

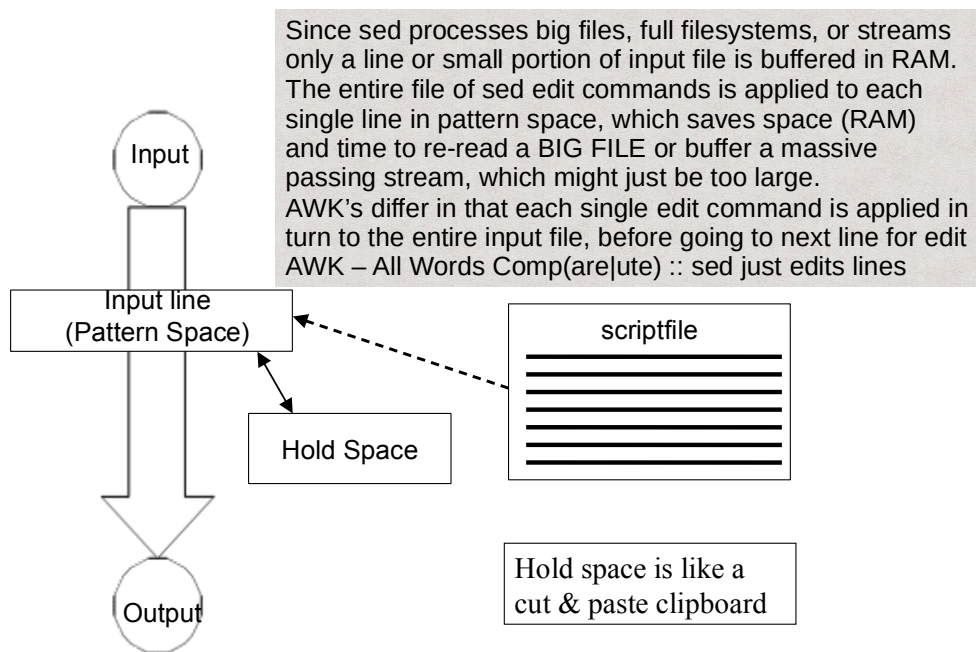
Sed – super ed!?



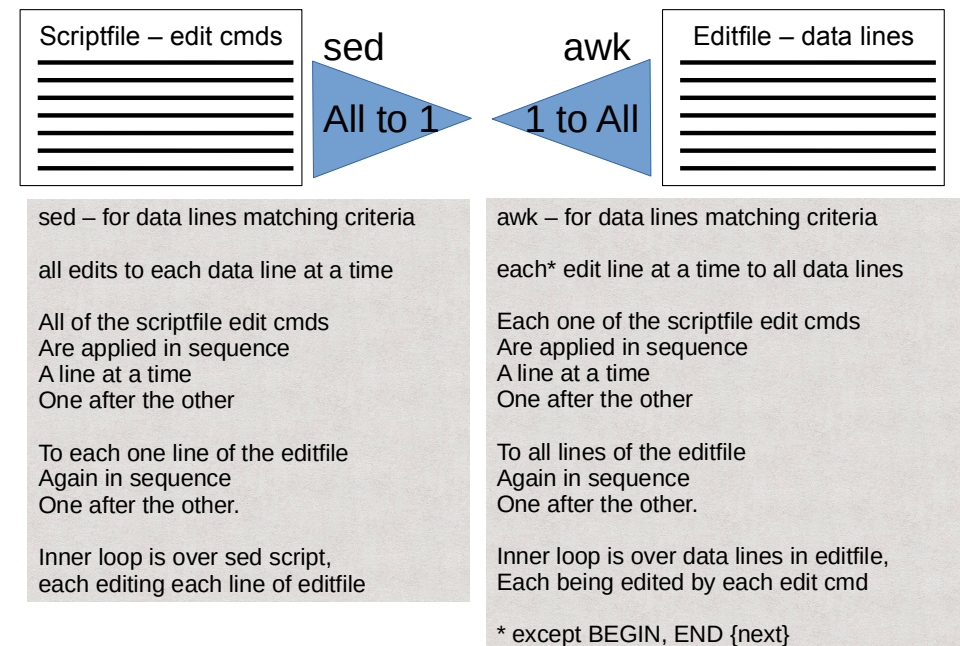
Sed - stream editor

- Sed:
 - a Stream version of ed, which preceded & influenced editors
 - Brief review provided in overview of editors soon
- But test sed commands before trying on a real file, although safety factor
 - sed writes to standard output, makes no change to original file,
 - changes not saved unless -i flag (in place) or with file redirection >
- Develop interactively with test text on a line:-
 - echo 'test text' | sed
 - Or using ed on a text file ... if you must!
 - Or use online sed regex tester, but may differ from system
- then with sed on test file.
- It's still good custom and practice to keep a backup copy
 - to avoid accidental corruption of file with any other tool,
 - In cases where you thought you had it right, until you looked again!

