at a more sophisticated, but still fairly simple Bash script. `mid` is a script to print particular line ranges in one or more files. It is intended to be similar to `head` and `tail`, but one must supply the *first line to print* as an command-line argument. It prints 10 lines by default as with `head`/`tail`, but this can be changed with the -n option (as with `head`/`tail`).

Example calls:

```
mid 10 foo
mid -n5 20 foo
mid -n5 20 foo1 foo2
```

mid is a simple script, since its functionality is ultimately implemented in a single line by using a *pipeline*, where `tail` is piped to `head` (with appropriate arguments). Obviously, one could simply always write the `tail`+`head` pipeline, but having `mid` defined as a script saves typing, and avoids a user having to think about how to correctly pass arguments to `tail` and `head` to get the desired functionality.

Here is the one line in the script that ultimately implements mid's functionality:

```
tail -n+${firstline} "$file" | head -n${numlines}
```

The rest of the code does the following:

- performs basic error checking on call syntax and command-line arguments
- "decodes" the command-line arguments (i.e., determines whether option is provided, etc.)
- handles multiple file arguments (or no file arguments)

The first code in the file is a Bash *function*, `print_usage`:

```
function print_usage()
{
    echo "Usage: mid [-nN] FIRST_LINENUM [FILE...]" >&2
    echo "(FIRST_LINENUM is a 1-based value)"       >&2
}
```

By putting the `echo` statements for the *"usage message"* into a function, the usage message can initiated from multiple places in the program if desired, and furthermore, it can be placed near the top of the program so that the syntax need not be repeated in the initial comments.

Next up is code to check for the `-n` option and determine the number of lines to be printed:

```
numlines=10
if [[ "$1" == -n* ]]; then
    numlines=${1#-n}
    shift
```

```
    if [[ ! "$numlines" =~ ^[0-9]+$ ]]; then
        echo "Invalid -n option N value: $numlines" >&2
        print_usage
        exit 1
    fi
fi
```

The variable `numliness` holds the number of lines to be printed. It is set to the default value of 10 initially, but will be changed if the `-n` option is given. This is a standard approach: set a variable to a default value and change its value if necessary. As opposed to having an if-then-else or case/switch statement, etc. that sets the variable's value only once. The advantage of the chosen approach is that we are guaranteed the variable has a value, even if we happen to screw up the logic for setting the correct value. Not a big deal either way, though.

The `-n` option must come *first* in the command-line arguments if it is supplied (see `mid`'s syntax usage message), so we can easily check if the option was supplied by checking `$1`, the value of the first command-line argument. (The command-line arguments with a shell invocation automatically become what Bash calls the *positional parameters*, which can be obtained with `$1`, `$2`, etc.)

There are multiple ways we could have checked if the first argument starts with `-n`. Here, the equality operator was used, taking advantage of the fact that Bash allows the RHS to be a *pattern*. (Don't quote the pattern or the `*` will be taken literally instead of matching anything!) Bash allows you to use either `==` or just `=` for the equality operator.

If the `-n` option was supplied, we need to then "decode" the option: separating the `N` numeric argument from the `-n`, and updating `numlines`. That is done here using a Bash *parameter expansion operator*, `${1#-n}`, which removes `-n` from the LHS of the `$1` value. Bash has only weak string processing capabilities, so if more complicated "decoding" were required, we probably would have had to use SED or AWK to accomplish what we needed. Still, the parameter expansion operators do support many common string operations, so be sure to consider them instead of automatically jumping to SED/AWK.

(The `shift` statement will be discussed below.)

Note that all the initial equality check does is see if the first command-line argument *starts* with `-n`. It does not determine whether the `-n` is followed by a valid, numeric argument. It is generally up to the author how much call validation to do before performing actions. Here, the decision was made to include code to check that `numlines` ends up a valid number, because if a non-number is given, it ends up being passed to `head`, producing an error message from `head`—which would be confusing for a user of this program. An easy way to check if a value consists of all digits is to use Bash' *regex* operator, `=~`, as was done here. (Again, don't quote the regex expression!)

Key in using a regex pattern is understanding that the one or more digits pattern "`[0-9]+`" must be surrounded by "`^`" and "`$`" indicating the start and end of text. Without those two metacharacters, the pattern "`[0-9]+`" would also match something like "`a1b`". Obviously when using regex's, one will often want to include the start and end metachars.

