Bash Cheatsheet

Multi-line Command

To execute multi-line command in bash script simple put a \ after breaking up your commands. There should be no white spaces after the backslash, else it will fail!

Every single line should be followed by a \ until you finished typing the command.

```
curl https://cat.png \#no spaces after!
-o here.txt \
```

If-Statements

In Bash the if-statements follows the structure:

```
if CONDITION
then
COMMAND
elif CONDITION
then
COMMAND
else
COMMAND
fi
```

As you can see every if chain will end with *fi* keyword. Every *if/elif* condition statement will be followed by the *then* keyword.

If you want to put the *then* on the same line with the *if* keyword, and because *if, then, else, elif, fi* are all shell keyword they cannot be used on the same-line. To fix this you have to put a ; to end the previous statement and the keyword before you can use another keyword. As an example:

```
if CONDITION; then
COMMAND
fi
```

Conditions

Okay, now we know the basic of if-statements how do I put it into use by filling in the conditions? There are couple different ways of writing conditions, here I will only go over the most commonly used ones.

1.

```
# The first way is using test
if test <expressions>; then
COMMAND
fi
```

Using this method you can test whether the files exists, and compare values. It has it's own set of syntax for example to check if a certain file named "foo.txt" exists then you would type `test -f "foo.txt"` and it will evaluate to true if only the file "foo.txt" exists.

For comparing values we cannot use the symbols directly, ==, <, >, <=, or >=. Instead we have to use their flag equivalent below:

Compartor	Flag Equivalent
==	-eq
>	-gt
<	-lt
>=	-ge
<=	-le
or	||
and	&&
not	!

You can also append *!* to negate the expression to check the opposite.

Use = to do string equality comparison.

2.

```
# The second way is using [] square brackets
if [ some test ]; then
```

```
COMMAND
fi
```

The square bracket is like *test* and essentially all the operators that you can use with *test* you can also use in the square brackets.

There must be a space between the test and the left bracket, [and the right bracket,], otherwise Bash cannot understand it!

3.

```
# The third way is using [[]] double square brackets
if [[ some test ]]; then
COMMAND
fi
```

The double square brackets is like an upgrade of the normal square bracket. It comes from *ksh*.

With the double square brackets you can use some of the comparison operators without using the flags. So we are allowed to use >, =, and <, but they are used in **lexicographical comparison!** However, <= and >= still requires the flag equivalent.

4.

```
if command;then
COMMAND
fi
```

Bash runs the command you have provided and then will run the if-statement according to the exit code. It will run it if the exit code is 0, and will not run it if it is not 0.

> Remember in programming, 0 represent the command carried out successfully, and anything not 0 represent some sort of errors occurred.

Variables & Arrays

To declare a variable in Bash follow the following syntax structure:

```
VAR_NAME=VALUE
```

You must not use any spaces between the variable name and the value! Otherwise, it will error out because you are not following Bash syntax!

Command Output -> Variable

To store the output of a command into a variable follow the following syntax structure

```
VAR_NAME=$(COMMAND ARGS)


OR


VAR_NAME=`COMMAND ARGS`
```

This will do command substitution, it will execute the command and then substitute the return value as the value.

To prevent the output of the command from being processed for word splitting (i.e. the \n loses their meaning in a text file), you would quote the command substitution `"$(COMMAND ARGS)"` to prevent word splitting.

From the GNU shell specification: **"The shell scans the results of parameter expansion, command substitution, and arithmetic expansion that did not occur within double quotes for word splitting."**

Arrays

To declare a array follow the following syntax structure:

```
arr=(1 2 3 4 5 6 7)


OR


arr=(
"hello"
    "world"
    "hehe"
    "xd"
)
```

The array can contain different types, either an integer, float, or even strings.

You can refers to each individual elements using `${var_name[index]}`, the *index* are 1-based indices.

To refer to the entire array for say the usage of running a command with arguments from the entire array you can use `$array_name[@]` to refer to all of the elements from the array.

Array Operation Summary Table

Syntax	What it Does
`arr=()`	Create an empty array
`arr=(1 2 3)`	Initialize an array
`${arr[2]}`	Get the third element
`${arr[@]}`	Get all of the elements
`${!arr[@]}`	Get the indices of all the elements
`${#arr[@]}`	Get the length of the array
`arr[0]=3`	Overwrite the first element with value 0
`arr+=(4)` or `arr+=($another_variable)`	Add a new value to the array

Let and (( )) Construct

The `let` built-in command allows you to do arithmetic operations on variables. It can be used to do a simple increment operations. Examples below:

Command	What it Does
`let a=11`	Same as `a=11`
`let a=a+5` or `let "a = a + 5"`	Both set `a` to be 5 more of itself
`let "a <<= 3"`	Left-shifts `a` 3 places
`let "a += 4"` or the other math operators	Same as `let "a /= 4"`
`let a++` or `let "a++"`	C-style operators works as well!

Some simple operations like increment cannot be carried out by the Bash directly so you would use `let` command to actually do the increments! You do not need to refer to the variable names using dollar signs.

You can replace all the operators described above in between `(( ))` to have the same effect, minus the need for using double quotations. Addition, subtraction, division, multiplication, bit shifts, post/pre increment all works.

For Loops

To do for loops there are couples of ways:

1. Looping through array elements

```
for ele in ${array[@]}; do
    echo $ele
done
```

2. Looping through array indices

```
for i in ${!array[@]}; do
    echo $i
done
```

3. Looping through range, the end is included

```
for value in {start..end..step}
do
    COMMAND
done
```

The keyword `break` and `continue` are also available for use in the for loop just like how it would work in any other languages.

While Loops

While loops have the basic structure as follows:

```
while [ some test ]
do
    COMMANDS
done
```

Bash Functions

There are two ways of writing a bash function

```
function_name() {
    echo "This is the body of the bash function"
}
```

```
function another_bash_function() {
    echo "This is another function!"
}
```

With either way you would be invoking the function by just calling it like it is a command. For example, to invoke the first function you would simply type `function_name` and then provide any argument that you would like to pass into the function. The arguments passed into the function can be accessed using `$1, $2, $3, ...` just like in bash script. `$#, $@` works as well with respective to the argument passed to the function, not the script!

Revision #12
Created 11 November 2022 04:14:13 by Tamarine
Updated 7 May 2023 17:08:49 by Tamarine