Programster

Tutorials focusing on Linux, programming, and open source

Bash Cheatsheet

SUNDAY, JULY 19, 2015

Below are a set of "cheats" specific to programming BASH scripts. Since my default terminal uses BASH, I quite often confuse my commands with what should really be in the Linux CLI Cheatsheet, so if you can't find what your looking for here, please check that out as well.

Related Posts

- Linux CLI Cheatsheet
- Docker CLI Cheatsheet

Bash Guard

Inject this into the top of your bash scripts to ensure that the script is running with in bash rather than sh. This is useful because bash and sh often require different syntax. It won't force the user to re-execute the script properly, but just re-execute itself as it was supposed to be done in the first place.

```
#!/bin/bash
if ! [ -n "$BASH_VERSION" ];then
    echo "this is not bash, calling self with bash....";
    SCRIPT=$(readlink -f "$0")
    /bin/bash $SCRIPT
```

```
    exit;
fi

# Put your code here
```

Get Absolute Path of Current Script
When writing scripts, I always like to use absolute references that are relative to the current file. This allows me to call the script from any directory, whilst also allowing me to pass that path to any other script working in any other directory as well. This is the same as using DIR in PHP. I also have a TCL equivalent.

```
DIR=$( cd "$( dirname "${BASH_SOURCE[0]}" )" && pwd )
```

Note:
This will not work if the last part of the path is a symlink (directory links are okay). If you want to also resolve any links to the script itself, you need a multi-line solution.

This snippet was taken from a Stack Overflow post.

Ensure Running as Root
Inject this into the top of your bash scripts to ensure that the script is running with sudo or root permissions. This is useful if you need to ensure that once a script is started, it is never going to ask for a password on a "sudo" line if you have any.

```
USER=`whoami`

if [ "$USER" != "root" ]; then
        echo "You need to run me with sudo!"
        exit
```

```
fi

# Your script goes here....
```

Load Environment Variables From File

```
source /path/to/settings/file.sh
```

Get Variable From User

```
echo -n "Enter your ____ and press [ENTER]: "
read MY_VARIABLE
echo $MY_VARIABLE
```

Ask User For Confirmation

```
read -p "Are you sure? " -n 1 -r
echo    # (optional) move to a new line
if [[ $REPLY =~ ^[Yy]$ ]]
then
    # do dangerous stuff
fi
```

Store Commands In Variables

You can store commands as strings in variables and execute them later. However, to resolve issues with commands that have quotes, and for readability, it is best to use the eval command on that variable when you want to execute it. E.g.

```
CMD='echo "hello world"'
```

```
eval $CMD
```

Run the script below to demonstrate why you need to use eval

```
CMD='echo "hello world"'
echo -n "With eval: "
eval $CMD

echo -n "Without eval: "
$CMD

# Output
# With eval: hello world
# Without eval: "hello world"
```

Unix Timestamp

```
TIMESTAMP=`date +%s`
```

Date Variable

```
DATE=`date +%Y_%m_%d`
```

Time Variable

```
TIME=`date +%H:%M:%S`
```

Date-Time Variable

```
DATETIME=`date +%Y_%m_%d_%H:%M:%S`
```

String Concatenation Example

```
prefix="MyPrefix_"
DATETIME=`date +%Y_%m_%d_%H:%M:%S`
name="$prefix$DATETIME"
echo $name
```

Alternatively, in order to stick characters in between the variables, perform the following.

```
prefix="MyPrefix"
DATETIME=`date +%Y_%m_%d_%H:%M:%S`
name="`echo $prefix`_`echo $DATETIME`"
echo $name
```

Handling Arguments
The easiest way to explain this is to demonstrate with the following script:

```
#!/bin/bash

echo "you passed $# arguments to the script"
echo "They are: $@"
echo "The script is called $0 "
echo "the first argument is $1"
echo "the second argument is $2"
```

Below is the output I got when I ran it on /tmp/script.sh

```
you passed 0 arguments to the script
They are:
The script is called /tmp/script.sh
the first argument is
the second argument is
```

Ensure Number Of Expected Arguments

```
EXPECTED_NUM_ARGS=1;

if [ "$#" -ne $EXPECTED_NUM_ARGS ]; then
    echo "Illegal number of aguments"
fi
```

Get File MD5

```
md5sum [filepath]
```

Ensure A Directory Exists

```
DIRECTORY="/my/directory/path"
if ! [ -d $DIRECTORY ]; then
    echo "$DIRECTORY does not exist!"
    exit;
fi
```

Ensure Running User Has Write Permission for a Directory

```
if ! [ -w $DIRECTORY ] ; then
    echo "You need to run this script with a user that has pe
```

```
    exit;
fi
```

Assign IP to Variable

```
OUTBOUND_INTERFACE="eth0"
SERVER_IP=$(ifconfig $OUTBOUND_INTERFACE | grep "inet addr" |
```

Note:
For OpenVZ you would want to change `$OUTBOUND_INTERFACE` to be `venet0:0`.

