# Introduction to the Linux Shell

Ade Fewings, Aaron Owen
Bangor University
Updated: 2022-04-28

# eResearch

- Supercomputing Wales
  - Available to researchers at Bangor
- Research Software Engineers
- Collate expert knowledge into an open and shared centralised repository
  - Yammer
  - Github
  - Workshops
  - Projects
  - Acknowledgements

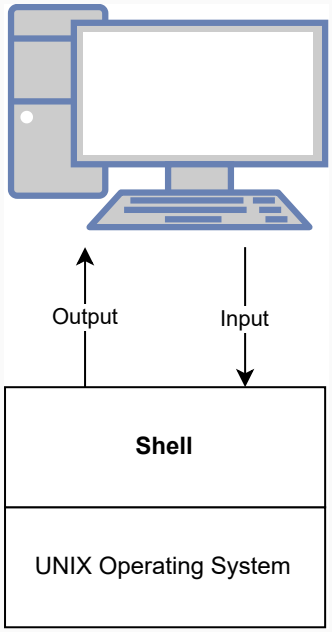
# Training Workshops

- **Introduction to the Linux Shell**
- Version Control Using Git
- Programming Principles and Practice using Python
- Advanced Python
- Parallel Processing in Python
- Machine Learning with Python*

See and discuss on the Yammer group.

Suggestions for new training welcome.

# What is a Shell?




- A program that provides a user interface for Unix-like operating systems (kernel)
- Its job is to translate the user's commands into operating system instructions
- The operating system starts a shell for each user when they log in or open a terminal window
- Examples of Shells
  - Bourne Family: sh, ash, ksh, **bash**
  - C Family: csh, tcsh
  - Perl Family: perlsh, zoidberg
  - Microsoft Family: cmd.exe, Windows PowerShell

# Why Bash?

- Installed as the default interactive shell for users on most Linux systems
- The use of **G**raphical **U**ser **I**nterfaces does not scale well when delivering instructions to a computer
- Repetitive tasks can be done automatically and quickly
- Sequences of commands can be written into a script, improving the reproducibility of workflows
- Two ways to use Bash: as a user interface and as a programming environment
- Bash will take some time to get familiar with, however, a small number of commands gets you a long way
- Basic Bash knowledge required to use the Supercomputing Wales service

# Option 1: Access a Linux machine

- Visit https://jupyter.bangor.ac.uk/jupyter/hub/login
- Login using your Bangor University credentials.
- Select the `Terminal` application.
- Type `ssh compute` and hit enter
- Can see last login credentials?
  - `Last login: Thu Nov 14 08:33:32 2019 from ssh.ad.bangor.ac.uk`
- `$` Presented with a prompt, indicating that the shell is ready to accept input

# Option 2: Access a Linux machine

- Open an application called 'Putty'
- Host Name : `ssh.bangor.ac.uk`
- Click 'Open'
- Enter your university username and hit enter
- Enter your university password and hit enter
- Type `ssh compute` and hit enter
- Can see last login credentials?
  - `Last login: Thu Nov 14 08:33:32 2019 from ssh.ad.bangor.ac.uk`
- `$` Presented with a prompt, indicating that the shell is ready to accept input

### Summary

Established a connection from your desktop to a remote Linux machine

# Explore

When you see an **Exercise** section, feel free to run the commands in the shell

### Exercise

- `cal` Display a calendar for the current month
- `cal 2022` Display a calendar for the year
- `date` Print the system date and time
- `w` See information about current users
- `whoami` Print your user id
- `users` Who is logged in?
- `echo 'Welcome to Bash'` Print a message
- `clear` or *Ctrl+L* Clear terminal
- `ks` Command not found? Usually means a user has mistyped the command

