Introduction and refresher

INTRODUCTION TO BASH SCRIPTING

Alex Scriven
Data Scientist

Introduction to the course

This course will cover:

- Moving from command-line to a Bash script
- Variables and data types in Bash scripting
- Control statements
- Functions and script automation

Why Bash scripting? (Bash)

Firstly, let's consider why Bash?

- Bash stands for '**B**ourne **A**gain **Sh**ell' (a pun)
- Developed in the 80's but a very popular shell today. Default in many Unix systems, Macs
- Unix is the internet! (Running ML Models, Data Pipelines)
  - AWS, Google, Microsoft all have CLI's to their products

Why Bash scripting? (scripting!)

So why Bash scripting?

- Ease of execution of shell commands (no need to copy-paste every time!)
- Powerful programming constructs

Expected knowledge

You are expected to have some basic knowledge for this course.

- Understand what the command-line (terminal, shell) is
- Have used basic commands such as `cat`, `grep`, `sed` etc.

If you are rusty, don't worry - we will revise this now!

Shell commands refresher

Some important shell commands:

- `(e)grep` filters input based on regex pattern matching
- `cat` concatenates file contents line-by-line
- `tail` \ `head` give only the last `-n` (a flag) lines
- `wc` does a word or line count (with flags `-w` `-l`)
- `sed` does pattern-matched string replacement

A reminder of REGEX

'Regex' or *regular expressions* are a vital skill for Bash scripting.

You will often need to filter files, data within files, match arguments and a variety of other uses. It is worth revisiting this.

To test your regex you can use helpful sites like `regex101.com`

Some shell practice

Let's revise some shell commands in an example.

Consider a text file `fruits.txt` with 3 lines of data:

```
banana
apple
carrot
```

If we ran `grep 'a' fruits.txt` we would return:

```
banana
apple
carrot
```

Some shell practice

But if we ran `grep 'p' fruits.txt` we would return:

```
apple
```

Recall that square parentheses are a matching set such as `[eyfv]`. Using `^` makes this an inverse set (**not** these letters/numbers)

So we could run `grep '[pc]' fruits.txt` we would return:

```
apple
carrot
```

Some shell practice

You have likely used 'pipes' before in terminal. If we had many many fruits in our file we could use `sort | uniq -c`

- The first will sort alphabetically, the second will do a count
- If we wanted the top *n* fruits we could then pipe to `wc -l` and use `head`

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

```
  14 apple
  13 bannana
  12 carrot
```

Your first Bash script

INTRODUCTION TO BASH SCRIPTING

Alex Scriven
Data Scientist

datacamp

Bash script anatomy

A Bash script has a few key defining features:

- It usually begins with `#!/usr/bash` (on its own line)
  - So your interpreter knows it is a Bash script and to use Bash located in `/usr/bash`
  - This could be a different path if you installed Bash somewhere else such as `/bin/bash` (type `which bash` to check)
- Middle lines contain code
  - This may be line-by-line commands or programming constructs

Bash script anatomy

To save and run:

- It has a file extension `.sh`
  - Technically not needed if first line has the she-bang and path to Bash (`#!/usr/bash`), but a convention
- Can be run in the terminal using `bash script_name.sh`
  - Or if you have mentioned first line (`#!/usr/bash`) you can simply run using `./script_name.sh`

Bash script example

An example of a full script (called `eg.sh`) is:

```
#!/usr/bash
echo "Hello world"
echo "Goodbye world"
```

Could be run with the command `./eg.sh` and would output:

```
Hello world
Goodbye world
```

Bash and shell commands

Each line of your Bash script can be a shell command.

Therefore, you can also include pipes in your Bash scripts.

Consider a text file (`animals.txt`)

```
magpie, bird
emu, bird
kangaroo, marsupial
wallaby, marsupial
shark, fish
```

We want to count animals in each group.

