Data Analysis in Geophysics
ESCI 7205

Class 7

Bob Smalley

Basics of UNIX commands

A few comments –

"extended keyboard" keys (arrow keys, number keys, cut, paste, etc.) typically don't work between systems, or possibly over the network.

Have to be careful while editing in vi/vim.

Simple example

```
%more 0.5/run.csh
mkdir 0.5_3
tomoDD2.pwave tomoDD.3.inp
mv red* 0.5_3
cp tomo* 0.5_3
mv Vp* 0.5_3

mkdir 0.5_20
tomoDD2.pwave tomoDD.20.inp
mv red* 0.5_20
cp tomo* 0.5_20
mv Vp* 0.5_20
```

Use more command to see what is in shell script (file) run.csh.

This csh script simply runs a series of tomographic inversions using different parameter setups.

Simple example

```
%more 0.5/run.csh
mkdir 0.5_3
tomoDD2.pwave tomoDD.3.inp
mv red* 0.5_3
cp tomo* 0.5_3
mv Vp* 0.5_3

mkdir 0.5_20
tomoDD2.pwave tomoDD.20.inp
mv red* 0.5_20
cp tomo* 0.5_20
mv Vp* 0.5_20
```

When we run the script, it runs the commands in the file – so it runs the program tomoDD2, and moves the output files to specially named directories.
It then does it again with a different input data set.

Simple example

```
%more 0.5/run.csh
mkdir 0.5_3
tomoDD2.pwave tomoDD.3.inp
mv red* 0.5_3
cp tomo* 0.5_3
mv Vp* 0.5_3

mkdir 0.5_20
tomoDD2.pwave tomoDD.20.inp
mv red* 0.5_20
cp tomo* 0.5_20
mv Vp* 0.5_20
```

The prep work of writing the script allows us to save time and effort later.

Simple example

```
%more 0.5/run.csh
mkdir 0.5_3
tomoDD2.pwave tomoDD.3.inp
mv red* 0.5_3
cp tomo* 0.5_3
mv Vp* 0.5_3

mkdir 0.5_20
tomoDD2.pwave tomoDD.20.inp
mv red* 0.5_20
cp tomo* 0.5_20
mv Vp* 0.5_20
```

This is an example only. If we really wanted to run the same program multiple times, we would write this as some sort of loop.
This way we would only write the commands once, and pass the info that changes to the commands.

Standard example

Create a file (typically with an editor), make it executable, run it.

```
% vim hello.sh
i#!/bin/bash
echo hello world.
a=`echo hello world. | wc`
echo This phrase contains $a lines, words and characters<Esc>
:wq
%chmod ug+x hello.sh
%./hello.sh
hello.sh
hello world.
This phrase contains 1 2 13 lines, words and characters
%
```

(i and <Esc> etc. above in magenta don't show up on screen.)

Shell Scripting

The `#!` in the first line (known as shebang).

The first line can be used to tell the system what language (shell) to use for command interpretation.

It is a very specific format

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

or

```
#!/bin/csh –f
```

(the `–f` is optional for csh – gives "fast" initialization – see man page (`-f` Fast start. Reads neither the .cshrc file, nor the .login file (if a login shell) upon startup.))

If you want your shell script to use the same shell as the parent process you don't need to declare the shell with the shebang at the beginning.

BUT

You can't put a comment (indicated by #) in the first line.

So the first line has to be one of

#!/shell_to_use

or

Some command (not a comment, and not "shell_to_use" without the shebang)

Scripting Etiquette

Most scripts are read by both a person and a computer.

Don't ignore the person using or revising your script (most likely you 6 months later – when you will not remember what you did, or why you did it that way – especially if you were in a UNIX mood when you wrote it.)

Verify inputs for legality, print out error message if something wrong (!UNIX).

Advice

1. Use comments to tell the reader what they need to know. The # denotes a comment in bash and csh.
2. Use indentation to mark the various levels of program control. (loops, if-then-else blocks)
3. Use meaningful names for variables and develop a convention that helps readers identify their function.
4. Avoid unnecessary complexity…keep it readable (this rule is definitely not UNIX philosophy compatible).

Usually you will find the obvious stuff will be commented and described fully (as in homeworks).

The stuff the original author did not understand that well – but somehow got to work – will generally not be commented (or usefully commented, or may even be commented incorrectly!) (as in homeworks).

