

# CITS4407 Open Source Tools and Scripting Introduction

Unit coordinator: Arran Stewart

# Overview

**Focus of the unit:** this unit focuses on the philosophy, design, tools, and practices that enable and facilitate the success of open source software, which runs much of the world's computer infrastructure.

We look at topics including:

- use of the shell as a programming language
- the use of the file system and pipes to support interprocess communication
- fundamental software components
- tools supporting the software development and maintenance process
- the importance of consistent interfaces to support software integration.

# Motivation

## Why learn about these things?

- Allows us to automate complicated tasks that would require arduous manual labor.
- Provides a set of small commands and utilities that can be combined in unlimited ways to perform complex custom tasks.
- Extremely useful computer skill that will be relevant many years from now.
- Means we are not limited to tools provided by software suppliers - we have our own toolbox, and can create our own tools.
- Very useful in fields like data science - a major portion of work is spent using tools to obtain and process data, and convert it from one form to another.

# Admin

# Unit Information

Unit Coordinator: Arran Stewart

Contact: [arran.stewart@uwa.edu.au](mailto:arran.stewart@uwa.edu.au)

Phone: +61 8 6488 1945

Office: Rm G.08 CSSE Building

Consultation: Email for an appointment.

Unit webpage: accessible via GitHub, at  
<https://github.com/cits4407>

# Announcements

Announcements will be made in lectures, and on the unit help forum, [help4407](#).

It's important to check the forum regularly – at least once a week – and it is possible to set up an email subscription so you're alerted when new postings are made.

# Unit contact hours – details

## Lectures:

- You should attend one lecture per week – you should either attend in person, attend online (we will use Zoom), or watch the recorded lecture. (Recorded lectures are available via the university's LMS, at <https://lms.uwa.edu.au/>.)

## Workshops:

- You should attend one lab/workshop each week, starting in week *two*.  
If there is room available for you, you are welcome to attend other lab sessions as well.
- In the lab/workshops, we will work through practical exercises related to the unit material. If you have a laptop, it will be very useful to bring it! But you can use lab computers if not.

# Non-timetabled hours

A six-point unit is deemed to be equivalent to one quarter of a full-time workload, so you are expected to commit 10–12 hours per week to the unit, averaged over the entire semester.

Outside of the contact hours (3 hours per week) for the unit, the remainder of your time should be spent reading the recommended reading, attempting exercises and working on assignment tasks.



# Best sources of information

- **Unit webpage.** Your first point of call for information.
- **Ask in lectures or labs.** Often the quickest way of getting an answer!
- **Help4407.** The discussion and question forum for the unit – if you have questions not answered on the webpage, and your question would benefit other students, ask it here.
- **Unit coordinator.** If you have questions that are not appropriate for the discussion forum, the next best avenue is to email the unit coordinator (me), or arrange to come to a consultation (either in person, subject to Covid restrictions, or by telephone or online).

# Textbook

There is no one textbook that covers all the content of this unit.

But an open-sourced textbook by William Shotts is helpful for many topics:

- William E. Shotts, Jr, *The Linux Command Line: A Complete Introduction*

See the website for details: <https://cits4407.github.io/#textbook>

It's available for download as a PDF, or via the UWA library, or for purchase as an e-book or paperback (either from Amazon, directly from No Starch Press, or from second-hand booksellers).

# Assessment

The assessment for CITS4407 consists of an online quiz (worth 10%, done on LMS, due in week 3), two assignments (due in weeks 7 and 12), and an exam: see the Assessment page at

- <https://cits4407.github.io/assessment/>

for more details.

# Assessment

In general, assessments will be due at 5pm Friday on the week they are due, but check the Help forum for more information closer to the date.

# Schedule

- The current unit schedule is available on the unit website:  
<https://cits4407.github.io/schedule/>
- The schedule gives recommended readings for each topic: either chapters from the Shotts textbook, or extracts or webpages. Your understanding of the lecture and workshop material will be greatly enhanced if you work through these readings prior to attending.

# Programming languages

We don't assume *any* familiarity with programming, but hope that by the end of the course, you will have a familiarity with [Bash](#), written by Brian Fox and released in 1989.

If you are also doing (or have already done) Python or Java programming, then learning bash should be fairly straightforward – but we will go at a pace that everyone should be able to keep up with.

(If you want more challenging exercises, let me know!)

# Why learn bash?

Bash is a very widely used *Unix shell* – a program that accepts commands written via the keyboard, and passes them to the operating system (or *OS*) to carry out.

Most operating systems (including MacOS X and Windows) do provide a shell, but non-Unix operating systems tends to place less emphasis on it.

On Windows, the default shell is called `cmd.exe`, and on MacOS X, the default shell is called [Zsh](#), and is related to Bash.

# Why learn how to use a Unix shell?

- On Unix-like systems, using a *shell* is still the primary way of getting many tasks done.
- Using a shell is much more *powerful* than trying to do things purely through the GUI (Graphical User Interface), and often much faster.



## Why not some other language?

- You might ask, Why not just use Python? Or Java?
- Any task you can do in Bash, it's always *possible* to do using another language – so you could use Python, or Java, or many other languages if you wished.
- Unix shells integrate very tightly with the operating system, and make it very quick and easy to control what the OS is doing.

# Operating system

The operating system on which all assignments will be marked, and on which they are expected to run, is [Linux](#) – specifically, the Ubuntu 20.04 distribution of Linux.

You can use Bash and many other Unix tools from other operating systems – but it is best to make sure you do test your programs on Ubuntu Linux.

More on how to get access to it in a second.

## Lab/workshops

- Much of the learning for this unit will take place in the labs, so make sure you can attend one!
- If you are attending online – the lab facilitators will be available via Zoom. (Watch the Help4407 forum for details of how to “join” the Zoom labs.)
- Some labs sessions are *purely* for online students (e.g. 2pm Friday) – others, students can attend in person, or online.

# Online access to computers

- UWA IT's recommended way of accessing computers online is via **Unidesk** (<https://unidesk.uwa.edu.au>). I am still attempting to confirm that this will work to allow you access to Linux.
- However – if you have access to a Windows laptop or PC, it will likely be much quicker and easier to install Ubuntu 20.04 on your computer and use it that way.
- For each lab, there will be a Zoom meeting you can join if attending online - keep an eye on the Help forum for details.

# Accessing Linux from Windows

If you have a Windows laptop or home PC, you can install a virtualised Linux operating system using what's called *the Windows Subsystem for Linux* (or WSL) – you just select a Linux *distribution* from the Microsoft Store.

More details in the first lab, next week – but for those on Windows who want to try it out before then, take a look at

<https://wiki.ubuntu.com/WSL>

and look for “Installing Ubuntu on WSL via the Microsoft Store (Recommended)”.

# Access Linux from MacOS X

If you have access to a MacOS X laptop or home PC, you may want to install one of

- Parallels Desktop for the Mac –  
<https://www.parallels.com/products/desktop/>
- VirtualBox – open source and free! <https://www.virtualbox.org>
- VMWare Workstation –  
<https://www.vmware.com/au/products/workstation-player/workstation-player-evaluation.html>. We have student licenses available for this software.

More details in the first lab worksheet; but once one of those packages is installed, you can install a virtual Linux OS.

# Access Linux from Linux

If you are already running Linux on a laptop or home PC ... then installing Linux is a done deal for you.

But if you want to ensure your assignments run correctly in an Ubuntu 20.04 environment, you might want to install Docker (<https://www.docker.com>) on your OS.

# Questions?

- Take a minute to discuss with a neighbour – what would you need to know to do well in this unit? How would you define “doing well”?