Bash and shell commands

In shell you could write a chained command in the terminal. Let's instead put that into a script (`group.sh`):

```
#!/usr/bash
cat animals.txt | cut -d " " -f 2 | sort | uniq -c
```

Now (after saving the script) running `bash group.sh` causes:

```
   2 bird
   1 fish
   2 marsupial
```

Standard streams & arguments

INTRODUCTION TO BASH SCRIPTING

Alex Scriven
Data Scientist

STDIN-STDOUT-STDERR

In Bash scripting, there are three 'streams' for your program:

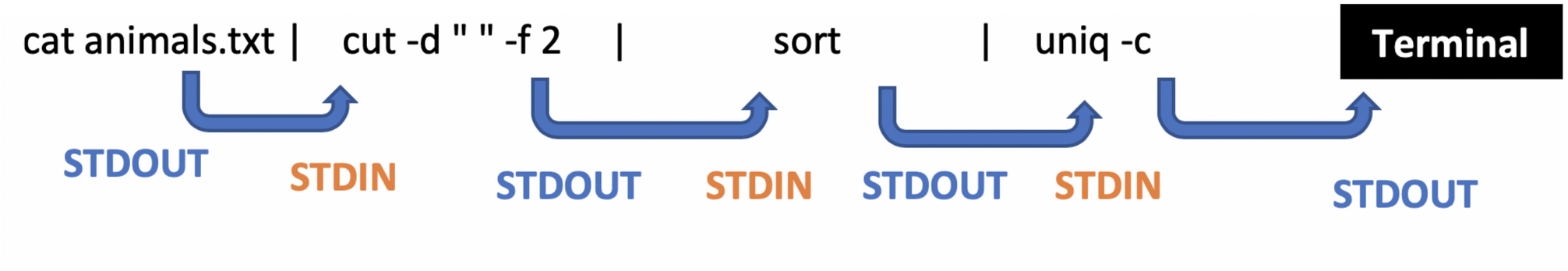
- STDIN (standard input). A stream of data into the program
- STDOUT (standard output). A stream of data **out** of the program
- STDERR (standard error). Errors in your program

By default, these streams will come from and write out to the terminal.

Though you may see `2> /dev/null` in script calls; redirecting STDERR to be deleted. (`1> /dev/null` would be STDOUT)

STDIN-STDOUT graphically

Here is a graphical representation of the standard streams, using the pipeline created previously:



STDIN example

Consider a text file (`sports.txt`) with 3 lines of data.

```
football
basketball
swimming
```

The `cat sports.txt 1> new_sports.txt` command is an example of taking data from the file and writing STDOUT to a new file. See what happens if you `cat new_sports.txt`

```
football
basketball
swimming
```

STDIN vs ARGV

A key concept in Bash scripting is **arguments**

Bash scripts can take **arguments** to be used inside by adding a space after the script execution call.

- ARGV is the array of all the arguments given to the program.
- Each argument can be accessed via the `$` notation. The first as `$1`, the second as `$2` etc.
- `$@` and `$*` give all the arguments in ARGV
- `$#` gives the length (number) of arguments

ARGV example

Consider an example script (`args.sh`):

```
#!/usr/bash
echo $1
echo $2
echo $@
echo "There are " $# "arguments"
```

Running the ARGV example

Now running

`bash args.sh one two three four five`

```
one
two
one two three four five
There are 5 arguments
```

```
#!/usr/bash
echo $1
echo $2
echo $@
echo "There are " $# "arguments"
```

Basic variables in Bash

INTRODUCTION TO BASH SCRIPTING

Alex Scriven
Data Scientist

datacamp

Assigning variables

Similar to other languages, you can assign variables with the equals notation.

```
var1="Moon"
```

Then reference with `$` notation.

```
echo $var1
```

```
Moon
```

Assigning string variables

Name your variable as you like (something sensible!):

```
firstname='Cynthia'
lastname='Liu'
echo "Hi there" $firstname $lastname
```

```
Hi there Cynthia Liu
```

Both variables were returned - nice!

Missing the $ notation

If you miss the `$` notation - it isn't a variable!

```
firstname='Cynthia'
lastname='Liu'
echo "Hi there " firstname lastname
```

```
Hi there firstname lastname
```

(Not) assigning variables

Bash is not very forgiving about spaces in variable creation. Beware of adding spaces!

```
var1 = "Moon"
echo $var1
```

```
script.sh: line 3: var1: command not found
```

Single, double, backticks

In Bash, using different quotation marks can mean different things. Both when creating variables and printing.

- Single quotes (`'sometext'`) = Shell interprets what is between literally
- Double quotes (`"sometext"`) = Shell interprets literally **except** using `$` and backticks

The last way creates a 'shell-within-a-shell', outlined below. Useful for calling command-line programs. This is done with backticks.

