Bash Scripting

BASH stands for "Bourne Again Shell"

Commands

(e) grep

filters input based on regex pattern matching

```
grep -i "Unix" my_file.txt
grep 'a' my_file.txt # returns all lines containing a
grep '[pc]' fruits.txt # it has p and c
```

cat

concatenate files contents line by line

```
cat > foo.txt # create a file
cat >> bar.txt # append output to same file
cat old.txt > new.txt # copy a file
cat file.txt # see content
cat file1.txt >> file2.txt # append file to th
                           end of another
cat file1.txt file2.txt > merged.txt
```

tail / head

give only the last -n (a flag) lines

```
tail myfile.txt
head myfile.txt
```

wc

does a word or line count (with flags -w -l)

```
wc mytext.txt
```

sed

does pattern-matched string replacement

```
sed 's/unix/linux/g' myfile.txt # all matches
```

```
cat myfile.txt | sort | uniq -c | head -n 3
```

cut cutting out the sections from each line of files

```
cut -d "," -f 2
```
↳ delimiter ↳ field number

uniq -c report repeated lines

```
uniq -c myfile.txt
```
↳ count lines

Bash Script Anatomy

- It begins with `#!/bin/bash`
- It has a file extension `.sh`
- Can be run in terminal using
  - ↳ `bash script_name.sh`
  - ↳ `./script_name.sh`

STDIN - STDOUT - STDERR

↳ **STDIN** (standard input) Stream of data into the program
↳ **STDOUT** (standard output) Stream of data out of the program
↳ **STDERR** (standard error) Errors in your program

STDIN vs ARGV

ARGV is the array of all arguments given to the program

↳ Each argument can be accessed via $
$1, $2 (first argument, second argument)
↳ $@ and $* - gives all the arguments
↳ $# gives the length (number) of arguments

```
bash args.sh one two three
```

```
#!/usr/bash
echo $1
echo $2
echo $@
```

```
cat hire_data/*.csv | grep "$1" > "$1".csv
```

Variables

```
var1 = "Moon"
```

```
echo $var1
```

```
firstName = `Cynthia'
lastName = `Liu'
```

↳ Never add spaces

```
echo "Hi there" $firstname $lastname
```

Quotation

Single Quotes Shell interprets what is between literally

Double Quotes Shell interprets literally except using $ and backticks

Backticks Shell runs the command and captures STDOUT bash into a variable

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

```
$now_var
```

```
now_var_doublequote = "$now_bar"
echo $now_var_doublequote
```

```
NOW
```

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

```
The date is Mon 5 Aug 2024 01:01
```

You can also use parenthesis instead of backtick

```
rightnow-doublequote = "The date is $(date)"
```

expr

It is a useful utility program which can be used for basic arithmetic

```
expr 1 + 4
```

* Limitations - it does not work with decimals

bc (basic Calculator)

It is a useful command-line program that allows you to do calculations

```
echo "5 + 7.5" | bc
```

* scale argument for how many decimal places

```
echo "scale = 3; 10 / 3" | bc
```

Declare Numbers

```
dog-age = 6
```

Double Bracket Notation

```
echo $((5 + 7))
```
same as `expr 5+7`

```
model1 = 87.65
model2 = 89.20
echo "The total score is $(echo "$model1 + $model2" | bc)"
echo "The average score is $(echo "($model1 + $model2/ 2" | bc)"
```

Arrays

Create numerical-indexed arrays

↳ Declare without adding elements

```
declare -a my_first_array
```

↳ Create and add elements at the same time

```
my_first_array = (1 2 3)
```

```
echo ${my_array[@]}      # returns all array elements
echo ${#my_array[@]}     # returns length of the array
echo ${my_array[1]}      # returns the third element
my_array[0] = 99         # change value of the first element
```

```
array[@]:N:M
```

↳ How many elements to return

↳ Starting index

```
echo ${my_first_array[@]:3:2}
array+=(elements)        # append to the array
my_array += (10)
```

↳ 10 is added to the end

Create associative arrays (dictionaries)

↳ Declaring using multiple lines

```
declare -A city_details
city_details = ([city_name]="Boston" [population]=133)
echo ${city_details[city_name]}
```

x = 10

↳ Declaring using one line [$x -lt 11]; then

```
declare -A city_details=( ["city_name"]="Boston" ["population"]=130000 )
f1
echo ${! city_details[@]} #Return all the keys
```

