Shell Scripting

Jeremy Sanders

October 2011

1 Introduction

If you use your computer for repetitive tasks you will find scripting invaluable (one of the advantages of a command-line interface). Basically a script is a file containing a list of commands that you would have typed in at the shell, e.g. make that file, modify it, rename it... Indeed as shells are virtually fully-fledged programming environments you can do quite a bit more than run through a list of commands. Scripts can have parameters, subroutines, functions, conditional parts and loops. We can also use a scripting language (these aren't shells!) like Python or Perl, which are far more powerful languages.

The shell you type your commands in, tcsh, is unfortunately rather brain-dead at scripting (see `http://www-uxsup.csx.cam.ac.uk/misc/csh.html` for the gory details). If we were not astronomers I'd advise everyone never to touch tcsh/csh with a bargepole (indeed I still do...), but I'm afraid we're stuck with it for historical reasons. If you understand what you're doing you might want to look at bash, but otherwise stick with tcsh, and use a better scripting language (like Perl or Python if you need to do more).

The possible options for writing a script or programming are:

1. We write the script in tcsh. We have to step around all the horrible problems with csh/tcsh. You can do this for simple scripts, but doing anything slightly more complex feels like your brain is sucked out through your nose, but fortunately we live to tell our tale to the others in the sanatorium between the nightmares.

2. We write the script in a good shell like bash. This has a lot going for it, but unfortunately we don't get access to all the aliases that setup scripts for the astronomical packages provide. Aliases are also a brain-dead way of providing commands in the shells without doing properly. That's what you get when you ask astronomers to write software packages without giving them training. I'm told to advise you against bash, as few other astronomers know what you're talking about.

3. We write in a much more featureful scripting language like Python or Perl. We also don't get the the aliases that the setup scripts provide, but we can write much more complex programs that can do proper maths. I'll cover scripting languages briefly elsewhere.

4. We can write a script in a proprietary language like IDL or Mathematica. Can be simple and powerful but be aware that people you might want to give your code to won't have the cash to buy these expensive programs to run your simple script (don't laugh - I know one astronomer who writes all his simple scripts in IDL, so no one else uses them). People won't always have IDL or Mathematica on their laptops.

5. We can write in a fully-fledged programming language (not a scripting language) like C, Fortran or C++. Typically it is harder to write a program in these languages but it will run more quickly than the above options. This is necessary if you need to do lots of calculations. I'll cover programming elsewhere.

6. If we are a hardcore masochistic nut, we can code in raw binary by flipping bits on a disk with a magnet. We would also be able to tell our exciting life story to those guys who are putting us in that nice soft room. I'll leave it others more qualified to cover this.

I'm in two minds what to give you here, so I'll give you examples of writing scripts in tcsh and bash, and strongly advise those who are doing more than running the same ten commands in a row to use Python or Perl (maybe bash if you feel up to understanding what's going on).

I'm not going to give you a fully fledged introduction to shell scripting, but leave that to the resources I'll give you on the links page.

2 Hello, world!

As conventional in the programming world, here is our first script:

```
#!/usr/bin/env tcsh
echo "Hello, world!"
```

We can put these lines in a file (say `myscript`) with our favourite editor. Next we mark the script as executable (so we can type its name into a shell and run it) by typing `chmod +x myscript`. Type `myscript` to run it.

The first line of the script tells the computer what type of script it is (or more precisely what shell to use to run it after `#!`). /usr/bin/env is a neat trick of a program to run its argument, and is guaranteed to be there on a Unix computer, so we don't need to know where tcsh is located. We could replace the first line with `#!/bin/tcsh` if tcsh lives in /bin). The second line just prints out "Hello, world!" as if we had typed it into the shell. We could add more commands if we wanted to something else, like "Hello, mum!". The same script for bash looks the same except we substitute `bash` for `tcsh`.

3 Arguments and variables

You can pass arguments to a script. Here is another example:

```
#!/usr/bin/env tcsh

# comments start with # symbols in shells
echo "Hello, $1!"
```

$1 corresponds to the first of the script's *arguments*. If we save the script as myscript2, chmod +x it, and run `myscript2 fred`, every $1 in the file (except for those in single ' quotes), is replaced by fred. You can pass more than one argument, and they are passed in $2, $3 and so on ($0 is the name of the script itself). $* expands to all the parameters to the script (or blank if there are none). A more useful example of this is

```
#!/usr/bin/env tcsh
ls -lrt $*
```

If we save this file as lsl and chmod +x it, doing `lsl *.tex *.txt` would show all the tex and txt files in order of modification time, most recent last, and showing all the details (replace tcsh with bash for the bash version of the script).

