SHELL SCRIPTING

Scripting allows you to:

- encapsulate common lists of commands in a file
- automate/batch processes
- make new flexible and configurable tools
- understand other people's scripts/tools

"I wish you'd forced me to learn scripting 3 years ago!" - Dr Parry

INTRODUCTION

- Common scripting languages include:
  - Shell scripts - sh, bash, csh, tcsh
  - Other scripting languages - TCL, Perl, Python
- We will look at sh as it is simple, portable, powerful and just like the command line

- Repeated tasks can be done ***much*** faster
- Scripts act as exact records of what commands were run
- Scripts avoid small inconsistent errors in processing

- Knowing scripting helps with any computer tasks, not just analysis
- Examples of useful tasks:
  - Automatically call BET with several different options
  - Measure image stats/volumes/PEs for a set of subjects with a single command
  - Extract timing info from stimulus / behavioural data files
  - Renaming sets of files
  - Changing the format of a set of files (e.g. png to tiff)

- In these slides there are several accompanying practicals that are *extremely* useful and can be found by following the links marked with  at the bottom right (to the left of the navigation arrows)

DATA MANAGEMENT

- In order to help scripting (and your own sanity) it is a good idea to give files and directories systematic names.
  - For example:
  - `Con0001 , Con0002, Con0003 , ... , Con0020, Pat0001, Pat0002, ... , Pat0020`

- Padding the numbers with zeros like this (`Con0020` instead of `Con20`) helps keep the ordering consistent for scripting

- In the UK (and other countries) you ***must not*** have identifiers (names, dates of birth, even *initials*) in the

filenames
- Consider renaming your original images to more meaningful (and consistent) names

- Keep track of your disk usage and delete any analyses that were incorrect (many people have `res.feat, res+.feat, res++.feat, res+++.feat` , etc. which is confusing and a waste of space)

scripting10 : 3 of 32

BASIC SHELL SCRIPT

- A bourne shell (sh) script is a list of lines in a file that are executed in the bourne shell (a forerunner of bash); simplest is just commands that could be run at the prompt.

- The first line in a sh script **MUST** be `#!/bin/sh`

- Things to remember:
  - always make sure it has executable status
    `chmod a+x filename`
  - script runs in the current directory (pwd)
  - may not inherit the same environment - esp. if used by others

- Example:
```
#!/bin/sh
bet im1 im1_brain -m
mv im1_brain_mask.nii.gz mask1.nii.gz
```

- A script can be stopped at any point by using `return`: e.g.
`return 0`

USEFUL SHELL SCRIPT TEMPLATE

- Before learning things systematically, here is a fairly simple script which is very powerful and useful for modifying for many different tasks.

```
#!/bin/sh
for filename in *.nii.gz ; do
  fname=`$FSLDIR/bin/remove_ext ${filename}`
  fslmaths ${fname} -s 2 ${fname}_smooth2
  mv ${fname}.nii.gz ${fname}_smooth0.nii.gz
done
```

- What this does:
  For each image (*.nii.gz) it smooths it to make a new one of the same name but ending in _smooth2 and also renames the unsmoothed image to end with _smooth0
- How this works:
  - The variable `filename` is used in a *for* loop to go through each name matching *.nii.gz
  - The variable `fname` is set to the filename with the ending (e.g. .nii.gz) removed.
    Don't worry about how this works for now - the details will be explained later.
  - `${filename}` and `${fname}` are used to get the values (contents) of the variables
  - fslmaths is used to do the smoothing.
  - mv is used to do the renaming (notice that .nii.gz is needed here, but not for the fsl tools, as they work with or without the .nii.gz endings).

SCRIPTING TIPS

Here are some basic, but useful, tips for writing scripts

- Put in comments (to jog your memory when you write your paper months/years later)
- Put in some `echo` output commands so that you get some feedback on what your script is doing as it runs
- If your script starts doing something bad (or nothing at all) then use control-C to stop it
- If your script makes new files, changes files or deletes files then start with a version which uses `echo` in front of the important commands. When you run this version it will just display the commands to the screen so that you can examine them carefully and make sure they are right. Once you are happy with them then remove the `echo` from in front of these commands and run this version.
- It doesn't hurt to make a backup of key files before running a script, just in case.

BASIC SCRIPTING CONCEPTS

- We will now look systematically at the following shell and scripting concepts:
  - *Wildmasks*
  - *Echo (printing to the screen/file)*
  - *Variables*

  - *Braces*
  - *Command Line Arguments*
  - *Single Quotes and Backslash*
  - *Double Quotes*
  - *Backquotes*
  - *Pipes*
  - *File Redirection*
- Following this some useful utilities and programming constructs (like the for loop) will be covered.

