SECTION 1

BASH SCRIPT STRUCTURE

SECTION CHEAT SHEET

THE 3 CORE COMPONENTS OF A BASH SCRIPT

1

SHEBANG LINE

This will pretty much always be #!/bin/bash

2

BASH COMMANDS

These will depend on the task at hand

3

EXIT STATEMENT

0 = Success
Non-Zero = Failure

Official Exit Code Guidance

THE 5 PROFESSIONAL COMPONENTS OF A BASH SCRIPT

```
# 1) Author: John Doe
# 2) Created: 7th July 2020
# 3) Last Modified: 7th July 2020

# 4) Description:
# Creates a backup in ~/bash_course folder of all files in the home directory

# 5) Usage:
# backup_script
```

HOW TO SET SECURE PERMISSIONS

chmod <octal code> <file>

- Find your octal code on permissions-calculator.org
- By default, go with **744** (rwxr--r--) or **754** (rwxr-xr--)

MAKING YOUR SCRIPTS ACCESSIBLE FROM ANY FOLDER

1. Edit your ~/.profile file to add a custom folder to your PATH

```
export PATH="$PATH:/path/to/script_directory
```

2. Reload the ~/.profile file

```
$ source ~/.profile
```

3. Add your scripts to the new folder and run like normal commands!

```
$ mv my_script script_directory
```

4. You can now run your scripts like regular commands!

```
$ my_script
```

Note: Scripts must have execute permissions to run.

SECTION 2

VARIABLES AND SHELL EXPANSIONS

SECTION CHEAT SHEET

PARAMETERS

DEFINITION:
Parameters are entities that store values

THERE ARE 3 TYPES OF PARAMETERS

VARIABLES

DEFINITION:
Variables are parameters that you can change the value of

2 TYPES OF VARIABLES

1. USER-DEFINED VARIABLES

2. SHELL VARIABLES
   - BOURNE SHELL VARIABLES
   - BASH SHELL VARIABLES

SETTING THE VALUE OF A VARIABLE

```
name=value
```

Note 1: There should be no spaces around the equals sign
Note 2: Names of user-defined variables should be all lowercase

SOME COMMON SHELL VARIABLES

HOME	Absolute path to the current user's home directory
PATH	List of directories that the shell should search for executable files
USER	The current user's username
HOSTNAME	The name of the current machine
HOSTTYPE	The current machine's CPU architecture
PS1	The terminal prompt string

Link to a list of Bourne shell variables

Link to a list of Bash shell variables

PARAMETER EXPANSION

SYNTAX:

- Simple Syntax: $parameter
- Advanced Syntax: ${parameter}

DEFINITION:

Parameter expansion is used to retrieve the value stored in a parameter

PARAMETER EXPANSION TRICKS

1. `${parameter^}`
   Convert the first character of the parameter to uppercase
2. `${parameter^^}`
   Convert all characters of the parameter to uppercase
3. `${parameter,}`
   Convert the first character of the parameter to lowercase
4. `${parameter,,}`
   Convert all characters of the parameter to lowercase
5. `${#parameter}`
   Display how many characters the variable's value contains
6. `${parameter : offset : length}`
   The shell will expand the value of the parameter starting at character number defined by "offset" and expand up to a length of "length"

Note: None of these alter the value stored in the parameter. They just change how it is displayed after the expansion.

Link to list of more parameter expansion tricks

COMMAND SUBSTITUTION

DEFINITION:

Command Substitution is used to directly reference the result of a command

Syntax for command substitution

```
$(command)
```

ARITHMETIC EXPANSION

DEFINITION :

Arithmetic Expansion is used to perform mathematical calculations in your scripts.

Syntax for Arithmetic Expansion

```
$(( expression ))
```

ARITHMETIC OPERATORS RANKED IN ORDER OF PRECEDENCE (HIGHEST PRECEDENCE FIRST):

OPERATOR(S)	MEANING(S)	COMMENTS
( )	Parentheses	Anything placed in parentheses is given the highest precedence and is always run first.
**	Exponentiation. 2**4 means 2 to the power of 4, which is 16	
*, /, and %	Multiplication, Division, and Modulo. Modulo calculates the remainder of a division.	These have the same precedence.
+ and -	Addition and substraction	These have the same precedence.

Note: When two operators have the same precedence, the one furthest to the left gets performed first.

THE BC COMMAND

Using the bc command

```
echo "expression" | bc
```

Using the scale variable to control the number decimal places shown

```
echo "scale=value; expression" | bc
```

TILDE EXPANSION

DEFINITION:

Tilde expansion provides various shortcut for referencing folders on the command line.

SYNTAX	MEANING
~	The current value of the **$HOME** shell variable (usually the current user's home directory)
~username	If username refers to a valid user, give the path to that user's home directory
~-	The current value stored in the **$OLDPWD** shell variable
~+	The current value stored in the **$PWD** shell variable

Note: **$PWD** stores the current working directory and **$OLDPWD** stores the previous working directory

BRACE EXPANSION

DEFINITION:

A way of automatically generating text according to a certain pattern.

SYNTAX	MEANING
{1,2,3,4,5}	1 2 3 4 5
{1..5}	1 2 3 4 5
{a..e}	a b c d e
{1..5..2}	1 3 5
Month{01..12}	Month01, Month02, Month03, Month04, Month05, Month06, Month07, Month08, Month09, Month10, Month11, Month12
file{1..5}.txt	file1.txt file2.txt file3.txt file4.txt file5.txt
~/{Documents, Downloads}/ file{1..2}.txt	~/Documents/file1.txt ~/Documents/file2.txt ~/Downloads/file1.txt ~/Downloads/file2.txt

Note: There should be no spaces around any commas or double dots (..)

SECTION 3

HOW BASH PROCESSES COMMAND LINES

SECTION CHEAT SHEET

When Bash receives a command line, it will follow the following 6-step process to interpret its meaning and execute it.

STEP 1: TOKENISATION

During **tokenisation**, bash reads the command line for **unquoted metacharacters**, and uses them to divide the command line into **words** and **operators**.

