CLI constructs and useful info

The purpose of this article is not to be a CLI tutorial, but rather to be an exposition of common constructs used in shell scripting for efficiently achieving a goal. There are also sections which simply help one understand a certain topic.

Constructs

rev | cut | rev

It is often useful to reverse a string and then use cut. For example, take a Slackware package and get its name:

```
echo dejavu-fonts-ttf-2.33-noarch-1 | rev | cut -d - -f 1-3 --complement | rev
ls -1 /var/log/packages | rev | cut -d - -f 1-3 --complement | rev
```

Or if you wanted to get the full path of a file, minus the suffix.

```
echo /proc/config.gz | rev | cut -d. -f1 --complement | rev
```

replace a suffix

Say you wanted to make a video conversion script, and you needed to change the suffix.

```
input=test.mkv
output="$(basename "$input" .mkv).avi"
```

find | xargs

This is a special interaction between find and xargs that allows one to deal with spaces in file names. It is very fast because many commands like `rm`, `rmdir`, and `shred` take multiple file inputs on the command line. A generic construct is something like:

```
find . -type f -print0 | xargs -0 "$command"
```

You can replace `$command` with whatever command you need to run on the files as long as it supports multiple file input. If you have a list of files you can still preserve spaces:

```
tr '\n' '\0' < "$file" | xargs -0 "$command"
```

comm before and after

This construct is useful for package management applications. From the comm man page:

```
With  no  options,  produce  three-column  output.
Column one contains lines unique to FILE1,
column two contains lines unique to FILE2, and
column three contains lines common to both files.
```

The options `-1 -2 -3` suppress the respective columns. Say you wanted to log files that were added to `/usr` after running command `$1`:

```
# before, make install, after
find /usr > "$tmp/before"
$1
find /usr > "$tmp/after"

# sort
sort "$tmp/before" > "$tmp/before-sorted"
sort "$tmp/after" > "$tmp/after-sorted"

# create log
comm -13 "$tmp/before-sorted" "$tmp/after-sorted" > "$log/$name"
```

Note that comm requires sorted files. Here `-1` suppresses lines unique to before, `-3` suppresses lines present in both files, so you are left with column 2 which contains files unique to after i.e. the files added. Many people would like to use diff to compare files, but it's mostly for creating patches.

while read line

This construct is common and is useful for reading files or stdin one line at a time. Here is an example that can be used to concatenate split files in order:

```
base="$(echo "$@" | rev | cut -d. -f1 --complement | rev)"

ls -1 "$base".* | sort -V | while read line
do
    cat "$line" >> "$base"
done
```

Also note that `sort -V` is a version sort and is useful in cases where `ls` sorts suffixes incorrectly. The usual way to prevent this is to name numbered suffixes with `0` padding like `file.001`, but it may overflow and this is why `sort -V` is useful.

for i in

Here is an example for extracting all rpms in a directory:

```
for i in *.rpm
do
    rpm2cpio "$i" | cpio -id --quiet
done
```

You can also use `seq` to make `i` a loop counter:

```
for i in $(seq 1 100)
do
    echo "$i"
done
```

Note that there are no quotes around `$(seq)` because otherwise it would quote the entire expanded number sequence and that wouldn't work right.

External Links

- http://www.commandlinefu.com/commands/browse

Quoting

Quoting may seem complicated, and reasons for it obscure, but there is a purpose to it and it is not that complicated.

Double quoting

The reason for double quoting is to preserve spaces, like spaces in file names. Double quoting a variable or command substitution makes it into a single argument. An example:

```
bash-4.2$ ls
file with spaces.txt  filewithoutspaces.txt
bash-4.2$ rm -f file with spaces.txt
bash-4.2$ ls
file with spaces.txt  filewithoutspaces.txt
bash-4.2$ rm -f "file with spaces.txt"
bash-4.2$ ls
filewithoutspaces.txt
bash-4.2$ rm -f filewithoutspaces.txt
bash-4.2$ ls
bash-4.2$
```

Clearly you need to quote a file with spaces. You could use single quotes here, because no variables

were inside the quotes. You should not quote in this case:

```
bash-4.2$ for i in $(seq 1 10); do printf "$i "; done; echo;
1 2 3 4 5 6 7 8 9 10
bash-4.2$ for i in "$(seq 1 10)"; do printf "$i "; done; echo;
1
2
3
4
5
6
7
8
9
10
bash-4.2$
```

Nor should you quote in any case where a command requires multiple variables and you give them to it inside one quoted variable. A quoted variable is then taken as the only argument, rather than multiple arguments. An example:

```
bash-4.2$ ls
file with spaces.txt  filewithoutspaces.txt
bash-4.2$ file1="file with spaces.txt"
bash-4.2$ file2="filewithoutspaces.txt"
bash-4.2$ rm -f "$file1 $file2"
bash-4.2$ ls
file with spaces.txt  filewithoutspaces.txt
bash-4.2$ rm -f "$file1" "$file2"
bash-4.2$ ls
bash-4.2$
```

Also note that you can and should quote within command substitutions, as shown by the `replace a suffix` example above and:

```
mkdir "$(basename "$(pwd)")"
```

This makes a directory within the current directory called the same name as the current directory. If pwd expands into something with spaces, the command will work.

