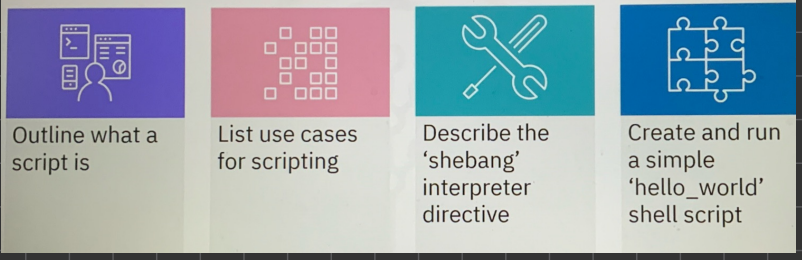
Shell Scripting

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 complied
- 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 [optional-arg]
```

- *'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
$ echo '#! /bin/bash' >> hello_world.sh
$ echo 'echo hello world' >> hello_world.sh
```

↗ create empty text file
turn to a bash script
convention used to indicate that the file is a shell script
↓ to print and redirect that output to bash script

'Hello World' example shell script

Make it executable:

```
$ ls -l hello_world.sh
-rw-rw-r-- 1 jgrom jgrom 12 Jun 27 09:13 hello_world.sh
$ chmod +x hello_world.sh
$ ls -l hello_world.sh
-rwxrwxr-x 1 jgrom jgrom 12 Jun 27 09:13 hello_world.sh
```

1. Owner (us)
2. the group
3. all users

} three user-based permission groups

check current permission

→ showed that it can read and write, but can't execute because it lacks of 'x'

make it executable for all users an 'x'

for all group and users

r = readable and writable

x = executable

'Hello World' example shell script

Run your bash script:

↓ to run

```
$ ./hello_world.sh

hello world
```

A Brief Introduction to Shell Variables

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

```
$ firstname=Jeff
$ echo $firstname
Jeff
```

→ assign string value 'Jeff' to variable 'firstname'

→ display with 'echo $_'

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

```
$ read lastname
```

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

```
$ read lastname
Grossman
$
```

← it's stored to 'lastname'

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

```
$ read lastname
Grossman
$ echo $lastname
Grossman
```

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

```
$ echo $firstname $lastname
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"
```

to prevent the output from ending a newline

to get input to

to not, like %

```
Enter your name :Gusty
Welcome Gusty
Congratulations! You just created and ran your first shell script 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`.

```
which bash
```

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

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:

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

The script should now look like the following:

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

add to first line

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:

```
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:

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

Verify the permissions using the command below:

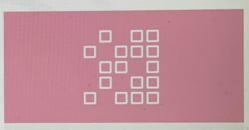
```
ls -l greet.sh
```

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

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

vertical slash

| – pipe command

- For chaining filter commands

```
command1 | command2
```

- Output of command 1 is input of command 2 (become to)
- "Pipe" stands for "pipeline"

chain to

output → input

```
$ ls | sort -r
Videos
Public
Pictures
Music
Downloads
Documents
Desktop
```

outputs have already sorted in reverse

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
BASH=/usr/bin/bash
BASHOPTS=checkwinsize:cm
  dhist:complete_fullquot
  e:expand_aliases:extglo
  b:extquote:force_fignor
  e:globasciiranges:hista
  ppend:interactive_comme
  nts:progcomp:promptvars
  :sourcepath
BASH_ALIASES=()
BASH_ARGC=([0]="0")
```

to show first 4 variable

Defining shell variables

```
var_name=value
```

- No spaces around '='

```
$ GREETINGS="Hello"
$ echo $GREETINGS
Hello

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

$ unset AUDIENCE
```

to clear variable

```
unset var_name
```

- deletes var_name deleted

Environment variables

like shell variables, except they have extended scope

- Extended scope

