Shell & Bash Scripting

(Learning from DataCamp's *Introduction to Shell* and *Introduction to Bash Scripting)*

1) TERMINOLOGY

Term	Explanation	Other notes
Operating system	- Controls computer's processor, hard drive, network connection and run other programs	
Graphical file explorer	- Translates clicks and double-clicks into commands to open files and run programs	
Command-line shell	- Processes instructions typed by users - Each time a command is entered: 1. Shell runs some other programs 2. Prints their output in human-readable form 3. Displays a *prompt* to signal that it's ready to accept the next command - Name comes from the notion that it's the "outer shell" of the computer - Ability to combine old commands to create new ones and automate repetitive operations with just a few keystrokes	
Filesystem	- Manages files and directories/folders - Each filesystem is identified by an **absolute path** which shows how to reach it from the filesystem's **root directory**	
Working directory	- Where the shell runs commands and looks for files	
Home directory	- Usually your starting directory in the Shell	
Absolute path	- Begins with "/"	
Relative path	- Specifies location starting from current file directory - Does not begin with "/"	
Parent of a directory	- Directory that is above a directory	
/tmp	- A directory where people and programs often keep files that they only need briefly - It is immediately below the root directory	
Tab completion	- Shell tries its best to auto-complete the name of a file/directory - Pressing tab a second time will display a list of possibilities if the path is ambiguous - Typing another character or two to make your path more specific and then pressing tab will fill in the rest of the name	
Command-line flag	- "Flag" for short - Considered good style to put all flags *before* any file names	
Redirection	- Save any command's output anywhere you want	
Wildcard	- Most command wildcard "*": "match zero or more characters"	
CTRL-C / ^C / ^c	- Stop a running program	
Environment variables	- Information stored by shell - Are available all the time	- By convention, environment variables are written in upper case

Variable	Purpose	Value
HOME	User's home directory	/home/repl
PWD	Present working directory	Same as pwd command
SHELL	Which shell program is being used	/bin/bash
USER	User's ID	repl
OSTYPE	Operating system in use	

Shell variable	- Similar to a local variable in a programming language	
Shell script	- A file full of shell commands - Sometimes called "*script*" for short - Don't have to have names ending with ".sh" but is a convention to keep track of which files are scripts - Can also contain pipes - Can be used in combination with redirection	
Not providing filenames	- Command mistake in shell scripts and commands - Commands waits to read input from keyboard	Example: tail -n 3
		Example: head -n 5 | tail -n 3 somefile.txt - *tail* goes ahead and prints the last 3 lines of *somefile.txt* - *head* waits forever for keyboard input since it wasn't given a filename and there isn't anything ahead of it in the pipeline
Standard streams	- In Bash scripting, there are 3 'streams' for your programs 1) **STDIN** (standard input) - A stream of data into the program 2) **STDOUT** (standard output) - A stream of data *out* of the program 3) **STDERR** (standard error) - Stream where errors and exceptions in program are written to - By default, these streams will come from and write out to the terminal - However, some scripts may be called ("*script calls*") with 2> /dev/null (redirecting *STDERR* to be deleted) or 1> /dev/null (*STDOUT*)	Example using standard output: cat cute.txt 1> cuter.txt
ARGV	Term to describe array of all the arguments given to the script (after the command, separated by a whitespace) - Access each argument with "$" notation	$1: First arg $2: Second arg $@, $*: All args in *ARGV* $#: Number of arg
Indexed array	- Normal numerical-indexed structure - Similar to Python's *lists* or R's *vectors*	
Associative array	- Similar to normal array but with key-value pairs - Similar to Python;s *dictionary* or R's *list* - Only available in Bash 4 onwards	
Scope	How accessible a variable is in a program - **GLOBAL** - Accessible anywhere in the program - Includes inside for-loops, if-statements, functions etc. - **LOCAL** - Only accessible in a certain part of the program	

