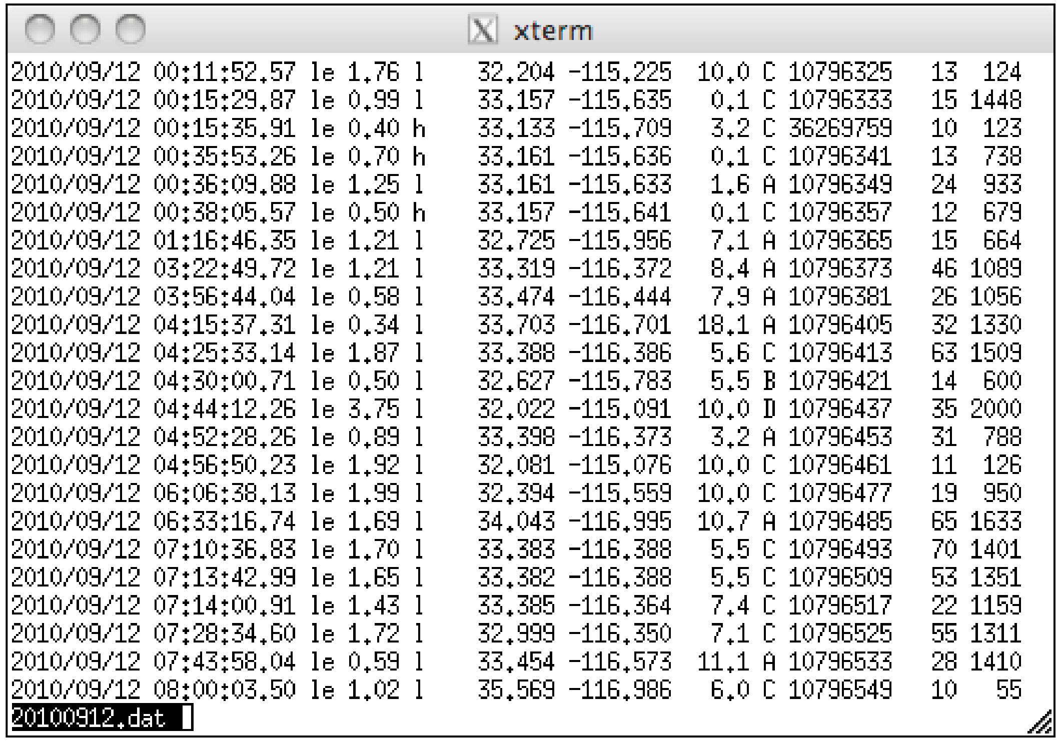
GEO 242: Numerical methods and modeling in the geosciences



```
2010/09/12 00:11:52.57 le 1.76 l    32.204 -115.225  10.0 C 10796325   13  124
2010/09/12 00:15:29.87 le 0.99 l    33.157 -115.635   0.1 C 10796333   15 1448
2010/09/12 00:15:35.91 le 0.40 h    33.133 -115.709   3.2 C 36269759   10  123
2010/09/12 00:35:53.26 le 0.70 h    33.161 -115.636   0.1 C 10796341   13  738
2010/09/12 00:36:09.88 le 1.25 l    33.161 -115.633   1.6 A 10796349   24  933
2010/09/12 00:38:05.57 le 0.50 h    33.157 -115.641   0.1 C 10796357   12  679
2010/09/12 01:16:46.35 le 1.21 l    32.725 -115.956   7.1 A 10796365   15  664
2010/09/12 03:22:49.72 le 1.21 l    33.319 -116.372   8.4 A 10796373   46 1089
2010/09/12 03:56:44.04 le 0.58 l    33.474 -116.444   7.9 A 10796381   26 1056
2010/09/12 04:15:37.31 le 0.34 l    33.703 -116.701  18.1 A 10796405   32 1330
2010/09/12 04:25:33.14 le 1.87 l    33.388 -116.386   5.6 C 10796413   63 1509
2010/09/12 04:30:00.71 le 0.50 l    32.627 -115.783   5.5 B 10796421   14  600
2010/09/12 04:44:12.26 le 3.75 l    32.022 -115.091  10.0 D 10796437   35 2000
2010/09/12 04:52:28.26 le 0.89 l    33.398 -116.373   3.2 A 10796453   31  788
2010/09/12 04:56:50.23 le 1.92 l    32.081 -115.076  10.0 C 10796461   11  126
2010/09/12 06:06:38.13 le 1.99 l    32.394 -115.559  10.0 C 10796477   19  950
2010/09/12 06:33:16.74 le 1.69 l    34.043 -116.995  10.7 A 10796485   65 1633
2010/09/12 07:10:36.83 le 1.70 l    33.383 -116.388   5.5 C 10796493   70 1401
2010/09/12 07:13:42.99 le 1.65 l    33.382 -116.388   5.5 C 10796509   53 1351
2010/09/12 07:14:00.91 le 1.43 l    33.385 -116.364   7.4 C 10796517   22 1159
2010/09/12 07:28:34.60 le 1.72 l    32.999 -116.350   7.1 C 10796525   55 1311
2010/09/12 07:43:58.04 le 0.59 l    33.454 -116.573  11.1 A 10796533   28 1410
2010/09/12 08:00:03.50 le 1.02 l    35.569 -116.986   6.0 C 10796549   10   55
20100912.dat
```