AWKs vs sed



sed – awk



Basic operating comparison : awk vs sed

- Awk takes each script line in turn, one by one
 - And applies it to the entire edit file all data lines, in sequence
 - Expressed as nested loops:
 - For each awk (edit command) line..
 - For each data (for edit) line (called edit file)
 - Fact? Act
- *sed* takes each data line in turn, one by one,
 - And applies all script lines, to it, in sequence
 - Expressed as nested loops
 - For each data line
 - For each sed line
 - Fact? Act!

Awk a data driven programming language processing data in input lines, one at a time

Sed is merely a programmable editor, editing line contents, one line at a time.

NB in both cases, but in different ways, preceding code lines may affect effects of succeeding ones, by changing data.

Sed - stream editor

- Most typical editors process full files in RAM :
 - NOT suitable for BIG files or data streams
- *ed* & *sed* use buffers to process part of file as a stream.
 - Ideal for modifying large files and streams of text
 - Normal editors need all the file in RAM, possible limits
- Any data stream available as text
 - Comms
 - Files,
 - entire filesystems as lists of filenames as a stream
 - e.g. for changing all filenames
 - e.g. changing file extensions – irrelevant in 'nix
 - Formatting commands for a print formatting program

sed, The Stream Editor

- *sed* is descended from an extremely basic line-editor *ed*
 - Both operate on files one line at a time
 - Both use a similar command format
 - *[address] operation [argument]*
 - *ed* can use command scripts; files containing *ed* editing commands
 - *ed filename <script_file*
- *sed* is a special purpose editor that
 - will only take commands from a script or the command line,
 - it cannot be used interactively, to go back and change as you go
- All editing
 - command input to *sed* comes from
 - Either standard input (indicated with an -e flag)
 - Or a file containing edit commands (indicate by an -f flag)
 - output goes to standard output,
 - which can be redirected,
 - **And must be redirected for changes to be saved.**

Sed - main points

- is a line editor, changing a line at a time
- Commands are given in a file or in command line
- All editing commands, in the command line or entire script,
 - if applicable, are applied to each line of edit file in turn, before processing the next line
 - Beware unintended consequences of command sequence
- Cannot be used interactively - i.e. midstream
- Does not modify original file, (unless -i (in place) flag is used)
 - but writes to output file when directed;
 - convenient for inserting formatting commands for a print pipeline, without changing the original text.
- And for those who are aware, editing commands resemble those in *ex*, *ed*, or last line (colon) mode in *vi*

It's sad sed doesn't change files... unless...

- Therefore,
 - changes are not made to the edit file itself,
 - instead the input file, along with any changes, is written to standard output
- This is an important difference between *ed* and *sed* - *ed* changes the edit file, *sed* does not
- If you want to save the changes from *sed*, they must be redirected from standard output to a file
 - `sed -f scriptfile editfile >outputfile`
 - Can be used to selectively generate / split data into multiple output files depending on data.
 - See slides 42-44 for example

sed Syntax

`sed [-n] [-e] ['command'] [file...]`

`sed [-n] [-f scriptfile] [file...]`

- `-e command` –
the next argument is an editing command not a filename, useful if multiple commands are specified
- `-f scriptfile`
next argument is filename with editing commands
- `-n` only print lines specified with the 'p' command or 'p' flag of the substitute ('s') command
- If first line of scriptfile is "#n", then same as `-n` above

Stream vs Line Addressing

- A very important difference is the stream orientation aspect of *sed*'s impact on line addressing, unlike *ed* (which is interactive)
 - *ed* operates only on lines that are specifically addressed or the current line if no address is specified
 - If you enter the command "s/dog/cat/" it would change the first instance of "dog" on the current line to "cat"
- *sed* goes through the edit file a line at a time, so if no specific address is provided for a command, it operates on **all** lines, like AWK
- The same command in *sed* would change the first occurrence of "dog" on **every** line to "cat"

Sed scripts

- A *sed* script is nothing more than a file of edit commands
 - Each command consists of an address and an action, where the address can be a pattern (regular expression)
 - As each line of the input file is read, *sed* reads the first command of the script and checks the address or pattern against the current input line of the file being edited
 - If there is a match, the command is executed
 - If there is no match, the command is ignored
 - *sed* then repeats this action for every command in the script file
 - All commands in script file, are checked for applicability to the current line of the edit file - not just the first match
 - Beware of unintended effects of command sequences!
e.g. to switch a&b in bac to get abc:
switching a before b works : bac s/a/b/ bbc; s/b/a/ abc
but switching b before a won't : bac s/b/a/ aac; s/a/b/ bac
- Chaos possible for longer sequences (scriptfiles) of edit commands

Lyin' line editors... ed vs. sed

- On reaching the end of the edit script,
 - *sed* outputs the current line
 - unless the *-n* option has been set, when it only prints those specified with *p*
- *sed* then reads the next line in the input file for edit and restarts from the beginning of the script file
- All commands in the script file are compared to, and potentially act on, all lines in the input data file

Note again the difference from *ed*

- If no address is given, *ed* operates only on the current line
- If no address is given *sed* operates on all lines

