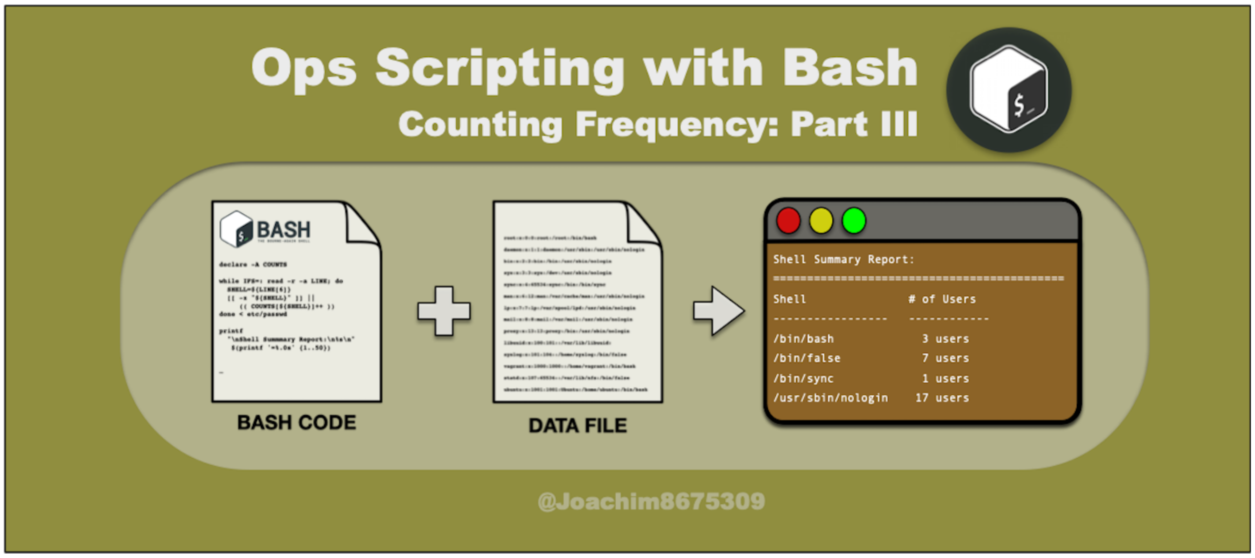

```
declare -A COUNTS

while IFS=: read -r -a LINE; do
  SHELL=${LINE[6]}
  [[ -z "${SHELL}" ]] ||
     (( COUNTS[${SHELL}]++ ))
done < etc/passwd

printf
  "\nShell Summary Report:\n%s\n"
   $(printf '=%.0s' {1..50})
```

```
Shell Summary Report:
==================================================
Shell                # of Users
----------------     ------------
/bin/bash              3 users
/bin/false             7 users
/bin/sync              1 users
/usr/sbin/nologin     17 users
```

Ops Scripting w. Bash: Frequency 3

Tracking Frequency in BASH (Bourne Again Shell): Part III

Joaquin Menchaca Follow
May 16 · 6 min read

This article shows how to do a serial pipeline style with loop constructs and a a survey of popular command line tools often used with shell programming.

Previously, I detailed how to process a file using *conditional loop* or a *collection loop* in a procedural way. In each cycle of the loop, we went through these steps:

1. slice off the shell string
2. verify if shell string is valid
3. create an entry in the associative array with the current count

For the serial pipeline style, we'll run through the process in this fashion:

```
filter_valid_lines | slice_7th_column | increment_count_entry
```

Previous Articles

The Problem

Ops Scripting w. BASH: Frequency

Tracking Frequency in BASH (Bourne Again Shell): Part I

medium.com

Procedural Solutions Way

Ops Scripting w. Bash: Frequency 2

Tracking Frequency in BASH (Bourne Again Shell): Part II

medium.com

The Solutions for the Serial Pipeline Way

For these solutions, we will use a sub-shell to pre-process the lines, and then send valid list of shells directly to the while loop with `<<<` operator:

```
while read -r SHELL; do
  process_shell_entry $SHELL
done <<< $( create_list_of_valid_shells )
```

For more information on this, see ***why subshell*** below.

