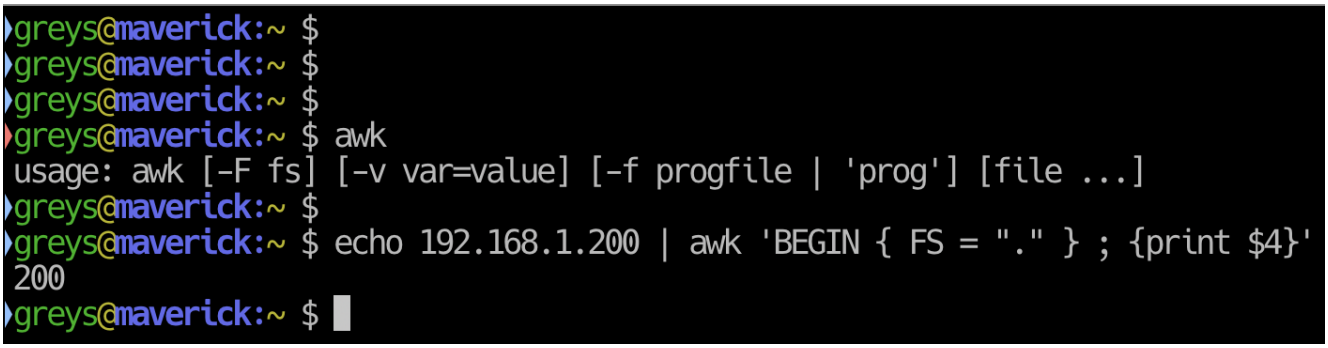
awk delimiter

Since awk field separator seems to be a rather popular search term on this blog, I'd like to expand on the topic of using **awk delimiters** (field separators).

Two ways of separating fields in awk

There's actually more than one way of **separating awk fields**: the commonly used -F option (specified as a parameter of the **awk command**) and the **field separator variable FS** (specified inside the **awk** script code).

awk -F field separator



As you may have seen from the **awk field separator** post, the easiest and quicked way to use one is by specifying -F command like option for awk (in the example we're extracting the last octet of the IPv4 address):

```
greys@maverick:/ $ ifconfig en0 | grep "inet " | awk '{print $2}'
192.168.1.220
greys@maverick:/ $ echo 192.168.1.220 | awk -F. '{print $4}'
220
```

Field Separator (FS) variable in awk

As your awk scripting gets better and more complex, you'll probably recognise that it's best to put such options inside the awk script instead of passing them as command line options. The benefit is, of course, that you don't risk getting different (wrong!) results just because you forgot to specify a command line option – everything is contained in your script, so you run it with minimal set of parameters and always get the same result.

So, the second way of separating fields in awk script is by using the FS (field separator) variable, like this:

```
greys@maverick:~ $ echo 192.168.1.200 | awk 'BEGIN { FS = "." } ; {print $4}'
200
```

See Also

- awk
- awk field separator
- top 5 awk field separators
- Review logs with tail and awk
- Unix commands
- Advanced Unix commands

Review Latest Logs with tail and awk

Part of managing any Unix system is keeping an eye on the vital log files.

Today I was discussing one of such scenarios with a friend and we arrived at a pretty cool example involving **awk command** and eventually a **bash command substitution.**

Let's say we have a directory with a bunch of log files, all constantly updated at different times and intervals. Here's how I may get the last 10 lines of the output from the most recent log file:

```
root@vps1:/var/log# cd /var/log
root@vps1:/var/log# ls -altr *log
-rw-r--r-- 1 root root 32224 Jul 10 22:49 faillog
-rw-r----- 1 syslog adm 0 Jul 25 06:25 kern.log
-rw-r--r-- 1 root root 0 Aug 1 06:25 alternatives.log
-rw-r--r-- 1 root root 2234 Aug 8 06:34 dpkg.log
-rw-rw-r-- 1 root utmp 294044 Aug 15 22:32 lastlog
-rw-r----- 1 syslog adm 12248 Aug 15 22:35 syslog
-rw-r----- 1 syslog adm 5160757 Aug 15 22:40 auth.log
```

Ok, now we just need to get that filename from the last line (**auth.log**).