Header

Adding a set of comments at the beginning that provides information on

1. Name of the script
2. How the script is called
3. What arguments the script expects
4. What does the script accomplish
5. Who wrote the script and when
6. When was it revised and how

```
#!/usr/bin/bash -f
#Script: prepSacAVOdata.pl
#Usage:  $script <unixDir> <dataDir> <staFile> <phaseFile> <eventFile>
#-----------------------------------
#Purpose: To prepare SAC AVAO data for further processing
#    (1) generate event information file and add the event info
#           (name, event location) to the SAC headers
#    (2) generate event phase file and add the phase info
#           (time and weights) to the SAC headers
#Original Author (prepSacData.pl: Wen-xuan Du, Date: Mar. 18, 2003
#            Modified: May 21, 2004
#
#Last Modified by Heather DeShon  Nov. 30, 2004
#       A) Reads AVO archive event format directly (hypo71):
#              subroutines rdevent and rdphase
#       B) Reads SAC KZDATA and KZTIME rather than NZDTTM, which is
#              not set in AVO SAC data
. . .
```

Variables

A variable is used to store some piece of (typically character string) information.

The $ tells the shell to return the value of the specified variable.

csh example

```
%set b = "Hello world."
%set a = `echo $b | wc`
%echo $a
1 2 13
```

bash example

```
%b="Hello world."
%a=`echo $b | wc`
%echo $a
1 2 13
```

cs/csh and sh/bash have different syntax for assigning the value of a shell variable.

(in bash cannot have spaces on either side of the equals sign, csh does not care, works with our without spaces.)

Constants

A constant is used to store some piece of (typically character string) information that is not expected to change.

In bash, variables are made constants by using the readonly command.

```
% x=2
% readonly x
% x=4
-bash: x: readonly variable
%
```

`...`: backquotes/command substitution can be used in shell scripts.

The output of backquotes can go into a variable, switch, redirected input (<<), etc.

```
a=`echo ls`
echo $a
```

Reading command line arguments.

You can send your script input from the command line just like you do with built-in commands. It also gets environment variables from the shell.

```
517:> vi hi.sh
"hi.sh" [New file]
i #!/bin/bash
echo Hello, my name is $HOST. Nice to meet you $1.<Esc>
:wq
"hi.sh" [New file] 2 lines, 63 characters
518:> x hi.sh
519:> hi.sh Bob
Hello, my name is alpaca.ceri.memphis.edu. Nice to meet you Bob.
520:>
```

think of the command line as an array whose index starts with 0.
When you enter

```
%command arg1 arg2 arg3 arg4 . . . arg10 arg11 . . . Arg_end
```

The shell produces the following array that is passed to the shell script.

```
array[0]=command
array[1]=arg1
array[2]=arg2
. . .
array[end]=arg_end
```

Within the script, access to this array is accomplished using the syntax `$n`, where `n` is the array index.

```
$0=command
$1=arg1
$2=arg2
. . .
$9=arg9
${10}=arg10
${11}=arg11
```

note the format for numbers ≥10, the braces are required (they are optional for numbers ≤9)

Remember the discussion of identifying the shell you are running?

```
%echo $0
```

The shell is (just) a program.

Your shell receives these variables from its parent process, just like any other program.

So apply Unix think.

Reading user input

(even though it goes against the grain of Unix filter/think philosophy)

read: reads screen input into the specified variable.

Script – `introduce.sh`

```
#!/bin/bash
echo Please, enter your firstname and lastname
read FN LN
echo "Hi! $FN, $LN !"
```

Running it

```
528:> introduce.sh
Please, enter your firstname and lastname
Bob Smalley
Hi! Bob Smalley !
529:>
```

Reading (sucking in) multiple lines.
Use the syntax "<< eof".

Where eof defines the (character string) end-of-file delimiter.

This syntax redirects standard-in to the shell script (or the terminal if you are typing) until it finds the characters specified in the eof field. (You have to be sure those characters are not in the file/text being sucked in – else it will stop there.)

Example.
File - my_thoughts.dat

```
I have a thousand thoughts in my head
and one line of text is not enough to get them
all out.  Hello world.
```

Script - suckitin.sh

```
#!/bin/bash
cat << END
`cat my_thoughts.dat`
END
```

Run it

```
540:> suckitin.sh
I have a thousand thoughts in my head
and one line of text is not enough to get them
all out.  Hello world.
541:>
```

Note – we would never program something this way.

