

comp.unix.shell FAQ – Answers to Frequently Asked Questions Part 2.

Source: <http://unix.derkeiler.com/Newsgroups/comp.unix.shell/2005-06/1079.html>

From: Joe Halpin (j.p.h_at_comcast.net)

Date: 06/28/05

Date: Mon, 27 Jun 2005 21:47:49 -0500

Archive-name: unix-faq/shell/sh

Posting-Frequency: monthly

Version: \$Id: cus-faq-2.html,v 1.3 2005/05/22 12:29:50 jhalpin Exp jhalpin \$

Maintainer: Joe Halpin

Please read the introduction in the first section of this document.

This section assumes you have read that introduction.

=====

11. How do I get the exit code of cmd1 in cmd1|cmd2

First, note that cmd1 exit code could be non-zero and still don't mean an error. This happens for instance in

```
cmd | head -1
```

you might observe a 141 (or 269 with ksh93) exit status of cmd1, but it's because cmd was interrupted by a SIGPIPE signal when "head -1" terminated after having read one line.

To know the exit status of the elements of a pipeline

```
cmd1 | cmd2 | cmd3
```

a. with zsh:

The exit codes are provided in the pipestatus special array.

cmd1 exit code is in `$pipestatus[1]`, cmd3 exit code in

`$pipestatus[3]`, so that `$?` is always the same as

`$pipestatus[-1]`.

b. with bash:

The exit codes are provided in the PIPESTATUS special array.

cmd1 exit code is in `${PIPESTATUS[0]}` , cmd3 exit code in

`${PIPESTATUS[2]}` , so that `$?` is always the same as

```
#{PIPESTATUS: -1}.
```

c. with any other Bourne like shells

You need to use a trick to pass the exit codes to the main shell. You can do it using a pipe(2). Instead of running "cmd1", you run "cmd1; echo \$?" and make sure \$? makes it way to the shell.

```
exec 3>&1
eval `
# now, inside the `...`, fd4 goes to the pipe
# whose other end is read and passed to eval;
# fd1 is the normal standard output preserved
# the line before with exec 3>&1
exec 4>&1 >&3 3>&-
{
  cmd1 4>&-; echo "ec1=$?;" >&4
} | {
  cmd2 4>&-; echo "ec2=$?;" >&4
} | cmd3
echo "ec3=$?;" >&4
`
```

d. with a POSIX shell

You can use this function to make it easier:

```
run() {
  j=1
  while eval "\${pipestatus_}$j+:} false"; do
    unset pipestatus_}$j
    j=$((j+1))
  done
  j=1 com= k=1 l=
  for a; do
    if [ "x$a" = 'x|' ]; then
      com="$com { $1 ""3>&-
        echo "pipestatus_'$j'=$?" >&3
      } 4>&- |"
      j=$((j+1)) l=
    else
      l="$l \"\$k\" ""
    fi
    k=$((k+1))
  done
  com="$com $1"" 3>&- >&4 4>&-
  echo "pipestatus_'$j'=$?"
  exec 4>&1
  eval "$(exec 3>&1; eval "$com")"
  exec 4>&-
```

```
j=1
while eval "\${pipestatus_$j+:} false"; do
  eval "[ \${pipestatus_$j} -eq 0 ]" || return 1
  j=$((j+1))
done
return 0
}
```

use it as:

```
run cmd1 \| cmd2 \| cmd3
exit codes are in $pipestatus_1, $pipestatus_2, $pipestatus_3
```

12. Why do I get "script.sh: not found"

a. While script starts with "#!/bin/sh" (^M issue)

That's the kind of error that occurs when you transfer a file by FTP from a MS Windows machine. On those systems, the line separator is the CRLF sequence, while on unix the line separator is LF alone, CR being just another ordinary character (the problem is that it is an invisible one on your terminal (where it actually moves the cursor to the beginning of the line) or in most text editors or paggers).

So, if a MSDOS line is "#!/bin/sh", when on a Unix system, it becomes "#!/bin/sh<CR>" (other names for <CR> are \r, \015, ^M, <Ctrl-M>).

So, if you run the file as a script, the system will look in /bin for an interpreter named "sh<CR>", and report it doesn't exist.

```
$ sed 'l;d;q' < script.sh
#!/bin/sh\r$
```

shows you the problem (\$ marks the end of line, \r is the CR character).

b. PATH issue

Sometimes a shell is installed someplace other than /bin or /usr/bin. For example, a shell which was not part of the OS installation might be installed into /usr/local/bin. If the script was written on a machine which had ksh located in /usr/bin, but was run on a machine where ksh was located in /usr/local/bin, the shebang line would not resolve correctly.