The next code does more basic syntax checking on the call:

```
if [[ $# -lt 1 ]]; then
    print_usage
    exit 1
fi
```

Often code to do basic syntax checking of the call will come first in a script, but if there can be options, such code will be simpler to write if it comes after options are processed (as here). By placing the basic syntax checking code after option decoding, we don't have to try to worry about whether the option was supplied or not. At this point, we must have `FIRST_LINENUM`, and we can have any number of file arguments (including zero). Thus, there must be at least one argument left in the arguments list, but there can be any number greater than or equal to one. So, the only basic syntax checking we do is to see if no arguments were supplied, and if so print the usage message. While this may seem like minimal "error checking," it does mean that you can easily produce the usage message with `mid`'s syntax by calling it without any arguments. This is often very useful!

As with `mid`, many shell scripts are designed to accept an *arbitrary* number of file (or other arguments). Bash has a *special parameter*, `@`, that will provide a *list* of arguments (no matter number), which is easy to use with a `for-in` loop. However, this means that we need to have only the file arguments left when we get to using `$@`, not things like the `-n` option or `FIRST_LINENUM`. Luckily, items can be removed from the (LHS of the) arguments list (i.e., positional parameters), with `shift`. That was the purpose of the `shift` statement after decoding the `-n` option above: remove the supplied option from the arguments list so we can end up with only file arguments.

Next comes:

```
firstline=$1
shift
```

We are saving `FIRST_LINENUM` (`$1`) into a variable with a meaningful name, and then removing `FIRST_LINENUM` from the arguments list, so we now will have only the `[FILE...]` arguments.

Finally, we get to actually processing the file arguments (or `stdin`):

```
for file in "${@:-/dev/stdin}"; do
    if [[ $# -gt 1 ]]; then echo "==> $file <=="; fi
    tail -n+${firstline} "$file" | head -n${numlines}
done
```

The standard approach to loop through a list of command-line arguments is:

```
for arg in "$@"; do...; done
```

The `for-in` loop sets `arg` to each argument successively and executes the loop body. Note that having `$@` inside of double quote is necessary to deal with filenames that may contain whitespace, so you should be in the habit of always using `"$@"` (or `"${@}"`).

If the arguments list is empty, `"$@"` will be the empty list, so the for loop body won't get executed at all. However, when dealing with argument lists of files (as here), in many cases we will want the for body to be executed for *standard input* when there are no file arguments.

This can be easily accomplished in Bash by using the "`:-`" parameter expansion operator along with the `@` special parameter:

```
for file in "${@:-/dev/stdin}"; do
```

Now, if the argument list is empty, so `"$@"` would be the empty list, Bash will instead make the list be the single element "`/dev/stdin`" (the Linux device name/path for standard input). The for loop body will end up being executed once, with `file` bound to `/dev/stdin`. This eliminates the need for testing whether there are no arguments or not, and then duplicating code (or writing a function). Simple and clean!

If there are multiple file arguments, `mid` duplicates the format used by `head`/`tail`, printing the filename before each file's output:

```
if [[ $# -gt 1 ]]; then echo "==> $file <=="; fi
```

At last, we get to the line that implements the `mid` functionality:

```
tail -n+${firstline} "$file" | head -n${numlines}
```

This is a *pipeline*, that uses `tail` to start printing lines at the right point in the file, and then uses `head` so that only the desired number of lines get printed (to standard output).

E.g., for the call "`mid -n5 20 foo`", the following pipeline results:

```
tail -n+20 foo | head -n5
```

It may appear that this could be very inefficient. E.g., what if file `foo` is a million lines long? It seems like `tail` will end up printing out lines 20 through one million, when just five lines are needed. *However, this is not what will happen, because it is not how pipelines work.* As `tail` prints lines, they are read by `head`. When `head` has printed out the number of lines it has been told to, it will *terminate!* If `tail` continues to try to write to the pipe, it will get an EOF (end-of-file) indication, which will cause it to terminate as well (so it won't write a million lines). In fact, experiments with large files found that the `tail`+`head` approach was faster than several other approaches, such as those using SED or AWK.

Finally, notice how different this solution is to the way you would have gone about implementing `mid` in a C-family language. First, we don't loop through the file arguments using a numeric for loop and accessing array elements (e.g., `argv[i]`). Instead, we use a `for-in` loop and take advantage of the list provided by `$@`. Much simpler, and we can handle no file arguments with little effort. Second, we don't have to open each file and then loop reading lines from the file, starting to print at the appropriate line and stopping after printing the desired number. Instead, we have `tail` open the file and read its lines, and we have `head` stop when desired. Less obvious is that we did not end up having to write any code to handle situations where the file had less than `FIRST_LINENUM` lines in it, or where there were less than `N` lines remaining from `FIRST_LINENUM`. Those cases will be automatically handled by `tail` and `head`.

