

# Shell Scripting

## What you will learn



Outline what a script is



List use cases for scripting



Describe the 'shebang' interpreter directive



Create and run a simple 'hello\_world' shell script

## What is a script?

- List of commands that can be interpreted
- Commands can be entered interactively → usually not compiled
- Scripting languages are interpreted at runtime
- Slower to run but faster to develop and easier

## What is a script used for?

- Widely used to automate processes
- Automates:
  - ETL jobs
  - File backups and archiving
  - System administration tasks general
- Used for nearly any computational task

including, application integration, plugin, and web app. development

## Shell scripts and the 'shebang'

- Shell script – executable text file with an *interpreter directive*
  - a.k.a 'shebang' directive:
- 'interpreter' – path to an executable program
- 'optional-arg' – single argument string

## Example – 'shebang' directives

Shell script directives: ↗ are scripts that invoke a shell program

#!/bin/sh → invokes the Bourne shell or other compatible shell program  
#!/bin/bash → invokes bash shell

Python script directive:

#!/usr/bin/env python3

## 'Hello World' example shell script

Create the shell script:

```
$ touch hello_world.sh → create empty text file
$ echo '#! /bin/bash' >> hello_world.sh → turn to a bash script
$ echo 'echo hello world' >> hello_world.sh → that the file is a shell script
$ chmod +x hello_world.sh → to print and redirect that output to bash script
```

## 'Hello World' example shell script

Make it executable:

```
1. Owner (U) } three user-based permission groups
2. the group } check current permission
3. all users } → showed that it can read and write, but can't execute because it lacks an 'X'
$ ls -l hello_world.sh → make it executable for all users an 'X'
$ chmod +x hello_world.sh → for all group and users
```

## 'Hello World' example shell script

Run your bash script:

```
$ ./hello_world.sh → to run
hello world
```

# A Brief Introduction to Shell Variables

- Shell variables offer a powerful way to store and later access or modify information

```
$ firstname=Jeff → assign string value 'Jeff' to variable 'firstname'  
$ echo $firstname → display with 'echo $'  
Jeff
```

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

Here's another way to create a shell variable, using the `read` command.

After entering

```
1 $ read lastname
```

on the command line, the shell waits for you to enter some text:

```
1 $ read lastname  
2 Grossman ← it's stored to 'lastname'  
3 $
```

Now we can see that the value `Grossman` has just been stored in the variable `lastname` by the `read` command:

```
1 $ read lastname  
2 Grossman  
3 $ echo $lastname  
4 Grossman
```

By the way, notice that you can `echo` the values of multiple variables at once.

```
1 $ echo $firstname $lastname  
2 Jeff Grossman
```

As you will soon see, the `read` command is particularly useful in shell scripting. You can use it within a shell script to prompt users to input information, which is then stored in a shell variable and available for use by the shell script while it is running. You will also learn about **command line arguments**, which are values that can be passed to a script and automatically assigned to shell variables.

## LAB

```
# This script accepts the user's name and prints  
# a message greeting the user  
  
# Print the prompt message on screen  
echo -n "Enter your name :"  
# Wait for user to enter a name, and save the entered name into the variable 'name'  
read name  
  
# Print the welcome message followed by the name  
echo "Welcome $name"  
# The following message should print on a single line. Hence the usage of '\n'  
echo -n "Congratulations! You just created and ran your first shell script "  
echo "using Bash on IBM Skills Network"
```

bash greet.sh to execute

## 2.1. Find the path to the interpreter

The `which` command helps you find out the path of the command `bash`.

```
1 which bash
```

In this case, it returns the path `/bin/bash`.

## 2.3. Check the permissions of the script

One more step needs to be completed to make `greet.sh` completely executable by name.

To add the execute permission for the user on `greet.sh`, enter the following:

```
1 chmod +x greet.sh
```

Verify whether the execute permission is granted.

*Tip: Generally it's not a good idea to grant permissions to a script for all users, groups, and others. It's more appropriate to limit the execute permission to only the owner, or the user who created the file (you).*

To change permissions for `greet.sh` to make the file executable for the user, run the command below:

```
1 chmod u+x greet.sh
```

Verify the permissions using the command below:

```
1 ls -l greet.sh
```

If you wish to grant execute permission to everyone, you need to run the command `chmod +x greet.sh`.

## 2.2. Edit the script `greet.sh` and add the shebang line to the script

Open the file and add the following line at the beginning of the script:

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

The script should now look like the following:

```
greetsh #! /bin/bash add to first line  
1 # This script accepts the user's name and prints  
2 # a message greeting the user  
3  
4 # Print the prompt message on screen  
5 echo -n "Enter your name :"  
6  
7 # Wait for user to enter a name, and save the entered name into the variable 'name'  
8 read name  
9  
10 # Print the welcome message followed by the name  
11 echo "Welcome $name"  
12
```

# Filters, Pipes and Variables

## What you will learn



Describe and use pipes and filters



Explain and set shell and environment variables

### Pipes and filters

Filters are shell commands, which:

- Take input from standard input
- Send output to standard output
- Transform input data into output data
- Examples are wc, cat, more, head, sort, grep
- Filters can be chained together

### Pipes and filters

| – pipe command

- For chaining filter commands

command1 | command2

- Output of command 1 is input of command 2
- “Pipe” stands for “pipeline”

chain to  
output ↓ input  
\$ ls | sort -r  
Videos ↘ outputs  
Public have already  
Pictures sorted  
Music in reverse  
Downloads  
Documents  
Desktop

### Shell variables

- Scope limited to shell  
shell can't see each other's shell variables

- set – list all shell variables

We can invoke the set command to list all variables that are visible to the current shell

pipe the output to  
\$ set | head -4 → to show first 4  
BASH=/usr/bin/bash variable  
BASHOPTS=checkwinsize:cm  
dhist:complete\_fullquot  
e:expand\_aliases:extglo  
b:extquote:force\_fignor  
e:globasciiranges:hista  
pend:interactive\_comme  
nts:progcomp:promptvars  
:sourcepath  
BASH\_ALIASES=()  
BASH\_ARGC=( [0]=""0"")

### Defining shell variables

var\_name=value

No spaces around '='

to clear variable

unset var\_name

• deletes var\_name deleted

\$ GREETINGS="Hello"  
\$ echo \$GREETINGS  
Hello

\$ AUDIENCE="World"  
\$ echo \$GREETINGS \$AUDIENCE  
Hello World

\$ unset AUDIENCE

### Environment variables

like shell variables, except they have extended scope

- Extended scope

export var\_name

- env – list all environment variables

makes GREETINGS an environment variable

↳ \$ export GREETINGS

\$ env | grep "GREE"  
\$ GREETINGS>Hello

← check GREETINGS has exported as environment variable

Command 1



stdin →

stdout →



Pipe

Command 2



stdin →

stdout → ...

