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
```

Let's practice!

INTRODUCTION TO BASH SCRIPTING

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?

Let's practice!

INTRODUCTION TO BASH SCRIPTING

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!

Let's practice!

INTRODUCTION TO BASH SCRIPTING

Thanks and wrap up

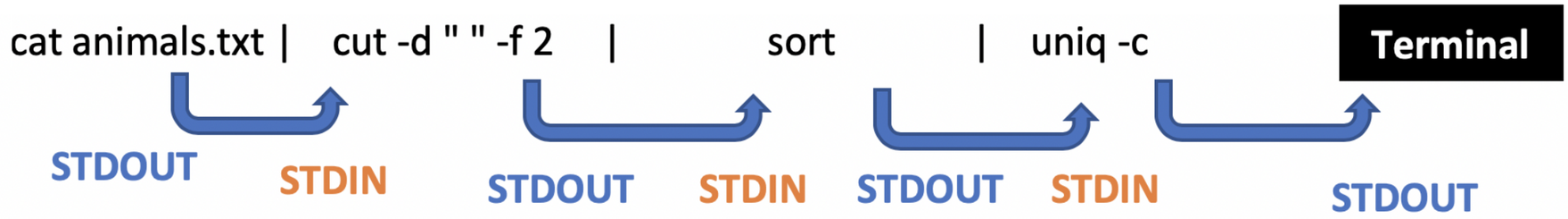
INTRODUCTION TO BASH SCRIPTING

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!

Thank you & Congratulations!

INTRODUCTION TO BASH SCRIPTING