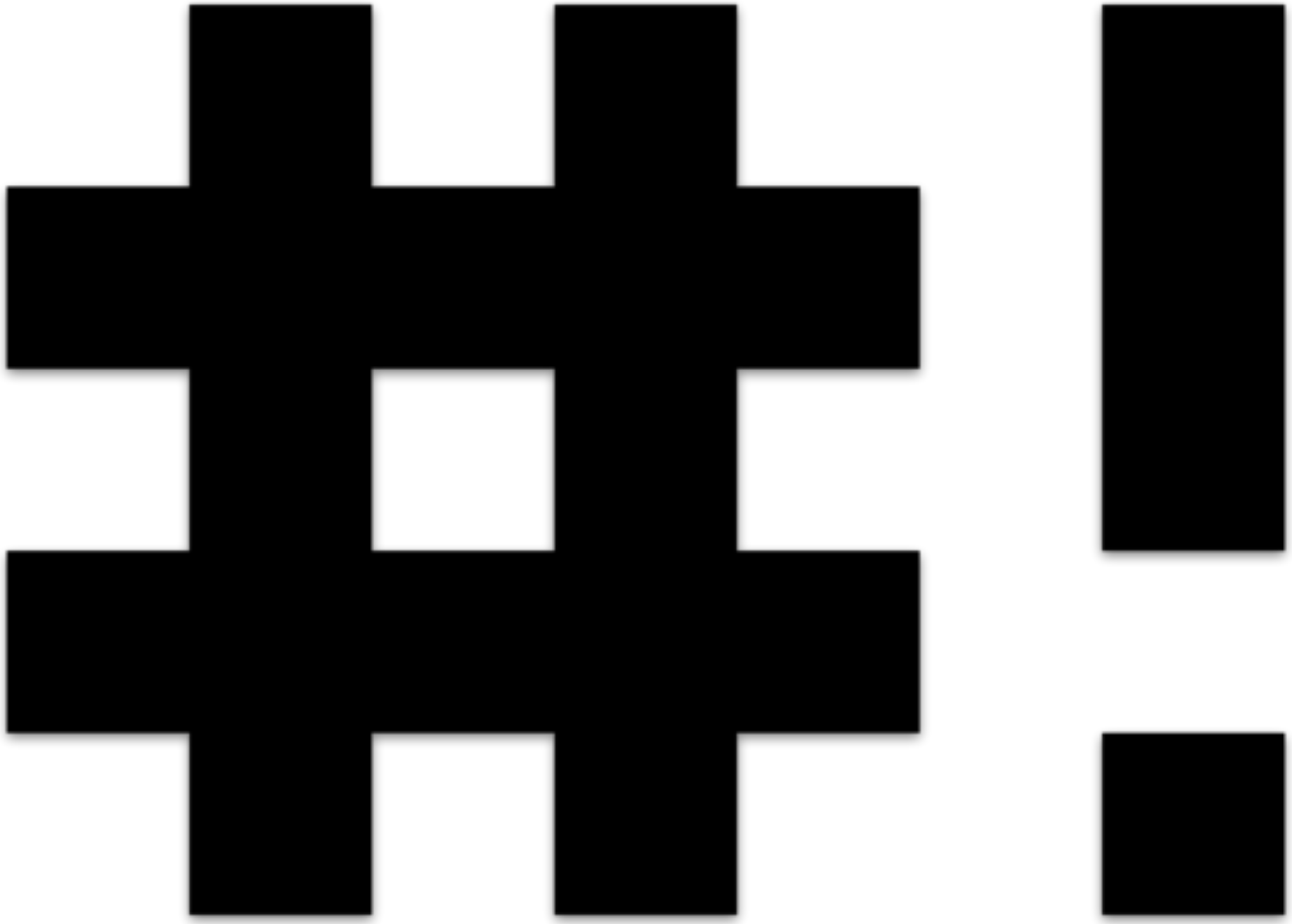


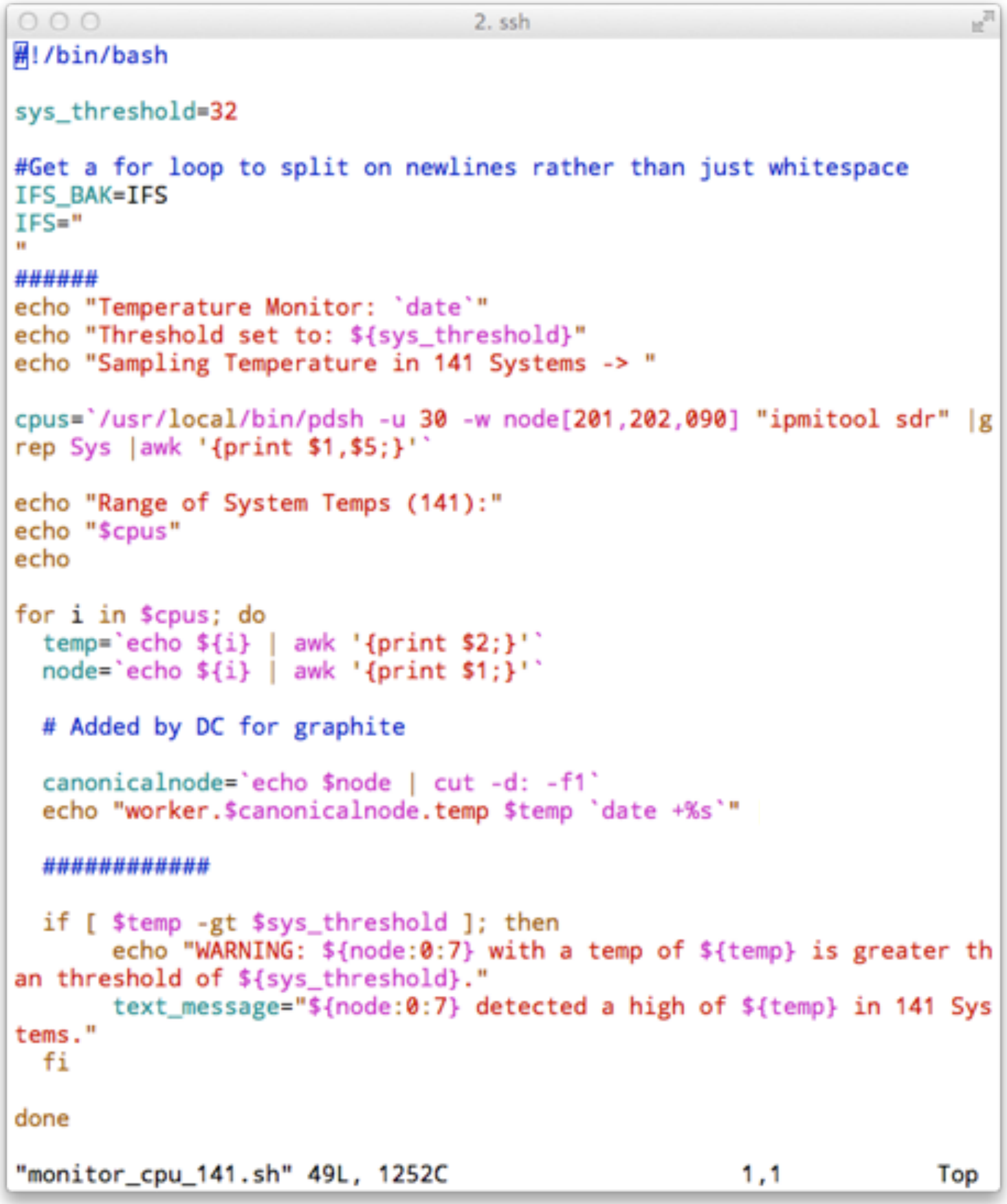
Bash Scripting

Dr. Gareth Roy (x6439)
gareth.roy@glasgow.ac.uk

- A simple script
- Tests (if)
- Ranges & Lists
- Loops (for, while)
- Special Vars

Shell Scripts

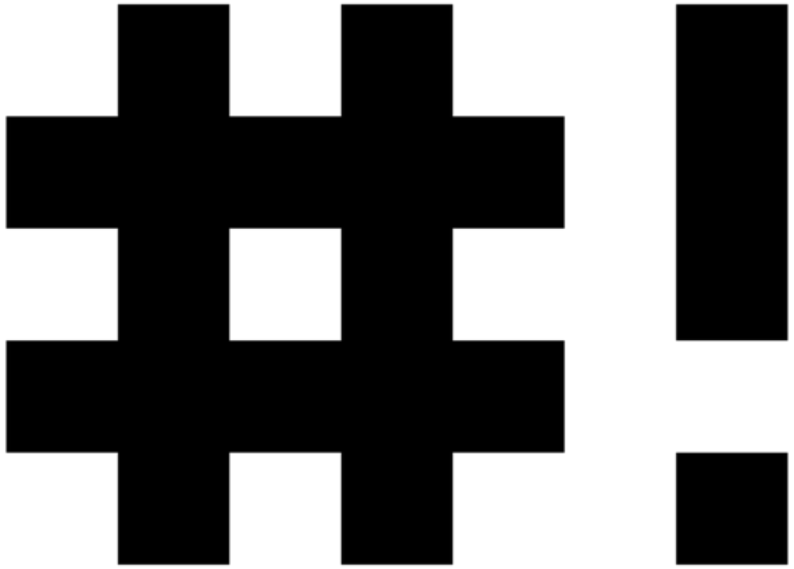
- Shell and script programming are useful tools in a physicist's armoury.
- Scripting allows the automation of easy, repetitive tasks.
- Example tasks include: checking webserver logs, backing up data to remote disk, creating directory structures or checking the workings of an automated experiment.
- Shell scripts are deceptively simple, but can be powerful when combined with the many tools found on a Linux platform
- Other scripting languages you may come across include perl, python and ruby. In the rest of this course we will focus on BASH.



```
#!/bin/bash

sys_threshold=32

#Get a for loop to split on newlines rather than just whitespace
IFS_BAK=IFS
IFS="
"
######
echo "Temperature Monitor: `date`"
echo "Threshold set to: ${sys_threshold}"
echo "Sampling Temperature in 141 Systems -> "

cpus=`/usr/local/bin/pdsh -u 30 -w node[201,202,090] "ipmitool sdr" |g
rep Sys |awk '{print $1,$5;}'`

echo "Range of System Temps (141):"
echo "$cpus"
echo

for i in $cpus; do
  temp=`echo ${i} | awk '{print $2;}'`
  node=`echo ${i} | awk '{print $1;}'`

  # Added by DC for graphite

  canonicalnode=`echo $node | cut -d: -f1`
  echo "worker.$canonicalnode.temp $temp `date +%s`"

  ############

  if [ $temp -gt $sys_threshold ]; then
      echo "WARNING: ${node:0:7} with a temp of ${temp} is greater th
an threshold of ${sys_threshold}."
      text_message="${node:0:7} detected a high of ${temp} in 141 Sys
tems."
  fi

done

"monitor_cpu_141.sh" 49L, 1252C                               1,1          Top
```