2) SHELL COMMANDS

Command /Special Path	Usage	Other
pwd	- Finds out where you are in the filesystem, prints current working directory - Short for "*print working directory*"	
ls	- Lists contents of current working directory	
cd	- Move around the filesystem - Short for "*change directory*"	
..	- Special path (two dots with no spaces) - Means "the directory above the one I'm currently in"	
.	- Special path (single dot on its own) - Means the "current directory"	ls and ls. does the same thing and cd . has no effect because it moves you into the directory you're currently in
~	- Special path (tilde) - Means "your home directory"	
ls ~	- No matter where you are, list the contents of your home directory	
ls -R	- List everything in a directory, no matter how deeply nested - Short for "recursive" - Shows every file and directory in the current level, then everything in each sub-directory and so on	
ls -R -F	- Prints a "/" after the name of every directory - Prints a "*" after the name of every runnable program	
cd ~	- No matter where you are, take you home	
cp	- Short for "copy"	
cp <existing file> <destination file>	- Creates a copy of the file - If already exists a file of same name, file will be overwritten	- Example: copy file *original.txt* and rename copied file as *duplicate.txt* cp original.txt duplicate.txt
cp in1.txt in2.txt dir2	- Copy multiple files into the same destination directory - Last parameter to *cp* is an existing directory - All files mentioned are copied to that directory	
mv	- Moves a file from one directory to another	
mv <old file name> <new file name>	- Rename a file	- Warning: *mv* will overwrite existing files (as it the contents were "moved" to a new file) - Be careful not to rename it to an existing file name
mv <new dir name> <old dir name>	- Rename a directory	
rm	- Short for "remove"	
rm in1.txt in2.txt in3.txt	- Remove multiple files	- Warning: once command is entered, files are gone for good (not into the trash can)
rmdir	- Deletes an **empty** directory (for safety) - To delete a directory full of work, delete the files in the directory first	
mkdir	- Creates a new empty directory	

cat <file1> <file2> ⋯	- View a/many files' contents - Short for "concatenate", meaning to "link things together" - Prints the contents of the file/s onto the screen, one after another	

Concerning contents of file(s)			
less <file>	- Display one page of the file at a time - Usually more convenient to *page* the output		- *Spacebar* (page down) - Command *q* (quit)
less <file1> <file2> ⋯	- Display the pages of multiple files		- Command *:n* (move to next file) - Command *:p* (go to prev file) - Command *:q* (quit)
head <file>	- Select **rows** - Prints the first 10 lines of a file - Good for csv files		
head -n <int> <file>	- Display the first lines of a file, where number of lines shown is specified by an integer - Short for "number of lines"		
head -n +<int> <file>	- Display all but first <*int*> minus one lines of file		
tail	- Select **rows** - Prints the last 10 lines of a file		
cut	- Select **columns** from a file - Note: cannot understand quoted strings (e.g. "Roger, Frank")		Example: cut -f 2-5,8 -d , values.csv - Selects columns 2 to 5 and column 8 - "-f" for "fields" to select specify columns - "-d" for "delimeter" to specify separator separating columns - Note adding space after flag is not compulsory (e.g. "f1") but is good style
grep grep <keyword> <file1> <file2> ⋯	- Select **lines** according to what they contain		Example: to print lines from *winter.csv* that contain "bicuspid" grep bicuspid seasonal/winter.csv
	Flags for *grep*	-c	- Print a count of matching lines rather than the lines themselves
		-h	- Do *not* print the names of the files when searching multiple files
		-i	- Ignore case
		-l	- Print the names of the files that contain the matches, not the matches
		-n	- Print the line numbers for matching lines
		-v	- Invert the match (i.e. only show lines that *don't* match)
		-E	- Same egrep - Example: grep -e 'You are|just fine' file.txt (match either phrase)
sed 's/<pattern>/<replacement>/g'	- Pattern-matched **string replacement** - Format:		

paste	- Combine data files instead of cutting them up - Similar to *cbind*	- Note: joining files with different number of rows can cause certain rows to have different number of columns (the file with more rows)
sort	- Sorts lines of data in order - (Default) Ascending alphabetical order	
sort -n	- Sort numerically	
sort -r	- Sort and reverse order of output	
sort -b	- Sort and ignore leading blanks	
sort -f	- Sort and fold case (i.e. case-insensitive)	
uniq	- Remove duplicated **adjacent** lines - Command often used with *sort*	- Reason: *uniq* was built to work with very large files, hence need to prevent keeping whole file in memory
uniq -c	- Display unique lines with a count of how often each occurs	

