

Bash scripting 2

Bash scripting 2 exercises Operating Systems CSC420 Spring '22

Table of Contents

- [1. README](#)
- [2. Revising shell commands](#)
- [3. Your first bash script](#)
- [4. Standard streams and arguments](#)
- [5. Single and double quotes](#)
- [6. The date program - shell within a shell](#)
- [7. Practice shell within a shell](#)
- [8. Numeric variables in bash](#)
- [9. Express yourself numerically with expr](#)
- [10. Assign numerical and string variables](#)
- [11. The basic calculator bc](#)

1 README

- Work through this file using `replit.com` or `Cygwin` when you have no access to a real bash shell. In class we'll use `replit.com`.
- In this practice, we're going through some of the material in the DataCamp course "Introduction to bash scripting" on DataCamp.
- Text is therefore rather terse - there are no explanations for the tasks. If you find yourself clueless, please go back to the DataCamp lesson and look e.g. at the presentation for the video and/or the transcript below it. You can download the presentation with a button at the upper left of the video dashboard.

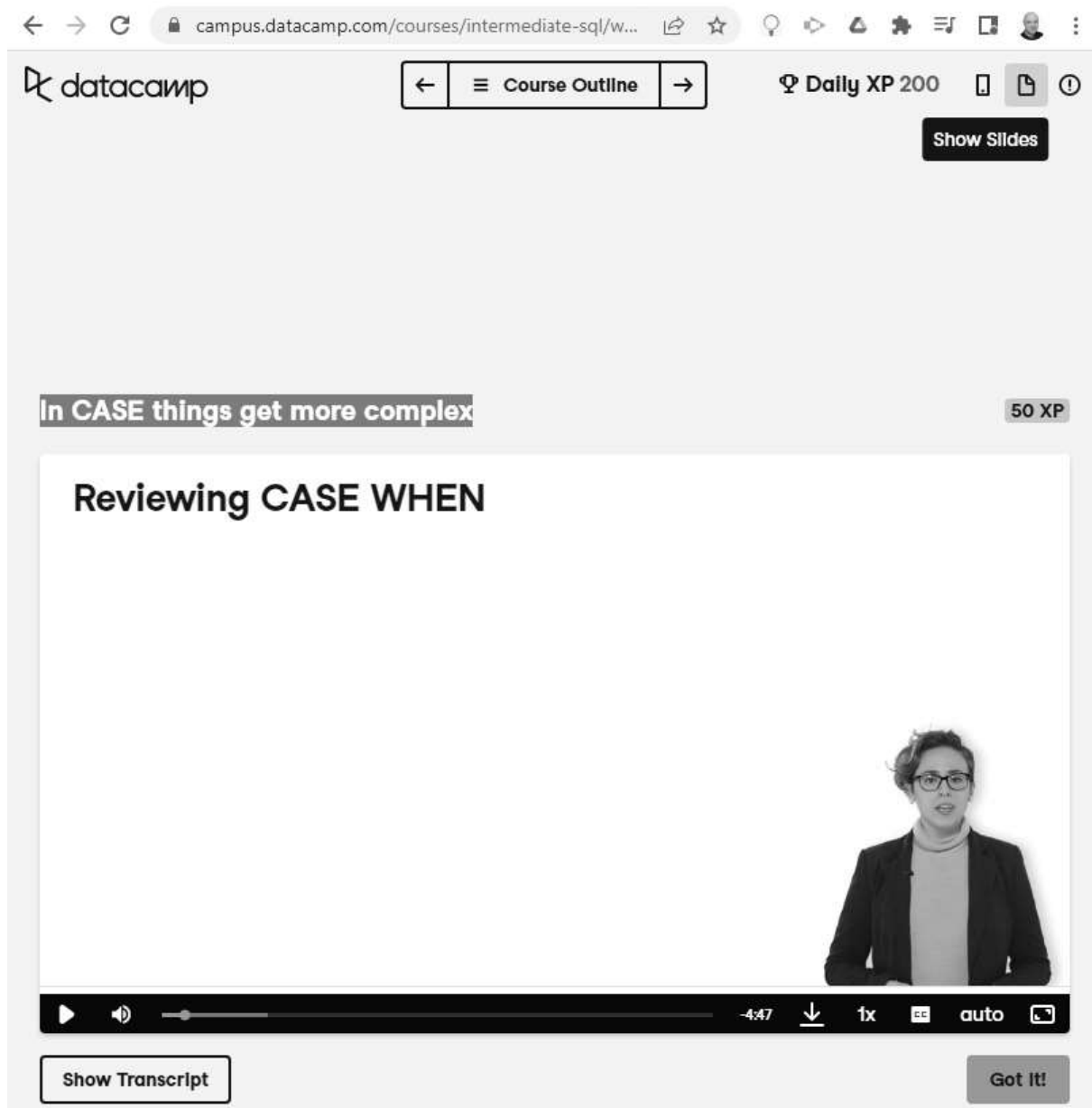


Figure 1: Getting the DataCamp slides for videos

2 Revising shell commands

- []

Create `fruit.txt` with the words `banana`, `apple` and `carrot` each on one line on one line **using the shell** (so not just by editing the file).

```
echo banana > fruits.txt
echo apple >> fruits.txt
echo carrot >> fruits.txt
cat fruits.txt
```

```
banana
apple
carrot
```

- []

Look for the pattern 'a' in fruits.txt.

```
grep 'a' fruits.txt
```

```
banana
apple
carrot
```

- []

Look for any line with a c or an e in fruit.txt.

```
grep '[ce]' fruits.txt
```

```
apple
carrot
```

- []

Invert the search. Look for anything that does not have a c or an e in fruit.txt.

```
grep -v '[ce]' fruits.txt
```

```
banana
```

- []