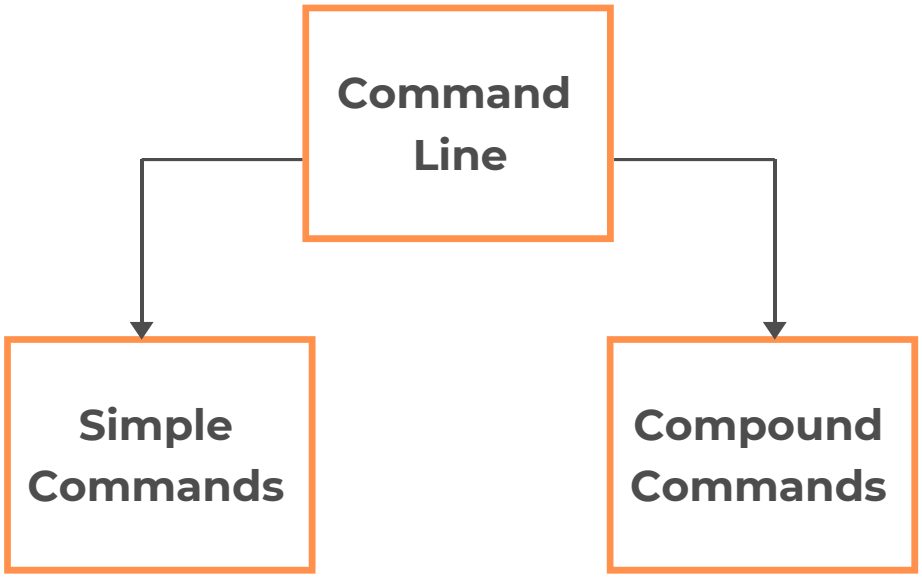
LIST OF METACHARACTERS
Space
Tab
Newline
|
&
;
(
)
<
>

WHAT ARE WORDS?

Words are tokens that **do not** contain any **unquoted metacharacters**

STEP 2: COMMAND IDENTIFICATION

Bash will then break the command line down into simple and compound commands.



SIMPLE COMMANDS

Simple commands are a set of words terminated by a control operator

- The first word is the command name.
- Subsequent words are taken as individual arguments to that command.

WHAT ARE OPERATORS?

Operators are tokens that contain at **least 1 unquoted metacharacter**

LIST OF CONTROL OPERATORS
Newline
|
||
&
&&
;
;;
;&
;;&
|&
(
)

LIST OF REDIRECTION OPERATORS
<
>
<<
>>
<&
>&
>|
<<-
< >

EXAMPLE:

```
echo $name > out
echo | $name > out | -- Identifies unquoted metacharacters
echo | $name > out | -- Identifies Words & operators
```

EXAMPLES OF SIMPLE COMMANDS:

Example 1: echo a b c echo 1 2 3

echo a b c echo 1 2 3 - Tokenisation

echo a b c echo 1 2 3 - Interpreted as one simple command because there are no control operators.

Example 2: echo a b c ; echo 1 2 3

echo a b c ; echo 1 2 3

echo a b c ; echo 1 2 3

echo a b c ; echo 1 2 3

This is interpreted as two simple commands because there is a control operator that ends the first command.

Example 3: echo $name > out

echo $name > out -- Tokenisation

echo $name > out -- Found a redirection operator but no control operators

echo $name > out -- All interpreted as one simple command, including redirection operator

COMPOUND COMMANDS

These are bash's programming constructs. They start with a reserved word and are terminated by the corresponding reserved word

COMPOUND COMMAND EXAMPLE:

Example:

```
if [[ 2 -gt 1 ]]; then
        echo "hello world"
fi
```

Note: We haven't covered how to use compound commands yet -- we will cover them later in detail.

STEP 3: EXPANSIONS

Note 1: Earlier stages are given higher precedence than later ones.

Note 2: Expansions in the same stage are all given equal precedence and are simply processed from left to right.

THERE ARE 4 STAGES TO PROCESSING EXPANSIONS.

STAGE 1	Brace Expansion
STAGE 2	Parameter expansion Arithmetic expansion Command substitution Tilde expansion
STAGE 3	Word Splitting
STAGE 4	Globbing (aka filename expansion)

STAGE 1 - BRACE EXPANSION

Note: Brace expansion is processed as discussed in section 2. Please see the section 2 cheat sheet for more information

STAGE 2

- Parameter expansion
- Arithmetic expansion
- Command substitution
- Tilde expansion

Note: Each of these is processed as discussed in section 2. Please see the section 2 cheat sheet for more information

STAGE 3: WORD SPLITTING

After processing the preceding expansions, the shell will try to split the results of **unquoted parameter expansions, unquoted arithmetic expansions** and **unquoted command substitutions** into individual words.

Note 1: Word splitting is a very important step, because each word provided to a command is considered as an individual argument to the command (see the "Command Identification" step above).

Note 2: Word splitting doesn't occur on the results of expansions that occurred inside double quotes.

Example 1 (Word Splitting)	Example 2 (No Word Splitting)
numbers="1 2 3 4 5" touch $numbers touch 1 2 3 4 5 Result: 5 files created	numbers="1 2 3 4 5" touch "$numbers" touch "1 2 3 4 5" Result: 1 file created called "1 2 3 4 5"

Note 3: Bash will split a word using the characters stored in the IFS variable, which by default contains space, tab, and newline.

Note 4: You can modify the IFS variable just like any other variable.

Note 5: Use **echo "${IFS@Q}"** to view what characters the IFS variable currently contains

For more information, see: GNU Bash Manual - Word Splitting

STAGE 4: GLOBBING (AKA FILENAME EXPANSION)

Upon reaching the globbing stage, bash scans each word for unquoted special pattern characters. These special pattern characters are *, ? and [.

Any word containing one of these characters is interpreted as a pattern, and replaced with a list of alphabetically-sorted filenames that match the pattern (if they exist).

