

# CITS4407 Open Source Tools and Scripting

## Shell functions and script design

Unit coordinator: Arran Stewart

# Overview

This week:

- Shell functions and script design
- Regular expressions

# Functions

# Bash scripts

From previous lectures, labs and reading, you should now know what a Bash script looks like:

```
some-script.sh
```

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

```
# A script to say hello
```

```
echo 'Hello World!'
```

# Bash scripts

- We can execute a file like this by typing  
`bash some-script.sh`
- Or, if we make the file *executable*, by typing  
`./some-script.sh`
  - By default, Linux doesn't let us run just any file as a program
  - Linux records whether a file is readable or writable, as well as whether it is *executable*
- So writing a script effectively lets us create a new command – `some-script.sh` – composed out of existing commands.

# Functions

Another way to create a new command is to write a Bash *function*.

Functions are somewhat like scripts – they execute a series of commands – but also like variables; they are stored in memory, and we can over-write them and un-define them.

We can create functions from within scripts, or at the command line.

## Functions at the command line

Functions are created by giving their name, a pair of parentheses – “( )” – and a sequence of commands, contained between braces. Once created, we can treat them much like any other command:

```
$ print_the_date () { echo "the date is: "; date; }  
$ print_the_date  
the date is:  
Tue 23 Mar 12:02:22 AWST 2021
```

Here, our function is all on one line, so we use the semicolon character “;” after each command, to show where it ends.

But you *can* just start typing `print_the_date (`, and Bash will prompt you to keep entering more lines, until you finish by typing a closing brace, `}`.

# Functions in scripts

Or, we can create functions from within a script.

In that case, we usually type one command per line, and don't need the semicolons.

```
another-script.sh
```

```
#!/bin/bash

# define a function
print_the_date () {
    echo "the_date_is:"
    date
}

# invoke the function
print_the_date
```



# Function arguments

Like other commands, functions can take *arguments*.

If we invoke a function like this:

```
my_function alpha beta zeta
```

Then within the function, the strings "alpha", "beta" and "zeta" will be available in variables called \$1, \$2, and \$3.

## Function arguments

So we can write a function that takes one argument – a person's name – and greets that person:

`greet.sh`

```
#!/bin/bash

greet_a_person () {
    echo "Hello, _$1."
    date
}

greet_a_person Bob
```

Running this script will print  
Hello, Bob.

## Similarity to variables

Like variables, functions are stored in memory by Bash.

If you run the “set” command, you can actually see the code for your functions which Bash has stored in memory:

```
$ print_the_date () { echo "the_date_is: "; date; }  
$ set  
# ... many lines omitted ...  
}  
print_the_date ()  
{  
    echo "the_date_is: ";  
    date  
}
```

# Similarity to variables

And you can un-define a function, using `unset -f`:

```
$ print_the_date () { echo "the_date_is:"; date; }  
$ unset -f print_the_date  
$ print_the_date  
print_the_date: command not found
```

## Similarity to variables

Also like shell variables, you can “export” a function, so that its definition will be passed on to spawned programs or sub-shells – use `export -f function_name`

```
$ print_the_date () { echo "the date is: "; date; }  
$ bash -c "print_the_date"  
bash: print_the_date: command not found  
$ export -f print_the_date  
the date is:  
Tue 23 Mar 12:06:50 AWST 2021
```

## Script design

# Tips for script design

The assignments will require you to write your own scripts, so this portion of the lecture provides some advice on how to tackle a programming problem in bash.

# Checking for mistakes

The shellcheck tool<sup>1</sup> helps spot some of the errors typically made by beginning and intermediate Bash programmers.

On Ubuntu 20.04, we can install it with:

```
$ apt-get install shellcheck
```

---

<sup>1</sup>The code for shellcheck is available on GitHub at <https://github.com/koalaman/shellcheck> – it is written in the Haskell programming language.



# Checking for mistakes

## some-script.sh

```
#!/bin/bash

file_to_look_for=$1

if `ls $1` then
    echo y;
else
    echo n;
fi
```

If we try running this file, we get an error message which tells us where Bash had troubles understanding what we mean – but doesn't give much advice on fixing it.

```
./some-script.sh: line 7: syntax error near unexpected token `else'
./some-script.sh: line 7: `else'
```

# Checking for mistakes

## some-script.sh

```
#!/bin/bash

file_to_look_for=$1

if `ls $1` then
    echo y;
else
    echo n;
fi
```

shellcheck provides some advice on what we might need to do:

```
$ shellcheck ./some-script.sh
```

**In ./some-script.sh line 5:**

```
if `ls $1` then
```

```
^-- SC1073: Couldn't parse this if expression.
```

```
    ^-- SC1010: Use semicolon or linefeed before 'then' (or quote to make it literal).
```