This is unlikely to occur when using sh. However, if the shell is bash, zsh, et al, it might be installed in different places on different machines.

One way around this is to use the env command in the shebang line. So instead of

```
#!/bin/sh
```

use

```
#!/usr/bin/env sh
```

Of course, env might itself live in some other directory than /usr/bin, but it's not likely.

=====

13. Why doesn't echo do what I want?

See also section 0a "Notes about using echo"

The echo command is not consistent from shell to shell. For example, some shells (bash, pdksh [,:?]) use the following arguments

- n suppress newline at the end of argument list
- e interpret backslash-escaped characters
- E disable interpretation of backslash-escaped characters, even on systems where interpretation is the default.

However, pdksh also allows using \c to disable a newline at the end of the argument list.

POSIX only allows \c to be used to suppress newlines, and doesn't accept any of the above arguments.

ksh88 and ksh93 leave the interpretation of backslash-escaped characters up to the implementation.

[descriptions of behavior of other shells welcome]

In short, you have to know how echo works in any environment you choose to use it in, and its use can therefore be problematic. If available, print(1) or printf(1) would be better.

=====

14. How do I loop through files with spaces in their name?

So, you're going to loop through a list of files? How is this list stored? If it's stored as text, there probably was already an assumption about the characters allowed in a filename. Every character except '\0' (NUL) is allowed in a file path on Unix. So the only way to store a list of file names in a file is to separate them by a '\0' character (if you don't use a quoting mechanism as for xargs input).

Unfortunately most shells (except zsh) and most standard unix text utilities (except GNU ones) can't cope with "\0" characters. Moreover, many tools, like "ls", "find", "grep -l" output a \n separated list of files. So, if you want to postprocess this output, the simpler is to assume that the filenames don't contain newline characters (but beware that once you make that assumption, you can't pretend anymore your code is reliable (and thus can't be exploited)).

So, if you've got a newline separated list of files in a list.txt file, Here are two ways to process it:

1–

```
while IFS= read -r file <&3; do
  something with "$file" # be sure to quote "$file"
done 3< list.txt
(if your read doesn't have the "-r" option, either make another
assumption that filenames don't contain backslashes, or use:
```

```
exec 3<&0
sed 's/\&&/g' < list.txt |
while IFS= read file; do
  something with "$file" <&3 3<&-
done
)
```

2–

```
IFS="
" # set the internal field separator to the newline character
# instead of the default "<space><tab><NL>".

set -f # disable filename generation (or make the assumption that
# filenames don't contain *, [ or ? characters (maybe more
# depending on your shell)).

for file in $(cat < list.txt); do
  something with "$file" # it's less a problem if you forget to
  # quote $file here.
done
```

Now, beware that there are things you can do before building

this list.txt. There are other ways to store filenames. For instance, you have the positional parameters.

with:
set -- /*.txt

you have the list of txt files in the current directory, and no problem with weird characters. Looping through them is just a matter of:

```
for file
do something with "$file"
done
```

You can also escape the separator. For instance, with

```
find . -exec sh -c 'printf %s\n "$1" | sed -n "":1
\${!N;b1
}
s/|/p/g;s/\n/\n/g;p' '{} '{}';
```

instead of

```
find . -print
```

you have the same list of files except that the \n in filenames are changed to "|n" and the "|" to "|p". So that you're sure there's one filename per line and you have to convert back "|n" to "\n" and "|p" to "|" before referring to the file.

15. how do I change my login shell?

See <http://www.faqs.org/faqs/unix-faq/shell/shell-differences>

Unless you have a very good reason to do so, do not change root's default login shell. By "default login shell" is meant the shell recorded in /etc/passwd. Note that "I login as root but don't like the default shell" isn't a good reason.

The default shell for root is one which will work in single user mode, when only the root partition is mounted. This is one of the contexts root works in, and the default shell must accommodate this. So if you change it to a dynamically linked shell which depends on libraries that are not in the root partition, you're asking for trouble.