Most obvious way would be to use tail command to extract the last line, and awk to show the 9th parameter in that line – which would be the filename:

```
root@vps1:/var/log# ls -altr *log | tail -1 | awk '{print $9}'
auth.log
```

Pretty cool, but can be optimised using awk's END clause:

```
root@vps1:/var/log# ls -altr *log | awk 'END {print $9}'
auth.log
```

Alright. Now we wanted to show the 10 lines of output, which we can use tail -10 for.

A really basic approach is to assing the result of the line we're using to a variable in Bash, and then access that variable, like this:

```
root@vps1:/var/log# FILE=`ls -altr *log | tail -1 | awk
```

'{print $9}'`
root@vps1:/var/log# **tail -10 ${FILE}**
Aug 15 22:40:37 vps1 sshd[26578]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost=159.65.145.196
Aug 15 22:40:39 vps1 sshd[26578]: Failed password for invalid user Fred from 159.65.145.196 port 47934 ssh2
Aug 15 22:40:39 vps1 sshd[26578]: Received disconnect from 159.65.145.196 port 47934:11: Normal Shutdown, Thank you for playing [preauth]
Aug 15 22:40:39 vps1 sshd[26578]: Disconnected from 159.65.145.196 port 47934 [preauth]
Aug 15 22:41:15 vps1 sshd[26580]: Connection closed by 51.15.4.190 port 44958 [preauth]
Aug 15 22:42:02 vps1 sshd[26585]: Connection closed by 13.232.227.143 port 40054 [preauth]
Aug 15 22:43:23 vps1 sshd[26587]: Connection closed by 51.15.4.190 port 52454 [preauth]
Aug 15 22:44:08 vps1 sshd[26589]: Connection closed by 13.232.227.143 port 47542 [preauth]
Aug 15 22:45:01 vps1 CRON[26604]: pam_unix(cron:session): session opened for user root by (uid=0)
Aug 15 22:45:01 vps1 CRON[26604]: pam_unix(cron:session): session closed for user root

But an ever shorter (better?) way to do this would be to use the **command substitution in bash**: the output of a command becomes the command itself (or string value in our case).

Check it out:

root@vps1:/var/log# **tail -10 $(ls -altr *log | tail -1 | awk '{print $9}')**
Aug 15 22:42:02 vps1 sshd[26585]: Connection closed by 13.232.227.143 port 40054 [preauth]
Aug 15 22:43:23 vps1 sshd[26587]: Connection closed by 51.15.4.190 port 52454 [preauth]
Aug 15 22:44:08 vps1 sshd[26589]: Connection closed by 13.232.227.143 port 47542 [preauth]
Aug 15 22:45:01 vps1 CRON[26604]: pam_unix(cron:session): session opened for user root by (uid=0)

```
Aug 15 22:45:01 vps1 CRON[26604]: pam_unix(cron:session): session closed for user root
Aug 15 22:45:26 vps1 sshd[26610]: Connection closed by 51.15.4.190 port 59872 [preauth]
Aug 15 22:46:15 vps1 sshd[26612]: Connection closed by 13.232.227.143 port 55030 [preauth]
Aug 15 22:46:23 vps1 sshd[26608]: Connection closed by 18.217.190.140 port 40804 [preauth]
Aug 15 22:47:28 vps1 sshd[26614]: Connection closed by 51.15.4.190 port 39044 [preauth]
Aug 15 22:48:20 vps1 sshd[26616]: Connection closed by 13.232.227.143 port 34286 [preauth]
```

So in this example **$(ls -altr *log | tail -1 | awk '{print $9}')** is a substituion – bash executes the command in the parenthesis and then passes the resulting value to further processing (becoming a parameter for the **tail -10** command).