WILDMASKS

Can use wildmasks for matching patterns in *filenames*; expand into a list of *all* filename matches. E.g.:

`*` matches any string
`?` matches any one character
`[abgj]` matches any one character in this range/list

```
$ ls
  sub1_t1.nii.gz sub1_t2.nii.gz sub2_t1.nii.gz
sub2_t2.nii.gz sub3_pd.nii.gz
$ ls sub*
  sub1_t1.nii.gz sub1_t2.nii.gz sub2_t1.nii.gz
sub2_t2.nii.gz sub3_pd.nii.gz
$ ls sub1*
  sub1_t1.nii.gz sub1_t2.nii.gz
$ ls sub*t1*
```

```
  sub1_t1.nii.gz sub2_t1.nii.gz
$ ls sub[13]*
  sub1_t1.nii.gz sub1_t2.nii.gz sub3_pd.nii.gz
$ ls sub?_t2.nii.gz
  sub1_t2.nii.gz sub2_t2.nii.gz
```


ECHO

- `echo` prints the rest of the line to the screen (standard output).
- This is useful for providing output or updates in a script.

- Wildmasks (for filenames) and variables (values) are substituted in the argument *before* echo prints them.

- Examples:

```
$ echo Hello All!
  Hello All!
$ echo sub*t1*
  sub1_t1.nii.gz sub2_t1.nii.gz
$ echo j*k
  j*k
```

VARIABLES

- Like most programming languages, the shell allows items to be stored in variables.
- All shell variables store *strings*.

- A variable is set using:
  `NAME=VALUE`
- The variable name should start with a letter but can contain numbers and underscores
- The value of a variable can be returned/used by adding a prefix `$`

- Examples:

```
$ var1=im1.nii.gz
$ echo $var1
  im1.nii.gz
$ echo var1
  var1
$ ls $var1
  im1.nii.gz
```


BRACES

- Any name that starts with a letter can be used as a variable name.
- For instance: `v, v1, v1_1, v_filename_4`

- To add a string immediately after a variable name can be confusing.

- The situation is solved by putting the variable name inside braces.

- Examples:

```
$ v=im1
$ echo $v_new

$ echo ${v}_new
  im1_new
```

- NB: all unused variables are blank by default (generate no error)

COMMAND LINE ARGUMENTS

- Inside a script the variables `$1 $2 $3` *etc.* store the value of the command line arguments.

- e.g. if a script called reg_vol is executed as:
  `$ reg_vol im1 3 abc`
  then `$1 = im1, $2 = 3, $3 = abc`

- Other special variables are:
  - `$0` = name of the script (often including the path)
  - `$#` = number of command line arguments given
  - `$@` = all the command line arguments (i.e. `$1 $2 $3 ...`)

- $$ = process ID number (unique to this process)

SINGLE QUOTES AND BACKSLASH

- The shell substitutes variable names and wildmasks *before* executing the command - sometimes this is undesirable.
- To avoid substitutions either
  1. prefix the special character (wildmask or $ sign) with a backslash: \
  2. put the desired string in single quotes: '

- Examples:

```
$ var1=im1.nii.gz
$ echo $var1
  im1.nii.gz
$ echo \$var1
  $var1
$ echo '$var1'
  $var1
```


DOUBLE QUOTES

- To group several strings together as one argument it is necessary to use double quotes: "

- For example:

```
$ v=Hello World
$ echo $v
  Hello
$ v="Hello World"
$ echo $v
  Hello World
```

- NB: Variable substitutions are done inside double quotes but wildmasks are *not* expanded:
  e.g. `echo "*"` just prints a *
  but `echo "$v"` is the same as `echo $v`

BACKQUOTES

- The (text) result of any command can be captured using backquotes: `

- This is very useful for setting variables.

- Examples:

```
$ v=`ls sub[13]*`
$ echo $v
```

```
  sub1_t1.nii.gz sub1_t2.nii.gz
sub3_pd.nii.gz
$ echo `fslval sub1_t1 pixdim2`
  4.0
```

- NB: the result is always treated as a single string, even if it contains spaces

PIPE

- One of the most powerful features of the shell is the ability to chain commands together, each taking its input from the previous command's output.
- This is done using the pipe symbol: `|`
- Examples (using the wordcount utility):

```
$ cat .bashrc | wc
  7   83   534
$ echo "Hello World" | wc
  1   2   12
```

- Technically this redirects standard output of one command to be the standard input of another.
- Error messages that are printed to standard error are *not* redirected with the pipe.