- Backticks (`sometext`) = Shell runs the command and captures STDOUT back into a variable

Different variable creation

Let's see the effect of different types of variable creation

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

```
$now_var
```

```
now_var_doublequote="$now_var"
echo $now_var_doublequote
```

```
NOW
```

The date program

The `Date` program will be useful for demonstrating backticks

Normal output of this program:

```
date
```

```
Mon  2 Dec 2019 14:07:10 AEDT
```

Shell within a shell

Let's use the shell-within-a-shell now:

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

```
The date is Mon 2 Dec 2019 14:13:35 AEDT.
```

The date program was called, output captured and combined in-line with the `echo` call.

We used a shell within a shell!

Parentheses vs backticks

There is an equivalent to backtick notation:

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

```
The date is Mon 2 Dec 2019 14:54:34 AEDT.
The date is Mon 2 Dec 2019 14:54:34 AEDT.
```

Both work the same though using backticks is older. Parentheses is used more in modern applications. (See http://mywiki.wooledge.org/BashFAQ/082)

Numeric variables in Bash

INTRODUCTION TO BASH SCRIPTING

Alex Scriven
Data Scientist

datacamp

Numbers in other languages

Numbers are not built in natively to the shell like most REPLs (console) such as R and Python

In Python or R you may do:

```
>>> 1 + 4
```

```
5
```

It will return what you want!

Numbers in the shell

Numbers are not natively supported:

(In the terminal)

```
1 + 4
```

```
bash: 1: command not found
```

Introducing expr

`expr` is a useful utility program (just like `cat` or `grep`)

This will now work (in the terminal):

```
expr 1 + 4
```

```
5
```

Nice stuff!

expr limitations

`expr` cannot natively handle decimal places:

(In terminal)

```
expr 1 + 2.5
```

```
expr: not a decimal number: '2.5'
```

Fear not though! (There is a solution)

Introducing bc

`bc` (basic calculator) is a useful command-line program.

You can enter it in the terminal and perform calculations:

```
~$ bc
bc 1.06
Copyright 1991-1994, 1997, 1998, 2000 Free Software Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type `warranty'.
5 + 7
12
quit
~$
```

Getting numbers to bc

Using `bc` without opening the calculator is possible by piping:

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

```
12.5
```

bc scale argument

`bc` also has a `scale` argument for how many decimal places.

```
echo "10 / 3" | bc
```

```
3
```

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

Note the use of `;` to separate 'lines' in terminal

```
3.333
```

Numbers in Bash scripts

We can assign numeric variables just like string variables:

```
dog_name='Roger'
dog_age=6
echo "My dog's name is $dog_name and he is $dog_age years old"
```

Beware that `dog_age="6"` will work, but makes it a string!

```
My dog's name is Roger and he is 6 years old
```

Double bracket notation

A variant on single bracket variable notation for numeric variables:

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

```
12
12
```

Beware this method uses `expr`, not `bc` (no decimals!)

Shell within a shell revisited

Remember how we called out to the shell in the previous lesson?

Very useful for numeric variables:

```
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)"
```

```
The total score is 176.85
The average score is 88
```

Arrays in Bash

INTRODUCTION TO BASH SCRIPTING

Alex Scriven
Data Scientist

datacamp

What is an array?

Two types of arrays in Bash:

- An array
  - 'Normal' numerical-indexed structure.
  - Called a 'list' in Python or 'vector' in R.

In Python: `my_list = [1,3,2,4]`

In R: `my_vector <- c(1,3,2,4)`

Creating an array in Bash

Creation of a numerical-indexed can be done in two ways in Bash.

1. Declare without adding elements

```
declare -a my_first_array
```

2. Create and add elements at the same time

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

Remember - no spaces around equals sign!

Be careful of commas!

Commas are not used to separate array elements in Bash:

This is **not** correct:

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

This **is** correct:

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

Important array properties

- All array elements can be returned using `array[@]` . Though do note, Bash requires curly brackets around the array name when you want to access these properties.

```
my_array=(1 3 5 2)
echo ${my_array[@]}
```

```
1 3 5 2
```

- The length of an array is accessed using `#array[@]`

```
echo ${#my_array[@]}
```

```
4
```

Manipulating array elements

Accessing array elements using square brackets.

```
my_first_array=(15 20 300 42)
echo ${my_first_array[2]}
```

```
300
```

- Remember: Bash uses zero-indexing for arrays like Python (but unlike R!)

