THE UNIVERSITY OF WESTERN AUSTRALIA

Shell Functions and Make/Makefiles

Lecture 16

Michael J Wise

Shell functions

- If a Shell script is calling another Shell script that you've written, you can also define the called script within the main script as a function.
- The format is:

```
function <name> {
    <commands>
    return [<exit status>]
}
```

Or

```
<name> ( ) {
    <commands>
    return [<exit status>]
}
```

Input values and return values

- There are no formal parameters between the brackets (e.g. as in Python functions), instead you use `$1`, `$2`, etc within the body of the function
- `$0` is the name of the function
- The values returned by Shell functions (like Shell scripts) are exit statuses. Default is 0.
- Other positive integers can be returned (like Shell scripts)

Scope of variables

- By default, the scope of all variables is global – all variables are visible everywhere
- To have a variable only visible within a function, use the command `local` after the function header to declare variable(s) as local to that function.

Example – regression testing

- Regression testing is where, as you work on your code, you check that any changes you've made don't break something else in the code
  - *Have a range of test that grows with new functionality*

Example – regression testing

```
function run_a_test {
  testagrepy.py $1 $2 > out
  if cmp out $3
  then
    echo "Test: $1 $2 ok"
  else
    echo "Test: $1 $2 fails"
  fi
}
```

- Testing an approximate string matching algorithm. e.g.

```
run_a_test oo wooloomooloo expected.1
run_a_test fred wooloomooloo expected.2
```

Limitation

- A function is not a script.
  - *Calling a function within a script is more efficient than calling a separate script*
  - *BUT a function (in a script) cannot be called by a different script*

Make – Doing only what's needed

- Historically, programs such as those written in C, were created from modules.
  - *Each module had to be compiled into a binary*
  - *Binaries had then to be linked to form an executable*
- If one module changes, no point recompiling every other module, just the affected module (and down-stream), and re-link
- The Unix tool `make` takes specification of what needs to be done, what the inputs are and what the processes are, in the form of a `Makefile`.
- Useful for any process where intermediate files expensive to recompute or there are multiple stages

Makefile format

- Unlike Sed, Awk, there is no command-line Make. Need to have a `Makefile` (or `makefile`). Can also specify makefile name with `make -f` (but not recommended)
- There are two sorts of components in Makefiles: Rules and Variables.
- Rules look like:

<target(s)> : *<pre-requisites>*

    <commands>

Makefile format

- There can be more than one targets (space separated) and zero or more pre-requisites, but keep it simple and have only one target
- Commands appear on successive lines. MUST begin with a <tab> character
- Execution begins with the first target

Example

```
C14UBT_results.txt : C14UBT_clean.tsv
	analyseUBT.py C14UBT_clean.tsv > C14UBT_results.txt

C14UBT_clean.tsv: PW_clean.csv  CP_clean.csv
	cat PW_clean.csv CP_clean.csv > C14UBT_clean.tsv

PW_clean.csv: PW_data.csv
	clean_C14UBT PW.csv > PW_clean.csv

CP_clean.csv: CP_data.csv
	clean_C14UBT CP.csv > CP_clean.csv
```

Make variables

- Make variables are typically found at the start of a Makefile.

<name> = *<string>*

```
data_root = /usr/home/michaelw/etseq/C14UBT/data
```

- In the body of the Makefile, use `$( )` to insert value

```
PW_clean.csv: PW_data.csv
	clean_C14UTB $(data_root)/PW.csv > PW_clean.csv
```

% Wildcard

- % is to Make what .* is to regular expressions – match zero or more characters, typically in a file name in a target or pre-cursor.

```
%_clean.csv: %_data.csv
```

Automatic (built-in) variables

- Like Sed and Awk, Makefiles have access to automatic (i.e. built-in variables
- `$@` - the target
- `$<` - the first precondition
- `$^` - a list of all the preconditions (space separated)
- `$*` - whatever has matched a wild-card pattern

Special targets

- There are a number of Special Targets, i.e. targets that are not intended to be made, but convey other information. One is particularly useful.

```
.PRECIOUS
```

- By default, Make tidies up by removing intermediate files. This may be undesirable if it's taken a lot of time to compute them and they've not changed

```
.PRECIOUS %.clean_tsv
```

Example – take 2

```
data_root = /usr/home/michaelw/etseq/C14UBT/data

OBJ = C14UBT_results.txt

.PRECIOUS %_clean.csv

What_to_make: $(OBJ) # allows for multiple top targets

C14UBT_results.txt : C14UBT_clean.tsv
	analyseUBT.py  $< >$@

C14UBT_clean.tsv: PW_clean.csv CP_clean.csv
	cat $^ > $@

%_clean.csv: %_data.csv
	clean_C14UTB  $(data_root)/$*.csv > $@
```

Invoking make

- Make will generally be used without command-line options. However, a couple are useful:

`-j` <*N*> - Instead of just one make target being made at a time, make N targets in parallel

`-k` - Keep going to next target if an error is encountered. Otherwise exits.

Caveat and competitor

- Make is a very brittle program.
  - *Easy to get the syntax errors or target errors (i.e. the item to be made fails to match any of the target patterns*
- Make is very widely used
- There are competitors, e.g. Snakemake

https://snakemake.github.io/

The original “Computers” at NASA Ames

https://twitter.com/nasaames/status/1204868782096699392?