Solution 3: Splitting with IFS and Read

In this solution, we use the internal *input-field-separator* `IFS` in conjunction with `read` to split each line into an array, where we slice off the shell column.

```
declare -A COUNTS

while read -r SHELL; do (( COUNTS[${SHELL}]++ )) done <<< \
 $(
    grep -v ':$' passwd |
    while IFS=: read -ra ITEMS; do
      echo ${ITEMS[6]};
    done
  )
```

Bash read split with Grep Filter

A reverse grep (`grep -v`) is used to filter out invalid lines that do not have a shell specified. If we wanted to stay pure-shell, we could do the same thing in BASH:

```
declare -A COUNTS

while read -r SHELL; do (( COUNTS[${SHELL}]++ )); done <<< \
 $( while read LINE;  do
      [[ $LINE =~ :$ ]] || echo $LINE
    done < etc/passwd |
    while IFS=: read -ra LINE; do
      echo ${LINE[6]}
    done
  )
```

Pure BASH Solution

Solution 4: Splitting with cut command

The `cut` command can remove sections form each line of a file (or files). We specify the 7th column; unlike an array, the count of columns starts at 1.

```
declare -A COUNTS

while read -r SHELL; do (( COUNTS[${SHELL}]++ )); done <<< \
$(
   grep -v ':$' passwd |
     cut -d: -f7
 )
```

frequency_4.sh hosted with ❤ by GitHub view raw

Cut with Grep Filter Solution

Solution 5: Splitting with sed

A popular tool in Linux and Unix is `sed` (stream editor), which has a substitute ability that we can use to only substitute the part we are interested in:

```
sed 's/pattern/replacement/' input_file
```

We can use a substitute operation to extract out the shell, which the part after the colon:

```
declare -A COUNTS

while read -r SHELL; do (( COUNTS[${SHELL}]++ )); done <<< \
$(
   grep -v ':$' passwd |
    sed -E 's/^.*:(.*)$/\1/'
 )
```

frequency_5a.sh hosted with ❤ by GitHub view raw

Sed Substitute with Grep Filter Solution

In our substitute, we are doing a *group match* with the `()`. We extract our group, which in this case is everything after the colon `:`, and then replace the whole string with only that group using `\1`.

Given that `sed` does more than *substitute*, being a full text editor driven by commands, we could also delete strings that are invalid, foregoing the need for grep:

```
declare -A COUNTS

while read -r SHELL; do (( COUNTS[${SHELL}]++ )); done <<< \
 $(sed -E -e 's/^.*:(.*)$/\1/' -e '/^$/d' passwd)
```

frequency_5b.sh hosted with ❤ by **GitHub** **view raw**

Sed Substitute with Sed Delete Filter Solution

The above demonstrates both *substitute* and *delete* commands for `sed`.

Solution 6: Splittling with awk command

The `awk` tool is extremely popular for working with field separated files. With `awk` we can print out any column:

```
awk '{print $5}' input_file # print's 5th field
```

This is the `awk` solution below:

```
declare -A COUNTS

# Process Input
while read -r SHELL; do (( COUNTS[${SHELL}]++ )); done <<< \
 $(
    grep -v ':$' passwd |
     awk -F: '{ print $NF }'
  )
```

frequency_6a.sh hosted with ❤ by GitHub view raw

Awk with Grep Filter Solution

In this example, we use a special variable in `awk` called `NF`, which represents the number of fields. The `awk` tool will slice off whatever is the last field using `NF` in this way.

As `awk` is a rather robust tool, we can conditionally match for each line to verify we have a valid line, so there's no need for `grep` in this case, as matching is built into `awk`:

```
declare -A COUNTS

while read -r SHELL; do (( COUNTS[${SHELL}]++ )); done <<< \
 $(awk -F: '!/:$/{ print $NF }' passwd)
```

frequency_6b.sh hosted with ❤ by **GitHub** **view raw**

Awk with Awk Match Filter Solution

Our negative match pattern with `!/:$/` are all lines that do not end with a colon `:`. We can do a positive match with `/[^:]$/`, which would match lines ending in non-colon characters.

Solution 7: Splitting with grep only-match command

The `grep` match tool has the ability to only print out the matched content, instead of the whole string when we use the *only-match* option of `-o`. This will extract only the shell portion of the string.

```
declare -A COUNTS

while read -r SHELL; do (( COUNTS[${SHELL}]++ )); done <<< \
 $(
    grep -v ':$' passwd |
     grep -o '[^:]*$'
  )
```

frequency_7.sh hosted with ❤ by **GitHub** **view raw**

Grep with Grep Filter Solution

Soluton 8: Splitting with perl command

The `perl` tool is a robust and fast tool to do kung-fu with strings. You can use Perl in command line mode with auto-splitting `-a` enabled, and then print out the column using the Perl's built-in `$F` variable.

```
declare -A COUNTS

while read -r SHELL; do (( COUNTS[${SHELL}]++ )); done <<< \
$(
   grep -v ':$' passwd |
    perl -naF':' -e 'print $F[6]'
 )
```