Four Basic Script Types

- Multiple edits to the same file
 - Change from one document formatter's codes to another
- Making changes across a set of files
 - Global changes (often to names or terms):-
 - e.g. code redesign & reassignments;
 - website and all publicity – e.g. business mergers, acquisitions, rebranding or product name changes
- Extracting the contents of a file
 - Flat-file database operations
- Making edits in a pipeline
 - when making changes prior to some other command that you don't want made permanently to the source file; e.g. formatting commands in a print pipeline

Three Basic Principles of sed

- All editing lines of a script
 - apply to all lines of the file being edited
 - unless line selection (numbers, patterns) blocks
- The original file is unchanged, unless redirected
 - the editing commands modify a copy of the original line and the copy is sent to standard output
- All editing commands in a script
 - are applied in order to each line of input file for editing,
 - unless the command is *d* (delete) or *c* (change)
 - in which case a new line from the edit file (file being edited) is read after the *d* or *c* command executes

sed Commands

- *sed* commands have the general form
 - *[address[, address]][/!]*command* [*arguments*]*
- *sed* copies each input line into a *pattern space*
 - If the address in the edit command matches the line in the *pattern space*, the edit command is applied to that line
 - If the command has no address, it is applied to each line as it enters *pattern space*
 - If a command changes the line in *pattern space*, subsequent commands operate on the modified line
 - E.g. try reversing the order of '**words two**' in a line..?
 - Identical to the 'bac' --- 'abc' example a few slides back!
- When all editing commands have been applied, the line in *pattern space* is written to standard output and a new line is read into *pattern space*

Addressing

- An address can be either a line number or a pattern, enclosed in slashes (/pattern/)
- A pattern is described using *regular expressions*
- Additionally a NEWLINE can be specified using the “\n” character pair
 - This is only really useful when two lines have been joined in pattern space with the N command* so that patterns crossing line boundaries can be searched
- If no pattern is specified, the command will be applied to all lines of the input file

* and unique variations : extract from slide 21 below

N Append the next line of input to the pattern space with an embedded NEWLINE. (The current line number changes.)

Many lines of edit commands... Many lines to edit...

- Most commands will accept two addresses
 - If only one address is given, the command operates only on that line
 - If two comma separated addresses are given, then the command operates on a range of lines between the first and second address, inclusively
- The ! operator can be used to negate an address, ie; *address!command* causes *command* to be applied to all lines that do not match *address*
- Braces { } apply multiple commands to an address

Many lines of edit commands, BUT no brace spaces...

- [/pattern[/,/pattern/]] - applies to 1 or range of lines
- Braces { } apply multiple commands to an address
- The opening brace must be the **last character on a line** and the closing brace must be on a line by itself
- **Make sure there are no spaces following the braces**

```
[/pattern[/,/pattern/]]{  
command1  
command2  
command3  
}
```

Many lines to edit : Address Examples

d	deletes the current line
6d	deletes line 6
/^\$/d	deletes all blank lines
1,10d	deletes lines 1 through 10
1,/^\$/d	deletes from line 1 through the first blank line
/^\$/,/\$/d	deletes from the first blank line through EOF
/^\$/,10d	deletes from the first blank line through line 10
/^Co*t/,/[0-9]\$/d	deletes from the first line that begin with Ct, Cot, Coot, etc. - - C(any no. of o's) through the first line that ends with a digit

The end of line \$/ pattern can be handy to fix \$PATH environment variable, if the last addition had an error.

... and many sed commands ...

- Although sed contains many editing commands, we need only consider a small subset, most of which are common throughout Unix editors.

◆ s - substitute	◆ p - print
◆ a - append	◆ ! - NOT
◆ i - insert	◆ r - read
◆ c - change	◆ w - write
◆ d - delete	◆ y - transform
◆ h,H - put pattern space into hold space	◆ q - quit
◆ g,G - Get hold space	

Hold space is like a clipboard for 'cut & paste'.

sed Command List

Commands to add text are followed with a \ after, to delimit added text.

a\ text	Append. Place text on output before reading next input line.
b label	Branch to the : command bearing the label. If label is empty, branch to the end of the script.
c\ text	Change:- i.e. Delete pattern space. Place text on the output. Start the next cycle – the line is gone. Further editing pointless! (i.e. apply edit scriptfile to all following lines in pattern space)
d	Delete the pattern space. Start the next cycle.
D	Delete the initial segment of the pattern space through the first NEWLINE. Start the next cycle.
g	Replace contents of pattern space with those of hold space.
G	Append contents of hold space to that of pattern space.

... and more sed commands ...

h	Replace contents of hold space with those of pattern space.
H	Append the contents of hold space to those of pattern space.
ì\text	Insert. Place text on standard output.
l	List pattern space on standard output in unambiguous form. Non-printable chars displayed in octal & long lines folded.
n	Copy the pattern space to standard output. Replace the pattern space with the next line of input.
N	Append next line of input to pattern space with embedded NEWLINE. (The current line number changes.)
p	Print. Copy the pattern space to standard output.
P	Copy the initial segment of the pattern space up through the first NEWLINE to standard output.
q	Quit. Branch to the end of the script. Don't start a new cycle.