Others		
man	- Find out what commands do - Short for "manual" - Automatically invokes *less*	- *Spacebar* (page down) - Command *:q* (quit)
	- Under "NAME" - One-line description - Briefly tells what the command does - Under "SYPNOPSIS" - Lists all the flags it understands - Anything optional is shown in square brackets "[…]" - Either/or alternatives are separated by "|" - Things that can be repeated are shown by "…"	
history	- Prints a list of commands run recently - Each command is preceded by a serial number to make it easy to re-run particular commands - Most recent command at the bottom/end of the list	Output example: 1 head summer.csv 2 cd seasonal 3 head summer.csv 4 history
!<int>	- Rerun a particular command, the <*int*>th command in your history	Example: !55 To rerun the most recent use of a command: !head !cut
<command> > <output file>	- Save output of command into a file	
<left command> | <right command>	- Pipe command - Tells the shell to use the output of the command on the left as the input to the command on the right - You can chain any number of commands together (use pipe as many times as you want)	
wc	- Short for "word count" - Prints the number of characters, words and lines in a file	To print only one of chars, words or lines, use flags: -c, -w, -l

<command> <filepath>/*.<file extension>	- Specify many files at once		Or: <command> <filepath>/*
Wildcards		**Meaning**	**Examples**
	*	Matches one or more character	*.csv
	?	Matches a single character	201?.txt
	[···]	Matches any one of the characters inside the square brackets	201[78].txt
	{···}	Matches any of the comma-separated patterns inside the curly brackets	{*.txt, *.csv}
for <var> in <···>; do <command with $var>; done	- Loop in shell - Uses semi-colons - Good to use with files and wildcards - Do more things in each loop using pipe operator		Examples: for filetype in gif jpg png; do echo $filetype; done for filename in seasonal/*.csv; do echo $filename; done for filename in $shellvar; ···
	- Loop in shell, with multiple actions per loop - Separate actions in 'do' with semi-colon (😉		Example: for f in seasonal/*.csv; do echo $f; head -n 2 $f | tail -n 1; done
bash filename.sh	- Save commands to re-run later		- Save your commands in a bash script
Variables			
set	- Get complete long list of environment variables		
echo $<variable name>	- Find a variable's value - **Always** reference variable with a "$" preceding it		Example: echo USER
<var name>=<file name, string, etc>	- Create a shell variable - **Do *not* include spaces** before or after the equals sign		Example: datasets=seasonal/*.csv
File editor			
nano <filename>	- Open file for editing (or create it if it doesn't already exist)		
nano	**Control-key combinations**		
	CTRL-K	Delete a line	
	CTRL-U	Un-delete a line	
	CTRL-O	Save the file, stands for "output"	Press *Enter* to confirm.
	CTRL-K	Copy a line	Navigate to line first
	CTRL-U	Paste a line	Navigate to line first
	CTRL-X	Exit editor	

3) BASH SCRIPTING

- Stands for *Bourne Again Shell (p*un for *Bourne Shell)*
- Developed in the 80s
- Unix systems for backbone of internet and servers (runs ML models, data pipelines)
- Usually starts with #!/usr/bash so that interpreter knows it's a Bash script and to use bash located in /usr/bash

which bash	Check where Bash is installed	
bash --version	Check version of Bash	
.sh	File extension - Technically not needed if first line has the she-bang and path to Bash	
date	Current date and time	
bash filename.sh ./filename.sh	- Save commands to re-run later - Second option: when she-bang available in script	- Save your commands in a bash script
$@	- Special expression to pass filenames to scripts - Means "all of the command-line parameters given to the script"	
bash file.sh <inputfile1> <inputfile2> ···	- Shell replaces "$@" with file(s) given to be processed - Processes each file one by one as if in a loop	Example: *unique-lines.sh* has sort $A | uniq bash unique-lines.sh seasonal/summer.csv
$1, $2, ···	- Process multiple specific arguments/ command-line parameters	
# print for f in $@ do <command1> <command2> ... done	- Write loops in shell script - 2 methods: 1) Using semi-colons 2) Split structure across lines without semi-colons to make them more readable (indentation is not compulsory)	
grep "$1", "$1".csv ...	Using arguments in commands and filenames (use quotes)	

Arguments

Special *ARGV* properties	
$1, $2, ···	Access each argument in *ARGV*
$@, $*	All arguments in *ARGV*
$#	Gives length/number of arguments
$?	Return value of function

Failure code meanings:
127 - cannot find program that does not exists (e.g. "echlo" instead of "echo")

