Shell Scripting 6: Practical Advice

1. Shells and Variables
2. Control Constructs
3. Shell Expansions
4. Functions and Arrays
5. Bash Development
6. **Practical Advice**
   - **verifying script invocation and command installation**
   - **verifying commands succeeded**
   - **lists and whitespace and word splitting and IFS**
   - **processing text files**
   - **reading lines from files**
   - **file paths**
   - **decoding command line arguments**
7. Bash Details and Advanced Features

Verifying Proper Script Invocation

It is nearly always a good idea to have some initial code in a script to test if the script was called appropriately.

This is standard practice for Linux commands, and incorrect calls typically result in a brief **usage message** that gives the command's correct call syntax.

For example, suppose we have a script named `myprog` that is to take a *single argument*, which is to be a *directory* path.

In this case, we would at least want to verify that the script was called with *exactly one command line argument*, and we might also verify that the argument names a valid directory.

Verifying Proper Script Invocation (contd.)

We can do both these checks with the following code:

```
# Make certain script was called with a single argument:
if [[ $# != 1 ]]; then
    echo "Usage: myprog DIRECTORY" >&2
    exit 1
fi

# Make certain that DIRECTORY exists and is a directory:
if [[ ! -d "$1" ]]; then
    echo "Invalid DIRECTORY argument: $1" >&2
    exit 1
fi
```

Note that usage messages and other error messages are typically printed to **standard error** (`stderr`), which is done above with the **redirection**: `>&2`

Verifying Proper Script Invocation (contd.)

Many scripts takes some fixed number of arguments plus an *unlimited* number of arguments, such as files:
e.g., `myprog HEADERSTR FOOTERSTR FILE...`

In this case our usage test will have to be like:

```
if [[ $# -lt 2 ]]; then
```

Sometimes scripts can take *any number* of arguments, including zero: e.g., `myprog [FILE...]`
(if no `FILE` is supplied, read from stdin)

In such cases, the script will either not have a usage message test, or one can print the message if the user calls the script like:

```
myprog --help
```

Verifying Command/Executable Installed

Another aspect of writing *robust* scripts is making certain that programs/commands required by the script have been installed on the system.

The best way to do this is usually using the `which` command: "For each of its arguments it prints to stdout the full path of the executables that would have been executed when this argument had been entered at the shell prompt. It does this by searching for an executable or script in the directories listed in the environment variable PATH using the same algorithm as bash(1)."

Example checking for `enscript` (text to Postscript convertor):

```
if ! which enscript &>/dev/null; then
  echo "Program enscript required, please install first!"
  exit 1
fi
```

Verifying Commands Succeeded

Many commands run by shell scripts can possibly *fail*.

Writing *robust* shell scripts means including code that ensures commands have succeeded before proceeding in the script.

Otherwise, invalid data may be passed on, leading to incorrect results or unexpected and difficult to interpret error messages.

(Remember also that we may want to suppress error messages from commands by *redirecting standard error:* "`2>/dev/null`".)

Success/failure of commands is most commonly determined by their **exit status**.

Another approach is to look at the result/output of the command.

Verifying Commands Succeeded (contd.)

There are three approaches for running a command and checking its exit status:

- run the command as a `test-command` inside an `if`:

```
if ! mkdir "$newdir" 2>/dev/null; then
  ...handle mkdir failure...
fi
```

- test the status of a command with `$?`:

```
mkdir "$newdir" 2>/dev/null
if [[ $? != 0 ]]; then
  ...handle mkdir failure...
fi
```

- run a sequence of commands using `&&` or `||`:

```
mkdir "$newdir" 2>/dev/null && cp "$file" "$newdir" || ...handle failure...
```

Verifying Commands Succeeded (contd.)

Sometimes the only way to determine if a command "fails" is by testing its output.

Typically this will mean testing whether results are empty or not.