IF statements

```
if [ CONDITION ]; then
    # SOME CODE
else
    # SOME OTHER CODE
fi
```

Spaces between square brackets and conditional elements inside (first line)

Semi-colon (;) after] closing bracket

```
if (( $x > 5 )); then
    echo "$x is more than 5!"
fi
```

Flags

- -eq equal to
- -ne not equal to
- -lt less than
- -le less than or equal to
- -gt greater than

-gt greater than

-ge greater than or equal to

File-related flags

-e if the file exists

-s if the file exists and has size greater than 0

-r if the file exists and it is readable

-w if the file exists and it is writable

&& for AN

|| for OR

```
x=10
if [ $x -gt 5 ] && [ $x -lt 11 ]; then
    echo "$x is more than 5 and less than 11"
fi
```

```
if [[ $x -gt 5 && $x -lt 11 ]]; then
    echo "$x is more than 5 and less than 11"
fi
```

```
if grep -q Hello world.txt; then
    echo "Hello is inside"
fi
```

```
if $(grep -q Hello words.txt; then
    echo "Hello is inside"
fi
```

For Loop

```
for x in 1 2 3
do
    echo $x
done
```

To create a numeric ranges

↳ { START .. STOP .. INCREMENT }

↳ default at 1 if not specified

```
for x in {1..5..2}
do
    echo $x
done
```

↳ Three expression syntax

First part is the start expression
Middle part is the terminating condition
Third part is increment or decrement

```
for ((x=2; x<=4; x+=2))
do
    echo $x
done
```

↳ Global Expansion

It allows pattern matching

```
for book in books/*
do
    echo $book
done
```

↳ Shell-within-a-shell

```
for book in $(ls books/ | grep -i `air')
do
   echo $book
done
```

While Loop

```
x=1
while [ $x -le 3 ];
do
      echo $x
      (( x+=1 ))
done
```

Case Statement

```
case 'STRINGVAR' in
    PATTERN1)
     COMMAND1;;
    PATTERN2)
     COMMAND2;;
*)
  DEFAULT COMMAND;;
esac
```

↳ Each pattern and command need to be separated by a close-parenthesis

↳ Finish commands with a double semi-colon

↳ Default command with run if no other pattern

are matched

```
case $(cat $1) in
  *sydney* )
    mv $1 sydney/ ;;
  *melbourne* | *brisbane* )
    rm $1 ;;
  *canberra* )
    mv $1 "IMPORTANT_$1" ;;
  * )
    echo "No cities" ;;
esac
```

Bash Function

```
function_name () {
    # function code
    return # something
}
```

* To call the function, just write the name

```
function_name
```

```
function function_name {
    # function code
    return # something
}
```

↳ Passing Arguments

```
function print_filename {

    echo "The first file is $1"
    for file in $@
    do
```

```
        echo "This file has name $file"
    done
}

print_filename "file1.txt" "file2.txt" "A.py"
```

Scope in bash functions

All variables in bash are global by default

↳ To restrict variable scope, use the `local` keyword

```
function print_filename {
    local first_filename = $1
}

print_filename "LO1.txt" "text2.csv"
```

Return Values

The return option in bash is only to determine if the function was successful (0) or failure (other values 1-255). It is capture in the global variable `$?`

To return values, we can

→ Assign to a global variable

→ echo what we want back (last line in a function) and capture using shell-within-a-shell

```
function convert_temp {
    echo $(echo "scale=2; ($1 - 32) * 5/9" | bc)
}
```

```
converted = $(convert_temp 30)
echo "30F in celsius is $converted C"
```

Cron Jobs

crontab driver of cronjobs
↳ file that contains cronjobs

cronjob tells the crontab what code to run and when

crontab -l # to see what schedules (cronjobs) are currently programmed

5 1 * * * bash myscript.sh

↳ It runs everyday at 1:05 am

15 14 * * 7 bash myscript.sh

↳ It runs 2:15 pm every Sunday

To schedule a job

1. In terminal type crontab -e to edit your list of jobs
2. Create the cronjob. Add a new line to the job
   ↳ 30 1 * * * extract_data.sh