

EXPERIMENT4:

AIM :Shell scripting in Linux

A shell script is a computer program designed to be run by the Unix/Linux shell which could be one of the following:

The Bourne Shell

The C Shell

The Korn Shell

The GNU Bourne-Again Shell

A shell is a command-line interpreter and typical operations performed by shell scripts include file manipulation, program execution, and printing text.

Extended Shell Scripts

Shell scripts have several required constructs that tell the shell environment what to do and when to do it. Of course, most scripts are more complex than the above one. The shell is, after all, a real programming language, complete with variables, control structures, and so forth. No matter how complicated a script gets, it is still just a list of commands executed sequentially.

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 e 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.feats, res+.feats, res++.feats, res+++.feats , etc. which is confusing and a waste of space)

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

prefix the special character (wildmask or \$ sign) with a backslash: \

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 wildcards 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
```

A simple program to add two numbers

```
#!/bin/bash
```

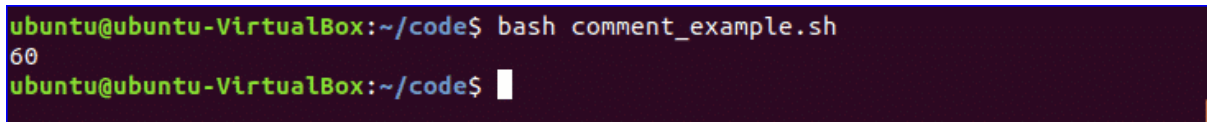
```
# Add two numeric value
```

```
((sum=25+35))
```

```
#Print the result
```

```
echo $sum
```

Output

A terminal window with a dark background. The prompt is 'ubuntu@ubuntu-VirtualBox:~/code\$'. The user has entered 'bash comment_example.sh'. The output is '60'. The prompt is now 'ubuntu@ubuntu-VirtualBox:~/code\$' with a cursor.

```
ubuntu@ubuntu-VirtualBox:~/code$ bash comment_example.sh
60
ubuntu@ubuntu-VirtualBox:~/code$
```

variables

A variable is a character string to which we assign a value. The value assigned could be a number, text, filename, device, or any other type of data.

A variable is nothing more than a pointer to the actual data. The shell enables you to create, assign, and delete variables.

Variable Names

The name of a variable can contain only letters (a to z or A to Z), numbers (0 to 9) or the underscore character (_).

By convention, Unix shell variables will have their names in UPPERCASE.

The following examples are valid variable names –

`_ALI`

`TOKEN_A`

`VAR_1`

`VAR_2`

Defining Variables

Variables are defined as follows:

`variable_name=variable_value`

For example:

`NAME="Zara Ali"`

The above example defines the variable NAME and assigns the value "Zara Ali" to it. Variables of this type are called scalar variables. A scalar variable can hold only one value at a time.

Accessing Values

To access the value stored in a variable, prefix its name with the dollar sign (\$).

For example, the following script will access the value of defined variable NAME and print it on STDOUT

```
#!/bin/sh
```

```
NAME="Zara Ali"
```

```
echo $NAME
```

The above script will produce the following value:

Zara Ali

Read-only Variables

Shell provides a way to mark variables as read-only by using the read-only command. After a variable is marked read-only, its value cannot be changed.

For example, the following script generates an error while trying to change the value of NAME:

```
#!/bin/sh  
  
NAME="Zara Ali"  
  
readonly NAME  
  
NAME="Qadiri"
```

The above script will generate the following result:

```
/bin/sh: NAME: This variable is read only.
```

Unsetting Variables

Unsetting or deleting a variable directs the shell to remove the variable from the list of variables that it tracks. Once you unset a variable, you cannot access the stored value in the variable.

Following is the syntax to unset a defined variable using the unset command –

```
unset variable_name
```

The above command unsets the value of a defined variable. Here is a simple example that demonstrates how the command works:

```
#!/bin/sh  
  
NAME="Zara Ali"  
  