E.g., we might expect/need a set of files in some directory, but they may fail to be found:

```
files=$(ls "$dir")
if [[ -z "$files" ]]; then
  ...handle failure to find any files in $dir...
fi
```

We will generally use the following *string operators*:

- `-n` – unary operator tests if non-null/non-empty string
- `-z` – unary operator tests if null/empty string

Using a Variable in Subsequent Commands

Variables are often used in shell scripts to capture the output from a command, to allow for further processing.

However, it may not be clear how text that is captured in a variable, especially in later *pipelines*.

The key is know that `echo` writes to *standard output*, which is what gets passed along a pipeline, so it can start a pipeline:

```
line=$(grep -m1 $pattern "$csv_file")
field1=$(echo "$line" | cut -d, -f1)
...
```

Of course if we only need `field1` from `line`, we can simply do:

```
field1=$(grep -m1 $pattern "$csv_file" | cut -d, -f1)
```

Lists and Whitespace

In Bash, a **list** is a string of words separated by separators.
(Note: a data list is not the same as a **list command**.)

E.g., `A B C` (this is a list of three letters, `A`, `B`, `C`)

Lists are heavily used with **for-in** loops:

```
for file in *; do ...; done
```

(filename expansion of `*` produces a list of filenames)

A fairly frequently encountered issue is that the elements of a list may contain whitespace, so might get split by **word splitting**.

If this happens, an incorrect, longer list results, with the pieces of the intended elements now treated as individual list elements.

Lists and Whitespace (contd.)

For example, suppose you had this 3-element list of filenames in the CWD:

```
"file 1 .txt" "file 2 .txt" "file 3 .txt"
```

Without the double quotes around each filename, word splitting would produce a 9-element list:

```
file 1 .txt file 2 .txt file 3 .txt
```

The good news is that lists can work properly with such files:
(`ls -i` lists the filesystem *inode number* for the file)

```
$ for file in *; do ls -i "$file"; done
2494675 file 1 .txt
2494676 file 2 .txt
2494677 file 3 .txt
```

This code works because *word splitting* is *not* applied to the results of *filename expansion*.

Lists and Whitespace (contd.)

Other approaches can lead to problems however:

```
$ for file in $(ls); do ls -i "$file"; done
ls: cannot access file: No such file or directory
ls: cannot access 1: No such file or directory
ls: cannot access .txt: No such file or directory
ls: cannot access file: No such file or directory
...
```

The problem here is that word splitting *is* applied to the result of *command substitution*, breaking the filenames produced by the `ls` call apart.

Lists and Whitespace (contd.)

You might surmise that the fix for that is easy, we simply need to double-quote command substitution to supress word splitting:

```
$ for file in "$(ls)"; do echo ls -i "$file"; done
ls: cannot access file 1 .txt
file 2 .txt
file 3 .txt: No such file or directory
```

Yes, we surpressed word splitting, but now we no longer have a *list* of filenames, rather, we have `ls`' output as a *single word:* the filenames *separated by newlines* (i.e., on separate lines).

Bottom line: list elements that contain whitespace can be problems, best to stick to filename expansion for producing filename lists.

The intricacies of word splitting are covered further in the **Bash Details and Advanced Features** lecture.

Word Splitting and Resetting IFS

The Bash variable `IFS` ("inter-field separator") is a string that controls which characters result in **word splitting**.

`IFS` and thus word splitting behavior can be easily changed.

The most common thing one might want to do is change `IFS` so that word splitting occurs *only* on *newlines*.

This is done with: `IFS=$'\n'`

Doing so can make it possible to work with lists produced by commands like `find`, `grep`, `awk`, `sed`, etc. that return results as *lines* (separated by newlines).

Word Splitting and Resetting IFS

By changing `IFS`, we can use `ls` to get the files:

```
$ IFS=$'\n'
$ for file in $(ls); do ls -i "$file"; done
2494675 file 1 .txt
2494676 file 2 .txt
2494677 file 3 .tx
```

Note that `IFS` has now been changed for our shell session!

