COMP2101
Summer 2022

bash Scripting Introduction

©DENNIS SIMPSON 2013-2022

Bash Background

- Linux is a derivative of UNIX and is very similar to UNIX
- UNIX was built on a philosophy of creating tools that each do a clearly defined task well, and making them work together
- Bash is typically the default shell program used to provide users with a command line interface to the UNIX and Linux operating systems, and is primarily used to start other programs
- Bash was produced in the late 1980s implementing the design philosophy and command structures of existing shells
- Bash is open source and actively maintained

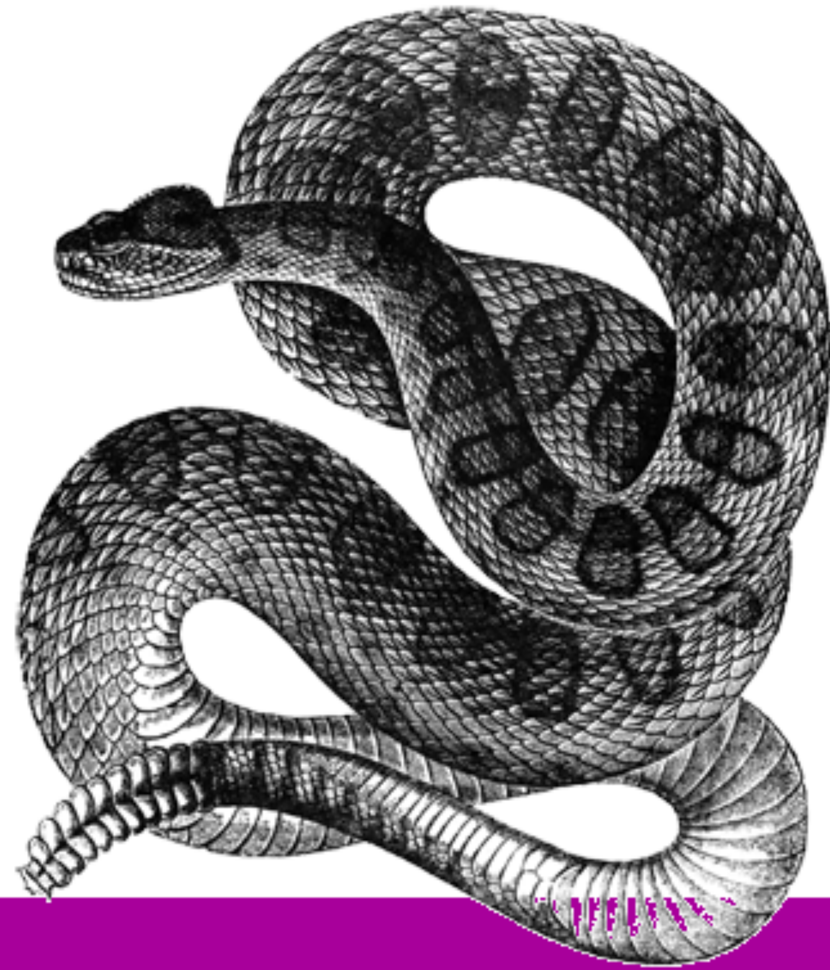
©DENNIS SIMPSON 2013-2022

Scripting With Bash

- Execution is structured and data is treated as a stream of text
- A script is simply one or more commands saved to a file
- A bash script is a script that is run by the bash program and the commands in it must be commands that work on the bash command line
- MacOS bash is the same as Linux bash, but MacOS bash scripts are often not portable to Linux systems due to many differences in the rest of the system

