`find, grep, sed, & awk`

Increasing productivity with command-line tools.

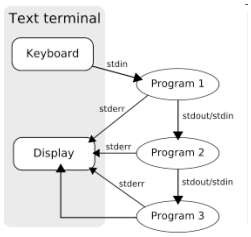
Review before bash scripting…
… y'all → yet another lousy language!
(After commands, regex, awk, (s)ed … before bash!
And that's only one shell
Then Windows : DOS, Powershell

Brief review

All use regex to specify string patterns

- Find – to find files by names
  - or other attributes such as size, age etc.
- Grep – to find strings within files
  - But can pipe output ls –R and use grep in place of find
  - How do you think find is done anyway…?
- Sed – to edit strings within files
  - Cryptic + poweful = danger! Always test first.
- Awk – to split and process fields within files.
  - can act as a basic accounting/monitoring language
  - can even edit like sed.

Unix philosophy



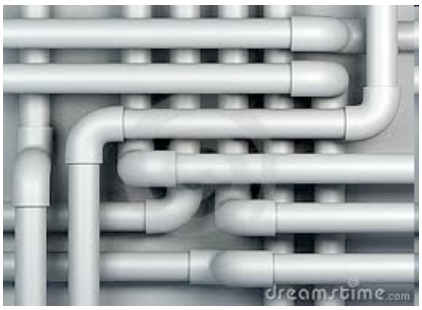
"This is the Unix philosophy: Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface."

--Doug McIlroy, inventor of Unix pipes

Why learn command-line utils?

- Simple – "do one thing"
- Flexible – built for re-use
- Fast – no graphics, no overhead
- Ubiquitous – available on every machine
- Permanent – 40 years so far …

Part 0 – pipes and `xargs`



Some simple programs

List files in current working directory:

```
$ ls
foo bar bazoo
```

Count lines in file `foo`:

```
$ wc –l foo
42 foo
```

Putting programs together

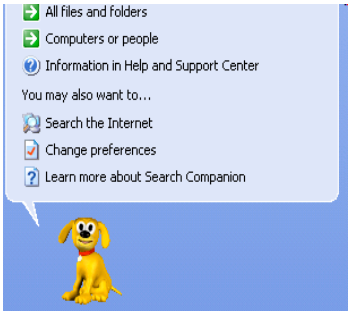
```
$ ls | wc –l
3
```

- l for line count, not 1 for one

```
$ ls | xargs wc –l
42 foo
31 bar
12 bazoo
85 total
```

Xargs - very handy, very common
- extracts arguments from stdin
- Runs the following command on these arguments one by one.

Part 1: `find`



find : search for a file in a directory tree to a specified level

Linux - the GNU version
find [-H] [-L] [-P] [path...] [expression]

This searches the directory tree rooted at each given file name by evaluating the given expression from left to right, according to the rules of precedence (see section OPERATORS), until the outcome is known (the left hand side is false for and operations, true for or), at which point find moves on to the next file name.

Mac Unix BSD
find [-H | -L | -P] [-EXdsx] [-f path] path ... [expression]
find [-H | -L | -P] [-EXdsx] -f path [path ...] [expression]

The find utility recursively descends the directory tree, for each path listed, evaluating an expression (composed of the ``primaries'' and ``operands'' listed below) in terms of each file in the tree.

GNU version

All of these are subject to the depth search restrictions from –maxdepth, -mindepth.

-P
Never follow symbolic links; just take the information from the properties of the symbolic link itself. This is the default behaviour in the absence of any options.

-L
Follow symbolic links.

-H
Do not follow symbolic links, except when the file is a link specified on the command line. If the link is broken, information about the link, itself is used as a fallback.

If more than one of -H, -L and -P is specified, each overrides the others; the last one appearing on the command line takes effect.

Basic find examples

```
$ find . –name Account.java
$ find /etc –name '*.conf'
$ find . –name '*.xml'
$ find . -not -name '*.java' -maxdepth 4
$ find . \(-name '*jsp' –o –name '*xml'\)
```

- `-iname` case-insensitive
- `!` == `-not`
- Quotes keep shell from expanding wildcards.

Find and do stuff

```
$ find . –name '*.java' | xargs wc –l | sort
```

Other options:

```
$ find . –name '*.java' -exec wc –l {} \; |
  sort
$ find . –name '*.java' -exec wc –l {} + |
  sort
```

Use your imagination. `mv, rm, cp, chmod . . .`

-exec – execute a cmd/script

Two basic purposes

1. runs a command within the current process
   1. Should be faster
   2. Inherits all variables, not just those specifically exported
   3. Frequently used in scripts
2. Redirects file descriptors within scripts

xargs – eXtract arguments

The xargs utility reads space, tab, newline and end-of-file delimited strings from the standard input and executes utility with the strings as arguments.