It be set back to the default with: `IFS=$' \t\n'`

(Since a *shell script* runs in a *new subprocess*, changing `IFS` in scripts won't affect its value in the calling shell.)

Word Splitting and Resetting IFS

The freqently used command `find` has similar issues and solutions (since it returns its filename results separated by newlines):

```
$ for file in "$(find . -name "*.txt")"; do ls -i "$file"; done
ls: cannot access ./file 1 .txt
./file 3 .txt
./file 2 .txt: No such file or directory

$ IFS=$'\n'
$ for file in $(find . -name "*.txt"); do ls -i "$file"; done
2494675 ./file 1 .txt
2494677 ./file 3 .txt
2494676 ./file 2 .txt
```

Processing Text Files

Because shell scripts can use any of the large number of Linux/UNIX filters/utilities combined in pipelines, they are particularly appropriate for working with *text files*.

`grep` is one of the more heavily used filter utilities.

E.g., we can test if a file contains any lines matching a **regular expression** (**regex**):

```
if grep --quiet "$regex" "$file"; then
  ...work on file containing regex lines...
fi
```

We can also retrieve those lines for further processing:

```
lines=$(grep "$regex" "$file")
```

Processing Text Files (contd.)

The following `grep` *options* are particularly useful in shell scripts:

- `-q`/`--quiet` – Quiet; do not write anything to standard output. Exit immediately with zero status if any match is found.
- `-v`/`--invert-match` – Invert the sense of matching, to select *non-matching* lines.
- `-c`/`--count` – Print only a *count* of matching lines (or non-matching lines With the `-v` option).
- `-o`/`--only-matching` – Print only the matched (non-empty) parts of a matching line.
- `-m NUM`/`--max-count=NUM` – Stop reading file after `NUM` matching lines.
- `-n`/`--line-number` – Prefix each line of output with the 1-based line number from file.

Processing Text Files (contd.)

Some other commands/filters that are particularly useful in shell scripts dealing with text files (e.g., as part of pipelines) are:

- `cut OPTION... [FILE]...` – Print selected parts of lines from each `FILE` to standard output.
- `tr [OPTION]... SET1 [SET2]` – Translate, squeeze, and/or delete characters from standard input, writing to standard output.
- `xargs [OPTION]... COMMAND [INITIAL-ARGUMENTS]` – Reads *items* from standard input and executes COMMAND for each, using INITIAL-ARGUMENTS followed a read item.

AWK and SED are also heavily used for manipulating text files.

Reading Lines from Files

There are a number of approaches that can work for reading through the lines in a file, but several have issues with *leading whitespace* in lines and/or with *blank lines*.

The following approach will loop through all the lines in a file (whose path is in the variable `filepath`), placing each entire line into the variable `line`, and terminating at the file-end.

```
IFS=$'\n'
while read line; do
  echo "$line"
done < "$filepath"
```

If *leading whitespace* in lines is not an issue, resetting of `IFS` can be *eliminated*.

Reading Lines from Files (contd.)

Instead of using *input redirection*, `exec` can be used to open the file on a new **file descriptor**:

```
exec 3<"$filepath"
IFS=$'\n'
while read -u3 line; do
  echo "$line"
done
```

Assumes this is only file opened, so ends up on *file descriptor* 3.

Instead of "`read -u3 line`" one can do "`read line <&3`".

Dealing with File Paths

Shell scripts frequently have to deal with **file paths** (pathnames).

E.g., the script is in one directory, source files in another, and target is yet another directory.

Code can often be simplified by `cd`'ing to an appropriate directory.

This is because **filename expansion** patterns that include directory info, will produce results containing that directory info:

```
$ for file in dir/*; do echo "$file"; done
dir/file1.txt
dir/file2.txt
...
```

Dealing with File Paths (contd.)

However, users may provide file/directory path arguments to scripts using either **absolute** or **relative** paths.

Once a script `cd`'s, arguments using *relative paths* will be *invalid!*

This often requires a script to determine the *absolute paths* for its arguments and/or the original CWD.

