

4th Edition



UNIX

IN A NUTSHELL

*A Desktop Quick Reference
Covers GNU/Linux, Mac OS X, and Solaris*

O'REILLY®

Arnold Robbins

Unix Tools 2

Jeff Freymueller

UNIX Power Tools

2nd Edition



sed & awk

O'REILLY®

Dale Dougherty & Arnold Robbins

*Demystifying the
Geekier Side of Mac OS X*

4th Edition
No Cover Image



Mac OS X

for Unix Geeks

O'REILLY®

Brian Johnson,
Ernest E. Robson
& Rich Rosen

*A Quick Pocket Reference for a Utility
Every Unix User Needs*

grep

Pocket Reference



O'REILLY®

John Bambenek &
Agnieszka Klus



Outline

- Remote access: ssh
- Variables as a collection of words
- Making basic output and input
- A sampler of unix tools: text processing, slicing and dicing
 - grep (“get regular expression”)
 - awk (a text processing language)
 - sed (“stream editor”)
 - tr (translator)
- A smorgasbord of examples

Access a “remote” machine: ssh

- You can open up a shell to work on any machine in the world with ssh (secure shell)
 - You can even forward the graphics that would be produced to your screen with X11 forwarding
- Why would you do that?
 - Why not? Is only one computer enough for you?
 - Process data from home, on the road, the other building, the next room, etc
 - Access stuff/programs that only reside on the big server

Using ssh

- `ssh [options] user@host [command]`
 - [options] use `-X` or `-Y` to forward graphics via X11 if your machine can display them
 - Unix, linux, MacOSX: yes; Windows, ??
 - With no command specified, you get an interactive shell. Otherwise, the command is run
- Example:
 - `ssh -X jeff@denali.gps.alaska.edu`
- In the old days, there were some other options, like rsh and telnet, but security is needed these days...

Variables as a collection of words

- The shell treats variables as a collection of words
 - `set files = (file1 file2 file3)`
 - This sets the variable `files` to have 3 “words”
- If you want the whole variable, access it with `$files`
- If you want just one word, use `$files[2]`
- The shell doesn't count characters, only words

Basic output: echo and cat

- *echo string*
 - Writes a line to standard output containing the text *string*. This can be one or more words, and can include references to variables.
 - `echo "Opening files"`
 - `echo "working on week $week"`
 - `echo -n "no carriage return at the end of this"`
- *cat file*
 - Sends the contents of a file to standard output

Input, Output, Pipes

- Output to file, vs. append to file.
 - `> filename` creates or overwrites the file *filename*
 - `>> filename` appends output to file *filename*
- Take input from file, or from “inline input”
 - `< filename` take all input from file *filename*
 - `<<STRING` take input from the current file, using all lines until you get to the label *STRING* (see next slide for example)

Example of “inline input”

```
gpsdisp << END
Okmok2002-2010.disp
Okmok2002-2010.vec
Y
Okmok2002-2010.gmtvec
Y
Okmok2002-2010.newdisp
Okmok2002_mod.stacov
5
5
Okmok2010.stacov
5
5
76
n
END
```

- Many programs, especially older ones, interactively prompt you to enter input
- You can automate (or self-document) this by using <<
- Standard input is set to the contents of this file between << END and END

grep (“get regular expression”)

- grep is a tool that allows you to extract lines from a file that contain some search pattern.
 - Plain text files only, of course!
- The basic usage is: `grep string filename`
 - All lines of the file *filename* that contain the string *string* will be written to standard output.
- You can use multiple filenames, and there are several options you can use with various flags. One handy option is `-v` (invert): `grep -v string filename`
 - All lines of the file *filename* that **do not contain** the string *string* will be written to standard output.

grep

- In its simplest form, `grep` finds every line in a plain text file that contains a certain string.
- Its “search string” can be more than a simple string: regular expressions
- You can include special characters and wild cards
 - `^` start of line
 - `$` end of line
 - `.` match exactly one character

Examples:

- Find every line with string “AK”
 - `grep AK city+state_list`
- A fancier example:
 - `wget -O - http://www.cygwin.com/ | grep “Windows 98”`
- Using some wildcards:
 - `grep "^AB.." ~/sitevecs`
- Search for two strings:
 - `grep AK city+state.list | grep River`

Variants of grep: egrep, etc

- There are a number of variants of grep, which behave a little differently.
- A useful one is egrep, which allows you to use an “OR” construct to find matches to any of two or more strings:
 - `egrep “(string1|string2)” file`
- For compressed files, use `zgrep`, `zipgrep`, `bzgrep`
- See `man fgrep`
- Example:
- List all PBO sites in Alaska
 - `egrep “(^AB..|^AC..|^AV..)” ~/sitevecs | more`
- Output is:
 - AB01 ATKA ISLAND
 - AB02 NIKOLSKI
 - AB04 SAVOONGA PBO

awk

- awk is an incredibly powerful text processing language (think food processor)
- If you want to get the third word of text from a line, or want to get the difference between the numbers in the 5th and 6th columns, divided by the square root of the sum of squares of the numbers in the first 3 columns, awk is your tool.
- Named for its authors: Aho, Weinberger, Kernighan
- Use it with pipes to get the full effect!

"AWK is a language for processing files of text. A file is treated as a sequence of records, and by default each line is a record. Each line is broken up into a sequence of fields, so we can think of the first word in a line as the first field, the second word as the second field, and so on. An AWK program is of a sequence of pattern-action statements. AWK reads the input a line at a time. A line is scanned for each pattern in the program, and for each pattern that matches, the associated action is executed." - Alfred V. Aho

awk Principles

- Every line is broken up into fields. By default, whitespace separates fields
- awk reads each line in its input file (or standard input) and does something based on its command program (a string, or a series of commands in a file)
awk “command string” file(s)
- The command string is of the form “pattern {action}” and you can have many pattern-action pairs
- Example: `awk 'NF > 3 {print $4}' myfile.txt`
 - What it does: If there are more than 3 fields on a line, print out the 4th field

Some awk examples

- Print the first field of every line
 - `awk '{print $1}' myfile.txt`
- Print every line with two fields
 - `awk 'NF == 2 {print $0}' myfile.txt`
- Get the day of the week
 - `date | awk '{print $1}'`
- Do some math
 - `awk '{print $5 " " sqrt($1*$1 + $2*$2 + $3*$3) $5}'`
- Print the first 4 characters of the second field
 - `awk '{print substr($2,1,4)}'`
- Use the character ":" to separate fields
 - `awk -F: '{print $1 " has the name " $5}' /etc/passwd | more`

Another awk example

- I have a file *allsites.gmt* with lines like this:
 - 159.451212001 54.035486000 KRMS
 - -152.148672605 61.260421190 SKID
- My awk command to extract a spatial subset
 - `awk '$1 > -179 && $1 < -130 && $2 > 55 {print $0}'`
allsites.gmt
- This outputs every line within the given lat-long box

awk -F is a wonderful thing

- Extract fields from a .csv file
 - If you save an excel file in .csv format, you get a text file with the cells of the spreadsheet separated by commas
 - `awk -F, '{print $1 " has " $4 " dollars left."}'`
- Parse a pathname/filename
 - The directory tree in a pathname/filename is separated by "/" characters
 - `awk -F/ '{print $NF}'`
 - `Awk -F/ 'NF > 1 {print "pathname contains a /"}'`

sed (the stream editor)

- sed is a program that lets you do a find-and-replace process on text files via the command line.
 - Simplest form: `sed 's/string1/string2/g' file1 > file2`
 - What it does: replace every occurrence of *string1* with *string2* in the file *file1*, sending output to *file2*.
 - The 's' in the command string is for search/replace
 - The 'g' at the end means to do it on every match. Without the g it will change only the first matching string on each line.
 - As usual, it can operate on a file or on standard input
- And you can do more as well, beyond the scope of a first lesson