In our command above, we're essentially executing the following command right now:

root@vps1:/var/log# **tail -10 auth.log**

only auth.log is always the filename of the log file that was updated the latest, so it could become syslog or dpkg.log if they're updated before next auth.log update.

See Also

- awk field separator
- awk delimiter
- awk command

Bash Scripts – Examples

I find it most useful when I approach any learning with a clear goal (or at least a specific enough task) to accomplish.

Here's a list of simple Bash scripts that I think could be a useful learning exercise:

- **what's your name?** – asks for your name and then shows a greeting
- **hello, world!** (that also shows hostname and list server IP addresses)
- **write a system info script** – that shows you a hostname, disks usage, network interfaces and maybe system load
- **directory info** – script that counts disks space taken up by a directory and shows number of files and number of directories in it
- **users info** – show number of users on the system, their full names and home directories (all taken from /etc/passwd file)
- **list virtual hosts on your webserver** – I actually have this as a login greeting on my webservers – small script that highlights which websites (domains) your server has in web server (Apache or Nginx) configuration.

Do you have any more examples of what you'd like to do in a Linux shell? If not, I'll just start with the examples above. The plan is to replace each example name in the list above with a link to the **Unix Tutorial** post.

Centralized BASH history with timestamps

For every Unix user, there comes a point where shell history suddenly becomes very relevant. You learn to consult it, then start recovering the last command, then switch to searching past commands history to save precious time normaly taken typing.
Shortly after such a point in your life, you'll probably want to enhance your shell history in two very common ways:

1. Make sure every terminal window can update AND access your centralized shell history. So you run a command or two in one window, then type "history" anywhere else and see them two commands right there.
2. Provide meanigful timeline, this is done with timestamps. Very simple and powerful change helps you see exactly when each command occured.

Here's how you achieve both of these massive improvents to your history in BASH. Just add this to **/etc/bashrc** on your Linux system:

```
export HISTTIMEFORMAT="%d.%m.%y %T "
export HISTCONTROL=ignoredups:erasedupsshopt -s histappend
export
PROMPT_COMMAND="${PROMPT_COMMAND:+$PROMPT_COMMAND$'\n'}history
-a; history -c; history -r;"
export HISTCONTROL=ignoreboth
```

Fixed calculations in Unix

scripts

Although I've already shown you how to sum numbers up in bash, I only covered the bash way of doing it. I really like scripting with bash, but when it comes to calculations, there's quite a few important features missing from bash, and fixed point (thanks for the correction, Azrael Tod!) calculations is one of them. Fortunately, **bc command** comes as a standard in most Unix distros, and can be used for quite complex calculations.

Basic calculations with bc

bc is a very simple command. It takes standard input as an expression and then evaluates this, performing all the necessary calculations and showing you the result. Thus, to quickly sum numbers up or get a result of some other calculation, simply echo the expression and then pipe it out to the **bc command**:

```
ubuntu$ echo "1+2" | bc
3
```

Now, in scripts your calculations with **bc** are done quite similarly to what we did in **bash**. Here's an example:

```
ubuntu$ NUMBER1=1
ubuntu$ NUMBER2=2
ubuntu$ SUM=$(echo "$NUMBER1+$NUMBER2"| bc)
ubuntu$ echo $SUM
3
```

I told you these calculations would be basic, right? Now onto the more interesting stuff!

Fixed point calculations with bc

Most people learn about bash math limitations when they attempt to do a simple calculation but can't get the current

answer with fixed point values. By default, all the operations happen with integers, and that's what you would get:

```
ubuntu$ echo "1/2" | bc
0
```

Now, if you expect 0.5 to be the result of dividing 1 by 2, you need to explain it to **bc**, because by default it doesn't show you any fractional part of the number.