# Getting Help

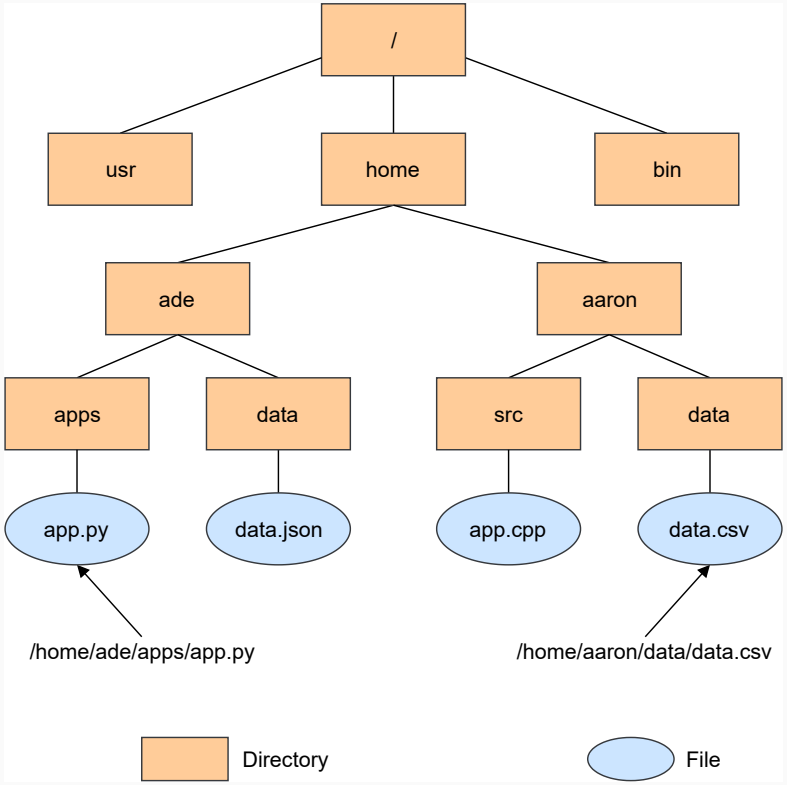
- `help` Provides information on commands in Bash
- `man` System manual
  - Press **q** to exit, **Spacebar** Go to next screen, **b** Go to the previous screen

Exercise

- `help`
- `man cal`
- `man w`
- `man whoami`
- `man bash`
- `man echo`

# The Working Directory

- The **working directory** is the directory that the shell is currently looking at
- Data in Unix is organised into files
- Files are organised into directories
- Directories are organised in a tree-like structure called the filesystem
- The filesystem is responsible for managing files and directories
- Absolute vs Relative Paths
- `pwd` Print the working directory
- `tree` Recursively list or display the content of a directory in a tree-like format




- Shell defaults to the home directory

# Files and Directories

## ls

- `ls` List the contents of a directory
- `man ls` What options are available? Press *q* to exit.
- `ls /tmp/bash` List the contents of */tmp/bash*
- `ls -a /tmp/bash` List contents of */tmp/bash*, including hidden files
- `ls -l /tmp/bash` Use long listing format when printing
- `ls ~` List contents of *your home* directory. `~` denotes your home directory

## Wildcard

- Metacharacters such as ***** and **?** have a special meaning in Unix.
- `ls *.py` Matches zero or more characters in a filename
- `ls ?c` Matches any single character

# Syntax of a Shell command

```
ls -l /tmp/bash
```

- `ls` command
- `-l` option
- `/tmp/bash` argument

### Help

- `man ls`
- `ls --help`
- `ls --help | less`
- `ls --x` Error
- Many commands have a `--help` option to display more information on how to use the command or program.

# Files and Directories

- `ls /tmp/bash` Lists contents by alphabetical order
- `ls -t /tmp/bash` Lists contents by last modified time
- `ls -r /tmp/bash` Reverse order when sorting
- `ls -tr /tmp/bash` or `ls -t -r /tmp/bash`
- `ls -ltr /tmp/bash` Use long listing format to verify

### Challenge

- List files in `/tmp/bash` by reverse alphabetical order

### Solution

- `ls -lr /tmp/bash`

# Files and Directories

```
ls -l

drwxr-xr-x. 2 issa16 is 4096 Nov 16 21:04 data
-rw-r--r--. 1 issa16 is  270 Nov 16 14:28 logins.txt
-rw-r--r--. 1 issa16 is   78 Nov 16 14:02 names.txt
```

### Columns

1. File type and permission given on the file (-,b,c,d,l,p,s)
2. Number of memory blocks taken by the file or directory
3. Owner of the file (creator)
4. Group owner. Each user will have a group.
5. File size in bytes.
6. Date and time the file was created or modified.
7. File or directory name.