Lecture 3: Handling data

Outline

Why use ASCII for data?

Data handling commands in Unix

Simple data handling

Using awk

Variables in bash scripts

Handling data in multiple files

Regular expressions and grep

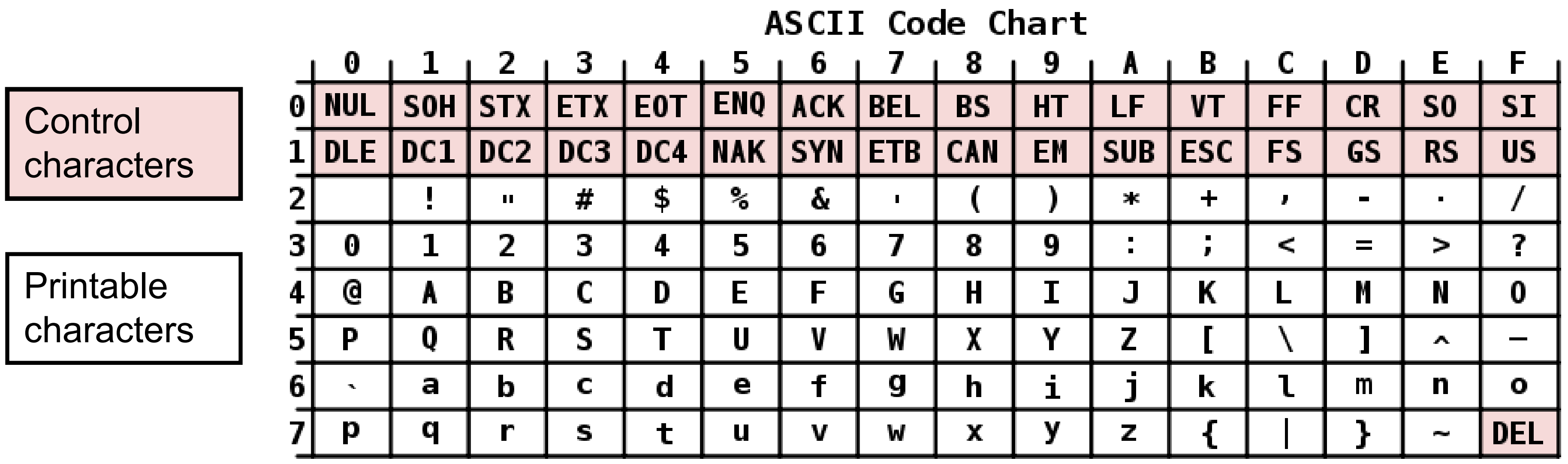
Using sed

Writing formatted output in the first place!

ASCII

The American Standard Code for Information Interchange (ASCII) was first published in 1963

Every letter and number, plus common punctuation is assigned a number



🟥 Control characters

⬜ Printable characters

ASCII Code Chart

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Why use ASCII for data?