... say it again, sed! ...

r rfile	Read the contents of rfile. Place them on standard output before reading the next input line.
s /regular expression/replacement/flags	Substitute the replacement string for instances of the regular expression in the pattern space. Flag is zero or more of: n n=1-512. Substitute the nth occurrence of the regular expression g Global. Substitute all non-overlapping instances of the regular expression rather than just the first one. p Print the pattern space if a replacement was made.
w wfile	Write. Append the pattern space to wfile if replacement made.
t label	Test. Branch to the : command bearing the label if any substitutions have been made since the most recent reading of the input line or execution of a t. If label is empty, branch to end of script.

... and again !?

w wfile	Write. Append the pattern space to wfile. The first occurrence of a w will caused wfile to be cleared. Subsequent invocations of w will append. Each time the sed command is used, wfile is overwritten.
x	Exchange the contents of the pattern and the hold space.
y/string1/string2/	Transform – similar to bash cmd ‘tr’. Replace all occurrences of the characters in string1 with the characters in string2. string1 and string2 must have the same number of characters.
!function	Don't apply the function (or group) if function is only to those lines not selected by the address(s).
: label	Just the target label for the b and t to branches.
=	Place the current line number on standard output as a line. {Execute the following commands through a matching } only when the pattern space is selected. An empty command is ignored.

trip-ups!

If an # appears as the first character on a line of script, then that line is treated as a comment

unless

it is the first line of the file

and the character after the # is an n.

Then the default output is suppressed (just like sed -n).

The rest of the line after the n is also ignored.

A script file must contain at least one non-comment line.

Substitute

- Syntax: *[address(es)]s/pattern/replacement/[flags]*
 - *pattern* - search pattern
 - *replacement* - replacement string for pattern
 - *flags* - optionally any of the following
 - n a number from 1 to 512
indicating which occurrence
of *pattern* should be replaced
 - g global, replace all occurrences
of *pattern* in pattern space
 - p print contents of pattern space
 - w *file* write the contents of pattern space to *file*

Replacement Patterns

Substitute can use several special characters in the *replacement* string

- & - replaced by the entire string
matched in the regular expression for pattern
- \n - replaced by the *n*th substring or subexpression
previously specified using “\(*“ and “ \)”*
- \ - used to escape
the ampersand (&) and the backslash (\)

Replacement Pattern Examples

As could be used to achieve insertions:	“ the UNIX system...”
s/.NI./wonderful &/	“ the wonderful UNIX system...”

& - means the entire string matched in the regexp /.NI./ -> UNIX,
/.NI./ matches UNIX above, so UNIX is replaced with /wonderful UNIX/

As can be used to reverse matches; e.g. reversing firstname, surname etc.

```
cat test1
```

first:second

one:two

```
sed 's/\(.*\):\(.*\)/\2:\1/' test1
```

second:first

two:one

Basic regex (BRE)

- need \ to esc () for normal interpretation as ()
- delimiters coloured only for easy reading

Not usually displayed as such in editors

But you could configure/amend

Note sed works on a file test1,

Switching strings around the :

Note :

- the colon ':' is detected and not escaped with '\' as any other character
- Can use '.'*', because the regex automata/state machine has a lookahead facility (to see ':') before definitively committing

Other Substitute Examples

s/cat/dog/

Substitute dog for the first occurrence of cat in *pattern space*

s/Sky/Sea/2

Substitutes **Sky** for the second occurrence of **Sea** in the *pattern space*

s/wood/plastic/p

Substitutes plastic for the first occurrence of wood
and outputs (prints) *pattern space*

s/Mr/Dr/g

Substitutes Dr for every occurrence of Mr in *pattern space*

Break!?

Append, Insert, and Change

- Syntax for these commands is a little strange because they **must** be specified on multiple lines
- append *[address]a*
 text
- insert *[address]i*
 text
- change *[address(es)]c*
 text
- Text is
 - Continued over multiple lines by escaping newline with a '\'
 - '\' the standard line continuation character...in bash etc.
i.e. to continue treat next visible line, as a continuation of the current 'code' line have '\' before the end of visible line
 - clearly terminated by omitting a '\' at the end of the last line

Append and Insert

- Append places *text* after the current line in pattern space
- Insert places *text* before the current line in pattern space
- Each of these commands requires an immediate line continuation backslash ' \ ' following it to "escape" the NEWLINE that is entered when you press RETURN (or ENTER). *text* must begin on the next line.
- To use multiple lines, simply ESCAPE all but the last with a ' \ '
- If text begins with whitespace, sed will discard it unless you start the line with a ' \ '

No messing : multiple append/insert blocked!

Append and Insert can only be applied to a single line address, not a range of lines, to avoid accidentally making a total mess of a file.
Remember insert, inserts before current line in pattern space.
The line must be matched by pattern (line number (range) or regex pattern)

Example:

```
/<Pattern Matching SearchText Here>/\
```

Line 1 of inserted text

Line 2 of inserted text

would leave the following in the pattern space:

Line 1 of inserted text

Line 2 of inserted text

```
<Current Line in Pattern space Matching SearchText Here>
```

Note

- 1 – ' at end of line-1 in text above escapes newline for line continuation
- 2 – and no ' at end of line-2, shows no line continuation, end of text
- 3 – ' at start of line-2 to ensure spaces at beginning of line aren't ignored;
(see last line on previous slide – insert lines starting with space ignored!)
Obviously, not needed unless line needs to be padded out with spaces.

