



Bash-Scripting Cheat-Sheet

for confidently writing shell-scripts with the bash

Bash-Script

Cheat-Sheet

1/5

... use shebang line as first line of your script ...

```
#!/bin/bash
... your code goes here ...
```

... to prevent users from running your script with the wrong interpreter

... start your script the right way ...

exact: copy to \$PATH and make executable **fast: start your script with bash**

```
#> cp script.sh /usr/local/bin
#> chod +x /usr/local/bin/script.sh
#> script.sh
#> Hello World!
```

```
#> bash script.sh
Hello World!
```

... defining and using variables ...

set with NAME=Value

```
#> NAME=John
#> FULLNAME="John Doe"
#> COUNTER=0
#> export TZ=asia/tokyo
```

use value with \$NAME

```
#> echo Hello $FULLNAME
Hello John Doe
#> PATH=$PATH:~/bin
#> export TZ=asia/tokyo date
```

***** don't forget to export variables, which are to be used by subsequent commands *****

... define variables as permanent ...

*** variables don't just exist "somewhere in the os" ***

they have to be defined within a process and are inherited by the it's sub-processes if exported

	<u>login-shell</u>	<u>other shell</u>
for all users	→ /etc/profile	/etc/bash.bashrc
for one user	→ ~/.profile or ~/.bash_profile	~/.bashrc

**** if bash is running as a script, it doesn't read any of the files above ****

Bash-Script

Cheat-Sheet

2/5

... quoting the right way ...

**** three ways to help the bash to not interpret special characters ****

<code>\x</code>	...	do not interpret the one character after the \
<code>"abc"</code>	...	do not interpret any special character except \$
<code>'abc '</code>	...	interpret nothing within the quotes

... working with variables ...

strings: no special voodoo needed

```
#> FIRST=John
#> LAST=Doe
#> FULL="$FIRST $LAST"
#> echo Hello $FULL
Hello John Doe
```

calculating: use \$((...)) syntax

```
#> A=5; B=13
#> SUM=$(( $A + $B ))
#> echo $(( $B / $A ))
2
```

... special variables ...

`$?` ...returncode of last command
`$$` ... PID of actual shell/script
`$!` ... PID of last background process

`$1..$9` ... positional parameters
use "shift" if you need more
than 9 parameters

... use "source" or "." to include other files ...

config.sh

```
LOGFILE=/var/log/debug.log
LOGTAG=DEBUG
```

script.sh

```
. config.sh
echo "$LOGTAG hi log" >> $LOGFILE
```

... use command-output just like variables ...

**** surround a command by \$(...) to use it's output directly on the command-line ****

example1

```
#> echo "Hello, I am $(whoami)"
Hello, I am john
```

example1

```
#> cp file file_$(date +%Y_%m_%d)
#> ls
file file_2017_10_31
```

Bash-Script

Cheat-Sheet

3/5

... make use of \$? for implementing logic ...

cmd1 && cmd2 → cmd2 only gets executed, if cmd1 returns 0 (\$? == 0)

cmd1 || cmd2 → cmd2 only gets executed, if cmd1 not returns 0 (\$? != 0)

**** chain as many commands, as you need ****

... pre-define your truth :-)

```
#> which true
/bin/true
#> true
#> echo $?
0
```

```
#> which false
/bin/false
#> false
#> echo $?
1
```

... set your own return-code ...

by using "exit N", you exit the script and set the return-code to N
N can be in the range from 0 to 255

... negate every return-code ...

**** prepend a command with an exclamation mark, to negate it's returncode ****

```
#> ! true
#> echo $?
1
```

```
#> ! false
#> echo $?
0
```

... "test" sets \$? to show results for your tests ...

compare strings / numbers

test string = string	→ equal
test string != string	→ not equal
test number -eq number	→ equal
test number -ne number	→ not equal
(other: -lt, -le, -ge, -ge)	→ (lower than, ...)

files

test -x file	→ file exists?
test -r file	→ file readable?
test -w file	→ file writeable?
test -d dir	→ is dir a directory?
test file1 -nt / -ot file2	→ file1 older/newer than file 2

**** test <expression> ist equivalent to [<expression>] mind the spaces around [and] ****

Bash-Script

Cheat-Sheet

4/5

... conditional logic ...

if - then - else

```
if test-command
then
    cmd
    ...
else
    cmd
    ...
fi
```

- runs test-command
- if it returns 0 (\$?), run cmds between then end else
- else run commands between else and fi
- else-block is optional

case

```
case $VAR in
    test1 )    cmd
               ...
               ;;
    test2 )    cmd
               ...
               ;;
    ...
esac
```

- tests content of variable VAR
- tests are written as strings (e.g. "yes" "no") or as search-patterns (e.g. "[Yy]*" or "[Nn]*")
- only the commands of the first match are run

... mostly used loops ...

for-loop

```
for VAR in E1 E2 E3 E4 ...
do
    cmd
    ...
done
```

- runs the commands between **do** and **done** for every element in the list after **in**
- while the commands are run, **VAR** contains the element for which the loop is run.

while-loop

```
while test-command
do
    cmd
    ...
done
```

- runs test-command
- if test-command returns "0" (\$?), run the commands between **do** and **done**
- starts again with test-command and continues until test-command does not return 0.

Bash-Script

Cheat-Sheet

5/5

... redirecting streams ...

**** every command leverages per default three data-streams ****



default:

stdin is read from console / keyboard
stdout and stderr are bound to console
stdout and stderr are mixed

connecting commands:

cmd1 | cmd2 → connect stdout of cmd1
to stdin of cmd2

working with files:

cmd > file	→ write stdout to new generated file
cmd >> file	→ append stdout to file
cmd 2> file	→ write stderr to new generated file
cmd 2>> file	→ append stderr to file
cmd < file	→ read stdin from file
cmd 2>&1	→ connect stderr and stdout both to stdout

Tip: use xargs to copy stdout to next command-line as parameters: #> ls | xargs rm

I hope you enjoyed



This cheat-sheet is meant to be a quick overview over fundamental concepts of bash-scripting (for some of you a review and new stuff for others).

If anything doesn't make sense or is confusing - don't worry.

I'll be back soon with great material to make it all crystal-clear :-)