We could have just done

```
540:> cat my_thoughts.dat
```

But we are trying to demonstrate how input redirection (plus command substitution).

How does this script work?

```
#!/bin/bash
cat << END
`cat my_thoughts.dat`
END
```

The cat command reads standard-in, which is redirected, by the <<, to the lines that follow in the shell script (or the keyboard if not in a shell script).

We then use command substitution to produce input to the cat command from the file my_thoughts.dat.
Finally we terminate the input redirection with the string "END"

This is a very powerful way to process data.

```
my_processing_program << END
`my_convert_program input_file1`
`cat input_file2`
END
```

If we only needed to process file 1 (no file2), we could have used a pipe or input redirect

```
my_convert_program < input_file1 | my_processing_program
```

But there is no way (we have seen so far) to pipe both outputs into the program (the pipe is serial, not parallel).

Another example

```
my_processing_program << END
class example
10.3
41
`my_convert_program input file1`
`cat input_file2`
END
```

Here we have a character string input,
"class example",
some numbers,
followed by the other data.

Again we can not use a pipe.

(Also notice that, following the Unix philosophy, the program is not "interactive", it is not prompting for the inputs. You have to know what it wants and how it wants it.)

Another example

```
my_processing_program inputvari1 inputvari2 << END
$1
class example
10.3
41
`my_convert_program input file1`
`cat input_file2`
`ls $2`
END
```

Now we have added two inputs from the command line.

The first one puts the string `inputvari1` into `stdin` for my program to read

The second one puts the results of looking for a file called `inputvari2` into stdin for my program to read.

further examples: command substitution in conjunction with the gmt psxy command

```
#!/bin/sh
#missing beginning  and end of script. This command alone will
not work
psxy -R$REGN -$PROJ$SCALE $CONT -W1/$GREEN << END >> $OUTFILE
-69.5 -29.5
-65 -29.5
-65 -33.5
-69.5 -33.5
-69.5 -29.5
`cat my_map_file.dat`
END
```

This will read the data between the psxy command and the END and plot it on the map that is being constructed (the redirected, appended output).

further examples of <<:
running sac from within a script.

```
# Script to pick times in sac file using taup
#  Usage: picktimes.csh [directory name]
#
sacfile=$1

sac << EOF >&! sac.log
r $sacfile
sss
traveltime depth &1,evdp picks 1 phase P S Pn pP Sn sP sS
qs
w over
q
EOF
```

Shell Scripting

Loops and Logic

```
do
. . .
done
```

Does the commands in the "block" between `do` and `done`.

in bash, this construct is used in conjunction with loop structures `for`, `while`, and `until` and list based.

for:

A 'for loop' is a programming language statement which allows code to be repeatedly executed, looks like it is based on counting (this first example is really list based as we will see later).

```
for VARIABLE in 1 2 3 4 5 .. N
do
. . .
done
```

example

```
for i in 1 2 3 4 5
do
    echo "Welcome $i times"
done
```

More examples

```
for i in $(seq 1 2 20)
do
    echo "Welcome $i times”
done

for (( c=1; c<=5; c++ ))
do
   echo "Welcome $c times..."
done

for (( ; ; ))
do
    echo "infinite loops [ hit CTRL+C to stop]”
done
```

while:

Based on condition - continues to loop as long as the condition tests true

```
#!/bin/bash
. . .
while read vari1 vari2 … varin
do
    . . .
done < inputfile
```

This will read from the input file till it hits EOF (read returns 0, true, if there were no errors, on EOF [or an error] it returns a non zero value – false)

Full example

Script

```
#!/bin/bash
cat<<EOF>cities.dat
105.87 21.02 Hanoi LM
282.95 -12.1 LIMA LM
178.42 -18.13 SUVA LM
EOF

while read clon clat city junk
do
    echo $city $clon $clat
done < cities.dat
```

Run it:

```
516:> junk.sh
Hanoi 105.87 21.02
LIMA 282.95 -12.1
SUVA 178.42 -18.13
517:
```

This script first makes the input data file, then reads it and prints out a part of it. Notice where the redirected input is located – at the end of the "command" (can get confusing when many lines away from beginning).

