Linux Shell Scripting

Course audience

Audience: Linux users, programmers, and system administrators.

Prerrequisite: Basic knowledge of the Linux command line

- Students learn to read, write, and debug Linux shell scripts, thus increasing productivity by taking full advantage of the bash shell.
- Linux Shell scripts, are the means by which a Linux shell is used as a programming language. Linux commands and shell language control constructs are entered into a file by the programmer, then the file is executed as a command and interpreted just as if the commands had been typed on the shell command line.
- Linux shell scripts provide a way to automate commonly executed groups of commands – but shell scripts can do much more than this. Although many simple tasks are automated with small scripts, large scripts hundreds of lines long are very common.

What is a shell?

- In computing, a shell is a user interface for access to an operating system services. In general, operating system shells use either a command-line interface (CLI) or graphical user interface (GUI), depending on the computer's role and particular operation. It is named a shell because it is the outermost layer around the operating system kernel.
- A program that interprets commands.
- Allows a user to execute commands by typing them manually at a terminal, or automatically in programs called shell scripts.
- A shell is not an operating system. It is a way to interface with the operating system and run commands.

Alternate names: console, terminal, command line, command line interface, command prompt

Shell types

Just like people know different languages and dialects, your UNIX system will usually offer a variety of shell types:

sh or Bourne Shell: the original shell still used on UNIX systems and in UNIX−related environments. This is the basic shell, a small program with few features. While this is not the standard shell anymore, it is still available on every Linux system for compatibility with UNIX programs.

bash or Bourne Again shell: the standard GNU shell, intuitive and flexible. Probably most advisable for beginning users while being at the same time a powerful tool for the advanced and professional user. On Linux, bash is the standard shell for common users. This shell is a so−called superset of the Bourne shell, a set of add−ons and plug−ins. This means that the Bourne Again shell is compatible with the Bourne shell: commands that work in sh, also work in bash. However, the reverse is not always the case. All examples and exercises in this course use bash.

csh or C shell: the syntax of this shell resembles that of the C programming language. Sometimes asked for by programmers.

tcsh or Turbo C shell: a superset of the common C shell, enhancing user−friendliness and speed.

ksh or the Korn shell: sometimes appreciated by people with a UNIX background. A superset of the Bourne shell; with standard configuration a nightmare for beginning users.

The file /etc/shells gives an overview of known shells on a Linux system:

```
$ cat /etc/shells
/bin/sh
/bin/bash
/usr/bin/sh
/usr/bin/bash
/bin/tcsh
/bin/csh
/bin/zsh
/bin/ksh
/bin/rksh
```

Shell types

Just like people k[illegible]anguages and dialects, your UNIX system will usually offer a variety of shell types:

**sh or Bourne S[illegible]al shell still used on UNIX systems and in UNIX−related environments. This is the basic shell, [illegible]w features. While this is not the standard shell anymore, it is still available on every Linux system [illegible]patibility with UNIX programs.

bash or Bourne Again shell: the standard GNU shell, intuitive and flexible. Probably most advisable for beginning users while being at the same time a powerful tool for the advanced and professional user. On Linux, bash is the standard shell for common users. This shell is a so−called superset of the Bourne shell, a set of add−ons and plug−ins. This means that the Bourne Again shell is compatible with the Bourne shell: commands that work in sh, also work in bash. However, the reverse is not always the case. All examples and exercises in this course use bash.

csh or C shell: the syntax of this shell resembles that of the C programming language. Sometimes asked for by programmers.

tcsh or Turbo C shell: a superset of the common C shell, enhancing user−friendliness and speed.

ksh or the Korn shell: sometimes appreciated by people with a UNIX background. A superset of the Bourne shell; with standard configuration a nightmare for beginning users.

The file /etc/shells gives an overview of known shells on a Linux system:

```
$ cat /etc/shells
/bin/sh
/bin/bash
/usr/bin/sh
/usr/bin/bash
/bin/tcsh
/bin/csh
/bin/zsh
/bin/ksh
/bin/rksh
```

What is a shell script?