The safest way of changing root's shell is to login as root and then

```
# SHELL=/preferred/shell; export SHELL
# exec <your preferred shell with login flag>
```

e.g.

```
# SHELL=/usr/bin/ksh; export SHELL
# exec $SHELL -l
```

Another possibility is to add something to root's .profile or .login which checks to see if the preferred shell is runnable, and then execs it. This is more complicated and has more pitfalls than simply typing "exec <shell>" when you login though. For example, one of the libraries that the desired shell relies on might have been mangled, etc. One suggestion that has been made is

```
if [ -x /usr/bin/ksh ]; then
  SHELL=/usr/bin/ksh; export SHELL
  ENV=/root/.kshrc; export ENV
  /usr/bin/ksh -l && exit
fi
```

A safer way is to try to run a command with the preferred shell before you try to exec it. This will lessen the possibility that the shell or one of the libraries it depends on has been corrupted, or that one of the libraries it depends on is not in the available mounted partitions.

```
if [ -x /usr/bin/ksh ]; then
  /usr/bin/ksh -c echo >/dev/null 2>&1
  if [ $? -eq 0 ];then
    SHELL=/usr/bin/ksh; export SHELL
    ENV=/root/.kshrc; export ENV
    /usr/bin/ksh -l && exit
  fi
fi
```

Another common approach is to create another user with UID 0. For example, FreeBSD systems commonly create an account named toor, which can be setup however you like. This bypasses the controversy.

=====

16. When should I use a shell instead of perl/python/ruby/tcl...

a. Portability

In many cases it can't be assumed that perl/python/etc are installed on the target machine. Many customer sites do not allow installation of such things. In cases like this, writing a shell script is more likely to be successful. In the extreme,

writing a pure Bourne shell script is most likely to succeed.

b. Maintainability

If the script is one which serves some important purpose, and will need to be maintained after you get promoted, it's more likely that a maintainer can be found for a shell script than for other scripting languages (especially less used ones such as ruby, rexx, etc).

c. Policy

Sometimes you're just told what to use :-)

=====

17. Why shouldn't I use csh?

<http://www.grymoire.com/Unix/CshTop10.txt>
<http://www.grymoire.com/Unix/Csh.html#uh-0>
<http://www.faqs.org/faqs/unix-faq/shell/csh-why-not/>

=====

18. How do I reverse a file?

Non-standard commands to do so are GNU tac and "tail -r". sed '1!G;h;\$!d' is subject to sed limitation on the size of its hold space and is generally slow.

The awk equivalent would be:

```
awk '{l[n++]=$0}END{while(n-->0)print l[n]}'
```

It stores the whole file in memory.

The best approach in terms of efficiency portability and resource consumption seems to be:

```
cat -n | sort -rn | cut -f2-
```

"cat -n" is not POSIX but appears to be fairly portable. Alternatives are "grep -n '^'", "awk '{print NR,\$0}'". Also, nl can be used as

```
nl -ba -d'
```

i.e. NL as the delimiter.

=====

19. how do I remove the last n lines?

First we need to tell the code how many lines we want to cut from the bottom of a file.

```
X=10
```

Then We can do this:

```
head -n $(( $(wc -l < file) - $X )) file >$$ \
&& cat $$ >file && rm $$
```

The break down:

1) `$(wc -l < file)`

Find out how many lines are in the file. Need to use redirection so wc won't print the file name.

2) `$(($lines_in_file - $X))`

Take the output from step one and do some math to find out how many lines we want to have when all is said and done.

3) `head -n $lines_when_said_and_done file`

extracts all but the unwanted lines from the file, and `>$$` puts those lines into a temp file that has the name of the pid of the current shell.

4) `&& cat $$ > file`

if everything has worked so far then cat the temp file into the original file. This is better than mv or cp because it insures that the permissions of the temp file do not override with the perms of the original file.

5) `&& rm $$`

Remove the temp file.

AWK solutions:

```
awk 'NR<=(count-12)' count="" awk 'END{print NR}' file`" file
```

```
awk 'NR>n{print a[NR%n]} {a[NR%n]=$0}' n=12 file
```

```
awk 'BEGIN{n=12} NR>n{print a[NR%n]} {a[NR%n]=$0}' file
```

Whenever a line is read, the line that came 12 lines ago is printed, and then overwritten with the newly read line, using an rolling array indexed 0..11.

See also question 26. for information about setting awk variables on the command line.