The structure of the while loop

While the test is true, do the block of code between the "do" and "done"

```
while test
do

   . . . block of code . . .

done
```

The structure of the while loop

The redirected input goes at the end.

As we saw before, one can enter the while command from the command line
(there is nothing special about it as far as the shell is concerned)
(also notice where the semicolons, that separate lines, and the input redirect goes).

```
%while read line; do echo "$line \n"; done < cities.dat
105.87 21.02 Hanoi LM \n
282.95 -12.1 LIMA LM \n
178.42 -18.13 SUVA LM \n
%
```

until:

Based on condition - until continues to loop as long as the condition exits unsuccessfully (is false)

(the until loop is used much less than the while loop)

```
#!/bin/bash
myvar=0
until [ $myvar –eq 5 ]     #until this expression is true
do
echo $myvar
myvar=$(( $myvar + 1 ))
done

% sh –f junk.sh
0
1
2
3
4
```

Break:

allows you to break out of a loop

can be used with a number to specify what do loop to break out of

```
while condition1        # Outer loop, loop 2
do
…
while condition2        # Inner loop, loop 1
do
…
break 2                 # Break out of outer loop (usually after
                                                    some test)
done
done
...                     # Execution continues here after break
```

List based - using a list to provide the items to loop over.

```
list=`ls z*xyz`
for ITEM in $list
do
 #echo plot contour $ITEM
 psxy -R$REGION -$PROJ$SCALE -M$ -W5/$VLTGRAY $CONTINUE\
    $ITEM $VBSE >> $OUTPUTFILE
Done
```

(in the first example I wrote out the list

```
for VARIABLE in 1 2 3 4 5 .. N)
```

```
-bash 623 rinex # fs=`ls anit*o`
-bash 624 rinex # echo $fs
anit0770.11o anit0780.11o
-bash 625 rinex # for f in $fs
do head -10 $f
done
     2.11           OBSERVATION DATA    G (GPS)             RINEX VERSION / TYPE
GPP.DLL V3.00                          15 - APR - 11 04:25 PGM / RUN BY / DATE
ANIT                                                       MARKER NAME
                                                           MARKER NUMBER
                                                           OBSERVER / AGENCY
                    Z-XII3             1D02                REC # / TYPE / VERS
                                                           ANT # / TYPE
        0.0000        0.0000        0.0000                 APPROX POSITION XYZ
        0.0000        0.0000        0.0000                 ANTENNA: DELTA H/E/N
     1     1                                               WAVELENGTH FACT L1/2
     2.11           OBSERVATION DATA    G (GPS)             RINEX VERSION / TYPE
GPP.DLL V3.00                          15 - APR - 11 04:25 PGM / RUN BY / DATE
ANIT                                                       MARKER NAME
                                                           MARKER NUMBER
                                                           OBSERVER / AGENCY
                    Z-XII3             1D02                REC # / TYPE / VERS
                                                           ANT # / TYPE
        0.0000        0.0000        0.0000                 APPROX POSITION XYZ
        0.0000        0.0000        0.0000                 ANTENNA: DELTA H/E/N
     1     1                                               WAVELENGTH FACT L1/2
-bash 626 rinex #
```

if/then/elif/else/fi

If the test is true, then run the block of code between the then and the fi (if spelled backwards – logical way to signify the end of an if block).

```
if    [ $1 = "Heather" ]
then
    printf "Hi %s.  We were expecting you.\n" $1
fi
```

if/then/elif/else/fi

If the test is true, run block of code between then and else. If the test is false, run block of code between else and fi.

```
if    [ $1 = "Heather" ]
then
    printf "Hi %s.  We were expecting you.\n" $1
else
    printf "Hi %s.  Nice to meet you.\n" $1
fi
```

if/then/elif/else/fi

If [test] is true, run block of code between then and elif. If it was false, do next [test]. If true, run block of code between else and elif. If false, do next [test], etc., or, finally (everything false to here) do block of code between else and fi.

```
if   [ $1 = "Heather" ]
then
    printf "Hi %s.  We were expecting you.\n" $1
elif [ $1 = "Andy"  ]
    printf "Hi %s.  We were expecting you too.\n" $1
elif [ $1 = "Gregg"  ]
    printf "Hi %s.  We were expecting you too.\n" $1
else
    printf "Hi %s.  Nice to meet you.\n" $1
fi
```