A shell script is a computer program designed to be run by the Unix shell, a command-line interpreter. The various dialects of shell scripts are considered to be scripting languages. Typical operations performed by shell scripts include file manipulation, program execution, and printing text. A script which sets up the environment, runs the program, and does any necessary cleanup, logging, etc. is called a wrapper. (Wikipedia)

Basically, is a text file containing commands and their respective flags and parameters, line by line, in the order that we want them to be executed.

```
#!/bin/bash
# My first script
command1 –flag parameter1
command2 –otherflag otherparameter
exit
```

Starting Off With a Sha-Bang

A sha-bang is the character sequence consisting of the characters number sign and exclamation mark (#!) at the beginning of a script. It is also called shebang, hashbang, pound-bang, or hash-pling.

The sha-bang (#!) at the head of a script tells your system that this file is a set of commands to be fed to the command interpreter indicated. Immediately following the sha-bang is a path name. This is the path to the program that interprets the commands in the script, whether it be a shell, a programming language, or a utility. This command interpreter then executes the commands in the script, starting at the top (the line following the sha-bang line), and ignoring comments.

```
#!/bin/sh
#!/bin/bash
#!/usr/bin/perl
#!/usr/bin/tcl
#!/bin/sed -f
#!/bin/awk -f
#!/bin/expect
#!/usr/bin/python
```

Starting Off With a Sha-Bang

The sha-bang is only mandatory for those scripts, which shall be executed by the operating system in the same way as binary executables. If you source in another script, then the sha-bang is ignored.

The sha-bang is valid and read by the system **ONLY** from the first line of the script. If there is another sha-bang in any line below, it is treated as a comment.

```
#!/bin/sh
#!/bin/bash
#!/usr/bin/perl
#!/usr/bin/tcl
#!/bin/sed -f
#!/bin/awk -f
#!/bin/expect
#!/usr/bin/python
```

My first script – Hello World!

Using your preferred text editor, (vi, emacs, nano) write the following lines:

```
#!/bin/bash
# This is my first script
echo “Hello World!”
exit 0
```

Save it as “myfirst.sh”

You can name your script whatever you want. As a good practice, always name your scripts in a descriptive manner so you know what the script does.

In the example above, we are appending “.sh” to the script name. There are no extensions in Linux and the dot “.” character is just another valid character in the name of the object, so the “.sh” is just there to remind us that this is a script, but it is not mandatory. The name of the script could be just “myfirst” and it will work the same way.

Invoking the script

Having written the script, you can invoke it by

```
$ bash myfirst.sh
```

Much more convenient is to make the script itself directly executable with a chmod:

```
$ chmod +x myfirst.sh
```

The "sha-bang" line, invoking the script calls the correct command interpreter to run it.

As a final step, after testing and debugging, you would likely want to move it to /usr/local/bin (as root, of course), to make the script available to yourself and all other users as a systemwide executable. The script could then be invoked by simply typing `myfirst.sh <ENTER>` from the command-line.

Special Characters

What makes a character special? If it has a meaning beyond its literal meaning, a meta-meaning, then we refer to it as a special character. Along with commands and keywords, special characters are building blocks of Bash scripts.

Special Characters

Comments. Lines beginning with a #, or anything to the right of a #, are comments and will not be executed:

This line is a comment.

Command separator [semicolon]. Permits putting two or more commands on the same line.

$ echo hello; echo there

A quoted or an escaped # in an echo statement does not begin a comment. Likewise, a # appears in certain parameter-substitution constructs and in numerical constant expressions.

```
echo "The # here does not begin a comment.”
echo 'The # here does not begin a comment.’
echo The \# here does not begin a comment.
echo The # here begins a comment.
echo ${PATH#*:} # Parameter substitution, not a comment.
echo $(( 2#101011 )) # Base conversion, not a comment.
```

Special Characters

Command separator [semicolon]. Permits putting two or more commands on the same line.

```
$ echo hello; echo there
```

Wild card [asterisk]. The * character serves as a "wild card" for filename expansion in globing by itself, it matches every filename in a given directory.

```
$ echo *
abs-book.sgml add-drive.sh agram.sh alias.sh
```

Special Characters

"dot" command [period]. Equivalent to `source`. This is a bash builtin.