Manipulating array elements

Set array elements using the index notation.

```
my_first_array=(15 20 300 42 23 2 4 33 54 67 66)
my_first_array[0]=999
echo ${my_first_array[0]}
```

```
999
```

- Remember: don't use the `$` when overwriting an index such as `$my_first_array[0]=999`, as this will not work.

Manipulating array elements

Use the notation `array[@]:N:M` to 'slice' out a subset of the array.

- Here `N` is the starting index and `M` is how many elements to return.

```
my_first_array=(15 20 300 42 23 2 4 33 54 67 66)
echo ${my_first_array[@]:3:2}
```

```
42 23
```

Appending to arrays

Append to an array using `array+=(elements)` .

For example:

```
my_array=(300 42 23 2 4 33 54 67 66)
my_array+=(10)
echo ${my_array[@]}
```

```
300 42 23 2 4 33 54 67 66 10
```

(Not) appending to arrays

What happens if you do not add parentheses around what you want to append? Let's see.

For example:

```
my_array=(300 42 23 2 4 33 54 67 66)
my_array+=10
echo ${my_array[@]}
```

```
30010 42 23 2 4 33 54 67 66
```

The string `10` will just be added to the first element. Not what we want!

Associative arrays

- An **associative** array
  - Similar to a normal array, but with key-value pairs, not numerical indexes
  - Similar to Python's dictionary or R's list
  - Note: This is only available in Bash 4 onwards. Some modern macs have old Bash! Check with `bash --version` in terminal

In Python:

```
my_dict = {'city_name': "New York", 'population': 14000000}
```

In R:

```
my_list = list(city_name = c('New York'), population = c(14000000))
```

Creating an associative array

You can only create an associative array using the declare syntax (and uppercase `-A`).

You can either declare first, then add elements or do it all on one line.

- Surround 'keys' in square brackets, then associate a value after the equals sign.
  - You may add multiple elements at once.

Associative array example

Let's make an associative array:

```
declare -A city_details # Declare first
city_details=([city_name]="New York" [population]=14000000) # Add elements
echo ${city_details[city_name]} # Index using key to return a value
```

```
New York
```

Creating an associative array

Alternatively, create an associative array and assign in one line

- Everything else is the same

```
declare -A city_details=([city_name]="New York" [population]=14000000)
```

Access the 'keys' of an associative array with an `!`

```
echo ${!city_details[@]} # Return all the keys
```

```
city_name population
```

IF statements

INTRODUCTION TO BASH SCRIPTING

Alex Scriven
Data Scientist

A basic IF statement

A basic IF statement in Bash has the following structure:

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

Two Tips:

- Spaces between square brackets and conditional elements inside (first line)
- Semi-colon after close-bracket `];`

IF statement and strings

We could do a basic string comparison in an IF statement:

```
x="Queen"
if [ $x == "King" ]; then
    echo "$x is a King!"
else
    echo "$x is not a King!"
fi
```

```
Queen is not a King!
```

You could also use `!=` for 'not equal to'

Arithmetic IF statements (option 1)

Arithmetic IF statements can use the double-parentheses structure:

```
x=10
if (($x > 5)); then
    echo "$x is more than 5!"
fi
```

```
10 is more than 5!
```

Arithmetic IF statements (option 2)

Arithmetic IF statements can also use square brackets and an arithmetic flag rather than (`>` , `<` , `=` , `!=` etc.):

- `-eq` for 'equal to'
- `-ne` for 'not equal to'
- `-lt` for 'less than'
- `-le` for 'less than or equal to'
- `-gt` for 'greater than'
- `-ge` for 'greater than or equal to'

Arithmetic IF statement example

Here we re-create the last example using square bracket notation:

```
x=10
if [ $x -gt 5 ]; then
    echo "$x is more than 5!"
fi
```

```
10 is more than 5!
```

Other Bash conditional flags

Bash also comes with a variety of file-related flags such as:

- `-e` if the file exists
- `-s` if the file exists and has size greater than zero
- `-r` if the file exists and is readable
- `-w` if the file exists and is writable