Key commands for dealing with paths are:

- `basename` – strip directory (and maybe suffix) from filename
- `dirname` – strip last component from filename (leaving directory)
- `realpath` – print resolved absolute file path (follow symlinks)
- `which` – print absolute path of command/executable (as would be found per `PATH`)
- `pwd` – print CWD

Dealing with File Paths (contd.)

Get/save CWD absolute path, e.g., so can return after `cd`'ing:

```
cwdsave=$(pwd)
```

Get absolute path of current script (required to call *recursively*):

```
cmd=$(which $0)
```

(**Note:** *Debian distros* use non-GNU version of `which` that returns `./command` when given as argument, instead of returning absolute path; suggest you replace it with *much better* GNU `which`!)

Getting absolute path of a file/directory argument (`$1`):

```
abs=$(realpath "$1")
```

(`realpath` does not gurantee returned path exists unless use *option* `-e`/`--canonicalize-existing`.)

Dealing with File Paths (contd.)

Example of path issues: copy all files from directory `$1` to `$2`.

Approach without `cd`'ing:

```
for file in "$1"/*; do
  cp "$file" "$2"/"$(basename "$file")"
done
```

Approach `cd`'ing to source directory:

```
targetdir=$(realpath "$2")
cd "$1"
for file in *; do
  cp "$file" "$targetdir"
done
```

(Note many *double quotes* to handle names containing *whitespace*.)

Decoding Command Line Arguments

Most shell scripts will require arguments be passed to them when invoked on the command line.

We have seen that command line arguments can be accessed via the **positional parameters** (e.g., `$1`) and certain **special parameters** (particularly `$@`).

However, many scripts must deal with *variable numbers of arguments*, for example when they can handle any number of file arguments (including sometimes, no file arguments).

In addition, it makes sense for scripts to use the same kind of **options** notation that is used for standard Linux/UNIX commands.

Dealling with options and variable numbers of arguments can complicate the code required to decode command line arguments (retrieve and use their values).

Decoding Command Line Arguments (contd.)

One typical pattern in script syntax is some number of required arguments followed by an arbitrary number of arguments of a certain type: e.g., `myscript ARG1 ARG2 FILE...`

This pattern is easily handled by retrieving the fixed args, calling `shift` to remove them from the arguments lists, and then using `"$@"` in a `for-in` loop:

```
arg1=$1
arg2=$2
shift2
for file in "$@"; do
  ...code to process file (a FILE argument)...
done
```

Decoding Command Line Arguments (contd.)

Frequently, one wants a script to make all `FILE` arguments optional, and read from **standard input** (`STDIN`) if no `FILE` arguments are given: e.g., `myscript ARG1 ARG2 [FILE...]`

The straightforward way to allow this leads to code duplication:

```
arg1=$1
arg2=$2
shift2
if [[ $# -gt 0 ]]; then
  for file in "$@"; do
    ...code to process file (a FILE argument)...
  done
else
  ...code to process STDIN...
fi
```

Decoding Command Line Arguments (contd.)

Various people have come up with somewhat tricky approaches that can avoid code duplication, though many are quite hard to understand.

In truth, it is very simple to handle this situation:

```
arg1=$1
arg2=$2
shift2
for file in "${@:-/dev/stdin}"; do
  ...code to process file (a FILE argument or STDIN)...
done
```

Trick: `${param:-word}` form gives value of `param`, unless it is unset/null, then the expansion of `word` is substituted.

Decoding Command Line Arguments (contd.)

We can do something similar if we might have only a single file argument or read from `STDIN`: e.g., `myscript ARG1 ARG2 [FILE]`:

```
arg1=$1
arg2=$2
shift2
file="${1:-/dev/stdin}"
...code to process file (FILE or STDIN)...
done
```

(Thanks is owed to the various posters on *stackoverflow* for their suggestions about handling files vs. stdin, nonetheless, the solution on the previous slide, synthesized from those postings, is either simpler or more general than every single posted solution!)