"dot", as a component of a filename. When working with filenames, a leading dot is the prefix of a "hidden" file, a file that an ls will not normally show.

escape [backslash]. A quoting mechanism for single characters.
\X escapes the character X. This has the effect of "quoting" X, equivalent to 'X'. The \ may be used to quote " and ', so they are expressed literally.

Variables and Parameters

Variables are how programming and scripting languages represent data. A variable is nothing more than a label, a name assigned to a location or set of locations in computer memory holding an item of data.

Variables appear in arithmetic operations and manipulation of quantities, and in string parsing.

Unlike many other programming languages, Bash does not segregate its variables by "type." Essentially, Bash variables are character strings, but, depending on context, Bash permits arithmetic operations and comparisons on variables. The determining factor is whether the value of a variable contains only digits.

Variable Substitution

The name of a variable is a placeholder for its value, the data it holds. Referencing (retrieving) its value is called variable substitution.

Let us carefully distinguish between the name of a variable and its value. If **variable1** is the name of a variable, then **$variable1** is a reference to its value, the data item it contains.

```
$ variable1=23
$ echo variable1
variable1
$ echo $variable1
23
```

The only times a variable appears "naked" -- without the $ prefix -- is when declared or assigned, when unset, when exported, in an arithmetic expression within double parentheses ((...)), or in the special case of a variable representing a signal. Assignment may be with an = (as in var1=27), in a read statement, and at the head of a loop (for var2 in 1 2 3).

Special Variable Types

Local variables

Variables visible only within a code block or function (see also local variables in functions)

Environmental variables

Variables that affect the behaviour of the shell and user interface.

Positional parameters

Arguments passed to the script from the command line: $0, $1, $2, $3 . . .

$0 is the name of the script itself, $1 is the first argument, $2 the second, $3 the third, and so forth. After $9, the arguments must be enclosed in brackets, for example, ${10}, ${11}, ${12}.

The special variables $* and $@ denote all the positional parameters.

Quoting

Quoting means just that, bracketing a string in quotes. This has the effect of protecting special characters in the string from reinterpretation or expansion by the shell or shell script. (A character is "special" if it has an interpretation other than its literal meaning. For example, the asterisk * represents a wild card character in globing and Regular Expressions).

Quoting Variables

When referencing a variable, it is generally advisable to enclose its name in double quotes. This prevents reinterpretation of all special characters within the quoted string -- except $, ` (backquote), and \ (escape). Keeping $ as a special character within double quotes permits referencing a quoted variable ("$variable"), that is, replacing the variable with its value.

Exit and Exit Status

The exit command terminates a script, just as in a C program. It can also return a value, which is available to the script's parent process.

Every command returns an exit status (sometimes referred to as a return status or exit code). A successful command returns a 0, while an unsuccessful one returns a non-zero value that usually can be interpreted as an error code. Well-behaved UNIX commands, programs, and utilities return a 0 exit code upon successful completion, though there are some exceptions.

Tests

Every reasonably complete programming language can test for a condition, then act according to the result of the test. Bash has the test command, various bracket and parenthesis operators, and the if/then/else construct.

An if/then construct tests whether the exit status of a list of commands is 0 (since 0 means "success" by UNIX convention), and if so, executes one or more commands.

Tests

There exists a dedicated command called [(left bracket special character). It is a synonym for test, and a builtin for efficiency reasons. This command considers its arguments as comparison expressions or file tests and returns an exit status corresponding to the result of the comparison (0 for true, 1 for false).

With version 2.02, Bash introduced the [[...]] extended test command, which performs comparisons in a manner more familiar to programmers from other languages. Note that [[is a keyword, not a command.

Bash sees [[$a -lt $b]] as a single element, which returns an exit status.

The ((...)) and let ... constructs return an exit status, according to whether the arithmetic expressions they evaluate expand to a non-zero value. These arithmetic-expansion constructs may therefore be used to perform arithmetic comparisons.

File test operators

Returns true if...