Evaluating expressions

Single quotes	Shell interprets what is between literally - Will treat the string as it is (even with "$")	Example: 'sometext'
Double quotes	Shell interprets literally **except** using '$' notation and backticks (it will understand variables etc)	Example: "sometext"
Backticks	Creates a 'shell-within-a-shell' - Shell runs command within backticks and captures *STDOUT* back into a variable - Older, more backwards compatible	Example: `sometext`
Parentheses	Same as backticks, evaluates command between it	Example: $(date)

	- Alternative to using backticks - Newer, used in more modern applications	$(5 + 7)
Double parentheses	Variant on single bracket variable notation for **numeric variables** - Note: this method uses "*expr*", so cannot manipulate decimal places	Example: expr 5 + 7 echo $((5 + 7)) will both return 12
Arithmetic expressions		
expr	Useful utility program for **arithmetic** - Limitation: cannot handle decimal places	Example: expr 1 + 4 return 5
bc	Useful command-line program - Stands for "basic **calculator**" - Benefits: handles decimal places - Opens calculator, close using "quit"	Example: To avoid opening the calculator echo "5 + 7.5" | bc returns 12.5
bc "scale=<int>; <expression>"	Specifies number of decimal places of output - Use semi-colons to separate lines in terminal	Example: echo "scale=3; 10 / 3" | bc returns 3.333
Indexed arrays		
declare -a arr_name arr_name=(<elements separated by whitespace>)	Create numerical-indexed array - Two methods: 1. Decare without adding data elements (use -a flag) 2. Create and add elements at the same time (looks like tuple) - Note: if string, use **double quotes**	Example: my_first_array=(1 2 3) string_arr=("A" "Bee Cat" "D")
arr[@]	Return all array elements - Note: Bash requires curly brackets around array name when accessing these properties	Example: echo ${my_array[@]}
#arr	Return length of array - Note: use curly brackets	Example: echo ${#my_array[@]}
arr[idx]	Return a single element - Note: Bash uses zero-indexing	Example: echo ${my_first_array[2]}
arr[idx]=<new value>	Change array values - Use index notation - Don't use "$" notation when overwriting an index	Example: echo ${my_first_array[0]}
arr:<start_idx> :<num_ele>	Slice an array - Specify starting index and number of elements to return after the index	Example: echo ${arr[@]:2:2}
arr+=(<new ele>)	Append to an array - Note: use parentheses (or else it just concatenates the input to the first element like a string)	Example: arr+=(10)
Associative arrays		
- Use declare syntax, either 1) Declare first, then add elements 2) Do it all on one line - Surround 'keys' in square brackets, then associate a value after the equals sign (can add multiple elements at once)		
#declare first declare -A arr_name #add elements	**Method 1** - Declare first, then add elements	Example: declare -A city_details

<table>
<tr><td>arr_name=([key]=value ···)</td><td></td><td>city_details[city_name]="New York"
city_details[population]=140</td></tr>
<tr><td>declare -A arr_name=([key]=value ···)</td><td>Method 2
- Declare and add elements in one line</td><td>Example:
declare -A city_detail=([city_name]="New York" [population]=140)</td></tr>
<tr><td>arr[key]</td><td>Return a value in the array using the key</td><td>Example:
echo ${city_details[city_name]}</td></tr>
<tr><td>!arr[@]</td><td>Return all keys in array
- Use exclamation mark</td><td>Example:
echo ${!city_details[@]}</td></tr>
<tr><td colspan="3">Conditional expressions</td></tr>
<tr><td>Alternatives to traditional comparison operators</td><td><table><tr><th>Flag</th><th>Meaning</th><th>Usually</th></tr><tr><td>-eq</td><td>Equal to</td><td>=</td></tr><tr><td>-ne</td><td>Not equal to</td><td>!=</td></tr><tr><td>-lt</td><td>Less than</td><td><</td></tr><tr><td>-le</td><td>Less than or equal to</td><td><=</td></tr><tr><td>-gt</td><td>Greater than</td><td>></td></tr><tr><td>-ge</td><td>Greater than or equal to</td><td>>=</td></tr></table></td><td>Example:
x=10
if [$x -gt 5]; then
  echo "$x is more than 5!"
fi
returns
10 is more than 5!</td></tr>
<tr><td>Other bash conditional flags</td><td><table><tr><th>Flag</th><th>Meaning</th></tr><tr><td>-e</td><td>If file exists</td></tr><tr><td>-s</td><td>If file exists and has size greater than zero</td></tr><tr><td>-r</td><td>If file exists and is readable</td></tr><tr><td>-w</td><td>If file exists and is writable</td></tr></table></td><td></td></tr>
<tr><td>Combining conditions</td><td><table><tr><td>&&</td><td>AND</td></tr><tr><td>||</td><td>OR</td></tr></table></td><td>Example:
if grep -q 'SRVM' $1 && grep -q 'vpt' $2; then
  ...</td></tr>
<tr><td>Multiple conditions</td><td><table><tr><td>Multiple square bracket conditions</td><td>… [CONDITION] && [CONDITION] …</td></tr><tr><td>Double-square-bracket notation</td><td>… [[CONDITION && CONDITION]] …</td></tr></table></td><td>Example:
x=10
if [$x -gt 5] && [$x -lt 11]; then
  echo ···
fi