Change – can change ranges...!?

- Unlike Insert and Append,
 - Which can only be applied to a single line – local copies pointless!Change can be applied to
 - either a single line address : matching a regex
 - or a range of addresses : matching regexes separated by a comma
- When a single Change command is applied to a range,
 - the entire range is replaced by text specified with change,
 - not each line
- BUT, ranges if braced { } and changes each line in a range
 - one of a group of commands enclosed in { }
 - that act on a range of lines,
 - Then the Change command is applied to each line
i.e. each line will be replaced with *text* (PTO-- ④)

Change changing range Examples

- Mail is normally stored on 'nix systems as a long contiguous textfile
- Each mail message header is a range of lines :
 - beginning with a line that begins with From /[^]From /
 - until the first blank line. /[^]\$/
- 1. Range Example 1 One for all :
replaces **all** lines with a single copy of <Mail Header Removed>.
/[^]From /,[^]\$/c\
- <Mail Header Removed>
- 2. All from One : Brace range { }
replaces **each** line in range with <Mail Header Removed>
beginning with /From /
to first blank line
/[^]From /,[^]\$/ {
s/[^]From //p
c\
<Mail Header Removed>
}

Swap /[^]From / at the beginning of a line, /[^]
With nothing //, but print to pattern space, p
p is vital, as c\ only applies to pattern space
(If not printed, then nothing to change!)
And change to <Mail Header Removed>

Side Effects – new text assumed correct!?

- Only the old input file is processed by sed, new is output
- New text appended or inserted is assumed to be correct
- Avoids risk of corrupting new text with commands for old
- Change clears the pattern space, replacing old with new
 - No command following the change command in the script is applied
- Insert and Append do not clear the pattern space
 - So none of the commands in the script will be applied to the new text that is inserted or appended
 - the next sed command will only process the text from the original input file, in the pattern space
- No matter what changes are made to pattern space,
 - the text from change, insert, or append will be output as supplied
- This is true even if default output (or old) is suppressed
 - using -n or #n option, *text* will still be output for these commands
- Need to check current implementation to be sure.

Delete

- Delete takes 0, 1 or 2 addresses & deletes respectively :-
 - 0 - either the current pattern space,
 - 1 - the pattern space when it matches the first address,
 - 2 - the range of lines contained within two addresses
- Once delete is executed,
 - no other commands are applied to pattern space.
 - Instead,
 - the next line from the edit file is read into pattern space
 - The edit script starts all over again with the first edit instruction
- Delete deletes the entire line,
 - not just the part that matches the address.
- To delete a portion of a line,
 - use substitute with a blank replacement string

NOT – Negation ! DON'T

If an address is followed by an exclamation point (!), the associated command is applied to all lines that don't match the address or address range

Example:

1,5!d

would delete all lines except 1 through 5

/black/!s/cow/horse/

would substitute "horse" for "cow"

on all lines, **except** those that contained "black"

"The brown cow" -> "The brown horse"

"The black cow" -> "The black cow"

Extract to & from other files with Read and Write

- These commands permit extraction to and from other files.
 - allow you to work directly with files
 - Both take a single argument, a file name
- The Read ([address]r filename)
 - The read command takes
 - an optional single address
 - and reads the specified file into *pattern space* after the addressed line.
 - It cannot operate on a range of lines
- Write ([address1[, address2]]w filename)
 - Write takes
 - an optional line address
 - or range of addresses
 - and writes the contents of *pattern space* to the specified file

Cases and places for spaces, else chaos!

- There must be a single space between the r or w command and the filename.
- There must not be any spaces after the filename or sed will include them as part of the file name
- Read will not complain if the file doesn't exist
- Write will
 - create it if it doesn't exist;
 - overwrite it if it already exists
 - unless created during the current invocation of sed
 - in which case write will append to it
- If there are multiple commands writing to the same file, each will append to it
- There are a maximum of ten files per script

Uses for Read and Write

- Read can be used for substitution in form letters
cat sedscrip
/^<Company-list>/r company.list
/^Company-list>/d

cat company.list Remember
CompUSA NEED a space between r and filename
MicroCenter NO space after filename
Lucky Computers

Clearly, editing the company.list changes the edit!

Read can be used for substitution in form letters e.g.

Replaces placeholder with file contents, then deletes placeholder
formletter:-

To purchase your own copy of FrontPage, contact any of the following companies: <Company-list> Thank you	Sedscrip:- /^<Company-list>/r company.list /^Company-list>/d
sed -f sedscrip formletter	
To purchase your own copy of FrontPage, contact any of the following companies: CompUSA MicroCenter Lucky Computers Thank you	Company-list:- CompUSA MicroCenter Lucky Computers

Clearly, editing the company.list changes the edit!

Simple data extraction / selection to files!

