Edith Cowan University
School of Science

CSI6203 Scripting Languages

Module 2

Interactive Scripts

Contents

- Interactive input
- Formatted output
- Comments
- Bash scripting standards
- Debugging

Learning Objectives

After finishing this module, you should be able to:

- Understand and execute scripts that require interactivity
- Write interactive scripts
- Debug scripts
- Control the formatting of inputs and outputs in interactive scripts

Simple output

echo

- echo is a built in bash command that outputs text to the standard output
- The fundamental use of echo that we have seen is to print simple text on the screen, followed by a newline character (enter)

echo

- It’s not always the intended behaviour for the text we print to also contain the newline key.

- For example
  - When asking a user to input information
  - When only part of a line has been printed
  - When the output is being used by another program or script that does not expect a newline

echo

- It’s not always the intended behaviour for the text we print to also contain the newline key.

- For example
  - When asking a user to input information
  - When only part of a line has been printed
  - When the output is being used by another program or script that does not expect a newline

echo

- With newlines:

```
echo "My name is"
echo "Frank"
```

```
My name is
Frank
```

- Without newlines:

```
echo -n "My name is "
echo "Frank"
```

```
My name is Frank
```

Escape Sequences

- Some characters are difficult to type in a script due to their use in the text editor
- For Example
  - Enter (adds a new line in the text editor)
  - Tab (adds an amount of space)
  - ctrl+c (copy)
  - backspace (delete characters)
- These characters can be printed using “Escape sequences”
- Escape sequences can be enabled in echo using the “-e” option

Escape Sequences

- Good use of escape sequences can be used to format outputs

```
echo -e "My name\n\nis\tFrank\n"
```

```
My name

is      Frank
```

Escape Sequences

Escape Sequence	Output
\b	Backspace
\c	Remove extra output (such as newlines)
\n	Newline
\r	Return to start of line (Carriage Return)
\t	Tab character
\v	Vertical Tab character
\\	Backslash (\ )

User Input

Using read for input

- Besides using command-line arguments like $1, $2, $3, etc. Scripts can be made interactive through the “read” command
- read will pause execution of the script and wait for input from stdin (the standard input stream)
- By default, this will be a user typing information into their terminal

Using read for input

```bash
#!/bin/bash
echo -n "What is your name?"
read
echo "Hello $REPLY"
```

- Whatever the user types when the script hits the “read” line will be stored in the $REPLY variable by default

Using read for input

```
#!/bin/bash
echo -n "What is your name?"
read name
echo "Hello $name"
```

- If the programmer specifies a variable name, that variable will be used instead.

Using read for input

```
#!/bin/bash
read -p "What is your name?" name
echo "Hello $name"
```

- Using read with –p can allow the read command to print a prompt, removing the need for a separate echo statement

Comments

- Any statement in a script that starts with a ‘#’ symbol is ignored by the computer

- This is very useful for leaving notes and explanations for yourself in your scripts to remind you of how and why they work

Comments

```
#!/bin/bash
#This script greets the user
#Author: Rob



#use read to load the user’s name
#into the “name” variable
read -p "What is your name?" name

#greet the user
echo "Hello $name"
```

hiding characters in read

- The input text can also be hidden for sensitive information using the –s option

```
#!/bin/bash
read -p "What is your name?" name
echo "Hello $name"
read -s -p "What is your password?" pass
echo "the secret is: $pass"

read –n1 -p "Press any key to continue"
```

Command and Scripting Standards

Command and Scripting standards

- There are some common things to expect in the behaviour of commands and scripts
- We’ve seen “options” used a few times, such as -n, -p, -s, -e
- Many commands use options. Later in the unit, our own scripts will also make use of options