And a variety of others:

- https://www.gnu.org/software/bash/manual/html_node/Bash-Conditional-Expressions.html

Using AND and OR in Bash

To combine conditions (AND) or use an OR statement in Bash you use the following symbols:

- `&&` for AND
- `||` for OR

Multiple conditions

In Bash you can either chain conditionals as follows:

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

Or use double-square-bracket notation:

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

IF and command-line programs

You can also use many command-line programs directly in the conditional, **removing the square brackets.**

For example, if the file `words.txt` has 'Hello World!' inside:

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

```
Hello is inside!
```

IF with shell-within-a-shell

Or you can call a shell-within-a-shell as well for your conditional.

Let's rewrite the last example, which will produce the same result.

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

```
Hello is inside!
```

FOR loops & WHILE statements

INTRODUCTION TO BASH SCRIPTING

Alex Scriven
Data Scientist

Basic FOR Loop structure

Python:

```
for x in range(3):
    print(x)
```

```
0
1
2
```

R:

```
for (x in seq(3)){
    print(x)
}
```

```
[1] 1
[1] 2
[1] 3
```

FOR Loop in Bash

The basic structure in Bash is a similar:

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

```
1
2
3
```

FOR loop number ranges

Bash has a neat way to create a numeric range called 'brace expansion':

- `{START..STOP..INCREMENT}`

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

```
1
3
5
```

FOR loop three expression syntax

Another common way to write FOR loops is the 'three expression' syntax.

- Surround three expressions with double parentheses
- The first part is the start expression (`x=2`)
- The middle part is the terminating condition (`x<=4`)
- The end part is the increment (or decrement) expression (`x+=2`)

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

```
2
4
```

Glob expansions

Bash also allows pattern-matching expansions into a for loop using the `*` symbol such as files in a directory.

For example, assume there are two text documents in the folder `/books` :

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

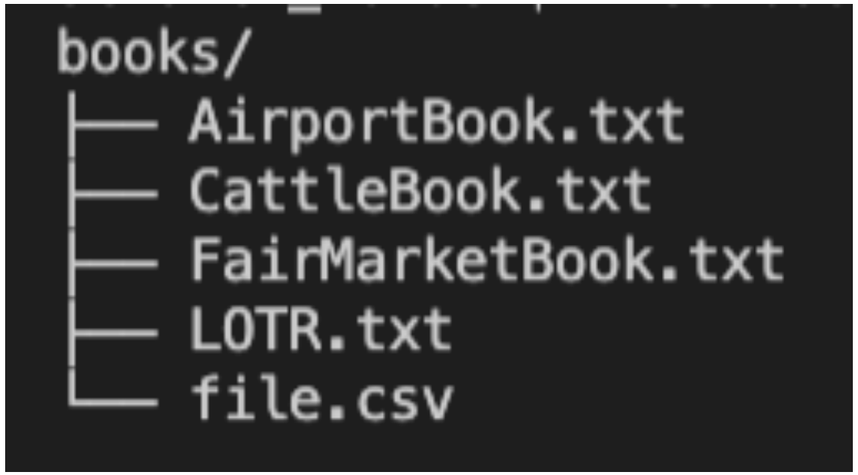
```
books/book1.txt
books/book2.txt
```

Shell-within-a-shell revisited

Remember creating a shell-within-a-shell using `$()` notation?

You can call in-place for a for loop!

Let's assume a folder structure like so:



Shell-within-a-shell to FOR loop

We could loop through the result of a call to shell-within-a-shell:

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

```
AirportBook.txt
FairMarketBook.txt
```

WHILE statement syntax

Similar to a FOR loop. Except you set a condition which is tested at each iteration.

Iterations continue until this is no longer met!

- Use the word `while` instead of `for`
- Surround the condition in square brackets
  - Use of same flags for numerical comparison from IF statements (such as `-le`)
- Multiple conditions can be chained or use double-brackets just like 'IF' statements along with `&&` (AND) or `||` (OR)
- Ensure there is a change inside the code that will trigger a stop (else you may have an infinite loop!)

WHILE statement example

Here is a simple example:

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

```
1
2
3
```

Beware the infinite loop

Beware the infinite WHILE loop, if the break condition is never met.

```
x=1
while [ $x -le 3 ];
do
    echo $x
    # don't increment x. It never reaches 3!
    # ((x+=1))
done
```

This will print out 1 forever!