# Unix

# Unix

# Unix systems

## What is Unix?

- Developed in 1969 at Bell Labs (part of AT&T, the American Telephone and Telegraph Company)
- Named for the founder of AT&T's precursor, the Bell Telephone Company, founded by Alexander Graham Bell in 1877.
- Rewritten in C in 1973 (before that, assembly language)

# Unix systems

From Wikipedia:

*Unix systems are characterized by various concepts: the use of plain text for storing data; a hierarchical file system; treating devices and certain types of inter-process communication (IPC) as files; and the use of a large number of software tools, small programs that can be strung together through a command line interpreter using pipes, as opposed to using a single monolithic program that includes all of the same functionality. These concepts are known as the Unix philosophy.*

# Unix-like

Again, from Wikipedia:

*A Unix-like (sometimes referred to as UN\*X or \*nix) operating system is one that behaves in a manner similar to a Unix system, while not necessarily conforming to or being certified to any version of the Single UNIX Specification.*

# OS components

An operating system distribution is normally divided into (at least) two major portions:

- A *kernel* - the “master control” program, which starts and stops processes, handles low-level interaction with hardware, etc.
- *Applications* - many modular tools and programs

Each program runs within its own *process*

# Programs

- Nearly every thing you use in Unix or Linux is really an external program (not a shell command or part of the kernel).
- Most of these communicate with the outside world in just four ways:
  - They get arguments on the command line
  - They receive input from standard input
  - They send output to standard output
  - (They also send error messages to standard error).
- They are small, reusable pieces that you can assemble in any way you like to do complex tasks.

# Files

- Unix treats almost everything (programs, hardware devices, kernel information, directories) as a type of *file*.



# Exercises

- The best way to learn how to use a Unix or Unix-like system is through practice.

# Open Source

# What is “open source”?

*open-source* əʊp(ə)n'sɔ:s / ► adjective Computing  
denoting software for which the original source code is made freely available and may be redistributed and modified.

— Oxford English Dictionary

# What is “open source”?

There is actually some contention over what exactly “open source” means.

But the term was originally used in relation to software, and in contrast to software which is simply (monetarily) “free”.

Lots of software is *free* in the sense that it doesn't cost you any money to use; but only some of that software gives you free access to the *source code* for the software, and allows you to change it, and redistribute your changes.

# What is “open source”?

“Open source” originally applied to software, but now is also used for other types of content – for instance, art works, written works other than software, and even music scores and performances.

# Why use (or create) open source software

Some key ones are:

- flexibility,
- cost, and
- transparency.

Some people also assert that open source software

- is of higher quality
- has better security, and
- is more stable

than closed-source software, although this is difficult to verify.

We will look more at the philosophy behind open source software later.

# CITS4407 Open Source Tools and Scripting

## Version control, processes and pipelines

Unit coordinator: Arran Stewart

# Overview

This week:

- files and filesystems – how does the Unix approach differ from, say, Windows?
- processes – what are they?
- what is version control, and why should we use it?



# Files and filesystems

- Some things about a Unix system are not too dissimilar to Windows.
- For instance – users' data is stored in files, files are stored in directories, and both are stored on disks.
- But some other aspects are quite different.

# “Everything is a file”

The Unix philosophy is to treat almost *everything* as a file.

- On Windows – if you want to see what tasks are running, there is a task manager program you can run which gives you this information.
- If you want to see what devices (like microphones or USB drives) are plugged in, there is a device manager program you can run.

# “Everything is a file”

On Linux – all that information is visible by looking at the content of particular files.

(These are *virtual files*, and are said to live in a *virtual filesystem*. This just means they don't represent an actual file stored on disk, but rather present a “view” of some aspect of the operating system.)

- For convenience, Linux does also have graphical (and command-line) programs much like Windows's task manager and device manager;
- however, they're not *necessary* – you *could* get all the information you needed just by looking in particular files.

In a lab, try typing “`ls /proc`” sometime. The `/proc` directory holds details of all the programs that are currently running.

# Processes

- To be more precise, what the `/proc` directory lists is details of *processes*.
- To a first approximation, you can think of a process as “a running program” ...
- But processes aren’t necessarily “running”.
- They could be running, or they could be
  - waiting for data from a device like a disk drive
  - sleeping, because the process is waiting for an event like the user hitting a key on the keyboard
  - stopped (which you might think of as like “suspended”) – usually because the user deliberately stopped the process.

# Processes

- We can also see what processes are running by using the command “ps”.

```
arran@barkley:cits4407-website$ ps
```

PID	TTY	TIME	CMD
2008	pts/24	00:00:00	ps
21347	pts/24	00:00:00	bash
31247	pts/24	00:00:08	evince

(Evince is a PDF viewer on Ubuntu Linux.)

# Processes

By default, `ps` only list processes that are part of what is called “the current session” (programs launched from the terminal, or terminal window, you’re currently in).

`ps -A` lists *all* processes on the computer. Not just commands run by you, the user, but also what are called *daemons* – programs started by the operating system, and which run constantly “in the background”.

(Windows has an equivalent notion; on Windows, programs like this are called “services”.)

# Processes and pipelines

One feature pioneered by Unix<sup>1</sup> is the idea of what are called *pipelines* between processes.

Often, we will want to perform multiple operations on a file – we might have student information stored in files, and might want to extract student names and marks; then sort the result of that, in descending order, by mark; then take just the top five students listed; and then send *that* list to a printer (or a program that, say, converts the result to PDF format).

---

<sup>1</sup>Although the idea had appeared independently before. 

# Processes and pipelines

We *could* do the extraction of names and marks, and store the result in a file; and then sort that file, and put the sorted result in a second file; and then run a command which just gives us the first five students.

But Bash allows us to avoid explicitly creating all those intermediate files, and instead “chain” commands together – *piping* the result of one command to another command.



# Processes and pipelines

Piping in bash is done using what is often called (by programmers) the “pipe” character, “|”.

(Usually found just above the “enter” key, on English-language keyboards.)

# Processes and pipelines

The sequence of commands to extract names and marks, sort, and get the first five lines could be done put in a “pipeline” like this:

```
$ cut -f 1,4 marks.txt | sort -n --key 2 | head -n 5
```

We'll see how to “compose” pipelines in the lab/workshops.

# Version Control, Git, and GitHub

# Version control

At its simplest, a *version control system* (VCS) lets you track how a set of files change over time, and lets you “roll back” to previous points in their history.

More complex things you might do are

- “roll back” to some previous point in time, and then “branch off”, making different changes to the ones you made previously
- delete “branches” of history you no longer need
- *merge* branches of history – combining the changes made in two separate branches.

# Version control

Often, people working with files end up creating “informal” version control systems themselves – keeping multiple versions of a file like

- AlgorithmsAssignment1Draft.doc
- AlgorithmsAssignment1FinalVersion.doc
- AlgorithmsAssignment1FinalFinalVersion.doc
- AlgorithmsAssignment1FinalFinalVersion (02).doc

so that they can get back to a previous version if they need to.

# Version control

But this kind of informal “version control system” relies on us remembering what each file name meant, and how it related to other files – and it becomes harder to manage if multiple people are trying to make many changes to the files at once.

But that sort of situation is exactly the case when writing software (or managing business data).

# Version control

So we use *software* version control systems, which track

- when a file was changed, and how
- who changed it
- *why* they changed it (by allowing users to leave *comments* about the changes they have made)
- how the change relates to other changes – what changes came before and after it.

# Advantages of version control

If we ever need to

- take all our files back to the state they were in on 1 March, 2020
- look at conflicting changes two users have made to a file, and decide which should take precedence
- work out exactly which change introduced a bug which crashed our system last week

then version control systems let us do this.



# Git

We will suggest using a version control system call *Git*, created by Linus Torvalds (the creator of Linux) for tracking changes made to the Linux kernel, in 2005.

# Terminology

Some terminology –

- a *repository* (or “repo”) is a set of files you wish to track
- a repository may be stored on a remote *server* – some computer that stores the files
- a *commit* (or “revision”) is a state of the files at a particular point in their “history”
- *cloning* a repository means to make a copy of it (including all the history details etc it contains) – typically the new repository will keep a record of where it was cloned from

# Very basic Git use

An example of use:

```
$ cd MyProjects  
$ mkdir assignment1  
$ cd assignment1  
$ git init
```

```
git init
```

# git init

Here we've create a directory to store files in, and *initialized* it as a Git repository.

That creates a hidden directory call “.git” within our assignment1 directory, and files in this hidden directory track all the information about what changes were made when.