- Pipes are commands in Linux, to use the output of one command as input of another
- There is no limit to the number of times to chain pipes in a row

## Ex.

```
1 $ sort pets.txt | uniq  
2 cat  
3 dog  
4 goldfish  
5 parrot
```

Since `sort` sorts all identical items consecutively, and `uniq` removes all consecutive duplicates, combining the commands prints only the unique lines from `pets.txt`!

## Applying a command to strings and files

Some commands such as `tr` only accept *standard input* - normally text entered from your keyboard - but not strings or filenames.

- `tr` (translate) - replaces characters in input text

```
1 tr [OPTIONS] [target characters] [replacement characters]
```

In cases like this, you can use piping to apply the command to strings and file contents.

With strings, you can use `echo` in combination with `tr` to replace all the vowels in a string with underscores `_`:

```
1 $ echo "Linux and shell scripting are awesome!" | tr "aeiou" "_"  
2 _l_n_x _nd sh_ll scr_pt_ng _r_ _w_s_m!
```

To perform the complement of the operation from the previous example - or to replace all the consonants (any letter that is not a vowel) with an underscore - you can use the `-c` option:

```
1 $ echo "Linux and shell scripting are awesome!" | tr -c "aeiou" "_"  
2 _i_u_a_e_i_i_a_e_a_e_o_e
```

With files, you can use `cat` in combination with `tr` to change all of the text in a file to uppercase as follows:

```
1 $ cat pets.txt | tr "[a-z]" "[A-Z]"  
2 GOLDFISH  
3 DOG  
4 CAT  
5 PARROT  
6 DOG  
7 GOLDFISH  
8 GOLDFISH
```



The possibilities are endless! For example, you could add `uniq` to the above pipeline to only return unique lines in the file, like so:

```
1 $ sort pets.txt | uniq | tr "[a-z]" "[A-Z]"  
2 CAT  
3 DOG  
4 GOLDFISH  
5 PARROT
```

## Extracting information from JSON Files:

Let's see how you can use this json file to get the current price of Bitcoin (BTC) in USD, by using grep command.

```
1  {
2      "coin": {
3          "id": "bitcoin",
4          "icon": "https://static.coinstats.app/coins/Bitcoin6l39t.png",
5          "name": "Bitcoin",
6          "symbol": "BTC",
7          "rank": 1,
8          "price": 57907.78008618953,
9          "priceBtc": 1,
10         "volume": 48430621052.9856,
11         "marketCap": 1093175428640.1146,
12         "availableSupply": 18877868,
13         "totalSupply": 21000000,
14         "priceChange1h": -0.19,
15         "priceChange1d": -0.4,
16         "priceChange1w": -9.36,
17         "websiteUrl": "http://www.bitcoin.org",
18         "twitterUrl": "https://twitter.com/bitcoin",
19         "exp": [
20             "https://blockchair.com/bitcoin/",
21             "https://btc.com/",
22             "https://btc.tokenview.com/"
23         ]
24     }
25 }
```

Copy the above output in a file and name it as `Bitcoinprice.txt`.

The JSON field you want to grab here is `"price": [numbers].[numbers]"`. To get this, you can use the following `grep` command to extract it from the JSON text:

```
1  grep -oE "\"price\"\\s*:\\s*[0-9]*?.[0-9]*"
```

Let's break down the details of this statement:

- `-o` tells `grep` to only return the matching portion
- `-E` tells `grep` to be able to use extended regex symbols such as `?`
- `\"price\"` matches the string `"price"`
- `\\s*` matches any number (including 0) of whitespace (`\s`) characters
- `:` matches `:`
- `[0-9]*` matches any number of digits (from 0 to 9)
- `?\\.` optionally matches a `.`

Use the cat command to get the output of the JSON file and pipe it with the grep command to get the required output.

```
1  cat Bitcoinprice.txt | grep -oE "\"price\"\\s*:\\s*[0-9]*?.[0-9]*"
```

# Useful Features of the Bash Shell

## What you will learn



Outline some key Bash shell-scripting features



Describe the characters and symbols in Bash shell scripting features

### Metacharacters

are special characters that have meaning to the shell

- # – precedes a comment ↳ that the shell ignores
- ; – command separator
- \* – filename expansion wildcard
- ? – single character wildcard in filename expansion

```
$ # Some metacharacters
↳ not return
$ echo "Hello"; whoami
Hello ↲
ravahuja ↲

$ ls /bin/ba
/bin/bash ↲ return all start with 'ba'
in bin folder
$ ls /bin/?ash
/bin/bash /bin/dash
```

### I/O redirection

Input/Output, or **I/O redirection**, refers to a set of features used for redirecting

- > – Redirect output to the file ↳ also create a file if it doesn't exist and overwrite its contents
- >> – Append output to the file ↳ to exist contents
- 2> – Redirect standard error to file
- 2>> – Append standard error to file
- < – Redirect file contents to standard input

### Command substitution

- Replace the command with its output `$(command)` or ``command``
- Store output of `pwd` command in here:

```
$ here=$(pwd)
$ echo $here
/home/jgrom
```

### Quoting

- \ – escape unique character interpretation ↳ except meta-characters
- " " – interpret literally, but evaluate metacharacters
- ' ' – interpret literally ↳ all contents

```
tell dollar sign is text
$ echo "\$1 each"
$1 each ↲
$ echo "$1 each"
each ↲ didn't print space
$ echo '$1 each'
$1 each ↲
```

### I/O redirection examples

```
$ echo "line1" > eg.txt ↲ write
$ cat eg.txt ↲ create
line1 ↲ appended
$ echo "line2" >> eg.txt
$ cat eg.txt
line1
line2
```

```
$ garbage
garbage: command not found
$ garbage 2> err.txt ↲ catches the error
$ cat err.txt ↲ and redirects to the error
garbage: command not found
to the err.txt
```

### Command line arguments

- Program arguments specified on the command line
- A way to pass arguments to a shell script
- Usage:

```
$ ./MyBashScript.sh arg1 arg2
```

### Batch versus concurrent modes

↳ has 2 main modes  
**Batch mode:** → usual mode

- Commands run sequentially

↳ will be executed in a particular order  
command1; command2  
↳ only run after command 1 is completed

**Concurrent mode:**

- Commands run in parallel

↳ operate in the background  
↓ passes control to  
command1 & command2

## Basic Command

# Introduction to Advanced Bash Scripting

```
if [ condition ]
then
    statement_block_1
else
    statement_block_2
fi
```



Ex.

```
if [[ $# == 2 ]]
then
    echo "number of arguments is equal to 2"
else
    echo "number of arguments is not equal to 2"
fi
```

AND

```
if [ condition1 ] && [ condition2 ]
then
    echo "conditions 1 and 2 are both true"
else
    echo "one or both conditions are false"
fi

if [ condition1 ] || [ condition2 ]
then
    echo "conditions 1 or 2 are true"
else
    echo "both conditions are false"
fi
```

OR

## Logical operators

The following logical operators can be used to compare integers within a condition in an `if` condition block.