# Navigating Directories

## cd

- `man cd` What options are available? Press *q* to exit.
- `cd` Change the working directory to your *home* directory
- `cd /tmp/bash` Change the working directory to */tmp/bash*
- `cd ..` Move up one directory
- `ls -l`?
- `pwd`
- `cd /tmp/bash` Change the working directory to */tmp/bash*
- `cd ../..` Move up two directories
- `ls -l`?
- `pwd`

# Navigating Directories

## Challenges

- Change the working directory to *home* directory
- Change the working directory to */tmp/bash*
- What happens if you try to change the working directory to */home/aaron*?

## Solutions

- `cd`
- `cd /tmp/bash`
- Permission denied or No such file or directory

## Tip

- Use the *up* and *down* arrow keys on the command line to navigate command history

# Creating Files and Directories

## touch

- `touch data.txt` Make an empty file called *data.txt*
- `man touch` What options are available? Press *q* to exit
- Also used to change the timestamp on existing files and directories

## mkdir

- `mkdir test` Make an empty directory called *test*
- `man mkdir` What options are available?
- `mkdir -vp ~/samples/feb` Make parent directories as needed

## Tips

- Do not use spaces for file and directory names, prefer `-` or `_`
- Do not begin file or directory names with `-`, usually reserved for options

# Creating Files and Directories

## Challenge

Create the following in your *home* directory

```
samples
  jan
    01.txt
  feb
    01.txt
```

## Solution

```
cd

mkdir -vp samples/jan
touch samples/jan/01.txt

mkdir -vp samples/feb
touch samples/feb/01.txt
```

# Removing Files and Directories

## rm

- `rm -i filename.py` Prompt before removing *filename.py* (y|n)
- `rm filename.py` Remove *filename.py* without prompt - Be careful
- `rm *.py` Remove all files that end with *.py*
- `rm -rfi ~/samples/feb` Remove all files and directories in the *path* directory

## rmdir

- `rmdir samples/feb`
  - Remove an empty directory using a relative path
- `rmdir /home/aaron/samples/feb`
  - Remove an empty directory using an absolute path

# Viewing Files

## cat

- Reads a file and outputs its content
- `cat /tmp/bash/names.txt` Print the contents of *names.txt*
- `cat -b /tmp/bash/colours.txt` Print the contents of *colours.txt* with line numbers
- `cat /tmp/bash/names.txt /tmp/bash/colours.txt` Print the contents of *names.txt* and *colours.txt*

## less

- View files with many lines of content
- `less /tmp/bash/colours.txt` View the contents of *colours.txt*
- *Spacebar* Go to next screen, *b* Go to the previous screen
- / Search for a word
- *q* Exit

# Editing Files

## nano

- `nano some-file` Open nano and start editing
- *Ctrl+O* Save changes
- *Ctrl+X* Exit
- `nano some-file` Open *some-file* for editing

## vim

- Opens in **command** mode, need to activate **insert** mode
- `vi some-file`
- `i` To start *insert* mode.
- `Esc :wq` To exit vi and save changes
- `Esc :q!` To exit vi and do not save changes

# Copying Files and Directories

## cp

- Copy source file or directory to destination file or directory
- `cp /tmp/bash/names.txt ~`
  - Copy */tmp/bash/names.txt* to your *home* directory
- `cp -r /tmp/bash ~`
  - Copy */tmp/bash* to your *home* directory
- `cp /tmp/bash/names.txt /tmp/bash/planets.txt ~`
  - Copy */tmp/bash/names.txt* and */tmp/bash/planets.txt* to your *home* directory

# Moving Files and Directories

## mv

- Moves one or more files or directories from a source to a destination
- `mv -i logins.txt logins_bk.txt`
  - Rename *logins.txt* as *logins_bk.txt* after confirmation from user (y|n)
- `mv logins.txt logins_bk.txt`
  - Rename *old_file* as *new_file* without confirmation check
- `mv *.txt data`
  - Move all python files to *data* directory

# Recap

- What is a Shell?
- The Working Directory?
- Syntax of a Shell command?
- Files and Directories
  - Create, edit, move and delete a file
  - Create, edit, move and delete a directory
- Navigating Directories

# Filters

A powerful feature in the Shell is the ability to combine commands and programs in new ways.