# Adding files

Suppose we now create a file in our `assignment1` directory (perhaps using a text editor) called “`fabulous-program.sh`”, and decide we now wish to keep this file under version control.

```
$ git add fabulous-program.sh  
$ git commit -m "initial version of the fabulous program"
```

Every time we make a change to the file which we wish to record, we repeat these commands.

## Other Git operations

“init”, “add” and “commit” are examples of what are sometimes called *subcommands* – key words that specify a particular way you want to use a command, and which have *arguments* (the words that appear afterwards) of their own that control their behaviour.

In lab/workshops, we will see how to perform more complex operations – but if you know the init, add and commit commands, you can be assured that you can retrieve previous versions of your files.

# GitHub

- Don't confuse the “git” command with **GitHub**
- “git” is an open source program; GitHub is a web-based hosting service for git repositories, currently owned by Microsoft
- GitHub is extremely popular, but there are many other competing hosting services:
  - BitBucket (<https://bitbucket.org/>)
  - GitLab (<https://gitlab.com>)
  - SourceHut (<https://sourcehut.org>)



# Hosting services

- Also note that nothing about `git` *requires* that you use a hosting service – they are simply convenient ways of sharing a repository with other people.
- Some software projects host their own repositories (“hosting” a repository just means having a website through which the repository is accessible)
- Some projects and organisations don’t use Git at all – there are many other version control systems available (though Git is certainly one of the most popular)

# CITS4407 Open Source Tools and Scripting

## Editors, scripts, and control structures

Unit coordinator: Arran Stewart

# Overview

This week:

- variables
- creating our own commands
- control flow

# Storing information in shell variables

Bash will let us store information we want to keep for later in *variables* – these let us give the information a name, and refer to it later.

```
$ useful_url="http://pixelastic.github.io/pokemonorbigdata/"  
$ echo $useful_url  
http://pixelastic.github.io/pokemonorbigdata/  
$ firefox $useful_url
```

# Unsetting variables

If we want to get rid of a variable, we can use unset:

```
$ unset useful_url  
$ echo $useful_url  
  
$
```

# Environment variables

In fact, every time we use a Linux environment, there are already a large number of variables defined.

```
$ echo $USER
arran
$ echo $PWD
/home/arran/teaching/cits4407
$ echo $BASH_VERSION
4.3.48(1)-release
```

## Environment variables

# Environment variables

- Every running process – not just bash programs – has a set of *environment* variables.
- These normally have names in uppercase.
- A few environment variables defined by Linux:
  - HOME – the path to your home directory
  - PATH – a colon-separated list of directories which will be searched for executables
  - PWD – the current working directory
  - USER – your username



## Environment variables

# Environment variables

Environment variables are a little different from normal variables.

When we run an external command – i.e. not a builtin bash command – it *inherits* its environment from bash.

It doesn't inherit normal variables – just environment variables.



# Environment variables

But we can *make* a normal variable an environment variable:

```
$ useful_url="http://pixelastic.github.io/pokemonorbigdata/"  
$ export useful_url
```

# Environment variables

# Environment variables

But we can *make* a normal variable an environment variable:

```
$ useful_url="http://pixelastic.github.io/pokemonorbigdata/"  
$ export useful_url
```

# Environment variables

# Environment variables

Or turn it back into a normal variable again.

```
$ export -n useful_url
```

(Typing `help export` gives a little more information on this.)



# Variables

If we want to see all variables – the command to use (unintuitively) is `set`:

```
$ set
BASH=/bin/bash
BASHOPTS=checkwinsize:cmdhist:complete_fullquote:expand_aliases
BASH_ALIASES=()
BASH_ARGC=()
BASH_ARGV=()
BASH_CMDS=()
BASH_COMPLETION_COMPAT_DIR=/etc/bash_completion.d
BASH_LINENO=()
BASH_REMATCH=()
BASH_SOURCE=()
...
```

# Numbers in variables

- By default, Bash treats variables as containing *strings* of text.
- But it's also possible to convince bash to treat variables as numbers:

```
$ myvar="3"  
$ echo $((myvar + 4))  
7
```

## help on bash commands and constructs



```
$ help "("
(( ... )): (( expression ))
    Evaluate arithmetic expression.
...
```

# Commands in variables

We could use variables to store frequently used commands, so we can refer to them later:

```
$ cd_scripting="cd /home/arran/teaching/cits4407"  
$ echo $cd_scripting  
/home/arran/teaching/cits4407  
$ $cd_scripting  
$ pwd  
/home/arran/teaching/cits4407
```

But there are better ways of creating our own commands.



# alias

A simple way to do so is to use the `alias` command.

For instance, if I frequently want to change directory into the directory where I keep my CITS4007 content, I might write:

```
alias cd-scripting="cd /home/arran/teaching/cits4407"
```

This creates a new command, `cd-scripting`, which runs the `cd` command with the argument `/home/arran/teaching/2021/cits4407`.



## alias

And now, I can just type `cd-scripting` to get to that directory.

In fact, you likely already have some aliases already defined. Typing “alias” on its own shows what they are:

```
$ alias
alias egrep='egrep --color=auto'
alias fgrep='fgrep --color=auto'
alias grep='grep --color=auto'
alias gs='git status'
alias l='ls -CF'
alias la='ls -A'
alias ll='ls -alF'
alias ls='ls --color=auto'
alias rl='readlink'
```

Some other ways I can define a command are:

- ```
cd_tmp () { cd /tmp; }
```

## Defining commands

Some other ways I can define a command are:

- write a function:  
`cd_tmp () { cd /tmp; }`
- put one or more bash commands in a *bash script*:  
`echo "cd /tmp" > cd_tmp`

# Defining commands

Some other ways I can define a command are:

- write a function:  
`cd_tmp () { cd /tmp; }`
- put one or more bash commands in a *bash script*:  
`echo "cd /tmp" > cd_tmp`
- create an executable program in some other language besides bash – for instance, C.

# Scripts

We'll look at function definitions later; today we'll consider scripts.

A *bash script* is just a file containing one or more bash commands.

my-script:

```
1 echo "Hello, the date today is:"  
2 date
```

# Scripts

If we know the location of a bash script, we can ask bash to run it:

```
$ bash /home/arran/my-script  
Hello, the date today is:  
Wednesday 10 March 13:43:53 AWST 2021
```

# Scripts

If we make the script *executable*, and give it a special first line – called the *shebang line* – then we tell Linux to always run that script using bash:

my-script:

```
1 #!/bin/bash
2
3 echo "Hello, the date today is:"
4 date
```

```
$ chmod a+rx /home/arran/my-script
$ /home/arran/my-script
Hello, the date today is:
Wednesday 10 March 13:44:17 AWST 2021
```

# Scripts

And if we tell bash a location where we are storing scripts, we can run our script without having to specify the location:

```
$ mkdir /home/arran/bin
$ mv /home/arran/my-script /home/arran/bin
$ PATH=/home/arran/bin:$PATH
$ my-script
Hello, the date today is:
Wednesday 10 March 13:49:42 AWST 2021
```



## Expansion

When we put a dollar sign in front of a variable, bash is said to *expand* the variable into the value we gave it:

```
$ echo $useful_url
http://pixelastic.github.io/pokemonorbigdata/
```

If we want to expand a variable, and have other text adjoining it, we can demarcate its name with braces:

```
... foo
```

```
echo ${useful_url}andotherstuff
http://pixelastic.github.io/pokemonorbigdata/andotherstuff
```



# Flow control

Often in bash scripts, we'll only want to do something *if* some condition is true.

The built-in “if” command lets us do this:

```
$ if ls xxx; then echo "xxx exists"; else echo "it doesn't"; fi
ls: cannot access 'xxx': No such file or directory
it doesn't
```

# if

In scripts, we normally don't write `if` statements on one line.

```
myscript.sh:
if /commands/; then
    /commands/
elif /commands/; then
    /commands/
else
    /commands/
fi
```

More on control flow in future classes.

# CITS4407 Open Source Tools and Scripting

## Text and regular expressions

Unit coordinator: Arran Stewart

# Overview

This week:

- File expansion and globbing
- Regular expressions

# Patterns in text

Often when using a system will want to be able to specify that we want to perform an operation on multiple files.