Shell Scripts

- In this course we're only going to cover the basics.
- If you want to know more there are some great resources on the web.
- There are also a lot of gotchas so things like stackoverflow.com can also be helpful.
- **Just remember, always cite sources if using things directly from online guides.**



Scripts in this Lecture:/students/p2t-16/share/LinuxLab04/scripts

Beginners Guide

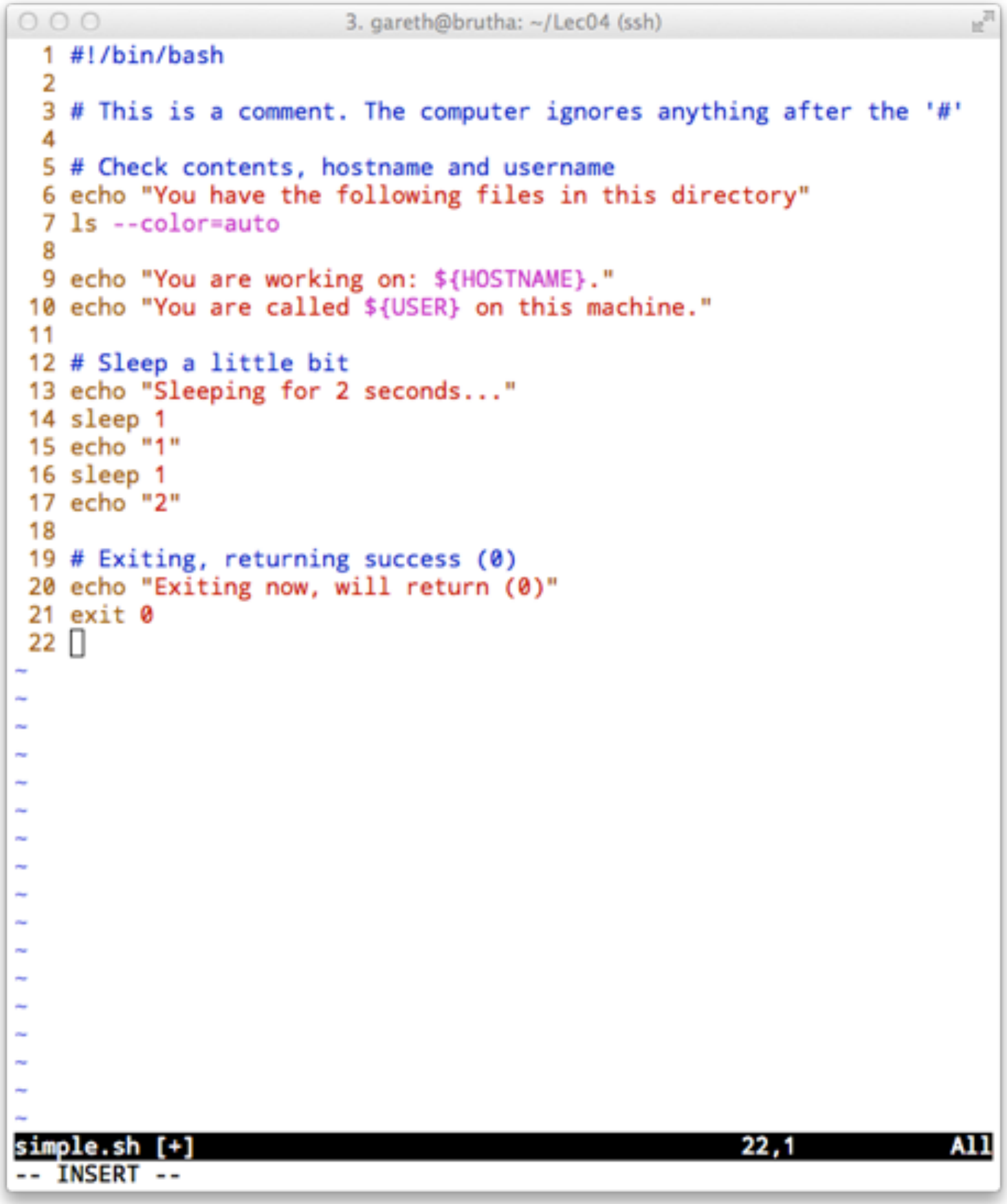
http://tldp.org/LDP/Bash-Beginners-Guide/html/index.html

Advanced Scripting

http://tldp.org/LDP/abs/html/index.html

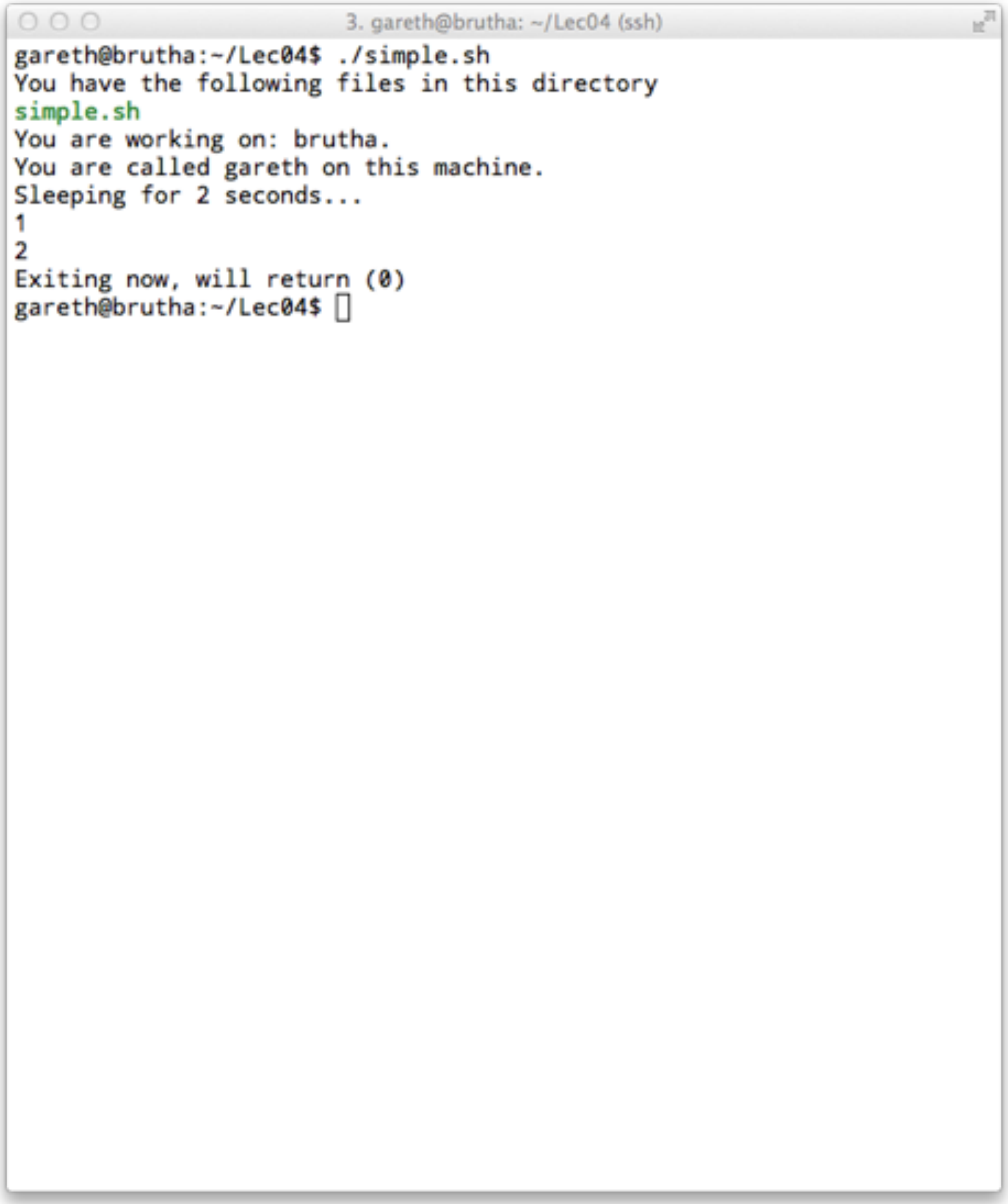
- A simple script
- Tests (if)
- Ranges & Lists
- Loops (for, while)
- Special Vars

A Simple Script



```
#!/bin/bash

# This is a comment. The computer ignores anything after the '#'

# Check contents, hostname and username
echo "You have the following files in this directory"
ls --color=auto

echo "You are working on: ${HOSTNAME}."
echo "You are called ${USER} on this machine."

# Sleep a little bit
echo "Sleeping for 2 seconds..."
sleep 1
echo "1"
sleep 1
echo "2"

# Exiting, returning success (0)
echo "Exiting now, will return (0)"
exit 0