`$SHELL/sed/mv` solutions:

```
L=`wc -l <file`
```

```
DL=`expr $L - 11`
```

```
sed "$DL,\$d" file
```

```
L=`wc -l <file`  
DL=`expr $L - 12`  
sed "${DL}q" file
```

```
sed ""`expr \ `wc -l <file\ ` - 12`q" file
```

```
sed -n -e :a -e '1,12{N;ba' -e '}' -e 'P;N;D' file
```

The last solution is basically same algorithm as the rolling array awk solutions, and shares with them the advantage that the file is only read once – they will even work in a pipe. There may be limitations in sed's pattern space which would make this unusable however.

PERL solution:

```
perl -ne' print shift @x if @x == 12; push @x, $_ ' file
```

Using GNU dd:

```
ls -l file.txt | {  
  IFS=" "  
  read z z z z sz z  
  last=`tail -10 file.txt | wc -c`  
  dd bs=1 seek=`expr $sz - $last` if=/dev/null of=file.txt  
}
```

20. how do I get file size, or file modification time?

If your system has stat(1), use it. On Linux, for example:

```
filesize=$(stat -c %s -- filename)
```

or use cut, awk, etc on the output.

Probably the most portable solution is to use wc

```
filesize=`wc -c < "$file"`
```

ls may be able to tell you what you want to know. From the man page for ls we learn about "ls -l" the file mode, the number of links to the file, the owner name, the group name, the size of the file (in bytes), the timestamp, and the filename. For the file size in human readable formate use the "-h" option.

For example:

```
$ ls -l timeTravel.html  
-rw-rw-r-- 1 user user 20624 Jun 19 2002 timeTravel1.html
```

so to get the file size:

```
$ set -- `ls -l timeTravel1.html`  
$ echo $5  
20624
```

Note that ls doesn't always give the date in the same format. Check the man page for ls on your system if that matters. If you're interested in the file modification time.

Another possibility is to use GNU ls, which has a -T option giving complete time information for the file, including month, day, hour, minute, second and year.

See also GNU find (-printf), GNU stat, GNU date (-r) and zsh stat (+mtime).

On FreeBSD 4, you can use the -IT option to ls(1) to get the full modification time and the -f option to date(1) to parse it, for example:

```
$ FILE=/etc/motd  
$ date -jf%b %d %T %Y +%Y-%m-%dT%T \  
$(ls -IT $FILE|tr -s ' ' \\t|cut -f6-9)  
2003-09-09T16:04:06
```

Adjust syntax as needed if your shell is FreeBSD sh

=====

21. How do I get a process id given a process name? Or, how do I find out if a process is still running, given a process ID?

There isn't a reliable way to do this portably in the shell. Some systems reuse process ids much like file descriptors. That is, they use the lowest numbered pid which is not currently in use when starting a new process. That means that the pid you're looking for is there, but might not refer to the process you think it does.

The usual approach is to parse the output of ps, but that involves a race condition, since the pid you find that way may not refer to the same process when you actually do something with that pid. There's no good way around that in a shell script though, so be advised that you might be stepping into a trap.

One suggestion is to use pgrep if on Solaris, and 'ps h -o pid -C \$STRING' if not, and your ps supports that syntax, but neither of those are perfect or ubiquitous.

The normal solution when writing C programs is to create a pid file, and then lock it with `fcntl(2)`. Then, if another program wants to know if that program is really running, it can attempt to gain a lock on the file. If the lock attempt fails, then it knows the file is still running.

We don't have options in the shell like that, unless we can supply a C program which can try the lock for the script. Even so, the race condition described above still exists.

22. How do I get a script to update my current environment?

Processes in unix cannot update the environment of the process that spawned them. Consequently you cannot run another process normally and expect it to do that, since it will be a child of the running process. There are a couple ways it can be done though.

a. source the script

This means that you use whatever syntax your shell has to read the desired script into the current environment.

In Bourne derived shells (`sh/ksh/bash/POSIX/etc`) the syntax would be

```
$ . script
```

In `csh` type shells this would be

```
$ source script
```

b. use eval

The `eval` command constructs a command by evaluating and then executing a set of arguments. If those arguments evaluate to a shell variable assignment, the current environment will be updated. For example

```
---- exportFoo
#!/bin/ksh
echo export FOO=bar
```

If you run this like

```
eval "`exportFoo`"
```

the value of `FOO` will be set to `'bar'` in the calling shell. Note that the quotes are recommended as they will preserve any whitespace that may be present in the variables

being set.