©DENNIS SIMPSON 2013-2022

Basic Topics



- Script creation and execution
- Script content
- Script storage

©DENNIS SIMPSON 2013-2022

Script Creation/Modification

- Scripts are plain text files that are created and edited the same as any other plain text file
- This is usually done with a command line interface (CLI) text editor such as nano or vi or a graphical user interface (GUI) program such as gedit, atom or vscode (packaged as codium)
- Although the operating system does not require any special file naming for scripts, .sh is typically used as the file suffix to overcome limitations in GUI programs
- Any program that puts plain text in a file can create a script
- Word processing programs do not normally create plain text files

```
nano scriptfile.sh
```

```
vi scriptfile.sh
```

```
cat > file.sh <<EoF
script stuff
EoF
```

©DENNIS SIMPSON 2013-2022

Script Execution

- Scripts can be run as commands or by specifying them as an argument (command line data) to the bash command
- Running a script as a command requires execute permission for the script file and that the shell can find the script file (it only looks in the directories listed in the PATH variable)
- Scripts can be copied and pasted onto a bash command line if you want to test the commands, be careful if you try to do this between Windows and other operating systems in VMs

```
cat > scriptfile.sh <<EOF
#!/bin/bash
echo "running script"
EOF
```

```
bash scriptfile.sh
```

```
chmod u+x scriptfile.sh
./scriptfile.sh
```

```
mv scriptfile.sh ~/bin
scriptfile.sh
```

©DENNIS SIMPSON 2013-2022

Script Content

- Scripts can contain commands, blank space, comments, and inline data
- Scripts contain a minimum of one command, with no practical limits on script length
- Commands in scripts are the exact same commands you could use on the command line interactively
- Scripts end when they encounter a fatal bash error, or the exit command, or run out of commands

helloworld.sh:

```
#!/bin/bash
# My first script

echo 'Hello World!'
echo "I am process # $$"
```

helloworldtemplated.sh:

```
#!/bin/bash
# My second script

cat <<EOF
Hello World!
I am process # $$
EOF
```

©DENNIS SIMPSON 2013-2022

Script Structure

- Linux scripts are free-form with one exception
- The first line identifies the script as a script (magic number #!)
- The first line specifies how to run the command interpreter for the script
- The remainder of the script can be anything valid for the command interpreter
- #! is sometimes called shebang by the same fools who call **vi** vie

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

```
#!/usr/bin/env bash
```