-e	file exists
-a	file exists
-f	file is a regular file (not a directory or device file or a symbolic link)
-s	file is not zero size
-d	file is a directory
-b	file is a block device
-c	file is a character device
-p	file is a pipe
-h	file is a symbolic link
-L	file is a symbolic link
-S	file is a socket
-t	file (descriptor) is associated with a terminal device
-r	file has read permission (for the user running the test)
-w	file has write permission (for the user running the test)
-x	file has execute permission (for the user running the test)
-g	set-group-id (sgid) flag set on file or directory
-u	set-user-id (suid) flag set on file
-k	sticky bit set
-O	you are owner of file
-G	group-id of file same as yours
-N	file modified since it was last read
f1 -nt f2	file f1 is newer than f2
f1 -ot f2	file f1 is older than f2
f1 -ef f2	files f1 and f2 are hard links to the same file
!	"not" -- reverses the sense of the tests above (returns true if condition absent)

Other Comparison Operators

Integer comparison

-eq	is equal to if ["$a" -eq "$b"]
-ne	is not equal to if ["$a" -ne "$b"]
-gt	is greater than if ["$a" -gt "$b"]
-ge	is greater than or equal to if ["$a" -ge "$b"]
-lt	is less than if ["$a" -lt "$b"]
-le	is less than or equal to if ["$a" -le "$b"]

<	is less than (within double parentheses) (("$a" < "$b"))
<=	is less than or equal to (within double parentheses) (("$a" <= "$b"))
>	is greater than (within double parentheses)(("$a" > "$b"))
>=	is greater than or equal to (within double parentheses) (("$a" >= "$b"))

Other Comparison Operators

String comparison

= is equal to
if ["$a" = "$b"]
Note the whitespace framing the =
if ["$a"="$b"] is not equivalent to the above.

== is equal to
if ["$a" == "$b"]
This is a synonym for =

!= is not equal to
if ["$a" != "$b"]

< is less than, in ASCII alphabetical order
if [["$a" < "$b"]]
if ["$a" \< "$b"]
Note that the "<" needs to be escaped within a [ ] construct.

> is greater than, in ASCII alphabetical order
if [["$a" > "$b"]]
if ["$a" \> "$b"]
Note that the ">" needs to be escaped within a [ ] construct.

-z string is null, that is, has zero length

-n string is not null.

Compound comparison

-a logical and
exp1 -a exp2 returns true if both exp1 and exp2 are true.

-o logical or
exp1 -o exp2 returns true if either exp1 or exp2 is true.

These are similar to the Bash comparison operators && and ||, used within double brackets.

Operators

Assignment

variable assignment

Initializing or changing the value of a variable

=

All-purpose assignment operator, which works for both arithmetic and string assignments.

Do not confuse the "=" assignment operator with the = test operator.

Arithmetic operators

+	plus
-	Minus
*	multiplication
/	division
**	exponentiation
%	modulo, or mod (returns the remainder of an integer division operation)

Operators

Bitwise operators

<<	bitwise left shift (multiplies by 2 for each shift position)
<<=	left-shift-equal let "var <<= 2" results in var left-shifted 2 bits (multiplied by 4)
>>	bitwise right shift (divides by 2 for each shift position)
>>=	right-shift-equal (inverse of <<=)
&	bitwise AND
&=	bitwise AND-equal
|	bitwise OR
|=	bitwise OR-equal
~	bitwise NOT
^	bitwise XOR
^=	bitwise XOR-equal

Logical (boolean) operators

!	NOT
&&	AND
||	OR

Arithmetic Expansion

Arithmetic expansion provides a powerful tool for performing (integer) arithmetic operations in scripts. Translating a string into a numerical expression is relatively straightforward using backticks, double parentheses, or let.

Arithmetic expansion with backticks (often used in conjunction with expr):

```
z=`expr $z + 3` # The 'expr' command performs the expansion.
```

Arithmetic expansion with double parentheses, and using let:

The use of backticks (backquotes) in arithmetic expansion has been superseded by double parentheses -- ((...)) and $((...)) -- and also by the very convenient let construction.

```
z=$(($z+3))
z=$((z+3))
let z=z+3
```

Numerical Constants

A shell script interprets a number as decimal (base 10), unless that number has a special prefix or notation. A number preceded by a 0 is octal (base 8). A number preceded by 0x is hexadecimal (base 16). A number with an embedded # evaluates as BASE#NUMBER (with range and notational restrictions).

