Data Engineering Day 13:

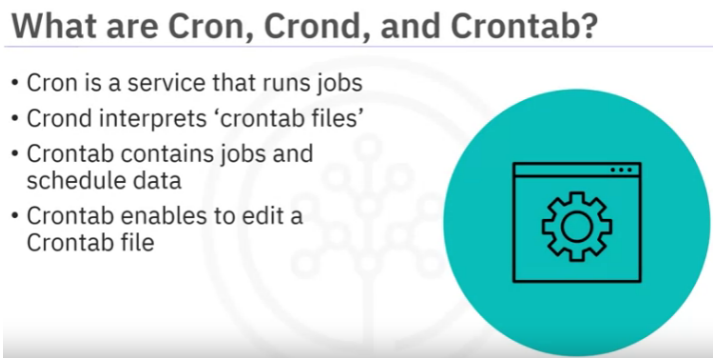
The credit for this course goes to Coursera. Click More

Another link : Azure data Engineer

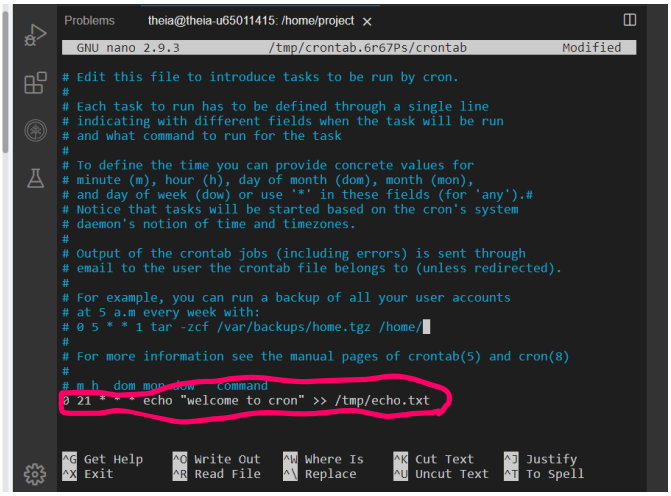
Advanced Shell Scripting

#Cron, Crond and Crontab

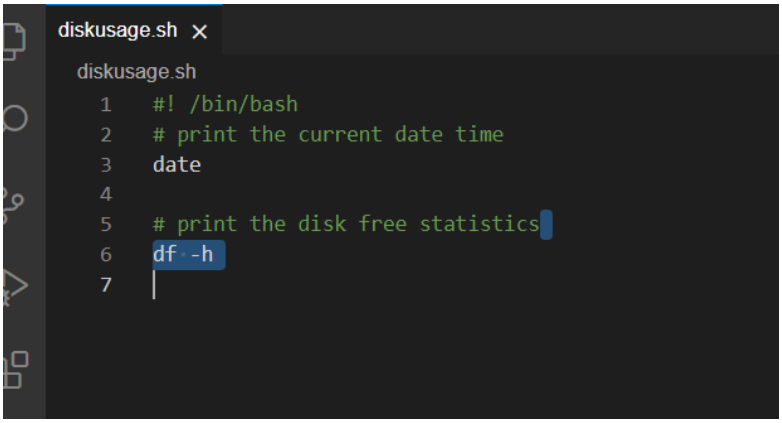
- ⇨ A cron is a system daemon which performs a task in the background of the system in allocated or designated time.
- ⇨ A crontab file is a simple text file containing a list of commands meant to be run at specified times. It is edited using the `crontab` command.

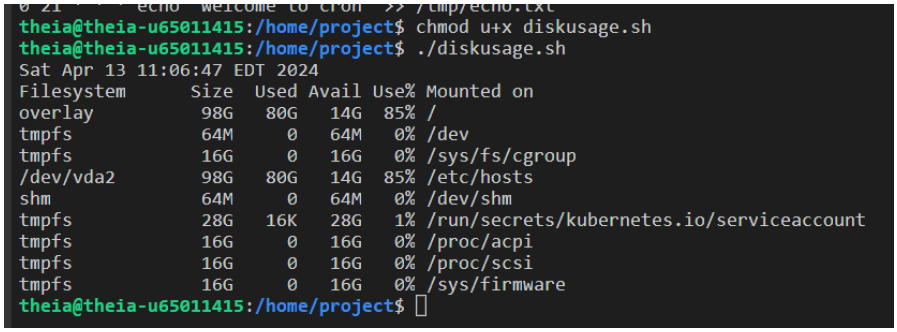


# commands	tasks
`crontab -l`	to check the existing crontab files in the system.
`crontab -e`	to create the new crontab files in the system.



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.





Introduction to Advanced Bash Scripting

Estimated time needed: **5** minutes

In the hands-on lab portion of the final project, you will be using more advanced scripting commands and concepts that the course has not covered yet. This reading will familiarize you with these more advanced concepts, so you can complete the lab with confidence.

