
System and Network Engineering - Lecture 6

\$ Bash scripting - best practices



Customize and use the shell environment

Main definitions of shells:

- ❑ **Interactive** - means that the commands are run with user-interaction from the keyboard (e.g. prompt the user to enter input)
- ❑ **Non-interactive** - type of shell that doesn't interact with the user. We can run it through a script or similar. Also, it can be run through some automated process.
- ❑ **Login** - shell is run as part of the login of the user to the system (e.g. ssh), Typically used to do any configuration that a user needs to establish shell environment
- ❑ **Non-login** - any other shell run by the user after logging on, or which is run by any automated process which is not coupled to a logged in user

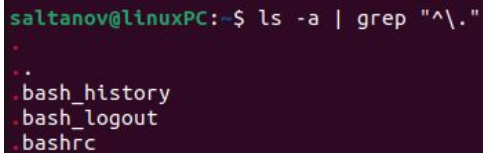
```
saltanov@linuxPC:~$ echo $-  
himbHs  
saltanov@linuxPC:~$ [[ $- == *i* ]] && echo 'Interactive' || echo 'Not interactive'  
Interactive
```

Customize and use the shell environment: login vs non-login

Types of shell configuration files:

- ❑ `/etc/profile:` - sets the environment information for each user of the system. When the user logs in for the first time, the file is run. The shell options are collected from a configuration file in the `/etc/profile.d` directory
- ❑ `/etc/bash.bashrc:` - run this file for every user running a bash shell. When the bash shell is opened, the file is read
- ❑ `~/.profile:` - each user can use this file to enter shell information for their own use. When the user logs in, the file is executed only once! By default, it sets some environment variables and runs the user's `.bashrc` file.
- ❑ `~/.bashrc:` - this file contains information about bash specific to your bash shell, which is read at login and every time you open a new shell
- ❑ `~/.bash_logout:` - every time you logout (logout from the bash login shell) file is run. It clears the screen whenever you log out. Without `.bash_logout` whatever you were working on could be visible for the next user

```
saltanov@linuxPC:~$ ls -a | grep "^\."
```



```
bash_history
bash_logout
bashrc
```

Customize and use the shell environment: login vs non-login

Login shell:

- ❑ Reads configuration files:
 - ❑ `/etc/profile` -> executes scripts in `/etc/profile.d` & `/etc/bash.bashrc`
 - ❑ `~/.profile` -> executes `~/.bashrc`
 - ❑ Type of the login shell defined in ???

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
```

Non-Login shell:

- ❑ Each time start reads the configuration file in `~/.bashrc` & `/etc/bash.bashrc`
- ❑ It originates from the login shell -> gets all the environment set by the login shell via the profile files.
- ❑ Interactive shell can define their own environment variables through rc (resource configuration) files in `/etc` or `$HOME` directory
- ❑ Non-interactive shell will not read any rc and `.profile` files (once its already logged in). So it is highly recommended to use the full path for a command in non-interactive shell (bash scripts)

Customize and use the shell environment: login vs non-login

❑ Summary:

- ❑ The profile files are for interactive login shells. The rc files are for interactive non-login shells.
- ❑ Files in `/etc` directory are executed first and then the files in the home directory.
- ❑ The non-login, interactive shells benefit from both profile and rc files.
- ❑ Use `$su - <user>` (or `--login` or `-L`) to invoke login shell and specific `<user>` environment
- ❑ You can check whether you are in login or non-login shell with `$0`
- ❑ Use `$sudo -i -u user echo \${HOME}` to execute commands under another user with its environment

```
saltanov@linuxPC:~$ sudo -i -u test echo \${HOME}
/home/test
```

```
saltanov@linuxPC:~$ echo $0
bash
saltanov@linuxPC:~$ echo $-
himBHs
saltanov@linuxPC:~$ su - saltanov
Password:
saltanov@linuxPC:~$ echo $0
-bash
```

Customize and use the shell environment: using set

- ❑ **\$set** - Setting your shell environment, allows you to control the behavior of your scripts by managing specific flags and properties.
 - ❑ Use `$set -` to view your current local shell settings and environment variables (not exported). Including exported use `$env`
 - ❑ Assign your positional parameters (e.g, `$1 $2 $3`) that you can use in the scripts
 - ❑ Use `$set -C` - prevent to writing to a file (to prevent possible accident)
 - ❑ Use `$set -x` and `$set +x` to debug your script. Additionally can include `$set -e` so script exists on the first failure