Any arguments specified on the command line are given to utility upon each invocation, followed by some number of the arguments read from the standard input of xargs. The utility is repeatedly executed until standard input is exhausted.

GNU version of xargs

xargs [-0prtx] [-E eof-str] [-e[eof-str]] [--eof[=eof-str]] [--null] [-d delimiter] [--delimiter delimiter] [-I replace-str] [-i[replace-str]] [--replace[=replace-str]] [-l[max-lines]] [-L max-lines] [--max-lines[=max-lines]] [-n max-args] [--max-args=max-args] [-s max-chars] [--max-chars=max-chars] [-P max-procs] [--max-procs=max-procs] [--interactive] [--verbose] [--exit] [--no-run-if-empty] [--arg-file=file] [--show-limits] [--version] [--help] [**command** [initial-arguments]]

xargs reads items from the standard input,

- delimited by
  - blanks (which can be protected with double or single quotes or a backslash)
  - or newlines,
- and executes the **command** (default is /bin/echo) one or more times with any initial-arguments followed by items read from standard input. Blank lines on the standard input are ignored.

Unix filenames containing blanks and newlines, are incorrectly processed by xargs.

Use the '-0' option, to overcome this and ensure the program which produces input for xargs also uses a null character as a separator, such as find with the '-print0' option.

BSD Unix version of xargs

xargs [-0opt] [-E eofstr] [-I replstr [-R replacements]] [-J replstr] [-L number] [-n number [-x]] [-P maxprocs] [-s size] [utility [argument ...]]

The xargs utility

- reads space, tab, newline and end-of-file delimited strings from the standard input
- and executes utility with the strings as arguments.

Arguments specified on the command line are given to utility upon each invocation, followed by some number of the arguments read from the standard input of xargs.

The utility is repeatedly executed until standard input is exhausted.

Spaces, tabs and newlines may be embedded in arguments using single (``' '') or double (``"'') quotes or backslashes (``\'').

Single quotes escape all non-single quote characters, excluding newlines.

Double quotes escape all non-double quote characters, excluding newlines.

Any single character, including newlines, may be escaped by a backslash.

The options are as follows:

-0 expect NUL (``\0'') characters as separators, instead of spaces and newlines. This is expected to be used in concert with the -print0 function in find(1).

`-exec` or `| xargs`?

- `-exec` has crazy syntax… {} for assumed args
- `| xargs` fits Unix philosophy.
- `\;` is slow, executes command once for each line.
- `\;` not sensible, sorts 'alphabetically.'
- | xargs may fail with filenames containing whitespace, quotes or slashes.

Find by type

Files:

```
$ find . –type f
```

Directories:

```
$ find . –type d
```

Links:

```
$ find . –type l
```

By modification time

Changed within day:

```
$ find . –mtime -1
```

Changed within minute:

```
$ find . –mmin -15
```

Variants `–ctime`, `-cmin`, `-atime`, `-amin` aren't especially useful.

By modification time, II

Compare to file

```
$ find . –newer foo.txt
```

```
$ find . ! –newer foo.txt
```

By modification time, III

Compare to date

```
$ find . -type f -newermt '2010-01-
  01'
```

Between dates!

```
$ find . -type f -newermt '2010-01-
  01' \
> ! -newermt '2010-06-01'
```

Find by permissions

```
$ find . –perm 644
```

```
$ find . –perm –u=w
$ find . –perm –ug=w
$ find . –perm –o=x
```

Find by size

Less than 1 kB:

```
$ find . –size -1k
```

More than 100MB:

```
$ find . –size +100M
```

`find` summary:

- Can search by name, path, depth, permissions, type, size, modification time, and more.
- Once you find what you want, pipe it to `xargs` if you want to do something with it

Part 2: grep

global / **r**egular **e**xpression / **p**rint

From ed command g/re/p

For finding text inside files.

Basic usage:

```
$ grep <string> <file or directory>
$ grep 'new FooDao' Bar.java
$ grep Account *.xml
$ grep –r 'Dao[Impl|Mock]' src
```

- Recursive flag is typical
- Quote string if spaces or regex.
- Don't quote shell filename with * wildcards!

  (* will be interpreted as regex multiplier not shell wildcard)

Common grep options

Case-insensitive search:

```
$ grep –i foo bar.txt
```

Only find word matches:

```
$ grep –rw foo src
```

Display line number:

```
$ grep –nr 'new Foo()' src
```

Filtering results

In**v**erted search:

```
$ grep –v foo bar.txt
```

Prints lines not containing `foo.`

Typical use:

```
$ grep –r User src | grep –v svn
```

Using `find … | xargs grep …` is faster.