This is what it means to use the appropriate style/idiom for the language, and why doing so will virtually always results in cleaner and shorter programs!

Entire script:

```
#!/usr/bin/bash

function print_usage()
{
    echo "Usage: mid [-nN] FIRST_LINENUM [FILE...]" >&2
    echo "(FIRST_LINENUM is a 1-based value)"       >&2
}


# Check for -n option:
numlines=10
if [[ "$1" == -n* ]]; then
    numlines=${1#-n}
    shift
    if [[ ! "$numlines" =~ ^[0-9]+$ ]]; then
        echo "Invalid -n option N value: $numlines" >&2
        print_usage
        exit 1
    fi
fi

# Check basic syntax:
if [[ $# -lt 1 ]]; then
    print_usage
    exit 1
fi

firstline=$1
shift

# Process the argument files (or stdin):
for file in "${@:-/dev/stdin}"; do
    if [[ $# -gt 1 ]]; then echo "==> $file <=="; fi
    tail -n+${firstline} "$file" | head -n${numlines}
done
```

3 backup-directory

`backup-directory` is a bit more complicated script in terms of its processing. It uses `tar` to create a *gzipped tarfile* backup of some directory:

```
backup-directory DIRECTORY_TO_BACKUP BACKUPS_DIRECTORY
```

`DIRECTORY_TO_BACKUP` is the path of the directory to be backed up, and `BACKUPS_DIRECTORY` is the directory in which to store the tarball.

`backup-directory` starts with a `print_usage` function and basic syntax checking (exactly two directory arguments must be supplied). It then get the arguments and puts them into meaningful variable names:

```
dirtobackup=$1
backupsdir=${2%/}  #Remove a trailing /
```

As we saw earlier, we might have trailing forward slash on `BACKUPS_DIRECTORY`, and while that won't cause an error in this script, it will make our messages look a bit weird (with multiple slashes in paths—which are ignored, but still look odd).

The script next verifies that both arguments are indeed directories, e.g.:

```
if [[ ! -d "$1" ]]; then
    echo "Invalid DIRECTORY_TO_BACKUP: $1" >&2
    exit 1
fi
```

Because we want `tar` to create a backup file that can be used to easily restore files from the backed up directory, we want to `cd` to the parent of the directory to be backed up and just give `tar` the final directory. E.g., if we are to backup `/home/carver/Documents`, we will need to `cd` to `/home/carver` and pass just `Documents` to `tar`. (If you pass `/home/carver/Documents` to `tar`, it creates a deeper structure that makes it difficult to pull individual `Document` files out of the tarball, and makes it difficult to reconstitute `Documents` under a different directory than `/home/carver`.)

The following code serves to separate the `DIRECTORY_TO_BACKUP` path into the two components we need below:

```
dirtobackupabs=$(realpath "$dirtobackup")
dirname=$(basename "$dirtobackup")
dirparent=$(dirname "$dirtobackup")
```

Since a *relative path* could be passed for `DIRECTORY_TO_BACKUP`, we need to make certain we have an *absolute path* in order to extract parent path. Here we use `realpath` to make certain we have an absolute path, then `basename` and `dirname` utilities to separate the path. All of these utilities are part of the *coreutils* package, which is standard, but sometimes isn't installed by default on every Linux. These are all handy and useful utilities to use in scripting.

Next comes code to create a filename for the tarball, where we want several pieces of information in the filename:

```
hostname="${HOSTNAME%%.*}"
os=$(grep DESCRIPTION /etc/lsb-release | cut -d= -f2 | tr -d "\"" | tr " " "_")
date=$(date "+%y%m%d")
filename="${hostname}-${os}-${dirname}-${date}.tgz"
```

This code contains some good examples of using basic Linux/UNIX utilities and pipelines. We want our tarball filename to contain the following info about the directory that was backed up: hostname of machine, OS version running, directory name, and date of backup. The hostname can be gotten from the *environment variable* `HOSTNAME`, but this may be the fully qualified hostname, e.g., `pc00.cs.siu.edu`, and we want just e.g. `pc00`, so the `%%` *parameter expansion operator* is used to strip longest `.*` from RHS. Every Linux distro has one or more files under `/etc` that specify the OS version. A standard one is `/etc/lsb-release` (LSB is Linux Standards Base). A pipeline using several steps is here used to extract the desired info and reformat it as desired. `grep`'ing `DISTRIBUTION` produces a line like: `DISTRIB_DESCRIPTION="Mageia 7"`. The `cut` utility is used to get from that: `"Mageia 7"`. The `tr` utility is then used to remove the double qoutes and change the space to an underscore, leaving using with e.g. `Mageia_7`, which is the desired format. Finally, the date string is gotten by calling the (extremely powerful) `date` command, giving it the desired formatting syntax. The filename is then constructed from each of the components (using Bash' concatenation).