```

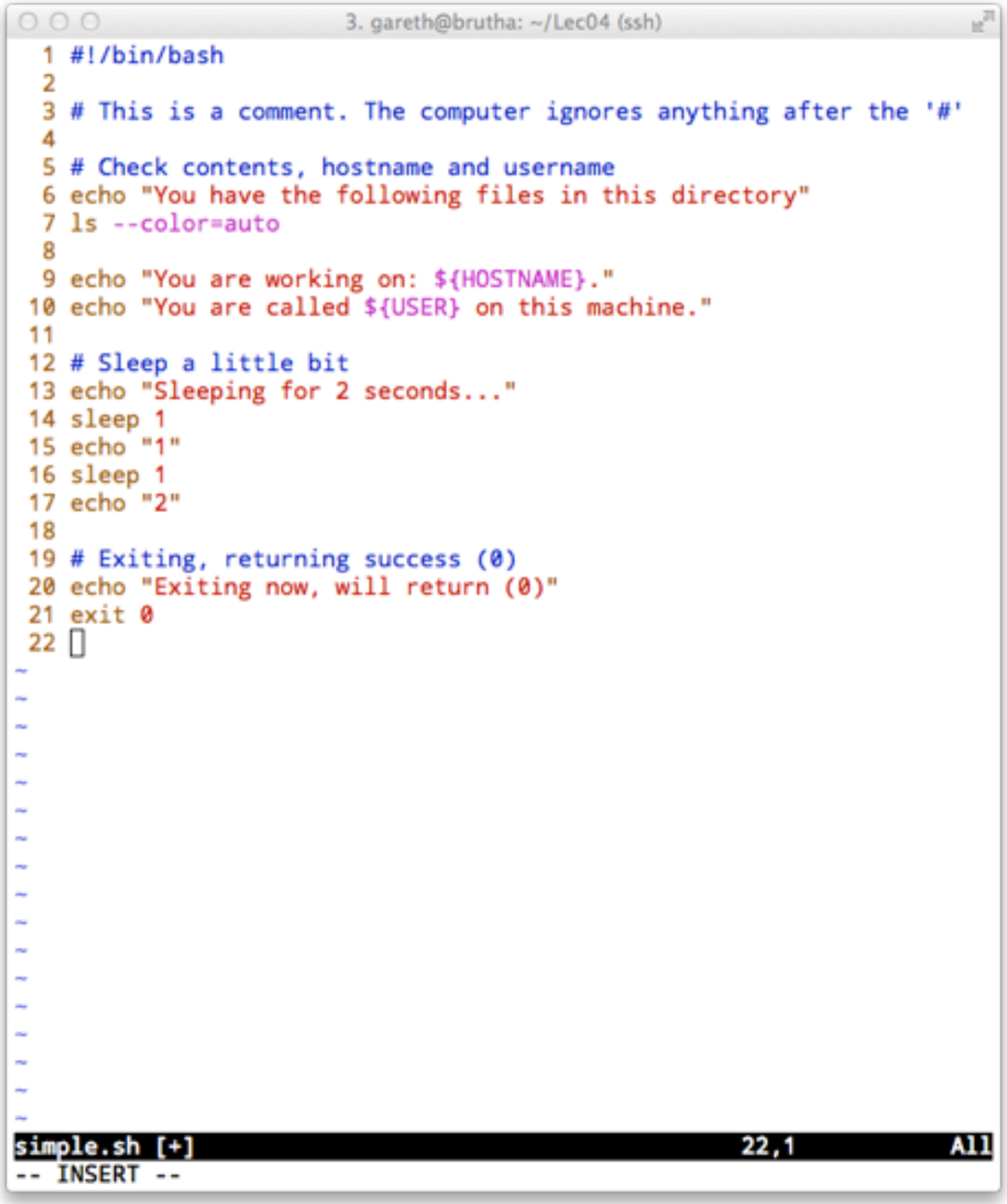
```
simple.sh [+]                               22,1            All
-- INSERT --
```



```
gareth@brutha:~/Lec04$ ./simple.sh
You have the following files in this directory
simple.sh
You are working on: brutha.
You are called gareth on this machine.
Sleeping for 2 seconds...
1
2
Exiting now, will return (0)
gareth@brutha:~/Lec04$ 
```

A Simple Script

- A simple script is just a text file with a series of commands.
- The usual file extension used to identify a shell script is ***.sh**
- **#!** is an interpreter directive. It instructs the shell to run the command specified with the current file.
- It's often referred to as the "**shebang**", "**hash-bang**" or "**hash-pling**".
- **#!/bin/bash** translates to:
    - **/bin/bash $PWD/simple.sh**
- All bash scripts should start with this line.



```bash
#!/bin/bash

# This is a comment. The computer ignores anything after the '#'

# Check contents, hostname and username
echo "You have the following files in this directory"
ls --color=auto

echo "You are working on: ${HOSTNAME}."
echo "You are called ${USER} on this machine."

# Sleep a little bit
echo "Sleeping for 2 seconds..."
sleep 1
echo "1"
sleep 1
echo "2"

# Exiting, returning success (0)
echo "Exiting now, will return (0)"
exit 0

```

```
simple.sh [+]                                   22,1            All
-- INSERT --
```

A Simple Script

- We can make our scripts more understandable by inserting comments

- The ‘**#**’ character at the beginning of the line denotes it is a comment and these are ignored by the interpreter.

- Our simple script:

  - uses **echo** to output to the screen

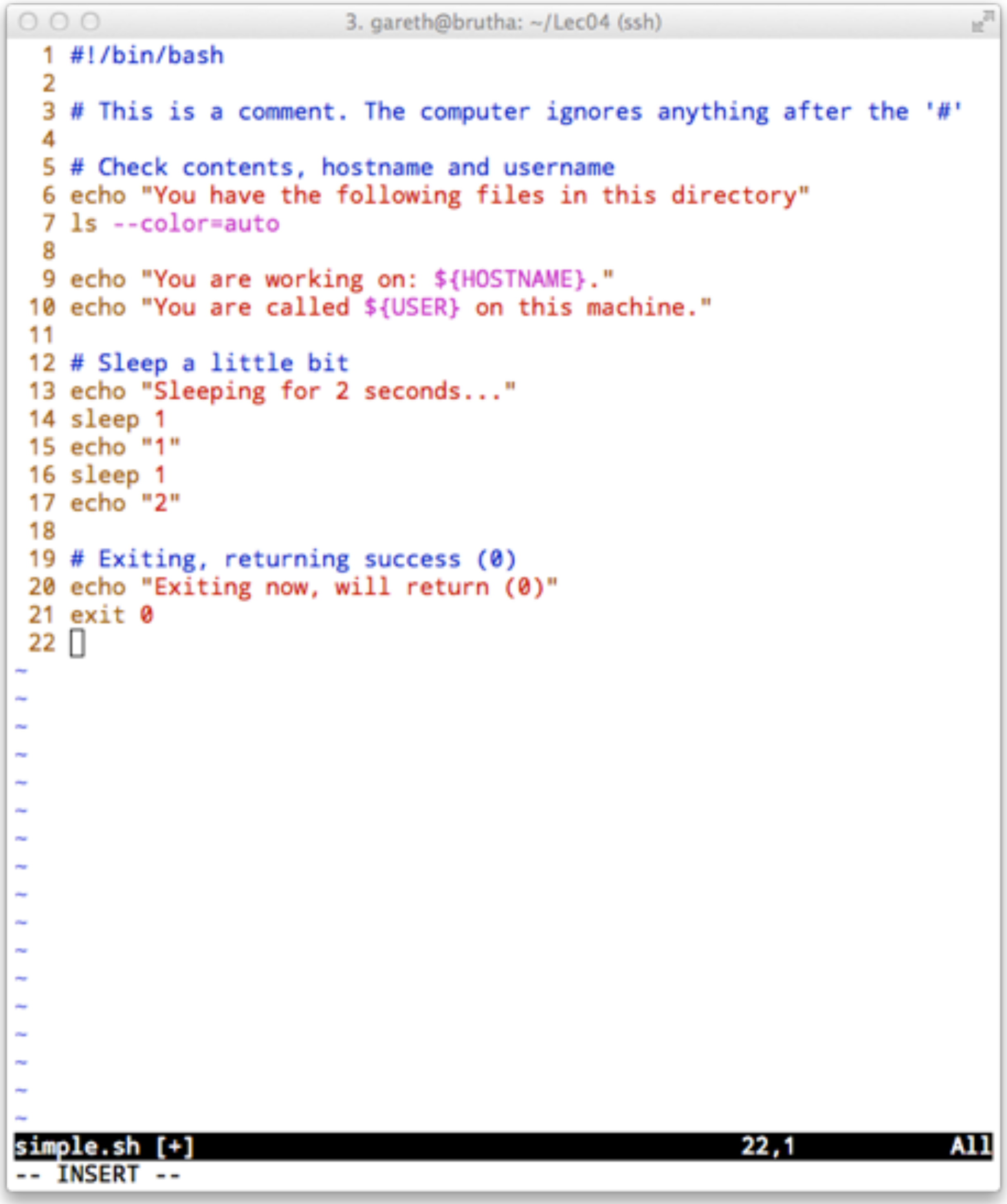
  - executes the **ls** command

  - displays the contents of some environment variables (**HOSTNAME** and **USER**)

  - uses **sleep** to pause for 2 seconds

  - uses **exit** to return the value 0 (denoting successful completion).

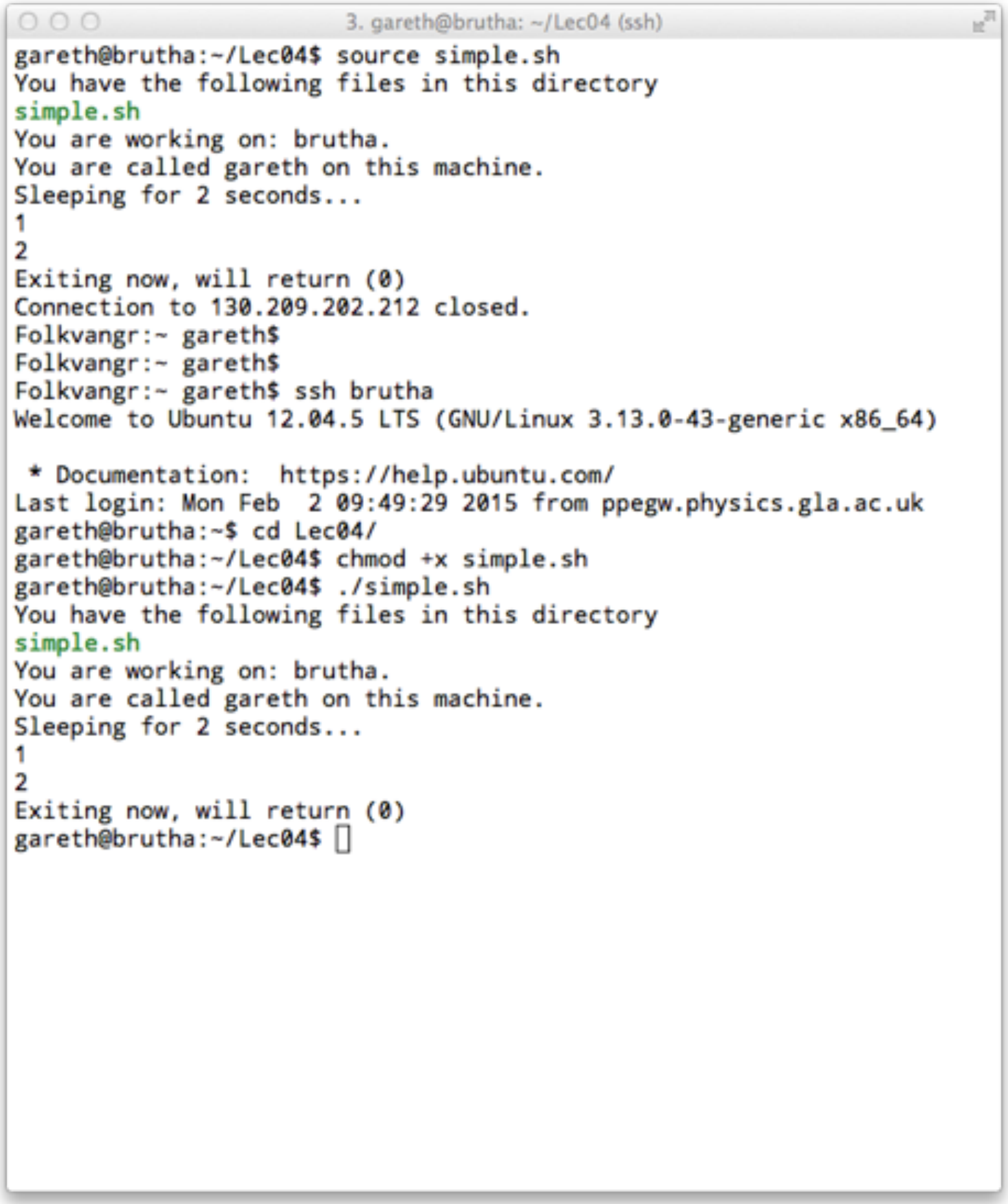
- **exit** terminates the program and returns a numeric value to the calling process. By default 0 is success and any numeric value is failure.



```
#!/bin/bash

# This is a comment. The computer ignores anything after the '#'

# Check contents, hostname and username
echo "You have the following files in this directory"
ls --color=auto

echo "You are working on: ${HOSTNAME}."
echo "You are called ${USER} on this machine."

# Sleep a little bit
echo "Sleeping for 2 seconds..."
sleep 1
echo "1"
sleep 1
echo "2"

# Exiting, returning success (0)
echo "Exiting now, will return (0)"
exit 0

simple.sh [+]                                         22,1           All
-- INSERT --
```

