Austin Tripp's website

A Quick Tutorial on Bash Quotes

Austin Tripp — 5 years ago

Today I learned **way** more about quotations in bash than I ever thought I needed to know. I thought I would highlight the interesting use case that I discovered, which requires some special trickery to write a script that executes arbitrary commands. First, let's quickly review some facts about bash quotes.

Quick review of bash quotes (skip this if you're an expert)

When you type a command into bash, it evaluates any special characters or sequences before executing the command. As a basic example, consider the following commands[1]:

```
$ echo $HOME
/home/austin
$ python -c print(7)
bash: syntax error near unexpected token `('
```

The first fragment uses the command `echo`, which naturally echos the argument to feed to it. The argument fed to it is $HOME, which due to the dollar sign is interpreted as a bash variable. Therefore it evaluates the contents of this variable, which is my home directory (/home/austin).

The second example runs python, using the `-c` command to execute python code passed as a string. `print(7)` is a completely valid python command that prints the number 7, however parentheses are special characters in bash. So, before that string makes it to the `python` command, bash attempts to evaluate it, determines that it is invalid bash syntax (since it's not bash it's python), and throws an error. Obviously that's not what that program is supposed to do.

Hence the purpose of quotes: they are used to prevent bash from evaluating text that you would like to leave as a string.

Single quotes

The most basic kind of quote is the single quote (\'). This quote completely prevents bash from evaluating a command. Let's look at a few examples.

```
$ echo '$HOME'
$HOME
```

This command literally prints the string "$HOME", since nothing was evaluated. Now is also maybe a good time to point out that the first $ character on the first line is the shell prompt (the last character on the line where you type in commands), so it isn't actually part of the command. Another example:

```
$ 'echo $HOME'
echo $HOME: command not found
```

Putting quotes around the whole command prevents bash from parsing it at all, not even breaking up the input into a command and an argument. So it just assumes that the literal string "echo $HOME" is a command, and doesn't know what command that is, so it gives an error. One final variant would be:

```
$ 'echo' $HOME
/home/austin
```

This command does the same thing as the initial command: it doesn't evaluate echo (which bash wouldn't modify anyways), then evaluates `$HOME` since that wasn't in quotes.

Moving onto the second example, we can make the python code run directly by enclosing the program in quotes.

```
$ python -c 'print(7)'
7
```

Double Quotes

The double quote (\") is also a valid quote in bash, but it works a bit differently. Think of it as a *selective quote*: it lets bash evaluate some things but not others. As explained in more detail in the bash manual, this quote stops all evaluations except for $, \, and `. For the examples above:

```
$ echo "$HOME"
/home/austin
```

As stated, since $HOME has a \$ the quotes don't stop bash from evaluating it.

```
$ "echo $HOME"
bash: echo /home/austin: No such file or directory
```

This one is funny: bash evaluates `$HOME` since it has a \$, but then doesn't evaluate the expression as a whole and treats it as a command. Although for some reason it treats it as a file instead of a command, and says it can't find the file.

```
$ "echo" $HOME
/home/austin
```

Of course, this one does the same thing as the single quotes case.

With the python example, single and double quotes make no different:

```
$ python -c "print(7)"
7
```

This is because the double quotes prevent the evaluation of the parentheses.

Quick recap

So essentially: - Without quotes, bash tries to evaluate all special characters - Single quotes (\') prevent all evaluation - Double quotes (\") prevent most evaluation, but notably not the evaluation of variables

Nested Commands: the case of eval

Where things get weird is when you need to nest commands. A particularly nasty command is `eval`, which evaluates arbitrary bash commands. In the examples below, it doesn't really make sense why you would need to use `eval`, but ignore that for now: a real example will come at the end.

Let's go back to our first example:

```
$ eval echo $HOME
/home/austin
```

Pretty much what you would expect: the variable `$HOME` gets replaced. Now what if we wanted to use `eval` to print the literal string `$HOME`? From above, we can put `$HOME` in single quotes:

```
$ eval echo '$HOME'
/home/austin
```

However, this doesn't work. Why? Because nested commands effectively get evaluated twice. Just look at the following example:

```
$ echo echo '$HOME'
echo $HOME
```

The evaluation procedure effectively has 2 stages: 1. Run through the string and do substitutions. In this case, due to the single quotes the part `$HOME` is just treated as the literal string "\$HOME". 2. Execute the command, which here is to echo the string `echo $HOME`

So in the case of `eval`, it bash first parses the command, performing no substitution for `$HOME`, and then it evaluates the resulting command, since that's what `eval` does. In this case, the command to be evaluated is the string `echo $HOME`, which as shown in the very first example above results in `$HOME` being substituted.

