

Module 3 Cheat Sheet - Introduction to Shell Scripting

Bash shebang

```
1. 1
1. #!/bin/bash
```

Copied!

Get the path to a command

```
1. 1
1. which bash
```

Copied!

Pipes, filters, and chaining

Chain filter commands together using the pipe operator:

```
1. 1
1. ls | sort -r
```

Copied!

Pipe the output of manual page for `ls` to head to display the first 20 lines:

```
1. 1
1. man ls | head -20
```

Copied!

Use a pipeline to extract a column of names from a csv and drop duplicate names:

```
1. 1
1. cut -d "," -f1 names.csv | sort | uniq
```

Copied!

Working with shell and environment variables:

List all shell variables:

```
1. 1
1. set
```

Copied!

Define a shell variable called `my_planet` and assign value `Earth` to it:

```
1. 1
1. my_planet=Earth
```

Copied!

Display value of a shell variable:

```
1. 1
1. echo $my_planet
```

Copied!

Reading user input into a shell variable at the command line:

```
1. 1
1. read first_name
```

Copied!

Tip: Whatever text string you enter after running this command gets stored as the value of the variable `first_name`.

List all environment variables:

- 1. 1
- 1. env

Copied!

Environment vars: define/extend variable scope to child processes:

- 1. 1
- 2. 2
- 1. export my_planet
- 2. export my_galaxy='Milky Way'

Copied!

Metacharacters

Comments #:

- 1. 1
- 1. # The shell will not respond to this message

Copied!

Command separator ;:

- 1. 1
- 1. echo 'here are some files and folders'; ls

Copied!

File name expansion wildcard *:

- 1. 1
- 1. ls *.json

Copied!

Single character wildcard ?:

- 1. 1
- 1. ls file_2021-06-???.json

Copied!

Quoting

Single quotes '' - interpret literally:

- 1. 1
- 1. echo 'My home directory can be accessed by entering: echo \$HOME'

Copied!

Double quotes "" - interpret literally, but evaluate metacharacters:

- 1. 1
- 1. echo "My home directory is \$HOME"

Copied!

Backslash \ - escape metacharacter interpretation:

- 1. 1
- 1. echo "This dollar sign should render: \\$"

Copied!

I/O Redirection

Redirect output to file and overwrite any existing content:

```
1. 1
1. echo 'Write this text to file x' > x
```

Copied!

Append output to file:

```
1. 1
1. echo 'Add this line to file x' >> x
```

Copied!

Redirect standard error to file:

```
1. 1
1. bad_command_1 2> error.log
```

Copied!

Append standard error to file:

```
1. 1
1. bad_command_2 2>> error.log
```

Copied!

Redirect file contents to standard input:

```
1. 1
1. $ tr "[a-z]" "[A-Z]" < a_text_file.txt
```

Copied!

The input redirection above is equivalent to:

```
1. 1
1. $cat a_text_file.txt | tr "[a-z]" "[A-Z]"
```

Copied!

Command Substitution

Capture output of a command and echo its value:

```
1. 1
2. 2
1. THE_PRESENT=$(date)
2. echo "There is no time like $THE_PRESENT"
```

Copied!

Capture output of a command and echo its value:

```
1. 1
1. echo "There is no time like $(date)"
```

Copied!

Command line arguments

```
1. 1
1. ./My_Bash_Script.sh arg1 arg2 arg3
```

Copied!

Batch vs. concurrent modes

Run commands sequentially:

```
1. 1
1. start=$(date); ./MyBigScript.sh ; end=$(date)
```

Copied!

Run commands in parallel:

```
1. 1
1. ./ETL_chunk_one_on_these_nodes.sh & ./ETL_chunk_two_on_those_nodes.sh
```

Copied!

Scheduling jobs with cron

Open crontab editor:

```
1. 1
1. crontab -e
```

Copied!

Job scheduling syntax:

```
1. 1
1. m h dom mon dow    command
```

Copied!

(minute, hour, day of month, month, day of week)

Tip: You can use the * wildcard to mean "any".

Append the date/time to a file every Sunday at 6:15 pm:

```
1. 1
1. 15 18 * * * 0 date >> sundays.txt
```

Copied!

Run a shell script on the first minute of the first day of each month:

```
1. 1
1. 1 1 0 1 * * ./My_Shell_Script.sh
```

Copied!

Back up your home directory every Monday at 3:00 am:

```
1. 1
1. 0 3 * * * 1 tar -cvf my_backup_path\my_archive.tar.gz $HOME\
```

Copied!

Deploy your cron job:

Close the crontab editor and save the file.

List all cron jobs:

```
1. 1
1. crontab -l
```

Copied!

Conditionals

if-then-else syntax:

```
1. 1
2. 2
3. 3
4. 4
5. 5
6. 6

1. if [[ $# == 2 ]]
2. then
3.   echo "number of arguments is equal to 2"
4. else
5.   echo "number of arguments is not equal to 2"
```

```
6. fi
```

Copied!

'and' operator &&:

```
1. 1
1. if [ condition1 ] && [ condition2 ]
```

Copied!

'or' operator ||:

```
1. 1
1. if [ condition1 ] || [ condition2 ]
```

Copied!

Logical operators

Operator	Definition
==	is equal to
!=	is not equal to
<	is less than
>	is greater than
<=	is less than or equal to
>=	is greater than or equal to

Arithmetic calculations

Integer arithmetic notation:

```
1. 1
1. $(( ))
```

Copied!

Basic arithmetic operators:

Symbol	Operation
+	addition
-	subtraction
*	multiplication
/	division

Display the result of adding 3 and 2:

```
1. 1
1. echo $((3+2))
```

Copied!

Negate a number:

```
1. 1
1. echo $((-1*-2))
```

Copied!

Arrays

Declare an array that contains items 1, 2, "three", "four", and 5:

```
1. 1
1. my_array=(1 2 "three" "four" 5)
```

Copied!

Add an item to your array:

```
1. 1
2. 2
1. my_array+= "six"
```

```
2. my_array+=7
```

Copied!

Declare an array and load it with lines of text from a file:

```
1. 1
1. my_array=$(echo $(cat column.txt))
```

Copied!

for loops

Use a for loop to iterate over values from 1 to 5:

```
1. 1
2. 2
3. 3
1. for i in {0..5}; do
2.     echo "this is iteration number $i"
3. done
```

Copied!

Use a for loop to print all items in an array:

```
1. 1
2. 2
3. 3
1. for item in ${my_array[@]}; do
2.     echo $item
3. done
```

Copied!

Use array indexing within a for loop, assuming the array has seven elements:

```
1. 1
2. 2
3. 3
1. for i in {0..6}; do
2.     echo ${my_array[$i]}
3. done
```

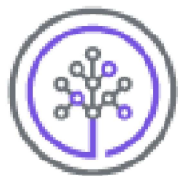
Copied!

Authors

Jeff Grossman
Sam Propupchuk

Other Contributors

Rav Ahuja



Skills Network