Here Documents

A here document is a special-purpose code block. It uses a form of I/O redirection to feed a command list to an interactive program or a command, such as ftp, cat, or the ex text editor.

```
COMMAND <<InputComesFromHERE ...
...
...
InputComesFromHERE
```

A limit string delineates (frames) the command list. The special symbol << precedes the limit string. This has the effect of redirecting the output of a command block into the stdin of the program or command. It is similar to interactive-program < command-file, where command-file contains

```
interactive-program <<LimitString
command #1
command #2
...
LimitString
```

Here Documents

Example. broadcast: Sends message to everyone logged in

```
#!/bin/bash

wall <<zzz23EndOfMessagezzz23
E-mail your noontime orders for pizza to the system administrator.
     (Add an extra dollar for anchovy or mushroom topping.)
# Additional message text goes here.
# Note: 'wall' prints comment lines.
zzz23EndOfMessagezzz23

# Could have been done more efficiently by
#              wall <message-file
# However, embedding the message template in a script
#+ is a quick-and-dirty one-off solution.

exit
```

Loops and Branches

for loops

for arg in [list]

This is the basic looping construct.

for arg in [list]

do

commands....

done

While loops

while

This construct tests for a condition at the top of a loop, and keeps looping as long as that condition is true (returns a 0 exit status). In contrast to a for loop, a while loop finds use in situations where the number of loop repetitions is not known beforehand.

while [condition]

do

commands....

done

The bracket construct in a while loop is nothing more than our old friend, the test brackets.

Loops and Branches

Until loops

Until

This construct tests for a condition at the top of a loop, and keeps looping as long as that condition is false (opposite of while loop).

–

until [condition-is-true]

do

commands....

done

–

Note that an until loop tests for the terminating condition at the top of the loop, differing from a similar construct in some programming languages.

Testing and Branching

Controlling program flow in a code block

case (in) / esac

The case construct is the shell scripting analog to switch in C/C++. It permits branching to one of a number of code blocks, depending on condition tests. It serves as a kind of shorthand for multiple if/then/else statements and is an appropriate tool for creating menus.

```
case "$variable" in
"$condition1" ) command...
;;
"$condition2" ) command...
;;
esac
```

Test Constructs

- An if/then/esle construct tests whether the exit status of a list of commands is 0 (since 0 means "success" by UNIX convention), and if so, executes one or more commands.

- There exists a dedicated command called [(left bracket special character). It is a synonym for test, and a builtin for efficiency reasons. This command considers its arguments as comparison expressions or file tests and returns an exit status corresponding to the result of the comparison (0 for true, 1 for false).

Test Constructs

- Bash has the [[...]] extended test command, which performs comparisons in a manner more familiar to programmers from other languages. Note that [[is a keyword, not a command.

- Bash sees [[$a -lt $b]] as a single element, which returns an exit status.

- The ((...)) and let ... constructs return an exit status, according to whether the arithmetic expressions they evaluate expand to a non-zero value. These arithmetic-expansion constructs may therefore be used to perform arithmetic comparisons.

Test Constructs

if/then/else

This is the way to construct an if/then/else test:

```
if [ condition-true ] then
command 1
command 2 ...
else
command 3 command 4 ...
fi
```

- An if can test any command, not just conditions enclosed within brackets.

```
if cmp a b &> /dev/null # Suppress output.
then echo "Files a and b are identical."
else echo "Files a and b differ."
fi
```

Test Constructs

else if and elif

Elif

elif is a contraction for ‘else if’. The effect is to nest an inner if/then construct within an outer one.

```
if [ condition1 ]
then
   command1
   command2
   command3
elif [ condition2 ]
then
   command4
   command5
else
default-command
fi
```

Functions

Like "real" programming languages, Bash has functions. A function is a subroutine, a ***code block*** that implements a set of operations, a "black box" that performs a specified task. Wherever there is repetitive code, when a task repeats with only slight variations in procedure, then consider using a function.

```
function function_name {
        commands...
}
```

or

```
function_name () {
        commands...
}
```

This second form will cheer the hearts of C programmers (and is more portable). As in C, the function's opening bracket may optionally appear on the second line.