`==` : is equal to

If a variable `a` has a value of 2, the following condition evaluates to `true`; otherwise it evaluates to `false`.

```
1 $a == 2
```

`!=` : is not equal to

If a variable `a` has a value different from 2, the following statement evaluates to `true`. If its value is 2, then it evaluates to `false`.

```
1 a != 2
```

*Tip:* The `!` logical negation operator changes `true` to `false` and `false` to `true`.

`<=` : is less than or equal to

If a variable `a` has a value of 2, then the following statement evaluates to `true`:

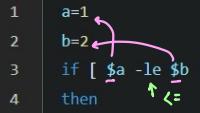
```
1 a <= 3
```

and the following statement evaluates to `false`:

```
1 a <= 1
```

Alternatively, you can use the equivalent notation `-le` in place of `<=`:

```
1 a=1
2 b=2
3 if [ $a -le $b ]
4 then
    echo "a is less than or equal to b"
5 else
    echo "a is not less than or equal to b"
6 fi
```



## Arithmetic calculations

`+`   `-`   `*`   `/`

You can perform integer addition, subtraction, multiplication, and division using the notation `$(( ))`.

For example, the following two sets of commands both display the result of adding 3 and 2.

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

or

```
1 a=3
2 b=2
3 c=$((a+b))
4 echo $c
```

Bash natively handles integer arithmetic but does not handle floating-point arithmetic. As a result, it will always truncate the decimal portion of a calculation result.

For example:

```
1 echo $((3/2))
```

prints the truncated integer result, `1`, not the floating-point number, `1.5`.

The following table summarizes the basic arithmetic operators:

Symbol	Operation
<code>+</code>	addition
<code>-</code>	subtraction
<code>*</code>	multiplication
<code>/</code>	division

Table: Arithmetic operators

## Arrays

The `array` is a Bash built-in data structure. An array is a space-delimited list contained in parentheses. To create an array, declare its name and contents:

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

This statement creates and populates the array `my_array` with the items in the parentheses: `1`, `2`, `"three"`, `"four"`, and `5`.

You can also create an empty array by using:

```
1 declare -a empty_array
```

If you want to add items to your array after creating it, you can add to your array by appending one element at a time:

```
1 my_array=("six")
```

```
2 my_array+=("7")
```

This adds elements `"six"` and `7` to the array `my_array`.

By using indexing, you can access individual or multiple elements of an array:

```
1 # print the first item of the array:  
2 echo ${my_array[0]}  
3  
4 # print the third item of the array:  
5 echo ${my_array[2]}  
6  
7 # print all array elements:  
8 echo ${my_array[@]}
```

**Tip:** Note that array indexing starts from 0, not from 1.

## for loops

You can use a construct called a `for` loop along with indexing to iterate over all elements of an array.

For example, the following `for` loops will continue to run over and over again until every element is printed:

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

or

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

The `for` loop requires a `; do` component in order to cycle through the loop. Additionally, you need to terminate the `for` loop block with a `done` statement.

Another way to implement a `for` loop when you know how many iterations you want is as follows. For example, the following code prints the number 0 through 6.

```
1 N=6  
2 for (( i=0; i<=$N; i++ )); do  
3     echo $i  
4 done
```

You can use `for` loops to accomplish all sorts of things. For example, you could count the number of items in an array or sum up its elements, as the following Bash script does:

```
1 #!/usr/bin/env bash  
2 # initialize array, count, and sum  
3 my_array=(1 2 3)  
4 count=0  
5 sum=0  
6 for i in ${!my_array[@]}; do  
7     # print the ith array element  
8     echo ${my_array[$i]}  
9     # increment the count by one  
10    count=$((count+1))  
11    # add the current value of the array to the sum  
12    sum=$((sum+${my_array[$i]}))  
13 done  
14 echo $count  
15 echo $sum
```

Go ahead and try running this script, so you get a sense of how this loop works.

# \* LAB

## 1.1. Create a new Bash script

Create a Bash script file and make it executable.

▼ Click here for Hint

Use the `echo` command to redirect a shebang to a new Bash script.

Alternatively, open a new text file using your favourite text editor and add a **shebang** to it. Remember to make your new script executable.

▼ Click here for Solution

Here's a solution using only the command line:

```
1 echo #!/bin/bash > conditional_script.sh
2 chmod u+x conditional_script.sh
```

use bash  
create or replace  
file name

## 1.2. Query the user and store their response

Now get your script to:

1. Ask the user a binary "yes or no" question of your choosing
2. Store the user's answer in a variable.

► Click here for Hint

▼ Click here for Solution

Your Bash script should now look something like this:

```
1 #!/bin/bash
2
3 echo 'Are you enjoying this course so far?'
4 echo -n "Enter \"y\" for yes, \"n\" for no"
5 read response
```

double quote

## 1.3. Use a conditional block to select a response for the user

Finally, use a conditional block to print a message to the user based on their response to your query.

**Tip:** It's best practice to also handle the case where the response doesn't match any of the allowable responses.

▼ Click here for Hint

Use a conditional `if elif else fi` block that uses a logical operator to compare the user's response to the available response options and prints an appropriate message in each case.

```
1 #!/bin/bash
2
3 echo 'Are you enjoying this course so far?'
4 echo -n "Enter \"y\" for yes, \"n\" for no"
5 read response
6 if [ $response == y ]
7 then
8   echo "I'm pleased to hear you are enjoying the course."
9   echo "Your feedback regarding what you have been learning is greatly appreciated."
10  elif [ $response == n ]
11  then
12    echo "I'm sorry to hear you are not enjoying the course."
13    echo "Your feedback regarding what we can do to improve the course would be greatly appreciated."
14  else
15    echo "Your response must be either 'y' or 'n'."
16    echo "Please re-run the script to try again."
17  fi
```

## Exercise 2 - Performing basic mathematical calculations and numerical logical comparisons

### 2.1. Create a Bash script

Create an executable Bash script that prompts the user for two integers, then stores and prints both the sum and product of the two integers.