Command and Scripting standards

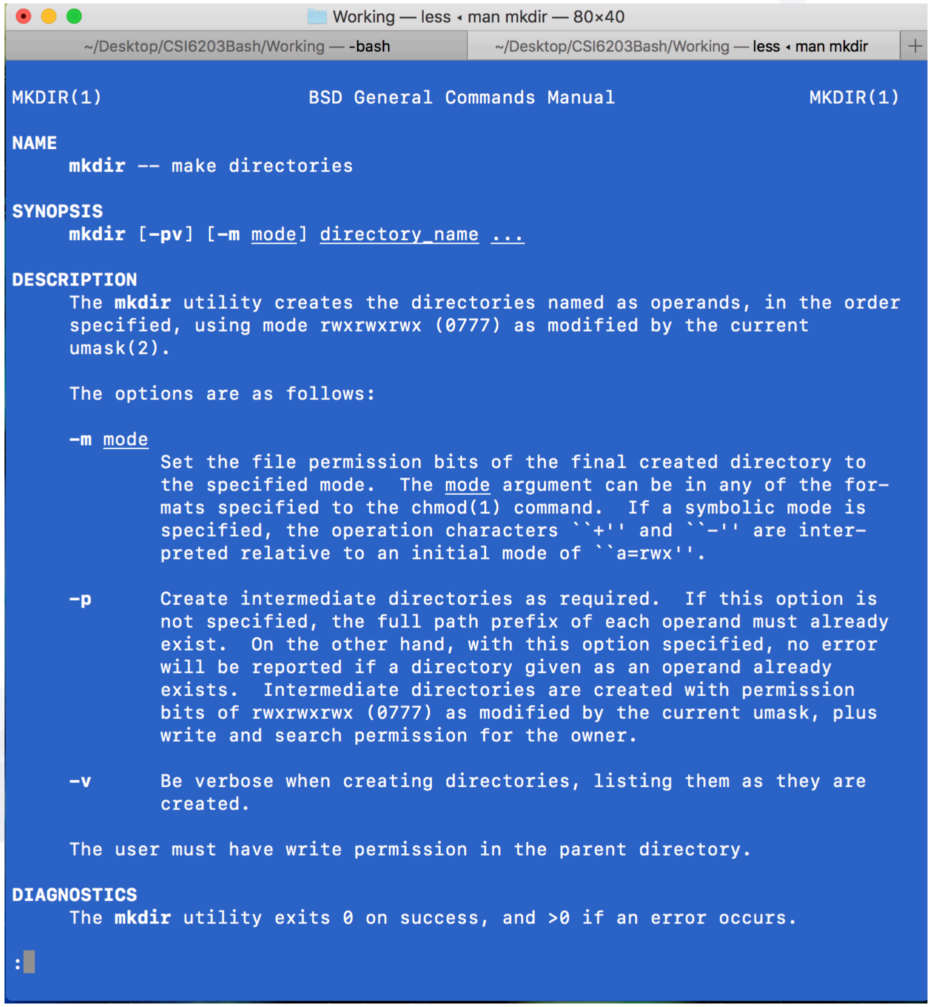
- Most commands and scripts will support some common options. eg.

Option	Common use
-h or --help	Print help text or usage instructions
-a or --all	Include all items
-e	Expand
-f	Specify a filename
-l or --list	Print a list of items
-r	Do something recursively
-v or --verbose	Print additional information while running

Command and Scripting standards

- Not all scripts or commands will support all common options
- Some commands will invent their own or use different meanings
  - eg. echo –e for “escape” not “expand”
- To print detailed help on a command, use the manual (man) pages built in to most operating systems

Manual Pages


Working — less ‹ man mkdir — 80×40
~/Desktop/CSI6203Bash/Working — -bash
~/Desktop/CSI6203Bash/Working — less ‹ man mkdir

```
MKDIR(1)                  BSD General Commands Manual                 MKDIR(1)

NAME
     mkdir -- make directories

SYNOPSIS
     mkdir [-pv] [-m mode] directory_name ...

DESCRIPTION
     The mkdir utility creates the directories named as operands, in the order
     specified, using mode rwxrwxrwx (0777) as modified by the current
     umask(2).

     The options are as follows:

     -m mode
             Set the file permission bits of the final created directory to
             the specified mode.  The mode argument can be in any of the for-
             mats specified to the chmod(1) command.  If a symbolic mode is
             specified, the operation characters ``+'' and ``-'' are inter-
             preted relative to an initial mode of ``a=rwx''.

     -p      Create intermediate directories as required.  If this option is
             not specified, the full path prefix of each operand must already
             exist.  On the other hand, with this option specified, no error
             will be reported if a directory given as an operand already
             exists.  Intermediate directories are created with permission
             bits of rwxrwxrwx (0777) as modified by the current umask, plus
             write and search permission for the owner.

     -v      Be verbose when creating directories, listing them as they are
             created.

     The user must have write permission in the parent directory.

DIAGNOSTICS
     The mkdir utility exits 0 on success, and >0 if an error occurs.

:
```