- `cat` Displays the contents of its inputs
- `head` Displays the first 10 lines of its input
- `tail` Displays the last 10 lines of its input
- `sort` Sorts its input
- `wc` Counts lines, words and characters in its inputs
- `uniq` Report or omit repeated lines

# Filters

## Exercise

- `cat /tmp/bash/colours.txt`?
- `head -n 5 /tmp/bash/colours.txt`?
- `tail -n 5 /tmp/bash/colours.txt`?
- `sort /tmp/bash/colours.txt`?
- `sort -R /tmp/bash/colours.txt`?
- `wc -l /tmp/bash/colours.txt`?
- `uniq /tmp/bash/colours.txt`

# Counting words in a file

## Challenge

- Use the `wc` command to count the number of words in `/tmp/bash/names.txt`

## Options

- `wc -l`
- `wc -c`

## Solution

- `wc /tmp/bash/names.txt`
    - 13 13 83 /tmp/bash/names.txt

## Columns

1. Total number of lines in the file
2. Total number of words in the file
3. Total number of bytes in the file (file size)

# Sort

### Challenge

- Apply the sort command to `/tmp/bash/planets.txt`

### Solution

- `sort /tmp/bash/planets.txt`

### Challenge

- Apply the sort command to `/tmp/bash/ages.txt` - numeric sort

### Solution

- `sort -n /tmp/bash/ages.txt`

# Sort

### Challenge

- Use the `sort` command to get the maximum age in `/tmp/bash/ages.txt`

### Solution

- `sort -n /tmp/bash/ages.txt | tail -n 1`

### Challenge

- Use the `sort` command to get the minimum age in `/tmp/bash/ages.txt`

### Solution

- `sort -n /tmp/bash/ages.txt | head -n 1`

# Pipelines

The `|` symbol between commands is called a **pipe**. It tells the shell that we want to use the output of the command on the left as input to the command on the right

Best way to use the shell is to use pipes to combine simple single-purpose programs (filters)

Task: Build a pipeline to count the number of unique login ids

- `cd` Change the working directory to your home directory
- `cp /tmp/bash/logins.txt .` Copy sample data to your home directory
- `cat logins.txt` View sample data
- `cat logins.txt | sort` Sort data
- `cat logins.txt | sort | uniq` View unique login ids
- `cat logins.txt | sort | uniq | wc -l` Count the number of unique login ids

# Redirection of data

Syntax

- `command-1 | command-2`
  - Use the output from command-1 as input to command-2
- `command > file`
  - Overwrite/create a file with output from the command
- `command >> file`
  - Append/create a file with output from the command
- `command < file`
  - Feed file to command as input
- `cat < file1 > file2` Any ideas?
- `cp file1 file2`

# Redirection of data

Exercise

- `echo "Line 1" > data.txt`
- `cat data.txt`
- `echo "Line 2" >> data.txt`
- `cat data.txt`
- `sort /tmp/bash/logins.txt > logins_sorted.txt`
- `cat logins_sorted.txt`
- Save all unique login ids to *unique_logins.txt*?
- `cat /tmp/bash/logins.txt | sort | uniq > unique_logins.txt`
- Read the last line of all python files and save to `last_lines.txt`
- `tail -n1 *.py > last_lines.txt`

# Search

- `grep` Select lines from text files that match simple patterns
- `find` Find files and directories whose names match simple patterns

## Exercise

- `cat /tmp/bash/colours.txt`
- `grep -n olive /tmp/bash/colours.txt`
- `grep -n olives /tmp/bash/colours.txt`
- `grep -nv olive /tmp/bash/colours.txt`
- `find . -type d` Find directories in the working directory
- `find . -name '*.txt'` Find all text files in the working directory
- `wc -l $(find . -name '*.txt')` Count the lines in all text files

# Scripts

- Move commands into script files
- Commands can be either directly entered by the user or read from a file called the shell script
- Scripts may contain commands, variables, loops, functions, conditionals etc.
- `nano test.sh`