However, be aware that eval'ing a script written in another shell could turn out to be the wrong thing to do. For example, eval'ing this from a ksh script

```
#!/bin/csh
echo setenv FOO bar
```

Would not do what you expect. It would produce an error, because ksh doesn't have a setenv command.

23. How do I rename *.foo to *.bar?

Naive examples in ksh/bash (which may or may not work many times)

```
$ ls *.foo | while read f;do mv "$f" "${f%.*}.bar"
```

More generically

```
$ ls *.foo | while read f;do mv "$f" `basename "$f" .foo`.bar
```

However, these examples contain a potentially unnecessary use of ls (ie, if the number of files is small enough to not overflow the command line buffer), and will fail if any file names contain a newline, or if there are leading or trailing spaces. An alternative is:

```
for file in *.foo
do
  mv -- "$file" "`basename -- \"$file\" .foo`.bar"
done
```

Also, tests for existence of files should also be incorporated, e.g.:

```
for file in /*.foo
do
  newfile=`basename "$file" .foo`.bar
  [ -f "$file" ] || continue
  [ -f "$newfile" -o -d "$newfile" ] && continue
  mv "$file" "$newfile"
done
```

In some linux distributions you may be able to use the rename command

```
$ rename .foo .bar *
```

If not (Debian, for one, comes with a perl version of rename that won't work with that command line) try

```
$ rename 's/.foo/.bar/' *.foo
```

More options, and much more discussion about this, is available from <http://www.faqs.org/faqs/unix-faq/faq/part2/section-6.html>

Note that for file specifications which don't match existing files, the shell usually responds with something like "ls: *.foo: No such file or directory", which will mess up your processing of file names. One possibility is

```
#!/bin/sh
set x [*].foo /*.foo
case "$2$3" in
  "[*].foo/*.foo" ;;
  *)
    shift 2
    for file
    do
      repl=`basename "$file" .foo`.bar
      mv "$file" "$repl"
    done;;
esac
```

Except that contrary to (zsh) mmv or zmv it doesn't check for file overwriting and fails for filenames with NLs before the "." and doesn't handle dotfiles.

24. How do I use shell variables in awk scripts

Short answer = either of these, where "svar" is a shell variable and "avar" is an awk variable:

```
awk -v avar="$svar" '... avar ...' file
awk 'BEGIN{avar=ARGV[1];ARGV[1]=""}... avar ...' "$svar" file
```

depending on your requirements for handling backslashes and handling ARGV[] if it contains a null string (see below for details).

Long answer = There are several ways of passing the values of shell variables to awk scripts depending on which version of awk (and to a much lesser extent which OS) you're using. For this discussion, we'll consider the following 4 awk versions:

```
oawk (old awk, /usr/bin/awk and /usr/bin/oawk on Solaris)
nawk (new awk, /usr/bin/nawk on Solaris)
sawk (non-standard name for /usr/xpg4/bin/awk on Solaris)
```

gawk (GNU awk, downloaded from <http://www.gnu.org/software/gawk>)

If you wanted to find all lines in a given file that match text stored in a shell variable "svar" then you could use one of the following:

- a) `awk -v avar="$svar" '$0 == avar' file`
- b) `awk -vavar="$svar" '$0 == avar' file`
- c) `awk '$0 == avar' avar="$svar" file`
- d) `awk 'BEGIN{avar=ARGV[1];ARGV[1]=""}$0 == avar' "$svar" file`
- e) `awk 'BEGIN{avar=ARGV[1];ARGC--}$0 == avar' "$svar" file`
- f) `svar="$svar" awk 'BEGIN{avar=ENVIRON["svar"]}$0 == avar' file`
- g) `awk '$0 == "$svar"' file`

The following list shows which version is supported by which awk on Solaris (which should also apply to most other OSs):

```
oawk = c, g
nawk = a, c, d, f, g
sawk = a, c, d, f, g
gawk = a, b, c, d, f, g
```

Notes:

- 1) Old awk only works with forms "c" and "g", both of which have problems.
- 2) GNU awk is the only one that works with form "b" (no space between "-v" and "var="). Since gawk also supports form "a", as do all the other new awks, you should avoid form "b" for portability between newer awks.
- 3) In form "c", ARGV[1] is still getting populated, but because it contains an equals sign (=), awk changes it's normal behavior of assuming that arguments are file names and now instead assumes this is a variable assignment so you don't need to clear ARGV[1] as in form "d".
- 4) In light of "3)" above, this raises the interesting question of how to pass awk a file name that contains an equals sign – the answer is to do one of the following:
 - i) Specify a path, e.g. for a file named "abc=def" in the current directory, you'd use:

```
awk '!..! ./abc=def
```

Note that that won't work with older versions of gawk or with sawk.