Running the script

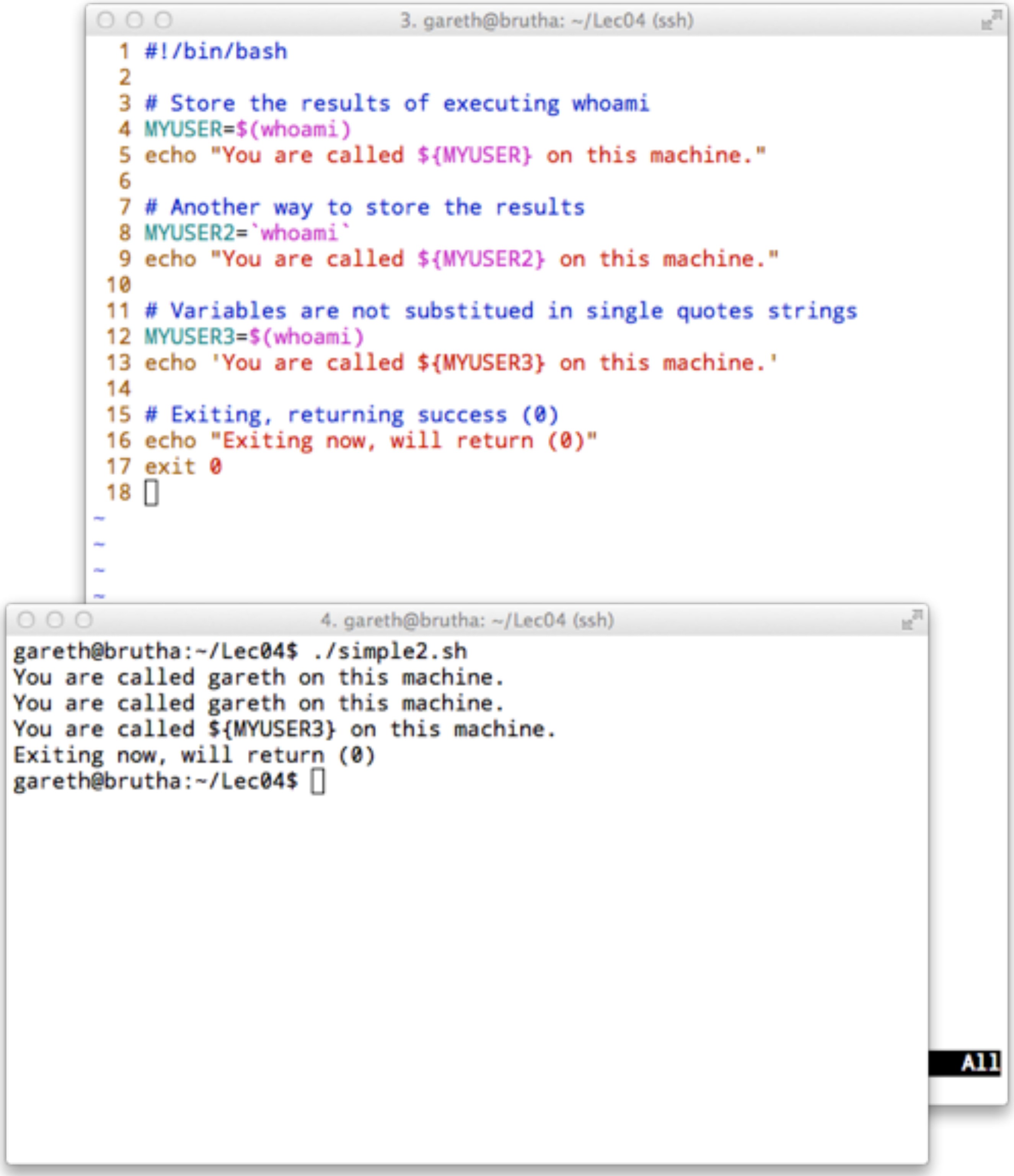
- We can run the script in two ways:
  - **source simple.sh**
  - **chmod +x simple.sh; ./simple.sh**
- **source** runs the commands in simple.sh one after another in the current shell.
- Marking the script as executable and running it via **./simple.sh** forks a new process.
- Running a script using **source** can be dangerous as it can overwrite variables in your current session.
- It will also exit you current session if it encounters an **exit** statement.



```
gareth@brutha:~/Lec04$ source simple.sh
You have the following files in this directory
simple.sh
You are working on: brutha.
You are called gareth on this machine.
Sleeping for 2 seconds...
1
2
Exiting now, will return (0)
Connection to 130.209.202.212 closed.
Folkvangr:~ gareth$
Folkvangr:~ gareth$
Folkvangr:~ gareth$ ssh brutha
Welcome to Ubuntu 12.04.5 LTS (GNU/Linux 3.13.0-43-generic x86_64)

 * Documentation:  https://help.ubuntu.com/
Last login: Mon Feb  2 09:49:29 2015 from ppegw.physics.gla.ac.uk
gareth@brutha:~$ cd Lec04/
gareth@brutha:~/Lec04$ chmod +x simple.sh
gareth@brutha:~/Lec04$ ./simple.sh
You have the following files in this directory
simple.sh
You are working on: brutha.
You are called gareth on this machine.
Sleeping for 2 seconds...
1
2
Exiting now, will return (0)
gareth@brutha:~/Lec04$ 
```

An aside on Variables

- We can assign the output of a command to a variable.
- There are two ways to do this:
  - **MYUSER=$(whoami)**
  - **MYUSER=`whoami`**
- There are two types of strings in bash
  - “” - double quoted
  - ‘’ - single quoted
- In double quotes strings variable names are replaced with their value.
- In single quoted strings variable substitution does not take place.


3. gareth@brutha: ~/Lec04 (ssh)
All
4. gareth@brutha: ~/Lec04 (ssh)

```
#!/bin/bash

# Store the results of executing whoami
MYUSER=$(whoami)
echo "You are called ${MYUSER} on this machine."

# Another way to store the results
MYUSER2=`whoami`
echo "You are called ${MYUSER2} on this machine."

# Variables are not substitued in single quotes strings
MYUSER3=$(whoami)
echo 'You are called ${MYUSER3} on this machine.'

# Exiting, returning success (0)
echo "Exiting now, will return (0)"
exit 0

```

```
gareth@brutha:~/Lec04$ ./simple2.sh
You are called gareth on this machine.
You are called gareth on this machine.
You are called ${MYUSER3} on this machine.
Exiting now, will return (0)
gareth@brutha:~/Lec04$
```

- A simple script
- Tests (if)
- Ranges & Lists
- Loops (for, while)
- Special Vars

If, then, else

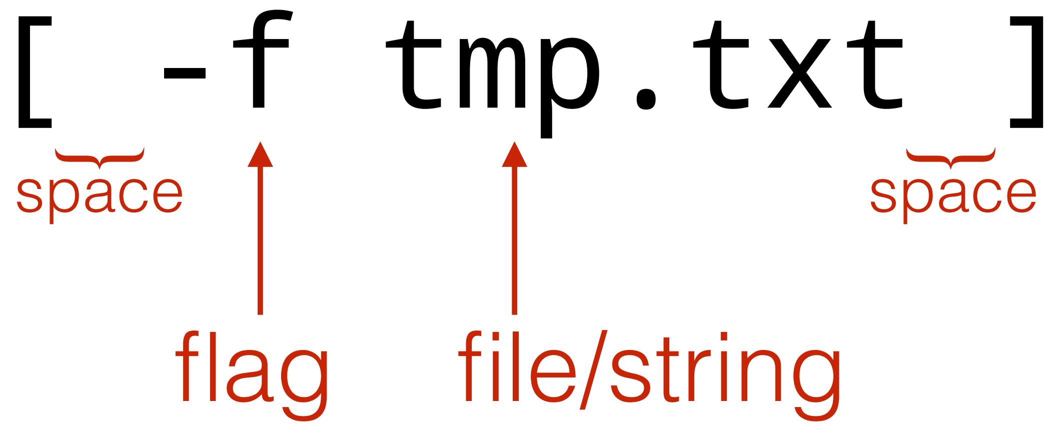
- **if** statements are similar to those found in C.
- They allow conditional execution of code, allowing decisions to be made about what to execute
- For instance **if** a file exists, **then** remove that file, **else** log an error.
- There are three general forms for an in **if** statement in Bash:
    - **if x then** y
    - **if x then** y **else** z
    - **if x then** y **elif a then** z

```
if [ conditional ]; then
      some command
fi
```

```
if [ conditional ]
then
      some command
else
      some other command
fi
```

```
if [ conditional ]
then
      some command
elif [ conditional ]
      some other command
else
      yet another command
fi
```

Bash Conditionals



```
[ -f tmp.txt ]
```

space

space

flag

file/string

String Comparision	Result
string 1 == string 2	True if the strings are equal
string 1 != string 2	True if the strings are different
-n string	True if the string is not null
-z string	True if the string is null

File Conditionals	Result
-d file	True if file is a directory
-e file	True if file exists
-f file	True if the file is regular
-r file	True if file is readable
-s file	True if file has nonzero size
-w file	True if file is writeable
-x file	True if file is executable

Arithmetic Comparision	Result
exp 1 -eq exp 2	True if both are equal
exp 1 -ne exp 2	True if both are different
exp 1 -gt exp 2	True if exp1 is greater than exp2
exp 1 -ge exp 2	True if exp1 is greater than or equal to exp2
exp 1 -lt exp 2	True if exp1 is less than exp2
exp 1 -le exp 2	True if exp1 is less than or equal to exp2
! exp	Invertes exp, true if exp is false. False if exp is true

If, then, else

- Here is an example script using if statements and conditionals.