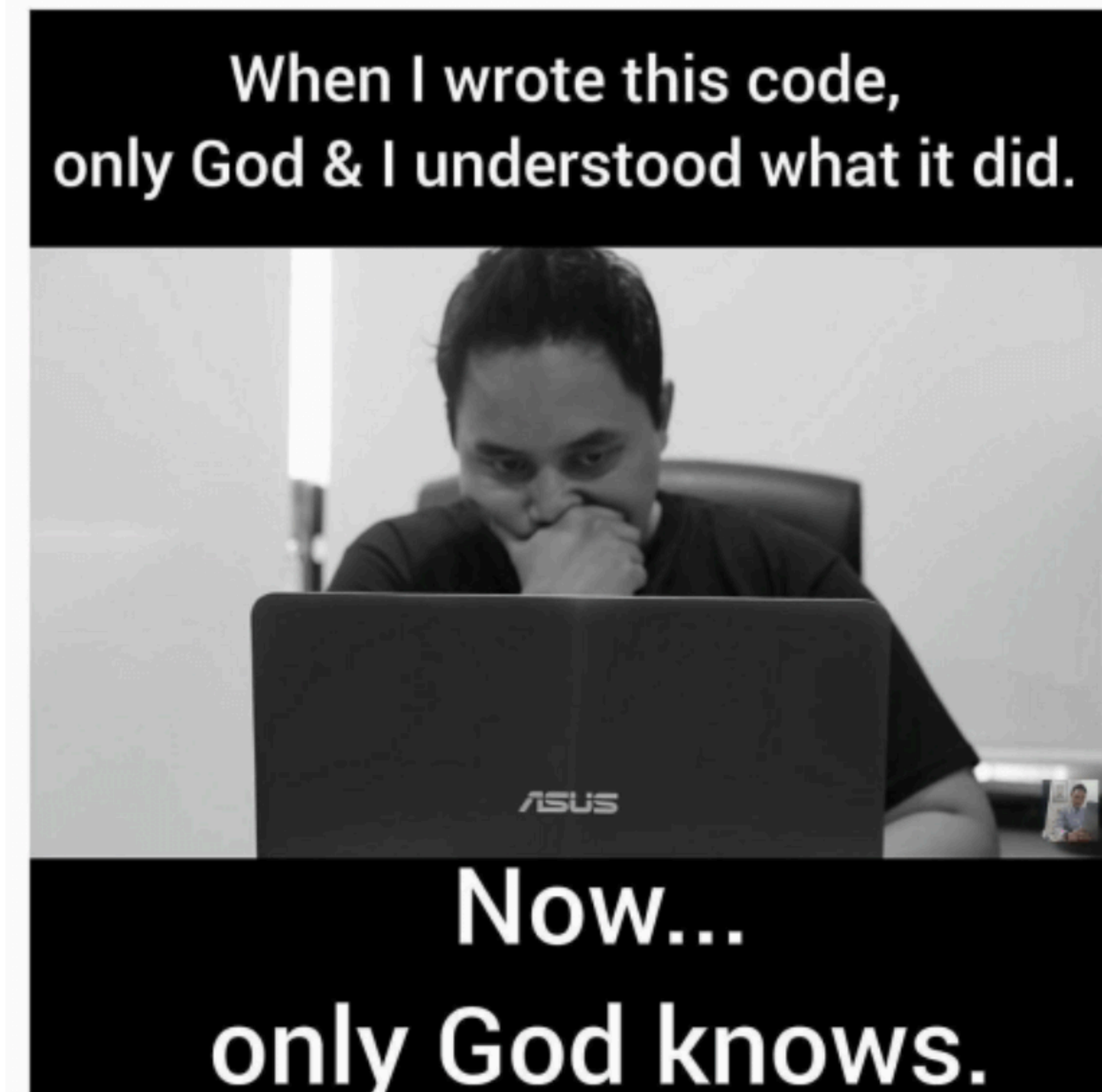
```
#!/bin/bash -x
```

```
#!/path/to/interpreter –option1 –option2 …
```

©DENNIS SIMPSON 2013-2022

Comments

- A comment is any text beginning with #
- They provide useful information about the script




```
# This is a comment
# Comments are ignored by the interpreter
echo "Hello World!" # this is a comment on the same line as a command
```

©DENNIS SIMPSON 2013-2022

Common Comment Use

- It can be very helpful to put some comments at the start of a script describing the script's purpose(s), inputs, and outputs
- Use comments to explain uncommon or difficult to read commands
- Comments can also be used to mark sections of a script

helloworldugly.sh:

```
#!/bin/bash
# helloworldugly.sh - an exercise in obfuscation
# This script displays the string “Hello World!”
#   and then displays its PID

# Function Definitions
function output-string { echo "$*"; }

# Main Script Body
# This is a silly way of creating the output text
# We start with similar text and stream edit it in a pipeline
# This is a trivial form of code obfuscation
# This version might require installing rot13 first
which rot13 >/dev/null || sudo apt install rot13
output-string $(rot13 <<< "uryo jbyq" |
   sed -e "s/b/o/g" -e "s/l/ll/" -e "s/ol/orl/" |
   tr "h" "H"|tr "w" "W"|
   awk '{print $1 "\x20" $2 "\41"}')
bc <<< "(($$ * 4 - 24)/2 + 12)/2" |
   sed 's/^/I am process # /'
```