▼ Click here for Hint

Use the `echo` and `read` commands as in the previous exercise.

Recall the notation for arithmetic calculations.

▼ Click here for Solution

```
1 #!/bin/bash
2
3 echo -n "Enter an integer: "
4 read n1
5 echo -n "Enter another integer: "
6 read n2
7
8 sum=$((n1+n2))
9 product=$((n1*n2))
10
11 echo "The sum of $n1 and $n2 is $sum"
12 echo "The product of $n1 and $n2 is $product."
```

### 2.2. Add logic to your script

Add logic to your script that determines whether the sum is greater than, less than, or equal to the product. Display an appropriate statement corresponding to each possible result.

Assume the user inputs two integers. Don't worry about handling the case where the user inputs a non-integer string by mistake.

▼ Click here for Hint

Use a conditional block. Recall the notation for logical operators.

▼ Click here for Solution

```
1 #!/bin/bash
2
3 echo -n "Enter an integer: "
4 read n1
5 echo -n "Enter another integer: "
6 read n2
7
8 sum=$((n1+n2))
9 product=$((n1*n2))
10
11 echo "The sum of $n1 and $n2 is $sum"
12 echo "The product of $n1 and $n2 is $product."
13
14 if [ $sum -lt $product ]
15 then
16   echo "The sum is less than the product."
17 elif [ $sum == $product ]
18 then
19   echo "The sum is equal to the product."
20 elif [ $sum -gt $product ]
21 then
22   echo "The sum is greater than the product."
23 fi
```

## Exercise 3 - Using arrays for storing and accessing data within for loops

### 3.1. Download a CSV file to your current working directory

The file, `arrays_table.csv`, is located at the following url:

```
1 https://cf-courses-data.s3.us.cloud-object-storage.a...
```

► Click here for Hint

▼ Click here for Solution

```
1 csv_file="https://cf-courses-data.s3.us.cloud-object-...
2 wget $csv_file
```

### 3.2. Display the CSV file to understand what it looks like

► Click here for Hint

▼ Click here for Solution

```
1 cat arrays_table.csv
```

### 3.3. Create a Bash script that parses table columns into 3 arrays

► Click here for Hint

▼ Click here for Solution

```
1 #!/bin/bash
2
3 csv_file="./arrays_table.csv"
4
5 # parse table columns into 3 arrays
6 column_0=$(cut -d "," -f 1 $csv_file)
7 column_1=$(cut -d "," -f 2 $csv_file)
8 column_2=$(cut -d "," -f 3 $csv_file)
9
10 # print first array
11 echo "Displaying the first column:"
12 echo "${column_0[@]}"
```

### 3.4. Create a new array as the difference of the third and second columns.

```
14 ## Create a new array as the difference of columns 1 and 2
15 # initialize array with header
16 column_3=(column_3)
17 # get the number of lines in each column
18 nlines=$(cat $csv_file | wc -l)
19 echo "There are $nlines lines in the file"
20 # populate the array
21 for ((i=1; i<$nlines; i++)); do
22   column_3[$i]=$(($column_2[$i] - $column_1[$i]))
23 done
24 echo "${column_3[@]}"
```

### 3.5. Create a report by combining your new column with the source table

```
## Combine the new array with the csv file
# first write the new array to file
# initialize the file with a header
echo "${column_3[0]}" > column_3.txt
for ((i=1; i<$nlines; i++)); do
  echo "${column_3[$i]}" >> column_3.txt
done
paste -d "," $csv_file column_3.txt > report.csv
```

# Scheduling Jobs using Cron

## What you will learn



Schedule Cron jobs with Crontab



Describe the Cron syntax



Apply and remove Cron jobs

## Job scheduling

- Schedule jobs to run automatically at certain times

Example:

Load script at midnight every night

Backup script to run every Sunday at 2 AM

- Cron allows you to automate such tasks

## What are Cron, Crond, and Crontab?

Cron is a service that runs jobs  
Crond interprets 'crontab files'

- Crontab contains jobs and schedule data
- Crontab enables to edit a Crontab file

name of tool that  
runs jobs  
every minute and submits  
the corresponding jobs  
to cron at scheduled

## Scheduling Cron jobs with Crontab

\$ crontab -e  
• Opens editor

Job syntax:

minute day\_of\_month day\_of\_week  
m h dom mon dow command  
hour month

Example job:  
It can't be empty, so put these asterisks (wild card)  
They mean Any  
30 15 \* \* 0 date >> sundays.txt

## Scheduling Cron jobs with Crontab

```
GNU nano 4.8 /tmp/crontab.BWe1DK/crontab
# Edit this file to introduce tasks to be run by cron.

# Each task to run has to be defined through a single line
# indicating with different fields when the task will be run
# and what command to run for the task

# To define the time you can provide concrete values for
# minute (m), hour (h), day of month (dom), month (mon),
# and day of week (dow) or use '*' in these fields (for 'any').
#
# Notice that tasks will be started based on the cron's system
# daemon's notion of time and timezone.
#
# Output of the crontab jobs (including errors) is sent through
# email to the user the crontab file belongs to (unless redirected).
#
# For example, you can run a backup of all your user accounts
# at 5 a.m every week:
# 0 5 * * 1 tar -zcf /var/backups/home.tgz /home/
#
# For more information see the manual pages of crontab(5) and cron(8)

# m h dom mon dow command
```

Get Help F1 Write Out F2 Where Is F3 Cut Text F4 To Spell F5 Cur Pos F6 Undo F7 Mark Text F8 Copy Text F9

## Entering jobs

```
# m h dom mon dow command
30 15 * * 0 date >> path/sundays.txt
0 0 * * * /cron_scripts/load_data.sh
0 2 * * 0 /cron_scripts/backup_data.sh
```

Run at  
15:30  
every  
Sunday

Run at  
midnight  
every day

Run at  
2:00  
every Sunday

## Exit editor and save

```
# m h dom mon dow command
30 15 * * 0 date >> path/sundays.txt
0 0 * * * /cron_scripts/load_data.sh
0 2 * * 0 /cron_scripts/backup_data.sh
First type control+x, then Enter 'y' to save.
Save modified buffer? Y Yes N No
Y Yes then Enter 'y' to save.
N No ^C Cancel
```

## Viewing and removing Cron jobs

```
jgrom@GROOT617:~$ crontab -l | tail -6
#
# m h dom mon dow command
30 15 * * 0 date >> path/sundays.txt
0 0 * * * /cron_scripts/load_data.sh
0 2 * * 0 /cron_scripts/backup_data.sh
jgrom@GROOT617:~$
```

\$ crontab -e # add/remove cron job with editor ← to remove

# LAB

## Exercise 1 - Understand crontab file syntax

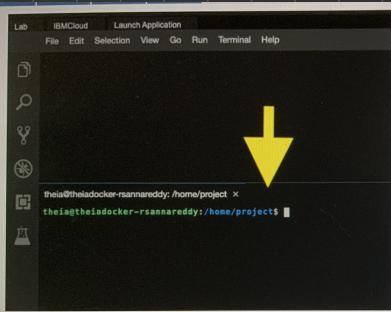
Cron is a system daemon used to execute desired tasks in the background at designated times.