More grep options

Search for multiple terms:

```
$ grep -e foo –e bar baz.txt
```

Find surrounding lines:

```
$ grep –r –C 2 foo src
```

Similarly –A or –B will print lines before and after the line containing match.

Example

Find tests that use the `AccountDao` interface.

Possible solution (arrive at incrementally):

```
$ grep –rwn –C 3 AccountDao src/test
> | grep –v svn
```

grep summary:

- `-r` **r**ecursive search
- `-i` case **i**nsensitive
- `-w` whole **w**ord
- `-n` line **n**umber
- `-e` multipl**e** searches
- `-A` **A**fter
- `-B` **B**efore
- `-C` **C**entered

Part 3: sed

stream **ed**itor

For modifying files and streams of text.

sed command #1: s

```
$ echo 'foo' | sed 's/foo/bar/'
bar

$ echo 'foo foo' | sed
  's/foo/bar/'
bar foo
```

`'s/foo/bar/g'` – global (within line)

Typical uses

```
$ sed 's/foo/bar/g' old
<output on screen to view b4 filing>

$ sed 's/foo/bar/g' old > new

$ sed –i 's/foo/bar/g' file
  -i extension
edits file in place, saving backup with extension, if
  given.

$ <stuff> | xargs sed –i 's/foo/bar/g'
```

Real life example I

Each time I test a batch job,
a flag file gets it's only line set to YES,
and the job can't be tested again
until it is reverted to NO.

```
$ sed –i 's/YES/NO/' flagfile
```

- Can change file again with up-arrow.
- No context switch.

Real life example II

A bunch of test cases say:

`Assert.assertStuff` which could be `assertStuff`, since using JUnit 3.

```
$ find src/test/ -name '*Test.java' \
> | xargs sed –i
  's/Assert.assert/assert/'
```

Backslash at end of line '\' merely signifies line continuation in scripts.

Real life example III

Windows CR-LF is mucking things up.

```
$ sed 's/.$//' winfile > unixfile
```

Replaces \r\n with (always inserted) \n

```
$ sed 's/$/\r/' unixfile > winfile
```

Replaces \n with \r\n.

Capturing groups

```
$ echo 'john doe' | sed 's/\b\(\w\)/\U\1/g'
John Doe

$ echo 'Dog Cat Pig' | sed 's/\b\(\w\)/(\1)/g'
(D)og (C)at (P)ig
```

Explanation :	' … '	== issue commands within single quote
	s/…/(\1)/g	== put brackets around first previous substring
	\(…\)	== substring specification,
	\b , \w	== beginning of a word, and word respectively
	\U	== switch to Uppercase

Exercise: formatting phone #.

Convert all strings of 10 digits to (###) ###-####.

Conceptually, we want:

```
's/(\d{3})(\d{3})(\d{4})/(\1) \2-\3/g'
```

- Must escape parenthesis and braces with \… { , } etc.
- Brackets are not escaped. ( ) are OK
- But \d (for decimal digit) is not supported in sed regex, so rewrite as:

```
's/\([0-9]\{3\}\)\([0-9]\{3\}\)\([0-9]\{4\}\)/(\1) \
  2-\3/g'
```

NB d for delete is supported, but not \d for digit!

Exercise: trim whitespace

Trim leading whitespace:

```
$ sed -i 's/^[ \t]*//' t.txt
```

Trim trailing whitespace:

```
$ sed –i 's/[ \t]*$//' t.txt
```

Trim leading and trailing whitespace:

```
$ sed -i 's/^[ \t]*//;s/[ \t]*$//'
  t.txt
```

Add comment line to file with s:

```
'1s/^/\/\/ Copyright FooCorp\n/'
```

- Prepends // Copyright FooCorp\n
- 1 restricts to first line, similar to vi search.
- ^ matches start of line.
- With find & sed insert in all .java files.

Shebang!

In my .bashrc:

```
function shebang {
  sed –i '1s/^/#!\/usr\/bin\/env python\n\n'
  $1
  chmod +x $1
}
```

Prepends #!/usr/bin/env python

and makes file executable

sed command #2: d

Delete lines containing foo:

```
$ sed –i '/foo/ d' file
```

Delete lines starting with #:

```
$ sed –i '/^#/ d' file
```

Delete first two lines:

```
$ sed –i '1,2 d' file
```

More delete examples:

Delete blank lines:

```
$ sed '/^$/ d' file
```

Delete up to first blank line (email header):

```
$ sed '1,/^$/ d' file
```

Note that we can combine range with regex.