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

echo "Hello from inside the test.sh file"
```

- `#!/usr/bin/env bash` - Ensure script is portable across different UNIX-like operating systems. Known as the shebang.
- `chmod +x test.sh` Convert the file to an exectuable
- `./test.sh` or `bash test.sh`

# Loops

- Allow us to repeat a command or set of commands for each item in a list
- Key to productivity improvements through automation
- Reduces the amount of code
- `help`
- `nano loop.sh`

```
#!/usr/bin/env bash
for idx in 1 2 3 4 5; do
  echo "Loop Index: ${idx}"
done
```

- `chmod +x loop.sh` Convert the file to an exectuable
- `./loop.sh` or `bash loop.sh`

# Loops

Task: Sort names, colours and planets and save results

- `nano filesort.sh`

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

for filename in names.txt colours.txt planets.txt; do

  sort "${filename}" > "${filename%%.*}_sorted.txt"

done
```

- `chmod +x filesort.sh` Convert the file to an exectuable
- `./filesort.sh` or `bash filesort.sh`

## Extract filename without extension

- `${filename%%.*}`

# Loops using a dry run

Task: Print the command but do not execute

- `nano dryrun.sh`

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

for filename in *.txt; do

  echo "uniq ${filename} > ${filename%%.*}_uniq.txt"
  # uniq ${filename} > ${filename%%.*}_uniq.txt

done
```

- `chmod +x dryrun.sh` Convert the file to an exectuable
- `./dryrun.sh` or `bash dryrun.sh`
- `#` Comment, not code. Purely for documentation.

# Command Substitution

- `nano users.sh`

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

DATE=`date`
echo "Date is ${DATE}"

ACTIVE_SESSIONS=`users | wc -w`
echo "There are ${ACTIVE_SESSIONS} user sessions currently active"
```

- `chmod +x users.sh` Convert the file to an executable
- `./users.sh` or `bash users.sh`
- `./users.sh > daily_users.txt`

# Variables

Task: Sum the integers from 1 to 10

- `nano sum.sh`

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

sum=0

for i in `seq 1 10`; do
      sum=$((sum+i))
      echo "idx:${i} sum:${sum}"
done

echo "Total Sum:${sum}"
```

- `chmod +x sum.sh` Convert the file to an executable
- `./sum.sh` or `bash sum.sh`
- `seq` Print a sequence of numbers, defaults to a step of 1

# Conditional Statements

Conditional statements enable you to code decisions into your Bash scripts

- `if` Perform a set of commands if a test is true
- `else` If the test is not true then perform a different set of commands
- `elif` If the previous test returned false perform a different test
- `&&` Perform the **AND** operation
- `||` Perform the **OR** operation
- `case` Choose a set of commands to execute depending on a string matching a particular pattern

# if

If a particular test it true, then perform a given set of commands

## Format

```
if [<test>]; then
  <set of commands>
fi
```

## Examples

```
i=10
if [ $i -gt 5 ]; then
  echo "$i is greater than 5"
fi

# Numerical comparison
if [ 001 -eq 1 ]; then
  echo "True"
fi
```

```
name="Aaron"
if [ $name = "Aaron" ]; then
  echo "Hi Aaron!"
fi

# String comparison
if [ 001 = 1 ]; then
    echo "Hi"
fi
```

# Arithmetic Comparisons

- `x -eq y` Test identity
- `x -ne y` Test inequality
- `x -lt y` Less than
- `x -gt y` Greater than
- `x -le y` Less than or equal
- `x -ge y` Greater than equal

# if else

If the test is not true then perform a different set of commands

**Format**

```
if [ <test> ]; then
  <set of commands>
else
  <set of commands>
fi
```

**Examples**

```
i=2
if [ $i -gt 5 ]; then
    echo "$i is greater than 5"
else
    echo "$i is less than or equal to 5"
fi
```

# if elif else

If the previous test returned false perform a different test

**Format**

```
if [ <test1> ]; then
  <set of commands>
elif [ <test2> ]; then
  <set of commands>
else
  <set of commands>
fi
```

**Examples**

```
i=20
if [ $i -lt 10 ]; then
    echo "$i is less than 10"
elif [ $i -gt 10 ]; then
    echo "$i is greater than 10."
else
    echo "None of the above tests returned true."
fi
```

# AND Operation

Perform the **AND** operation