# Patterns in text

Suppose we want to perform some operation on all files ending with “.sh” in the current directory – say, copy them to a backup directory.

We could write in all the filenames by hand:

```
$ cp myscript.sh myotherscript.sh first_assignment.sh ~/backups
```



# Patterns in text

Suppose we want to perform some operation on all files ending with “.sh” in the current directory – say, copy them to a backup directory.

We could write in all the filenames by hand:

```
$ cp myscript.sh myotherscript.sh first_assignment.sh ~/backups
```

Or we could use a *wildcard character*, as follows:

```
$ cp *.sh ~/backups
```

# Wildcards

The asterisk (“\*”) represents an arbitrary string of characters; typing a command like `ls *.sh` will list all files ending in “.sh”.<sup>1</sup>

(Other systems besides Unix-like ones use wildcard characters; Windows and MacOS X do, too.

And often, search engines or library-catalogue databases will allow users to include wildcard characters of some sort when searching for items.)

---

<sup>1</sup>Actually, it will not match files whose name starts with a full stop (“.”); they are called “dotfiles” and are Unix-like systems’ equivalent of *hidden files* in Windows.

# Filename expansion

When Bash sees a word containing `*`, it performs what's called *filename expansion*.

This process is also called *globbing*<sup>2</sup>, and `*`, together with `?` are known as *glob patterns*.

`?` matches a *single* letter in a filename:

```
$ ls
file1 file10 file2 file3
$ ls file?
file1 file2 file3
```

---

<sup>2</sup>See [https://en.wikipedia.org/wiki/Glob\\_\(programming\)](https://en.wikipedia.org/wiki/Glob_(programming)) for an explanation of why it is called this.

# Filename expansion

Another filename expansion lets us specify that a character must match one out of a set of characters we specify:

```
$ ls
file1 file10 file2 file3
$ ls file[23]
file2 file3
```

# Going beyond files

Sometimes we will want to specify patterns in text other than filenames.

For instance, we might want to display ...

- all lines in a log file which start with the text “ALERT:”
- all lines in a file containing sales information, which contain the text “keyboards”
- all processes on the system started by a user who isn't us, and isn't root

*Regular expressions* allow us to do this.

# What are regular expressions?

Regular expressions are a way of specifying *patterns* in text, and the `grep` command<sup>3</sup> gives us a way of searching for patterns in text.

The simplest sort of expression is just a sequence of ordinary characters.

`grep ALERT` will print all lines from standard input which contain the string “ALERT”.

---

<sup>3</sup>Short for “**g**lobal **r**egular **e**xpression **p**rint”.

# Patterns in files

If we want to list all files in our current directory containing the word expression, we can do this using grep:

```
$ grep expression *  
grep: lect01-images: Is a directory  
lect03.md:(( ... )): (( expression ))  
lect03.md:    Evaluate arithmetic expression.
```

# grep options

Some useful options for the grep command:

- `-i` or `--ignore-case`: do a *case-insensitive* search
- `-v` or `--invert-match`: print all lines that *don't* match the regular expression.
- `-l` or `--files-with-matches`: list all files with lines that match the regular expression, but not their contents.



# Metacharacters

The following characters, called *metacharacters*, have a special meaning to grep:

`^ $ . [ ] { } - ? * + ( ) | \`

We will look at each in turn and what they do.

# 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
```

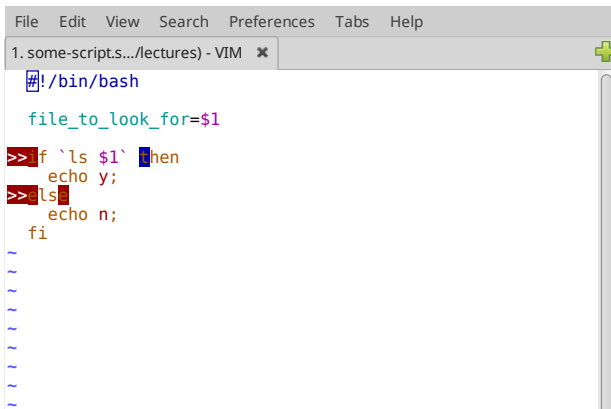
```
^-- SC1050: Expected 'then'.
```

```
^-- 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-scripts.s.../lectures) - VIM". The editor window displays a Bash script with the following content:

```
#!/bin/bash

file_to_look_for=$1

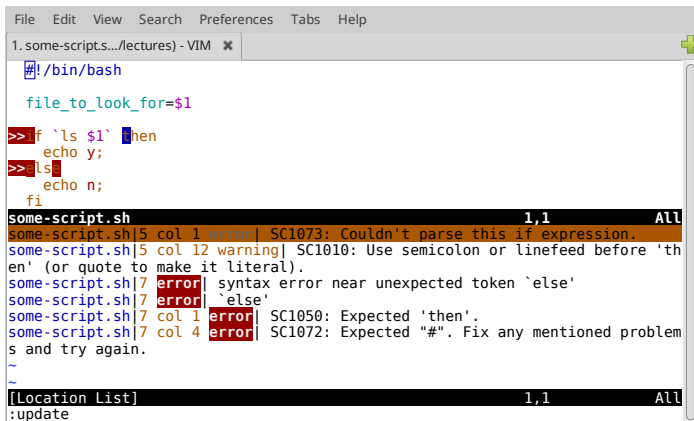
>> if `ls $1` then
    echo y;
>> else
    echo n;
fi

~
~
~
~
~
~
~
```

The script is syntax-highlighted: the shebang is blue, the variable assignment is purple, 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 displays a shell script with syntax highlighting:

```
#!/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, showing a list of errors and warnings:

| File            | Line | Column | Type    | Message                                                                        |
|-----------------|------|--------|---------|--------------------------------------------------------------------------------|
| some-script.sh  | 1    | 1      | All     |                                                                                |
| 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                                |
| s               |      |        |         | and try again.                                                                 |
| ~               |      |        |         |                                                                                |
| ~               |      |        |         |                                                                                |
| [Location List] | 1    | 1      | All     |                                                                                |
| :update         |      |        |         |                                                                                |

# 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.

# CITS4407 Open Source Tools and Scripting

## Shell functions and script design

Unit coordinator: Arran Stewart

# Overview

This week:

- Regular expressions

## Regular expressions, cont'd

# Regular expressions

We've mentioned one tool that makes use of regular expressions, `grep`: it looks for lines in a file that match some pattern.

```
$ ls /bin | grep ^b
bash
btrfs
bunzip2
busybox
bzipcat
bzipcmp
bzdiff
bzegrep
bzexe
# ... more lines omitted
```

# Special characters in regular expressions

We said that the following characters, called *metacharacters*, have a special meaning to grep:

`^ $ . [ ] { } - ? * + ( ) | \`

## Special characters in regular expressions

We've seen that the caret character, “^” means “at the start of the line” – `grep ^b` means, “print all lines beginning with the character b”.

Other special characters:

- “\$” means “at the end of the line”
  - `grep s$` means ‘print lines ending with the character “s”’.

```
$ ls /bin | grep 's$'
btrfs
bzless
less
ls
ps
ss
zless
# ... more lines omitted
```

# Special characters in regular expressions

```
$ ls /bin | grep 's$'  
btrfs  
bzless  
less  
ls  
ps  
ss  
zless  
# ... more lines omitted
```

(Note the use of single quotes – why might we use them?)



## Special characters in regular expressions – \$

What regular expression would match *only* the string “ls”?

## Special characters in regular expressions – .

A full stop matches any one character.

- `grep c.t` means 'print lines containing "c", some other character, then "t" '.

