Session 4 - Repeat, Practice and Expand

You've heard of:

- Variables
- Conditionals
- Loops
- Some *nix tools
- Functions

Let's repeat, practice and expand.

:)

Variables

Declaration:

- Strings: `NAME="value"`
- Numbers: `VAR=value`

Text should always be surrounded by quotes.

Numbers not. So the computer knows that it can use numbers to count and stuff.

Variables

Usage:

- `echo "Hello $NAME."`
- `echo "Hello ${NAME}."`
- `VAR=$((VAR + 1))`

Variables: Local vs Global

```
function test {
    VAR="$1"
    echo "$VAR"
}

VAR="a"
echo "$VAR"
test "b"
echo "$VAR"
```

Variables: Local vs Global

```
function test {
    local VAR="$1"
    echo "$VAR"
}

VAR="a"
echo "$VAR"
test "b"
echo "$VAR"
```

Functions

```
function runCmd {
    local CMD="$1"
    echo "Command: $CMD"
    eval $CMD
    return $?
}

runCmd "ffmpeg -i xxx ..."
```

What are the benefits of having this “execute” function in comparison to doing it in-line without a function?

Functions

```
function checkError {
    local VALUE=$1
    local MSG="$2"

    if [ $VALUE -ne 0 ]; then
        echo "ERROR: $MSG"
        exit $VALUE
    fi
}

...
RETVAL=$?
checkError $RETVAL "This went wrong."
```

Store output of command

```
TEXT=$(command arg1 arg2 arg3 ...)
echo "$TEXT"
```

Store output of command

```
TEXT=$(cat /etc/passwd)
echo "$TEXT"
```

Store output of function

```
TEXT=$(myFunction "$X")
echo "$TEXT"
```

Do *not* mistake the text output stored here with the return value/exitcode stored in "$?" !

NOTE: $TEXT will contain any visible text output that was produced during the execution of the function.

Tests / Conditionals

Bash has 3 different types of comparison operators:

1. File test operators
2. Integer comparison
3. String comparison

Tests / Conditionals

- Check if variable matches string: `if [ "$NAME" == "Dragon" ]; then ...; fi`
- Check if variable is empty: `if [ -z "$NAME" ]; then ...; fi`
- Check if parameter is a folder: `if [ -d "$SOURCE" ]; then ...; fi`
- Check if value is greater-than 1: `if [ $VALUE -gt 1 ]; then ...; fi`

The space around the "[]" brackets is important. Without the spaces the interpreter will throw a syntax error.

In these examples: Which one is of which comparison class?

Comparison operators (numeric)

- -eq : equal to
- -ne : not equal to
- -gt : greater than
- -ge : greater or equal
- -lt : less than
- ...

Comparison operators (string)

- =/== : string equal to
- != : string not equal to
- -z : string is NULL
- -n : string is not NULL
- ...

More: Other comparison operators (tldp.org)

File test operators

- -e: file exists
- -f: is a file (not a folder or device)
- -d: is a directory
- -s: is *not* zero size
- ...

More: File test operators (tldp.org)

btw: NULL

= The computer void.

- string: length of zero
- device: /dev/null
- character: ASCII character index 0
- ...

See: https://en.wikipedia.org/wiki/Null#Computing

Spaces and Quotes

```
NAME="My Name"
if [ $NAME == "My Name" ]; then ...
```

Throws the error: `line x: [: too many arguments`

Spaces and Quotes

Because this is what happens internally:

```
if [ My Name == "My Name" ], then ...
```

The error is thrown, because of the space between “My” and “Name”.

The double-quotes around the variable names are also important, if the variable value contains spaces.

Comparison Tests

- Does FFmpeg executable exist?
- Is $DIR_IN really a folder?
- Is $VAR empty or not?
- Was the last command successful?
- Is $CHOICE this, this or that?