Errors

- Often when we write our own scripts, they do not do what we want them to do

- This is normal and even expert programmers will still have trouble writing perfect scripts

Errors

- Think of all the times that you have seen a program freeze, crash or show an error message

- This is software written by professionals employed by some of the largest companies in the world and their code STILL doesn’t work perfectly

Types of Errors

- Syntax errors
  - Caused by mistakes in code structure
  - Can be caused by
    - Misspelling (eg. ecko instead of echo)
    - Incorrect symbol use (missing $ in variables)
    - Invalid options for commands (-z does not exist)
- If you're lucky, a useful error will tell you something is wrong:

```
-bash: ecko: command not found
```

Types of Errors

- Logical errors
  - Caused by mistakes in sequence and logic
  - Can be caused by
    - Typing commands in the wrong order
    - Using incorrect conditional statements
    - Not understanding the problem
- Logical errors are much harder to find

```
Your name is
please enter your name:
```

Debugging

- The process of finding these errors is called “Debugging”

- bash offers a few debugging options to help with this

Verbose mode

- bash can run a script in verbose mode which, not only prints the output of the script, but will also print how each command is processed as it is executed.
- To enable verbose mode, use the –v option

Execution mode

- bash can run a script in execution mode which, not only prints the output of the script, but will also print each command as it is executed.
- This is far more common than verbose mode
- To enable execution mode, use the –x option

Execution mode

Without verbose mode

```
$ ./my_script.sh
Your name is
please enter your name:
```

With verbose mode

```
$ bash –x ./my_script.sh
echo "your name is $name"
Your name is
read -p "please enter your name:" name
please enter your name:
```

Execution mode

Without verbose mode

```
$ ./my_script.sh
Your name is
please enter your name:
```

With verbose mode

```
$ bash –v ./my_script.sh
echo "your name is $name"
Your name is
read -p "please enter your name:" name
please enter your name:
```

Command

Execution mode

Without verbose mode

```
$ ./my_script.sh
Your name is
please enter your name:
```

With verbose mode

```
$ bash –v ./my_script.sh
echo "your name is $name"
Your name is
read -p "please enter your name:" name
please enter your name:
```

Output

Execution mode

Without verbose mode

```
$ ./my_script.sh
Your name is
please enter your name:
```

With verbose mode

```
$ bash –v ./my_script.sh
echo "your name is $name"
Your name is
read -p "please enter your name:" name
please enter your name:
```

Command

Execution mode

Without verbose mode

```
$ ./my_script.sh
Your name is
please enter your name:
```

With verbose mode

```
$ bash –v ./my_script.sh
echo "your name is $name"
Your name is
read -p "please enter your name:" name
please enter your name:
```

Output

Execution mode

Without verbose mode

```
$ ./my_script.sh
Your name is
please enter your name:
```

With verbose mode

```
$ bash –v ./my_script.sh
echo "your name is $name"
Your name is
read -p "please enter your name:" name
please enter your name:
```