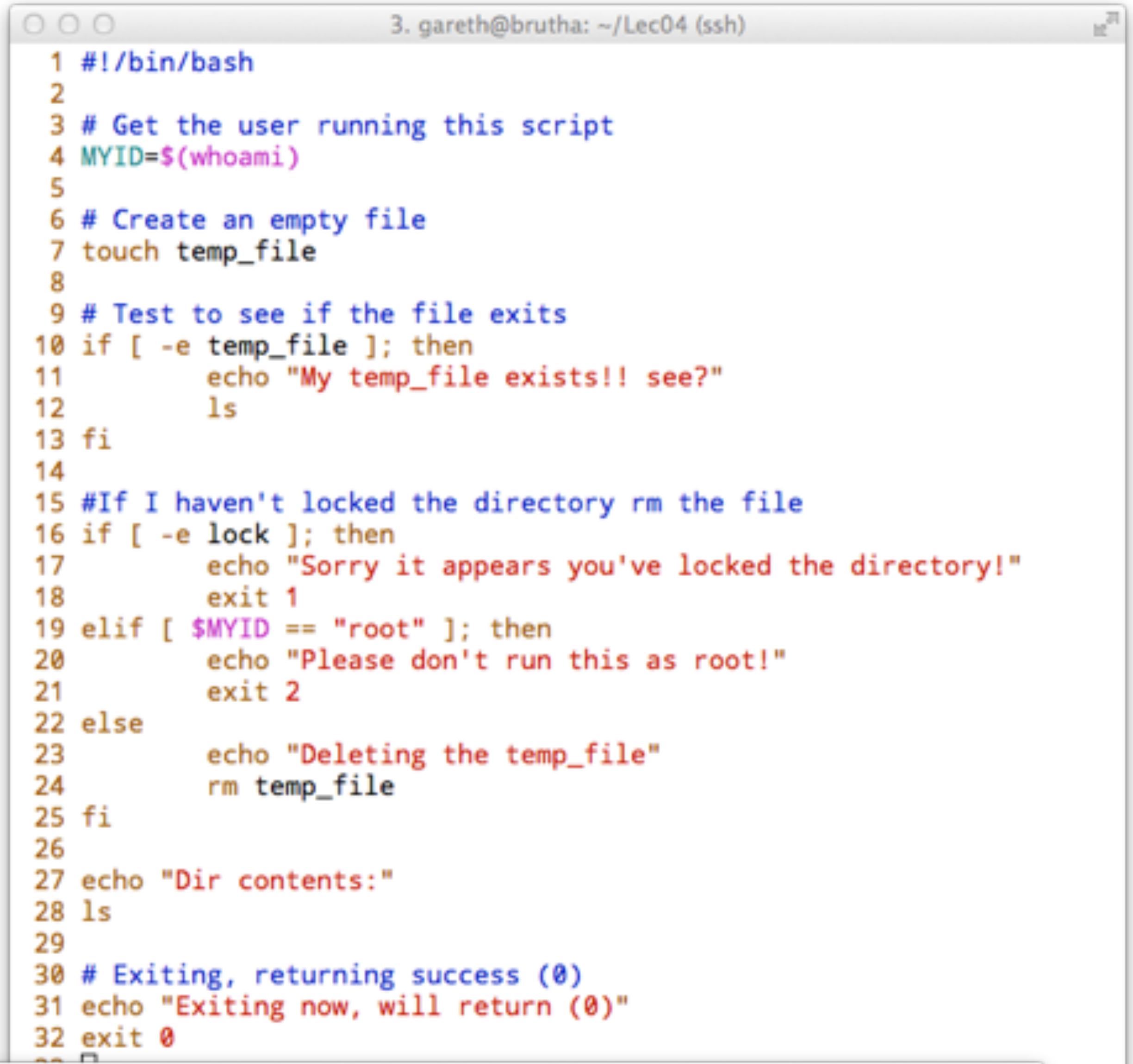
- The script runs **whoami**, and stores the value in a variable. It then creates an empty file called **temp_file**.

- If the file exists it writes a message and lists the directory.

- The script checks to see if a **lock** file is present, if it is it writes a message and exits with an error.

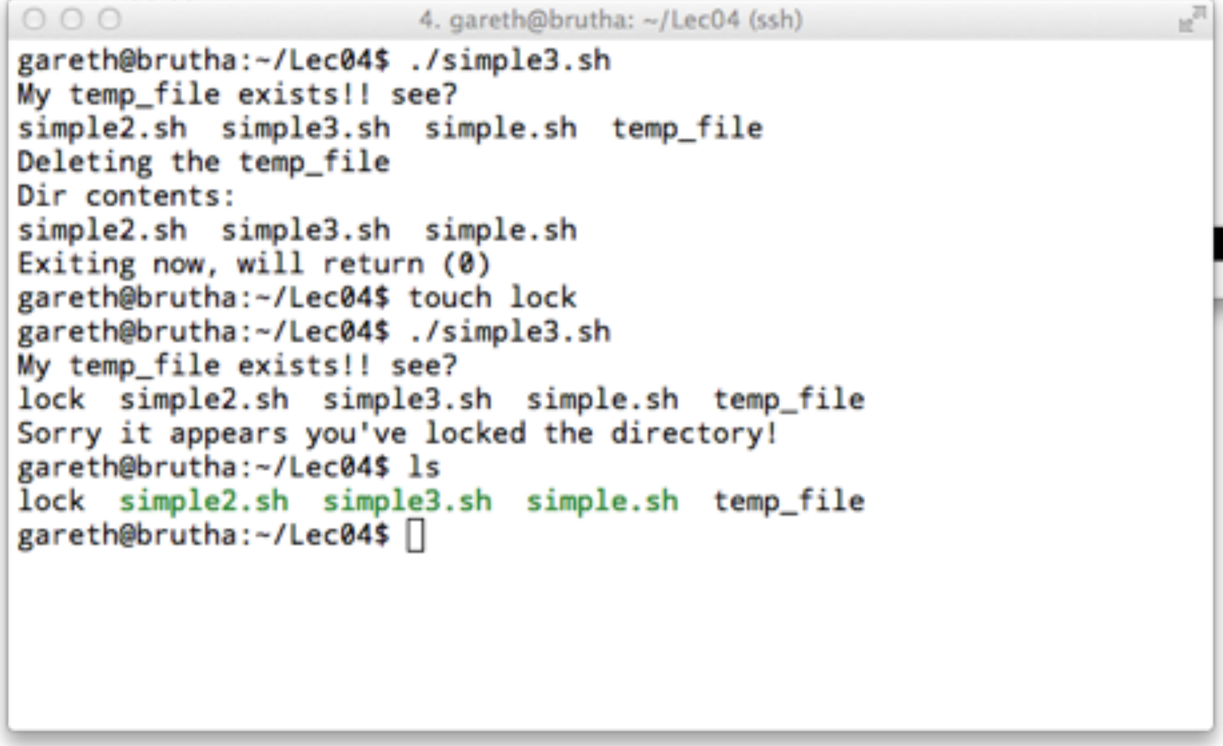
- If the lock file is not present it checks to make sure we are not the **root** user, and if not it removes **temp_file**.

- Finally it lists the directory contents and exits with a success



3. gareth@brutha: ~/Lec04 (ssh)

```
#!/bin/bash

# Get the user running this script
MYID=$(whoami)

# Create an empty file
touch temp_file

# Test to see if the file exits
if [ -e temp_file ]; then
        echo "My temp_file exists!! see?"
        ls
fi

#If I haven't locked the directory rm the file
if [ -e lock ]; then
        echo "Sorry it appears you've locked the directory!"
        exit 1
elif [ $MYID == "root" ]; then
        echo "Please don't run this as root!"
        exit 2
else
        echo "Deleting the temp_file"
        rm temp_file
fi

echo "Dir contents:"
ls

# Exiting, returning success (0)
echo "Exiting now, will return (0)"
exit 0
```



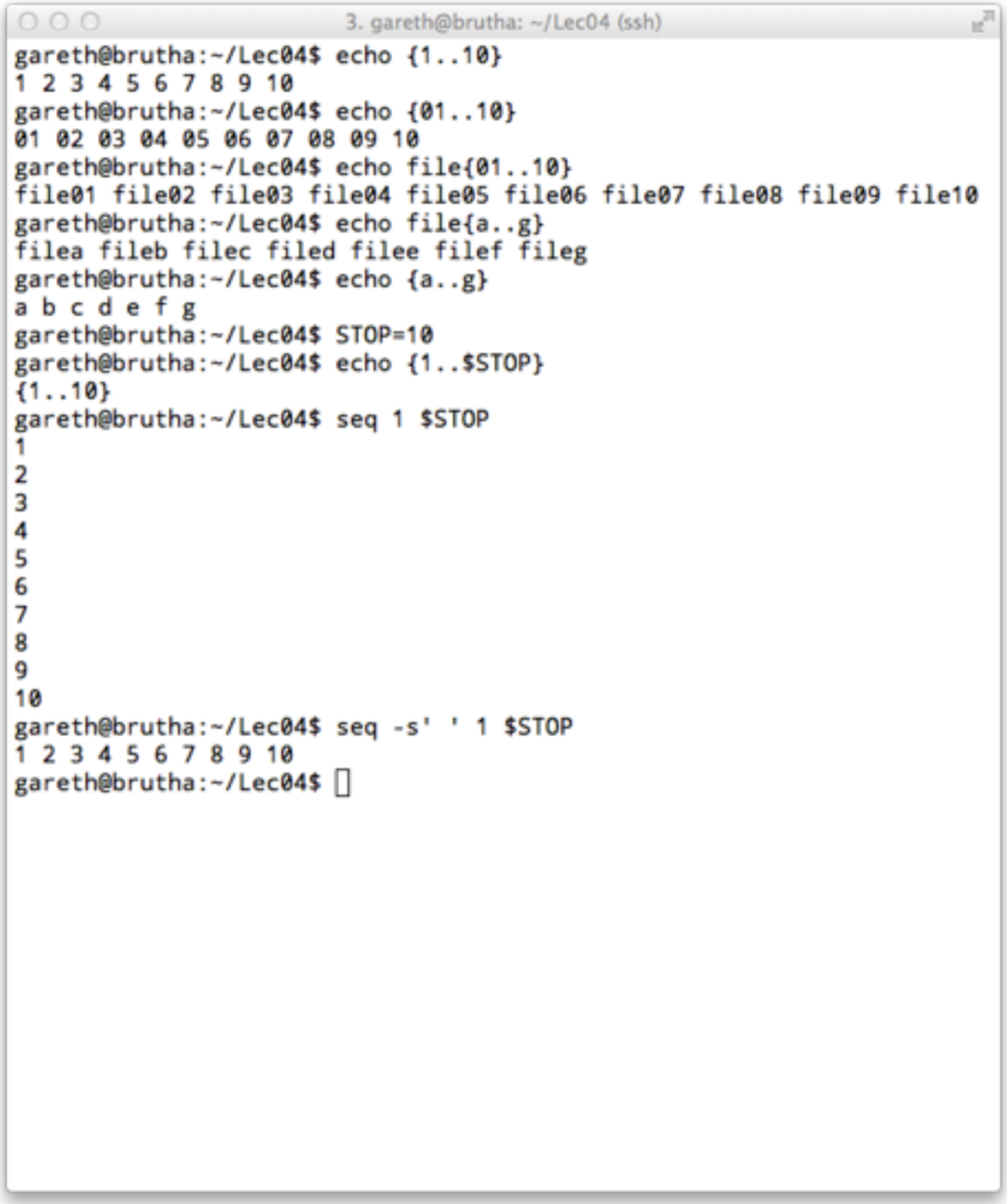
4. gareth@brutha: ~/Lec04 (ssh)

```
gareth@brutha:~/Lec04$ ./simple3.sh
My temp_file exists!! see?
simple2.sh  simple3.sh  simple.sh  temp_file
Deleting the temp_file
Dir contents:
simple2.sh  simple3.sh  simple.sh
Exiting now, will return (0)
gareth@brutha:~/Lec04$ touch lock
gareth@brutha:~/Lec04$ ./simple3.sh
My temp_file exists!! see?
lock  simple2.sh  simple3.sh  simple.sh  temp_file
Sorry it appears you've locked the directory!
gareth@brutha:~/Lec04$ ls
lock  simple2.sh  simple3.sh  simple.sh  temp_file
gareth@brutha:~/Lec04$ 
```