Real life example II, ctd

A bunch of test classes have the following unnecessary line:

```
import junit.framework.Assert;
```

```
$find src/test/ -name *.java | xargs \
> sed -i '/import
  junit.framework.Assert;/d'
```

Backslash at end of line '\' merely signifies line continuation in scripts.

sed summary

- With only s and d you should probably find a use for sed once a week.
- Combine with find for better results.
- sed gets better as your regex improves.
- Syntax often matches vi.

Part 4: awk

- **A**ho, **W**einberger, **K**ernighan
- pronounced *auk.*
- Useful for text-munging.

Simple awk programs

```
$ echo 'Jones 123' | awk '{print $0}'
Jones 123

$ echo 'Jones 123' | awk '{print $1}'
Jones

$ echo 'Jones 123' | awk '{print $2}'
123
```

Example `server.log` file:

```
fcrawler.looksmart.com [26/Apr/2000:00:00:12] "GET
/contacts.html HTTP/1.0" 200 4595 "-"
fcrawler.looksmart.com [26/Apr/2000:00:17:19] "GET
/news/news.html HTTP/1.0" 200 16716 "-"
ppp931.on.bellglobal.com [26/Apr/2000:00:16:12] "GET
/download/windows/asctab31.zip HTTP/1.0" 200 1540096
"http://www.htmlgoodies.com/downloads/freeware/webdevelopment/15.html
   "
123.123.123.123 [26/Apr/2000:00:23:48] "GET /pics/wpaper.gif HTTP/1.0"
200 6248 "http://www.jafsoft.com/asctortf/"
123.123.123.123 [26/Apr/2000:00:23:47] "GET /asctortf/ HTTP/1.0" 200
8130
"http://search.netscape.com/Computers/Data_Formats/Document/Text/RTF"
123.123.123.123 [26/Apr/2000:00:23:48] "GET /pics/5star2000.gif
HTTP/1.0" 200 4005 "http://www.jafsoft.com/asctortf/"
123.123.123.123 [26/Apr/2000:00:23:50] "GET /pics/5star.gif HTTP/1.0"
200 1031 "http://www.jafsoft.com/asctortf/"
123.123.123.123 [26/Apr/2000:00:23:51] "GET /pics/a2hlogo.jpg
   HTTP/1.0"
200 4282 "http://www.jafsoft.com/asctortf/"
<snip>
```

Built-in variables: NF, NR

- NR – Number of Record
- NF – Number of Fields
- With $, gives field, otherwise number

```
$ awk '{print NR, $(NF-2)}'
  server.log
1 200
2 200
```

Structure of an awk program

condition { actions }

```
$ awk 'END { print NR }' server.log
9

$ awk '$1 ~ /^[0-9]+.*/ { print $1,$7}' \
> server.log
123.123.123.123 6248
123.123.123.123 8130
```

Changing delimiter

```
$ awk 'BEGIN {FS = ":"} ; {print
  $2}'
```

- FS – Field Seperator
- BEGIN and END are special patterns

Or from the command line:

```
$ awk –F: '{ print $2 }'
```

Get date out of `server.log`

```
$ awk '{ print $2 }' server.log
[26/Apr/2000:00:00:12]

$ awk '{ print $2 }' server.log \
> | awk –F: '{print $1}
[26/Apr/2000

$ awk '{ print $2 }' server.log \
> | awk –F: '{print $1} | sed 's/\[//'
26/Apr/2000
```

Maintaining state in awk

Find total bytes transferred from `server.log`

```
$ awk '{ b += $(NF-1) } END { print b }'
  server.log
1585139
```

Find total bytes transferred to `fcrawler`

```
$ awk '$1 ~ /^fcraw.*/ { b += $(NF-1) } END { print
  b }'\
> server.log
21311
```

One more example

Want to eliminate commented out code in large codebase.

Makea one-liner to identify classes that > 50% comments,

A comment line has “//” as the first non-whitespace chars.

```
$ awk '$1 == "//" { a+=1 } END { if (a*2 >
  NR) {print FILENAME, NR,  a}}'
```

Example, ctd.

To execute on all Java classes:

```
$ find src -name '*.java' -exec awk '$1 ==
  "//" { a+=1 } END { if (a * 2 > NR)
  {print FILENAME, NR, a}}' {} \;
```

- Here –exec with \; is the right choice, as the awk program is executed for each file individually.
- It should be possible to use xargs and FNR, but I'm trying to keep the awk simple.

awk summary

- NF – Number of Field
- NR – Number of Records
- FILENAME – filename
- BEGIN, END – special events
- FS – Field Seperator (or –F).
- awk 'condition { actions }'