Part One

Scripting in the Bourne-Again Shell (bash)

Part 1: Outline

1. executing a shellscript
2. variables, quotes, and substitution
3. shellscript arguments
4. for loops and if statements

Datasets and exercises

Move to directory section-1/

There should be 2 files:

1. `names.txt`
2. `script.sh`

Shellscripting

What are shellscripts

Anything you type into your terminal, can be pasted into a file and executed

The code in the shellscript is read line-by-line by the bash interpreter, just like the lines you type into your terminal

Exercise 1.0

Move to directory section-1/ try the following:

```
$ bash script.sh
$ chmod 755 script.sh
$ ./script.sh
```

‘./’ is used to execute an executable file

script.sh

Open `script.sh` in vi

Note the first line:

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

Hashbang (#!)

You need to tell the system what program should interpret your script

Syntax: #! <path to program>

Examples (first line of file):

#! /bin/bash

#! /usr/bin/python

Comments

Anything following a ‘#’ is ignored by bash

```
# list doc files in the current directory
ls *.doc
```

Comments are notes to human readers

Variables and quotes

Variables

Declaring variables:

```
$ x=5
$ y='a.txt'
```

Accessing variables:

```
$ echo $x
$ echo $y
```

Space in Bash is special

```
$ a=cats and dogs                    # WRONG
bash: and: command not found
$ a = 'cats and dogs'                # Also WRONG
bash: a: command not found
$ a='cats and dogs'                  # RIGHT
```

Combining strings

```
$ x='cats and dogs'
$ y=' and a pony'
$ echo $x$y
cats and dogs and a pony
```

Double and single quotes

```
$ x='Alice'
$ y='Bob'
$ echo "$x sent $y a message"
Alice sent Bob a message
$ echo '$x sent $y a message'
$x sent $y a message
```

Exercise 1.1

Open `script.sh` in vi

Follow the instructions for Exercise 1.1

Parameter expansion ${}

```
$ x='cat'
$ echo $xs   # you want ‘cats’
             # you get absolutely nothing
$ echo ${x}s
cats
```

Command expansion $()

Evaluate a command, retrieving output as a variable

```
# The two lines below have the same output
$ head *.txt
$ head $(ls *.txt)
```

Command expansion (2)

```
$ echo .$(head -2 a.txt).
.Alice Bob.
$ echo “.$(head -2 a.txt).”
.Alice
Bob.
```

Arithmetic expansion $(())

```
$ x=10
$ y=20
$ echo $(( x + y * 2 ))
50
```

Note: within $(()) strings are treated as variables, no ‘$’ needed before the variable

Warnings about variables

An undefined variable is an empty string, e.g.

```
$ echo ${asdf}a
$ ls ${asdf}/*
```

Run these two commands, what happens?

Exercise 1.2

Follow the instructions in `script.sh` for Excercise 1.2

Shellscript Arguments

Sending stuff to the script

All the words that come after a command are the command's arguments, e.g.

```
$ rm -f a.txt b.txt c.txt
  $0 $1   $2    $3    $4
```

Your shellscript can also take arguments

Getting arguments

```
$ cat myscript.sh
#!/bin/bash
echo “$2 $1 $3”

$ ./myscript.sh 10 20 30
20 10 30
```

Arguments must be space-separated

Exercise 1.3

Follow the instructions in `script.sh` for Excercise 1.3

For-loops and if statements

For-loop demo

```
for f in *.csv
do
   echo “processing $f”
   awesome_script.sh $f > ${f}.output
done
```

Bash for-loops: Syntax

```
for x in <list>
do
    <code>
done
# Or replacing newlines with semicolons
for x in <list>; do <code>; done
```

For-loop example (1)

```
#!/bin/bash
for x in 1 2 3; do
   echo $x
done

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

For loop example (2)

```
# For each file, write its head to new file
$ for x in *.txt; do head $x > ${x}.txt; done
$ for x in $(head a.txt); do echo $x > ${x}.txt; done
```

For-loops are particularly useful when you have many input files *and* many output files

Exercise 1.4

Follow the instructions in `script.sh` for Excercise 1.4

If-statement syntax

```
if [[ <condition> ]]
then
    <code>
fi
```

Dying gracefully

```
# If myfile.txt isn't readable stop the script
if [[ ! -r myfile.txt ]]; then
   exit
fi
```

! means NOT

-r tests readability of the file

Useful Tests

```
-r file is readable
-d directory exists
-z test is a variable is empty
```

Conclusion

If you are working with lots of input and output files, calling lots of programs, and making sophisticated pipelines, shellscripts are wonderful.

But don’t code deep logic and algorithms in it.