FILE REDIRECTION

- Command input can be taken from a file with: `<`
- Command output can be redirected to a file with: `>`
- Command output can be *appended* to a file with: `>>`
- Examples:

```
$ echo "smoothing=10mm" > settings.txt
$ echo "No lowpass" >> settings.txt
$ cat settings.txt
smoothing=10mm
No lowpass
```


HANDY SHELL UTILITIES

- Some common and nearly essential utilities/programs for shell scripting are:

Basic	Advanced
◦ `test` (or `[ ]`) ◦ `if` ◦ `for` ◦ `while`	◦ `grep` (search) ◦ `bc` (calculator) ◦ `sed` (find and replace) ◦ `awk` (select columns)

TEST

- The command `test` allows two strings or two integers to be compared.
- A shorthand version uses `[` and `]` around the arguments. *WARNING*: be very careful to put the spaces in correctly!
- The syntax is very different for comparing numbers or strings (see `help test` for the syntax options).
- `test` can also be used to check the status of files (whether they exist, are writable, etc.)

`test $a = my`	string equality
`[ $a = my ]`	as above
`[ $a -eq 2 ]`	tests numerical equality
`[ 11 -gt 2 ]`	compares numbers (greater than)
`[ 11 > 2 ]`	*does NOT do numerical comparison*
`[ -e im1.nii.gnii.gz ]`	tests if file exists

Note: for non-integer numerical comparisons, see `bc`

IF

- The `if` command works like in most programming languages.
- It usually uses the result of the `test` command and its syntax is:

```
if [ EXPRESSION ] ; then
COMMANDS ;
 else
COMMANDS2 ;
fi
```

- The `else` part is optional.
- For example:

```
if [ $a = 2 ] ; then
  b="y-axis";
fi
```


FOR

- The `for` command executes a set of commands for every word in a list of words.
- Syntax:

```
for VARIABLE in LIST OF VALUES ; do
    COMMANDS ;
done
```

- The commands are executed once for each entry in the words list.
- Each time the variable specified is equal to the current word.

- Example:

```
for filename in im1 im2 im3 ; do
    bet $filename ${filename}_brain ;
done
```


BC

- The `bc` command acts like a calculator.

- It is usually used to set variables, using *echo*, *pipe* and *backquotes*.
- Use the `-l` option to get accurate floating-point arithmetic.

- Example:

```
$ a=2;
$ a=`echo "3 * $a + 1" | bc -l`;
$ echo $a
  7
```

- Note: the double quotes stop the * being used as a wildmask for filenames.
- Numerical comparisons of non-integers can be done with `bc` where 0 is returned for false and 1 for true.
  e.g. `echo "-1.2 < 0.5" | bc` gives 1
  e.g. `echo "-1.2 > 0.5" | bc` gives 0
  So in an if statement do something like:

```
 if [ `echo "$a < $b" | bc -l` = 1 ] ;
then ...
```


WHILE

- The `while` command executes a set of commands as long as the condition is true.
- Syntax:

```
while CONDITION ; do
    COMMANDS ;
done
```

- The condition is usually a `test` statement.

- Example:

```
a=1
while [ $a -lt 4 ] ; do
    bet im$a brain$a ;
    a=`echo $a + 1 | bc` ;
done
```


GREP

- The `grep` command finds patterns in strings.
- It is usually used to extract lines from a file that contain a given word or phrase.
- This is a very powerful tool when used together with pipes to filter outputs.
- Example:
  Find the value of pixdim1 from the fslhd output

```
fslhd im1 | grep pixdim1
```

AWK

- The `awk` command is a very general pattern matching facility.
- One simple but useful capability is to pick out columns of text.
- This is particularly handy for manipulating tabular information such as in stimulus files.

- Syntax for selecting column *N* is:
  ```
  awk '{print $N}'
  ```

- Note that the *exact* syntax (quotes and braces) must be used.
- Example:
  ```
  $ v="im*.nii.gz"
  $ echo $v
    im1.nii.gz im2.nii.gz im3.nii.gz
  $ echo $v | awk '{print $2}'
    im2.nii.gz
  ```


SED

- The `sed` command performs string substitutions.

- It is usually used to add, remove or change parts of a string.
- This is often invaluable for modifying variables.