```
export var_name
```

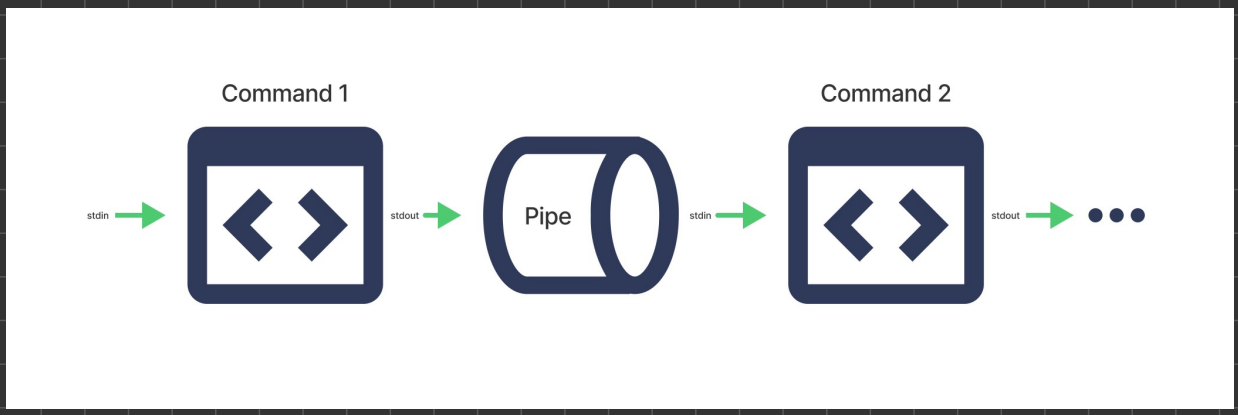
- `env` – list all environment variables

makes GREETINGS an environment variable

```
$ export GREETINGS

$ env | grep "GREE"
$ GREETINGS=Hello
```

← check GREETINGS was exported as environment variable



- 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.

```
$ sort pets.txt | uniq
cat
dog
goldfish
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

```
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 `_` :

```
$ echo "Linux and shell scripting are awesome\!" | tr "aeiou" "_"
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:

```
$ echo "Linux and shell scripting are awesome\!" | tr -c "aeiou" "_"
_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:

```
$ cat pets.txt | tr "[a-z]" "[A-Z]"
GOLDFISH
DOG
CAT
PARROT
DOG
GOLDFISH
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:

```
$ sort pets.txt | uniq | tr "[a-z]" "[A-Z]"
CAT
DOG
GOLDFISH
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.


```
{
  "coin": {
    "id": "bitcoin",
    "icon": "https://static.coinstats.app/coins/Bitcoin6l39t.png",
    "name": "Bitcoin",
    "symbol": "BTC",
    "rank": 1,
    "price": 57907.78008618953,
    "priceBtc": 1,
    "volume": 48430621052.9856,
    "marketCap": 1093175428640.1146,
    "availableSupply": 18877868,
    "totalSupply": 21000000,
    "priceChange1h": -0.19,
    "priceChange1d": -0.4,
    "priceChange1w": -9.36,
    "websiteUrl": "http://www.bitcoin.org",
    "twitterUrl": "https://twitter.com/bitcoin",
    "exp": [
      "https://blockchair.com/bitcoin/",
      "https://btc.com/",
      "https://btc.tokenview.com/"
    ]
  }
}
```


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:

```
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 `.`

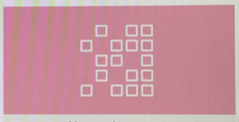
Use the cat command to get the output of the JSON file and pipe it with the grep command to get the required output.

```
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
```

Quoting

\ – escape unique character interpretation

" " – interpret literally, but evaluate metacharacters
except meta-characters

' ' – interpret literally
all contents

```
                    tell dollar sign is text
$ echo "\$1 each"
$1 each

$ echo "$1 each"
 each  ← didn't print $1
space

$ echo '$1 each'
$1 each
```

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

I/O redirection examples

```
$ echo "line1" > eg.txt
$ cat eg.txt
line1
$ echo "line2" >> eg.txt
$ cat eg.txt
line1
line2
```

write
create
appended

```
$ garbage
garbage: command not found
$ garbage 2> err.txt
$ cat err.txt
garbage: command not found
```

← catches the error and redirects to the err.txt

Command substitution

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

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

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

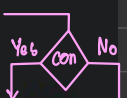
↳ operate in the background ↓ passes control to

```
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
```

OR

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

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 evalutes to `false` .

```
$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 evalutes to `false` .