Functions

- The commands between the curly braces ({}) are called the body of the function. The curly braces must be separated from the body by spaces or newlines.
- Defining a function doesn’t execute it. To invoke a bash function, simply use the function name. Commands between the curly braces are executed whenever the function is called in the shell script.
- The function definition must be placed before any calls to the function.
- When using single line “compacted” functions, a semicolon ; must follow the last command in the function.
- Always try to keep your function names descriptive

Functions

calling a function:

A function must exist before it is called:

```
#!/bin/bash
hello_world () {
   echo 'hello, world'
}
hello_world
```

Let's analyze the code line by line:

- In line 2, we are defining the function by giving it a name. The curly brace { marks the start of the function's body.
- Line 3 is the function body. The function body can contain multiple commands, statements and variable declarations.
- Line 4, the closing curly bracket }, defines the end of the hello_world function.
- In line 5 we are executing the function. You can execute the function as many times as you need.

If you run the script, it will print hello, world.

Functions

You can pass parameters to a function:

- To pass any number of arguments to the bash function simply put them right after the function's name, separated by a space. It is a good practice to double-quote the arguments to avoid the misparsing of an argument with spaces in it.
- The passed parameters are $1, $2, $3 … $n, corresponding to the position of the parameter after the function's name.
- The $0 variable is reserved for the function's name.
- The $# variable holds the number of positional parameters/arguments passed to the function.
- The $* and $@ variables hold all positional parameters/arguments passed to the function.
  - When double-quoted, "$*" expands to a single string separated by space (the first character of IFS) - "$1 $2 $n".
  - When double-quoted, "$@" expands to separate strings - "$1" "$2" "$n".
  - When not double-quoted, $* and $@ are the same.

Here is an example:

```
#!/bin/bash
greeting () {
  echo "Hello $1"
}
greeting "Joe"
```

Functions

Return Values

Unlike functions in “real” programming languages, Bash functions don’t allow you to return a value when called. When a bash function completes, its return value is the status of the last statement executed in the function, 0 for success and non-zero decimal number in the 1 - 255 range for failure.

The return status can be specified by using the return keyword, and it is assigned to the variable $?. The return statement terminates the function. You can think of it as the function’s exit status .

```
#!/bin/bash
my_function () {
  echo "some result"
  return 55
}
my_function
echo $?
```

Arrays

Bash support one-dimensional arrays. Array elements may be initialized with the variable[xx] notation. Alternatively, a script may introduce the entire array by an explicit declare -a variable statement. To dereference (retrieve the contents of) an array element, use curly bracket notation, that is, ${element[xx]}.

Any variable may be used as an indexed array; the declare builtin will explicitly declare an array. There is no maximum limit on the size of an array, nor any requirement that members be indexed or assigned contiguously. Indexed arrays are referenced using integers (including arithmetic expressions) and are zero-based; associative arrays are referenced using arbitrary strings. Unless otherwise noted, indexed array indices must be non-negative integers.

An indexed array is created automatically if any variable is assigned to using the syntax name[subscript]=value. The subscript is treated as an arithmetic expression that must evaluate to a number. To explicitly declare an indexed array, use declare -a name. declare -a name[subscript] is also accepted; the subscript is ignored.

sed and awk

This is a very brief introduction to the ***sed*** and ***awk*** text processing utilities. We will deal with only a few basic commands here, but that will suffice for understanding simple sed and awk constructs within shell scripts.

sed: a non-interactive text file editor

awk: a field-oriented pattern processing language with a C-style syntax

For all their differences, the two utilities share a similar invocation syntax, use regular expressions, read input by default from stdin, and output to stdout. These are well-behaved UNIX tools, and they work together well. The output from one can be piped to the other, and their combined capabilities give shell scripts some of the power of Perl.

sed

sed is a non-interactive stream editor. It receives text input, whether from stdin or from a file, performs certain operations on specified lines of the input, one line at a time, then outputs the result to stdout or to a file. Within a shell script, sed is usually one of several tool components in a pipe.

sed determines which lines of its input that it will operate on from the address range passed to it. Specify this address range either by line number or by a pattern to match. For example, 3d signals sed to delete line 3 of the input, and /Windows/d tells sed that you want every line of the input containing a match to "Windows" deleted.