Now comes the code to call `tar` and create the backup file:

```
echo "Creating backup of directory $dirtobackup:"
echo "  backup file is ${backupsdir}/${filename} ..."
cd "$dirparent"
tar -czf "${backupsdir}/${filename}" "$dirname"
```

Some messages are printed so that the user is clear on what is created and where. As noted above, we want to call `tar` with the the name of `DIRECTORY_TO_BACKUP`, not its path, so we need to have `cd`'d to its parent before calling `tar`.

The final code checks the *exit status* of `tar` and prints appropriate messages:

```
if [[ $? == 0 ]]; then
    echo "Done!" >&2
    exit 0
else
    echo "FAILED!!!!" >&2
    exit 1
fi
```

After a command is run, "`$?`" can be used to get the exit status of the command. This code specifically checks exit status using this approach. 0 (zero) means success, anything else means failure.

Entire script:

```
#!/usr/bin/bash

function print_usage()
{
    echo "Usage: backup-directory DIRECTORY_TO_BACKUP BACKUPS_DIRECTORY" >&2
}


# Check basic call syntax:
if [[ $# != 2 ]]; then
    print_usage
    exit 1
fi

dirtobackup=$1
backupsdir=${2%/}  #Remove a trailing /

# Make certain that DIRECTORY_TO_BACKUP exists:
if [[ ! -d "$dirtobackup" ]]; then
    echo "Invalid DIRECTORY_TO_BACKUP: $dirtobackup" >&2
    exit 1
fi

# Make certain that BACKUPS_DIRECTORY exists:
if [[ ! -d "$backupsdir" ]]; then
    echo "Invalid BACKUPS_DIRECTORY: $backupsdir" >&2
    exit 1
fi

# Parse DIRECTORY_TO_BACKUP to use with tar:
dirtobackupabs=$(realpath "$dirtobackup")
dirname=$(basename "$dirtobackupabs")
dirparent=$(dirname "$dirtobackupabs")

# Setup tarball filename:
hostname="${HOSTNAME%%.*}"
os=$(grep DESCRIPTION /etc/lsb-release | cut -d= -f2 | tr -d "\"" | tr " " "_")
date=$(date "+%y%m%d")
filename="${hostname}-${os}-${dirname}-${date}.tgz"

# Perform backup:
echo "Creating backup of directory $dirtobackup:"
echo "  backup file is ${backupsdir}/${filename} ..."
cd "$dirparent"
tar -czf "${backupsdir}/${filename}" "$dirname"

# Check status:
if [[ $? == 0 ]]; then
    echo "Done!" >&2
    exit 0
else
    echo "FAILED!!!!" >&2
    exit 1
fi
```

4 backup-directory-remote

`backup-directory-remote` is a variation on `backup-directory`, which is used to put the backup tarball on another machine. Obviously, storing a backup on another machine is safer than storing it on the machine you have backed up, and SSH makes that simple to do as part of the backup process (instead of having to create a local file and then copy it). There are just a few differences between `backup-directory-remote` and `backup-directory`.

Instead of supplying `BACKUPS_DIRECTORY`, one must now supply `REMOTE_HOST_BACKUPS_DIRECTORY`, specifying the username and hostname for SSH, in addition to the directory to place the tarball into. An example call could be:

```
backup-directory-remote ~/Documents carver@pc00.cs.siu.edu:/backups
```

The script starts off similar to `backup-directory`, but `REMOTE_HOST_BACKUPS_DIRECTORY` includes both the SSH username and hostname, as well as the remote directory, those two components will need to be separated:

```
remotehost=${remotebackupsdir%:*}
remotedir=${remotebackupsdir#*:}
```

As we can see in the example, a colon (":") separates the username-hostname from the remote directory. We use *parameter exapansion operators* to strip off the unwanted components for each of our two variables.