```
saltanov@linuxPC:~$ set first second third
saltanov@linuxPC:~$ echo $*
first second third
saltanov@linuxPC:~$ echo $1, $2, $3
first, second, third
```

```
#!/bin/bash

set -x
echo '1 2 3 4 5 6' | while read a b c; do
echo RESULT : $c $b $a
done
set +x
echo "no debug here"
```

```
saltanov@linuxPC:~/123$ bash 123.sh
+ echo '1 2 3 4 5 6'
+ read a b c
+ echo RESULT : 3 4 5 6 2 1
RESULT : 3 4 5 6 2 1
+ read a b c
+ set +x
no debug here
```

Quoting and escaping

- ❑ when you want to expand a variable i.e. "get the content", the variable name needs to be prefixed with a \$. But, since Bash knows various ways to quote and does word-splitting, the result isn't always the same.

```
$example="Hello world"
```

| Used form | result | number of words |
|--------------------------|------------------------|-----------------|
| <code>\$example</code> | Hello world | 2 |
| <code>"\$example"</code> | Hello world | 1 |
| <code>\\$example</code> | <code>\$example</code> | 1 |
| <code>'\$example'</code> | <code>\$example</code> | 1 |

Quoting and escaping: example

```
$mylist="DOG CAT BIRD HORSE"
```

❑ Wrong

```
for animal in "$mylist"; do  
    echo $animal  
done
```

❑ Correct

```
for animal in $mylist; do  
    echo $animal  
done
```

```
saltanov@linuxPC:~/123$ mylist="DOG CAT BIRD HORSE"  
saltanov@linuxPC:~/123$ for animal in "$mylist"; do echo $animal; done  
DOG CAT BIRD HORSE  
saltanov@linuxPC:~/123$ for animal in $mylist; do echo $animal; done  
DOG  
CAT  
BIRD  
HORSE
```


Quoting and escaping: ANSI C like strings

- ❑ Composed in `$'string'`
- ❑ This is especially useful when you want to pass special characters as arguments to some programs, like passing a newline to `sed`

| Code | Meaning |
|------------------------|--|
| <code>\"</code> | double-quote |
| <code>\'</code> | single-quote |
| <code>\\</code> | backslash |
| <code>\a</code> | terminal alert character (bell) |
| <code>\b</code> | backspace |
| <code>\e</code> | escape (ASCII 033) |
| <code>\E</code> | escape (ASCII 033) \E is non-standard |
| <code>\f</code> | form feed |
| <code>\n</code> | newline |
| <code>\r</code> | carriage return |
| <code>\t</code> | horizontal tab |
| <code>\v</code> | vertical tab |
| <code>\cx</code> | a control-x character, for example, <code>\$'\cz'</code> to print the control sequence composed of Ctrl-Z (^Z) |
| <code>\uxxxx</code> | Interprets <code>xxxx</code> as a hexadecimal number and prints the corresponding character from the character set (4 digits) (Bash 4.2-alpha) |
| <code>\uxxxxxxx</code> | Interprets <code>xxxx</code> as a hexadecimal number and prints the corresponding character from the character set (8 digits) (Bash 4.2-alpha) |
| <code>\nnn</code> | the eight-bit character whose value is the octal value <code>nnn</code> (one to three digits) |
| <code>\xHH</code> | the eight-bit character whose value is the hexadecimal value <code>HH</code> (one or two hex digits) |

Quoting and escaping

- ❑ Enclosing characters in single quotes (') preserves the literal value of each character within the quotes. A single quote may not occur between single quotes, even when preceded by a backslash.

a=apple # a simple variable

arr=(apple) # an indexed array with a single element

| # | Expression | Result | Comments |
|----|---------------|------------|---|
| 1 | "\$a" | apple | variables are expanded inside "" |
| 2 | '\$a' | \$a | variables are not expanded inside '' |
| 3 | ""\$a"" | 'apple' | '' has no special meaning inside "" |
| 4 | ""\$a"" | "\$a" | "" is treated literally inside '' |
| 5 | '\'' | invalid | can not escape a ' within ''; use "" or '\$\'' (ANSI-C quoting) |
| 6 | "red\$arocks" | red | \$arocks does not expand \$a; use \${a}rocks to preserve \$a |
| 7 | "redapple\$" | redapple\$ | \$ followed by no variable name evaluates to \$ |
| 8 | '\'' | \ | \ has no special meaning inside '' |
| 9 | '\'' | \ | \ is interpreted inside "" but has no significance for '' |
| 10 | '\"' | " | \ is interpreted inside "" |
| 11 | "*" | * | glob does not work inside "" or '' |

| | | | |
|----|----------------|------------|---|
| 12 | "\t\n" | \t\n | \t and \n have no special meaning inside "" or ''; use ANSI-C quoting |
| 13 | ""echo hi"" | hi | "" and \$() are evaluated inside "" (backquotes are retained in actual output) |
| 14 | ""echo hi"" | 'echo hi' | "" and \$() are not evaluated inside '' (backquotes are retained in actual output) |
| 15 | '\${arr[0]}' | \${arr[0]} | array access not possible inside '' |
| 16 | ""\${arr[0]}"" | apple | array access works inside "" |
| 17 | '\$a\'' | \$a' | single quotes can be escaped inside ANSI-C quoting |
| 18 | ""\$\t"" | \$_t | ANSI-C quoting is not interpreted inside "" |
| 19 | '!cmd' | !cmd | history expansion character '!' is ignored inside '' |
| 20 | ""!cmd"" | cmd args | expands to the most recent command matching "cmd" |
| 21 | \$_!cmd' | !cmd | history expansion character '!' is ignored inside ANSI-C quotes |

Environment variables and the current context

1. When you call a script such as `./myscript.sh` or `bash myscript.sh` - it will be executed in its own process context (a new child process environment), so all variables set in the script will not be available in the calling shell.
 - ❑ Using `$source myscript.sh` - execute within the context of the calling script
 - ❑ So, environment variables which are set inside the script will be available for the current shell context
2. Environment variables are usually only valid in the current context of the process. So, if you execute something (script or program) which requests a new process environment, the local environment will not be seen within the new process. To pass environment values to a child process, you have to 'export' them e.g. `$export $VAR=value`
 - ❑ Using `$export -p` - can see all env variables that are configured to export to child processes

