Cheat sheet

Bash commands

A Bash script is a text file that contains programming statements that execute commands that are part of the host computer's operating system. Typically system administrators and programmers use Bash scripts to avoid having to repetitively execute tasks manually in a terminal.

A typical use case for a Bash script is to do set up tasks for a newly provisioned computer. Thus, in addition to being a tool for system administrators and programmers, a Bash script can also be used by system provisioning software. (The `$` symbol that proceeds commands in the examples represents the command line prompt.)

Running a Bash script

There are two ways to run a Bash script. The first way is to execute it as a parameter of a direct call to the `bash` executable binary, like so:

```
$ bash myscript

Hello from Bash
```

WHERE the content of the file `myscript` is as follows:

```
echo "Hello from Bash"
```

The second way to use the `chmod` command to change the permissions of the bash script to make it a standalone executable, like so:

```
$ chmod +x ./myscript

$ ./myscript
```

Using a shebang header

Unlike a binary executable file which knows how to interact with the computer's operating system directly, a Bash script, which is always text based, requires another program to run its commands. This other program is called an interpreter. The interpreter that runs a Bash script is, as the name implies, `bash`. However, in other cases, a bash script can be run by another interpreter that's installed on the host computer. An example of another interpreter is `sh`.

The way a Bash script lets the operating system know which interpreter to use is according to a declaration made at the first line of the script. This first line declaration is called a script header. It is also called a shebang. A shebang starts with the characters `#!`

An example of a shebang is as follows:

```
#!/usr/bin/bash
```

The shebang shown above tells the operating system to use the interpreter located at the filepath `/usr/bin/bash` to execute the lines of code that will follow in the script.

Another form of a shebang is as follows:

```
#!/usr/env bash
```

The shebang #!/usr/env bash tells the operating system to search the computer's `$PATH` to find the `bash` interpreter. Thus, any instance of `bash` can be used as long as the it's in a location defined by the `$PATH` environment variable.

Executing scripts in another language

As mentioned above, a host operating system has to have the capability to run scripts for other interpreted programming languages as long as the given interpreter is installed on the host computer and the shebang has been configured properly.

The example below shows a Perl script file. Notice that the script header (a.k.a. shebang) declares the file as a Perl script:

```
#!/usr/bin/perl
use strict;
use warnings;

print "Hello World from Perl\n";
```

Variables

Using variables is a critical factor for programming Bash scripts. The following sections describe various aspects of working with variables under Bash.

Variable declaration

You declare a variable in a bash script like so `VARIABLE_NAME=<value>` WHERE `<value>` is the value assigned to the variable. Then, to reference the variable, put the `$` symbol before the variable name being referenced, like so: `$VARIABLE_NAME`.

Example:

```
#!/usr/bin/env bash
MSG="Hello World"
echo "$MSG " # Hello World
```

BE CAREFUL to make sure there is no space on either side of the `=` symbol when declaring a variable. The following will not work: `MYVARIABLE = foo`.

String manipulation using parameter expansion

Parameter expansion a technique to get the value from the referenced entity such as a variable in a Linux script or an environment variable according to a piece of processing logic. A variable is processed by enclosing the variable name within the `${}` characters. The processing logic is defined by characters that follow the variable name. For example the variable named `MSG`, the statement `${MSG^^}` turns all lowercase characters in the variable `MSG` to uppercase, like so:

```
MSG="aBcDeFg"
echo ${MSG^^}

#returns ABCDEFG
```

And the statement `${MSG,,}` turns all uppercase characters in the variable `MSG` to lowercase, like so:

```
MSG="aBcDeFg"
echo ${MSG,,}

#returns abcdef
```

Examples:

The following examples demonstrate various ways to use parameter expansion on Linux variables.