Can have logical combination of [tests]

|| is or, && is and

```
if    [ $1 = "Heather" ] || [ $1 = "Andy" ]
then
    printf "Hi %s.  We were expecting you.\n" $1
elif [ $1 = "Andy"  ]
    printf "Hi %s.  We were expecting you too.\n" $1
else
    printf "Hi %s.  Nice to meet you.\n" $1
fi
```

```
if   [ $1 = "Heather" ] || [ $1 = "Andy" ]
then
    printf "Hi %s.  We were expecting you.\n" $1
elif [ $1 = "Andy"  ]
    printf "Hi %s.  We were expecting you too.\n" $1
else
    printf "Hi %s.  Nice to meet you.\n" $1
fi
```

Would this script ever output "We were expecting you too"?

(i.e. what is wrong with it?)

NOTE

The formatting with respect to spaces, and lines of the "`if [ ]`", "`then`", ("`else`", "`elif`"), "`fi`" are very specific.

```
if [ . . . ]
then
      . . .
fi
```

It has to be written exactly as above – where the test and code to be performed replace the ". . ." .

(look back at previous slide to see where spaces go)

The part in red is the test, it may be different (but still a rigid format)

Example - Do (at least simple) error checking of a call and print some sort of message for error.

```
if [ $# -ne 5 ]
then
        printf "Usage:\t\t$script <unixDir> <dataDirList>\
<staFile> <phaseFile> <eventFile>\n"
        printf "<unixDir>:\tdirectory in the unix system\ where
pick file is stored;\n"
        printf "<dataDirList>:\tlist of data directories\ under
<unixDir>; 'dataDir.list';\n"
        printf "<staFile>:\tstation file;\n"
        printf "<phaseFile>:\tphase info file for program\
'ph2dt';\n"
        printf "<eventFile>:\tevent info file (one line for\
 one event);\n"
exit (-1)
fi
```

if does everything between "then" and "fi" (in the box) if the test is true. We will get to the test later.

Check there are 5 input parameters.

$ to have shell return the value of a variable.

is the shell variable (the shell gives you this variable when it starts a script) that contains the number of parameters (does not include the shell name) on input line.

-ne is the numerical test for not equal (also have alphabetical tests) (-eq is the numerical test for equal).

```
if [ $# -ne 5 ]
then
. . .
fi
```

So if the number of input parameters is not 5, it will do what is between "then" and "fi".

One of the things done in the error processing (in the box) is the command "`exit(-1)`".

```
if [ $# -ne 5 ]
then
        . . .
        exit (-1)
fi
```

This returns a message, a numeric "return value" (in this case a `-1`) to the parent process.

The parent process can access this return value using the shell variable `?`
(to obtain the value one uses `$?` (of course)).

This allows the parent process to get information about what happened in the daughter process.

You can set the return code to give you information about the type or error, etc.

This information can be used to control the execution of the parent process.
(does the parent process continue, quit, try to fix it, etc.?)

Many programs return a value of 0 (zero) upon successful completion.

From the ls man page -

```
EXIT STATUS
     0     All information was written successfully.

     >0    An error occurred.
```

So we can tell if it terminated successfully (but not what the error was if not).

The case statement is an elegant replacement for if/then/else if/else statements when making numerous comparisons.

This recipe describes the case statement syntax for the Bourne family of shells

```
case "$var" in
value1)
commands for value 1;
;;
value2)
commands for value 2;
;;
*)
commands for every other value (did not do any of the above);
;;
esac
```

```
case "$var" in
value1)
commands;
;;
*)
commands;
;;
esac
```

The case statement compares the value of the variable (`$var` in this example) to one or more values (`value1, value2, …`).
Once a match is found, the associated commands are executed and the case statement is terminated.
The optional last comparison "`*)`" is a default case and will match anything.

For example, branching on a command line parameter to the script, such as 'start' or 'stop' with a runtime control script.

The following example uses the first command line parameter ($1):

```
case "$1" in
'start')
/usr/app/startup-script
;;
'stop')
/usr/app/shutdown-script
;;
'restart')
echo "Usage: $0 [start|stop]"
;;
esac
```

Shell Scripting

Intro – relational and logical operators, test

`test`

`Test or [...]`: condition evaluation utility

common scripting tool that tests expressions and many details about files using a long list of flags

Returns

0 if expression true and
1 if expression false or does not exist
(backwards to normal logic!)

test
two formats in bash scripting

`test [flags] expression`

```
test ! -s "$1"; echo $?
0
```

or

`[ expression ]`

```
bash-2.05$ [ 'abc' == 'abc' ]; echo $?
0
bash-2.05$ [ 'abc' = 'abc' ]; echo $?
0
bash-2.05$ [ "abc" != "def" ];echo $?
0
```

Note – we are testing character strings.