Because the backups directory is remote, if we want to verify that it exists, we will have to use SSH to do that:

```
if ! ssh ${remotehost} "cd \"${remotedir}\" &> /dev/null"; then
    echo "REMOTE_HOST_BACKUPS_DIRECTORY not reachable or does not exist: ...
    exit 1
fi
```

The method we are using to check for the existence of the remote directory is to use `ssh` to run the `cd` command on the remote machine. Basically we are doing:

```
ssh ${remotehost} "cd ${remotedir}"
```

This command should be clear if you have studied the SSH material. But what else is going on in that line (since it is more complicated)? First, since `remotedir` could contain spaces, we need to make sure the command run on the remote machine includes double quoting of the name. We do that by including *escaped double quotes* around the name (`\"${remotedir}\"`). The plain double quotes keep the `cd` command together for our local call to `ssh`. Escaping the double quotes means there will be double quotes around the directory when the `cd` command is run on the remote machine. Second, if the remote directory does not exist, we would see an error message from `cd`, but that would be confusing to users of our script. The *redirection* "`&>/dev/null`" suppresses all output from `cd`: "`&>`" redirects both standard out and standard error; "`/dev/null`" is often termed the "*bit bucket*" as it is a fake device that just results in output being discarded.

So how exactly do we figure out if the remote directory doesn't exist? Notice that we are calling `ssh` inside of an `if`. Recall that when we showed the syntax for if-then-else we had: `if test-command; then.....` Normally we use "`[[...]]`" (double square bracket) notation to write a logical expression and turn it into a "`test-command`", but here we are literally

using a command, i.e., `ssh`. By writing "`if ! ssh...`", we check if `ssh` returns a failure exist status, and if so, we print out the error message. The way `ssh` works when you run a remote command is that it returns the exit status of the remote command (if it gets run). While we could have just run the `ssh` command and then checked "`$?`" for its exit status, wrapping the `ssh` call inside the `if` is much shorter (and more obvious if you understand how `if` works with commands).

The other key difference in this script is combining `ssh` with `tar` to have the backup tarball created on the remote machine:

```
echo "Creating backup of directory $dirtobackup to $remotehost:"
echo "  backup file is ${remotehost}:${remotedir}/${filename} ..."
cd "$dirparent"
tar -cz -f - "$dirname" | ssh ${remotehost} "cat > \"${remotedir}/${filename}\""
```

As compare with the `backup-directory` script, here we tell `tar` to write the tarball to *standard out* by doing: "`-f -`". We *pipe* the tarball to `ssh`, and then have SSH pass the stream on to create a file on the remote machine: `"cat > \"${remotedir}/${filename}\""`. Again, how to do this sort of thing is discussed in the SSH lectures (e.g., why we need `cat`).

Entire script:

```
#!/usr/bin/bash

function print_usage()
{
    echo "Usage: backup-directory-remote DIRECTORY_TO_BACKUP REMOTE_HOST_BACKUPS_DIRECTORY" >&2
}


# Check basic call syntax:
if [[ $# != 2 ]]; then
    print_usage
    exit 1
fi

dirtobackup=$1
remotebackupsdir=$2

# Make certain that DIRECTORY_TO_BACKUP exists:
if [[ ! -d "$dirtobackup" ]]; then
    echo "Invalid DIRECTORY_TO_BACKUP: $dirtobackup" >&2
    exit 1
fi

# Parse REMOTE_HOST_BACKUPS_DIRECTORY to use with SSH:
remotehost=${remotebackupsdir%:*}
remotedir=${remotebackupsdir#*:}

# Make certain that remote backup directory is reachable and exists:
if ! ssh ${remotehost} "cd \"${remotedir}\" &> /dev/null"; then
    echo "REMOTE_HOST_BACKUPS_DIRECTORY not reachable or does not exist: $remotebackupsdir" >&2
    exit 1
fi

# Parse DIRECTORY_TO_BACKUP to use with tar:
dirtobackup=$(realpath "$1")
dirname=$(basename "$dirtobackup")
dirparent=$(dirname "$dirtobackup")

# Setup tarball filename:
hostname="${HOSTNAME%%.*}"
os=$(grep DESCRIPTION /etc/lsb-release | cut -d= -f2 | tr -d "\"" | tr " " "_")
date=$(date "+%y%m%d")
filename="${hostname}-${os}-${dirname}-${date}.tgz"

# Perform backup:
echo "Creating backup of directory $dirtobackup to $remotehost:"
echo "  backup file is ${remotehost}:${remotedir}/${filename} ..."
cd "$dirparent"
tar -cz -f - "$dirname" | ssh ${remotehost} "cat > \"${remotedir}/${filename}\""

# Check status:
if [[ $? == 0 ]]; then
    echo "Done!" >&2
    exit 0
else
    echo "FAILED!!!!" >&2
    exit 1
fi
```