Single quoting

The reason for single quoting is to escape special characters from the shell, while passing them to a command so it can use them. You should use single quotes for every argument passed to another program that contains shell characters to be interpreted by that program and **NOT** by the shell. Example:

```
bash-4.2$ find -name *.txt
./list.txt
bash-4.2$ find -name '*.txt'
./list.txt
./results/002.txt
./results/006.txt
./results/013.txt
./results/wipe.txt
bash-4.2$
```

Here the shell expands * before find sees it. You should single quote input to awk, `find`, sed, and `grep`, as each of these uses special characters that overlap with the shells', and thus they must be protected from shell expansion.

External Links

- http://www.grymoire.com/Unix/Quote.html

Regular expressions

Basic

- `.` matches any single character.
- \ escapes the next character.

Remember to escape the `.` using `\.` if you want an actual `.`

```
bash-4.2$ cat test.txt
testtxt
test.txt
bash-4.2$ sed 's/.txt//g' test.txt
tes
test
bash-4.2$ sed 's/\.txt//g' test.txt
testtxt
test
```

- `[]` is a class and matches anything inside the brakets for a single character. Examples:
  - `[Yy]` matches Y or y.
  - `[a-z0-9]` includes a range, and in this case matches a through z and 0 through 9.
  - `[^a-z]` negates the range, so in this case it matches anything but a through z.
- ^ matches the beginning of a line. Example: ^a matches an a at the beginning of a line.
- $ matches the end of a line. Example: a$ matches an a at the end of a line.
- \< matches the beginning of a word. Example: \<a matches an a at the beginning of a word.
- \> matches the end of a word. Example: a\> matches an a at the end of a word.
  - Example: `\<[tT]he\>` matches the word `the` or The.
- * matches any number of the previous character or nothing = no character. Example: `[0-9]*` which will match any number of numbers. `.*` matches any number of anything.

Extended regular expressions

The following must be supported by the program for them to work. For example for grep you must run `egrep` or `grep -E`.

- + matches any number of the previous character, like *, but there must be at least one to match, so it will not match nothing or no character.
- ? makes the previous character optional (it can be missing), and is matched at most once.
- `(|)` acts like an OR statement. Example: `(it|her|this)` matches any of those words.
- `a{3}` matches aaa = 3 a's.
- `a{4,8}` matches an a at least 4 times and at max 8 times, so aaaa, aaaaa, aaaaaa, aaaaaaa, and aaaaaaaa.
- `{0,}` = *
- `{1,}` = +
- `{,1}` = ?

External Links

- http://www.grymoire.com/Unix/Regular.html
- http://www.regular-expressions.info/

Useful commands and info

stat

Stat is the most accurate way to determine:

- File size in bytes:

```
stat -c '%s' file.txt
```

- File permissions in octal:

```
stat -c '%a' file.txt
```

awk variable defaults

An important point is that awk variables are set to zero by default. This may cause problems in some situtations. Example:

```
echo -ne '-321\n-14\n-1\n-34\n-4\n' | awk 'BEGIN{max=""}{if ($1 > max) max=$1; if ($1 < min) min=$1;}END{print min"\t"max}'
```

This works properly because max is set to an empty string and thus has a lower value than any

number. Try removing the BEGIN clause and see what happens. Also note that adding `min=""` to the BEGIN clause fails as well.

no data directory test

You can use this to test if a directory contains no data. For example, it will say `0` if the directory only contains empty files and directories = no data.

```
du -s directory
```

cmp

This can compare two files byte by byte, and can be more useful than checksums. For example, after you burn a CD/DVD, you can run:

```
cmp slackware.iso /dev/sr0
```

It should say the following if the disk burned correctly:

```
cmp: EOF on slackware.iso
```

shell math

Remember that shell utilities like `let` and `expr` only do integer math. For floating point use either `bc` or awk.

shell GUI

There are numerous programs that allow you to create GUIs from a shell script.

- Xdialog is fully dialog compatible.
- A function library that increases shell GUI portability.
- Gtkdialog has advanced features for customized GUIs.

External Links

- Guide to awk, bash, sed, find, and more
- Advanced bash guide
- Cheat sheets on Linux CLI programs, the main site is also useful
- Guide to file permissions

Sources

- I quoted `man comm`
- I used `man grep` for the regex section.
- Written by H_TeXMeX_H

howtos, software, author htexmexh