[(the left bracket special character) is a dedicated command. It is a synonym for **test**, and a builtin for efficiency reasons.

(you also need the closing])

Relational Operators (between character strings)

Returns 1 if true and 0 if false

All relational operators are left to right associative.

= or == : test for equal to

< : test for less than

> : test for greater than

!= : test for not equal

To test numerical values

```
$ test 3 -gt 4, echo $?
1
$ [ 3 -gt 4 ], echo $?
1
```

Note – the numerical tests are specified with a different format (Fortran like).

Returns

0 if expression true and
1 if expression false or does not exist
(backwards to normal logic!)

Relational Operators (between numerical values)

Returns 1 if true and 0 if false

All relational operators are left to right associative

```
-lt (<)
-gt (>)
-le (<=)
-ge (>=)
-eq (==)
-ne (!=)
```

```
bash-2.05$ a=1
bash-2.05$ b=2
bash-2.05$ c=3
bash-2.05$ [ $a = 1 ];echo $?
0
bash-2.05$ [ $a -eq 1 ];echo $?
0
bash-2.05$ [ $a > 1 ];echo $?
0
bash-2.05$ [ $a \> 1 ];echo $?
1
bash-2.05$ [[ $a > 1 ]];echo $?
1
bash-2.05$ [ $a -gt 1 ];echo $?
1
bash-2.05$ [ $b -eq 1 ];echo $?
1
bash-2.05$ [ $b -eq $c ];echo $?
1
bash-2.05$ [ $b -eq $(($c-1)) ];echo $?
0
bash-2.05$ [ $b == $(($c-1)) ];echo $?
0
bash-2.05$ [ $b == $(($c-2)) ];echo $?
1
```

Or `[ $a = 1 ]`

What is this? Seems to say it is true?

Needs to be escaped, but why?

Why works with double brackets w/o excape?

Test combinations with

-a (and) and –o (or)

```
if [ $# -eq 0 -o $# -ge 3 ]
then
. . .
fi

if [ \( $REGPARM = spat -o $REGPARM = chile \) -a $CMT = 1 ]
then
. . .
fi
```

(the [. . .]'s above are a form of the test expression)

(the backslashes are needed to "escape" the parentheses in the test expression)

You can use the return values together with && and || using the two test constructs

examples

```
$ test 3 -gt 4 && echo True || echo false
false
$[ $a = 1 ]&&[ $b == $(($c-1)) ];echo $?
0
$[ $a = 1 ]&&[ $b == $(($c-1)) ]&&[ $b -eq $c ];echo $?
1
$[ $a = 1 ]&&[ $b == $(($c-1)) ]||[ $b -eq $c ];echo $?
0
$[ $a = 1 ]&&( [ $b == $(($c-1)) ]||[ $b -eq $c ] );echo $?
0
$[ $a = 1 ]||( [ $b == $(($c-1)) ]&&[ $b -eq $c ] );echo $?
0
```

Some tests

-d Directory

-e Exists (also -a)

-f Regular file

-h Symbolic link (also -L)

(remember 0 is TRUE and 1 if FALSE!!!)

```
$ [ -e 'eqs.vim' ]; echo $?
0
$ [ -e 'eqs' ]; echo $?
1
$ filename=eqs.vim
$ echo $filename
eqs.vim
$ [ -e $filename ]; echo $?
0
$ test -d "$HOME" ;echo $?
0
```

More fun with syntax

The [[...]] construct is the more versatile Bash version of [...].

It is know as the *extended test command*,

(although [[is a keyword, not a command).

No filename expansion or word splitting takes place between [[and]], but there is parameter expansion and command substitution.

More fun with syntax

Using the `[[ ... ]]` test construct, rather than `[ ... ]` can prevent many logic errors in scripts.