BASIC GLOBBING PATTERN CHARACTER

?	Matches any single character, but requires a character to be there.
*	Matches any string, regardless of length or content. Also matches empty strings
[	Matches any one of the enclosed characters, but requires a character to be there.
[!	Matches any single character *except* those enclosed in the brackets, but requires a character to be there.

Examples of Basic Globbing Patterns:

Consider the following example files: filea.txt, fileb.txt, filec.txt, file1.txt, file2.txt, file3.txt, fileabc.txt, file123.txt	
ls file?.txt	Will match all files with exactly one character between "file" and ".txt". In this case, this pattern will match all files except fileabc.txt and file123.txt
ls file??.txt	Will match all files with exactly two character between "file" and ".txt". In this case, this pattern would match none of the files
ls file???.txt	Will match all files with exactly three character between "file" and ".txt". In this case, this pattern will match fileabc.txt and file123.txt
ls file[abc].txt	Will match all files that include either a single “a”, “b”, or “c” between “file” and “.txt”. In this case, the pattern will match filea.txt, fileb.txt, and filec.txt
ls file[123].txt	Will match all files that include either a single “1”, “2”, or “3” between “file” and “.txt”. In this case, the pattern will match file1.txt, file2.txt, and file3.txt
	Will match all files that include any single character between “file” and “.txt” except an “a”, “b”, or “c” . In this case, the pattern will match file1.txt, file2.txt, and file3.txt
ls file*.txt	Will match all files with any number of characters (even none) between “file” and “.txt”. In this case, the pattern would match all of the example files.

CHARACTER CLASSES

To make it easier to construct ranges of characters within square brackets, bash makes several character classes available for use.

> **Note:** When used in a pattern, character classes must themselves be placed inside square brackets

[:alpha:]❌ [[:alpha:]]✅

[:alpha:] Includes all the letters of the alphabet, in both upper and lowercase	**[:lower:]** Includes just lowercase letters	**[:upper:]** Includes just uppercase letters	**[:digit:]** Includes the numbers 0–9
[:alnum:] Includes the numbers 0–9, and all upper and lowercase letters	**[:punct:]** Includes all forms of punctuation	**[:space:]** Includes all forms of whitespace, such as tab and space characters	**[:word:]** Includes all uppercase and lowercase letters, as well as "_"

Examples of Character Class usage:

Consider the following example files: filea.txt, fileb.txt, filec.txt, file1.txt, file2.txt, file3.txt, fileabc.txt, file123.txt	
ls file[[:lower:]].txt	Will match all files with exactly one lowercase letter character between “file” and “.txt” In this case, this pattern will match filea.txt, fileb.txt and filec.txt
ls file[[:alnum:]].txt	Will match all files with exactly one character between “file” and “.txt” that is either an uppercase letter, a lowercase letter, or a number from 0-9. In this case, this pattern will match filea.txt, fileb.txt, filec.txt, file1.txt, file2.txt and file3.txt
ls file[[:digit:]].txt	Will match all files with exactly one character between “file” and “.txt” that is a number from 0-9. In this case, this pattern will match file1.txt, file2.txt and file3.txt
ls file[![:digit:]].txt	Will match all files with exactly one character between “file” and “.txt” that is not a number from 0-9. In this case, this pattern will match filea.txt, fileb.txt and filec.txt

Extended Globbing Patterns

Important: Use **shopt -s extglob** to enable extended globbing in bash scripts

Extended Pattern General Form:

```
S(pattern1 | pattern2 | ... | patternN)
```

Where **S** is one of the symbols below:

Examples of Extended Globbing Patterns:

Consider the following example files: touch london_july_2001_{001..100}.jpeg, touch london_march_2004_{001..100}.png touch paris_sept_2007_{001..100}.jpg touch newyork_dec_2012_{001..100}.jpeg	
ls *.@(jpeg|png|jpg)	This will list each file that has one of the given file extensions (jpeg, png & jpg) In this case, this pattern will match all the image files
ls @(london|paris)*.@(jpeg|png|jpg)	This will list each file that has one of the given file extensions (jpeg, png & jpg) and that were taken in london and paris. In this case, this pattern will match all the image files with the words “london” or “paris” at the beginning.
ls !(london|paris)*.@(jpeg|png|jpg)	This will list each file that has one of the given file extensions (jpeg, png & jpg) and that do not start with the word london or paris In this case, this pattern will match all the image files with the word newyork at the beginning.
paris+([[:word:]]).@(jpeg|png|jpg)	This will list each file that starts with the word “paris”, then is followed by a series of characters containing only numbers, letters and the “_” character, and finally ends with a file extension of .jpeg, .png or .jpg In this case, this pattern will match all the image files with the word paris at the beginning.

For more information, see: GNU Bash Manual - Pattern Matching

STEP 4: QUOTE REMOVAL

During quote removal, the shell removes all **unquoted backslashes, single quote** characters**,** and **double quote** characters **that did NOT result** from a shell expansion.

echo “hello”	The double quotes are removed because they are not quoted and do not result from a expansion Result: echo hello
echo ‘ “hello” ’	The backslashes are removed, because they are unquoted and do not result from an expansion. The double quotes are retained, however, because they are quoted by the single quotes that surround them. Result: echo “hello”
echo \”hello\”	The backslashes are removed, because they are unquoted and do not result from an expansion. The double quotes are retained, however, because they are quoted by their preceding backslashes. Result: echo “hello”
path=”C:\Users\Karen\Documents” echo $path	On line 2, the backslashes in the path are retained because they result from an expansion (i.e. the parameter expansion of the $path variable). Result: echo C:\Users\Karen\Documents

STEP 5: REDIRECTION

The shell then processes any redirection operators to determine where the standard input, standard output and standard error data streams for the command should connect to

Note 1: Not all commands use every data stream. The best way to find out what streams a command uses is to try it out, or to read its manual page

Note 2: A data stream can only connect to one location at a time

Note 3: Redirections are processed from left to right

Example Redirection Operators

command < file	Redirects the contents of file to the standard input of command.
command > file	Truncates file and then redirects standard output of command to it
command >> file	Appends standard output of command to file.
command 2> file	Truncates file and then redirects standard error of command to it
command 2>> file	Appends standard error of command to file
command &> file	Truncates file, and then redirects both standard output and standard error of command to it.
command &>> file	Appends both standard output and standard error of command to file.