Example:
if [[$x -gt 5 && $x -lt 11]]; then
  echo ···
fi</td></tr>
<tr><td>Using command-line programs directly in conditional</td><td>- Remove the square brackets</td><td>Example:
if grep -q 'Hello' words.txt; then
  echo "Hello is inside!"
fi</td></tr>
<tr><td>Using a shell-within-a-shell</td><td>Notation: "$(···)"</td><td>Example:
if $(grep -q 'Hello' words.txt); then
  echo···
fi</td></tr>
<tr><td colspan="3">IF-statements</td></tr>
<tr><td>if [CONDITION]; then</td><td>General if-statement format</td><td>Example:</td></tr>
</table>

# SOME CODE else # SOME OTHER CODE fi	- Ends with "*fi*" which is a reverse "if" - Note: 1) Put spaces between square brackets [ ] and conditional elements in first line 2) Put semi-colon after close-bracket	x="Queen" if [$x == "King"]; then echo "$x is a King!" else echo "$x is not a King!" fi returns Queen is not a King!
if ((CONDITION)); then ...	For numerical comparisons - Use double parentheses to evaluate the expressions	Example: x=10 if (($x > 5)); then echo "$x is more than 5!" fi
FOR-loops		
for x in <elements> do <command> done	General for-loop format	Example: for x in 1 2 3 do echo $x done
for x in {START..STOP..INCREMENT} do <command> done	Brace expansion - Shortcut to create numeric ranges	Example: for x in {1..5..2} do echo $x done returns 1 3 5
for ((START EXP; TERMINATING COND; INCREMENT/DECREMENT)) ...	'Three expression' syntax - Another way to write for-loops - Surround three expressions with double parenthesis	Example: for ((x=2;x<=4;x+=2)) do echo $x done
Using Glob expansions	- Allows pattern-matching expansions into a for-loop using the wildcard * symbol - For example, files in a directory	Example: for book in books/* do echo $book done # prints each filename in new line
Using shell-within-a-shell	- Notation: $()	Example: for book in $(ls books/ | grep -I 'air') do echo $book done
WHILE-loops		
while [CONDITION]; do ... done	- Iterations continue until condition is no longer met - Surround conditions in square brackets - Use same **flags for numerical comparison** from IF-statements (e.g. -le) - Make sure there a change within such that the loop will terminate	Example: x=1 while [$x -le 3]; do echo $x ((x+=1)) done

		returns 1 2 3

CASE statements

- More optimal than IF-statements when there are multiple or complex conditional

case 'STRINGVAR' in PATTERN1) COMMAND1;; PATTERN2) COMMAND2;; … *) DEFAULT COMMAND:: esac	1) First select which variable or string to match against - Can use shell-within-shell here 2) Add as many possible matches and actions desired - Separate pattern and code with close-parenthesis - Finish commands with **double semi-colons** - Can use regex for PATTERN - Example: starts with air (Air*), contains hat (*hat*) 3) End with default statement	Example: case $(cat $1) in # case 1 *sydney*) my $1 sydney/ ;; # case 2 *melbourne*|*brisbane*) rm $1 ;; # case 3 *canberra*) mv $1 "IMPORTANT_$1" '' # default *) echo "No cities found" ;; esac

Bash FUNCTIONS

→ **Advantages**
1) Reusable
2) Allow neat, compartmentalized (modular) code
3) Aids sharing code (to use, only need to know inputs and outputs)

→ **Scope**
- All variables global by default
- E.g. variables declared in functions are accessible outside it