A crontab file is a simple text file containing a list of commands meant to be run at specified times. It is edited using the `crontab` command.

Each line in a crontab file has five time-and-date fields, followed by a command, followed by a newline character (`\n`). The fields are separated by spaces.

The five time-and-date fields cannot contain spaces and their allowed values are as follows:

Field	Allowed values
minute	0-59
hour	0-23, 0 = midnight
day	1-31
month	1-12
weekday	0-6, 0 = Sunday



Run the commands below on the newly opened terminal.

The `-l` option of the `crontab` command prints the current crontab.

1 crontab -l  
You may get a message `no crontab for theia` if your crontab is empty.

## Exercise 3 - Add a job in the crontab file

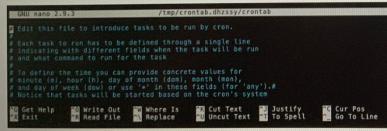
### 3.1. Add a job to crontab

To add a cron job, run the command below:

1 crontab -e

This will create a new crontab file for you (if you don't have one already). Now you are ready to add a new cron job.

Your crontab file will be opened in an editor as shown in the image below:



Scroll down to the end of the file using the arrow keys:

1 0 21 \* \* \* echo "Welcome to cron" >> /tmp/echo.txt  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14  
15  
16  
17  
18  
19  
20  
21  
22  
23  
24  
25  
26  
27  
28  
29  
30  
31  
32  
33  
34  
35  
36  
37  
38  
39  
40  
41  
42  
43  
44  
45  
46  
47  
48  
49  
50  
51  
52  
53  
54  
55  
56  
57  
58  
59  
60  
61  
62  
63  
64  
65  
66  
67  
68  
69  
70  
71  
72  
73  
74  
75  
76  
77  
78  
79  
80  
81  
82  
83  
84  
85  
86  
87  
88  
89  
90  
91  
92  
93  
94  
95  
96  
97  
98  
99  
100  
101  
102  
103  
104  
105  
106  
107  
108  
109  
110  
111  
112  
113  
114  
115  
116  
117  
118  
119  
120  
121  
122  
123  
124  
125  
126  
127  
128  
129  
130  
131  
132  
133  
134  
135  
136  
137  
138  
139  
140  
141  
142  
143  
144  
145  
146  
147  
148  
149  
150  
151  
152  
153  
154  
155  
156  
157  
158  
159  
160  
161  
162  
163  
164  
165  
166  
167  
168  
169  
170  
171  
172  
173  
174  
175  
176  
177  
178  
179  
180  
181  
182  
183  
184  
185  
186  
187  
188  
189  
190  
191  
192  
193  
194  
195  
196  
197  
198  
199  
200  
201  
202  
203  
204  
205  
206  
207  
208  
209  
210  
211  
212  
213  
214  
215  
216  
217  
218  
219  
220  
221  
222  
223  
224  
225  
226  
227  
228  
229  
230  
231  
232  
233  
234  
235  
236  
237  
238  
239  
240  
241  
242  
243  
244  
245  
246  
247  
248  
249  
250  
251  
252  
253  
254  
255  
256  
257  
258  
259  
260  
261  
262  
263  
264  
265  
266  
267  
268  
269  
270  
271  
272  
273  
274  
275  
276  
277  
278  
279  
280  
281  
282  
283  
284  
285  
286  
287  
288  
289  
290  
291  
292  
293  
294  
295  
296  
297  
298  
299  
300  
301  
302  
303  
304  
305  
306  
307  
308  
309  
310  
311  
312  
313  
314  
315  
316  
317  
318  
319  
320  
321  
322  
323  
324  
325  
326  
327  
328  
329  
330  
331  
332  
333  
334  
335  
336  
337  
338  
339  
340  
341  
342  
343  
344  
345  
346  
347  
348  
349  
350  
351  
352  
353  
354  
355  
356  
357  
358  
359  
360  
361  
362  
363  
364  
365  
366  
367  
368  
369  
370  
371  
372  
373  
374  
375  
376  
377  
378  
379  
380  
381  
382  
383  
384  
385  
386  
387  
388  
389  
390  
391  
392  
393  
394  
395  
396  
397  
398  
399  
400  
401  
402  
403  
404  
405  
406  
407  
408  
409  
410  
411  
412  
413  
414  
415  
416  
417  
418  
419  
420  
421  
422  
423  
424  
425  
426  
427  
428  
429  
430  
431  
432  
433  
434  
435  
436  
437  
438  
439  
440  
441  
442  
443  
444  
445  
446  
447  
448  
449  
450  
451  
452  
453  
454  
455  
456  
457  
458  
459  
460  
461  
462  
463  
464  
465  
466  
467  
468  
469  
470  
471  
472  
473  
474  
475  
476  
477  
478  
479  
480  
481  
482  
483  
484  
485  
486  
487  
488  
489  
490  
491  
492  
493  
494  
495  
496  
497  
498  
499  
500  
501  
502  
503  
504  
505  
506  
507  
508  
509  
510  
511  
512  
513  
514  
515  
516  
517  
518  
519  
520  
521  
522  
523  
524  
525  
526  
527  
528  
529  
530  
531  
532  
533  
534  
535  
536  
537  
538  
539  
540  
541  
542  
543  
544  
545  
546  
547  
548  
549  
550  
551  
552  
553  
554  
555  
556  
557  
558  
559  
560  
561  
562  
563  
564  
565  
566  
567  
568  
569  
570  
571  
572  
573  
574  
575  
576  
577  
578  
579  
580  
581  
582  
583  
584  
585  
586  
587  
588  
589  
589  
590  
591  
592  
593  
594  
595  
596  
597  
598  
599  
600  
601  
602  
603  
604  
605  
606  
607  
608  
609  
610  
611  
612  
613  
614  
615  
616  
617  
618  
619  
620  
621  
622  
623  
624  
625  
626  
627  
628  
629  
630  
631  
632  
633  
634  
635  
636  
637  
638  
639  
640  
641  
642  
643  
644  
645  
646  
647  
648  
649  
650  
651  
652  
653  
654  
655  
656  
657  
658  
659  
660  
661  
662  
663  
664  
665  
666  
667  
668  
669  
669  
670  
671  
672  
673  
674  
675  
676  
677  
678  
679  
680  
681  
682  
683  
684  
685  
686  
687  
688  
689  
689  
690  
691  
692  
693  
694  
695  
696  
697  
698  
699  
700  
701  
702  
703  
704  
705  
706  
707  
708  
709  
709  
710  
711  
712  
713  
714  
715  
716  
717  
718  
719  
719  
720  
721  
722  
723  
724  
725  
726  
727  
728  
729  
729  
730  
731  
732  
733  
734  
735  
736  
737  
738  
739  
739  
740  
741  
742  
743  
744  
745  
746  
747  
748  
749  
749  
750  
751  
752  
753  
754  
755  
756  
757  
758  
759  
759  
760  
761  
762  
763  
764  
765  
766  
767  
768  
769  
769  
770  
771  
772  
773  
774  
775  
776  
777  
778  
779  
779  
780  
781  
782  
783  
784  
785  
786  
787  
788  
789  
789  
790  
791  
792  
793  
794  
795  
796  
797  
798  
799  
800  
801  
802  
803  
804  
805  
806  
807  
808  
809  
809  
810  
811  
812  
813  
814  
815  
816  
817  
818  
819  
819  
820  
821  
822  
823  
824  
825  
826  
827  
828  
829  
829  
830  
831  
832  
833  
834  
835  
836  
837  
838  
839  
839  
840  
841  
842  
843  
844  
845  
846  
847  
848  
849  
849  
850  
851  
852  
853  
854  
855  
856  
857  
858  
859  
859  
860  
861  
862  
863  
864  
865  
866  
867  
868  
869  
869  
870  
871  