For more information, see: GNU Bash Manual - Redirections

STEP 6: EXECUTE

At this stage the shell has completed its processing of the command line and it now executes the command that have resulted from all the above steps.

And You're Done!!

CONGRATULATIONS

SECTION 4

REQUESTING USER INPUT

SECTION CHEAT SHEET

POSITIONAL PARAMETERS

The shell assigns numbers called positional parameters to each command-line argument that is entered. ($1, $2, $3…)

Example Script:

```
myscript Tom /home/Tom red
```

```
#!/bin/bash
echo “My Name is $1”
echo “My home directory is $2”
echo “My favourite colour is $3”
```

SPECIAL PARAMETERS

Special parameters are like regular parameters, but are created for us by the shell, and are unmodifiable

Special Parameter	Description	Command Line	Parameter Value for Command Line
$#	Stores the number of command line arguments provided to the script	example.sh 1 2 3	3
$0	Stores the script name	test.sh 1 2 3	test.sh
$@	Expands to each positional parameter as its own word with subsequent word splitting	example.sh "daily reports"	daily reports (2 words)
"$@"	Expands to each positional parameter as its own word without subsequent word splitting	example.sh "daily reports"	daily reports (1 word)
$*	Exactly the same as $@	example.sh "daily reports"	daily reports (2 words)
"$*"	Expands to all positional parameter as one word separated by the first letter of the IFS variable without subsequent word splitting	IFS=, example.sh "daily reports"	daily,reports (1 word)
$?	Gives the exit code returned by the most recent command	echo "hello" echo $?	0 (because echo "hello" was successful)

THE READ COMMAND:

The read command asks for input from the user and saves this input into a variable

Syntax for the read command:

```
read variable
```

OPTIONS

-p "prompt"	Displays a prompt to user about what information they must enter
-t time	Timeout if the user does not enter any value within time seconds.
-s	Prevent the input that the user enters from being shown in the terminal. The "secret" option.
-N chars	Limit the users response to exactly chars characters
-n chars	Limit the users response to a maximum of chars characters

Example Script:

```
#!/bin/bash

read -t 5 -p "Input your first name within 5 seconds: " name
read -n 2 -p "Input your age (max 2 digits): " age
read -s -N 5 -p "Enter your zip code (exactly 5 digits): " zipcode
echo "$name, $age, $zipcode" >> data.csv
```

THE SELECT COMMAND

The select command provides the user with a dropdown menu to select from. The user may select an option from a list of options.

It is also possible to provide a prompt to a user using the PS3 shell variable.

Syntax for the select command:

```
PS3="Please select an option below: "
select variable in options; do
        commands...
        break
done
```

Example Script:

```
#!/bin/bash
PS3="What is the day of the week?: "
select day in mon tue wed thu fri sat sun; do
        echo "The day of the week is $day"
        break
done
```

SECTION 5

LOGIC

SECTION CHEAT SHEET

CHAINING COMMANDS WITH LIST OPERATORS

KEY DEFINITIONS

LIST	LIST OPERATORS
When you put one or more commands on a given line	Types of control operators that enable us to create lists of commands that operate in different ways

LIST OPERATORS

Operator	Example	Meaning
&	command1 & command2	Sends command1 into a subshell to run "asynchronously" in the background, and continues to process command2 in the current shell.
;	command1 ; command2	Runs command1 and command2, i.e. one after the other. The shell will wait for command1 to complete before starting command2.
&&	command1 && command2	The "and" operator. The shell will only run command2 if command1 is successful (i.e. returns an exit code of 0).
||	command1 || command2	The "or" operator. The shell will only run command2 if command1 is **un**successful (i.e. returns a **non-zero** exit code).

TEST COMMANDS + CONDITIONAL OPERATORS:

TEST COMMANDS

"a command that can be used in bash to compare different pieces of information"

Syntax:

```
[ EXPRESSION ]
```

Operators to use:

OPERATOR	EXAMPLE	MEANING
-eq	[2 -eq 2]	Successful if the two numbers are equal
-ne	[2 -ne 2]	Successful if the two numbers are not equal
=	[$a = $b]	Successful if the two strings are equal
!=	[$a != $b]	Successful if the two strings are not equal
-z	[-z $c]	Successful if a string is empty
-n	[-n $c]	Successful if a string is not empty
-e	[-e /path/to/file]	Successful if a file system entry /path/to/file exists
-f	[-f /path/to/file]	Successful if a file system entry /path/to/file exists **and is a regular file**
-d	[-d /path/to/file]	Successful if a file system entry /path/to/file exists **and is a directory**
-x	[-x /path/to/file]	Successful if a file system entry /path/to/file exists **and is executable by the current user**

IF STATEMENTS:

start and end using the reserved words "if" and "fi"

check the exit status of a command and only runs the command if a certain condition is true

Syntax for if statements:

```
if test1; then
      Commands... # only run if test1 passes
elif test2; then
      Commands... # only run if test1 fails and test2 passes
elif testN; then
      Commands... # only run if all previous tests fail, but testN passes
else
      Commands... # only run if all tests fail
fi
```

Example Script:

```
#!/bin/bash

read -p "Please enter a number" number

if [ $number -gt 0 ]; then
   echo "Your number is greater than 0"
elif [ $number -lt 0 ]; then
   echo "Your number is less than 0"
else
   echo "Your number is 0!"
fi
```

IF STATEMENTS - COMBINING CONDITIONS:

It is possible to chain together multiple test commands using list operators to create more powerful conditions.