CASE statements

INTRODUCTION TO BASH SCRIPTING

Alex Scriven
Data Scientist

The need for CASE statements

Case statements can be more optimal than IF statements when you have multiple or complex conditionals.

Let's say you wanted to test the following conditions and actions:

- If a file contains `sydney` then move it into the `/sydney` directory
- If a file contains `melbourne` or `brisbane` then delete it
- If a file contains `canberra` then rename it to `IMPORTANT_filename` where `filename` was the original filename

A complex IF statement

You could construct multiple IF statements like so:

- This code calls `grep` on the first ARGV argument for the conditional.

```
if grep -q 'sydney' $1; then
    mv $1 sydney/
fi
if grep -q 'melbourne|brisbane' $1; then
    rm $1
fi
if grep -q 'canberra' $1; then
    mv $1 "IMPORTANT_$1"
fi
```

- Seems complex and repetitious huh?

Build a CASE statement

- Begin by selecting which variable or string to match against
  - You could call shell-within-a-shell here!
- Add as many possible matches & actions as you like.
  - You can use regex for the `PATTERN`. Such as `Air*` for 'starts with Air' or `*hat*` for 'contains hat'.
- Ensure to separate the pattern and code to run by a close-parenthesis and finish commands with double semi-colon

Basic CASE statement format:

```
case 'STRINGVAR' in
  PATTERN1)
  COMMAND1;;
  PATTERN2)
  COMMAND2;;
```

Build a CASE statement

- `*) DEFAULT COMMAND;;`
  - It is common (but not required) to finish with a default command that runs if none of the other patterns match.
- `esac` Finally, the finishing word is 'esac'
  - This is 'case' spelled backwards!

Basic CASE statement format:

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

From IF to CASE

Our old IF statement:

```
if grep -q 'sydney' $1; then
    mv $1 sydney/
fi

if grep -q 'melbourne|brisbane' $1; then
    rm $1
fi

if grep -q 'canberra' $1; then
    mv $1 "IMPORTANT_$1"
fi
```

Our new CASE statement:

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

Basic functions in Bash

INTRODUCTION TO BASH SCRIPTING

Alex Scriven
Data Scientist

datacamp

Why functions?

If you have used functions in R or Python then you are familiar with these advantages:

1. Functions are reusable
2. Functions allow neat, compartmentalized (modular) code
3. Functions aid sharing code (you only need to know inputs and outputs to use!)

Bash function anatomy

Let's break down the function syntax:

- Start by naming the function. This is used to call it later.
  - Make sure it is sensible!
- Add open and close parentheses after the function name
- Add the code inside curly brackets. You can use anything you have learned so far (loops, IF, shell-within-a-shell etc)!
- Optionally return something (beware! This is not as it seems)

A Bash function has the following syntax:

```
function_name () {
    #function_code
    return #something
}
```

Alternate Bash function structure

You can also create a function like so:

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

The main differences:

- Use the word `function` to denote starting a function build
- You can drop the parenthesis on the opening line if you like, though many people keep them by convention

Calling a Bash function

Calling a Bash function is simply writing the name:

```
function print_hello () {
    echo "Hello world!"
}
print_hello # here we call the function
```

```
Hello world!
```

Fahrenheit to Celsius Bash function

Let's write a function to convert Fahrenheit to Celsius like you did in a previous lesson, using a static variable.

```
temp_f=30
function convert_temp () {
    temp_c=$(echo "scale=2; ($temp_f - 32) * 5 / 9" | bc)
    echo $temp_c
}
convert_temp # call the function
```

```
-1.11
```

Arguments, return values, and scope

INTRODUCTION TO BASH SCRIPTING

Alex Scriven
Data Scientist

datacamp

Passing arguments into Bash functions

Passing arguments into functions is similar to how you pass arguments into a script. Using the `$1` notation.

You also have access to the special `ARGV` properties we previously covered:

- Each argument can be accessed via the `$1`, `$2` notation.
- `$@` and `$*` give all the arguments in `ARGV`
- `$#` gives the length (number) of arguments

Passing arguments example

Let's pass some file names as arguments into a function to demonstrate. We will loop through them and print them out.

```
function print_filename {
    echo "The first file was $1"
    for file in $@
    do
        echo "This file has name $file"
    done
}
print_filename "LOTR.txt" "mod.txt" "A.py"
```

```
The first file was LOTR.txt
This file has name LOTR.txt
This file has name mod.txt
This file has name A.py
```

Scope in programming

'Scope' in programming refers to how accessible a variable is.

