Dissect a bad oneliner

```
$ ls *.zip | while read i; do j=`echo $i | sed 's/.zip//g'`; mkdir $j
; cd $j; unzip ../$i; cd ..; done
```

This is an actual one-liner someone asked about in `#bash`. **There are several things wrong with it. Let's break it down!**

```
$ ls *.zip | while read i; do ...; done
```

(Please read http://mywiki.wooledge.org/ParsingLs (http://mywiki.wooledge.org/ParsingLs).) This command executes `ls` on the expansion of `*.zip`. Assuming there are filenames in the current directory that end in '.zip', ls will give a human-readable list of those names. The output of ls is not for parsing. But in sh and bash alike, we can loop safely over the glob itself:

```
$ for i in *.zip; do j=`echo $i | sed 's/.zip//g'`; mkdir $j; cd $j;
unzip ../$i; cd ..; done
```

Let's break it down some more!

```
j=`echo $i | sed 's/.zip//g'` # where $i is some name ending in '.zi
p'
```

The goal here seems to be get the filename without its `.zip` extension. In fact, there is a POSIX®-compliant command to do this: `basename` The implementation here is suboptimal in several ways, but the only thing that's genuinely error-prone with this is "`echo $i`". Echoing an *unquoted* variable means wordsplitting will take place, so any whitespace in `$i` will essentially be normalized. In `sh` it is necessary to use an external command and a subshell to achieve the goal, but we can eliminate the pipe (subshells, external commands, and pipes carry extra overhead when they launch, so they can really hurt performance in a loop). Just for good measure, let's use the more readable, modern `$()` construct instead of the old style backticks:

```
sh $ for i in *.zip; do j=$(basename "$i" ".zip"); mkdir $j; cd $j; u
nzip ../$i; cd ..; done
```

In Bash we don't need the subshell or the external basename command. See Substring removal with parameter expansion:

```
bash $ for i in *.zip; do j="${i%.zip}"; mkdir $j; cd $j; unzip ../$i
; cd ..; done
```

Let's keep going:

```
$ mkdir $j; cd $j; ...; cd ..
```

As a programmer, you **never** know the situation under which your program will run. Even if you do, the following best practice will never hurt: When a following command depends on the success of a previous command(s), check for success! You can do this with the " `&&` " conjunction, that way, if the previous command fails, bash will not try to execute the following command(s). It's fully POSIX®. Oh, and remember what I said about wordsplitting in the previous step? Well, if you don't quote `$j`, wordsplitting can happen again.

```
$ mkdir "$j" && cd "$j" && ... && cd ..
```

That's almost right, but there's one problem – what happens if `$j` contains a slash? Then `cd ..` will not return to the original directory. That's wrong! `cd -` causes cd to return to the previous working directory, so it's a much better choice:

```
$ mkdir "$j" && cd "$j" && ... && cd -
```

(If it occurred to you that I forgot to check for success after cd -, good job! You could do this with `{ cd - || break; }`, but I'm going to leave that out because it's verbose and I think it's likely that we will be able to get back to our original working directory without a problem.)

So now we have:

```
sh $ for i in *.zip; do j=$(basename "$i" ".zip"); mkdir "$j" && cd "
$j" && unzip ../$i && cd -; done
```

```
bash $ for i in *.zip; do j="${i%.zip}"; mkdir "$j" && cd "$j" && unz
ip ../$i && cd -; done
```

Let's throw the `unzip` command back in the mix:

```
mkdir "$j" && cd "$j" && unzip ../$i && cd -
```

Well, besides word splitting, there's nothing terribly wrong with this. Still, did it occur to you that unzip might already be able to target a directory? There isn't a standard for the `unzip` command, but all the implementations I've seen can do it with the -d flag. So we can drop the cd commands entirely:

```
$ mkdir "$j" && unzip -d "$j" "$i"
```

```
sh $ for i in *.zip; do j=$(basename "$i" ".zip"); mkdir "$j" && unzi
p -d "$j" "$i"; done
```

```
bash $ for i in *.zip; do j="${i%.zip}"; mkdir "$j" && unzip -d "$j"
"$i"; done
```

There! That's as good as it gets.

Discussion

Michael Shigorin (http://www.altlinux.org), 2012/01/11 10:55 ()

Thanks, nice walkthrough :) I'd only stress proper quoting a bit more, not introducing any unquoted variable expansions myself.

Eduardo Bustamante (http://dualbus.com/), 2012/06/18 01:04 ()

I'd like to address some issues I noticed in the dissection.

First, in the section that states: ``but the only thing that's genuinely error-prone with this is "echo $i". . I think ``sed 's/.zip*g''' is also error prone. Let me explain it. The dot is a RE meta-character, which matches *anything*. So, it will match things like azip, bzip & czip. Also, if the goal is to strip the extension, it will require some anchoring (i.e. s/\.zip$*); or else, it will remove more than just the extension. If that anchor is used, there's no need for the `g' flag. I'm not stating that using sed is the way to go; I'm merely remarking on its usage in that one-liner. Also, echo has multiple incompatible implementations, and using it to print an arbitrary string is risky, since that string can take the form of an option (-e or -n for example). There's no way to avoid this, like using `–', since echo will just print it. Its replacements are printf (printf '%s\n' "$i") or using the here-string syntax in bash («< "$i"). The next thing to note is in the differentiation between sh and bash regarding the ${i%.zip} expansion. The ${name%foo} expansion is standardized in POSIX, so it's safe to use it in sh also. And it's clearly simpler, since you can do just ${i%.*}. And the last thing. It can be made to work with other casings of .zip, like .Zip, .ZIP, and all the possible permutations, using *.[Zz] [Ii][Pp] as the pattern, or just using shopt -s nocaseglob.