```
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` :

```
a <= 3
```

and the following statement evalutates to `false` :

```
a <= 1
```

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

```
a=1
b=2
if [ $a -le $b ]
then
   echo "a is less than or equal to b"
else
   echo "a is not less than or equal to b"
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.

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

or

```
a=3
b=2
c=$(($a+$b))
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:

```
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
`+`	addition
`-`	subtraction
`*`	multiplication
`/`	division

*Table: **Arithmetic operators***

Arrays

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

```
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:

```
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:

```
my_array+=("six")
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:

```
# print the first item of the array:
echo ${my_array[0]}

# print the third item of the array:
echo ${my_array[2]}

# print all array elements:
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:

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

or

```
for i in ${!my_array[@]}; do
  echo ${my_array[$i]}
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.

```
N=6
for (( i=0; i<=$N; i++ )) ; do
  echo $i
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:

tell OS this file is script and specifies the inpreter to execute

the path to env command

interpreter that should be used to execute

permission

```
#!/usr/bin/env bash
# initialize array, count, and sum
my_array=(1 2 3)
count=0
sum=0
for i in ${!my_array[@]}; do
  # print the ith array element
  echo ${my_array[$i]}
  # increment the count by one
  count=$(($count+1))
  # add the current value of the array to the sum
  sum=$(($sum+${my_array[$i]}))
done
echo $count
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:

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

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:*

```
#!/bin/bash

echo 'Are you enjoying this course so far?'
echo -n "Enter \"y\" for yes, \"n\" for no."
read response
```

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 reponse 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.*

```
#!/bin/bash

echo 'Are you enjoying this course so far?'
echo -n "Enter \"y\" for yes, \"n\" for no"
read response
if [ "$response" == "y" ]
then
    echo "I'm pleased to hear you are enjoying the co
    echo "Your feedback regarding what you have been
elif [ "$response" = "n" ]
then
    echo "I'm sorry to hear you are not enjoying the c
    echo "Your feedback regarding what we can do to im
    echo "for this course would be greatly appreciated
else
    echo "Your response must be either 'y' or 'n'."
    echo "Please re-run the script to try again."
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

```
#!/bin/bash

echo -n "Enter an integer: "
read n1
echo -n "Enter another integer: "
read n2

sum=$(($n1+$n2))
product=$(($n1*$n2))

echo "The sum of $n1 and $n2 is $sum"
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

```
#!/bin/bash

echo -n "Enter an integer: "
read n1
echo -n "Enter another integer: "
read n2

sum=$(($n1+$n2))
product=$(($n1*$n2))

echo "The sum of $n1 and $n2 is $sum"
echo "The product of $n1 and $n2 is $product."

if [ $sum -lt $product ]
then
    echo "The sum is less than the product."
elif [[ $sum == $product ]]
then
    echo "The sum is equal to the product."
elif [ $sum -gt $product ]
then
    echo "The sum is greater than the product."
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:

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

▸ Click here for Hint

▾ Click here for Solution

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

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

▸ Click here for Hint

▾ Click here for Solution

```
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

```
#!/bin/bash

csv_file="./arrays_table.csv"

# parse table columns into 3 arrays
column_0=($(cut -d "," -f 1 $csv_file))
column_1=($(cut -d "," -f 2 $csv_file))
column_2=($(cut -d "," -f 3 $csv_file))

# print first array
echo "Displaying the first column:"
echo "${column_0[@]}"
```