Word replacement

```
MSG="Say hi to Chris and Sidney"
echo ${MSG//Chris/Billy}

#returns Say hi to Billy and Sidney
```

Character replacement using regular expressions

Replace all alphabetic characters with the character `X` but leave the numerals alone

```
MSG="I need 10"
echo ${MSG//[a-zA-Z]/X}

#returns X XXXX 10
```

Replace all numeric characters with the character `Z` but leave alphabetic characters alone

```
MSG="I need 10"
echo ${MSG//[0-9]/Z}

#returns I need ZZ
```

Extracting substrings

Use the `:` symbol to get the substring of all the characters after the starting at position 4

```
MSG="The Rolling Stones"
echo ${MSG:4}

#returns Rolling Stones
```

Use the `:` symbol to get the substring that has 7 characters starting at position 4

```
MSG="The Rolling Stones"
echo ${MSG:4:7}

#returns Rolling
```

Use the `#` symbol to get the substring after the characters `The` starting from the left side of the string

```
MSG="The Rolling Stones"
echo ${MSG#The}

#returns Rolling Stones
```

Use the `%` symbol to get the substring before the characters `Rolling Stones` starting the right side of the string

```
MSG="The Rolling Stones"
echo ${MSG%Rolling Stones}

#returns The
```

Case conversion

Use the `^` symbol to convert the first character in a string to uppercase.

```
MSG="aBcDeFg"
echo ${MSG^}

#returns ABcDeFg
```

Use the `^^` symbols to convert the all lowercase characters in a string to uppercase.

```
MSG="aBcDeFg"
echo ${MSG^^}

#returns ABCDEFG
```

Use the `,` symbol to convert the first character in a string to lowercase.

```
MSG="TuVwXyZ"
echo ${MSG,}

#returns tuVwXyZ
```

Use the `,,` symbols to convert all characters in a string to lowercase.

```
MSG="TuVwXyZ"
echo ${MSG,,}

#returns tuvwxyz
```

Collections

The following sections describe how to group data as a collection in a bash script. Bash supports two types of collections. One type is an `array`. The other type is a `map`.

An `array` is a collection in which elements of the collection are accessed according to a number.

A `map` is a collection in which elements of the collection a key value.

Arrays

Creating an array

The following creates an array with three elements and assigns the array to the variable named `my_array`.

```
my_array=('Alex' 'Ada' 'Alexandra')
```

Adding an element to an array

The following uses the `+=` operator to add an element with the value `Soto` to the array named `my_array`.

```
my_array+=('Soto')
```

Removing an element to an array

The following uses the `unset` keyword to remove the fourth element from the array named `my_array` at index `3`.

```
unset my_array[3]
```

Viewing data in an array

The following uses an index number to view the data in the first element of the array named `my_array`.

```
echo ${my_array[0]}
```

The following uses an index number to view the data in the third element of the array named `my_array`.

```
echo ${my_array[2]}
```

The following uses the `@` symbol to view all elements in the array named `my_array`.

```
echo ${my_array[@]}
```

Getting the number of elements in an array

The following uses the `#` and `@` symbols to get a count of the number of elements in the array named `my_array`.

```
echo ${#names[@]} # 3
```

Copy, paste and run in your terminal:

Copy and paste the following code into your terminal window to create and execute a Bash script with the filename `arrays-01.sh`.

The Bash script demonstrates the array commands described above.

```
cat << 'EOF' > arrays-01.sh
#!/usr/bin/env bash

names=('Alex' 'Ada' 'Alexandra')
names+=('Soto') # Appends element, Soto
unset names[3] # Removes element at index 3, (Soto)

echo ${names[0]} # Alex
echo ${names[1]} # Ada
echo ${names[2]} # Alexandra

# @ indicates all elements in the array
echo ${names[@]} # Alex Ada Alexandra

# Count of names
echo ${#names[@]} # 3
EOF
bash arrays-01.sh
```

Maps

In Bash, a map is a collection of elements that are organized as key-value pairs. Another way to think of a map is as a named associative array.

To access an element in a map you reference its key.

Creating a map

You create a map using the command `declare -A <map_name>` WHERE the option `-A` indicates that the variable represents an associative array, which is that same as a map.