872  
873  
874  
875  
876  
877  
878  
879  
879  
880  
881  
882  
883  
884  
885  
886  
887  
888  
889  
889  
890  
891  
892  
893  
894  
895  
896  
897  
898  
899  
900  
901  
902  
903  
904  
905  
906  
907  
908  
909  
909  
910  
911  
912  
913  
914  
915  
916  
917  
918  
919  
919  
920  
921  
922  
923  
924  
925  
926  
927  
928  
929  
929  
930  
931  
932  
933  
934  
935  
936  
937  
938  
939  
939  
940  
941  
942  
943  
944  
945  
946  
947  
948  
949  
949  
950  
951  
952  
953  
954  
955  
956  
957  
958  
959  
959  
960  
961  
962  
963  
964  
965  
966  
967  
968  
969  
969  
970  
971  
972  
973  
974  
975  
976  
977  
978  
979  
979  
980  
981  
982  
983  
984  
985  
986  
987  
988  
989  
989  
990  
991  
992  
993  
994  
995  
996  
997  
998  
999  
1000  
1001  
1002  
1003  
1004  
1005  
1006  
1007  
1008  
1009  
1009  
1010  
1011  
1012  
1013  
1014  
1015  
1016  
1017  
1018  
1019  
1019  
1020  
1021  
1022  
1023  
1024  
1025  
1026  
1027  
1028  
1029  
1029  
1030  
1031  
1032  
1033  
1034  
1035  
1036  
1037  
1038  
1039  
1039  
1040  
1041  
1042  
1043  
1044  
1045  
1046  
1047  
1048  
1049  
1049  
1050  
1051  
1052  
1053  
1054  
1055  
1056  
1057  
1058  
1059  
1059  
1060  
1061  
1062  
1063  
1064  
1065  
1066  
1067  
1068  
1069  
1069  
1070  
1071  
1072  
1073  
1074  
1075  
1076  
1077  
1078  
1079  
1079  
1080  
1081  
1082  
1083  
1084  
1085  
1086  
1087  
1088  
1089  
1089  
1090  
1091  
1092  
1093  
1094  
1095  
1096  
1097  
1098  
1098  
1099  
1099  
1100  
1101  
1102  
1103  
1104  
1105  
1106  
1107  
1108  
1109  
1109  
1110  
1111  
1112  
1113  
1114  
1115  
1116  
1117  
1118  
1119  
1119  
1120  
1121  
1122  
1123  
1124  
1125  
1126  
1127  
1128  
1129  
1129  
1130  
1131  
1132  
1133  
1134  
1135  
1136  
1137  
1138  
1139  
1139  
1140  
1141  
1142  
1143  
1144  
1145  
1146  
1147  
1148  
1149  
1149  
1150  
1151  
1152  
1153  
1154  
1155  
1156  
1157  
1158  
1159  
1159  
1160  
1161  
1162  
1163  
1164  
1165  
1166  
1167  
1168  
1169  
1169  
1170  
1171  
1172  
1173  
1174  
1175  
1176  
1177  
1178  
1179  
1179  
1180  
1181  
1182  
1183  
1184  
1185  
1186  
1187  
1188  
1189  
1189  
1190  
1191  
1192  
1193  
1194  
1195  
1196  
1197  
1198  
1198  
1199  
1199  
1200  
1201  
1202  
1203  
1204  
1205  
1206  
1207  
1208  
1209  
1209  
1210  
1211  
1212  
1213  
1214  
1215  
1216  
1217  
1218  
1219  
1219  
1220  
1221  
1222  
1223  
1224  
1225  
1226  
1227  
1228  
1229  
1229  
1230  
1231  
1232  
1233  
1234  
1235  
1236  
1237  
1238  
1239  
1239  
1240  
1241  
1242  
1243  
1244  
1245  
1246  
1247  
1248  
1249  
1249  
1250  
1251  
1252  
1253  
1254  
1255  
1256  
1257  
1258  
1259  
1259  
1260  
1261  
1262  
1263  
1264  
1265  
1266  
1267  
1268  
1269  
1269  
1270  
1271  
1272  
1273  
1274  
1275  
1276  
1277  
1278  
1279  
1279  
1280  
1281  
1282  
1283  
1284  
1285  
1286  
1287  
1288  
1289  
1289  
1290  
1291  
1292  
1293  
1294  
1295  
1296  
1297  
1298  
1298  
1299  
1299  
1300  
1301  
1302  
1303  
1304  
1305  
1306  
1307  
1308  
1309  
1309  
1310  
1311  
1312  
1313  
1314  
1315  
1316  
1317  
1318  
1319  
1319  
1320  
1321  
1322  
1323  
1324  
1325  
1326  
1327  
1328  
1329  
1329  
1330  
1331  
1332  
1333  
1334  
1335  
1336  
1337  
1338  
1339  
1339  
1340  
1341  
1342  
1343  
1344  
1345  
1346  
1347  
1348  
1349  
1349  
1350  
1351  
1352  
1353  
1354  
1355  
1356  
1357  
1358  
1359  
1359  
1360  
1361  
1362  
1363  
1364  
1365  
1366  
1367  
1368  
1369  
1369  
1370  
1371  
1372  
1373  
1374  
1375  
1376  
1377  
1378  
1379  
1379  
1380  
1381  
1382  
1383  
1384  
1385  
1386  
1387  
1388  
1389  
1389  
1390  
1391  
1392  
1393  
1394  
1395  
1396  
1397  
1398  
1398  
1399  
1399  
1400  
1401  
1402  
1403  
1404  
1405  
1406  
1407  
1408  
1409  
1409  
1410  
1411  
1412  
1413  
1414  
1415  
1416  
1417  
1418  
1419  
1419  
1420  
1421  
1422  
1423  
1424  
1425  
1426  
1427  
1428  
1429  
1429  
1430  
1431  
1432  
1433  
1434  
1435  
1436  
1437  
1438  
1439  
1439  
1440  
1441  
1442  
1443  
1444  
1445  
1446  
1447  
1448  
1449  
1449  
1450  
1451  
1452  
1453  
1454  
1455  
1456  
1457  
1458  
1459  
1459  
1460  
1461  
1462  
1463  
1464  
1465  
1466  
1467  
1468  
1469  
1469  
1470  
1471  
1472  
1473  
1474  
1475  
1476  
1477  
1478  
1479  
1479  
1480  
1481  
1482  
1483  
1484  
1485  
1486  
1487  
1488  
1489  
1489  
1490  
1491  
1492  
1493  
1494  
1495  
1496  
1497  
1498  
1498  
1499  
1499  
1500  
1501  
1502  
1503  
1504  
1505  
1506  
1507  
1508  
1509  
1509  
1510  
1511  
1512  
1513  
1514  
1515  
1516  
1517  
1518  
1519  
1519  
1520  
1521  
1522  
1523  
1524  
1525  
1526  
1527  
1528  
1529  
1529  
1530  
1531  
1532  
1533  
1534  
1535  
1536  
1537  
1538  
1539  
1539  
1540  
1541  
1542  
1543  
1544  
1545  
1546  
1547  
1548  
1549  
1549  
1550  
1551  
1552  
1553  
1554  
1555  
1556  
1557  
1558  
1559  
1559  
1560  
1561  
1562  
1563  
1564  
1565  
1566  
1567  
1568  
1569  
1569  
1570  
1571  
1572  
1573  
1574  
1575  
1576  
1577  
1578  
1579  
1579  
1580  
1581  
1582  
1583  
1584  
1585  
1586  
1587  
1588  
1589  
1589  
1590  
1591  
1592  
1593  
1594  
1595  
1596  
1597  
1598  
1598  
1599  
1599  
1600  
1601  
1602  
1603  
1604  
1605  
1606  
1607  
1608  
1609  
1609  
1610  
1611  
1612  
1613  
1614  
1615  
1616  
1617  
1618  
1619  
1619  
1620  
1621  
1622  
1623  
1624  
1625  
1626  
1627  
1628  
1629  
1629  
1630  
1631  
1632  
1633  
1634  
1635  
1636  
1637  
1638  
1639  
1639  
1640  
1641  
1642  
1643  
1644  
1645  
1646  
1647  
1648  
1649  
1649  
1650  
1651  
1652  
1653  
1654  
1655  
1656  
1657  
1658  
1659  
1659  
1660  
1661  
1662  
1663  
1664  
1665  
1666  
1667  
1668  
1669  
1669  
1670  
1671  
1672  
1673  
1674  
1675  
1676  
1677  
1678  
1679  
1679  
1680  
1681  
1682  
1683  
1684  
1685  
1686  
1687  
1688  
1689  
1689  
1690  
1691  
1692  
1693  
1694  
1695  
1696  
1697  
1698  
1698  
1699  
1699  
1700  
1701  
1702  
1703  
1704  
1705  
1706  
1707  
1708  
1709  
1709  
1710  
1711  
1712  
1713  
1714  
1715  
1716  
1717  
1718  
1719  
1719  
1720

