Quoting in Linux

There are three types of quotes used by the Bash shell: single quotes ('), double quotes (") and back quotes (`). These quotes have special features in the Bash shell as described below.

To understand single and double quotes, consider that there are times that you don't want the shell to treat some characters as *special*. For example, the * character is used as a *wildcard*. What if you wanted the * character to just mean a literal asterisk?

- Single ' quotes prevent the shell from "interpreting" or expanding all special characters. Often single quotes are used to protect a string (a sequence of characters) from being changed by the shell, so that the string can be interpreted by a command as a parameter to affect the way the command is executed.
- Double " quotes stop the expansion of glob characters like the asterisk (*), question mark (?), and square brackets ([]). Double quotes *do* allow for both variable expansion and command substitution (see back quotes) to take place.
- Back ` quotes cause *command substitution* which allows for a command to be executed within the line of another command.

When using quotes, they must be entered in pairs or else the shell will not consider the command complete.

While single quotes are useful for blocking the shell from interpreting one or more characters, the shell also provides a way to block the interpretation of just a single character called "escaping" the character. To *escape* the special meaning of a shell metacharacter, the backslash \ character is used as a prefix to that one character.

Execute the following command to use back quotes ` (found under the ~ character on some keyboards) to execute the `date` command within the line of the `echo` command:

```
echo Today is `date`
```

Your output should be similar to the following:

```
sysadmin@localhost:~$ echo Today is `date`
Today is Mon Dec 3 21:29:45 UTC 2018
```

You can also place `$(` before the command and `)` after the command to accomplish command substitution:

```
echo Today is $(date)
```

Your output should be similar to the following:

```
sysadmin@localhost:~$ echo Today is $(date)
Today is Mon Dec 3 21:33:41 UTC 2018
```

Why two different methods that accomplish the same thing? Backquotes look very similar to single quotes, making it harder to "see" what a command is supposed to do. Originally, shells used backquotes; the `$(command)` format was added in a later version of the Bash shell to make the statement more visually clear.

If you don't want the backquotes to be used to execute a command, place single quotes around them. Execute the following:

```
echo This is the command '`date`'
```

Your output should be similar to the following:

```
sysadmin@localhost:~$ echo This is the command '`date`'
This is the command `date`
sysadmin@localhost:~$
```

Note that you could also place a backslash character in front of each backquote character. Execute the following:

```
echo This is the command \`date\`
```

Your output should be similar to the following:

```
sysadmin@localhost:~$ echo This is the command \`date\`
This is the command `date`
sysadmin@localhost:~$
```

Double quote characters don't have any effect on backquote characters. The shell will still use them as command substitution. Execute the following to see a demonstration:

```
echo This is the command "`date`"
```

Your output should be similar to the following:

```
sysadmin@localhost:~$ echo This is the command "`date`"
This is the command Mon Dec  3 21:37:33 UTC 2018
```

5.6.6 Step 6

Double quote characters will have an effect on wildcard characters, disabling their special meaning. Execute the following:

```
echo D*
echo "D*"
```

Your output should be similar to the following:

```
sysadmin@localhost:~$ echo D*
Desktop Documents Downloads
sysadmin@localhost:~$ echo "D*"
D*
sysadmin@localhost:~$
```

Important
Quoting may seem trivial and weird at the moment, but as you gain more experience working in the command shell, you will discover that having a good understanding of how different quotes work is critical to using the shell.

5.7 Control Statements

Typically, you type a single command and you execute it when you press **Enter**. The Bash shell offers three different statements that can be used to separate multiple commands typed together.

The simplest separator is the semicolon (`;`). Using the semicolon between multiple commands allows for them to be executed one right after another, sequentially from left to right.

The `&&` characters create a logical "and" statement. Commands separated by && are conditionally executed. If the command on the left of the `&&` is successful, then the command to the right of the `&&` will also be executed. If the command to the left of the `&&` fails, then the command to the right of the `&&` is not executed.

The `||` characters create a logical "or" statement, which also causes conditional execution. When commands are separated by `||`, then only if the command to the left fails, does the command to the right of the `||` execute. If the command to the left of the `||` succeeds, then the command to the right of the `||` will not execute.

To see how these control statements work, you will be using two special executables: `true` and `false`. The `true` executable always succeeds when it executes, whereas, the `false` executable always fails. While this may not provide you with realistic examples of how `&&` and `||` work, it does provide a means to demonstrate how they work without having to introduce new commands.

5.7.1 Step 1

Execute the following three commands together separated by semicolons:

```
echo Hello; echo Linux; echo Student
```

As you can see the output shows all three commands executed sequentially:

```
sysadmin@localhost:~$ echo Hello; echo Linux; echo Student
Hello
Linux
Student
sysadmin@localhost:~$
```

5.7.2 Step 2

Now, put three commands together separated by semicolons, where the first command executes with a failure result:

```
false; echo Not; echo Conditional
```

Your output should be similar to the following:

```
sysadmin@localhost:~$ false; echo Not; echo Conditional
Not
Conditional
sysadmin@localhost:~$
```

Note that in the previous example, all three commands still executed even though the first one failed. While you can't see from the output of the `false` command, it did execute. However, when commands are separated by the `;` character, they are completely independent of each other.

5.7.3 Step 3

Next, use logical "and" to separate the commands:

```
echo Start && echo Going && echo Gone
```

Your output should be similar to the following:

```
sysadmin@localhost:~$ echo Start && echo Going && echo Gone
Start
Going
Gone
sysadmin@localhost:~$
```

Because each `echo` statement executes correctly, a return value of success is provided, allowing the next statement to also be executed.

5.7.4 Step 4

Use logical "and" with a command that fails as shown below:

```
echo Success && false && echo Bye
```

Your output should be similar to the following:

```
sysadmin@localhost:~$ echo Success && false && echo Bye
Success
sysadmin@localhost:~$
```

The first `echo` command succeeds and we see its output. The `false` command executes with a failure result, so the last `echo` statement is not executed.

5.7.5 Step 5

The "or" characters separating the following commands demonstrates how the failure before the "or" statement causes the command after it to execute; however, a successful first statement causes the command to not execute:

```
false || echo Fail Or
true || echo Nothing to see here
```

Your output should be similar to the following:

```
sysadmin@localhost:~$ false || echo Fail Or
Fail Or
sysadmin@localhost:~$ true || echo Nothing to see here
sysadmin@localhost:~$
```