CITS4407 Open Source Tools and Scripting
Semester 1, 2021
Week 4 workshop – Editors, scripts, and control structures

Before starting this workshop, make sure you've reviewed the recommended reading for weeks 1–3, and completed the lab sheets for weeks 2–3.

0. Exercises from *Shotts*

The reading for weeks 1–3 includes some simple exercises, which we won't repeat here, but which you should work through if you haven't already:

- Shotts chap 24, "Writing your first script"
- Shotts chap 27, "Flow Control: Branching with `if`"

1. Editors

Text editors are programs which allow us to interactively edit text files – files containing human readable text. By contrast, *pagers* only allow us to *view* the contents of text files.

Editors can be *graphical* or *terminal-based*. An example of a graphical editor available on many Linux distributions is `gedit`. However, we will focus on using text-based editors, because:

- they are available on all Linux systems, even ones which lack a graphical environment
- they use very few resources, and thus can be used even on very constrained systems
- skills you learn when using a terminal-based editor can usually be applied to a graphical editor, but not necessarily vice versa.

Terminal-based text editors you may encounter on Linux systems include:

- `nano` – an easy-to-use editor but with limited features
- `vim` – a highly-configurable text editor based on the older `vi` editor
- `emacs` – a highly-configurable text editor which uses the Lisp programming language for configuration and scripting.

We will focus on using `nano` and `vim`. As described in the week 3 lab, you can check if `nano` and `vim` are installed by typing:

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

And you can install them (if they are not installed) by typing:

```
$ sudo apt update
$ sudo apt install nano vim
```

See the previous lab for details.

When programs such as `git` give you the option of editing a file, they will often look in an environment variable named `EDITOR` to see if you have a preferred editor.

See if the `EDITOR` variable is already defined by typing:

```
$ echo $EDITOR
```

Usually, the result will be a blank line, indicating the variable is not set.

We can ensure the EDITOR variable is set appropriately, every time we start Bash, by editing the file `.bashrc` in the `$HOME` directory. First of all, make a backup of your `~/.bashrc` file:

```
$ cp ~/.bashrc ~/.bashrc.bak
```

Then type:

```
$ nano ~/.bashrc
```

to edit the `~/.bashrc` file using the `nano` editor.

Challenge exercise – get started with the `vim` editor

A simple editor like `nano` is sufficient for our needs in this unit.

However, editors like `vi` and `vim` (**`vi` im**proved, which adds further features to `vi`) are much more configurable, and allow us to automate many editing tasks.

`vim` is *backwards-compatible* with `vi`: it can be configured to behave in exactly the same way as the older `vi` program (and on many systems, the executable binary for `vi` actually *is* in reality `vim`). Additionally, you can use any command from `vi` in `vim` as well.

Chapter 12 of the Shotts text provides an introduction to using `vi`, as does the "Getting Started with Vim" tutorial.

To exit `vi` or `vim` at any time, hit the ESC key, then type `:q!` and hit the ENTER key.

As a further challenge exercise: read about using mappings and abbreviations in `vim`, and see if you can construct a vim command which will, with a single keystroke, add a line at the end of the current file which sets the `EDITOR` environment variable to the value "`vim`".

Navigate to the end of the file – you can use the `Page Down` and arrow keys on your keyboard – and on a new line, type

```
EDITOR=nano
```

Then hit `ctrl-O` to save, and `ctrl-X` to exit.

Your `.bashrc` file, which Bash runs every time it starts, now sets the `EDITOR` environment variable. We can also run the `.bashrc` file even without re-starting Bash – type

```
$ source ~/.bashrc
$ echo $EDITOR
```

and you should see that the `EDITOR` environment variable is set to the new value.

The `source` command

If we have a file named `somefile` containing Bash commands, then how does `source somefile` differ from running `bash somefile` or `/path/to/somefile`?

In both cases, Bash reads and executes commands from the file. But when we use `source`, those commands are executed *as if we had typed them in the current shell.* That means that any changes we make to the contents of variables will persist after we have run `source`, until we exit Bash.

In the latter two cases, however, the commands are executed *in a new process* – a new "instance" of Bash is created to execute the file, and when it finishes, the contents of variables in our current Bash session will remain unchanged.

2. Control flow structures

Make a new directory (e.g. `mkdir week04-workshop`), and clone the example repository from the Week 3 lab/workshop into it:

```
$ git clone https://github.com/cits4407/example.git week04-workshop
```

`cd` into the directory, and edit the file `numbers2` using your preferred text editor.

This file contains one bash command, a `for` loop which prints the numbers 1 to 10, inclusive.

Try the following:

- Define a variable `upper_bound` in the file, before the `for` loop, and set its value to 10.
- Amend the `for` loop to use the value of your new `upper_bound` variable, instead of the literal number `10` that it currently uses.
- Save the file, and run the script to ensure it behaves the same as before.

In the `for` loop, what happens if you type `upper_bound` with a dollar sign at the start? How about without a dollar sign – is there any difference? (You can read more about *arithmetic expansion* in Chapter 34 of the Shotts text.)

Try replacing the `echo` command inside the `for` loop with an `if` command:

```
if ((i % 2 == 0)); then
  echo $i;
fi
```

The "`%`" sign is the *modulus* operator. `i % 2` returns the *remainder* when we divide `i` by 2. Run the script again – does it do what you would expect?

Try changing `numbers2` so it does the following:

- The loop executes for all numbers from 1 to 100, inclusive.
- If `i` is divisible by 3, instead of printing the number, print the word "fizz".
- If `i` is divisible by 5, instead of printing the number, print the word "buzz".

A solution that does *exactly* what the exercise asks for, and only uses simple `if` statements:

```
#!/bin/bash

for ((i=1; i<=100; i=i+1)); do
  if ((i % 3 == 0)) ; then
    echo "fizz"
  fi
  if ((i % 5 == 0)) ; then
    echo "buzz"
  fi
  if ((i % 3 != 0 && i % 5 != 0)); then
    echo $i;
  fi
done
```

When the number is divisible by three *and* five, the statement of the problem suggests we should print the word “fizz” and the word “buzz” – and this is exactly what the solution above does.

One can also write a solution using an `if-elif-else` statement (covered in the *Shotts* text):

```
#!/bin/bash

for ((i=1; i<=100; i=i+1)); do
  if ((i % 3 == 0 && i % 5 == 0)) ; then
    echo "fizz"
    echo "buzz"
  elif ((i % 3 == 0)) ; then
    echo "fizz"
  elif ((i % 5 == 0)) ; then
    echo "buzz"
  else
    echo $i;
  fi
done
```

Challenge exercises

Write a script, `process-count.sh`, which counts the total number of processes running on your system.

Commands you will probably want to use:

- `ps`, to show all processes running – check the man page for `ps` to see what `ps -a` does.
- `wc`, to count the number of lines in the output of `ps`. Check the man page for `wc` to see what option will count the number of lines in a file.

Note that the *first* line of the output from `ps` is a header line, and should not be included in the total count of processes. Check the man page for the `tail` command (especially the `--lines` option) for ways of removing it.