Decoding Command Line Arguments (contd.)

As already noted, when scripts need to allow options to be specified, following the standard Linux/UNIX syntax makes the most sense since users should understand it..

To recap, there are two Linux/UNIX options styles:

- short/old-style options:
  - start with a single dash/hyphen ("-")
  - are single letters (e.g., `-a`)
  - can take arguments (e.g., `-n10`)
- long/new-style options:
  - start with two dashes/hyphens ("--")
  - are words (e.g., `--all` or `--almost-all`)
  - can take arguments (e.g., `--number=10`)

Decoding Command Line Arguments (contd.)

If a script must handle only a small number of options, they can probably be decoded with relatively simple code—particularly if they are required to come first, args cannot be separated by whitespace, etc.

Many standard Linux/UNIX commands support a large number of options and allow significant flexibility with options (short options can be combined (e.g., `-acl`), arguments can be adjacent or whitespace-separated (e.g., `-n10` or `-n 10`), and options may even be allowed to be interspersed with command parameters).

Writing code to handle many options and/or allow such flexbility, would be a significant undertaking!

Instead, most programs decode options by using the POSIX/GNU C functions `getopt()` and `getopt_long()`.

Decoding Command Line Arguments (contd.)

For Bash shell scripts, there are two options that provide similar capabilities to the `getopt()` and `getopt_long()` C functions.

- `getopts` – a Bash builtin command
- `getopt` – a standalone utility provide by kernel.org's util-linux package

`getopts` is builtin to Bash so will be available with every Bash, is not as complicated to use for short options, but was not designed for long options so requires a bit of fussing to handle them.

`getopt` was designed to be more portable (across different shells), and can handle long options, but may not be installed by default and is more complicated to use.

Decoding Command Line Arguments (contd.)

Consider a script with a small number of options that come first:
E.g., `myscript [-a|--all] [-nN|--number=N] ARG1 ARG2`:

Decoding this script's options could be done like:

```
all=false; num=10  #default settings
while [[ "$1" == -* ]]; do
  case "$1" in
    -a|--all)   all=true ;;
    -n*)        num=${1#-n} ;;
    --number=*) num=${1#--number=} ;;
    *)          echo "Invalid option: $1" ;;
  esac
  shift  #remove processed option
done
arg1=$1
arg2=$2
```

Decoding Command Line Arguments (contd.)

Noq consider that script with only short options:
E.g., `myscript [-a] [-nN] ARG1 ARG2`:

`getopts` could be used to decode options like:

```
all=false; num=10  #default settings
while getopts ":an:" opt; do
  case $opt in
    a)  all=true ;;
    n)  num=$OPTARG ;;
    \?) echo "Invalid option: $1" ;;
  esac
done
shift $((OPTIND-1))
arg1=$1
arg2=$2
```

Decoding Command Line Arguments (contd.)

Returning to the script with both short and long options we can modify our use of `getopts` to handle this too:

```
all=false; num=10  #default settings
while getopts ":an:-:" opt; do
  case $opt in
    a)  all=true ;;
    n)  num=$OPTARG ;;
    -)  #Handle possible long option:
       LONG_OPT=${OPTARG%=*}
       if [[ "$LONG_OPT" == "$OPTARG" ]]; then LONG_OPTARG=""
       else LONG_OPTARG=${OPTARG#*=}; fi
       case $LONG_OPT in
         all) if [[ -n "$LONG_OPTARG" ]]; then echo "opt error: extra_arg"; fi
              all=true ;;
         number) if [[ -z "$LONG_OPTARG" ]]; then echo "opt error: missing_arg"; fi
                 num=$LONG_OPTARG ;;
         *) opt_error "invalid_opt" ;;
       esac ;;
    \?) echo "opt error: invalid_opt" ;;
    :) echo "opt error: missing_arg" ;;
  esac
done
shift $((OPTIND-1))
arg1=$1
arg2=$2
```