- Write can be used to pull selected lines and segregate them into individual files
- Suppose I have a customer file (customers) containing the following (US states) data:
 - John Cleese WA
 - Jerry Smith CA
 - Tom Jones VA
 - Gene Autry CA
 - Ranger Bob VA
 - Annie Oakley CA

Selective splitting of files on contents

Now, suppose I want to segregate all of the customers from each state into a file of their own

`cat sedscr`

<code>/CA\$/w customers.CA</code>	Remember
<code>/VA\$/w customers.VA</code>	a space between w and filename
<code>/WA\$/w customers.WA</code>	NO space after filename

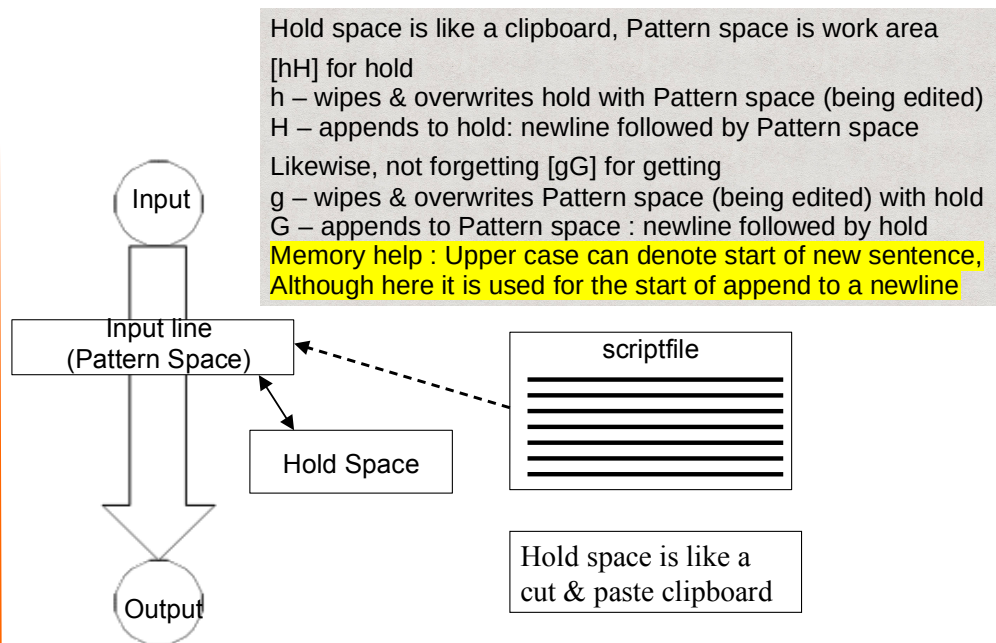
`sed -f sedscr customers`

will create files for each state that contain only the customers from that state

Transform

- The Transform command (y) operates like **tr**, doing a 1-to-1 or character-to-character replacement
- Transform accepts zero, one or two addresses
- `[address[, address]]y/abc/xyz/`
 - every a within the specified address(es) is transformed to an x. The same is true for b to y and c to z
 - `y/abcdefghijklmnopqrstuvwxyz/ABCDEFGHIJKLMNOPQRSTUVWXYZ/` changes **all** lower case characters on the addressed line to upper case
 - If you only want to do specific characters, or a word, in the line, it is much more difficult and requires use of the *hold space*

Cut & Paste with Holding Patterns!



Copy Pattern Space to Hold Space

- Like copy to clipboard as in copy & paste
- The **h** and **H** commands move the contents of *pattern space* to **hold space**
- **h** copies *pattern space* to *hold space*, **replacing** anything that was there
- **H appends** an embedded NEWLINE (“\n”) to whatever is currently in *hold space* followed by the contents of *pattern space*
 - Even if the *hold space* is empty, the embedded NEWLINE is appended to *hold space* first

Get Contents of Hold Space ~ paste

- Like paste from clipboard as in cut & paste
- **g** and **G** get the contents of *hold space* and place it in *pattern space*
- **g** copies the contents of *hold space* into *pattern space*, replacing whatever was there
- **G** appends an embedded NEWLINE character (“\n”) followed by the contents of *hold space* to *pattern space*
- when *pattern space* is empty, a NEWLINE is still appended to *pattern space* before contents of *hold space*
 - Helps delineate for rearrangement as we’ll see soon.

Memory help : Upper case can denote start of new sentence, Although here it is used for the start of append to a newline

CAPITALISTS!? (all are equal, some more than others!?)

- If a word in a specific phrase should be capitalized, e.g. changing “the abc statement” to “the ABC statement”
- A script to do this looks like this (explanation on following slides):

```
/the .* statement/{  
h  
s/. *the \(.*\) statement.*\1/  
y/abcdefghijklmnopqrstuvwxyz/ABCDEFGHIJKLMNOPQRSTUVWXYZ/G  
s/\(.*\) \n \(.*the \(.*\) statement.*\) \2\1\3/  
}
```

Similar but different, same basic regex pattern, but different bracket selections for different backreferences

Extended regex, use character classes [:upper:], [:lower:] were standard in GNU & BSD Basic Regular Expressions (BRE) but now need to specify extended in both to be POSIX compliant! See ~+8 slides ‘As special..’