**In ./some-script.sh line 7:**

```
else
```

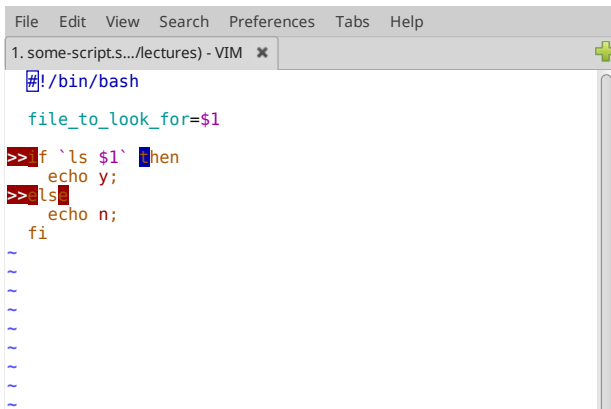
```
^-- SC1050: Expected 'then'.
```

```
    ^-- SC1072: Expected "#". Fix any mentioned problems and try again.
```

# Using a syntax-highlighting editor

You can create your scripts using any simple editor like nano (or gedit).

But more powerful editors like vim understand the syntax of Bash commands, and can also highlight problems in your scripts:



The screenshot shows the Vim text editor interface. The menu bar at the top includes File, Edit, View, Search, Preferences, Tabs, and Help. The tab bar shows a single tab titled "1. some-script.s.../lectures) - VIM". The editor window displays a Bash script with the following content:

```
#!/bin/bash

file_to_look_for=$1

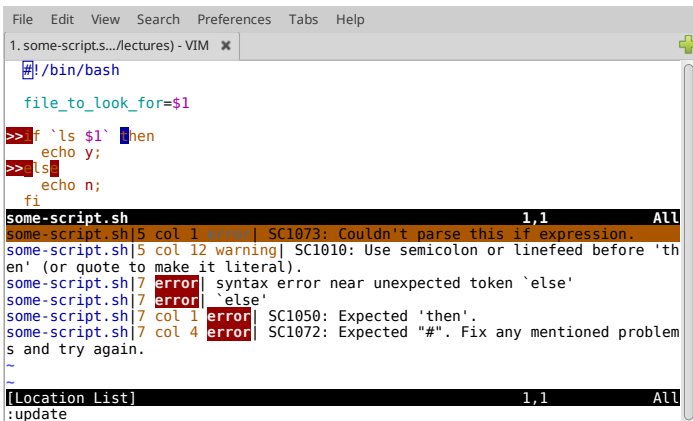
>> if `ls $1` then
    echo y;
>> else
    echo n;
fi

~
~
~
~
~
~
~
```

The script is syntax-highlighted: the shebang is blue, the variable assignment is green, the if statement is red, the echo commands are green, and the fi statement is red. The cursor is positioned at the end of the first echo command in the if block. The bottom right corner of the editor shows navigation icons.

# Using a syntax-highlighting editor

In vim, the command `:lopen` opens a window within vim that contains a list of errors found, and their locations, so the editor can show you errors while you're writing your script:



The screenshot shows a Vim editor window with a menu bar (File, Edit, View, Search, Preferences, Tabs, Help) and a tab titled "1. some-script.s.../lectures) - VIM". The main editor area contains a shell script:

```
#!/bin/bash

file_to_look_for=$1

>> if `ls $1` then
    echo y;
>> ls
    echo n;
fi
```

Below the script, a "Location List" window is open, displaying the following errors:

File	Line	Column	Severity	Message
some-script.sh	5	col 1	error	SC1073: Couldn't parse this if expression.
some-script.sh	5	col 12	warning	SC1010: Use semicolon or linefeed before 'then' (or quote to make it literal).
some-script.sh	7		error	syntax error near unexpected token `else'
some-script.sh	7		error	`else'
some-script.sh	7	col 1	error	SC1050: Expected 'then'.
some-script.sh	7	col 4	error	SC1072: Expected "#". Fix any mentioned problem

The Location List window also shows the command `:update` at the bottom.

## Using a syntax-highlighting editor

If you're interested in setting up `vim` to do this, ask about it in the workshop/labs.

Other editors and IDEs (Integrated Development Environments) exist that will also show you errors – for instance, Visual Studio Code, from Microsoft, or the Eclipse IDE – but we will focus on `vim`, as it is available on a wider range of Linux systems.

# Start small and prototype

The *Shotts* textbook suggests using “top-down design” to solve problems – try to break a problem down into smaller parts, and solve *them*.

This is good advice – but sometimes you may not know how to solve the smaller problems either, at first.

My suggestion – make a new script that allows you to experiment with solving the smaller problem. Once you understand it – incorporate your understanding into the larger script.

## Always have a working script

Don't ever let errors stay in your script.

If Bash starts giving syntax errors – fix them, before trying to write anything else.

There's no point writing *more* code, if the code you have doesn't work.

# Commit frequently

Ensure you have a way of getting back to the last running version of your script.

We suggest you create a Git repository for your code, and frequently add the files you have changed, and `commit` them.

You can store your code on GitHub, if you like; but, a warning! Don't store your code in a *public* repository; make it “private”.

A public repository on GitHub is viewable by everyone, and sharing your code for an assignment breaches the University's Policy on Academic Conduct – the CITS4407 assignments are to be worked on individually.