# LAB

## Exercise 1 - Initialize your weather report log file

### 1.1 Create a text file called `rx_poc.log`.

This will be your POC weather report log file, simply a text file which contains a growing history of the daily weather data you will scrape. Each entry in the log file corresponds to a row as in [Table 1](#).

```
1
```

```
touch rx_poc.log
```

create log file to get info. of weather

### 1.2 Add a header to your weather report.

Your header should consist of the column names from [Table 1](#), delimited by tabs. Write the header to your weather report.

Using a shell variable and command substitution:

```
1 header=$(echo -e "year\month\day\thour\tobs\tmp\tfc_temp")  
2 echo $header>rx_poc.log
```

→ Type in Linux Terminal!

OR, more directly, using `echo` and redirection:

```
1 echo -e "year\month\day\thour\tobs\tmp\tfc_temp">rx_poc.log  
{: codeblock}
```

to be headers in  
for it or \n

*Tip: Although it might seem redundant, my preference is to use variables in such cases. It makes for much cleaner code, which is easier to understand and safer to modify by others or even yourself at a later date. Using meaningful names for your variables also makes the code more ‘self-documenting’.*

## Exercise 2 - Write a bash script that downloads the raw weather data, and extracts and loads the required data

### 2.1. Create a text file called `rx_poc.sh` and make it a bash script.

Remember to make it executable.

▼ Click here for Solution 1

Include the bash ‘shebang’ on the first line of `rx_poc.sh`:

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

{: codeblock}

▼ Click here for Solution 2

Make your script executable:

```
1 chmod u+x rx_poc.sh
```

{: codeblock}

Verify your changes using the `ls` command with the `-l` option.

create .sh  
bash file)  
and set  
permission

## 2.2 Download today’s weather report from [wttr.in](#)

*Tip: It’s good practice to keep a copy of the raw data you are using to extract the data you need.*

- By appending a date or time stamp to the file name, you can ensure it’s name is unique.
- This builds a history of the weather forecasts which you can revisit at any time to recover from errors or expand the scope of your reports
- Using the prescribed date format ensures that when you sort the files, they will be sorted chronologically. It also enables searching for the report for any given date.
- If needed, you can compress and archive the files periodically. You can even automate this process by scheduling it.

Follow the steps below to download and save your report as a datestamped file named `raw_data_<YYYYMMDD>`, where `<YYYYMMDD>` is today’s date in Year, Month, Day format.