Objectives

After completing this reading, you will be able to create Bash scripts that:

- use conditional statements to run a set of commands only if a specified condition is `true`
- apply logical operators to create `true` / `false` comparisons
- perform basic arithmetic calculations
- create list-like arrays and access their elements
- implement `for` loops to execute operations repeatedly, based on a looping index

Conditionals

Conditionals, or `if` statements, are a way of telling a script to do something only under a specific condition.

Bash script conditionals use the following `if - then - else` syntax:

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

If the `condition` is `true`, then Bash executes the statements in `statement_block_1` before exiting the conditional block of code. After exiting, it will continue to run any commands after the closing `fi`.

Alternatively, if the `condition` is `false`, Bash instead runs the statements in `statement_block_2` under the `else` line, then exits the conditional block and continues to run commands after the closing `fi`.

> **Tips:**

> - You must always put spaces around your condition within the square brackets `[ ]`.
> - Every `if` condition block must be paired with a `fi` to tell Bash where the condition block ends.
> - The `else` block is optional but recommended. If the condition evaluates to `false` without an `else` block, then nothing happens within the `if` condition block. Consider options such as echoing a comment in `statement_block_2` to indicate that the condition was evaluated as `false`.

In the following example, the condition is checking whether the number of command-line arguments read by some Bash script, `$#`, is equal to 2.

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

Notice the use of the double square brackets, which is the syntax required for making integer comparisons in the condition `[[ $# == 2 ]]`.

You can also make string comparisons. For example, assume you have a variable called `string_var` that has the value `"Yes"` assigned to it. Then the following statement evaluates to `true`:

```
`[ $string_var == "Yes" ]`
```

Notice you only need single square brackets when making string comparisons.

You can also include multiple conditions to be satified by using the "and" operator `&&` or the "or" operator `||`. For example:

```
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
```

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
```

We've only provided a small sampling of logical operators here. You can explore resources such as the Advanced Bash-Scripting Guide to find out more.

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.ʊTo 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:

```
#!/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.

Summary

In this lab, you learned that:

- Conditional statements can be used to run commands based on whether a specified condition is `true`
- Logical operators do `true` / `false` comparisons
- Arithmetic operators perform basic arithmetic calculations
- You can create list-like arrays and access their individual elements

Module 3 Cheat Sheet - Introduction to Shell Scripting

Bash shebang

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

Get the path to a command

```
which bash
```

Pipes, filters, and chaining

Chain filter commands together using the pipe operator:

```
ls | sort -r
```

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

```
man ls | head -20
```

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

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

Working with shell and environment variables:

List all shell variables:

```
set
```

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

```
my_planet=Earth
```

Display value of a shell variable:

```
echo $my_planet
```

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

```
read first_name
```

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

List all environment variables:

```
env
```

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

```
export my_planet
export my_galaxy='Milky Way'
```

Metacharacters

Comments `#` :

```
# The shell will not respond to this message
```

Command separator `;` :

```
echo 'here are some files and folders'; ls
```

File name expansion wildcard `*` :

```
ls *.json
```

Single character wildcard `?`:

```
ls file_2021-06-??.json
```

Quoting

Single quotes `''` - interpret literally:

```
echo 'My home directory can be accessed by entering: echo $HOME'
```

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

```
echo "My home directory is $HOME"
```

Backslash `\` - escape metacharacter interpretation:

```
echo "This dollar sign should render: \$"
```

I/O Redirection

Redirect output to file and overwrite any existing content:

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

Append output to file:

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

Redirect standard error to file:

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

Append standard error to file:

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

Redirect file contents to standard input:

```
$ tr “[a-z]” “[A-Z]” < a_text_file.txt
```

The input redirection above is equivalent to:

```
$cat a_text_file.txt | tr “[a-z]” “[A-Z]”
```

Command Substitution

Capture output of a command and echo its value:

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

Capture output of a command and echo its value:

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

Command line arguments

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

Batch vs. concurrent modes

Run commands sequentially:

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

Run commands in parallel:

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

Scheduling jobs with cron

Open crontab editor:

```
crontab -e
```

Job scheduling syntax:

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

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

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

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

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

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

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

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

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

Deploy your cron job:

> Close the crontab editor and save the file.

List all cron jobs:

```
crontab -l
```

Conditionals

`if - then - else` syntax:

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

'and' operator `&&` :

```
if [ condition1 ] && [ condition2 ]
```

'or' operator `||` :

```
if [ condition1 ] || [ condition2 ]
```

Logical operators

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

Arithmetic calculations

Integer arithmetic notation:

```
$(())
```

Basic arithmetic operators:

Symbol	Operation
+	addition
-	subtraction
*	multiplication
/	division

Display the result of adding 3 and 2:

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

Negate a number:

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

Arrays

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

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

Add an item to your array:

```
my_array+="six"
my_array+=7
```

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

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

`for` loops

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

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

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

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

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

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

Authors

Jeff Grossman
Sam Propupchuk

Other Contributors

Rav Ahuja