Script: If file1.txt equals file2.txt AND file3.txt, then delete file2.txt and file3.txt

```
#!/bin/bash

a=$(cat file1.txt) #  "a" equals contents of file1.txt
b=$(cat file2.txt) # "b" equals contents of file2.txt
c=$(cat file3.txt) #  "c" equals contents of file3.txt

if [ "$a" = "$b" ] && [ "$a" = "$c" ]; then
  rm file2.txt file3.txt
else
  echo "File1.txt did not match both files"
fi
```

Script: If file1.txt equals file2.txt OR file3.txt, then delete file2.txt and file3.txt

```
#!/bin/bash

a=$(cat file1.txt) # "a" equals contents of file1.txt
b=$(cat file2.txt) # "b" equals contents of file2.txt
c=$(cat file3.txt) # "c" equals contents of file3.txt

if [ "$a" = "$b" ] || [ "$a" = "$c" ]; then
  rm file2.txt file3.txt
else
  echo "File1.txt did not match either file"
fi
```

CASE STATEMENTS:

Case statements provide us with an elegant way to implement branching logic, and are often more convenient than creating multiple “elif” statements.

The tradeoff, however, is that case statements can only work with 1 variable.

Case statements start and end using the reserved words “case” and “esac”

Syntax for case statements:

```
case "$variable" in # don't forget the $ and the double quotes!
  pattern1)
    Commands ...
    ;;
  pattern2)
    Commands ...
    ;;
  patternN)
    Commands ...
    ;;
  *)
    Commands ... # run these if no other pattern matches
    ;;
esac
```

Example Script:

```
#!/bin/bash

read -p "Please enter a number: " number

case "$number" in
    "") echo "You didn't enter anything!"
    [0-9]) echo "you have entered a single digit number";;
    [0-9][0-9]) echo "you have entered a two digit number";;
    [0-9][0-9][0-9]) echo "you have entered a three digit number";;
    *) echo "you have entered a number that is more than three digits";;
esac
```

KEY POINTS ON CASE STATEMENTS:

1. It's very important to **remember to use a $ in front of the variable name** otherwise the case statement won't work, as it cannot access the variable's value

2. **Remember to wrap the expansion of the variable name in double quotes** to avoid word splitting issues

3. Patterns follow the **same rules as globbing patterns.**

4. **Patterns are evaluated from top to bottom**, and only the commands associated with the first pattern that matches will be run.

5. ***) is used as a "default" case,** and is used to hold commands that should run if no other cases match.

SECTION 6

PROCESSING OPTIONS & READING FILES

SECTION CHEAT SHEET

WHILE LOOPS:

while loops run a set of commands ***while*** a certain condition is true, hence their name.

while loops will continue to run until either:

1. The condition command that they're provided with becomes false (i.e. returns a non-zero exit code)
2. The loop is interrupted.

Syntax for the while loop:

```
while condition; do
  commands...
done
```

Example Script:

```
#!/bin/bash

read -p "Enter your number: " num

while [ "$num" -gt 10 ]; do
  echo "$num"
  num=$(( "$num" - 1 ))
done
```

Remember!

You must ensure that the condition you provide the while loop does become false at some point to avoid infinite loops!

HANDLING COMMAND LINE OPTIONS:

The getopts command enables bash to **get** the the **opt**ion**s** provided to the script on the command line.

However, getopts does not get all the options at once; it only gets the very next option on the command line each time it is run.

Therefore, the getopts command is often used as part of a while loop, to ensure that all command line options are processed.

Syntax for the getopts command

```
getopts "optstring" variable
```

You can call variable whatever you like. However, it is conventionally called “opt” because it stores the most recent **opt**ion that getopts has found.

Syntax for optstrings

Any single letter we place in the optstring is considered as its own option.

getopts can only process one-letter options (long-form options such as --all are not supported.)

For Example:

if we wanted to accepted the options “-A” and “-b”, we could write:

```
getopts "Ab" variable
```

Notice how **options are case-sensitive.**

Option Arguments

Sometimes, options can accept arguments of their own. For example, let's say we had the command:

```
ourscript -A 10
```

In order to allow the -A option to accept an argument, such as "10", we would need to place a colon (:) after the letter "A" in the optstring, like so:

```
getopts "A:b" variable
```

Whatever argument that is provided with an option is stored in the $OPTARG shell variable.

So, if we ran the command ourscript -A 10, the $OPTARG variable would store the value of 10 when the getopts command processed the -A option

Practical examples

The getopts command is often used in conjunction with a while loop so that we ensure that each option on the command line gets processed.

In order to allow the script to perform different actions based on the options that are provided, we often also put a case statement inside the while loop, with one case for each option.

Syntax for using the getopts command with a while loop and case statement:

```
while getopts "A:b" variable; do
        case "$variable" in
                A)
                  commands
                  ;;
                b)
                  commands
                  ;;
                \?)
                  commands
                  ;;
        esac
done
```

When an unexpected option is provided to the getopts command, it stores a literal question mark inside the variable.

Therefore, it is good practice to create a \? case to respond to any invalid options. The backslash (\) ensures that the ? is interpreted literally, and not as a special globbing pattern character.

Example Script:

```
#!/bin/bash

while getopts "c:f:" opt; do
    case "$opt" in
      c)  # convert from celsius to farenheit
          result=$(echo "scale=2; ($OPTARG * (9 / 5)) + 32" | bc)
          ;;
      f)  # convert from fahrenheit to celsius
          result=$(echo "scale=2; ($OPTARG - 32) * (5/9)" | bc)
          ;;
      \?)
          Echo "Invalid option provided"
          ;;
    esac
    echo "$result"
done
```

READ-WHILE LOOPS:

Read-while loops are simply while loops that use the read command as their test command

They are used to read lines of output one by one, and do something for each line.

A read-while loop can be used to iterate over the contents of files, or over the output of a command (or pipeline)

ITERATING OVER THE CONTENTS OF FILES

```
while read line; do
  commands...
done < file
```

Example Script: Iterating over a file line by line, and printing each line out

```
#!/bin/bash

while read line; do
  echo "$line"
done < file1.txt
```

ITERATING OVER THE OUTPUT FROM COMMANDS

This is achieved using a technique known as **process substitution**. Process substitution simply allows us to treat the output of a command (or commands) as a file.