Simply done by using : sed -E ...Implementations may vary.
NOT TO BE CONFUSED WITH sed -e edit1 e edit2 etc. for multiple edits at once from a one – liner at the shell command line....

Remember no spaces around the braces...for many edit cmds see slide 18!

So How Does It Work?

- **/the .* statement/**
 - The regex limits the procedure to lines that match “the .* statement” ... as ‘.’ regex does not match newline
- **h** copies current line into hold space, overwriting hold
 - After the h, pattern space and hold space are identical
 - pattern space - “find the abc statement”
 - hold space - “find the abc statement”
- **s/. *the \(.*\) statement.* \1/**
 - gets text between first occurrence of words the statement (\1)
 - But could be ‘the any text except ‘\n’ statement’
 - replaces entire line (in pattern space, editing area) with it
 - pattern space - “abc”
 - hold space - “find the abc statement”

CAPITALISTS!? (all are equal, some more than others!?)

- 1) **/the .* statement/{**
- 2) **h**
- 3) **s/. *the \(.*\) statement.*\1/**
- 4) **y/abcdefghijklmnopqrstuvwxyz/ABCDEFGHIJKLMNOPQRSTUVWXYZ/G**
- 5) **s/\(.*\) \n \(.*the \(.*\) statement.*\) \2\1\3/**
- 6) **}**

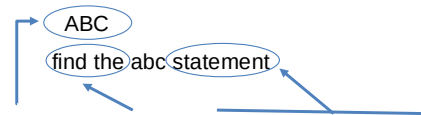
- 1 matches any line with “the (any string except newline) statement”
- 2 transfers it to hold, overwriting anything there
- 3 reduces pattern space using backreference \1, to refer to the \(.*\) match between “the” & “statement”.

- 4 **y** - Transforms what’s left in pattern space at 3 to UPPERCASE
G – appends newline + hold space to 4 in pattern space

- 5 Reorders using backreferences – explained in following slides.
\1 : the transformed uppercase from pattern space in 4y, which was originally between “the” & “statement”, is now placed between them

Holding patterns... all up in the air!?

- `y/abc.../ABC.../G` changes each lowercase to uppercase
 - pattern space - "ABC"
 - hold space - "find the abc statement"
- The G command fortuitously appends a NEWLINE (`"\n"`), to pattern space, before appending hold text;
- This is critical, as `'.'` matches any character except `"\n"`; so without the newline `\(.*)` would fail to terminate and fail to select PRINT
- So pattern space, where editing is done, now has these 2 lines:-



- Then `s/\(.*)\n\(.*.the \).*(statement.*)/\2\1\3/`
 - matches three different parts of the pattern space as shown
 - and rearranges them as :- find the ABC statement
 - NB 'find' isn't deleted, because all on line until 'the' is copied!

Print

- The Print command (`p`)
 - can be used to force the pattern space to be output, even if the `-n` or `#n` option has been specified
- Syntax: `[address1[, address2]]p`
 - Note:
if the `-n` or `#n` option has not been specified, `p` will cause the line to be output twice!
- Examples:
`1,5p` will display lines 1 through 5
`/^$/,/$/p` will display the lines from the first blank line through the last line of the file

Quit

- **quit** causes **sed** to stop reading new input lines and stop sending them to standard output
- It takes at most a single line address
 - Once a line matching the address is reached, the script will be terminated
 - This can be used to save time when you only want to process some portion of the beginning of a file
- Example:
- To print the first 100 lines of a file (like *head*) use:
 - *Sed '100q' filename*
 - sed will, by default, send the first 100 lines of *filename* to standard output and then quit processing
 - Of course : *head -100 filename* will do same

Regex Metacharacters for sed

Character Use i.e. matches

.	any single character except NEWLINE
*	zero or more occurrences of the single preceding char
[?]	any one of the class of characters contained
\	Escapes follow special character
\()	saves enclosed pattern for backreferencing
\n	matches the n th pattern saved via \()
\{n,m\}	a range of occurrences of the regex immediately preceding it <ul style="list-style-type: none">\{n\} will match exactly n occurrences\{n,\} will match at least n occurrences (note the comma!)\{n,m\} will match any number of occurrences from n to m
^	beginning of line
\$	end of line
&	prints all matched text when used in a replacement string

The only difference between basic and extended regular expressions is in the behavior of a few characters:

'?', '+', parentheses ('{}').

Basic regular expressions require these to be escaped
... if you want them to behave as special characters.

Extended regular expressions require these to be escaped
... if you want them to match a literal character.

Special : escape BRE Literal : escape ERE

More ridiculous conventions...

GNU & BSD used ERE for ages without specifying E,
but now need to specify an E to be POSIX compliant!

As special : escape in BRE As literal : escape in ERE

To match the literal string 'abc?'.

'abc?' 'abc\?'

To match one or more 'c's.

'c\+' 'c+'

To match three or more 'a's.

'a\{3,\}' 'a{3,}'

To match either 'abcabc' or 'abcabcabc'.