Making an input file with sed

- Many scientific programs have specific input files, which might contain the names of files, values of parameters, etc

```
set master_dir = /home/jeff/taboo
foreach viscosity ( 1 3 10 30 100 300)
  foreach thickness ( 25 30 35 40 45 50 55 60 )
    cd ${master_dir}
    mkdir ${thickness}_${viscosity}
    cat master_input | sed s/VISCOSITY/${viscosity}/ \
      | sed s/THICKNESS/${thickness}/ \
      > Alaska05_${thickness}_${viscosity}/taboo.input
    cd Alaska05_${thickness}_${viscosity}
    ./taboo < taboo.input > taboo.output
  end
end
end
```

tr

- tr is a character-based translator, while sed is a word-based translator. A common use of tr is to change uppercase to lowercase text
- Examples
 - tr '[a-z]' '[A-Z]' < input > output
 - tr '[A-Z]' '[a-z]' < input > output
 - tr ' ' '_' < input > output
 - This last example changes every space to an underscore

Example scripts using these tools

- Several practical examples follow. Most of these will combine some elements of control structures, use of variables, and the use of some unix tools through pipes.
- Some of these may be useful for you to copy.
 - If you do, be careful with quotes and such. Powerpoint uses fancy quotes that look like typeset text, but the shell only knows plain quotes.

Syntax: MATLAB vs. tcsh

MATLAB	tcsh
<code>i = 20;</code>	<code>set i = 20</code>
<code>i = i + 1</code>	<code>@ i = \$i + 1</code>
<code>if (a == b) i = i + 1; disp(num2str(i)); end</code>	<code>if (\$a ==\$b) then @ i = \$i + 1 echo \$i endif</code>
	<code>if (\$a ==\$b) echo "a and b are equal"</code>
<code>for i = 1:10 disp(['The number is ' num2str(i)]); end</code>	<code>foreach i (1 2 3 4 5 6 7 8 9 10) echo "The number is \$i" end</code>

The shell also has a “while” control structure, with the same syntax as MATLAB

Processing files from some stations

- Maybe you have a set of files, and you want to “process” the file from certain stations.

```
set process_list = /home/jeff/stations_to_process
foreach file ( *.dat )
    set station = `echo $file | awk '{print substr($0,1,4)}'`
    if ( `grep $station $process_list | wc -l` > 0 ) then
        echo "Processing file $file from station $station"
        process_file $file
    endif
end
end
```

- For this to work, you need to adopt a systematic naming convention for filenames.
 - In this example, the first 4 characters of the filename must be the station name

Same example, elaborated

- You have already processed some files, so you only want to process the files you haven't already done.
 - Suppose that the `process_file` script creates some output files, so you can test for the existence of these files.

```
set process_list = /home/jeff/stations_to_process
foreach file ( *.dat )
  set station = `echo $file | awk '{print substr($0,1,4)}'`
  if ( `grep $station $process_list | wc -l` > 0 ) then
    set base = `basename $file .dat`
    if ( ! -e ${base}.jpg ) then
      echo "Processing file $file from station $station"
      process_file $file
    endif
  endif
end
```

Produce an organized list of files

- Suppose you have a set of files named by date and by station. Example: 10nov08FAIR.dat. Make a list of files for each station.
 - Suppose you wanted a formatted list of every station for each date?

```
set filespec = '*.dat'
set stations = ` /bin/ls $filespec | awk '{print substr($0,8,4)}' \
    | sort -u`
foreach station ( $stations )
    echo "======"
    echo -n "Number of files for station $station : "
    /bin/ls ??????${station}.dat | wc -l
    /bin/ls ??????${station}.dat | \
        awk '{n += 1} {printf("%3.3i: %s\n", n, $0)}'
    echo
end
```


Produce an organized list of files

- The output will look something like this:

```
=====
Number of files for station OK23 : 3
001: 05jul02OK23.dat
002: 07jul22OK23.dat
003: 10jul28OK23.dat

=====
Number of files for station FAIR : 2
001: 99feb12FAIR.dat
002: 03sep30FAIR.dat
003: 10aug19FAIR.dat
```