- A simple script
- Tests (if)
- Ranges & Lists
- Loops (for, while)
- Special Vars

Ranges

- We can specify numeric ranges in Bash using the **{start..stop}** notation.
- A range of 1 to 3 would be written as:
  - **{1..3}** => 1 2 3
  - **{01..03}** => 01 02 03
- Range can be characters as well as numbers:
  - **{a..c}** => a b c
  - **{A..C}** => A B C
- Unfortunately we cannot use variables in the above definition, which means we cannot change the range while running a script.
- For numerical ranges we can use a command called **seq**.
  - **STOP=20; seq 1 ${STOP}**
- We'll use ranges frequently while writing loops in bash.



```
gareth@brutha:~/Lec04$ echo {1..10}
1 2 3 4 5 6 7 8 9 10
gareth@brutha:~/Lec04$ echo {01..10}
01 02 03 04 05 06 07 08 09 10
gareth@brutha:~/Lec04$ echo file{01..10}
file01 file02 file03 file04 file05 file06 file07 file08 file09 file10
gareth@brutha:~/Lec04$ echo file{a..g}
filea fileb filec filed filee filef fileg
gareth@brutha:~/Lec04$ echo {a..g}
a b c d e f g
gareth@brutha:~/Lec04$ STOP=10
gareth@brutha:~/Lec04$ echo {1..$STOP}
{1..10}
gareth@brutha:~/Lec04$ seq 1 $STOP
1
2
3
4
5
6
7
8
9
10
gareth@brutha:~/Lec04$ seq -s' ' 1 $STOP
1 2 3 4 5 6 7 8 9 10
gareth@brutha:~/Lec04$ 
```

Lists

- Bash treats a group of space separated strings as a list.

- You can specify a list using brace expansion similar to ranges (be careful not to include spaces):

  - **{a,b,c,d}**

- Bash treats a double or single quoted string as a list if it contains spaces so:

  - **MYVAR="a b c d"**

  - **MYVAR='a b c d'**

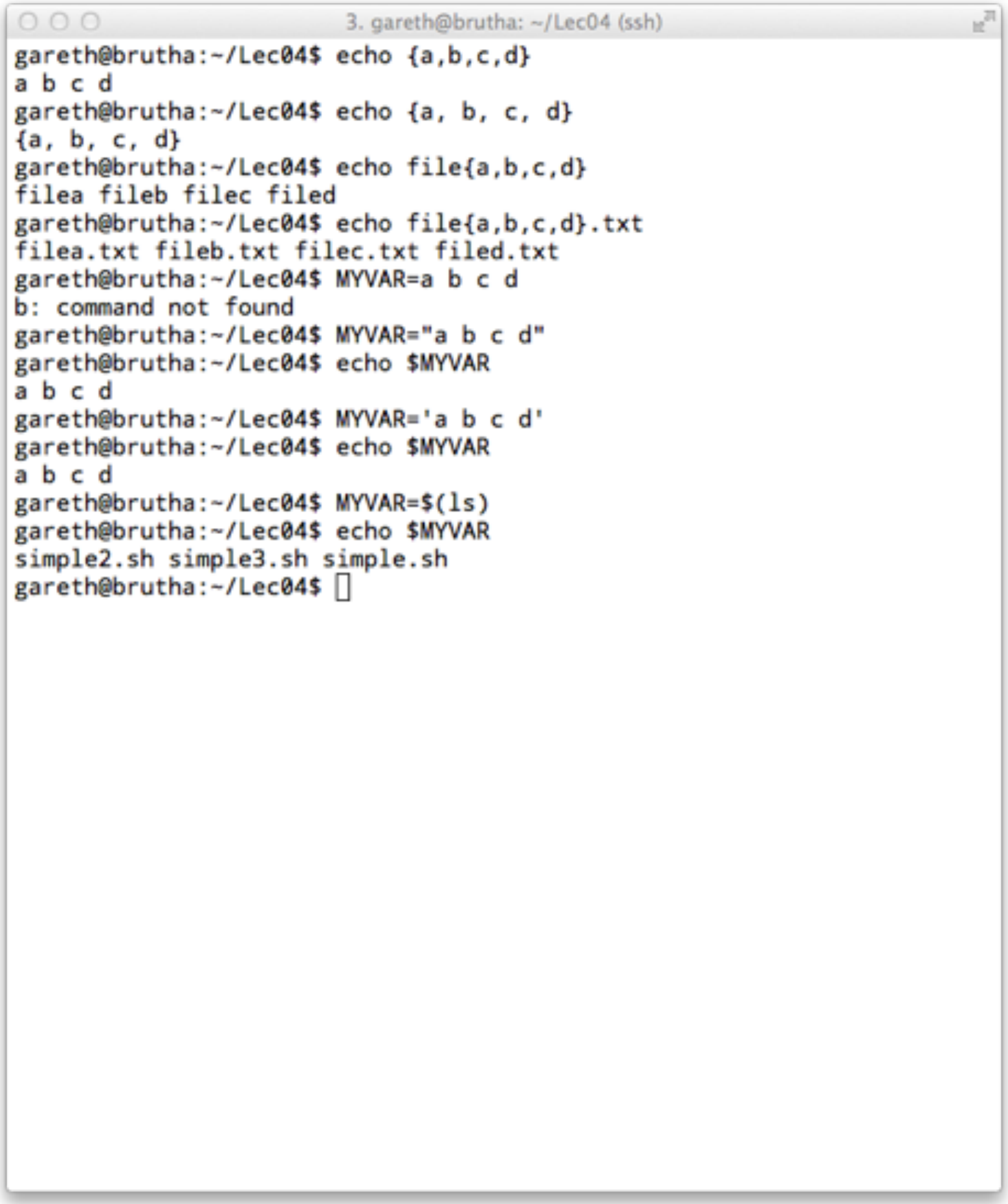
- This useful when using filesystem commands such as **ls**:

  - **MYVAR=$(ls)**

- When we come to loops we can also specify a list as below although it's usually better to wrap values in a variable:

  - **a b c d**

  - **"a" "b" "c" "d"**



```
gareth@brutha:~/Lec04$ echo {a,b,c,d}
a b c d
gareth@brutha:~/Lec04$ echo {a, b, c, d}
{a, b, c, d}
gareth@brutha:~/Lec04$ echo file{a,b,c,d}
filea fileb filec filed
gareth@brutha:~/Lec04$ echo file{a,b,c,d}.txt
filea.txt fileb.txt filec.txt filed.txt
gareth@brutha:~/Lec04$ MYVAR=a b c d
b: command not found
gareth@brutha:~/Lec04$ MYVAR="a b c d"
gareth@brutha:~/Lec04$ echo $MYVAR
a b c d
gareth@brutha:~/Lec04$ MYVAR='a b c d'
gareth@brutha:~/Lec04$ echo $MYVAR
a b c d
gareth@brutha:~/Lec04$ MYVAR=$(ls)
gareth@brutha:~/Lec04$ echo $MYVAR
simple2.sh simple3.sh simple.sh
gareth@brutha:~/Lec04$ 
```

Arrays

- Newer versions of Bash also have arrays, which can be accessed via indexes.