Settings in your shell are stored in *variables* and *enviroment variables*. Variables aren't inherited by programs started by shells, but environment variables are. Use `set` and `setenv` to set variables and environment variables respectively in tcsh (type `setenv` or `set` to get a list of them in tcsh), which can also be accessed by putting a dollar sign in front of them in a script or at the prompt. Important ones include PATH, which contains a list of directories searched for programs when you type their name in the shell (type `echo $PATH` into tcsh), and HOSTNAME which holds the hostname of your computer. Indeed $1, $2, etc, are special variables with numbers as names.

Variables are often used if you use a *loop* in a script—a way of executing the same set of commands many times. An example in bash (also demonstrating simple usage of variables):

```
#!/usr/bin/env bash

for file in *.txt; do
  newname="$file.old"
  cp "$file" "$newname"
done
```

or tcsh:

```
#!/usr/bin/env tcsh

foreach file (*.txt)
  set newname="$file.old"
  cp "$file" "$newname"
end
```

These scripts both copy all the files in the current directory with the extension .txt to files with .txt.old, for example `fred.txt` gets copied ot `fred.txt.old`.

4 A more complete example

Here is an example in tcsh, for renaming files with one set of extensions to another. Quoting is very iffy in tcsh, so we have to do some extra work to the sed line than in bash.

```
#!/usr/bin/env tcsh

# test if we have two arguments, exit with warning if we don't
if ($# != 2) then
 # we can't (AFAIK) redirect stdout to stderr in tcsh
 echo "Usage $0 oldextension newextension"
 exit 1
endif

foreach file (*.$1)
  # back ticks start commands, returning result
```

```
  # this one echos filename, and replaces old extension
  # with the new one
  set newname=‘echo $file | sed s/$1\$/$2/‘

  # actually rename the file
  echo "Renaming $file to $newname"
  mv "$file" "$newname"
end
```

Here is the bash version:

```
#!/usr/bin/env bash

# test whether two parameters weren’t given (special variable $#)
# we exit with an error if they aren’t

if [[ $# != 2 ]]; then
  # redirect output to error output
  echo "Usage: $0 oldextension newextension" 1>&2
  # return with non-zero status (indicating error)
  exit 1
fi

for file in *.$1; do
  # back ticks start commands, returning result
  # this one echos filename, and replaces old extension
  # with the new one
  newname=‘echo $file | sed "s/$1$/$2/"‘

  # actually rename the file
  echo "Renaming $file to $newname"
  mv "$file" "$newname"
done
```

The function of the example script above is to rename a set of files with one extension to another (so you could do `renamefiles txt tex`, and all files called .txt would be renamed to .tex in the current directory).

The hardest bit to understand in that example is the line with the sed and the *back-ticks* (you need to hunt on your keyboard for that strange symbol, hint—it’s not ’ or ”). Commands in a back-tick are run by the shell separately, the text they output are taken, and that text is inserted where the text between the backticks were, e.g.

```
echo ‘date‘
echo "The date is ‘date‘"     [or]
```

Takes the output of the date command, and uses echo to display it (rather perversely), or you could even do

```
thedate=‘date‘
echo "The date is $thedate"
```

The function of the sed command between the backticks is to take the text `$1[end of line]` ($ confusingly marks the end of the line) and replace it with $2.

Also to note is the exit line. All programs return a status code, by default 0, meaning success. The `exit 1` line indicates to return an error (read man pages to find a particular program's exit codes). You can test the error status of a previous command in a script by looking at the `$?` variable:

```
#!/usr/bin/env bash

# let's look for files called *.fish
ls *.fish

if [[ $? != 0 ]]; then
 echo "That didn't work!"
fi
```

or in tcsh:

```
#!/usr/bin/env tcsh

ls *.fish
if ($? != 0) then
 echo "That didn't work!"
endif
```

5 Testing for files

In bash and tcsh we can also test for the existence of files or directories. An example is:

```
#!/usr/bin/env tcsh

set datafile=data.dat
if ( -f $datafile ) then
 echo "We found $datafile"
else
 echo "We didn't find $datafile"
endif
```

or

```
#!/usr/bin/env bash

datafile=data.dat
if [ -f $datafile ]; then
 echo "We found $datafile"
else
 echo "We didn't find $datafile"
fi
```

6 Going further

This is only a small sample of what you can do in shells for further information read:

1. `man tcsh` - how to program in tcsh. Rather complex. See my links page for an easier to read web version.
2. `info bash` - bash docs. Rather complex. Best place to learn scripting is the Advanced Bash Programming Howto (see my links page).