The way you do this is quite simple: all you have to do is specify the number of digits you'd like to see after the radix point of your result. For example, if I set this number to 5, I'll get **bc** to output the result of my calculation with 5 digits after the radix point. The special keyword to convey this intention to the **bc** command is called **scale**. Just specify the scale value and separate it from the rest of your expression by the semicolon sign:

```
ubuntu$ echo "scale=5; 1/2" | bc
.50000
```

Here's another example:

```
ubuntu$ echo "scale=5; 0.16*10.79" | bc
1.7264
```

Hope this answers your question! **bc command** is very powerful, so I'll definitely have to revisit it again in the future. For now though, enjoy the fixed point calculations and be sure to ask questions if you think I can help!

See also:

- **Summing numbers up in Unix scripts**
- **Basic math in Unix scripting**
- **Another way of doing calculations in scripts**

Unix scripts: how to sum numbers up

If you're ever thought of summing up more than two numbers in shell script, perhaps this basic post will be a good start for your Unix scripting experiments.

Basic construction for summing up in shell scripts

In my Basic arithmetic operations in Unix shell post last year, I've shown you how to sum up two numbers:

```
SUM=$(($NUMBER1 + $NUMBER2)
```

using the same approach, it's possible to calculate sums of as many numbers as you like, if you use one of the loops available in your shell.

Before we get started, let's create a simple file with numbers we'll work with:

```
ubuntu$ for i in 1 2 3 4 5 6 7 8 9; do echo $i >> /tmp/nums; done;
ubuntu$ cat /tmp/nums
1
2
3
4
5
6
7
8
9
```

Using a while loop to sum numbers up in Unix

Here's an example of how you can use a **while loop** in Unix shell for summing numbers up. Save it as a **/tmp/sum.sh** script, and don't forget to do **chmod a+x /tmp/sum.sh** so that you can run it!

```
#!/bin/sh
#
SUM=0
#
while read NUM
do
        echo "SUM: $SUM";
        SUM=$(($SUM + $NUM));
        echo "+ $NUM: $SUM";
done < /tmp/nums
```

Here's how the output will look when you run it:

```
ubuntu$ /tmp/sum.sh
SUM: 0
+ 1: 1
SUM: 1
+ 2: 3
SUM: 3
+ 3: 6
SUM: 6
+ 4: 10
SUM: 10
+ 5: 15
SUM: 15
+ 6: 21
SUM: 21
+ 7: 28
SUM: 28
+ 8: 36
SUM: 36
+ 9: 45
```

Of course, your data will most probably will be in a less readable form, so you'll have to do a bit of parsing before you get to sum the numbers up, but the loop will be organized in the same way.

That's it for today, enjoy and feel free to ask questions!

See also:

- **Basic arithmetics in scripts**
- **Variables in Unix shells**
- **Unix glossary**

Easy date calculations in Unix scripts with GNU date

When I was writing a post about using date command to confirm date and time in your Unix scripts, I made a note in my future posts list to cover the date calculations – finding out the date of yesterday or tomorrow, and so on. Today I'll show you the simplest way to calculate this.

GNU date command advantage

GNU version of the **date** command, although supporting a common syntax, has one great option: it allows you to specify the desired date with a simple string before reporting it back. What this means is that by default this specified date is assumed to be "now", but you can use other keywords to shift the result of the date command and thus show what date it was yesterday or a week ago:

Here’s a default **date** output for the current date and time:

```
ubuntu$ date
Fri Sep 19 08:06:41 CDT 2008
```

Now, the parameter for specifying desired date is **-d** or **–date**, and if you use it with the “now” or “today” parameter, you’ll get similar output:

```
ubuntu$ date -d now
Fri Sep 19 08:06:44 CDT 2008
ubuntu$ date -d today
Fri Sep 19 08:06:50 CDT 2008
```

Showing tomorrow’s date

Similarly, you can get tomorrow’s date:

```
ubuntu$ date -d tomorrow
Sat Sep 20 08:02:12 CDT 2008
```

Obviously, if you feel like specifying a format for the date, you can do it:

```
ubuntu$ date -d tomorrow "+%b %d, %Y"
Sep 20, 2008
```

Find out yesterday’s date

Again, there’s no rocket science involved in showing yesterday’s date neither:

```
ubuntu$ date -d yesterday "+%b %d, %Y"
Sep 18, 2008
```