For example, the `&&`, `||`, `<`, and `>` operators work within a `[[ ]]` test, despite giving an error within a `[ ]` construct.

```
569 $ decimal=15
570 $ octal=017   # = 15 (decimal)
571 $ hex=0x0f    # = 15 (decimal)
572 $ if [ "$decimal" -eq "$octal" ]
> then
>   echo "$decimal equals $octal"
> else
>   echo "$decimal is not equal to $octal"       # 15 is not equal to 017
> fi      # Doesn't evaluate within [ single brackets ]!
15 is not equal to 017
573 $ if [[ "$decimal" -eq "$octal" ]]
> then
>   echo "$decimal equals $octal"                 # 15 equals 017
> else
>   echo "$decimal is not equal to $octal"
> fi      # Evaluates within [[ double brackets ]]!
15 equals 017
574 $
574 $ if [[ "$decimal" -eq "$hex" ]]
> then
>   echo "$decimal equals $hex"                   # 15 equals 0x0f
> else
>   echo "$decimal is not equal to $hex"
> fi      # [[ $hexadecimal ]] also evaluates!
15 equals 0x0f
575 $
```

More fun with syntax

Similar to the let command, the double parentheses `(( ... ))` construct permits arithmetic expansion and evaluation.

In its simplest form, `a=$(( 5 + 3 ))` would set `a` to `5 + 3`, or `8`.

However, this double-parentheses construct is also a mechanism for allowing C-style manipulation of variables in Bash, for example, `(( var++ ))`.

```
575 $ var=1
576 $ ((var++))
577 $ echo $var
2
578 $ ((var>3));echo $?
1
579 $ ((var==2));echo $?
0
580 $ ((var=1));echo $?
0
581 $ echo $var
1
582 $
```

Without the $ the ((construct returns the exit status of the mathematical or logical operation. With the $ it returns the value (and you still have the exit status).

```
$ B=$((A + 1)); echo $?, $A, $B
0, 2, 3
$ A=$((var++));echo $?, $A
0, 3
```

For completeness since I mentioned it

The **let** command carries out *arithmetic* operations on variables.

In many cases, it functions as a less complex version of expr.

```
let a=11                    # Same as 'a=11'
let a=a+5                   # Equivalent to let "a = a + 5" #
echo "11 + 5 = $a"  # 16
let "a <<= 3"               # Equivalent to let "a = a << 3"
```

Relational Operators (in arithmetic expressions $((. . .)))

Returns 1 if true and 0 if false

All relational operators are left to right associative

== : test for equal to
< : test for less than
<= : test for less than or equal to
> : test for greater than
>= : test for greater than or equal to
!= : test for not equal

Bash does not understand floating point arithmetic.

It treats numbers containing a decimal point as strings.

Boolean (Logical) Operators

Boolean operators return 1 for true and 0 for false

&& : logical AND

tests that both expressions are true left to right associative

```
%echo $((   (3 < 4) && (10<15) ))
1
%echo $(( (3<4) && (10>15) ))
0
```

|| : logical OR

tests that one or both of the expressions are true
left to right associative.

```
%echo $(( (3<4) || (10>15) ))
1
```

! : logical negation

tests negation of expression.

Bitwise Operators

Bitwise Operators treat operands as 16 (actually depends on word size on computer) bit binary values

Example: 4019 equals 0000111110110011_{base2} ($0FB3_{16}$ in hexadecimal) in integer format.

(Internally in the computer, integers are expressed in a format called two's-complement. Positive integers are in straight base 2. Negative integers are "funny".)

Bitwise Operators

~ : bitwise negation changes 0's to 1's (bits) and vice versa

& : bitwise AND

^ : bitwise exclusive OR

| : bitwise OR

<< : bitwise left shift (numerically is *2)

<<=n : bitwise left shift by n bits (numerically is $*2^n$)

>> : bitwise right shift (numerically is $\div 2^n$)

<<=n : bitwise left shift by n bits (numerically is $\div 2^n$)

Shell Scripting

Intro - arithmetic

Arithmetic
bash shell arithmetic resembles C programming language arithmetic

(very helpful if you don't already know C!) .