```
name="Aaron"
i=20
if [ $name == "Aaron" ] && [ $i -eq 20 ]; then
    echo "Both conditions returned true."
fi
```

# OR Operation

Perform the **OR** operation

```
name="Aaron"
i=10
if [ $name == "Aaron" ] || [ $i -eq 20 ]; then
    echo "One of the conditions returned true."
fi
```

# Case Statements

Choose a set of commands to execute depending on a string matching a particular pattern

**Format**

```
case <variable> in
<pattern 1>)
  <set of commands>
  ;;
<pattern 2>)
  <set of commands>
  ;;
esac
```

**Example**

```
status="start"
case $status in
start)
    echo "Starting"
    ;;
stop)
    echo "Stopping"
    ;;
restart)
    echo "Restarting"
    ;;
*)
    echo "Unknown status"
    ;;
esac
```

# Functions

A function is essentially a set of commands that can be called numerous times. It improves the readability of code and avoids writing the same code repeatedly.

**Format**

```
// Multi line
function_name () {
  commands
}

// Single line
function_name () { commands; }
```

**Example**

```
hello_world() {
    echo "Hello World!"
}
hello_world

hello() {
    echo "Hello $1!"
}
hello "Aaron"
```

# Variables Scope

**Global variables** are variables that can be accessed from anywhere in the script regardless of the scope. In Bash, all variables by default are defined as global, even if declared inside the function.

**Local variables** can be declared within the function body with the local keyword and can be used only inside that function. You can have local variables with the same name in different functions.

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

x=1
y=2

test_function() {
    local x=3
    y=4
    echo "Inside function: x: $x, y: $y"
}

echo "Before executing function: x: $x, y: $y"
test_function
echo "After executing function: x: $x, y: $y"
```

# User Environment

- TODO

# Change Password

- TODO

# Brace Expansion

- TODO

# wget

- TODO

# File Transfer

- TODO

# Arrays

- TODO

# Dictionaries

- TODO

# Jobs

- Typically when you run a command in the terminal, you have to wait until the command finishes before you can enter another one.
- This is called running the command in the **foreground** or a **foreground process**.

```
sleep 5 && echo "Hello World"
```

- If a command takes a long time to finish and you want to run another command in the meantime you have two options. You can either open up a new terminal and run the command or run the command as a **background process**.
- Think of running a **background process** as dispatching the command to the operating system and freeing up the terminal.

# Jobs

- To run a command as a background process, add `nohup` before the command and the ampersand symbol `&` at the end of the command.

  ```
  nohup sleep 5 && echo "Hello World" &
  ```

- The **job id** and **process id** will be printed to the terminal.

  ```
  [1] 195684
  ```

- By default, the example above will print the message "Hello World" to the terminal after 5 seconds. This is less than ideal, especially if we are the middle of writing a complex command to launch. The solution is to capture any error or output messages from the command into a log file.

- To redirect error and output messages, use the ampersand and greater than symbols `&>` following by the name of the log file.

  - `nohup sleep 5 && echo "Hello World" &> job.log &`

# Jobs

- Once we have created a background process we can view its status via the `jobs` command.
    - `jobs -l`
- The output includes the job id, process id, job state and the command that started the job.
- If you wanted to bring background process to the foreground you can use the `fg` command.
    - `fg`
- If you have multiple background processes, include `%` and the job id after the `fg` command.
    - `fg %2`
- Note: The `jobs` command will only show background processes that are started, running or stopped in the current terminal.

# Jobs

## Exercise

- Log in into to the compute node at Bangor via JupyterLab.
- Start a Matlab background process.

  ```
  nohup matlab -nodesktop -nosplash -r "disp('Hello World'); exit;" &> matlab.log &
  ```

  ```
  [1] 80706
  ```

- Logout using the `exit` command.
- Log back in into to the compute node using the `ssh compute` command.
- View the contents of matlab.log using the `cat matlab.log` command.
- If you want to stop a background process manually, you can use the `kill` command followed by the jobs process id.

  ```
  kill 80706
  ```

# Parallel

- TODO

# Links

- Bash scripting cheatsheet
- GNU Bash
- The Unix Shell
- Bash scripting cheatsheet
- ShellCheck – shell script analysis tool
- Shell Style Guide
- Learn X in Y minutes - Bash