```
$ ls /bin | grep 'c.t'  
bzcat  
cat  
netcat  
ntfscat  
ntfsfallocate  
ntfstruncate  
zcat
```

## Special characters in regular expressions – \*

- An asterisk (“Kleene star”) means “zero or more of the previous item”

```
$ ls /bin | grep '^b.*'  
bash  
btrfs  
bunzip2  
busybox  
bzip2  
bzip  
bzipdiff  
bzipgrep  
bzipexec  
# ... more lines omitted
```

So `grep '^b.*'` should actually give us identical results to `grep '^b'` – why?

## Special characters in regular expressions

Square brackets match a set or range of characters.

- `ls /bin | grep '^[bcd]'` finds commands starting with b, c or d.
- `ls /bin | grep '^[b-d]'` does the same.

## Special characters in regular expressions

Originally, grep only used the metacharacters we've mentioned, but later, others were added:

- | – match one thing OR another
- ? – match zero or one times
- + – match one or more times
- {n} – (where n is a number) match exactly n times
- {n,} – match n or more times
- {,n} – match at most n times
- {n,m} – match from n to m times

## Special characters in regular expressions

Because they were added later, these metacharacters are treated a bit differently. To use them we need to either

- use `grep` with the `-E` option, meaning “use *extended* regular expressions”, or
- put a backslash in front of the special characters, so `grep` knows they have a special meaning.

# Examples

- How can we find commands in `/bin` whose names are exactly three letters long, and start with “c”?

# Examples

- How can we find commands in `/bin` whose names are exactly three letters long, and start with “c”?
- How can we find commands in `/bin` whose names are exactly four letters long, and end with a letter from the range “d” through “g”?



# Examples

- How can we find commands in `/bin` whose names are exactly three letters long, and start with “c”?
- How can we find commands in `/bin` whose names are exactly four letters long, and end with a letter from the range “d” through “g”?
- How can we find commands in `/bin` whose names match *either* of the criteria above?

## sed – the “stream editor”

We can think of `grep` as being a little like the “find” functionality in a word processor or browser – it finds lines matching a pattern.

Is there an equivalent of “find and replace”?

# sed

There is – sed, the “stream editor”.

grep takes a single pattern to search for.

But sed takes two: a pattern to search for, and a string to replace it with.

## sed

```
$ ls /bin | grep '^c' | sed 's/^c/d/'  
dat  
dhac1  
dhgrp  
dhmod  
dhownd  
dhvt  
dp  
dpio
```

## sed

```
$ ls /bin | grep '^c' | sed 's/^c/d/'  
dat  
dhac1  
dhgrp  
dhmod  
dhownd  
dhvt  
dp  
dpio
```

The “s” means to search for the regular expression `^c`, and replace it with the letter `d`.

## sed

Conventionally, the forward slash ("/") is used to separate patterns, but we can use any character – handy if the forward slash turns up within the text we're trying to match.

```
$ ls /bin | grep '^c' | sed 's|^c|d|'  
dat  
dhacł  
dhgrp  
dhmod  
dhowñ  
dhvt  
dp  
dpio
```

## sed

```
$ ls /bin | grep '^c' | sed 'sZ^cZdZ'  
dat  
dhacĹ  
dhgrp  
dhmod  
dhowñ  
dhvt  
dp  
dpio
```

For clarity, it's usually best to stick to “/” or “|”.

You can also add g at the end to mean “search and replace, multiple times”.

- `sed s/aa/bb/g`

# Functions with regexes

Can we write a function which gives us a new command, `extension-rename`, which looks for files matching some particular extension, and renames them so they have another?



# Functions with regexes

A start: let's just print files matching some extension.

myfunctions.sh

```
extension_rename () {  
    orig_ext=$1  
    new_ext=$2  
    for file in *.${orig_ext}; do  
        echo $file;  
    done  
}
```

How does this work?

# Functions with regexes

Trying it out:

```
$ source myfunctions.sh  
$ extension_rename pdf  
lect01.pdf  
lect04.pdf  
lect05.pdf  
lect06.pdf
```

# Functions with regexes

Now let's print the new name, as well.

myfunctions.sh

```
extension_rename () {  
    orig_ext=$1  
    new_ext=$2  
    for file in *.${orig_ext}; do  
        new_file=`echo $file | sed "s/${orig_ext}/${new_ext}/"`;  
        echo $file $new_file;  
    done  
}
```

## Functions with regexes

Trying it out:

```
$ source myfunctions.sh
$ extension_rename pdf pdx
lect01.pdf lect01.pdx
lect04.pdf lect04.pdx
lect05.pdf lect05.pdx
lect06.pdf lect06.pdx
```

(There are actually ways of getting Bash to do what sed is doing here – look in the Bash Reference Manual, [§3.5.3 “Shell Parameter Expansion”](#). This would run faster, for very very large numbers of files; but we will stick to sed for the moment.)

## Functions with regexes

Finally, let's use the `mv` command to rename each file from the old name to the new name:

myfunctions.sh

```
extension_rename () {  
    orig_ext=$1  
    new_ext=$2  
    for file in *.${orig_ext}; do  
        new_file=`echo $file | sed "s/${orig_ext}/${new_ext}/"`;  
        mv $file $new_file;  
    done  
}
```

# Functions with regexes

Trying it out:

```
$ source myfunctions.sh
$ extension_rename pdf pdx
$ ls *pdf
ls: cannot access '*pdf': No such file or directory
$ ls *pdx
lect01.pdx  lect04.pdx  lect05.pdx  lect06.pdx
```

## Program size

In fact, we could put the body of our function in a script, `extension_rename.sh`, and use it that way:

```
extension_rename.sh
```

```
#!/bin/bash

orig_ext=$1
new_ext=$2
for file in *.${orig_ext}; do
    new_file=`echo $file | sed "s/${orig_ext}/${new_ext}/"`;
    mv $file $new_file;
done
```

The whole program takes 5 lines of code – small scripts are often shorter and more convenient to write than coding in Python, and *much* shorter than writing a program in a compiled language like Java or C.

## Other sed features

sed has many other features.

We can ask it to do a search and replace, but only on line 4, for instance:

```
sed '4s/dog/cat'
```



## Other sed features

For instance:

```
$ for ((i=0; i<5; i=i+1)); do echo $i dog; done | sed '4s/dog/cat/'  
0 dog  
1 dog  
2 dog  
3 cat  
4 dog
```

What is this doing?

## Search and replace in vi and vim

We will not cover it in detail, but vi and vim use much the same syntax for doing “search and replace”.

In either of these, the command

```
:%s/dog/cat/g
```

means “in the file we are currently editing – on all lines of the file, replace every occurrence of “dog” with “cat”.

# CITS4407 Open Source Tools and Scripting Conditionals

Unit coordinator: Arran Stewart

# Overview

This week:

- Assignment questions
- Conditionals

# Conditionals in Bash

We've seen that Bash has “if” statements and “while” loops – both of these use what are often called *conditional expressions* – expressions which evaluate in some sense to being “true” or “false”.

```
if conditional ; then  
    statement 1;  
    statement 2;  
fi
```

```
while conditional ; do  
    statement 1;  
    statement 2;  
done
```

# Conditionals in Bash

What is actually happening when Bash encounters a conditional in an “if” statement? What does it expect to see in that spot?

The answer is, it expects to see a *command*, which succeeds or fails.

## Commands in conditionals

So the following will print “no such directory” (assuming the directory /xxxx doesn’t exist):

```
if ls /xxx; then
  echo it exists;
else
  echo no such directory;
fi
```

## Commands in conditionals

```
if ls /xxx; then
    echo it exists;
else
    echo no such directory;
fi
```

The command “ls” succeeds when it is able to list something, and fails when it can’t.

In Bash, *every* command can “succeed” or “fail”.



# Exit codes

The way an external command tells the operating system whether it succeeded or failed is by returning an *exit code*.

In Unix-like systems, an exit code of 0 means “success”, and anything else means “failure”.

For instance, `grep` returns “success” when it finds matching lines, and “failure” when it doesn’t.

(However, it *also* returns “failure” when something else went wrong – for instance you tried to `grep` for a pattern in a file that doesn’t exist.)

# Exit codes

Linux has two programs that do nothing but return an exit code:

- The “true” program always returns an exit code of 0
- The “false” program always returns an exit code of 1

```
$ if true; then echo hi there; fi  
hi there
```

## Examining exit codes

Bash stores the exit code of the most recently executed command in a special variable called “\$?” (it allows you to *query* the most recent exit code, hence the question mark).

```
$ true
$ echo $?
0
$ false
$ echo $?
1
```

## Exit codes and non-external commands

In Bash, *every* command has an exit code – not just external programs, but also built-in commands and user-defined functions.

When defining a function, you can use the “return” statement to specify the exit value of your function.