- ii) Redirect the input from a file so it's opened by the shell rather than awk having to parse the file name as an argument and then open it:

```
awk '...' < abc=def
```

Note that you will not have access to the file name in the `FILENAME` variable in this case.

- 5) An alternative to setting `ARGV[1]=""` in form "d" is to delete that array entry, e.g.:

```
awk 'BEGIN{avar=ARGV[1];delete ARGV[1]}$0 == avar' "$svar" file
```

This is slightly misleading, however since although `ARGV[1]` does get deleted in the `BEGIN` section and remains deleted for any files that precede the deleted variable assignment, the `ARGV[]` entry is recreated by awk when it gets to that argument during file processing, so in the case above when parsing "file", `ARGV[1]` would actually exist with a null string value just like if you'd done `ARGV[1]=""`. Given that it's misleading and introduces inconsistency of `ARGV[]` settings between files based on command-line order, it is not recommended.

- 6) An alternative to setting `svar="$svar"` on the command line prior to invoking awk in form "f" is to export `svar` first, e.g.:

```
export svar  
awk 'BEGIN{avar=ENVIRON["svar"]} $0 == avar' file
```

Since this forces you to export variables that you wouldn't normally export and so risk interfering with the environment of other commands invoked from your shell, it is not recommended.

- 7) When you use form "d", you end up with a null string in `ARGV[1]`, so if at the end of your program you want to print out all the file names then instead of doing:

```
END{for (i in ARGV) print ARGV[i]}
```

you need to check for a null string before printing, or store `FILENAMES` in a different array during processing. Note that the above loop as written would also print the script name stored in `ARGV[0]`.

- 8) When you use form "a", "b", or "c", the awk variable assignment gets processed during awk's lexical analysis stage (i.e. when the internal awk program gets built) and any backslashes present in the shell variable may get

expanded so, for example, if svar contains "hi\there"
then avar could contain "hi<tab>there" with a literal tab
character. This behavior depends on the awk version as
follows:

oawk: does not print a warning and sets avar="hi\there"
sawk: does not print a warning and sets avar="hi<tab>here"
nawk: does not print a warning and sets avar="hi<tab>here"
gawk: does not print a warning and sets avar="hi<tab>here"

If the backslash precedes a character that has no
special meaning to awk then the backslash may be discarded
with or without a warning, e.g. if svar contained "hi\john"
then the backslash precedes "j" and "\j" has no special
meaning so the various new awks each would behave differently
as follows:

oawk: does not print a warning and sets avar="hi\john"
sawk: does not print a warning and sets avar="hi\john"
nawk: does not print a warning and sets avar="hijohn"
gawk: prints a warning and sets avar="hijohn"

- 9) None of the awk versions discussed here work with form "e" but
it is included above as there are older (i.e. pre-POSIX) versions
of awk that will treat form "d" as if it's intended to access a
file named "" so you instead need to use form "e". If you find
yourself with that or any other version of "old awk", you need
to get a new awk to avoid future headaches and they will not be
discussed further here.

So, the forms accepted by all 3 newer awks under discussion (nawk,
sawk, and gawk) are a, c, d, f, and g. The main differences between
each of these forms is as follows:

	BEGIN		files		requires
			accepts		expands
			null		
	avail		set		access
			backslash		backslash
			ARGV[]		
a)		y		all	
		n		n	
c)		n		sub	
		n		n	
d)		y		all	
		n		n	
f)		y		all	
		y		n	
g)		y		all	
		n		y	
		n/a		n	

where the columns mean:

BEGIN avail = y: variable IS available in the BEGIN section
BEGIN avail = n: variable is NOT available in the BEGIN section

files set = all: variable is set for ALL files regardless of command–line order.

files set = sub: variable is ONLY set for those files subsequent to the definition of the variable on the command line

requires access = y: variable DOES need to be exported or set on the command line

requires access = n: shell variable does NOT need to be exported or set on the command line

accepts backslash = y: variable CAN contain a backslash without causing awk to fail with a syntax error

accepts backslash = n: variable can NOT contain a backslash without causing awk to fail with a syntax error

expands backslash = y: if the variable contains a backslash, it IS expanded before execution begins

expands backslash = n: if the variable contains a backslash, it is NOT expanded before execution begins

null ARGV[] = y: you DO end up with a null entry in the ARGV[] array

null ARGV[] = n: you do NOT end up with a null entry in the ARGV[] array

For most applications, form "a" and "d" provide the most intuitive functionality. The only functional differences between the 2 are:

- 1) Whether or not backslashes get expanded on variable assignment.
- 2) Whether or not ARGV[] ends up containing a null string.

so which one you choose to use depends on your requirements for these 2 situations.