frequency_8a.sh hosted with ❤ by GitHub view raw

Perl with Grep Filter Solution

Like previous solutions, `grep` is pre-processing the lines to filter out invalid lines, but `perl` can do this by itself as well:

```
declare -A COUNTS

while read -r SHELL; do
  (( COUNTS[${SHELL}]++ ))
done <<< $(perl -naF':' -e 'print $F[6] if $F[6] !~ /^$/' passwd)
```

frequency_8b.sh hosted with ❤ by **GitHub** **view raw**

Perl with Perl Filter Solution

In addition to *auto-splitting mode* with `-a`, which is similar to `awk`, you can also simulate other tools like `grep` and `sed` to extract out the final shell column. Here's a summary of alternatives that can be used in the sub-shell:

```
###################### awk-like ######################
```

```
# perl auto-split w/ grep filter
grep -v ':$' passwd | perl -naF':' -e 'print $F[6]'

# perl auto-split w perl match filter
perl -naF':' -e 'print $F[6] if $F[6] !~ /^$/' passwd

############## grep-like with group-match ##############
# perl group-match w/ grep filter
grep -v ':$' passwd | perl -ne 'print $1 if /([^:]*)$/'

# perl group-match w/ perl match filter
perl -ne 'print $1 if /([^:]*)$/ and $1 !~ /^$/' passwd

##################### sed-like #####################
# perl substitute + group-match w/ grep filter
grep -v ':$' passwd | perl -pne 's/.*:([^:].*)$/$1/;'

# perl substitute + group-match and substitute to filter
perl -pne 's/.*:([^:].*)$/$1/ and s/^\s*$//' passwd
```

Why SubShell?

This is an excellent question. The current pattern above is to do the following:

```
while read -r VAR; do process $VAR; done <<< $(cmd < file |
cmd)
```

Would it not be simpler to just do something like:

```
cmd < file | cmd | while read -r VAR; do process $VAR; done
```

The second snippet is more intuitive, but unfortunately this will not work in our case. Specifically, when we leave the conditional loop, we lose any variables we were saving.

Pipe to a Loop

When you do a `command | loop`, you fork a sub-process with the loop, which does a fork/exec command and links the standard output of the command to the standard input of loop process. Unfortunately, variable changes in the loop are force scoped to just the loop, and thus will be lost outside of the loop.

Eval to a Loop

When you do a `loop <<< $(command)`, all of the standard output from the command is gathered into a buffer, then the loop will use the buffer as loop input. The subshell with `$(command)` is run at `eval` and standard out is substituted in-place of the `$(command)`, then the execution of the statement happens. Thus, your variables set inside the loop are preserved outside of the loop.

Solution Example with For Loop

Another way to use subshells if this format looks too funky, is to go back to a `for` loop and split the lines using this construct:

```
for VAR in $(cmd < file); do process $VAR; done
```

Below is an example solution for using the `for` loop:

```
declare -A COUNTS

for SHELL in $(awk -F: '{ print $NF }' passwd ); do
   (( COUNTS[${SHELL}]++ ))
done
```

frequency_9.sh hosted with ❤ by **GitHub** **view raw**

We can use the *default* `IFS`, as this will split that buffer by spaces, tabs, and new lines, and so if we have an empty shell, it will be skipped. We therefore no longer need a filter with `grep`, as it is implicit with the *default* separator.

The Conclusion

So these solutions are essentially a survey of popular tools that can be used for processing strings as an alternative to the internal *splitting* mechanism with `IFS`.

Takeways for looping constructs:

- pipe into a command from a string using `<<< $(command)`
- limitations of `command | while -r VAR` construct

- alternative loop construct with `for VAR in $(command)` construct
- default `IFS` will filter empty lines as separator is space, tab, and newlines.

Takeaways on methods to extracting a sub-string from text:

- using ***splitting*** from internal `IFS` variable in subshell.
- using ***splitting*** with an external tool: `cut` , `awk` , `perl`
- using ***substitute*** with ***group match*** with an external tool: `sed` , `perl`
- using ***group match*** or ***only-match*** with an external tool: `grep` , `perl`

Tool Specific Takeaways:

- `grep` can match lines or filter lines or a slice columns with regex
- `awk` can ***match*** as well as ***slice*** columns as well as other features
- `sed` can ***substitute*** and ***delete*** as well as other features
- `perl` can ***match***, ***group-match***, ***substitute***, and ***slice*** columns.