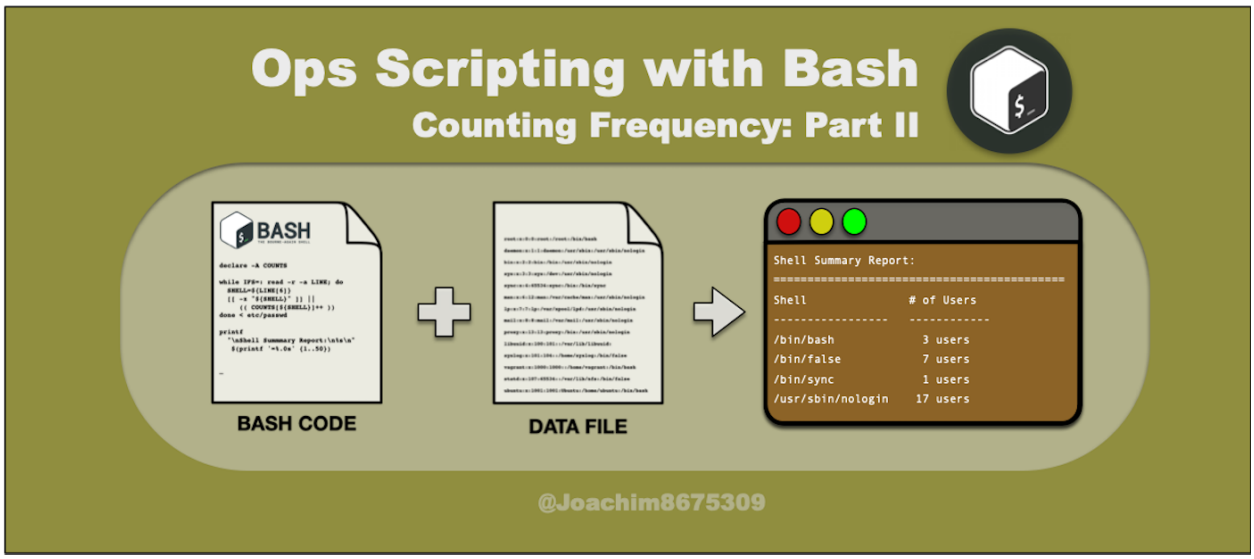


Ops Scripting w. Bash: Frequency 2

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

Joaquin Menchaca Follow

May 16 · 4 min read

Like other solutions in these series, I will divide this into to parts:

- ***Procedural Way***: show how to process each line where we slice out the shell and then build the data structure.
- ***Serial Pipeline***: show how to do a pipeline, where list of shells are piped into loop that simply adds to the data structure.

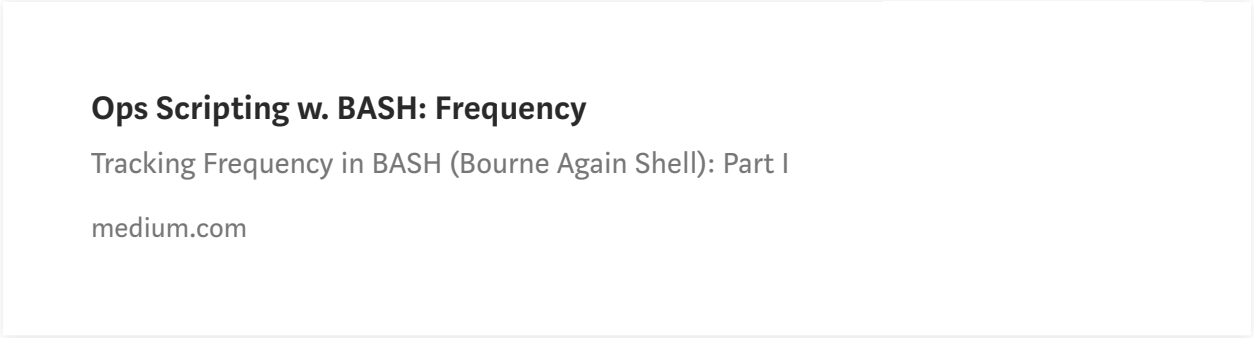
In this article, I'll show how to process colon delimited files using the

shell's built-in *auto-splitting mechanism* with the built-in `$IFS` environment variable.

In the **third part**, I'll show how to feed data in from a sub-shell to our loop construct, using either built-in split mechanism, or an external tool.