- 'Global' means something is accessible anywhere in the program, including inside FOR loops, IF statements, functions etc.
- 'Local' means something is only accessible in a certain part of the program.

Why does this matter? If you try and access something that only has local scope - your program may fail with an error!

Scope in Bash functions

Unlike most programming languages (eg. Python and R), all variables in Bash are global by default.

```
function print_filename {
    first_filename=$1
}
print_filename "LOTR.txt" "model.txt"
echo $first_filename
```

```
LOTR.txt
```

Beware global scope may be dangerous as there is more risk of something unintended happening.

Restricting scope in Bash functions

You can use the `local` keyword to restrict variable scope.

```
function print_filename {
    local first_filename=$1
}
print_filename "LOTR.txt" "model.txt"
echo $first_filename
```

Q: Why wasn't there an error, just a blank line?

Answer: `first_filename` got assigned to the **global** first ARGV element (`$1`).

I ran the script with no arguments (`bash script.sh`) so this defaults to a blank element. So be careful!

Return values

We know how to get arguments in - how about getting them out?

The `return` option in Bash is only meant to determine if the function was a success (0) or failure (other values 1-255). It is captured in the global variable `$?`

Our options are:

1. Assign to a global variable
2. `echo` what we want back (**last line** in function) and capture using shell-within-a-shell

A return error

Let's see a return error:

```
function function_2 {
    echlo # An error of 'echo'
}
function_2 # Call the function
echo $? # Print the return value
```

```
script.sh: line 2: echlo: command not found
127
```

What happened?

1. There was an error when we called the function
   - The script tried to find 'echlo' as a program but it didn't exist
2. The return value in `$?` was 127 (error)

Returning correctly

Let's correctly return a value to be used elsewhere in our script using `echo` and shell-within-a-shell capture:

```
function convert_temp {
    echo $(echo "scale=2; ($1 - 32) * 5 / 9" | bc)
}
converted=$(convert_temp 30)
echo "30F in Celsius is $converted C"
```

```
30F in Celsius is -1.11 C
```

- See how we no longer create the intermediary variable?

Scheduling your scripts with Cron

INTRODUCTION TO BASH SCRIPTING

Alex Scriven
Data Scientist

datacamp

Why schedule scripts?

There are many situations where scheduling scripts can be useful:

1. Regular tasks that need to be done. Perhaps daily, weekly, multiple times per day.
   - You could set yourself a calendar-reminder, but what if you forget!?
2. Optimal use of resources (running scripts in early hours of morning)

Scheduling scripts with `cron` is essential to a working knowledge of modern data infrastructures.

What is cron?

Cron has been part of unix-like systems since the 70's. Humans have been lazy for that long!

The name comes from the Greek word for time, *chronos*.

It is driven by something called a `crontab`, which is a file that contains `cronjobs`, which each tell `crontab` what code to run and when.

Crontab - the driver of cronjobs

You can see what schedules (`cronjobs`) are currently programmed using the following command:

```
crontab -l
```

```
crontab: no crontab for user
```

Seems we need to make a schedule (cronjob) then!

Crontab and cronjob structure

This great image from Wikipedia demonstrates how you construct a `cronjob` inside the `crontab` file. You can have many `cronjobs`, one per line.

```
# ┌─────────────── minute (0 - 59)
# │ ┌─────────────── hour (0 - 23)
# │ │ ┌─────────────── day of the month (1 - 31)
# │ │ │ ┌─────────────── month (1 - 12)
# │ │ │ │ ┌─────────────── day of the week (0 - 6) (Sunday to Saturday;
# │ │ │ │ │                                       7 is also Sunday on some systems)
# │ │ │ │ │
# │ │ │ │ │
# * * * * * command to execute
```

- There are 5 stars to set, one for each time unit
- The default, `*` means 'every'

Cronjob example

Let's walk through some cronjob examples:

`5 1 * * * bash myscript.sh`

- Minutes star is 5 (5 minutes past the hour). Hours star is 1 (after 1am). The last three are `*` , so every day and month
  - Overall: **run every day at 1:05am**.