So, working backwards, the solution is to somehow pass the string "`echo '$HOME'`" to `eval`. A natural approach to do this would be to quote the entire phrase you don't want bash to evaluate, so we will place the whole phrase `echo '$HOME'` in single quotes.

```
$ eval 'echo '$HOME''
/home/austin
```

It didn't work! What happened? This one took me a long time to figure out, but the answer is that bash parses the string from left to right, not inside to outside like a nested function call in most programming language. Let's try it without the `eval`:

```
$ 'echo '$HOME''
bash: echo /home/austin: No such file or directory
```

This is the same result as the command `"echo $HOME"` from earlier! Evaluating literally left to right, it actually sees it as the string "`echo`" (interpreted literally so including the space), the variable `$HOME`, which has no quotes beside it so it is evaluated to /home/austin, and finally an empty string between 2 quotes. It then concatenates the two strings "`echo`" and "`/home/austin`" and treats it as a single string, therefore giving the same result as using double quotes from earlier. So the double quoting simply can't be done this way.

In the end, I found two solutions to this problem: showing them both here.

```
$ eval echo \''$HOME'\'
$HOME
$ eval echo "'"'$HOME'"'"
$HOME
```

Both solutions take advantage of the fact that when two strings are beside each other, they are concatenated. A quick example:

```
$ echo "hel""lo"
hello
$ echo hel"lo"
hello
```

Here, the two strings "hel" and "lo" are simply concatenated when executing the command. The second example clarifies that both strings do not have to be in quotes. The two `eval` commands above work in much the same way. 1. The string `echo` is concatenated with the escaped literal `\'`, which is taken to simply be a single quote string due to the backslash. This is concatenated with `'$HOME'`, which just gets evaluated to the string `$HOME`, and finally another literal single quote is appended. The result of the evaluation is that `eval` is passed the literal string `echo '$HOME'` as desired. 2. Much like the first, except that instead of escaping the single quotes they are enclosed in double quotes, since they block the evaluation of the single quotes (remember the only exceptions to double quotes are $, \, and `, not \'). So the end result is the same.

Both of these solutions look pretty gross, but to put it in perspective, imagine what you have to do to use a doubly nested `eval`! To save you the trouble of imagining, you have to do this:

```
$ eval eval echo \\\'\''$HOME'\'\\\'
$HOME
```

Figuring it out is left as an exercise to the reader (try replacing the first `eval` with `echo` to see how all the escaped characters are evaluated).

The script

So, what exactly was the use case that made me encounter a weird problem like this? Essentially, at the time of writing this post I was working at a research institute with a computing cluster than ran the job manager slurm. The basic workflow of slurm is that you write a script that runs your job, then you submit the script to a job manager. Naively, you need one bash script for each program you run, but I was running a lot of programs with similar setup, so I wanted to just write one script. Eventually I came up with something like this:

```bash
#!/bin/bash
#
#SBATCH --job-name=test
#SBATCH --output=res.txt
#
#SBATCH --ntasks=1
#SBATCH --time=10:00

eval "$@"
```

This clever script just calls `eval` on all the arguments passed to it (represented by the variable `$@`). The double quotes are essentially a safety measure (it is generally recommended to quote all variables in double quotes). Another nice part of it is that it can be called with either `source` to run it locally or `sbatch` to submit it to the job scheduler.

The script works great when you are doing something simple. For example:

```
$ source script.sh pwd
/home/austin
```

This works because `pwd` is a nice simple command: no weird characters. What if I want to run a python script? Let's try the following:

```
$ source script.sh python -c "print(7)"
bash: eval: line 9: syntax error near unexpected token `('
bash: eval: line 9: `python -c print(7)'
$ source script.sh 'python -c "print(7)"'
7
```

The first example fails because the quotes are "used up" on the initial evaluation, so there are no quotes when `eval` is called in the script. The second example avoids this by putting the whole thing in single quotes, thus delaying the evaluation until `eval` is called in the script.

But what if you wanted to print a more complicated string, like "Only $1? Let\'s go!". This string has 3 challenging symbols: "\$", "\'", and "!". Let's try to print it. Of course unlike printing 7, we want to print a string, so we will need to put an additional set of quotes around the string. Python allows strings to be used with single or double quotes, but unfortunately using either will cause the number of single quotes to be imbalanced. Furthermore, the "$1" and the "!" need to be escaped with single quotes or they will evaluate to something. There are probably multiple solutions to this, but what I ended up doing was:

```
$ source script.sh python -c \''print('\"Only '$1'? Let\'\\\'\'s go\!\"')'\'
Only $1? Let's go!
```

To be honest I hate quotation marks now.

1. Evaluated in Ubuntu terminal running bash. ↩

frontpage programming

Previous post Next post

Email | GitHub | Scholar | Bluesky | Twitter/X