Bash Cheatsheet $

Pablo Jesús González Rubio (n0nuser) - nonuser.es

Programming

Variables

Variables in bash aren't like in C that you have to declare them before using them and which type of data are they. They can't start with a number.

In Bash, variables can take any form, much like in Python.

```
  var1 = 1
  var2 = 1.5
  var3 = hello
  # var4 will get the result of the bash function
  var4 = $(whoami)
```

To call the value of a variable and use it in another function:

```
  # To call the value of var 4:
  $var4
```

For arrays just:

```
declare -a ARRAY_VARIABLE
# To fill an array with values:
for (i=0;i<n;i++);do
  ARRAY_VARIABLE[$i]=VALUE
  #ARRAY_VARIABLE[$i]="VALUE"
done
```

To call an array with variable index:

`${ARRAY_VARIABLE[$POSITION]}`

Important for conditions

Arithmetic operators:

- `-eq` : is equal to
- `-ne` : not equal to
- `-gt` : greater than
- `-ge` : greater than or equal to
- `-lt` : less than
- `-le` : less than or equal to

Strings operators:

```
if [ $VARIABLE = "string" ]; then
  # commands
fi
```

```
if [[ $VARIABLE == "string" ]]; then
  # commands
fi
```

Conditions: if, elif, case

In the conditions, it's super important to leave one blank space after the first `[` and one before the `]`.

- `if`

```
if [ conditions ]
then
  commands
fi
```

- `elif` (If the condition before this one hasn't matched)

```
if [ conditions ]
then
  commands
fi

elif [ conditions ]
then
  commands
fi
```

- `else` (If no condition has matched)
- `case` (It's a Switch)

```
var = #some input
case $VARIABLE in
  value1)
    commands
    ;;
  valueN)
    commands
    ;; #Like a Break
  *)
    # This is default value
    # Used when the value of VARIABLE is not valid
    commands
    ;;
esac
```

⚠️ In Bash, it's not like C, we have to end all the values with `;;`.

Otherwise, the program will not work.

Loop: for

```
for VARIABLE in 1 2 3 4 5 .. N
do
  command1
  //...
  commandN
done
```

- `$@` Is an array with all the arguments.
- `$#` Is the number or passed arguments.
- `$1` Selects the first argument.
- `$i` Calls the value of *i* variable.

Loop: while/until

```
while [ condition ]
do
  command1
  command2
  command3
done
```

```
# Similar to while
# Does the loop until the condition is true, then stops.
until [ condition ]
do
  command1
  command2
  command3
done
```

Functions

```
# They are much like in C.
# You have to write the function in the upper part
# and the call to the function at the end.

function FUNCTION_NAME {
  commands
  # commands with parameter:
  echo $1
}

...

# There's no need in supplying parameters if the function doesn't need one
FUNCTION_NAME paratemer1 ... parameterN
```

Traps

A trap is a function to modify the program behaviour based on user keyboard combinations.

i.e: When you pulse `Ctrl + C` the program usually aborts, that's because the Operative System sends a `SIGINT` signal to make it abort if it was a `SIGKILL` it would actually assassinate the program in the act without letting it finish normally. Well, with a *trap* you can alter that behaviour into calling a function. This has the advantage of doing some cleanup before the program ends for example.
It can also make a call to another function if it detects another type of signal like a `SIGCHLD`, when a child of a parent process has died.

```
# Trap for Ctrl + C
trap ctrl_c INT

function ctrl_c(){
  commands when pulsed Ctrl + C
  //The program will finnish after the commands
}
```

Useful commands

This command verifies a program is installed in the PC:

```
command -v PROGRAM >/dev/null 2>&1 || {echo >&2 "PROGRAM is not installed.
Aborting."; exit 1;}
# Sends the results to null and only return if it's not installed.
# This could be modified to ask to install or whatever.
```

Verifies if a file or directory exists.

```
FILE=/path/to/file/file.ext
if [ -f "$FILE" ]; then
    echo "$FILE exist."
else
    echo "$FILE doesn't exist."
fi
```

```
# If just verifying a file doesn't exists.
FILE==/path/to/file/file.ext
if [ ! -f "$FILE" ]; then
    echo "$FILE doesn't exist."
fi
```

Parses text of a delimited file (E.G: `*.csv`)

```
FILE="/path/to/file/file.ext"
i=0
while IFS='=' read VALUE1 ... VALUEN
do
      echo "$i value : $VALUE1"
      # ...
      echo "$i value : $VALUEN"
done < $FILE
```

In this fragment of code, we are taking the values of a file delimited by `=` and echoing the values of it.
Instead of echoing them, you can input them into a variable you're going to use later, or else.

E.G: `tea = delicious` then `VALUE1 = tea`, and `VALUE2 = delicious`.

This command searches for specific words or text in some text:

```
grep ARGUMENT
  -w, --word-regexp              match only whole words
  -x, --line-regexp              match only whole lines
  -v, --invert-match             select non-matching lines
  -m, --max-count=NUM            stop after NUM selected lines
  -H, --with-filename            print file name with output lines
  -r, --recursive                greps also in directories
  -R, --dereference-recursive  also follows all symlinks
  -L, --files-without-match      print only names of FILEs with no selected lines
  -l, --files-with-matches       print only names of FILEs with selected lines
  -c, --count                    print only a count of selected lines per FILE
```

I usually use a lot the `-v` and the `-r` arguments.

This command cuts text. Useful to select parts of repetitive static information.

```
  cut ARGUMENTS
    -b, --bytes=LIST        select only these bytes
    -c, --characters=LIST   select only these characters
    -d, --delimiter=DELIM   use DELIM instead of TAB for field delimiter
    -f, --fields=LIST       select only these fields;  also print any line

  cut -d 1-7 #Will only select the info between character 1 and 7
```

'*Awk*' a language for text processing:

```
# Supossing we cat a file and it returns:
# This is a file
awk '{print $1}' thatfile.txt
# Or
cat thatfile.txt | awk '{print $1}'
# Will return 'This'
```

```
# This will return column 2 elements that match the pattern.
awk ' /'PATTERN'/ {print $2} '
```

Awk is very extensive so this link can be useful for those who wants to learn more. Sed is another very similar program.

EOF

Here ends the article for now. If while reading this you missed something, contact me and I'll upload it as fast as I can!