Examples:

The following example demonstrates creating a map variable named `score`. The variable `score` has four elements that describe the scores of four people named `alex`, `edson`, `sebi` and `chris`.

```
declare -A score
score[alex]="1"
score[edson]="2"
score[sebi]="3"
score[chris]="4"
```

The following example demonstrates using the `!` and `@` symbols to show all the keys in the map named score.

```
echo ${!score[@]}
```

The following example demonstrates using the `unset` keyword to delete the element identified by the key `chris` from the map variable named `score`.

```
unset score[chris] # Delete chris entry
```

The following example demonstrates using the `@` symbol to show all the values in the map named `score`.

```
echo ${score[@]} # show all the values
```

The following example demonstrates calling the value of the element associated with the key `edson`.

```
echo ${score[edson]} # show the value of edson: 2
```

The following example demonstrates using the `#` and `@` symbols to get a count of the number of elements in the map variable named `score`.

```
echo ${#score[@]} # show the number of elements in the map: 3
```

Copy, paste and run in your terminal:

Examples:

```
cat << 'EOF' > maps-01.sh
#!/usr/bin/env bash

declare -A score
score[alex]="1"
score[edson]="2"
score[sebi]="3"
score[chris]="4"
echo ${!score[@]} # alex edson sebi chris
unset score[chris] # Delete chris entry
echo ${score[@]} # show all the values
echo ${!score[@]} # show all keys
echo ${score[edson]} # show the value of edson: 2
echo ${#score[@]} # show the number of elements in the map: 3
EOF
bash maps-01.sh
```

Collections

Functions provide a way to group commands in a bash script together under a common name for reuse.

Basic function syntax

The following demonstrates basic function syntax. The function is named `printmessages`. The function uses the echo command to send two messages to standard output.

```
printmessages() {
  echo "I am message 1"
  echo "I am message 2"
}
```

Copy, paste and run in your terminal:

Copy and paste the following code into your terminal window to create and execute a Bash script that has a function named `printmessages`.

```
cat << 'EOF' > function-example-01.sh
#!/usr/bin/env bash

printmessages() {
  echo "I am message 1"
  echo "I am message 2"
}

# call the function
printmessages
EOF

bash function-example-01.sh
```

Using parameters

Parameters are passed to a function implicitly when added to the execution command of the function.

Parameters are detected in a function by using the `$` symbol to call the parameter according the position of the parameter in the command line.

The following code demonstrates a function that reads the parameter passed as the first argument in the command line.

```
chelloworld() {
  echo "Hello World from $1"
  }

helloworld "Alex"
```

Copy, paste and run in your terminal:

Copy and paste the following code into your terminal window to create and execute a Bash script that has a function named `helloworld` that processes the first parameter in the command line execution.

```
cat << 'EOF' > function-example-02.sh
#!/usr/bin/env bash

helloworld() {
  echo "Hello World from $1"
  }

# call the function
helloworld "Alex"
EOF

bash function-example-02.sh
```

Returns `Hello World from Alex`

Copy, paste and run in your terminal:

Copy and paste the following code into your terminal window to create and execute a Bash script that has a function named `helloworld` that processes the two parameters in the command line execution.

```
cat << 'EOF' > function-example-03.sh
#!/usr/bin/env bash

helloworld() {
  echo "Hello World from $1 and $2"
  }

# call the function
helloworld "Alex" "Edson"
EOF

bash function-example-03.sh
```

Returns `Hello World from Alex and Edson`

Setting a global variable

A function can write data to a variable previous defined in a Bash script. The following bash script demonstrates the technique.

```
function set_favorite_food(){
  favorite_food=$1
}

favorite_food="apples"
echo favorite_food

set_favorite_food "cheese"

echo favorite_food
```

Copy, paste and run in your terminal:

```
cat << 'EOF' > function-04.sh
set_favorite_food(){
  favorite_food=$1
}

favorite_food="apples"
echo $favorite_food

set_favorite_food "cheese"

echo $favorite_food
EOF

bash function-04.sh
```

Returns

```
apples
cheese
```

Conditional statements

A conditional statement is an `if..then..else` statement. When writing a conditional statement you check to see if an expression is true or false and respond accordingly.

A simple conditional statement uses the following syntax:

```
if [<statement>]; then
   <consequence statement(s)>
fi
```

WHERE `if`, `then` and `fi` are keywords with `if` indicating the beginning of the conditional statement and `fi` indicating the end of the conditional statement.