=====

25. How do I get input from the user with a timeout?

In bash or ksh93 you can use the read built–in with the "–t" option.

In zsh, use the zsh/zselect module.

You can also use your terminal capability to do that.

```
{
  s=$(stty –g)
  stty –icanon min 0 time 100
  var=$(head –n 1)
  stty "$s"
}
```

For a 10 second timeout (reset at each key press).

=====
26. How do I get one character input from the user?

In bash this can be done with the "-n" option to read.

In ksh93 it's read -N

In zsh it's read -k

More portably:

```
OLDSTTY=$(stty -g) # save our terminal settings
```

```
stty cbreak # enable independent processing of each input character
```

```
ONECHAR=$(dd bs=1 count=1 2>/dev/null) # read one byte from standard in
```

```
stty "$OLDSTTY" # restore the terminal settings
```

Use the `something` format if your shell doesn't understand \$(something). This reads from standard input, which may or may not be desirable. If you want to read from the terminal regardless of where standard input is, add "if=\$(tty)" to the dd command.

=====
27. why isn't my .profile read?

~/.profile is only read for login shells. In short if you don't see a login prompt then your ~/.profile isn't being read. You can fix this by either porting all of the things in your ~/.profile to /etc/profile or your shells rc script such as ~/.bashrc or ~/.zshrc.

You may have to set the ENV variable in your login shell to get the .*rc shell read. See the man page for your shell to understand how it works.

=====
28. why do I get "[5" not found in "[\${1} -eq 2]"?

Because you didn't RTFM :-)

"[" is an alias for the "test" command. As such, it's called by a script like any other command (this applies even if test is builtin). Since the command line uses spaces to separate a command from its arguments, you have to put a space between '[' and its argument. So:

```
$ [ -f xxx ] isn't the same as
```

```
$ [-f xxx ]
```

In the latter case, the shell will think that "[–f" is the command, not "[" with arguments "–f xxx]

29. How do I exactly display the content of \$var (with a \n appended).

A: on POSIX systems or with shells with builtin printf (bash2, ksh93, zsh4.1, dash...)

```
printf '%s\n' "$var"
```

(except for memory/environment full errors, should be expected to work at least if \$var is not longer than LINE_MAX (supposed to be at least _POSIX2_LINE_MAX == 2048), no hardcoded limits in zsh/ksh/bash/dash builtins)

ksh, zsh:
print -r -- "\$var"

zsh:
echo -E - "\$var"

Other bourne like shells:

```
cat << EOF  
$var  
EOF  
(creates a temporary file and forks a process)
```

```
expr "x$var" : 'x\(.*)'
```

(limited to 126 characters with some exprs, may return a non-null exit code).

With ash:
(unset a; \${a?\$var}) 2>&1

```
printf %s "$var" # posix  
print -rn -- "$var" # zsh/ksh  
echo -nE - "$var" # zsh
```

```
awk 'NR>1{print ""}{printf("%s",$0)}' << EOF  
$var  
EOF
```

30. How do I split a pathname into the directory and file?

The most portable way of doing this is to use the external commands dirname(1) and basename(1), as in

```
pathname='/path/to/some/file'  
dir=`dirname "$pathname"`  
file=`basename "$pathname"`
```

However, since this executes an external command, it's slower than using shell builtins (if your shell has them). For ksh, bash, zsh and POSIX shells the following will do the same thing more efficiently:

```
pathname=/path/to/some/file  
file=${pathname##*/}
```

To get the directory using the shell builtin, you should first ensure that the path has a '/' in it.

```
case $pathname in  
  */*) dir=${pathname%/*};;  
  *) dir=""  
esac
```

In zsh, (abd csh, tcsh), you have

```
${pathname:h} (head) ${pathname:t} (tail).
```

=====

31. How do I make an alias take an argument?

In Bourne-derived shells aliases cannot take arguments, so if you need to be able to do that, define a shell function rather than

n