`15 14 * * 7 bash myscript.sh`

- Minutes star is 15 (15 minutes past the hour). Hours star is 14 (after 2pm). Next two are `*` (Every day of month, every month of year). Last star is day 7 (on Sundays).
  - Overall: **run at 2:15pm every Sunday.**

Advanced cronjob structure

If you wanted to run something multiple times per day or every 'X' time increments, this is also possible:

- Use a comma for specific intervals. For example:
  - `15,30,45 * * * *` will run at the 15,30 and 45 minutes mark for whatever hours are specified by the second star. Here it is every hour, every day etc.
- Use a slash for 'every X increment'. For example:
  - `*/15 * * * *` runs every 15 minutes. Also for every hour, day etc.

Your first cronjob

Let's schedule a script called `extract_data.sh` to run every morning at 1.30am. Your steps are as follows:

1. In terminal type `crontab -e` to edit your list of cronjobs.
   - It may ask what editor you want to use. `nano` is an easy option and a less-steep learning curve than vi (vim).
2. Create the cronjob:
   - `30 1 * * * extract_data.sh`

Your first cron job

3. Exit the editor to save it

If this was using `nano` (on Mac) you would use `ctrl` + `o` then `enter` then `ctrl` + `x` to exit.

You will see a message `crontab: installing new crontab`

4. Check it is there by running `crontab -l`.

```
30 1 * * * extract_data.sh
```

Nice work!

Thanks and wrap up

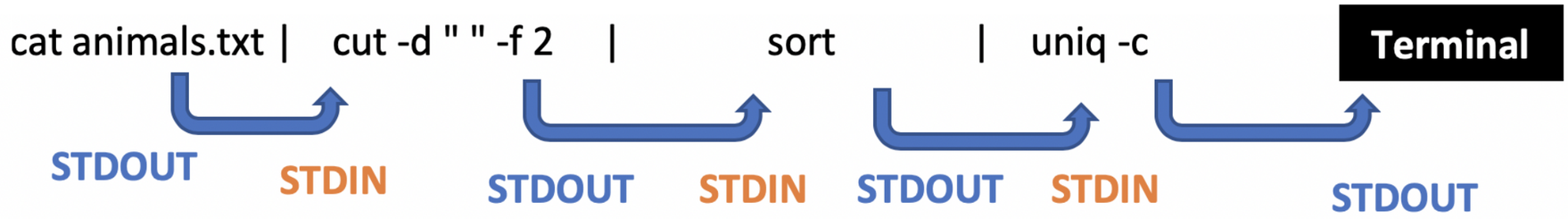
INTRODUCTION TO BASH SCRIPTING

Alex Scriven
Data Scientist

What we covered (Chapter 1)

Chapter 1 - The basics:

- How Bash scripts work with the command-line
- The anatomy of a Bash script
  - Including STDIN, STDERR and STDOUT



Chapter 1 - ARGV

ARGV is the *array* of all the arguments given to the program. ARGV is **vital** knowledge.

- Some special properties we learned:
  - Each argument can be accessed via the `$` notation. (`$1`, `$2` etc.)
  - `$@` (and `$*`) return all the arguments in ARGV
  - `$#` gives the length (number) of arguments

In an example `script.sh`:

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

Call with

`bash script.sh FirstArg SecondArg`

```
FirstArg
FirstAg SecondArg
```

What we covered (Chapter 2)

You learned about creating and using different Bash variables including:

- Creating and using both string, numerical and array variables
  - Arithmetic using `expr` and (for decimals) `bc`
- Different quotation marks mean different things:
  - Single (interpret all text literally)
  - And double (interpret literally **except** `$` and backticks)

Chapter 2 - Shell-within-a-shell

A concept we used again and again (and again!) was the shell-within-a-shell.

- Very powerful concept; calling out to a shell in-place within a script and getting the return value.

```
sum=$(expr 4 + 5)
echo $sum
```

```
9
```

What we covered (Chapters 3 & 4)

Mastering control of your scripts with:

- FOR, WHILE, CASE, IF statements
- Creating functions, calling them and pushing data in (arguments) and out (return values)
- Scheduling your scripts with cron so you don't need to remember to run another script!