Of all the operations in the sed toolkit, we will focus primarily on the three most commonly used ones. These are printing (to stdout), deletion, and substitution.

sed

Operator	Name	Effect
[address-range]/p	print	Print [specified address range]
[address-range]/d	delete	Delete [specified address range]
s/pattern1/pattern2/	substitute	Substitute pattern2 for first instance of pattern1 in a line
[address-range]/s/pattern1/pattern2/	substitute	Substitute pattern2 for first instance of pattern1 in a line, over address-range
[address-range]/y/pattern1/pattern2/	transform	replace any character in pattern1 with the corresponding character in pattern2, over address-range (equivalent of tr)
[address] i pattern Filename	insert	Insert pattern at address indicated in file Filename. Usually used with -i in-place option.
g	global	Operate on every pattern match within each matched line of input

sed

```
sed -e '/^$/d' $filename
# The -e option causes the next string to be interpreted as an editing
instruction.
# (If passing only a single instruction to sed, the "-e" is optional.)
# The "strong" quotes ('') protect the RE characters in the instruction
#+ from reinterpretation as special characters by the body of the
script.
# (This reserves RE expansion of the instruction for sed.)
#
# Operates on the text contained in file $filename.
```

awk

awk is a full-featured text processing language with a syntax reminiscent of C. While it possesses an extensive set of operators and capabilities, we will cover only a few of these here - the ones most useful in shell scripts.

awk breaks each line of input passed to it into fields. By default, a field is a string of consecutive characters delimited by whitespace, though there are options for changing this. **awk** parses and operates on each separate field. This makes it ideal for handling structured text files -- especially tables -- data organized into consistent chunks, such as rows and columns.

awk

```
# $1 is field #1, $2 is field #2, etc.

echo one two | awk '{print $1}'
# one

echo one two | awk '{print $2}'
# two
# But what is field #0 ($0)?
echo one two | awk '{print $0}'
# one two
# All the fields!

awk '{print $3}' $filename
# Prints field #3 of file $filename to stdout.

awk '{print $1 $5 $6}' $filename
# Prints fields #1, #5, and #6 of file $filename.

awk '{print $0}' $filename
# Prints the entire file!
# Same effect as: cat $filename . . . or . . . sed '' $filename
```

Cron jobs

The software utility ***cron*** also known as ***cron job*** is a time-based job scheduler in Unix-like computer operating systems. Users that set up and maintain software environments use cron to schedule jobs (commands or shell scripts) to run periodically at fixed times, dates, or intervals. It typically automates system maintenance or administration—though its general-purpose nature makes it useful for things like downloading files from the Internet and downloading email at regular intervals. The origin of the name cron is from the Greek word for time, χρόνος (chronos).(Wikipedia)

The typical format of a cron job is:

```
# ┌──────────── min (0 - 59)
# │ ┌────────── hour (0 - 23)
# │ │ ┌──────── day of month (1 - 31)
# │ │ │ ┌────── month (1 - 12)
# │ │ │ │ ┌──── day of week (0 - 6) (0 to 6 are Sunday to
# │ │ │ │ │                  Saturday, or use names; 7 is also Sunday)
# │ │ │ │ │
# │ │ │ │ │
# * * * * *  command to execute
```

Cron jobs

To display the contents of the crontab file of the currently logged in user:

```
$ crontab -l
```

To edit the current user's cron jobs, do:

```
$ crontab -e
```

Let us see some examples.

1. To run a cron job at every minute, the format should be like below.

```
* * * * * <command-to-execute>
```

2. To run cron job at every 5th minute, add the following in your crontab file.

```
*/5 * * * * <command-to-execute>
```

Assignment

Write a script to do a backup of your $HOME directory.

The name of the backup file should be something like backup-<date>.tar.gz

The backup file is to be written to your $SCRATCH directory.