An `if..then` conditional statement uses the `if`llowing syntax with the else keyword :

```
if [<statement>]; then
   <consequence statement(s)>
else
  <consequence statement(s)>
fi
```

Numeric statements

The following bash script demonstrates using a conditional statement to test numeric values. The code uses the `$RANDOM` function to get a random number. `$RANDOM` is defined by the operating system and always present. The `expr` keyword is the bash command that evaluates an expression. Also, the bash script uses the predefined modulus (`%`) operator which is available to the script by default from the operating system.

```
mynum=$RANDOM
echo $mynum
if [ $(expr $mynum % 2) == "0" ]; then
    echo even
else
  echo odd
fi
```

Copy, paste and run in your terminal:

Copy and paste the following code into your terminal window to create and execute a Bash script that creates a random number and then runs an `if..then..else` conditional statement to determine if the random value is even or odd.

```
cat << 'EOF' > conditional-example-01.sh
#!/usr/bin/env bash
mynum=$RANDOM
echo $mynum
if [ $(expr $mynum % 2) == "0" ]; then
    echo even
else
  echo odd
fi
EOF

bash conditional-example-01.sh
```

String statements

The following bash script demonstrates using a conditional statement to check if a word exists in a string.

```
mystring="I like cherries"
positive_indicator=" like "
if [[ "$mystring" == *"$positive_indicator"* ]]; then
  echo "It's a good review"
fi
EOF
```

Copy, paste and run in your terminal:

Copy and paste the following code into your terminal window to create and execute a Bash script that tests if certain substrings exist and do not exist in a string provided as a parameter to the script.

```
cat << 'EOF' > conditional-example-02.sh
#!/usr/bin/env bash
mystring=$1

positive_indicator=" like "
negative_indicator=" don't "

if [[ ("$mystring" == *"$positive_indicator"* ) && ( "$mystring" !=
*"$negative_indicator"* )]]; then
  echo "It's a good review."
else
  echo "It's a bad review."
fi

EOF

bash conditional-example-02.sh  "I like cherries"

bash conditional-example-02.sh  "I hate cherries"

bash conditional-example-02.sh  "I don't like cherries"

bash conditional-example-02.sh  "I like apple"
```

File statements

The following bash script demonstrates using a conditional statement to determine if a file exists.

```
FILE=/<path/to/filename>
if test -f "$FILE"; then
    echo "$FILE exists."
fi
```

Copy, paste and run in your terminal:

Copy and paste the following to create a file and then run the Bash script that checks for the file's existence.

```
touch newfile.txt

cat << 'EOF' > conditional-example-03.sh
#!/usr/bin/env bash
FILE=newfile.txt
if test -f "$FILE"; then
    echo "$FILE exists."
fi
EOF

bash conditional-example-03.sh
```

Loops

Looping is a technique that enables Bash scripts to run programming statements and expressions continuously.

The following sections describe different types of loops.

Range

The following code demonstrates running a loop over a range according to lower and upper limits.

```
for i in {1..5}; do
  echo "Hello World $i"
done
```

Copy, paste and run in your terminal:

Copy and paste the code to run a Bash script that runs a loop over 5 iterations.

```
cat << 'EOF' > basic-range-01.sh
#!/usr/bin/env bash

for i in {1..5}; do
  echo "Hello World $i"
done

EOF

bash basic-range-01.sh
```

Looping collections

The following code uses the `do` keyword to demonstrate running printing all elements from a plain array:

```
for i in "${names[@]}"; do
  echo "Hello $i"
done
```

Copy, paste and run in your terminal:

Copy and paste the code to run a Bash script that prints all the elements in an array using a `for` loop and the `@` keyword.

```
cat << 'EOF' > range-names-01.sh
#!/usr/bin/env bash

names=('Alex' 'Ada' 'Alexandra', 'Soto')

for i in "${names[@]}"; do
  echo "Hello $i"
done

EOF

bash range-names-01.sh
```

Print keys of all elements from a key/value array:

```
for key in "${!score[@]}"; do
  echo $key
done
```