'\{abc\}\{2,3\}' '(abc){2,3}'

Backreference numbers must still be escaped in both

'\{abc*\}\1' '(abc*)\1'

The 'Capitalisation' slides re 'the ABC statement', can be rewritten as above. See the manual for more info..

<https://www.gnu.org/software/sed/manual/sed.html#BRE-vs-ERE>

Online repositories of example seds!

- Check, possibly in this order
- local system documentation : man, info, or GUI
 - As it should be specific to local implementation
- The original, for general information:
 - <https://www.gnu.org/software/sed/>
 - <https://www.gnu.org/software/sed/manual/>
- For specific needs search 'sed examples' online, these will change, and update, so best to search, But 10/2016 here is a reasonable one:-
 - <http://sed.sourceforge.net/sed1line.txt>
 - With links to other resources (books, source) within.

Example of a tree cmd using seds!

find . -type d -print | sed \ # line continuations 1) – 3) for explanation
 1) -e ' /\.\$/d '\ # delete lines with a single char ('e.g. '.')
 2) -e s:[^/]*\([^/]*\)\$:|-----\1: \ # pad out last level name\$, backrefn \1
 # switch last_noslashes_in_line, for |-----last_noslashes_in_line
 3) -e s:[^/]*:/: :g" # then pad out lower level dirs with spaces
 # all non-slash chars, up to last slash/dir

\$ tree

```

.
├── bags
│   ├── bag_in_bags
│   │   ├── sedtree
│   │   ├── trav_1
│   │   └── walk
│   ├── sedtree
│   ├── trav_1
│   └── walk
├── sedtree
├── trav_1
├── tree.sh
└── walk
    
```

\$ find . -type d
 ./bags
 ./bags/bag_in_bags
 .
 ./bags
 ./bags/bag_in_bags
 NB
 Omits hidden dirs

After 1)
 ./bags
 ./bags/bag_in_bags

After 2)
 |-----bags
 ./|-----bag_in_bags

After 3)
 |-----bags
 | |-----bag_in_bags

Extract from bash tree script for OS X

```
ls -R | grep ":$" | \
```

```
sed -e 's:/ /' -e 's/[^-][^V]*V/--/g' -e 's/^/ /' -e 's/-/|/'
```

- 1) 's:/ /' : remove colons [^-][^V]* = anything but these!
- 2) 's/[^-][^V]*V/--/g' : replace higher level folder names with 2 dashes
- 3) 's/^/ /' : indent graph three spaces, to stand out
- 4) 's/-/|/' : replace first dash with a vertical bar

```
$ ls -R
.:
bags sedtree trav_1 tree.sh walk

./bags:
bag_in_bags sedtree trav_1 walk

./bags/bag_in_bags:
sedtree trav_1 walk
```

```
$ tree
tree
├── bags
│   ├── bag_in_bags
│   ├── sedtree
│   ├── trav_1
│   └── walk
├── sedtree
│   ├── trav_1
│   └── walk
├── sedtree
│   ├── trav_1
│   └── tree.sh
└── walk
```

After 1) & 2).

```
--bags
----bag_in_bags
```

After 3) start of line padded out with spaces

```
    --bags
    ----bag_in_bags
```

After 4) Final result

```
    |--bags
    |--bag_in_bags
```

"Definitive Guide to sed"

sed (stream editor) efficiently transforms text files. This book is a full tutorial and reference, to greatly help any user (beginner to experienced) learn and review sed.

sed is a uniquely useful text processing tool, installed by default on Unix computers. sed can also be used on a Windows PC.

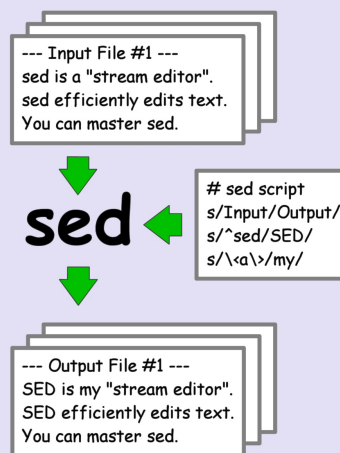
By Daniel Goldman

Publisher: [EHDP press](#)

Release Date: November 2013

Pages: 132

Definitive Guide to sed Tutorial and Reference



Daniel A. Goldman

Extras – for later

- Sed to edit directory hierarchy
 - Again best done in conjunction with bash scripts
 - directory crawling (directory tree walking) loops
 - File selection using regex with
 - find
 - ls | grep
 - Or list of files, in a file!

Example, bash snippet to edit files in current directory

for file in * # short for files in current directory!

do

```
    sed -i -f sedscrip $file # -i – edit in place, no redirect for output
done
```

may see bash script later that walks directory tree if not installed

sed & awk, 2nd Edition

UNIX Power Tools

By [Dale Dougherty](#), [Arnold Robbins](#)

Publisher: [O'Reilly Media](#)

Release Date: November 2010

Pages: 432

...and another book

A smaller pocket reference by Robbins, who has written another reasonable book on Linux internals, (dated now) but with lots of illustrative C code