Syntax for process substitution

```
<(command) # You can run one command...
<(command1 | command2 | ... | commandN) #... Or an entire pipeline
```

We can then simple read the output of the command into the while loop as if it was a file:

```
while read line; do
  commands...
done < <(command)
```

Example Script: Iterating over each line of output from a command

```
#!/bin/bash

while read line; do
  echo "$line"
done < <(ls $HOME)
```

SECTION 7

ARRAYS + FOR LOOPS

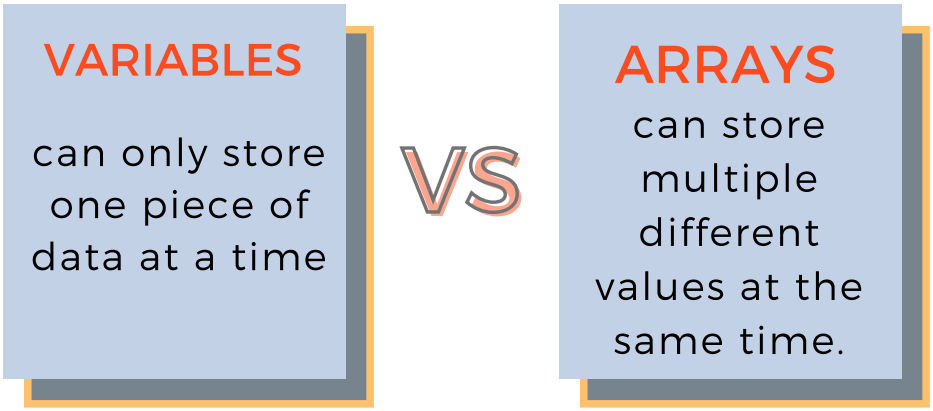
SECTION CHEAT SHEET

WORKING WITH INDEXED ARRAYS:

Variables can only store one piece of data at a time.

Arrays, however, can store multiple different values at the same time.

This makes arrays great tools for storing a lot of data in one place for batch processing.



Syntax for creating a literal indexed array:

```
array=(element1 element2 element3 ... elementN)
```

Example:

```
numbers=(1 2 3 4)
```

Note: Bash uses spaces to separate values within an array, not commas

Each entry of an array is called an element and each element has its own index. Indexes start from 0 and count up.

EXPANDING AN ARRAY

Expansions help us to draw data from an array without modifying the underlying data. Some of these will be familiar to you from our discussion of "parameter expansion tricks" from section 2

Assume we have the array: array=(1 2 3 4 5)

Expansion	Description	Result
${array}	Gives the first element of an array	1
${array[@]}	Expands to all the elements of an array	1 2 3 4 5
${!array[@]}	Expands to all the index numbers of all the elements of an array	0 1 2 3 4
${array[@]:offset}** **E.g.** **${array[@]:2}	Starts at the index specified by offset rather than at index 0, and then continue until the end of array[@]. So, in this example, we would start at index 2, which is the number 3.	3 4 5
${array[@]: -2}	We can also provide negative offsets. In this example, we will start two elements from the end, which is the number 4. **Note:** You must have a space after the ":" and before the "-".	4 5
${array[@]:offset:length}** **E.g.** **${array[@]:2:2}	Skips the first offset elements, and continues until the whole length of the array is length So, in this example, we would skip the first 2 elements, and continue until we had a total of 2 elements.	3 4

MODIFYING AN ARRAY

Once again, assume we have the array: array=(1 2 3 4 5)

Operation	Description	Result
array+=(6)	Appends 6 to the end of array	array becomes (1 2 3 4 5 6)
array+=(a b c)	Appends (a b c) to the end of array	array becomes (1 2 3 4 5 a b c)
unset array[2]	Removes the specified element from the array. In this example the element at index 2 will be removed Note: Index numbers do not update automatically, so this will create a gap in the indexes!	array becomes (1 2 4 5 6)
array[0]=100	Changes the value of a specific element of the array. In this example the element with index 0 will become 100	array becomes (100 2 3 4 5 6)

THE READARRAY COMMAND:

The readarray command converts what it reads on its standard input stream into an array.

CREATING AN ARRAY FROM A FILE:

```
readarray -t arrayname < file
```

Example:

```
readarray -t days < days.txt
```

CREATING AN ARRAY FROM THE OUTPUT OF A COMMAND:

This is achieved using a technique known as “process substitution”. Process substitution simply allows us to treat the output of a command (or commands) as a file.

Syntax for process substitution

```
<(command) # You can run one command...
<(command1 | command2 | ... | commandN) #... Or an entire pipeline
```

We can then simple read the output of the command into the readarray command as if it was a file:

```
readarray -t arrayname < <(command)
```

Example:

```
readarray -t files < <(ls ~/Documents)
```

A NOTE ABOUT NEWLINE CHARACTERS

By default, when the readarray command reads a file (or process substitution), it will save each entire line as a new array element, including the invisible newline character at the end of the line.

Storing new lines characters can cause issues when it comes to formatting, so it is best to remove them.

Therefore, it is suggested that you always use the readarray command's -t option unless you have a strong reason otherwise. This will prevent any newline characters from being stored in your array values, and help prevent issues down the road.

ITERATING OVER ARRAYS WITH FOR LOOPS:

A **for** loop iterates over a list of words or elements and performs a set of commands ***for*** each element within that list, hence its name.

for loops are an amazingly powerful tool when coupled with arrays, as they allow us to create "assembly lines" that we can use for batch processing.

for loop syntax (without array):

```
for <variable> in value1 value2 value3; do
   commands...
done
```

Example Script:

```
#!/bin/bash

for element in first second third; do
        echo "This is $element"
done
```

forloop syntax (with array):

```
for element in "${array[@]}"; do
   commands...
done
```

Example Script:

```
#!/bin/bash

readarray -t files < file_list.txt

for file in "${files[@]}"; do
    if [ -f "$file" ]; then
       echo "$file already exists"
    else
       touch "$file"
       echo "$file was created"
    fi
done
```

SECTION 8

DEBUGGING

SECTION CHEAT SHEET

DEBUGGING USING SHELLCHECK:

Shellcheck is a great tool that can be used either via its web interface at www.shellcheck.net or by running the shellcheck command in your terminal

Install shellcheck on Ubuntu:

```
sudo apt install shellcheck
```

Command to run shellcheck:

```
shellcheck <script>
```

Shellcheck is able to:

- Check if your script contains syntax errors
- Identify potential issues and make suggestions to improve your script

Shellcheck is not able to:

Detect issues with the environment that the script will be running in

COMMON ERRORS AND HOW TO FIX THEM

Error	What it means	How to fix it
No Such File or Directory	The file or directory that you are trying to access does not exist.	1) Check that you didn't mistype the file path 2) Check that Word splitting has not interfered with how a file path is being interpreted (did you wrap everything in double quotes?) 3) Check that the files/folders you want to work with actually exist.
File Exists	You are trying to create a file/directory that already exists	Modify the script's logic to check whether the file you want to create already exists before you run the command to create it. e.g: [-e <my file/directory>] || create it
Permission Denied	You do not have the required permissions to do what you are trying to do	Put the word sudo before the script name when you run it
Operation Not Permitted	You are trying to do something on the system that regular users are not allowed to do	Put the word sudo before the script name when you run it
Command not found	You are trying to run a program that the shell could not find on its path	1) Check for typo in the command name 2) Ensure that the program you are trying to run is on your system's PATH 3) Ensure that the program you are trying to run has execution permissions 4) Install any required packages

HOW TO FIND HELP:

From a documentation perspective, there are two types of commands:

1

INTERNAL COMMANDS.

These are commands that are built into the bash shell. You can see a full list of internal commands by running the help command.

2

EXTERNAL COMMANDS

These are commands external to the bash shell. You can find help on these using the man and info commands.

How to Check if a command is Internal or External:

```
type -a <command>
```

- If the type command says that the command is a “built-in”, then it is an internal command.
- However, if the type command gives you a path to the executable (e.g. “ls is /bin/ls”), then it is an external command.

THE HELP COMMAND

Use the help command to get information on internal commands.

Structure

- Synopsis
- Description
- Options and details of what each option does

Options

-d	Prints only the description of the command
-s	Prints only the usage information of the command

THE MAN COMMAND

The man command provides access to the systems "manual" and is a great first point of call for documentation on external commands.

Man Page Structure

Due to the creative discretion of the developers that wrote them, the structure of man pages can vary from one page to another.

However, they do tend to follow a similar format:

Name Section	The name of the command and a very short description of what it does.
Synopsis Section	How to type out the command. See here for a detailed explanation of how to read the synopsis within the man pages.
Description Section	Detailed information about how the command works and what it can do
General Info	Exit statuses, author contact details, and how to report bugs

Hint: For more information on man pages, try the command "man man"

Searching the man pages

man -k "<keyword/phrase>" allows you to search the "name section" of all man pages for a description in the man pages for commands that contain the keyword provided.

Here are the top 5 lines I get from running man -k ls

```
_llseek (2)          - reposition read/write file offset
add-shell (8)        - add shells to the list of valid login shells
afs_syscall (2)      - unimplemented system calls
ansi_send (3tcl)     - Output of ANSI control sequences to terminals
assert (3)           - abort the program if assertion is false
```

As you can see, the results aren't very relevant.

Trick: However, we can use a bit of regular expression magic to make the matches more accurate.

For example:

Here are the top 5 results when I run man -k "\bls\b". Wrapping "ls" within "\b" here helps ensure that the matching ls must be its own word before being included in the results.

```
dircolors (1)          - color setup for ls
git-ls-files (1)       - Show information about files in the index and the working tree
git-ls-remote (1)      - List references in a remote repository
git-ls-tree (1)        - List the contents of a tree object
git-mktree (1)         - Build a tree-object from ls-tree formatted text
```

As we can see, these are much more relevant results.

MORE IN DEPTH SEARCH

man -K “<keyword/phrase>” performs a more extensive search, as it will search for the keyword/phrase within all sections of all man pages, not just the name section at the top.

Do note however that this is a slower process, so man -K gives you an alternative way of interaction with man pages.

Viewing a Page

In the above lists, you will notice that each result is accompanied by a number in parentheses.

For example:

add-shell (**8**) - add shells to the list of valid login shells
afs_syscall (**2**) - unimplemented system calls
dircolors (**1**) - color setup for ls

These numbers refer to the section of the manual that these commands are found in.

Therefore, to see the add-shell command, which is in section 8, the best way to open its man page is as follows:

```
man 8 add-shell
```

This approach even works for complicated results like this one:

```
ansi_send (3tcl)      - Output of ANSI control sequences to terminals
```

You could open this result like so:

```
man 3tcl ansi_send
```

Note: To exit a man page you have to press q

MANUAL SECTIONS

Below is a list of the manual's sections, which you can use to get an idea of what type of command you are looking at:

1. Executable programs or shell commands
2. System calls (functions provided by the kernel)
3. Library calls (functions within program libraries)
4. Special files (usually found in /dev)
5. File formats and conventions eg /etc/passwd
6. Games
7. Miscellaneous (including macro packages and conventions), e.g. man(7), groff(7)
8. System administration commands (usually only for root)
9. Kernel routines [Non standard]

THE INFO COMMAND

The info command also provides information on external commands, and it typically provides more detailed information than the man command does.

Info pages also contain hyperlink-style references, which makes it easy to hop from one info page to another.

Note: Not every command has an entry for the info command, so the man command is often the most sure point of call.

Check out what the info command can do by running "info" in your terminal.

Hint: Press] to see the next page. Press [to see the previous page. Press "enter" on a link to follow it.

SECTION 10

SCHEDULING AND AUTOMATION

SECTION CHEAT SHEET

THE “AT” COMMAND:

The at command is the most basic way to schedule tasks in Linux. It allows you to run a script at a *one-off point in time*.
You **cannot set up repeated scheduling** with the at command.