Previous Article

The Problem



The Solutions for the Procedural Way

Solution 1: Collection Loop

In our first solution, we'll demonstrate the collection loop, which has an auto-splitting facility. The `for` loop will automatically split text into parts, with the field separator specified by the built in `$IFS` environment variable. It then iterates through each part, for an

environment variable you specify.

```
declare -A COUNTS
declare -a LINE_ITEMS

IFS=$'\n'
for LINE in $(cat passwd); do
  IFS=: LINE_ITEMS=(${LINE})
  SHELL=${LINE_ITEMS[6]}
  [[ -z "${SHELL}" ]] || (( COUNTS[${SHELL}]++ ))
done
```

Process using Collection Loop

Before we begin we need to declare an *associative array* (also called a hash, map, or dictionary) with `declare -A my_assoc_array`. We also declare an array as well with `declare -a my_array`.

When processing text from a file, we need to split the text by the newline `\n`, so that we can process each line separately. We construct our basic collection loop like this:

```
IFS=$'\n'
for LINE in $(cat passwd); do
  process_each $LINE
done
```

As each line contains fields separated by a colon `:`, e.g. `field1:field2:field3`, we'll need the line into pieces by setting the

input field specifier `$IFS` to a colon `:` this time. With Bash, we can do this by surrounding a string with parenthesis `(string)` to create an array from a string:

```
IFS=: LINE_ITEMS=($LINE)
```

Side note: For the shell, if you prepend a line with environment variables, e.g. `VARA=foo VARB=bar command "$VARA $VARB"`, they will only apply to that command. So above, we only set the the input field separator to a colon `:` for a `ITEMS=()`, then it reverts back to what is was previous, which is the newline `\n` char.

After we create an array, we save the 7th column (indexed by 6) to a local variable. This is unnecessary, but done to make the code easier to read:

```
SHELL=${LINE_ITEMS[6]}
```

Now we need make sure we didn't copy a blank item, by seeing if our `$SHELL` variable is an empty string. If it is not empty, we can process further.

```
[[ -z "${SHELL}" ]] || create_shell_entry
```

This could also be written as:

```
if ! [[ -z "${SHELL}" ]]; then
  create_shell_entry
fi
```

Finally, we create our *associative array* entry, defaulting to `0` if the key does not yet exist:

```
(( COUNTS[${SHELL}]++ ))
```

The double parenthesis is for arithmetic operations `(( arithmetic_operation ))`, such as increment operator `++`.

Solution 2: Conditional Loop with Read

This is the most common approach, where read *"will read a line from the standard input and split it into fields" (**man page entry**)*.

```
declare -A COUNTS

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

frequency_1.sh hosted with ❤ by GitHub view raw

This is the basic construct on how to open a file and read into a variable:

```
while read -r LINE; do
  process_line $LINE
done < input_file
```

We can split each line of input by the `IFS` (input field separator), such as a colon `:`, an then extract the shell info as one of the array elements:

```
while IFS=: read -ra LINE_ITEMS; do
  SHELL=${LINE_ITEMS[6]}
  process_shell_item $SHELL
done < passwd
```

Notice above that we use the IFS for only the read command, and after it reverts back to the previous setting.

Now, we'll create a new shell entry, but only if we have a valid shell

info:

```
[[ -z "${SHELL}" ]] || create_shell_entry
```

And like before we use the arithmetic operator `(( arithmetic_expression ))` to increment the count. If we the key does not have an associated value, it will treat a blank string as `0`, and increment it.

```
(( COUNTS[${SHELL}]++ ))
```

Which Solution is better?

For processing files and splitting the string, `while read` is typically preferred combination simply because it is less work.

Default Field Separator

Reviewing from above, compare these two:

```
############ for loop way ############
IFS=$'\n'
for LINE in $(cat input_file); do
  ### process_line ####
done
```

```
########### while loop way ###########
while read -r LINE; do
  ### process_line ####
done < input_file
```

Alternative Field Separator

And we have these two:

```
############ for loop way ############
IFS=$'\n'
for LINE in $(cat passwd); do
  IFS=: LINE_ITEMS=($LINE)
  ### process_fields ####
done
```

```
########### while loop way ###########
while IFS=: read -ra LINE_ITEMS; do
  ### process_fields ####
done < passwd
```

Next Article

Ops Scripting w. Bash: Frequency 3

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

medium.com

The Conclusion

So there you have it, two ways to process files in a procedural way, and use the built-in *input field separator* to split a line of text.

In the **next article**, I'll show how to use more serial pipeline approach, where only a list of shells are sent to the main loop.

The takeaways from this include:

- process file through `while read -r VAR; do ...; done < file` loop construction
- process a file through `for VAR in $(cat file); do ...; done` loop construction
- split a string into an Bash array with `read -a` or with array notation `()`.
- *associative arrays*: creating, referencing, enumerating values and keys
- arithmetic with `(( ))`

Some more subtle takeaways:

- Default *input-field-separator* IFS separates spaces, tabs, and newlines.
- Need to add `IFS='\n'` when lines themselves need to be further

split, unless using `read` command.