```
saltanov@linuxPC:~$ export -p | grep PATH
declare -x PATH="/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin"
```

```
saltanov@linuxPC:~$ export abc=100
saltanov@linuxPC:~$ export -p | grep abc
declare -x abc="100"
saltanov@linuxPC:~$ bash
saltanov@linuxPC:~$ echo $abc
100
```

Environment variables and the current context: persistence

❑ Example:

```
#!/bin/bash  
export hello=world
```

```
saltanov@linuxPC:~/123$ ./321.sh  
saltanov@linuxPC:~/123$ echo $hello  
  
saltanov@linuxPC:~/123$ source 321.sh  
saltanov@linuxPC:~/123$ echo $hello  
world
```

❑ Setting Permanent Environment Variables in Bash:

- ❑ `$export VAR="My permanent variable"` in `~/.bashrc`
 - ❑ `$source ~/.bashrc` - to be applied in the current context and to all sessions for the current user
- ❑ `$export GLOBAL="This is a global variable"` in `/etc/environment`
 - ❑ `$source /etc/environment` - to be applied for the all users in the system

Usage of the exec

- ❑ Whenever we run any command in a Bash shell, a subshell is created by default, and a new child process is spawned (forked) to execute the command.
- ❑ When using exec, however, the command following exec replaces the current shell. This means no subshell is created and the current process is replaced with this new command.

```
saltanov@linuxPC:~$ echo $$  
17218  
saltanov@linuxPC:~$ exec sleep 100
```

```
saltanov@linuxPC:~$ ps -ef | grep sleep  
saltanov 17218 17200 0 04:41 pts/0 00:00:00 sleep 100  
saltanov 17246 17236 0 04:44 pts/1 00:00:00 grep --color=auto sleep
```

Usage of the exec

- ❑ `$exec bash` - we can replace the default shell in memory with the Bash
- ❑ We can call scripts or other programs within a script using `exec` to override the existing process in memory. This saves the number of processes created and hence the systems resources. This implementation is particularly useful in cases when we don't want to return to the main script once the sub-script or program is executed:

```
#!/bin/bash

while true
do
    echo "1. Disk Stats "
    echo "2. Send Evening Report "
    read Input
    case "$Input" in
        1) exec df -kh ;;
        2) exec /home/SendReport.sh ;;
    esac
done
```

Usage of the exec

- ❑ Logging within scripts - The exec command is a powerful tool for manipulating file-descriptors (FD), creating output and error logging within scripts with a minimal change

```
#!/bin/bash
script_log="/home/shubh/log_`date +%F`.log"
exec 1>>$script_log
exec 2>&1
datee
echo "Above command is wrong, error will be logged in log file"
date
echo "Output of correct date command will also be logged in log file,
including these echo statements"
```

- ❑ Running scripts in a clean environment - `$exec -c printenv`