- Syntax for changing `STRING1` to `STRING2` is:
```
sed s/STRING1/STRING2/g
```

- Example:
```
$ v="im*.nii.gz"
$ v=`echo $v | sed s/im/Subject/g`
$ echo $v
  Subject1.nii.gz Subject2.nii.gz
Subject3.nii.gz
```

- Warning: The characters `. * [ ] /` have special meaning in the first string unless preceded by a backslash
- Tip: Any character can be used instead of `/`
  e.g. `sed s@STRING1@STRING2@g`
  which can be *very* handy when dealing with directories/files

FUNCTIONS

- Like other languages, functions can be defined in shell scripts.
- Useful for splitting up scripts into understandable, reusable pieces.
- Functions can be called like independent scripts.

- Syntax for creating a function is:
```
function NAME { COMMANDS ; }
```

or the short form:

```
NAME () { COMMANDS ; }
```

- Example:

```
$ function hi { echo "Hi! $1" ; }
$ hi
  Hi!
$ hi There
  Hi! There
```


REGULAR EXPRESSIONS

- *Regular expressions* are a form of pattern matching syntax which many commands use. (e.g. `grep, sed`)
- They are very flexible and not quickly learnt.
- Some basic forms are easy to learn and very useful.
- *Not* the same as shell wildmasks, although some are similar.
- Special characters used in regular expressions include:

`.`	matches any one character
`*`	matches zero or more of the last character
`.*`	matches any string
`[ ]`	matches any character in the range
`^`	represents the start of the line
`$`	represents the end of the line
`[^ ]`	matches any character *not* in the range

REGULAR EXPRESSIONS

- Examples:

```
$ echo "Hello world" | sed 's/w.*/X/g'
  Hello X
$ echo "Hello world" | sed 's/w*/X/g'
  XHXeXlXlXoX XXoXrXlXdX
$ echo "Hello world" | sed 's/^.* /X/g'
  Xworld
$ echo "Hello world" | sed 's/.$/X/g'
  Hello worlX
$ echo "Hello world" | sed 's/[wo]/X/g'
  HellX XXrld
$ echo "Hello world" | sed 's/[^wo]/X/g'
  XXXXoXwoXXX
```


FSL COMMANDS

- There are many fsl command line utilities, but there are also specific utilities to assist with scripting:
  1. `remove_ext`
     this removes only image specific extensions: .nii.gz .nii .hdr .img .hdr.gz .img.gz
     e.g. `remove_ext /Volumes/MJ/img1.nii.gz` gives `/Volumes/MJ/img1`
  2. `imtest`
     this returns the character 1 if the specified file exists and is an image, or 0 otherwise
     e.g. `imtest ../img1` gives 1 if `../img1.nii.gz` exists (or if `../img1.hdr` exists, etc.)
  3. `imglob`
     this expands into a list of full filenames for images only
     e.g. `imglob *` lists only the images out of all matches for *
  4. `imcp, immv, imrm, imln`
     these do `cp, mv, rm` and `ln` for images, without needing to specify the extensions (useful when dealing with Analyze or nifti files without needing to know which)
     e.g. `imcp im1 im1_orig` would be the same as `cp im1.nii.gz im1_orig.nii.gz` for nifti files, but would do `cp im1.hdr im1_orig.hdr` and `cp im1.img im1_orig.img` for Analyze files

scripting10 : 30 of 32

FURTHER COMMANDS

- There are many other commands which are useful. For example:

- `basename`
  removes all leading directory info and specified extensions
  e.g. `basename /tmp/epi.nii.gz .nii.gz` gives `epi`
- `dirname`
  just returns the directory path to the specified file
  e.g. `dirname /tmp/epi.nii.gz` gives `/tmp`
- `sort`
  sorts files (by line) according to alphabetic or numerical order
- `which`
  reports where an executable file can be found
  e.g. `which flirt` gives `/usr/local/fsl/bin/flirt`
- `head`
  prints the first n lines of a file
- `tail`
  prints the last n lines of a file
- `touch`
  creates an empty file
- `paste`
  merges files together (horizontally)

LEARNING MORE

- For more image analysis command line tools see:

  www.fmrib.ox.ac.uk/fsl/avwutils/index.html

- To get details about a specific command:
  - run `man` or `help` on that command

- To learn more about general commands and scripting try:
  - searching the web
  - looking at other scripts (e.g. in $FSLDIR/bin)
    Note: check if they are scripts by running `file` on them
  - `apropos` searches for any commands with matching keywords in their description
    e.g. `apropos merge` shows all commands that have something to do with merging
  - books on unix, shell and scripting (though many of them are very technical)