### 2.2.1 Create the filename for today’s [wttr.in](#) weather report

### 2.2.1 Create the filename for today's wttr.in weather report

▼ Click here for full Solution

```
1 today=$(date +%Y%m%d)
2 weather_report=raw_data_${today}
```

format

{: codeblock}

OR, more directly:

```
1 weather_report=raw_data_$(date +%Y%m%d)
```

### 2.2.2 Download the wttr.in weather report for Casablanca and save it with the filename you created

▼ Click here for Hint

> Use the `curl` command with the `--output` option.

▼ Click here for Solution

```
1 city=Casablanca
2 curl wttr.in/$city --output $weather_report
```

specify where curl should  
be save

{: codeblock}

retrieve this city

### 2.3. Extract the required data from the raw data file and assign them to variables `obs_tmp` and `fc_temp`.

Extracting the required data is a process that will take some trial and error until you get it right.  
Study the weather report you downloaded, and determine what you need to extract. Look for patterns.  
You must find a way to 'chip away' at the weather report:

- Use shell commands to extract only the data you need (the *signal*)
- Filter everything else out (the *noise*)
- Combine your filters into a pipeline (recall the use of *pipes* to chain *filters* together)

*Tip:* We introduce three more simple filters here that you will find very useful for your data extraction strategy.

1. `tr` - trim repeated characters to a single character

For example, to remove extra spaces from text:

```
1 echo "There are    too    many spaces in this    sentence." | tr -s "
```

2. `xargs` - can be used to trim leading and trailing spaces from a string.

For example, to remove the spaces from the begining and the end of text:

```
1 echo " Never start or end a sentence with a space. " | xargs
```

3. `rev` - reverse the order of characters on a line of text.

Try entering the following command:

```
1 echo ".sdrawkcab saw ecnetnes sihT" | rev
```

{: codeblock}

You will find `rev` to be a useful operation to apply in combination with the `cut` command.

For example, suppose you want to access the last field in a delimited string of fields:

```
1 # print the last field of the string
2 echo "three two one" | rev | cut -d " " -f 1 | rev → output: one
```

You will also find `xargs` to be a useful operation to apply in combination with the `cut` command.

For example, suppose you want to access the last word in a sentence as above, but there happens to be an extra space at the end:

```
1 # Unfortunately, this prints the last field of the string, which is empty:
2 echo "three two one" | rev | cut -d " " -f 1 | rev → empty
3 # But if you trim the trailing space first, you get the expected result:
4 echo "three two one" | xargs | rev | cut -d " " -f 1 | rev → output: one
```

Let's now return to extracting the temperature data of interest.

▼ Click here for Solution

```
1 grep ^C $weather_report > temperatures.txt
```

find in

redirect output to file

### 2.3.1. Extract the current temperature, and store it in a shell variable called `obs_tmp`

▼ Click here for Solution

Here's the full solution. Ensure you understand what each filter accomplishes.

Using the command line, try adding one filter at a time to see what the outcome is as you develop the pipeline.

```
1 obs_tmp=$(head -1 temperatures.txt | tr -s " " | xargs | rev | cut -d " " -f2 | rev)
```

get first line flag → trim space → to set least → like split  
→ trim begin and tail

### 2.3.2. Extract tomorrow's temperature forecast for noon, and store it in a shell variable called `fc_tmp`

▼ Click here for Solution

set first three lines from

```
1 fc_temp=$(head -3 temperatures.txt | tail -1 | tr -s " " | xargs | cut -d "C" -f2 | rev | cut -d " " -f2 | rev)
```

## 2.4. Store the current hour, day, month, and year in corresponding shell variables

► Click here for Hint

▼ Click here for Solution

```
1 hour=$(TZ='Morocco/Casablanca' date -u +%H)
2 day=$(TZ='Morocco/Casablanca' date -u +%d)
3 month=$(TZ='Morocco/Casablanca' date +%m)
4 year=$(TZ='Morocco/Casablanca' date +%Y)
```

{ codeblock }

### 2.5. Merge the fields into a tab-delimited record, corresponding to a single row in Table 1.

Append the resulting record as a row of data to your weather log file.

► Click here for Hint

▼ Click here for Solution

```
1 record=$(echo -e "$year\t$month\t$day\t$hour\t$obs_tmp\t$fc_temp")
2 echo $record>rx_poc.log
```

{ codeblock } append to

## Exercise 3 - Schedule your bash script `rx_poc.sh` to run every day at noon local time

### 3.1. Determine what time of day to run your script

Recall that you want to load the weather data corresponding to noon, local time in Casablanca, every day.

Check the time difference between your system's default time zone and UTC.

► Click here for Hint 1

Use the `date` command twice with appropriate options.

► Click here for Hint 2

Now you can determine how many hours difference there are between your system's local time and that of Casablanca.

▼ Click here for Solution

From the example above, we see that the system time relative to UTC is UTC+5 (i.e.,  $16 - 11 = 5$ ).

We know Casablanca is UTC+1, so the system time relative to Casablanca is 4 hours earlier. Thus, to run your script at noon Casablanca time, you need to run it at 8 am.

## 3.2 Create a cron job that runs your script

► Click here for Hint

▼ Click here for Solution

Edit the crontab file

```
1 crontab -e
```

Add the following line at the end of the file:

```
1 0 8 * * * /home/project/rx_poc.sh
```

Save the file and quit the editor.

## Full Code

```
1 #! /bin/bash
2
3 # create a dated stamped filename for the raw wttr data
4 today=$(date +%Y%m%d)
5 weather_report=raw_data_$today
6
7 # download today's weather report from wttr.in:
8 city=Casablanca
9 curl wttr.in/$city --output $weather_report
10
11 # use command substitution to store the current day,
12 hour=$(TZ='Morocco/Casablanca' date -u +%H)
13 day=$(TZ='Morocco/Casablanca' date -u +%d)
14 month=$(TZ='Morocco/Casablanca' date +%m)
15 year=$(TZ='Morocco/Casablanca' date +%Y)
16
17 # extract all lines containing temperatures from the
18 grep ^C $weather_report > temperatures.txt
19
20 # extract the current temperature
21 obs_tmp=$(head -1 temperatures.txt | tr -s " " | xargs
22
23 # extract the forecast for noon tomorrow
24 fc_temp=$(head -3 temperatures.txt | tail -1 | tr -s "
25
26 # create a tab-delimited record
27 # recall the header was created as follows:
28 # header=$(echo -e "year\tmonth\tday\thour_UTC\tobs_t
29 # echo $header>rx_poc.log
30
31 record=$(echo -e "$year\t$month\t$day\t$hour\t$obs_tmp\t$fc_
32 # append the record to rx_poc.log
33 echo $record>rx_poc.log
```