©DENNIS SIMPSON 2013-2022

Script Storage

- In order to run a script, the shell must be able to find the script file
- bash uses the PATH variable to locate commands which are not built-in to the bash program itself
- Scripts are often stored in a directory associated with their purpose
  - Personal use scripts are often stored in ~/bin
  - Ubuntu and many other Linux distros have ~/bin in the default PATH for normal users
- Any script storage directory can be added to your shell command path by changing the content of your PATH variable (it holds a colon-delimited list of directories to look in for commands)
  - Be sure to add this to your bash startup file (typically ~/.bashrc) if you want it to be there every time you login - this example uses the script storage directory we are using in our labs

```
PATH=$PATH:~/COMP2101/bash
```

©DENNIS SIMPSON 2013-2022

Insufficient Data

Combining Commands

- Going beyond simple commands requires thinking about how commands can be combined
- A command which summarizes data might require the data to come from another command
- The output of a command may have extra information which is unwanted and must be filtered
- Output often requires context to be meaningful, labelling matters

©DENNIS SIMPSON 2013-2022

Command Pipeline

- A command pipeline is a sequence of commands separated by the | character
- The | character causes the output of the command on the left of the | to be connected to the input of the command on the right
- This allows us to run commands that work with the data from other commands without having to save that data first
- Every pipeline implements the following possible sequence of activity:
  - Gather data | Filter data | Manipulate data | Summarize or Format data

```
ls | wc -l
ps -eo user --no-headers | sort | uniq -c
ip a s eno1 | grep -w inet | awk '{print $2}'
```

©DENNIS SIMPSON 2013-2022

Gathering Data

- Any command that produces output can be described as a command to gather data
- The echo command gathers whatever is on the command line after the command and produces that evaluated text as output - e.g. echo "You are using the computer named $(hostname), $USER"
- The cat command reads all data from a data source such as a file and produces it as output
- The find command can be used to produce a wealth of information about the files on a computer and produce a wide variety of output about what is found
- The read command can be used to ask for data from a user interactively and produces a variable as its output

©DENNIS SIMPSON 2013-2022

Filtering Commands

- Commands that produce data may produce more data than is desired or needed
- That output can be sent to a file, or through a pipe to a command for immediate processing
- If part of the processing includes deciding what data to keep and what to discard, that is called filtering
- Filtering commands examine data and apply rules to decide whether to keep the data or discard it, and they can send the filtered data to a file, or through a pipe to another command for additional immediate processing
- There are many commands that can use rules to filter data - grep is a very common one that can filter data using both pattern matching and command options to make decisions about what data to keep and what to discard

```
grep root /etc/passwd
grep sudo /var/log/auth.log
find ~ | grep -i lostfile
ip a | grep -w inet
ip r | grep -v default
```

©DENNIS SIMPSON 2013-2022

Manipulating Data

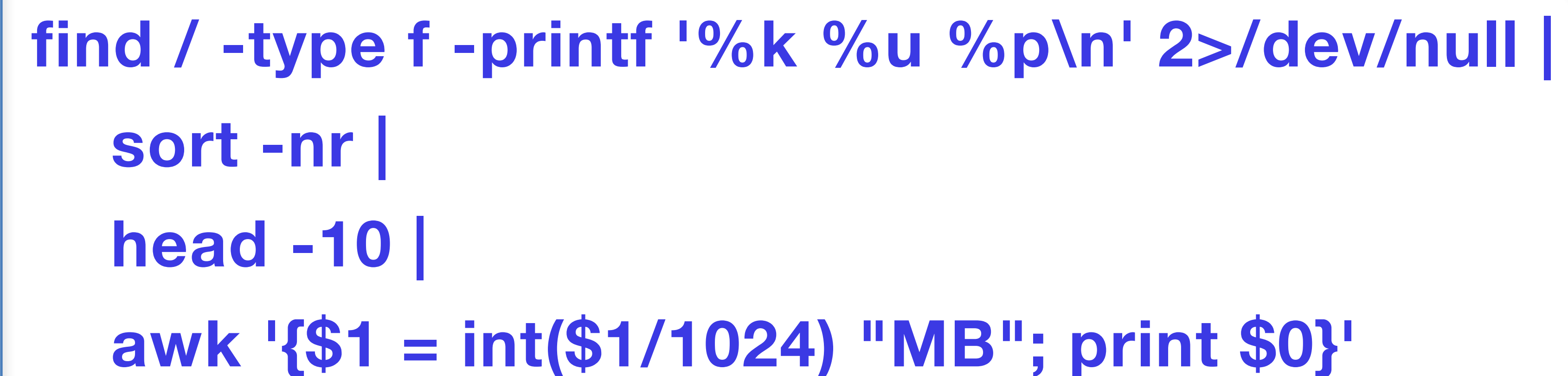
- Manipulating data involves making changes or edits to data based on rules
- You might want to change all numbers from bytes to megabytes in a text file
- You might want to encrypt or decrypt data
- You might want to reorder the data to some useful sequence such as biggest to smallest, or first to last, or alphabetically sorted
- Many of the commands in a computer exist to manipulate data and there are thousands of them
- Examples might include sort, tac, sed, awk, etc.
- Manipulation does not just mean reformatting - it can be summarizing or extrapolating as well using commands such as wc, awk, uniq, bc, printf, etc.