→ **Return values**
- Only meant to determine if the function was a success (0 – zero) or failure (1-255 – other values)
- This is captures in the global variable $?
- To get values out of function:
  1) Assign to global variable and use from there
  2) Echo back the needed variables (**last line** in function) and capture using shell-within-a-shell

function_name () { # function_code return #something }	Basic function structure - Able to **optionally** return something	
function fn_name { # function_code return #something }	Alternate function structure - Denote a function build with keyword *function* - Drop parenthesis on opening line (optional)	
function_name	Call a Bash function by writing its name	
function fn_name { local var_name=$… … }	Restrict the scope of a variable in a function to local - Place keyword *local* in front of variable declaration	- If variable called outside of scope/function, a *blank line* is printed (not an error) - Because variable is assigned to the **global** *ARGV* element ($...) - And function was run without arguments

Example 1	Function that converts Fahrenheit to Celsius
	temp_f=30 function convert_temp () {

	temp_c=$(echo "scale=2; ($temp_f – 32) * 5 / 9" | bc) echo $temp_c } convert_temp # call the function
	Returns -1.11
Example 2	Function that loops through a list of user input filenames and printing them out
	function print_filename { echo "The first file was $1" for file in $@ do echo "This file has name $file" done } print_filename "LOTR.txt" "mod.txt", "A.py"
	Returns The first file was LOTR.txt This file has name LOTR.txt This file has name mod.txt This file has name A.py
Example 3	Function that uses *echo* to return a value to be used somewhere else in the script with shell-within-a-shell capture - No longer need to create intermediary variable
	function convert_temp { echo $(echo "scale=2; ($1 -32) * 5 / 9" \| bc) } converted=$(convert_temp 30) echo "30F in Celsius is $converted C"
	Returns 30F in Celsius is -1.11 C

SCHEDULING scripts

- Situations to use scheduling:
  1) Regular tasks that need to be done (e.g. daily, weekly, multiple times per day)
  2) Optimal use of resources (e.g. running scripts in early hours of morning)
- Schedule with *cron*
  - Part of unix-like systems since 70s
  - Name comes from Greek word for 'time', *chronos*
  - Driven by *crontab* (a file that contains *cronjobs*)
  - *Cronjobs* tell *crontab* what code to run and when

crontab -l	See what schedules/*cronjobs* are currently programmed	
crontab -e	Edit list of cronjobs	

Crontab and cronjob structure

```
# ┌──────────── minute (0 - 59)
# │ ┌────────── hour (0 - 23)
# │ │ ┌──────── day of the month (1 - 31)
# │ │ │ ┌────── month (1 - 12)
# │ │ │ │ ┌──── day of the week (0 - 6) (Sunday to Saturday;
# │ │ │ │ │                                7 is also Sunday on some systems)
# │ │ │ │ │
# │ │ │ │ │
# * * * * * <command to execute>
```

- Each **line** in the *crontab* file is a *cronjob*
- There can be as many jobs as you like in a *crontab*
- Each **star** (* * * * *) represents a time unit (i.e. minutes, hours etc), in total 5 stars to set
- Default, a single star (*) means to "run at every interval"

Create cronjob - Write schedule (structure given below) in file editor		Success message: "*crontab: installing new crontab*"
X _ _ _ _ bash script.sh	Schedule a script to run at a certain time - Allows for repeated and frequency to be set	Fill in the blanks with integers, subject to the range written in the structure above
X,X,X * * * * bash ···	Schedule with a more **specific interval** - Use *commas*	
X/X/X * * * * bash ···	Schedule jobs for "**every X increment**" - Use *slashes*	

Example 1	- *Minutes* star: **5** (5 mins past the hour) - *Hours* star: **1** (after 1 am) - *Day of month, month, day of week* stars: *** * *** (every day and month)
	5 1 * * * bash myscript.sh
	Overall, script is run everyday at 1:05am
Example 2	- *Minutes* star: **15** (15 mins past the hour) - *Hours* star: **14** (after 2pm) - *Day of month, month* stars: *** *** (every day, every month of the year) - *Day of week* star: **7** (on Sundays)
	15 14 * * 7 bash myscript.sh
	Overall, script is run at 2.15pm every Sunday.
Example 3	- *Minutes* star: run at the **15, 30 and 45** minutes mark for whatever hours are specified by second star - *Hours, Day* stars etc: Every hour, every day etc
	15,30,45 * * * * bash myscript.sh
	Overall, run 3 times per hour, every day.
Example 4	
	*/15 * * * * bash myscript.sh
	Overall, run **every 15 minutes** and also for **every hour**, day etc