```
always_fails () {  
    return 1;  
}
```

## Exit codes and non-external commands

```
always_fails () {  
    return 1;  
}
```

If you don't specify a return value, the exit value of the function will be that of the last command it executes. So the following function is equivalent to the one above.

```
always_fails () {  
    false;  
}
```

# Exit codes

Even built-in commands like “declare” (which can be used to explicitly declare variables) have an exit code.

```
$ declare myvar=0
$ echo $?
0
$ declare 000=0
bash: declare: `000=0': not a valid identifier
$ echo $?
1
```

## Arithmetic expansion

In Bash, an arithmetic expression inside double brackets (“( (” and “) )”) also has an “exit code”.

```
$ (( 1 == 1 && 2 == 2 ))  
$ echo $?  
0  
$ (( 1 > 10 ))  
$ echo $?  
1
```

It exits with 1 (failure) if the expression inside evaluates to “false” or “0”, and 0 (success) otherwise.

## Arithmetic expansion

If you have several conditionals you want to check, joined with “and” or “or” – you might want to see if you can write them using arithmetic expansion, which is often more convenient than using square brackets (“[” and “]”).

```
if [ "$var1" -eq 1 ] && [ "$var2" -eq 1 ] && [ "$var3" -eq 1 ] ; then
    echo "correct";
fi
```

*versus,*

```
if (( var1 == 1 && var2 == 1 && var3 == 1 )) ; then
    echo "correct";
fi
```



## Square brackets (test)

The left-square-bracket (“[”) is just another command, as far as Bash is concerned. In fact, it’s usually available as an external program:

```
$ which [  
/usr/bin/[]
```

## Square brackets (test)

The square-bracket command takes multiple arguments, and expects the last argument to be a right-square-bracket (“]”). And it then exits with success or failure depending on its interpretation of those arguments.

```
$ [ -d /tmp -a -d / ]  
$ echo $?  
0
```

```
$ [ -d /tmp -a -d /xxx ]  
$ echo $?  
1
```

## Square brackets (test)

However, because it's so frequently used, Bash also defines a built-in command called “[”, and this will normally get called instead of the external program.

If you absolutely wanted to use the external program, you would have to write:

```
$ /usr/bin/[ -d /tmp -a -d /xxx ]  
$ echo $?  
1
```

This explains why Bash (or rather, the “[” command) is so picky about spacing – it needs spaces to tell it where different arguments start and end, and looks for “]” as its last argument.

## Square brackets (test)

The `/usr/bin/[]` program also comes in a variant called `/usr/bin/test`, which doesn't take any square brackets.

```
$ test -d /tmp -a -d /xxx  
$ echo $?  
1
```

# CITS4407 Open Source Tools and Scripting

## Writing in Markdown

Unit coordinator: Arran Stewart

# Overview

This week:

- Creating documents using Markdown

## CITS4407 documents and web pages

All the documents created for this course – web pages, lecture slides, “README” files, lab worksheets, and assignment specs – weren’t written using a word processor or any specialised tool, but just in the `vim` text editor, in a format called Markdown.

# What is Markdown?

Markdown specifies a way of writing text files so that you can indicate whether text is intended to be bold, italic, a heading, a link to a webpage, and so on.



# What is Markdown?

Markdown specifies a way of writing text files so that you can indicate whether text is intended to be bold, italic, a heading, a link to a webpage, and so on.

In a word processor like Microsoft Word, you can format paragraphs or portions of text using menus, buttons and shortcut keys. Once you do that, the formatting you've applied is immediately visible on-screen.

This is called WYSIWYG (“what you see is what you get”) formatting.

But Markdown works differently to that.

# What is Markdown?

When writing a document in Markdown format, you use special syntax to indicate how words and phrases should look when formatted.

The syntax is intended to be “intuitive” – reasonably similar to the sorts of informal “formatting” conventions people use when writing plain text files and messages. (For instance, you may see people use this in Help4407 messages.)

As an example – to indicate text should be bold, in Markdown, you put it between double asterisks:

```
**some bold text**
```

# Markdown syntax

Some other sorts of formatting you can indicate in Markdown include:

- italic - surrounded by single asterisks

You should *\*always\** remember to floss

- top-level heading - prefixed with a hash character

# My summer holiday

# Markdown syntax

- Bullet points - each item prefixed with an asterisk or hyphen

```
- one fish  
- two fish  
- red fish  
- blue fish
```

# Markdown syntax

- Numbered lists - first item prefixed with a number, other items with a number or hash

I sampled the defendant's wares:

1. on a boat
- #. with a goat

# Markdown syntax

This special syntax is called *markup*, and Markdown is called a *markup language*.

The term “markup” comes from the practice of editors “marking up” author’s typewritten manuscripts with a red or blue pencil to indicate to typesetters how the text should be formatted.

# Markup languages

Other sorts of Markup language exist. For instance, HTML is the Markup language used in web pages sent from a webserver to your browser.

In most browsers, there should be an option for “viewing code” of a web page. Doing that for Wikipedia’s page on “Markup languages” shows that the web page starts with content like this:

```
<!DOCTYPE html>
<html class="client-nojs" lang="en" dir="ltr">
<head>
<meta charset="UTF-8"/>
<title>Markup language - Wikipedia</title>
<script>document.documentElement.className="client-js";RLCONF={
"wgBreakFrames":!1,"wgSeparatorTransformTable":["",""],
"wgDigitTransformTable":["",""],"wgDefaultDateFormat":"dmy","wgMonthNames":
```

# Markup languages

In HTML, one can indicate a portion of text should be displayed in bold as follows:

```
<b>some bold text</b>
```



# Markup languages

Another markup language is Rich Text Format (RTF), devised by Microsoft for use in MS Word.

Text intended to be bold looks something like this:

```
{\pard \ql \f0 \sa180 \li0 \fi0 {\b some bold text}\par}
```

Markdown, however, tries to keep its syntax as readable as possible, and as close as possible to informal conventions for writing plain text.

# Processing Markdown

So how do we turn Markdown into web pages, or lecture slides, or PDF documents?

We need to use what's called a Markdown *processor*, which turns a text file written in Markdown into some other format.

# Processing Markdown

Sometimes, the processing is done for us “behind the scenes”.

For instance, if you write comments on the social media platform Reddit, the Reddit servers handle the task of turning your comments into HTML that can be displayed on the website.

all 5 comments

sorted by: best ▼

Hi, I'm having the same problems with a Bash script - I've tried  
\*everything\*!

save

[content policy](#) [formatting help](#)

# Processing Markdown

Likewise, if we create a website using [GitHub Pages](#) (which is how the website for CITS4407 is created), GitHub's server will turn our Markdown into HTML:

```
---  
title: "Resources"  
tags: ['toppage']  
layout: page  
---
```

Lecture slides and lab exercises will appear here as the semester progresses.

All material used in the unit is available online. Readings for each week are given in the  
[\*\*schedule\*\*]({{ "/schedule" | relative\_url }}){: class="hi-pri" :}.

# Processing Markdown

And some online Markdown editors like <https://dillinger.io> do something similar:

The screenshot displays the Dillinger online Markdown editor. The interface is split into two main sections: a source code editor on the left and a live preview on the right.

**Source Code Editor (Left):**

- Line 1: `1 * Dillinger`
- Line 2: `2 * ## _The Last Markdown Editor, Ever_`
- Line 3: (empty)
- Line 4: `4 [![N|Solid](https://cloudup.com/dTwpP1910f.thumb.png)](https://nodesource.com/products/nsolid)`
- Line 5: (empty)
- Line 6: `6 [![Build Status](https://travis-ci.org/joemccann/dillinger.svg?branch=master)](https://travis-ci.org/joemccann/dillinger)`
- Line 7: (empty)
- Line 8: `8 Dillinger is a cloud-enabled, mobile-ready, offline-storage compatible,`
- Line 9: `9 AngularJS-powered HTML5 Markdown editor.`
- Line 10: (empty)
- Line 11: `11 - Type some Markdown on the left`
- Line 12: `12 - See HTML in the right`
- Line 13: `13 - Magic ✨`

**Live Preview (Right):**

- Header: 

## Dillinger
- Section Header: 

### The Last Markdown Editor, Ever
- Powered by: **POWERED BY N|Solid**
- Build Status: **build passing**
- Text: 

Dillinger is a cloud-enabled, mobile-ready, offline-storage compatible, AngularJS-powered HTML5 Markdown editor.
- List of instructions:
  - Type some Markdown on the left
  - See HTML in the right
  - Magic ✨

# Processing Markdown

But for the most control, you can download and run a Markdown processor yourself – the one we will use in lab/workshops is called *Pandoc* (<https://pandoc.org>).

## About pandoc

---

If you need to convert files from one markup format into another, pandoc is your swiss-army knife. Pandoc can convert between the following formats:

(← = conversion from; → = conversion to; ↔ = conversion from and to)

### Lightweight markup formats

- ↔ Markdown (including CommonMark and GitHub-flavored Markdown)
- ↔ reStructuredText
- AsciiDoc
- ↔ Emacs Org-Mode
- ↔ Emacs Muse
- ↔ Textile
- ← txt2tags

### HTML formats

- ↔ (X)HTML 4
- ↔ HTML5

### Ebooks

- ↔ EPUB version 2 or 3
- ↔ FictionBook2

### Word processor formats

- ↔ Microsoft Word docx
- ↔ OpenOffice/LibreOffice ODT
- OpenDocument XML
- Microsoft PowerPoint

### Interactive notebook formats

- ↔ Jupyter notebook (ipynb)

### Page layout formats

- InDesign ICML

### Wiki markup formats

- ↔ MediaWiki markup
- ↔ DokuWiki markup
- ← TikiWiki markup
- ← TWiki markup
- ← Vimwiki markup

# Processing Markdown

We'll see in labs how you can create your own webpages, PDF documents, or even e-books using Pandoc.

In the meantime, you can find out more about what Markdown looks like by looking at how it's used for the assignment README – <https://raw.githubusercontent.com/cits4407/assignment1/master/README.md>

```
# CITS4407 Assignment - Semester 1, 2021
```

```
## Details
```

```
Version: 0.3
```

```
Date: 2021-04-21
```

```
Changelog:
```

- 2021-04-07 - initial version
- 2021-04-19 - minor corrections and clarifications
- 2021-04-21 - submission procedure

```
Please check the CITS4407 website or the Git repository
```

# CITS4407 Open Source Tools and Scripting

## Build management

Unit coordinator: Arran Stewart



# Overview

This week:

- make and Makefiles

# make and Makefiles

The make tool is used for *building* things on Unix-like systems, using “recipes” contained in *Makefiles*.

# make and Makefiles

The make tool is used for *building* things on Unix-like systems, using “recipes” contained in *Makefiles*.

What sort of “things”?

They could be:

- PDF documents reporting on the results of experiments
- complex programs, built from multiple files or packages
- web sites (the CITS4407 site is tested using [a Makefile](#))
- Docker images, like the one used for Assignment 1
- virtual machines

or almost any other sort of complex software artifact.

# Makefiles

The recipes are contained in plain text files, conventionally named Makefile, and at their simplest look something like this:

## Makefile

```
workshop09.pdf: workshop09.md
    pandoc --from markdown --to html \
        --pdf-engine=weasyprint \
        --output=workshop09.pdf workshop09.md
```

# Makefile rules

```
workshop09.pdf: workshop09.md
    pandoc --from markdown --to html \
        --pdf-engine=weasyprint \
        --output=workshop09.pdf workshop09.md
```

Makefile contain what are called “rules” for building things; their syntax is

```
output: ingredient1 ingredient2 ...
→ command1
→ command2
→ command3
→ ...
```

# Makefile rules

```
output: ingredient1 ingredient2 ...  
→command1  
→command2  
→command3  
→...
```

The rule specifies:

- something we want to make (the output);
- what ingredients we need to make it; and
- what commands we need to run, to turn those ingredients into the output.

The arrows indicate “tab” characters. In a Makefile, the instructions for *how* to build the output always start with a tab character. (But we will not show them from now on.)

# Makefile rules

```
workshop09.pdf: workshop09.md
  pandoc --from markdown --to html \
    --pdf-engine=weasyprint \
    --output=workshop09.pdf workshop09.md
```

Here, `workshop09.pdf` is the thing we want to build; `workshop09.md` is the ingredient we need in order to build it (a Markdown file); and we use the `pandoc` tool to convert from Markdown format to PDF.

# Running make

## Makefile

```
workshop09.pdf: workshop09.md
  pandoc --from markdown --to html \
    --pdf-engine=weasyprint \
    --output=workshop09.pdf workshop09.md
```

To run make, we simply give the name of some output we want to build – in this case,

```
$ make workshop09.pdf
```



# Build management tools

make is an example of a *build-management* (or *build automation*) tool.

Many others exist, but make is one of the oldest (originally created by Stuart Feldman in April 1976 at Bell Labs)<sup>1</sup> and most widely used.

---

<sup>1</sup>[https://en.wikipedia.org/wiki/Make\\_\(software\)#Origin](https://en.wikipedia.org/wiki/Make_(software)#Origin) 

# Dependencies

## Makefile

```
workshop09.pdf: workshop09.md
    pandoc --from markdown --to html \
        --pdf-engine=weasyprint \
        --output=workshop09.pdf workshop09.md
```

In build automation terminology, we say that `workshop09.md` is a *dependency* of `workshop09.pdf`, and that `workshop09.pdf` is a *target*.

# Dependencies

If all make did was run a set of commands to build something, there would be no great advantage to using it over a Bash script.

However, it *also* tracks whether any of the dependencies (and *their* dependencies, and so on – there could be hundreds, in a large Makefile) have changed and are *newer* than the the output.

If so, it knows the recipe needs to be re-run; if not, it knows the output is up to date.

# Dependencies

## Makefile

```
workshop09.pdf: workshop09.md
    pandoc --from markdown --to html \
        --pdf-engine=weasyprint \
        --output=workshop09.pdf workshop09.md
```

```
$ make workshop09.pdf
make: 'workshop09.pdf' is up to date.
$ touch workshop09.md
$ make workshop09.pdf
pandoc --from markdown --to html --pdf-engine=weasyprint
```

# Dependencies

If we are building a very large software program, or a complex output of some other sort, make ensures that

- we don't re-build things if we don't have to – this can save a great deal of time
- we *do* re-build things when their dependencies have changed

# Generic recipes

What if have multiple files, that are all built in the same general way?

For instance,

- workshop06.pdf
- workshop07.pdf
- workshop08.pdf

... and so on.

## Generic recipes

If using pandoc to turn Markdown files into PDFs, we'll always need to run a command of the form

```
pandoc --from markdown --to html \  
  --pdf-engine=weasyprint \  
  --output=output.pdf input.md
```

but the files *output.pdf* and *input.md* will vary.

# Generic recipes

make lets us write *generic* recipes, which use the following “special variables”:

- $\$@$  – the current target file
- $\$^$  – all dependencies listed for the current target
- $\$<$  – the first (left-most) dependencies for the current target

(There are many more special variables, but these are the most common.)



# Generic recipes

Using special variables, we can create a rule that looks like this:

## Makefile

```
%.pdf: %.md
    pandoc --from markdown --to html \
        --pdf-engine=weasyprint \
        --output=$@ $<
```

We can read this as saying:

“To build a PDF file, look for a Markdown file of the same base name, but ending in”.md“.

Run the pandoc command specified in the recipe, but after --output put the name of the output file we're producing, and as a final argument, put the name of the left-most dependency.”

# Generic recipes

## Makefile

```
%.pdf: %.md
    pandoc --from markdown --to html \
        --pdf-engine=weasyprint \
        --output=$@ $<
```

(In this case, there's only one dependency – but we could have more, for instance, if our Markdown document refers to images.)

# Makefile variables

The term “special variables” suggests make has the concept of *ordinary* variables as well – and it does.

Let's see how we might use them.

We might want to say, in order to run pandoc, you first have to fetch the pandoc binary from the Internet.

## Makefile

