Difference between single (' ') quote vs double (" ") quote

Single Quote

Using single quote in bash script will preserve the literal value of each characters in the single quotes. It will not do any variable interpolation but instead treat it as it is. Basically what you entered is what you will see. None of that escape character bullshit either. If you enter `'\n'` you will see `"\\n"` in your program since it preserve the literal value!

```
echo -E '"\hello!\n"'


# In the program the argument will be ["\"\\hello!\\n\""]
```

However, the downside of using this is that you cannot put a single quote literal in between. Even if you use backslash! It will not work.

Workaround with ' using ANSI-C Quoting

If you want to use single quotes in the parameter you are passing then you might want to use the `$''` ANSI-C quoting for parameter passing.

With `$''` you can pass a string with single quotes in them like so:

```
echo -E $'\'helloworld\\n'
```

However, then you will lose the literal preserving property of single quotes and then you will have to escape single quotes, backslashes and other special characters, accordingly because they will be treated as special characters if escaped!

You cannot do variable interpolation in ANSI-C Quoting! If you want to do variable interpolation use double quotes.

Double Quote

Using double quote it will preserve the literal value as well, except $, `, \ which will expand those variables or commands.

Backslashes will escape the following special characters and will be removed: *$*, ", ` *(backtik)*, \, *<newline>*

```
echo -E "\$\`$\""


# In the program the argument is ["$`\""] because dollar sign, backtik don't
# have special meaning in the program. However, the quotes are escaped
# because they have special meaning. Note this is more for context of
# ruby programming when you are printing the literal string.
```

Otherwise, if you backslash a non special character, the backslash will stay there as well as the non special character.

```
./myProgram "\n" vs ./myProgram "\\n"


# The argument passed in both cases are the same! ["\\n"] because again since
# the backslash is preceded by n which is not one of the special characters
# that double quotes knows how to escape, it will be treated as string literals!
# For the first case. For the second case because you escaped the escape character
# it will be treated as string literals, so they have the same effect in the end!
# surprise pikachu.
```

Aside: Passing \n as Bash Argument

To pass the actual newline character not the literal in Bash you can do it in the following ways.

```
./myProgram "
"


OR
```

```
./myProgram $'\n'
```

Do not try to pass it via `./myProgram "\\n"` this will escape the backslash so your actual string that gets passed is "\\n", a length of 2!

Aside: zsh echo

The zsh version of the echo will automatically append the `-e` flag which enable interpretation of the escape characters. Which mean the escaped character will be interpreted again after you pass it in! Which often conflict with the examples provided in this chapter. So to prevent it either use `bash` or append the `-E` flag to prevent interpretation of the escape character again.

Revision #6
Created 9 November 2022 15:03:58 by Tamarine
Updated 14 May 2023 02:36:41 by Tamarine