©DENNIS SIMPSON 2013-2022

Long Pipelines

- When building pipelines to process data, it is not uncommon for command lines to grow rather long
- For readability, long pipelines can be entered as multiple lines of text by splitting the line after the pipe symbols
- When using continuation lines like this, it is good practice to indent the continuation lines to make it clear to the reader that they are continuation lines - most scripting shells allow this

```
find / -type f -printf '%k %u %p\n' 2>/dev/null| sort -nr | head -10|awk '{$1 = int($1/1024) "MB"; print $0}'
```



```
find / -type f -printf '%k %u %p\n' 2>/dev/null |
    sort -nr |
    head -10 |
    awk '{$1 = int($1/1024) "MB"; print $0}'
```

©DENNIS SIMPSON 2013-2022

Labelling Output

- Labelling output makes it easier to read, reduces interpretation errors, and makes the results of your commands more meaningful
- A label might be for a specific data item such as a number or name
- You might use a set of labels such as the headers for a table of output
- echo is often used to print out labels in scripts

```
6

enp0s2

down

sda2 27GB    12GB  /
sdb2 200GB 172GB /data
```

vs

```
Online User Count: 6

Primary Network Interface: enp0s2

Webcam privacy cover position: down

Partition  Size    Free    Mount Point
sda2       27GB    12GB    /
sdb2       200GB   172GB   /data
```

©DENNIS SIMPSON 2013-2022

How Scripts End

- Scripts end when they encounter a fatal bash error, or the exit command, or run out of commands
- When a user enters the name of a script as a command for bash to run, bash runs that script in a new process which is a child of the current bash process
- If a script ends because it ran out of commands to run, **the script is considered to have succeeded if the last command it ran succeeded**
- If a script ends because of a fatal bash error (like if the script tries to divide by zero), the script is considered to have failed
- The success or failure of individual commands in the script are not relevant to the question of whether the script failed
- A good script will include logic to verify proper outcomes, and will use the exit command to end the script early if there is a reason to do that

```
#!/bin/bash
# this script would end as failed

myvar=$((1/0))
```

```
#!/bin/bash
# this script would end as successful

myvar=$((1/0))
echo that did not work
```

```
#!/bin/bash
# this script would end as successful, without
# printing hello world

exit
echo hello world
```

©DENNIS SIMPSON 2013-2022

Debugging The Command Line

- You can watch how bash does command line evaluation if you use set -x
- Don't forget to turn it off with set +x if you use this
- You can run scripts using bash -x scriptfilename to see all command lines in the script get evaluated before they execute
- You can put #!/bin/bash -x as the first line of your script to make it always run in debug mode

©DENNIS SIMPSON 2013-2022

Lab 1 - Beginner's bash

- first scripts
- bash runtime environment
- first challenge script

©DENNIS SIMPSON 2013-2022