

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 "("
(( ... )): (( 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

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
```



## Expansion

Bash does *many* different sorts of expansion besides expanding variables.

From the bash man page:

## EXPANSION

Expansion is performed on the command line after it has been split into words. There are seven kinds of expansion performed: brace expansion, tilde expansion, parameter and variable expansion, command substitution, arithmetic expansion, word splitting, and pathname expansion.

The order of expansions is: brace expansion; tilde expansion, parameter and variable expansion, arithmetic expansion, and command substitution (done in a left-to-right fashion); word splitting; and pathname expansion.

• • •

We'll look at these in more detail in lab/workshops.

# 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.