

# CITS4407 Open Source Tools and Scripting

## Semester 1, 2021

### Week 3 workshop – Version control, pipelines and redirection

*Before starting this workshop, make sure you’ve reviewed the recommended reading for weeks 1–2, and completed the Week 2 lab sheet.*

You should now have some way of accessing a Linux environment for your work in this unit – let one of the lab facilitators know if not.

The reading for weeks 1–2 includes some simple exercises, which we won’t repeat here, but which you should work through if you haven’t already:

- Shotts chap 1, “Try Some Simple Commands”
- Shotts chap 2, “Navigation” (the whole chapter – it’s short)

## 0. The superuser and `sudo`

Most operating systems allow for multiple different *users* to exist on a computer system, and Linux is one of these – many different users can all be logged in at once, and performing separate tasks.

When you log into a Linux environment (or open a Linux terminal using Windows WSL2), you will use a specified *user ID* and *password*. On the VirtualBox images we recommend, the user ID and password are just “ubuntu” and “ubuntu”. On the Windows WSL2, you select a user ID and password when you install Ubuntu.

Most of the tasks you’ll need to perform in this unit can be done using this user ID and password, but sometimes you’ll need to make changes to the way the system is configured, and this requires what is called *superuser access*.

In Linux (and in Unix-like systems in general), a user ID exists called “**root**” – this is the *superuser*. (For Windows users, the **root** user is somewhat similar to the “Administrators” group on Windows.)

Typically, Linux systems are configured so that you can’t log into the system as **root**; but you *can* temporarily “become” root for long enough to run a single program. A frequent reason to do this is to install new software, and the command which lets us do this is called “**sudo**” (short for “**Superuser Do**”).

For this lab, you’ll need to ensure that the **git**, **nano** and **vim** programs are installed. On most Ubuntu systems, they already should be, but it is possible they are not.

Try entering the follow in the terminal:

```
$ which git
$ which nano
$ which vim
```

In each case, if the program exists on your Linux system, the **which** command should print out the path to that program on the filesystem. If the program doesn't exist, **which** prints nothing.

If any of the programs don't exist, try typing:

```
$ sudo apt update
```

You'll be prompted for your password, and if you type it correctly, the “**apt**” program (which is used to install and uninstall software packages on Ubuntu; it's name is an abbreviation for “**A**dvanced **P**ackage **T**ool”) may produce quite a bit of output; it is checking with remote servers to ensure it has up-to-date lists of what software can be installed, and should end with a message something like:

```
Fetchd 14.3 MB in 31s (453 kB/s)
Reading package lists... Done
Building dependency tree
Reading state information... Done
33 packages can be upgraded. Run 'apt list --upgradable' to see them.
```

Once it has completed, type

```
$ sudo apt install git nano
```

to ensure all the software you need is installed.

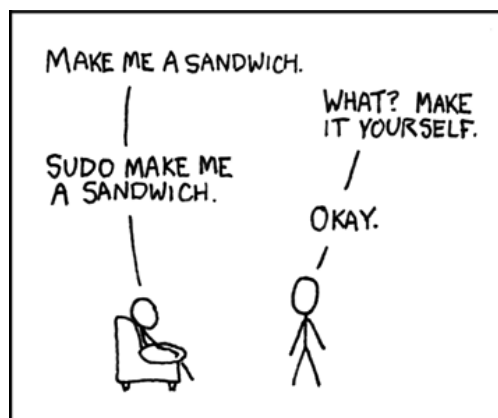


Image credit: [xkcd](#)

## 1. Version control with Git

When completing assignments for CITS4407, we recommend you keep track of your files using the `git` program.

We will assume `git` is installed in your Linux environment.

The first step in using `git` is to tell `git` who you are:

```
$ git config --global user.name 'My Name'
$ git config --global user.email 'my.address@somewhere.com'
```

(Fill in your name, and any email address you like, in the appropriate spots.) When displaying who made what changes to what files, `git` will use this name and address to indicate changes made by you.

We will start by *cloning* an existing repository from GitHub. (This means that we make a copy of the repository, and keep a link to it, as a kind of “parent” repository.) Enter:

```
$ git clone https://github.com/cits4407/example.git
```

This will create a directory called “**example**”, and download the contents of the repository into it.

Try `cd`ing into the directory, and experimenting with the files there.

You should be able to run the program called `numbers` (one you are in the `example` directory) by entering:

```
$ ./numbers
```

Many programs we run on Linux are in the directory `/usr/bin/`, and when we log in, `bash` already knows to look in that directory (and several others) for programs to run. (We will see later how it does this; ask your lab facilitators about the `$PATH` environment variable if you want to know more.)

To run other programs, you have to tell `bash` exactly where they are.

On Linux, the full stop (“.”) means “the current directory”; and “`./numbers`” means “Run the `numbers` program, which is in the current directory.”

Try other tasks:

- Run the program `numbers2`
- View the contents of the `numbers`, `numbers2` and `README` files using `less`

Now add your own file to the project.

Enter:

```
$ echo 'My Name' > AUTHORS
$ git add AUTHORS
$ git commit -m "adding an AUTHORS file"
```

filling your name instead of “My Name”.

`echo` simply prints a string, or sequence of strings, to *standard output*. The “>” symbol *redirects* the output of the `echo` command into a file called `AUTHORS`. `git add` tells `git` to start tracking changes to this file, and `git commit` *commits* a change to the repository (i.e., makes a permanent record of it). (If you don’t enter the commit command exactly correctly, you will probably end up in the `nano` text editor; type `ctrl-X` to exit it, and try again.)

Now we’ll add another line to the `AUTHORS` file, and commit it. Enter the following:

```
$ echo 'Lab facilitator name' >> AUTHORS
$ git add AUTHORS
$ git commit -m "added my lab facilitator as co-author"
```

replacing ‘Lab facilitator name’ with the name of your lab facilitator.

This time, we use what’s called the *append operator*, written as “>>”, to redirect the output of `echo`. Using two angle brackets means the output of `echo` is added to the end of the `AUTHORS` file; if we only use one angle bracket, we would instead *replace* the content of the `AUTHORS` file.

Now try typing `git log`: you should see a brief description of three *commits* that have been made to the project:

- the initial commit, by the unit coordinator
- your first commit, adding the `AUTHORS` file
- a second commit, adding a co-author

Arguments to `git log` can be added to give you more detail – for instance `git log --> name-status` will, in addition to showing a brief description of each commit, show what files were added or deleted in that commit.

Now try entering

```
$ git checkout -b better-version HEAD~2
```

This rolls back the files in the repository to an earlier state. `HEAD` means the latest of the revisions; `HEAD~2` means two commit earlier than that. If you type `ls` and look at the contents of the directory, you should find the `AUTHORS` file no longer there.

Git has also created what’s called a *branch*, named “better-version”. If you now decided to make *different* changes to the files from last time, they would be recorded in this new “branch” of history.

Enter

```
$ git branch
```

You should be able to see two “branches”: one called “better-version” (the current branch, marked by an asterisk), and one called “master” (the original branch).

We can easily go back to the “master” branch: enter

```
$ git checkout master
```

and the files will be restored to the state they had before we created a new branch.

## Challenge tasks

Are on their way...