Check If Variable is Empty

```
if [ -z "$EMPTY_STRING" ]; then
    echo "That is an empty string" 1>&2;
fi
```

Alternatively, if you want to perform the reverse, and check that the variable is not empty:

```
USER="ubuntu"
if [ -n "$USER" ]; then
    echo "Hello $USER" 1>&2;
fi
```

Switch / Case Statement

Below is an example switch statement stolen from The Geek Stuff.

```
case "$1" in

1)  echo "Sending SIGHUP signal"
    kill -SIGHUP $2
    ;;
2)  echo  "Sending SIGINT signal"
    kill -SIGINT $2
    ;;
3)  echo  "Sending SIGQUIT signal"
    kill -SIGQUIT $2
    ;;
9) echo  "Sending SIGKILL signal"
   kill -SIGKILL $2
   ;;
*) echo "Signal number $1 is not processed"
   ;;
esac
```

Lists (Arrays)
You can create an array with one like like so:

```
MY_ARRAY=("item1" "item2")
```

Or you can spread it over multiple lines by continuously appending to the array:

```
MY_ARRAY=()
MY_ARRAY+=('foo')
MY_ARRAY+=('bar')
```

Variable Variables
I recommend avoiding variable variables, but they can be used in bash like so:

```
foo="something"
bar="foo"
echo ${!bar} # outputs "something"
```

Get Password From User

When getting a password from the user, you want to use stars to prevent others from seeing the inputs. However, because this is the case, you need to make sure they have to re-enter the password to ensure that they did not make a spelling mistake. At no point should you echo out the password.

```
function getPasswordFromUser()
{
    while [ -z "$PASSWORD" ]
    do
        echo "Please enter a password:"
        read -s PASSWORD1
        echo "Please re-enter the password to confirm:"
        read -s PASSWORD2

        if [ "$PASSWORD1" = "$PASSWORD2" ]; then
            PASSWORD=$PASSWORD1
        else
            # Output error message in red
            red='\033[0;31m'
            NC='\033[0m' # No Color
            echo ""
            echo -e "${red}Passwords did not match!${NC}"
        fi
    done
}

# This works because there is no scope in the function
# any variables defined within it are defined outside it
getPasswordFromUser;
```

```
echo "your password is $PASSWORD"
```

Note:
This is also a good example of how to use the while loop.

Switch / Case Statement
Below is an example switch statement stolen from The Geek Stuff.

```
case "$1" in

1)  echo "Sending SIGHUP signal"
    kill -SIGHUP $2
    ;;
2)  echo  "Sending SIGINT signal"
    kill -SIGINT $2
    ;;
3)  echo  "Sending SIGQUIT signal"
    kill -SIGQUIT $2
    ;;
9) echo  "Sending SIGKILL signal"
   kill -SIGKILL $2
   ;;
*) echo "Signal number $1 is not processed"
   ;;
esac
```

Lists (Arrays)
You can create an array with one like like so:

```
MY_ARRAY=("item1" "item2")
```

Or you can spread it over multiple lines by continuously appending to the array:

```
MY_ARRAY=()
MY_ARRAY+=('foo')
MY_ARRAY+=('bar')
```

Variable Variables
I recommend avoiding variable variables, but they can be used in bash like so:

```
foo="something"
bar="foo"
echo ${!bar} # outputs "something"
```

Get Password From User
When getting a password from the user, you want to use stars to prevent others from seeing the inputs. However, because this is the case, you need to make sure they have to re-enter the password to ensure that they did not make a spelling mistake. At no point should you echo out the password.

```
function getPasswordFromUser()
{
    while [ -z "$PASSWORD" ]
    do
        echo "Please enter a password:"
        read -s PASSWORD1
        echo "Please re-enter the password to confirm:"
        read -s PASSWORD2

        if [ "$PASSWORD1" = "$PASSWORD2" ]; then
            PASSWORD=$PASSWORD1
        else
            # Output error message in red
```

```
            red='\033[0;31m'
            NC='\033[0m' # No Color
            echo ""
            echo -e "${red}Passwords did not match!${NC}"
        fi
    done
}

# This works because there is no scope in the function
# any variables defined within it are defined outside it
getPasswordFromUser;
echo "your password is $PASSWORD"
```

Note:
This is also a good example of how to use the while loop.

References

- Stack Overflow - How to check if a directory exists in a shell script
- Stack Overflow - How do I prompt a user for confirmation in bash script? [duplicate]
- Stack Overflow - Bash: add value to array without specifying a key
- Stack Overflow - Bash - variable variables
- Stack Overflow - How to change the output color of echo in Linux
- Stack Overflow - How can I write a here doc to a file in Bash script?

TAGS: BASH, CHEATSHEET

SHARE THIS POST

AUTHOR
Programster

Stuart is a software developer with a passion for Linux and open source projects.