Show a date few days away

If you’re interested in a certain date a few days or even weeks away, here’s how you can do it:

Example 1: a date 5 days ago

```
ubuntu$ date -d "-5 days"
Sun Sep 14 08:45:57 CDT 2008
```

Example 2: a day 2 weeks from now

```
ubuntu$ date -d "2 weeks"
Fri Oct  3 08:45:08 CDT 2008
```

Example 3: a day two weeks ago from now

```
ubuntu$ date -d "-2 weeks"
Fri Sep  5 08:45:11 CDT 2008
```

Extreme example: a day 50 years ago

If you're really curious about dates in Unix, you can even make GNU date go back a few years:

```
ubuntu$ date -d "-50 years"
Fri Sep 19 08:47:51 CDT 1958
```

That's it for today, hope you like this little discovery – having mostly worked with Solaris systems most of my career, I didn't know my Ubuntu had this functionality bonus. Really useful!

See also:

- **Time and date in Unix scripts**
- **Variables in Unix shell scripts**
- **Updating shell variables in Unix**

Another way to use math expressions in shell scripts

Today I'd like to expand a bit more on the basic calculations in Unix scripts.

Use Parenthesis to simplify math expressions

In Basic Arithmetic Operations post I've shown you how expression evaluation can be used to calculate simple math expressions in your Unix shell:

```
ubuntu$ START=1
ubuntu$ FINISH=10
ubuntu$ ELAPSED=$(($FINISH - $START))
ubuntu$ echo $ELAPSED
9
```

Although this approach looks clean enough, there's a way to simplify it even further if you put everything in parenthesis. In other words, the same result (ELAPSED receiving a correct value of FINISH value minus START value) can be achieved this way:

```
ubuntu$ ((ELAPSED=FINISH-START))
ubuntu$ echo $ELAPSED
9
```

It's a matter of preference, and I used to always do such calculations the former way shown, but now that one of the readers of this blog pointed out the latter way of using it, I think I might change my habits – I actually like this way much better.

See also:

- Time and date in Unix scripts

- Using variables in scripts
- Basic math in Unix scripts

Updating Values of Your Shell Variables in Unix

If you're just getting started with your **Unix scripting** or new to Unix shell, today's little tip will make your life much easier: I'm going to show you how to update your shell variables.

Updating shell variable in Unix

If you have a text variable in your script and would like to append some text to it or somehow process this value, it's perfectly normal to reuse the current variable value using a syntax like this:

```
ubuntu$ echo $VAR1
hello
ubuntu$ VAR1=$VAR1" world!"
ubuntu$ echo $VAR1
hello world!
```

You see? It's this easy!

Common cases of reusing variable values in Unix shells

Most frequently, I use the technique described above to update some of my environment variables. I'm sure you'll find them useful too.

Updating the PATH variable

PATH is the user environment variable responsible for the directories where Unix shell looks for executable commands every time you type something. Quite often you get a “file not found” error not because there isn’t such a command installed in your OS, but simply because your PATH variable has not a correpct path to a directory with that command.

Here’s an example from a standard user on one of my Red Hat Enterprise Linux systems.

I like the runlevel command, it’s quite simple and can be useful in scripts. When I attempt to run it, here’s what happens:

```
redhat$ runlevel
bash: runlevel: command not found
```

Like I said earlier, it’s most likely because my PATH variable doesn’t have the /sbin directory which is where this command resides. Let’s confirm this:

```
redhat$ echo $PATH
/usr/lib/qt-3.3/bin:/usr/kerberos/bin:/usr/local/bin:/bin:/usr
/bin:/usr/X11R6/bin
```

Now it’s time to update the PATH and append the /sbin path to it:

```
redhat$ PATH=$PATH:/sbin
```

runlevel command will be found now and here’s a proof:

```
redhat$ runlevel
N 5
```

Updating the MANPATH variable

Another good example is the **MANPATH** variable, which contains the list of directories with manpages which **man command** uses.