Output

Debuggers

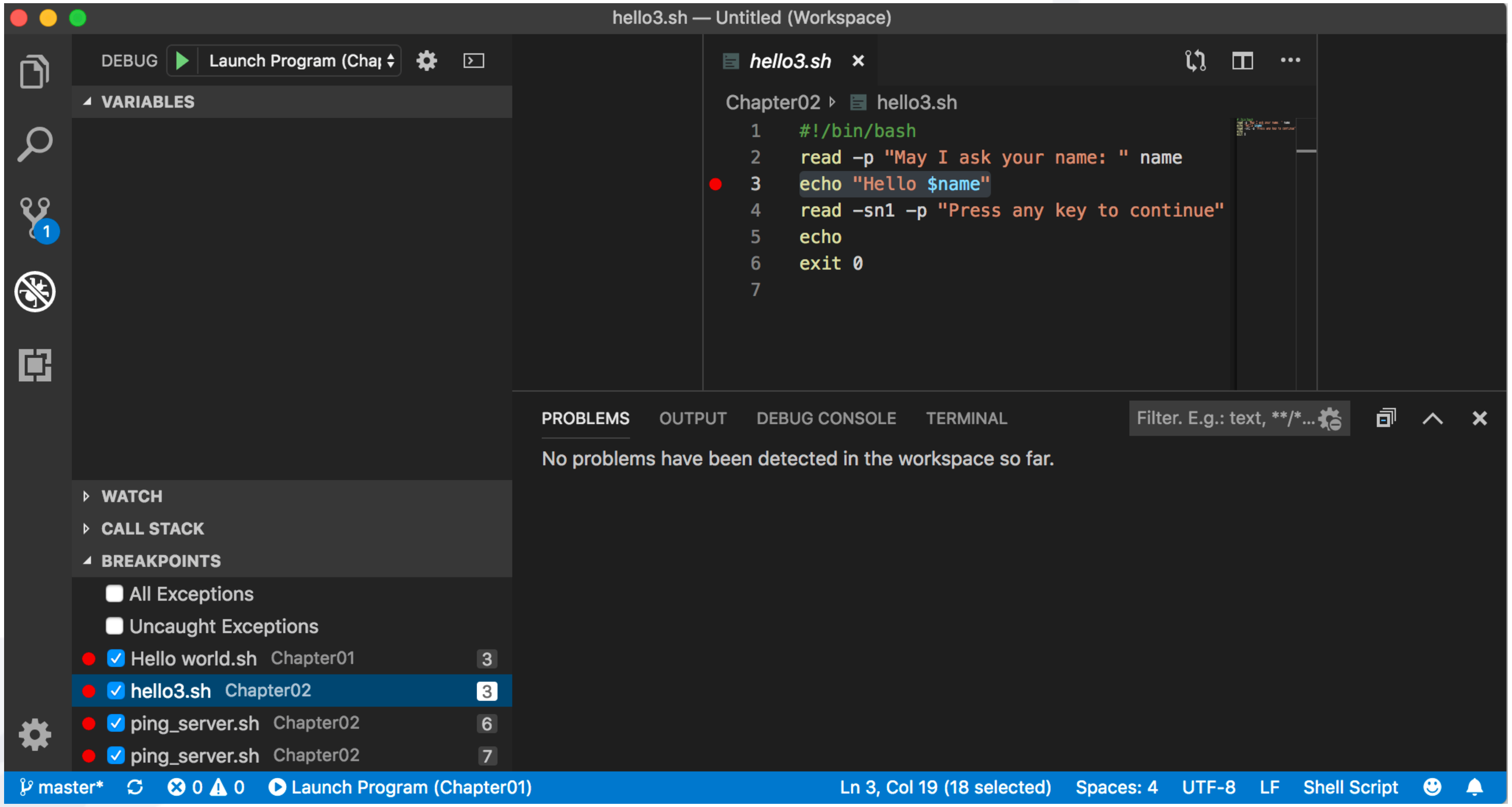
- Some advanced text editors allow the use of debuggers to step through each command in the editor as it is run
- This is very useful for figuring out where scripts are going wrong

Debuggers

- Visual Studio Code has a visual debugger using the popular “bash debug” plugin

- VSCode uses a configuration file called “launch.json” to enable debugging options

Debuggers



hello3.sh — Untitled (Workspace)

DEBUG Launch Program (Cha

VARIABLES

Chapter02 › hello3.sh

```
#!/bin/bash
read -p "May I ask your name: " name
echo "Hello $name"
read -sn1 -p "Press any key to continue"
echo
exit 0
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

No problems have been detected in the workspace so far.

WATCH

CALL STACK

BREAKPOINTS

- ☐ All Exceptions
- ☐ Uncaught Exceptions
- ● ☑ Hello world.sh Chapter01 3
- ● ☑ hello3.sh Chapter02 3
- ● ☑ ping_server.sh Chapter02 6
- ● ☑ ping_server.sh Chapter02 7

master* ⊗ 0 ⚠ 0 Launch Program (Chapter01) Ln 3, Col 19 (18 selected) Spaces: 4 UTF-8 LF Shell Script

Breakpoints

- With a visual debugger, script execution can be paused at a specific location
- This allows a programmer to check to see what the contents of variables are at that point in the script and identify any errors that are occurring at that moment

Summary

- Terms to review and know include:
  - Interactive Input
  - Formatting
  - Command-line options
  - Comments
  - Escape characters
  - Debugging
  - Syntax and Logical errors
  - Breakpoints

References and Further Reading

- Ebrahim, M. and Mallet, A. (2018) Mastering Linux Based Scripting (2nd Ed) Chapter 2, pp 35-52

- http://tldp.org/LDP/Bash-Beginners-Guide/html/sect_08_02.html
- https://ss64.com/bash/read.html
- http://www.manpagez.com/man/1/getopt/
- https://ss64.com/bash/getopts.html