- To create an array you would do the following:

  - **MYARRAY=(a b c d)**

- Arrays are 0 indexed so below would return the second element of the array:

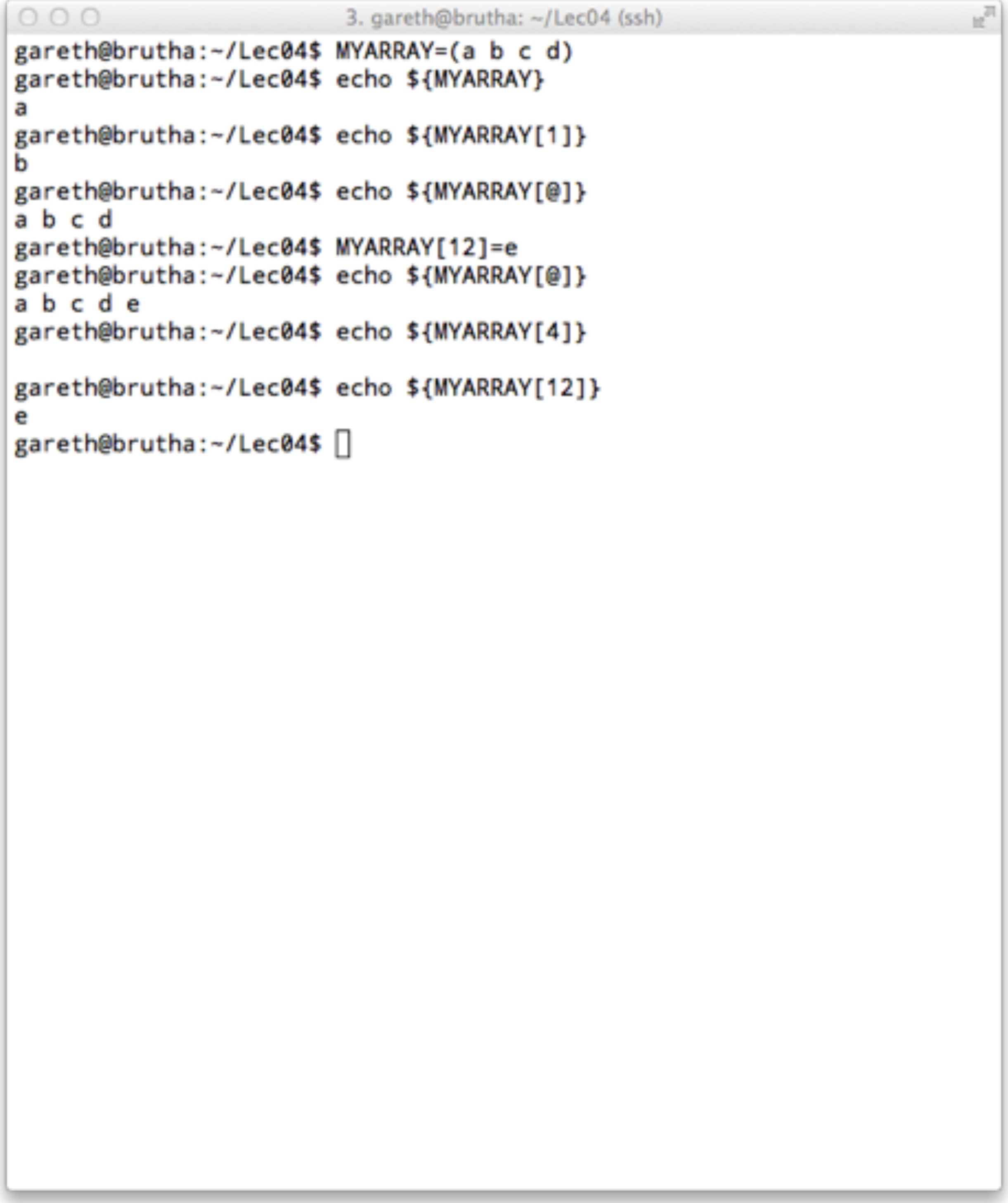
  - **echo ${MYARRAY[1]}**

- You can access all the elements of an array by passing the special ‘@‘ character as an index.

  - **echo ${MYARRY[@]}**

- Array indexes don’t need to be consecutive, so we could add an index at 12 by doing:

  - **MYARRAY[12]=e**



```
gareth@brutha:~/Lec04$ MYARRAY=(a b c d)
gareth@brutha:~/Lec04$ echo ${MYARRAY}
a
gareth@brutha:~/Lec04$ echo ${MYARRAY[1]}
b
gareth@brutha:~/Lec04$ echo ${MYARRAY[@]}
a b c d
gareth@brutha:~/Lec04$ MYARRAY[12]=e
gareth@brutha:~/Lec04$ echo ${MYARRAY[@]}
a b c d e
gareth@brutha:~/Lec04$ echo ${MYARRAY[4]}

gareth@brutha:~/Lec04$ echo ${MYARRAY[12]}
e
gareth@brutha:~/Lec04$
```

- A simple script
- Tests (if)
- Ranges & Lists
- Loops (for, while)
- Special Vars

For loops

- There are two types of for loops used in Bash scripts.
- A C-style three expression for loop, with an initialiser, condition and increment
- An iterator-style for loop, which take each item in a list an places that value in a loop variable.
- **Note:** It is rare to find the C-style for loop used in Bash (if ever).
- The iterator style is more prevalent as the list can come from Ranges, Lists, Arrays or running commands.
- This style is similar to the foreach construct found in other scripting/ programming languages (java, C#, etc).

```
for (( exp1;exp2;exp3 ))
do
    some command
done
```

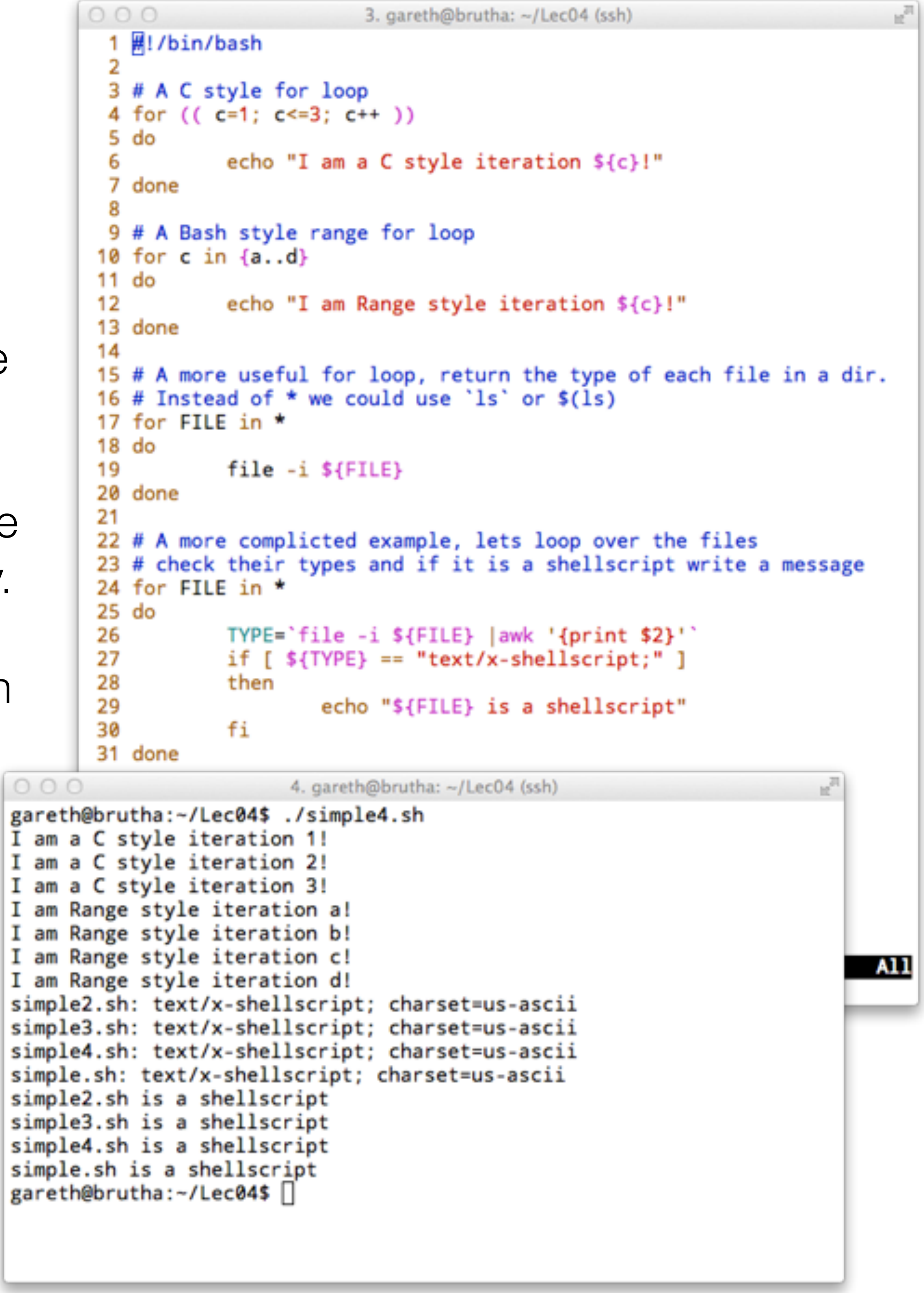
```
for i in {1..5}
do
    some command
done
```

```
for file in $(ls)
do
    some command
done
```

```
for file in /etc/*
do
    some command
done
```

For loops

- In our example, we can see C and Range style for loops.
- Notice that the ranges don’t need to be numerical.
- The more complicated examples iterate over all the files in our current directory.
- The first loop prints the output of **file** on each item in the directory.
- The second, takes the output of **file**, grabs the second element from the output and stores that in a variable.
- It then tests to see if that is a shell script, and if so prints a message to screen.


3. gareth@brutha: ~/Lec04 (ssh)
All
4. gareth@brutha: ~/Lec04 (ssh)

```
#!/bin/bash

# A C style for loop
for (( c=1; c<=3; c++ ))
do
        echo "I am a C style iteration ${c}!"
done

# A Bash style range for loop
for c in {a..d}
do
        echo "I am Range style iteration ${c}!"
done

# A more useful for loop, return the type of each file in a dir.
# Instead of * we could use `ls` or $(ls)
for FILE in *
do
        file -i ${FILE}
done

# A more complicted example, lets loop over the files
# check their types and if it is a shellscript write a message
for FILE in *
do
        TYPE=`file -i ${FILE} |awk '{print $2}'`
        if [ ${TYPE} == "text/x-shellscript;" ]
        then
                echo "${FILE} is a shellscript"
        fi
done
```

```
gareth@brutha:~/Lec04$ ./simple4.sh
I am a C style iteration 1!
I am a C style iteration 2!
I am a C style iteration 3!
I am Range style iteration a!
I am Range style iteration b!
I am Range style iteration c!
I am Range style iteration d!
simple2.sh: text/x-shellscript; charset=us-ascii
simple3.sh: text/x-shellscript; charset=us-ascii
simple4.sh: text/x-shellscript; charset=us-ascii
simple.sh: text/x-shellscript; charset=us-ascii
simple2.sh is a shellscript
simple3.sh is a shellscript
simple4.sh is a shellscript
simple.sh is a shellscript
gareth@brutha:~/Lec04$
```

While loops

- While loops will continue to loop while some condition holds true.
- You can use any of the conditionals we've already discussed (file, string or arithmetic).
- While loops are often used to create infinite loops that will exit on external conditions:
  - waiting for a file to be created then carry out an action.
  - carry out a task at a specific time interval (sampling data).
  - waiting for a long running process to complete.