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

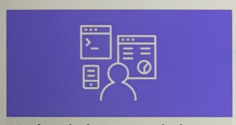
```
## Create a new array as the difference of columns 1
# initialize array with header
column_3=("column_3")
# get the number of lines in each column
nlines=$(cat $csv_file | wc -l)
echo "There are $nlines lines in the file"
# populate the array
for ((i=1; i<$nlines; i++)); do
  column_3[$i]=$((column_2[$i] - column_1[$i]))
done
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.cs
```

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

Scheduling Cron jobs with Crontab

```
$ crontab -e
```

- Opens editor

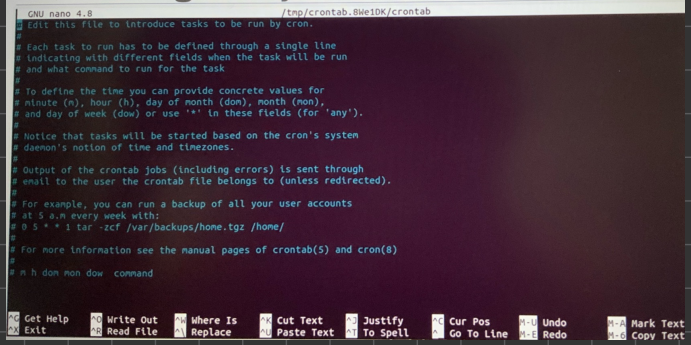
Job syntax:

```
m h dom mon dow command
```

Example job:

```
30 15 * * 0 date >> sundays.txt
```

Scheduling Cron jobs with Crontab



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
```

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
Save modified buffer?
Y Yes
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
```

LAB

Exercise 1 - Understand crontab file syntax

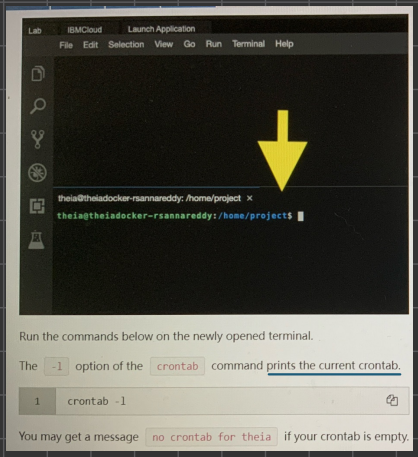
Cron is a system daemon used to execute desired tasks in the background at designated times.

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.

```
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

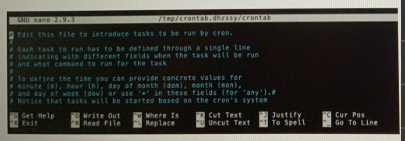
3.1. Add a job to crontab

To add a cron job, run the command below:

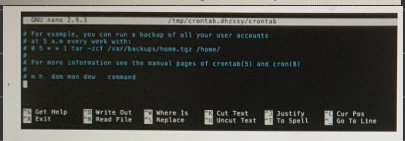
```
crontab -e
```

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:

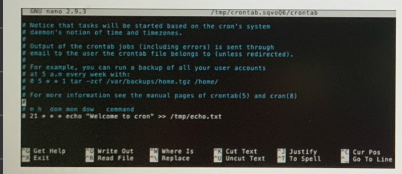


Scroll down to the end of the file using the arrow keys:



Add the below line at the end of the crontab file:

```
0 21 * * * echo "Welcome to cron" >> /tmp/echo.txt
```

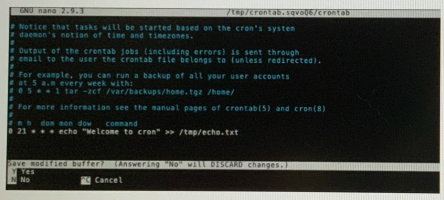


The above job specifies that the `echo` command should run when the minute is 0 and the hour is 21. It effectively means the job runs at 9.00 p.m every day.

The output of the command should be sent to a file `/tmp/echo.txt` .

Press `Ctrl` + `x` to save the changes.

Press `y` to confirm.

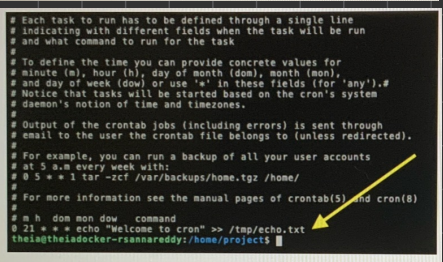


Press Enter to come out of the editor.

Check if the job is added to the crontab by running the following command.