If some command doesn’t have its man page in neither of the directories specified by MANPATH, you’ll get an error.

Here’s an example from one of the Solaris systems, I’m looking for a Veritas vxprint command man page:

```
solaris$ man vxprint
No manual entry for vxprint.
solaris$ echo $MANPATH
/usr/man:/opt/sfw/man:/usr/perl5/man
```

Let’s add the /opt/VRTS/man to the MANPATH variable:

```
bash-2.03# MANPATH=$MANPATH:/opt/VRTS/man
solaris$ echo $MANPATH
/usr/man:/opt/sfw/man:/usr/perl5/man:/opt/VRTS/man
```

It’s bound to work now:

```
solaris$ man vxprint
Reformatting page.  Please Wait... done
...
```

That’s all I wanted to share with you today. Hope you liked the tip, and as always – feel free to ask any questions!

See also

- **Environment variables in Unix**
- **Using variables in shell scripts**
- **Unix scripts: basic arithmetic operations**
- Unix commands
- Unix Basic Commands

How To Parse Text Files Line by Line in Unix scripts

I'm finally back from my holidays and thrilled to be sharing next of my Unix tips with you!

Today I'd like to talk about parsing text files in Unix shell scripts. This is one of the really popular areas of scripting, and there's a few quite typical limitations which everyone comes across.

Reading text files in Unix shell

If we agree that by "reading a text file" we assume a procedure of going through all the lines found in a clear text file with a view to somehow process the data, then **cat command** would be the simplest demonstration of such procedure:

```
redhat$ cat /etc/redhat-release
Red Hat Enterprise Linux Client release 5 (Tikanga)
```

As you can see, there's only one line in the **/etc/redhat-release** file, and we see what this line is.

But if you for whatever reason wanted to read this file from a script and assign the whole release information line to a Unix variable, using cat output would not work as expected:

```
bash-3.1$ for i in `cat /etc/redhat-release`; do echo $i; done;
RedHat
Enterprise
Linux
Client
release
5
(Tikanga)
```

Instead of reading a line of text from the file, our one-liner

splits the line and outputs every word on a separate line of the output. This happens because of the shell syntax parsing – Unix shells assume space to be a delimiter of various elements in a list, so when you do a for loop, Unix shell interpreter treats each line with spaces as a list of elements, splits it and returns elements one by one.

How to read text files line by line

Here's what I decided: if I can't make Unix shell ignore the spaces between words of each line of text, I'll disguise these spaces. Since my solution was getting pretty bulky for a one-liner, I've made it into a script. Here it is:

```
bash-3.1$ cat /tmp/cat.sh
#!/bin/sh
FILE=$1
UNIQUE='-={GR}=-'
#
if [ -z "$FILE" ]; then
        exit;
fi;
#
for LINE in `sed "s/ /$UNIQUE/g" $FILE`; do
        LINE=`echo $LINE | sed "s/$UNIQUE/ /g"`;
        echo $LINE;
done;
```

As you can see, I've introduced an idea of a UNIQUE variable, something containing a unique combination of characters which I can use to replace spaces in the original string. This variable needs to be a unique combination in a context of your text files, because later we turn the string back into its original version, replacing all the instances of $UNIQUE text with plain spaces.

Since most of the needs of mine required such functionality for a relatively small text files, this rather expensive (in terms of CPU cycles) approach proved to be quite usable and pretty fast.

Update: please see comments to this post for a much better approach to the same problem. Thanks again, Nails!

Here's how my script would work on the already known **/etc/redhat-release** file:

```
bash-3.1$ /tmp/cat.sh /etc/redhat-release
Red Hat Enterprise Linux Client release 5 (Tikanga)
```

Exactly what I wanted! Hopefully this little trick will save some of your time as well. Let me know if you like it or know an even better one yourself!

Related books

If you want to learn more, here's a great book:

Classic Shell Scripting

See also:

- **Using variables in scripts**
- **Unix scripting: time and date**