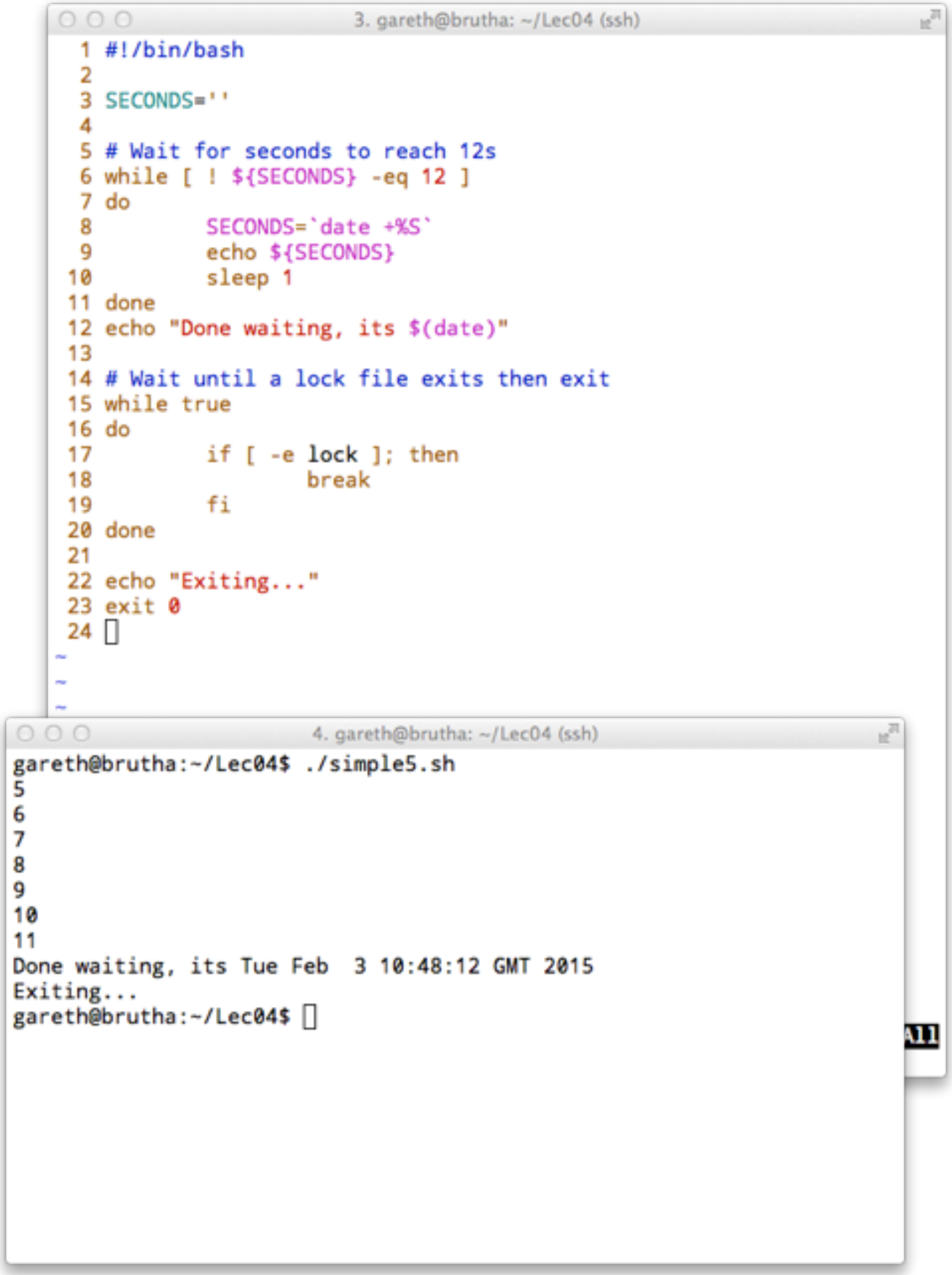
```
while [ conditional ]
do
    some command
done
```

```
while [ conditional ]; do
    some command
done
```

```
while true
do
    some command
    break
done
```

While loops

- In this example we use two while loops.
- The first loops until a variable **SECONDS** is equal to **12**
- **SECONDS** is set using the **date** command to contain the seconds of the current time.
- When this loop exits it **echo's** the current time.
- The second while loop continues to loop unless a **lock** file exits at which point we **break** out of the loop and exit.



3. gareth@brutha: ~/Lec04 (ssh)

```
#!/bin/bash

SECONDS=''

# Wait for seconds to reach 12s
while [ ! ${SECONDS} -eq 12 ]
do
        SECONDS=`date +%S`
        echo ${SECONDS}
        sleep 1
done
echo "Done waiting, its $(date)"

# Wait until a lock file exits then exit
while true
do
        if [ -e lock ]; then
                break
        fi
done

echo "Exiting..."
exit 0
```

4. gareth@brutha: ~/Lec04 (ssh)

```
gareth@brutha:~/Lec04$ ./simple5.sh
5
6
7
8
9
10
11
Done waiting, its Tue Feb  3 10:48:12 GMT 2015
Exiting...
gareth@brutha:~/Lec04$
```

- A simple script
- Tests (if)
- Ranges & Lists
- Loops (for, while)
- Special Vars

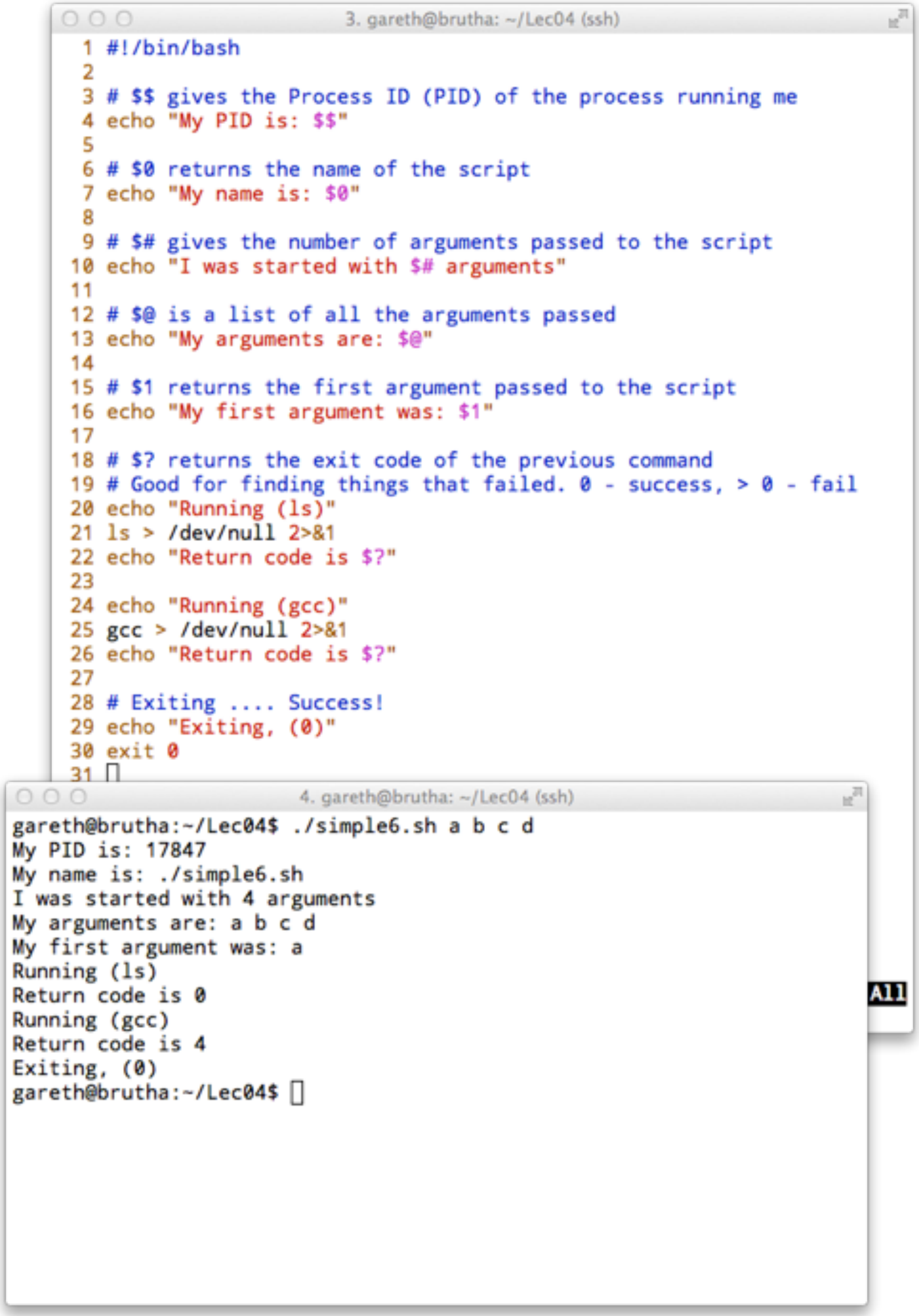
Special Variables

- Bash has some special variables that provide useful information.
- $1, $2, $@ and $# are used to work with arguments given to the shell script.
- $0 give the name of the running script
- $$ provides the script with it's PID when running.
- $? is used to check the return code of a command run as part of the script.
- This is useful to check for any error that occurred.

Variable	Usage
$$	Process ID of the running Bash script.
$0	The name of the Bash script
$1, $2, …	The first, second, … argument passed to the Bash script
$#	The number of arguments passed to the Bash script
$?	The return code of the previous command

Special Variables

- In this example we:
  - echo our PID
  - echo our scripts name
  - echo the number of arguments
  - echo the argument list
  - echo the first argument
  - tests the exit code of running ls and gcc (note ls succeeds and gcc fails).



3. gareth@brutha: ~/Lec04 (ssh)
4. gareth@brutha: ~/Lec04 (ssh)
All

```
#!/bin/bash

# $$ gives the Process ID (PID) of the process running me
echo "My PID is: $$"

# $0 returns the name of the script
echo "My name is: $0"

# $# gives the number of arguments passed to the script
echo "I was started with $# arguments"

# $@ is a list of all the arguments passed
echo "My arguments are: $@"

# $1 returns the first argument passed to the script
echo "My first argument was: $1"

# $? returns the exit code of the previous command
# Good for finding things that failed. 0 - success, > 0 - fail
echo "Running (ls)"
ls > /dev/null 2>&1
echo "Return code is $?"

echo "Running (gcc)"
gcc > /dev/null 2>&1
echo "Return code is $?"

# Exiting .... Success!
echo "Exiting, (0)"
exit 0
```

```
gareth@brutha:~/Lec04$ ./simple6.sh a b c d
My PID is: 17847
My name is: ./simple6.sh
I was started with 4 arguments
My arguments are: a b c d
My first argument was: a
Running (ls)
Return code is 0
Running (gcc)
Return code is 4
Exiting, (0)
gareth@brutha:~/Lec04$
```

- A simple script
- Tests (if)
- Ranges & Lists
- Loops (for, while)
- Special Vars