Sort the file new_fruits.txt, tee off the sorted result, and count only the the *distinct* entries.

```
cat new_fruits.txt | sort | tee new_fruits_sorted.txt | uniq -c | head -n 3
```

```
6 apple
```

```
5 banana
```

```
6 carrot
```

- []

egrep is the same as grep -E and amounts to an OR operator between different patterns. That is, egrep 'p|q' checks if either p or q are found.

Use this knowledge to count the lines in `new_fruits.txt` that contain either 'na' or 'ca'.

```
cat new_fruits.txt | egrep 'na|ca' | wc -l
```

```
11
```

3 Your first bash script

- []

Create a bash file `fruit.sh` that executes the code in 1 that you just created. Print the file first, then run it using the code block below.

```
cat fruit.sh  
. fruit.sh
```

- []

When you run shell commands outside of an Emacs code block, you need to change file permissions. Change permissions for `fruit.sh` so that the owner can run the file from the shell, too, and list the file with permissions.

```
chmod u+x fruit.sh  
ls -l fruit.sh
```

- []

Change the permissions of this file for all using the `700` or `755` octal codes.

```
chmod 755 fruit.sh  
ls -l fruit.sh
```

4 Standard streams and arguments

- []

Create an example script `args.sh` as shown below.

```
#~/usr/bin/bash  
echo $1  
echo $@  
echo "There are " $# "arguments"
```

- []

Run the script with a few words as arguments:

```
. args.sh one two
```

- [X]

Now do it on your own:

- Write a script `echo.sh` that echos out the **second** and **fourth** argument of your input only
- Then echo out the entire input array
- Then echo out the size of the array
- Run the script with the arguments

```
one two three four five six seven
```

Solution:

```
#!/usr/bin/bash
echo "Second argument: $2"
echo "Fourth argument: $4"
echo "Input array: @$@"
echo "Length of the array: $#"
```

Second argument:

Fourth argument:

Input array:

Length of the array: 0

Code to run the file:

```
. echo.sh one two three four five six seven
```

```
Second argument: two
Fourth argument: four
Input array: one two three four five six seven
Length of the array: 7
```

5 Single and double quotes

- []

What should the output of the code below be? Run the code to check your guess.

```
now_var='NOW'
now_var_singlequote='$now_var'
echo $now_var_singlequote
```

```
$now_var
```

```
$now_var
```

- []

What should the output of the code below be? Run the code to check your guess.

```
now_var='NOW'  
now_var_doublequote="$now_var"  
echo $now_var_doublequote
```

```
NOW
```

```
NOW
```

6 The date program - shell within a shell

- [] Print the current data and time on the shell.

```
date
```

```
Sat Apr 16 10:04:51 CDT 2022
```

- []

Generate the output shown below using the "shell within a shell" operator. There are two ways to do this.

Desired output:

```
The date is Wed Apr 13 22:57:49 CDT 2022.
```

Code:

```
rightnow_doublequote="The date is `date`."  
echo $rightnow_doublequote
```

```
The date is Sat Apr 16 10:04:52 CDT 2022.
```

```
rightnow_parentheses="The date is $(date)."  
echo $rightnow_parentheses
```

```
The date is Sat Apr 16 10:04:52 CDT 2022.
```

7 Practice shell within a shell

- []

Which of these three commands uses a "shell within a shell" to print out the date? Guess and then run the code.

```
echo "Right now it is `date`"  
echo "Right now it is `date`"  
echo "Right now it is $date"
```

Right now it is date

Right now it is Sat Apr 16 10:04:52 CDT 2022

Right now it is

8 Numeric variables in bash

- Arithmetic is not automatically built into bash as it is in R e.g. Numbers are not natively supported.

```
1 + 4
```

```
[1] 5
```

Try this on the shell, but redirect the error message to the "bit bucket" `/dev/null` (make it disappear).

Tip: watch the standard error stream (descriptor 2)

*In Emacs, this means that you won't see the *Org-Babel Error Output*.*

Code:

```
1 + 4 > /dev/null 2>&1
```

9 Express yourself numerically with expr

- []

Compute `$1+4~` on the shell using the built-in `expr` function.

Tip: whitespace (empty space) is meaningful on the shell.

Code:

```
expr 1 + 4
```

```
5
```

- []

As you experiment with this, you'll find that `expr` is not reliable. Compute `2*2` with `expr` and then with the **arithmetic expansion operator** `$((...))`

Code:

```
expr 2 * 2
expr 2*2
echo $((2*2))
```

```
2*2
```

```
4
```

On a real shell (not on Windows), you can do niftier things that you may know from C, like compound operators `++i` (prefix only). This would work on a real shell, and it'll work in repl.it.com, too:

```
foo=2      # assigns 2 to variable foo
echo $foo  # print 2
echo $((++foo)) # print 3
```

You can use this to build for loop iterative structures.

/Note: `expr` and `$((...))` only work for **integer constants**.

10 Assign numerical and string variables

- []

Define a string variable `cat_name` and a numerical variable `cat_age`. Assign the values Jack and 1 to them and generate the following output:

```
My cat's name is Jack and his age is 1.
```

Code:

```
cat_name='Jack'
cat_age=1
echo "My cat's name is $cat_name and his age is $cat_age."
```

```
My cat's name is Jack
and his age is 1
.
```


Note: in Emacs, you may get some control characters like ^M, too.

11 The basic calculator bc

- This will **not** work in replit.com because bc is not installed. It will work in Cygwin and under Linux.
- []

Compute $2*2$ using bc and a pipe.

```
echo "2*2" | bc
```

```
4
```

- []

Compute $314 * 0.01$ using bc, the scale attribute, and a pipe.

```
echo "scale=3; 314 * 0.01" | bc
```

```
3.14
```

Author: Bash scripting 2

Created: 2022-04-16 Sat 10:04