Copy, paste and run in your terminal:

Copy and paste the code to run a Bash script that prints all the elements in a key/value array.

```
cat << 'EOF' > range-keys-01.sh
#!/usr/bin/env bash

declare -A score

score[alex]="1"
score[edson]="2"
score[sebi]="3"
score[chris]="4"

for key in "${!score[@]}"; do
  echo $key
done

EOF
bash range-keys-01.sh
```

Print keys of all elements from a key/value array:

```
for val in "${score[@]}"; do
  echo $val
done
```

Copy, paste and run in your terminal:

```
cat << 'EOF' > value-keys-01.sh
#!/usr/bin/env bash

declare -A score

score[alex]="1"
score[edson]="2"
score[sebi]="3"
score[chris]="4"

for val in "${score[@]}"; do
  echo $val
done

EOF
bash value-keys-01.sh
```

Files and directories

Get all files in a directory sub-directories

The following script gets all files in the directory `/tmp` that have the extension `.log`:

```
for i in /tmp/*.log; do
  echo $i
done
```

Copy, paste and run in your terminal:

```
cat << 'EOF' > files-01.sh
#!/usr/bin/env bash

echo All log files in the /tmp directory

for i in /tmp/*.log; do
  echo $i
done
EOF
bash files-01.sh
```

Get all sub-directories

The following script gets all subdirectories in the directory `/var`

```
for i in /var/*; do
  echo $(basename "$i")
done
```

Copy, paste and run in your terminal:

Copy and paste the code to run a Bash script that traverses all the subdirectories in the directory `/var`.

```
cat << 'EOF' > files-02.sh
#!/usr/bin/env bash

echo All subdirectories in /var

for i in /var/*; do
  echo $(basename "$i")
done
EOF

bash files-02.sh
```

While loop

A while loop runs continuously until a certain condition is met.

The following code uses the less then or equal to symbol `-le` to run a loop until the counter variable `x` reaches the number `5`.

`x`

```
x=1;
while [ $x -le 5 ]; do
  echo "Hello World"
  ((x=x+1))
done
```

Copy, paste and run in your terminal:

Copy and paste the following code to create and run a Bash script that demonstrates a `while` loop.

```
cat << 'EOF' > while-loop-01.sh
#!/usr/bin/env bash

x=1;
while [ $x -le 5 ]; do
  echo "Hello World"
  ((x=x+1))
done
EOF

bash while-loop-01.sh
```

Working with status codes

Reporting success and error in a Bash script is accomplished using status codes. By convention success is reported by exiting with the number `0`. Any number greater than `0` indicates an error. Also, there is a convention for error numbers which is explained in the article on Red Hat System Admin Bash command line exit codes demystified.

Using the exit keyword

The following demonstrates a Bash code that returns an error code `22` when the script is executed without a parameter.

```
if [ -z "$1" ]; then
  echo "No parameter";
  exit 22;
fi
```

Copy, paste and run in your terminal:

Copy and paste the following code to create and run a Bash script that returns an error code when the script is executed without a parameter.

```
cat << 'EOF' > status-code-01.sh
#!/usr/bin/env bash
if [ -z "$1" ]; then
  echo "No parameter";
  exit 22;
fi
EOF

bash status-code-01.sh

echo $?
```

Returns 22

Return a status value from a function

The following code demonstrates using the `return` keyword to return a status code from a function in a Bash script.

```
function echoMessage(){
  if [ -z "$1" ]; then
   return 22;
  fi
}
```

```
cat << 'EOF' > status-code-02.sh
#!/usr/bin/env bash
function echoMessage(){
  if [ -z "$1" ]; then
   return 22;
  fi

  echo $1 after the IF/THEN STATEMENT
}

echoMessage
res=$?
echo The first result of the call to echoMessage is $res

echoMessage "Bash Rocks!"

res=$?
echo The second result of the call to echoMessage is $res

EOF

bash status-code-02.sh
```

Returns

```
The first result of the call to echoMessage is 22
Bash Rocks! after the IF/THEN STATEMENT
The second result of the call to echoMessage is 0
```