Combined Tests

```
# OR = ||
if [[ "$1" == "x" || "$1" == "X" ]]; then
    echo "It's an x (or an X)!"
fi
```

```
# AND = &&
if [[ $1 -gt 3 && $1 -lt 7 ]]; then
    echo "Bingo!"
fi
```

if - then - else if - else

```
if [ $VAL -eq 1 ]; then
    # Do if VAL=1
elif [ $VAL -eq 2 ]; then
    # Do if VAL=2
elif [ $VAL -eq 3 ]; then
    # Do if VAL=3
else
    # Do if anything else.
fi
```

Case statement

Multiple choice tests ;)

```
case "$CHOICE" in
    one)
        ...
    ;;

    two)
        ...
    ;;

    *)
        ...
    ;;
esac
```

See: Testing and Branching (tldp.org)

Case statement

- Excellent for commandline parameter mode switching.
- Without “break”, execution will continue until break or esac encountered!
- Quotes not mandatory, since word splitting does not take place.
- Each test line ends with a “)”.
- Each condition block ends with a double semicolon ;;.
- If a condition tests true, then the associated commands execute and the case block terminates.
- The entire case block ends with an ‘esac’.

Return and Exit

> Return is to a function what Exit is to a program/script.

And they both allow to return a numeric value, stored in **$?**

Return True or False

```
function isDir() {
    if [ -d "$1" ]; then
        # 0 = true
        return 0
    else
        # not 0 = false
```

```
        return 1
    fi
}

DIR="$1"
echo "Is $DIR a folder...?"

if isDir "$DIR"; then echo "Yes!"; else echo "No."; fi
```

true/false: Usually it's the other way around:

- true = 1
- false = 0

But since BASH expects programs to exit with "0", it's more convenient to assume "0=true", since it makes the code shorter/easier here.

String manipulation

```
${#string}                  Length of $string

${string:position}          Extract substring from $string at $position
${string:position:length}   Extract $length characters substring from $string at $position

${string#substring}         Strip shortest match of $substring from front of $string
${string##substring}        Strip longest match of $substring from front of $string
${string%substring}         Strip shortest match of $substring from back of $string
${string%%substring}        Strip longest match of $substring from back of $string

${string/substring/replacement}     Replace first match of $substring with $replacement
${string//substring/replacement}    Replace all matches of $substring with $replacement
${string/#substring/replacement}    If $substring matches front end of $string, substitute $replacement
${string/%substring/replacement}    If $substring matches back end of $string, substitute $replacement
```

See "Advanced Bash-Scripting Guide (tldp.org)" for additional information.

Extract text

Extract fixed length:

```
LINE=$(md5sum "$0")
HASH=${LINE:0:32}

echo "Line: '$LINE'"
echo "Hash: '$HASH'"
```

Extract file suffix

```
FILE="bla.txt"
NAME="${FILE##*.}"
echo $NAME
```

> ${string##substring} Strip longest match of $substring from front of $string

Exercise:

DPX - 1 to many

Take 1 image file and copy it n times with subsequent filenames, according to a given naming rule.

Programmer Questions

- How could you do that?
- Which input information do you require?
- Which steps are necessary?
- Which problems could you encounter?
- How could you avoid them?
- Or how could you handle errors if they happen?

Brainstorm to get a rough idea of how to do that, then continue with the other considerations and start building up your program step by step. Like with Lego blocks.

Don't expect to do it completely right in one go ;) Regardless how experienced you become.

Steps?

- Copy one file.
- Repeat n times.
- Input data:
  - Sourcefile
  - Output folder
- Other: define naming rule

Take 1 image file and copy it n times with subsequent filenames, according to a given naming rule.

Options

For loop with "seq": `for i in $(seq 1 $n); do ... done`

While loop: `while [ $i -le $n ]; do ... done`

Homework 1/2

Write a function that takes an integer as input and increases its value by 1 and returns this number. Echo the incremented value.

Optional: Use this function as in a while loop to count up until 13.

Homework 2/2

Write a function that you can use to check if a given string has the following properties:

- The string is not empty.
- It is a file with filesize > 0
- It is not a directory
- Your user is allowed to read that file (see: "-r" in "Bash Tests")

Return *True* if the string matches these conditions, *False* if otherwise.