In bash, the syntax $(( )) can be used to calculate arithmetic expressions or to set variables to complex arithmetic expressions

```
%echo $((3+4))
7
%echo $((x=2))
2
%echo $((++x))
3
%echo $((x++))
3
```

```
%echo $x
4
%((y=10))
%echo $y
10
```

Basic Arithmetic Operators

shell arithmetic is integer only

\+ : addition
\- : subtraction
* : multiplication
/ : division
% : remainder or modulus

```
% echo $(( 10%3 ))
1
% echo $(( 10/3 ))
3
```

Assignment Operators

= : set variable equal to value on right
(no spaces allowed around equals sign)

```
%x=2; echo $x
2
```

+= : set variable equal to itself plus the value on right (spaces allowed, but not required)

```
%x=2; echo $(( x +=2 ))
4
```

-= : set variable equal to itself minus the value on right (spaces allowed, but not required)

```
%x = 2; echo $((x-=2))
0
```

Assignment Operators

*= : set variable equal to itself times the value on right (spaces allowed, but not required).

```
%x = 2; echo $((x *= 4))
8
```

Assignment Operators

/= : set variable equal to itself divided by value on right (spaces allowed, but not required).

```
%x = 2; echo $((x/= 2))
1
```

%= : set variable equal to the remainder of itself divided by the value on the right

```
%x = 4; echo $((x %= 3))
1
```

Unary Operations

A unary expression contains one operand and one operator.

++ : increment the operand by 1

Unary Operations

if ++ occurs after the operand, `$x++`, the original value of the operand is used in the expression and then incremented.

if ++ occurs before the operand, `++$x`, the incremented value of the operand is used in the expression.

Unary Operations

`--` : decrement the operand by 1

`+` : unary plus maintains the value of the operand,
`x=+x`

`-`: unary minus negates the value of the operand,
`-1*x=-x`

`-!` : logical negation

Some tcsh/csh syntax

A shell with C language-like syntax.

Control structures

- foreach, if, switch and while

foreach : a tcsh command

is a powerful way to iterate over files from the tcsh command line (can also put in shell scripts – don't get prompts).

```
%foreach file ( 828/*BHZ* )#set variable file to each sac file
foreach? echo $file
foreach? set name = `echo $file | cut -f2 -d'/' `
foreach? set sta = `echo $name | cut -f1 -d'.' `
foreach? echo "copy $file to $sta.BHZ.SAC
foreach? cp $file $sta.BHZ.SAC
foreach? end
828/GAR.BHZ_00.D.1989.214:10.24.59
copy 828/GAR.BHZ_00.D.1989.214:10.24.59 to GAR.BHZ.SAC
```

Aside – new command

cut

The cut command has the ability to cut out characters or fields. cut uses delimiters.

```
file = 828/GAR.BHZ_00.D.1989.214:10.24.59

Set name = `echo $file | cut -f2 -d'/'`
```

Says return the second field (-f2), using '/' as a delimiter (-d'/') (assign it to the variable name)

```
name = GAR.BHZ_00.D.1989.214:10.24.59

set sta = `echo $name | cut -f1 -d'.' `
```

Says return the first field (-f1), using '.' as a delimiter (-d'.') (assign it to the variable sta)

If-then-else block in tcsh/csh

Two formats

```
if (expression) simple command
```

or

```
if (expression) then
  ...
else
  ...
endif
```

The tcsh/csh switch statement can replace several if ... then statements.

```
switch (string)
  case pattern1:
    commands...
    breaksw
  case pattern2:
    commands...
    breaksw
  default:
    commands...
    breaksw
endsw
```

For the string given in the switch statement's argument, commands following the case statement with the matching pattern are executed until the endsw statement.

These patterns may contain ? and * to match groups of characters or specific characters.

switch/case in tcsh syntax

```
foreach plane(0035.0 0050.0)
set cnt=`expr $cnt + 1`
switch ($cnt)
    case 1:
        set xpos=-5.
        set ypos=4.75
        set min=-2.5
        set max=2.5
        breaksw
    case 2:
        set xpos=-6.6
        set ypos=-3.5
        set min=2.5
        set max=7
        breaksw
    endsw
. . . such as excessive amounts of GMT
end
```

Another example

```
# Get the arguments
set source_dir = $1
set target_dir = $2
shift argv
shift argv
while ($#argv > 0 )
    set input = ( $argv )
    switch($input[1])
       case -m:
          set module = $input[2]
          breaksw
       case -auto:
          set auto = 'Y'
          breaksw
       case -full:
          set full = 'Y'
          breaksw
    endsw
    shift argv
end
```

Built-in shell variables

argv Special variable used in shell scripts to hold the value of command line arguments.