unset NAME  
  
echo $NAME
```

The above example does not print anything. You cannot use the unset command to unset variables that are marked readonly.

Variable Types

When a shell is running, three main types of variables are present:

- **Local Variables** – A local variable is a variable that is present within the current instance of the shell. It is not available to programs that are started by the shell. They are set at the command prompt.

- **Environment Variables** – An environment variable is available to any child process of the shell. Some programs need environment variables in order to function correctly. Usually, a shell script defines only those environment variables that are needed by the programs that it runs.
- **Shell Variables** – A shell variable is a special variable that is set by the shell and is required by the shell in order to function correctly. Some of these variables are environment variables whereas others are local variables.

Control constructs

You can control the execution of Linux commands in a shell script with control structures. Control structures allow you to repeat commands and to select certain commands over others. A control structure consists of two major components: a test and commands. If the test is successful, then the commands are executed. In this way, you can use control structures to make decisions as to whether commands should be executed.

There are two different kinds of control structures: loops and conditions. A loop repeats commands, whereas a condition executes a command when certain conditions are met. The BASH shell has three loop control structures: while, for, and for-in. There are two condition structures: if and case. The control structures have as their test the execution of a Linux command. All Linux commands return an exit status after they have finished executing. If a command is successful, its exit status will be 0. If the command fails for any reason, its exit status will be a positive value referencing the type of failure that occurred. The control structures check to see if the exit status of a Linux command is 0 or some other value. In the case of the if and while structures, if the exit status is a zero value, then the command was successful and the structure continues.

Test Operations

With the test command, you can compare integers, compare strings, and even perform logical operations. The command consists of the keyword test followed by the values being compared, separated by an option that specifies what kind of comparison is taking place. The option can be thought of as the operator, but it is written, like other options, with a minus sign and letter codes. For example, -eq is the option that represents the equality comparison. However, there are two string operations that actually use an operator instead of an option. When you compare two strings for equality you use the equal sign, =. For inequality you use !=. Here is some of the commonly used options and operators used by test.

The syntax for the test command is shown here:

```
test value -option value
```

```
test string = string
```

Integer Comparisons

Function

-gt	Greater-than
-lt	Less-than
-ge	Greater-than-or-equal-to
-le	Less-than-or-equal-to
-eq	Equal
-ne	Not-equal

String Comparisons Function

-z	Tests for empty string
=	Equal strings
!=	Not-equal strings

Logical Operations Function

-a	Logical AND
-o	Logical OR
!	Logical NOT

File Tests Function

-f	File exists and is a regular file
-s	File is not empty
-r	File is readable
-w	File can be written to, modified
-x	File is executable
-d	Filename is a directory name

Conditional Control Structures

The BASH shell has a set of conditional control structures that allow you to choose what Linux commands to execute. Many of these are similar to conditional control structures found in programming languages, but there are some differences. The if condition tests the success of a Linux command, not an expression. Furthermore, the end of an if-then command must be indicated with the keyword `fi`, and the end of a case command is indicated with the keyword `esac`.

The if structure places a condition on commands. That condition is the exit status of a specific Linux command. If a command is successful, returning an exit status of 0, then the commands within the if structure are executed. If the exit status is anything other than 0, then the command has failed and the commands within the if structure are not executed. The if command begins with the keyword `if` and is followed by a Linux command whose exit condition will be evaluated. The keyword `fi` ends the command. The `elsels` script in the next example executes the `ls` command to list files with two different possible options, either by size or with all file information. If the user enters an `s`, files are listed by size; otherwise, all file information is listed.

`if`

`if` executes an action if its test command is true.

Syntax:

`if` command then

command

`fi`

`if` then else

`if-else` executes an action if the exit status of its test command is true; if false, then the else action is executed.

Syntax:

`if` command then

command

else

command

`fi`

elif

elif allows you to nest if structures, enabling selection among several alternatives; at the first true if structure, its commands are executed and control leaves the entire elif structure.

Syntax:

if command then

 command

elif command then

 command

else

 command

fi

case

case matches the string value to any of several patterns; if a pattern is matched, its associated commands are executed.

Syntax:

case string in

pattern)

command;;

esac

Logical AND

The logical AND condition returns a true 0 value if both commands return a true 0 value; if one returns a non-zero value, then the AND condition is false and also returns a non-zero value.

Syntax:

command && command

Logical OR

The logical OR condition returns a true 0 value if one or the other command returns a true 0 value; if both commands return a non-zero value, then the OR condition is false and also returns a non-zero value.

Syntax:

command || command

Logical NOT

The logical NOT condition inverts the return value of the command.

Syntax:

! command

Loop Control Structures

The while loop repeats commands. A while loop begins with the keyword while and is followed by a Linux command. The keyword do follows on the next line. The end of the loop is specified by the keyword done. The Linux command used in while structures is often a test command indicated by enclosing brackets.

The for-in structure is designed to reference a list of values sequentially. It takes two operands—a variable and a list of values. The values in the list are assigned one by one to the variable in the for-in structure. Like the while command, the for-in structure is a loop. Each time through the loop, the next value in the list is assigned to the variable. When the end of the list is reached, the loop stops. Like the while loop, the body of a for-in loop begins with the keyword do and ends with the keyword done.

while

while executes an action as long as its test command is true.

Syntax:

while command

do

command

done

until

until executes an action as long as its test command is false.

Syntax:

until command

do

command

done

for-in

for-in is designed for use with lists of values; the variable operand is consecutively assigned the values in the list.

Syntax:

for variable in list-values

do

command

done

for

for is designed for reference script arguments; the variable operand is consecutively assigned each argument value.

Syntax:

for variable

do

command

done

select

select creates a menu based on the items in the item-list; then it executes the command; the command is usually a case.

Syntax:

select string in item-list

do

command

done

Scope of an environment variable

Scope of any variable is the region from which it can be accessed or over which it is defined. An environment variable in Linux can have global or local scope.

Global

A globally scoped ENV that is defined in a terminal can be accessed from anywhere in that particular environment which exists in the terminal. That means it can be used in all kind of scripts, programs or processes running in the environment bound by that terminal.

Local

A locally scoped ENV that is defined in a terminal cannot be accessed by any program or process running in the terminal. It can only be accessed by the terminal (in which it was defined) itself.

Accessing ENVs

Syntax:

`$NAME`

NOTE: Both local and global environment variables are accessed in the same way.

Displaying ENVs

To display any ENV

Syntax:

`$ echo $NAME`

To display all the Linux ENVs

Syntax:

`$ printenv //displays all the global ENVs`

or

\$ set //display all the ENVs(global as well as local)

or

\$ env //display all the global ENVs

Setting environment variables

To set a global ENV

\$ export NAME=Value

or

\$ set NAME=Value

To set a local ENV

Syntax:

\$ NAME=Value

Some commonly used ENVs in Linux:

\$USER : Gives current user's name.

\$PATH : Gives search path for commands.

\$PWD : Gives the path of present working directory.

\$HOME : Gives path of home directory.

\$HOSTNAME : Gives name of the host.

\$LANG : Gives the default system language.

\$EDITOR : Gives default file editor.

\$UID : Gives user ID of current user.

\$SHELL : Gives location of current user's shell program.

Shell functions

Functions enable you to break down the overall functionality of a script into smaller, logical subsections, which can then be called upon to perform their individual tasks when needed.

Using functions to perform repetitive tasks is an excellent way to create code reuse. This is an important part of modern object-oriented programming principles.

Shell functions are similar to subroutines, procedures, and functions in other programming languages.

Creating Functions

To declare a function, simply use the following syntax:

```
function_name () {  
    list of commands  
}
```

The name of your function is `function_name`, and that's what you will use to call it from elsewhere in your scripts. The function name must be followed by parentheses, followed by a list of commands enclosed within braces.

Example

Following example shows the use of function:

```
#!/bin/sh  
  
# Define your function here  
  
Hello () {  
    echo "Hello World"  
}  
  
# Invoke your function  
  
Hello
```

Upon execution, you will receive the following output:

```
$/test.sh
```

```
Hello World
```

Pass Parameters to a Function

You can define a function that will accept parameters while calling the function. These parameters would be represented by \$1, \$2 and so on.

Following is an example where we pass two parameters Zara and Ali and then we capture and print these parameters in the function.

```
#!/bin/sh

# Define your function here

Hello () {
    echo "Hello World $1 $2"
}

# Invoke your function

Hello Zara Ali
```

Upon execution, you will receive the following result:

```
$/test.sh

Hello World Zara Ali
```

Returning Values from Functions

If you execute an exit command from inside a function, its effect is not only to terminate execution of the function but also of the shell program that called the function.

If you instead want to just terminate execution of the function, then there is way to come out of a defined function.

Based on the situation you can return any value from your function using the return command whose syntax is as follows:

return code

Here code can be anything you choose here, but obviously you should choose something that is meaningful or useful in the context of your script as a whole.

Example

Following function returns a value 10:

```
#!/bin/sh

# Define your function here

Hello () {
```

```
    echo "Hello World $1 $2"

    return 10
}

# Invoke your function

Hello Zara Ali

# Capture value returned by last command

ret=$?

echo "Return value is $ret"
```

Upon execution, you will receive the following result:

```
$/test.sh

Hello World Zara Ali

Return value is 10
```

Nested Functions

One of the more interesting features of functions is that they can call themselves and also other functions. A function that calls itself is known as a recursive function.

Following example demonstrates nesting of two functions:

```
#!/bin/sh

# Calling one function from another

number_one () {

    echo "This is the first function speaking..."

    number_two

}

number_two () {

    echo "This is now the second function speaking..."

}

# Calling function one.

number_one
```

Upon execution, you will receive the following result:

This is the first function speaking...

This is now the second function speaking...

Function Call from Prompt

You can put definitions for commonly used functions inside your profile. These definitions will be available whenever you log in and you can use them at the command prompt.

Alternatively, you can group the definitions in a file, say test.sh, and then execute the file in the current shell by typing:

```
$. test.sh
```

This has the effect of causing functions defined inside test.sh to be read and defined to the current shell as follows:

```
$ number_one
```

This is the first function speaking...

This is now the second function speaking...

```
$
```

To remove the definition of a function from the shell, use the unset command with the .f option. This command is also used to remove the definition of a variable to the shell.

```
$ unset -f function_name
```