Starting the at service:

Command to install the at command on Ubuntu:

```
sudo apt install at
```

To check if the at service running:

```
service atd status
```

To start the at service

```
service atd start
```

To stop the at service

```
service atd stop
```

To restart the at service

```
service atd restart
```

SCHEDULING JOBS WITH “AT”

Syntax to schedule one or more commands:

```
at <time> <date>
```

This will cause the at command to open a prompt where you can add as many commands as you like to be run as part of this job.

You end the prompt with ctrl + d once you have added all your commands.

Syntax to schedule a script:

```
at <time> <date> -f /path/to/script
```

Some Useful Options:

-l	List all jobs that we have scheduled
-r <id>	Remove the job with id “id” from the schedule

Expressing dates and times

Some Sample Time Formats:

HH:MM	Run at HH:MM in 24-hour format
10pm	Run at the upcoming 10 pm timeslot (either today if it's currently before 10pm, or tomorrow if not.)

Some Sample Date Formats:

7/12/2021	Date in month/day/year. This represents 12th July 2021
7.12.2021	Date in day/month/year format.
next week	Exactly 7 days from now
+ 3 days	3 days from now

Hint: Please view the at command's man page by running "man at" for more ways to express timepoints.

Limitations of the "at" command:

- The at command will only execute jobs at the scheduled time if your PC is turned on and the at daemon is running at that time.
- There is no way to set up recurring jobs. Jobs scheduled by the at command only run once.

THE “CRON” COMMAND:

Like the at command, cron requires a daemon service, crond, to be running on your system in order to work.

To check if cron is running:

```
service crond status
```

To start cron:

```
service crond start
```

To stop cron:

```
service crond stop
```

To restart cron:

```
service crond restart
```

EDITING CRONTABS

Crontabs are files that express, in table format, the jobs that we want to schedule and at what time.

Each user has a crontab that contains the jobs that they want to run from their user account.

To edit the current user's crontab:

```
crontab -e
```

Important: Always use `crontab -e` to edit your user's crontab. This will ensure that the cron service is restarted and that your changes take effect.

Crontab expression syntax

Each row in a crontab expresses a new job, and each row is made up of 6 columns separated by white space.

Each element of the expression must be separated, but the amount of space between each column doesnt matter and does not have to be consistent

Crontab Column Title						
	minutes	hours	days of the month	month	days of the week	command
Valid Values	0-59	0-23	1-31	1-12	0-6 or MON-SUN	Your command or PATH to your script

Important: You must ensure that the scripts referred to in the “command” section have been given execution permissions!

Some handy shortcuts

Character	Example	Meaning
*	n/a	Putting * in the hour column is the same as entering the numbers 0-23 Putting * in the minute column is the same as entering the numbers 0-59 Putting a * in the days of the week is the same as entering the numbers 0-6
,	1,5,8	Enter the values 1, 5, and 8 into the current column
-	1-8 MON-WED	Enter the values 1,2,3,4,5,6,7,8 into the current column. Enters the values MON,TUE,WED into the current column.

Hint: You can check whether your crontab expressions match your intentions by using a great tool called crontab.guru

Hint: Run the command man crontab for more information on crontabs

CRON DIRECTORIES:

Cron directories are folders on your system where you can place a group of scripts that you want to run at the same frequency.

Preconfigured directories:

/etc/cron.hourly	Scripts in this folder will run once per hour
/etc/cron.daily	Scripts in this folder will run once per day
/etc/cron.weekly	Scripts in this folder will run once per week
/etc/cron.monthly	Scripts in this folder will run once per month

Important Note 1: Any script placed within these folders will be run by the root user

Important Note 2: Scripts placed within these folders must not contain "." characters in the filenames (e.g. .sh)

Important Note 3: Like all cron scripts, you must ensure that scripts you place in these folders have execution permissions! Otherwise, they won't run!

Creating your own cron directories

Any folder can become a cron directory. All you need to do is to add a new row in your user's crontab with the time you want the directory's contents to run, and then enter the following in the command column:

```
run-parts --report /path/to/directory
```

Limitations of Cron:

Cron will only execute jobs if your PC is turned on and the cron daemon is running at the scheduled time.

ANACRON:

The advantage of anacron is that it has the ability to monitor if a cron job has been run or not, and if not, then run it when this lapse is discovered.

By default, there is only one anacrontab on a system by default. Anacron's crontab is located at /etc/anacrontab.

As /etc/anacrontab requires is a system-wide configuration file, root privileges are required to modify it.

Therefore, to modify the anacrontab, you must run:

```
sudo nano /etc/anacrontab
```

Anacrontab syntax

Each row in an anacrontab is made up of 4 columns as follow:

Anacrontab Column Title				
	period	delay	job-identifier	command
Meaning	How many days between each time your command is run. E.g putting a "1" here would make the command run every day	The delay in minutes from when anacron starts to when this command is run	The name given to the command in the anacron logs. The job identifier can contain any characters except blanks and slashes.	The command you want to run, or path to your script.

Hint: Please run the command man anacrontab for more information on anacrontabs

Limitations of Anacron

- Unlike cron, you cannot run anacron jobs any more frequently than daily.
- Anacron will only recover 1 missed job!

SECTION 11

WORKING WITH REMOTE SERVERS

SECTION CHEAT SHEET

SSH:

SSH, or the **“secure shell”** protocol, is used to provide a secure connection to a shell on a remote system.

Connecting via SSH will enter you into a command-line shell on the remote system.

Syntax for connecting via ssh:

```
ssh user@ip
```

Note: If you are having difficulty connecting, you may have issues with your workplace’s firewall or security policies.

SCP:

SCP, or the **“secure copy”** protocol, is used to transfer files between machines over a secure SSH network connection.

In this section, we used SCP to transfer files between a remote server and our local system in both directions.

Syntax for scp:

```
scp source target
```

SOURCE: The source is where you want to copy ***from***.

TARGET: The target is where you want to copy ***to***.

Syntax to transfer from remote system to local system

```
cp user@ip:/path/to/file /path/to/destination
```

Syntax to transfer file from local system to remote system

```
scp /path/to/file user@ip:/path/to/destination
```