```
crontab -l
```

You should see the newly added job in the output.



3.2. Schedule a shell script

Let us create a simple shell script that prints the current time and the current disk usage statistics.

Step 1: On the menu on the lab screen, use **File->New File** to create a new file:

Step 2: Give the file name as `diskusage.sh` and click 'OK'

Step 3: Save the following commands into the shell script:

```
#! /bin/bash
# print the current date time
date
# print the disk free statistics
df -h
```

Step 4: Save the file using the **File->Save** menu option.

Step 5: Verify that the script is working:

```
chmod u+x diskusage.sh
./diskusage.sh
```

The script should print the current timestamp and the disk usage stats.

Let us schedule this script to be run everyday at midnight 12:00 (when the hour is 0 on the 24 hour clock).
We want the output of this script to be appended to `/home/project/diskusage.log` .

Edit the crontab:

```
crontab -e
```

Add the following line to the end of the file:

```
0 0 * * * /home/project/diskusage.sh >>/home/project/diskusage.log
```

Press `Ctrl` + `x` to save the changes.

Press `y` to confirm.

Press `Enter` to come out of the editor.

Check if the job is added to the crontab by running the following command:

```
crontab -l
```

You should see the newly added job in the output.

Exercise 4 - Remove the current crontab

The `-r` option causes the current crontab to be removed.

> ***Caution:*** *This removes all your cron jobs. Be extra cautious when you use this command on a production server.*

```
crontab -r
```

Verify if your crontab is removed:

```
crontab -l
```

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\tmonth\tday\thour\tobs_tmp\tfc_temp")
2  echo $header>rx_poc.log
```

→ Type in Linux Terminal

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

to be headers in

```
1  echo -e "year\tmonth\tday\thour\tobs_tmp\tfc_temp">rx_poc.log
```

{: codeblock}

for \t 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

format

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

{: codeblock}

> *OR, more directly:*

```
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

specify where curl should be save

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

retrieve this city

{: codeblock}

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:

```
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:

```
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 commmand:

```
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:

```
# print the last field of the string
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:

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

find in

redirect output to file

```
grep °C $weather_report > temperatures.txt
```

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.

get first line flag — pass to — trim space — to set last — like split — trim begin and tail

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

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 line from...

```
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

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

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

append to

{: codeblock}

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 coresponding 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

```
crontab -e
```

Add the following line at the end of the file:

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

Save the file and quit the editor.

Full Code

```
#! /bin/bash

# create a datestamped filename for the raw wttr data
today=$(date +%Y%m%d)
weather_report=raw_data_$today

# download today's weather report from wttr.in:
city=Casablanca
curl wttr.in/$city --output $weather_report

# use command substitution to store the current day,
hour=$(TZ='Morocco/Casablanca' date -u +%H)
day=$(TZ='Morocco/Casablanca' date -u +%d)
month=$(TZ='Morocco/Casablanca' date +%m)
year=$(TZ='Morocco/Casablanca' date +%Y)

# extract all lines containing temperatures from the
grep °C $weather_report > temperatures.txt

# extract the current temperature
obs_tmp=$(head -1 temperatures.txt | tr -s " " | xarg

# extract the forecast for noon tomorrow
fc_temp=$(head -3 temperatures.txt | tail -1 | tr -s

# create a tab-delimited record
# recall the header was created as follows:
# header=$(echo -e "year\tmonth\tday\thour_UTC\tobs_t
# echo $header>rx_poc.log

record=$(echo -e "$year\t$month\t$day\t$obs_tmp\t$fc_
# append the record to rx_poc.log
echo $record>>rx_poc.log
```