```
pandoc:
```

```
wget https://github.com/jgm/pandoc/releases/download/2.13/pandoc-2.13-linux-amd64
```

```
tar xf pandoc-2.13-linux-amd64.tar.gz
```

```
mv ./pandoc-2.13/bin/pandoc .
```

The URL where the pandoc binary is located is lengthy, and also might change if we start to use a new version.

# Makefile variables

So make lets us put fragments of text in *variables*;  
once they are defined, we can *expand* them using the syntax `$(var)`.

(This is similar to, but not quite the same as, the expansion syntax Bash uses. Why might we not want to use the same syntax?)

## Makefile

```
PANDOC_URL=https://github.com/jgm/pandoc/releases/download/2.13/pandoc-2.13-linux-
```

```
pandoc:
```

```
  wget $(PANDOC_URL)  
  tar xf pandoc-2.13-linux-amd64.tar.gz  
  mv ./pandoc-2.13/bin/pandoc .
```

# Writing your own Makefiles

In the lab/workshops, there will be opportunity to write your own Makefiles – and we will use them in Assignment 2.

# CITS4407 Open Source Tools and Scripting

## Network tools

Unit coordinator: Arran Stewart

# Overview

This week:

- communicating over a network

Note that this lecture gives a *simplified* explanation of networking concepts.

For a fuller one, you should take a look at the materials for a unit like CITS3002 “Networks and Security”.

## Commands that use the network

We've already seen a number of commands that contact other systems over the Internet.

- Whenever we run `apt-get update` the `apt-get` command contacts computers holding copies of Ubuntu's software repositories, and fetches the current list of available packages.
- When we run `apt-get install`, the `apt-get` command download copies of the software packages we want.
- `git clone`, when we give it a URL like <https://github.com/cits4407/assignment1.git>, contacts GitHub's servers to fetch a copy of the repository we're cloning.
- The command `wget` (used in labs) retrieves copies of files or pages on Web servers.
- Whenever we visit a website in a web browser, copies of the pages we see are being fetched from remote servers.



# Hosts, domains and IP addresses

Everyone should be familiar with what a web address looks like:

<https://cits4407.github.io/resources/>

A web address is formally known as a *URL* (Uniform Resource Locator).

# URLs

A URL contains the following parts:

- `cits4407.github.io` – this specifies a *host*: a system that can be contacted over the Internet. We can think of `cits4407.github.io` as being an address, that tells us how to get to that host.
- `.github.io` – this specifies a *domain*: a collection of hosts. (We could think of this as being like a suburb or region, for an address.)
- `/resources/` – this specifies the location of a particular file or directory on the host.
- `https://` – this specifies a particular **protocol** – like a common language spoken by your computer and the remote one – which should be used when contacting the host and asking for a particular file or directory.

# IP addresses

In fact, hostnames like `cits4407.github.io` are just a *human-readable* way of referring to a computer on the Internet.

Your computer's operating system knows how to convert a human-readable hostname like `cits4407.github.io` into an *IP address* – an address that is convenient for *computers* to use when locating a remote host.

They look like this:

- `172.217.25.132` (an Internet Protocol Version 4 address)

or this:

- `fe80::7ef3:e931:830c:6471` (an Internet Protocol Version 6 address)

## Tools for working with remote hosts – ping

It can often be helpful to know how to diagnose network problems using Linux tools.

One of the most fundamental tools is `ping`.

It sends a very small network message to a remote host, and asks that host to respond.

```
ping www.google.com
```

will try and contact the host `www.google.com`.

# ping

If the remote host is contactable, ping will produce a report like this:

```
arran@barkley:~$ ping www.google.com
PING www.google.com (172.217.25.132) 56(84) bytes of data.
64 bytes from syd15s03-in-f4.1e100.net (172.217.25.132): icmp_seq=1 ttl=113
time=48.2 ms
64 bytes from syd15s03-in-f4.1e100.net (172.217.25.132): icmp_seq=2 ttl=113
time=48.1 ms
64 bytes from syd15s03-in-f4.1e100.net (172.217.25.132): icmp_seq=3 ttl=113
time=48.4 ms
^C
--- www.google.com ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2002ms
rtt min/avg/max/mdev = 48.156/48.278/48.471/0.226 ms
```

Just as web browsers use a particular protocol or “language” to talk to remote hosts, so does ping – it uses ICMP (Internet Control Message Protocol).

You may not *always* be able to use ping and ICMP to contact a remote host, even if it is “up”: sometimes, network administrators will block the use of ICMP, on the grounds that only malicious hackers would want to know whether a remote host is up.

# traceroute

The traceroute command reports whether a remote host could be contacted – but it also lists all the “hops” taken by network traffic as it goes from our local system to a remote host.

```
arran@beaker:~$ traceroute www.google.com
traceroute to www.google.com (142.250.70.228), 30 hops max, 60 byte packets
 1  kermit.lan (192.168.10.1)  0.285 ms  0.337 ms  0.429 ms
 2  10.20.26.6 (10.20.26.6)  7.504 ms  8.451 ms  8.422 ms
 3  203.29.134-193.tpgi.com.au (203.29.134.193)  9.298 ms  9.269 ms
10.089 ms
 4  nme-sot-dry-crt1-Be40.tpgi.com.au (203.219.107.225)  50.808 ms
50.779 ms  51.714 ms
 5  27-32-160-69.static.tpgi.com.au (27.32.160.69)
50.762 ms 27-32-160-5.static.tpgi.com.au (27.32.160.5)  51.736 ms
51.695 ms
 6  72.14.213.0 (72.14.213.0)  53.631 ms  51.545 ms  52.374 ms
 7  * * *
 8  mel05s02-in-f4.1e100.net (142.250.70.228)
47.809 ms 216.239.54.50 (216.239.54.50)  48.874 ms
...
```

# Diagnosing network problems

If you are unable to run commands that make use of the network (`git clone`, `wget` and so on) – Service Desk or teaching staff will often ask you to run a command like `ping` so they can tell whether the problem is with the command (maybe `git` has a bug) or your computer.



# localhost

Unix-like systems use the name `localhost` to refer to the local system.

You can ping the localhost:

```
arran@barkley:lectures$ ping localhost
PING localhost (127.0.0.1) 56(84) bytes of data.
64 bytes from localhost (127.0.0.1): icmp_seq=1 ttl=64 time=0.066 ms
64 bytes from localhost (127.0.0.1): icmp_seq=2 ttl=64 time=0.069 ms
^C
--- localhost ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1004ms
rtt min/avg/max/mdev = 0.066/0.067/0.069/0.008 ms
```

# localhost

However, the `ping` command in this case isn't *really* using the network.

Messages sent to `localhost` never actually leave your computer.

# Retrieving a web page

We will examine what happens when you visit a web page like <https://cits4407.github.io/resources/>.

On Linux, a good way to see what is going on is to run

```
curl -v https://cits4407.github.io/resources/ 2>&1 | less
```

# Retrieving a web page

```
curl -v https://cits4407.github.io/resources/ 2>&1 | less
```

% Total	% Received	% Xferd	Average Speed	Time	Time	Time	
Current			Dload	Upload	Total	Spent	Left

Speed

^M	0	0	0	0	0	0	0
----	---	---	---	---	---	---	---

0 --:--:-- --:--:-- --:--:-- 0\* Trying 185.199.109.153...

\* Connected to cits4407.github.io (185.199.109.153) port 443 (#0)

\* found 129 certificates in /etc/ssl/certs/ca-certificates.crt

\* found 521 certificates in /etc/ssl/certs

\* ALPN, offering http/1.1

\* SSL connection using TLS1.2 / ECDHE\_RSA\_AES\_128\_GCM\_SHA256

\* server certificate verification OK

\* server certificate status verification SKIPPED

\* common name: www.github.com (matched)

\* server certificate expiration date OK

\* server certificate activation date OK

\* certificate public key: RSA

\* certificate version: #3

\* subject: C=US,ST=California,L=San Francisco,O=GitHub\, Inc.,CN=www.github

\* start date: Wed, 06 May 2020 00:00:00 GMT