ASCII is the de facto standard for text handing in computing – every computer can read it

Therefore maintaining data in ASCII format is the best way to ensure maximum compatibility between your data and software

If you have data in Excel (say) it is possible to export it in plain text, although you usually have to choose either comma-separated or tab-separated columns

Data handling commands

awk – perform operations on columned data

cat – concatenate files

cut – select columns based on defined separators

grep – search for a specific string in files

head – read the first (ten) lines of a file

paste – assemble columns from multiple files

sort – sort the contents of a file alphabetically and/or numerically

tail – read the last (ten) lines of a file

cat

concatenate files and write to standard output

```
cat <file1> <file2> …
```

will append file2 to file1 and write to standard output

head

view the first few lines of a file

```
head <filename>
```

shows the first 10 lines of ‘filename’

```
head -<number> <filename>
```

shows the first ‘number’ lines of ‘filename’

tail

view the last few lines of a file

```
tail <filename>
```

shows the last 10 lines of ‘filename’

```
tail -<number> <filename>
```

shows the last ‘number’ lines of ‘filename’

sort

sorts file contents based on the start of each line

```
sort <filename>
```

sorts the files in ASCIIbetical order

```
sort -f <filename>
```

ignores case

```
sort -n <filename>
```

sorts numerically

```
sort -r <filename>
```

shows results in reverse order

paste

assembles columns from multiple files or multiple columns from a single input

```
paste <file1> <file2> …
```

combines the columns of the specified files and outputs to standard output

When fed standard input, paste can change the number of columns

```
ls | paste - - -
```

writes a directory listing out in three columns

```
cat <file> | paste - -
```

writes each pair of lines of ‘file’ out as two columns

cut

extracts columns from a file

```
cut -c <char_num(s)> <file>
```

extracts character(s) ‘char_num(s)’ using the symbol ‘delim’ to delimit the column

```
cut -f <col_num(s)> -d <delim> <file>
```

extracts column(s) ‘col_nums’ using the symbol ‘delim’ to delimit the column

Character numbers and column numbers can be defined as comma-separated lists or hyphenated ranges, or both – e.g. 1,3 (1 and 3), 4-7 (4 to 7), 2,6-11 (2, and 6 to 11)…

grep

search for a specific string in one or more files

```
grep <word> <file(s)>
```

writes each line in ‘file(s)’ that contains ‘word’ to standard output

```
grep '<phrase in quotes>' <file(s)>
```

if the search string contains spaces, put the phrase in quotes and grep can search for it

```
grep -n <word> <file(s)>
```

returns the line number of each file, along with the line

```
grep -i <word> <files>
```

case insensitive search

awk

a command/scripting language that specializes in manipulating columns in data files

This is going to require more than one slide! The full syntax is:

```
awk 'BEGIN{<commands…>} {<commands…>}
END{<commands…>}' <set-variables> <file>
```

However, in most cases, the more simple syntax is applicable:

```
    awk '{<commands…>}' <file>
```

awk

By default the expected column delimiter in files is spaces. To use other symbols, use this option:

```
awk -F<delimiter> '{<commands…>}' <file>
```

So if you have a comma-delimited file, for instance:

```
awk -F,'{<commands…>}' <file>
```

awk: print

When the simple awk syntax is used, awk will execute the same set of commands on every line of the input file

The most basic awk command is `print` – this writes output to standard output

`print` can write a defined character string (i.e. text), or the content of various columns or variables, or actions on those columns or variables

```
awk '{print "boo"}' <file>
```

would print "boo" for every line in 'file'

[Note that `awk` uses single quotes outside the curly brackets, and double quotes inside…]

awk: print

awk refers to individual columns within each line of the input file as `$1` (column 1), `$2` (column 2), etc (the whole line is referred to as `$0`).

```
    awk '{print $1, $3}' <file>
```

will print columns 1 and 3 of ‘file’

```
awk '{print "I like",$1,"more than",$3}' <file>
awk '{print "I like "$1" more than "$3}' <file>
```

both will print the same output. What is the difference?

awk: built-in variables

awk has several built-in variables. The most important:

`NF` – ‘number of fields’ = number of columns in the current line

`NR` – ’number of records’ = the current line number

These can be used in a `print` command in `awk` in a similar way to the way we used the column designators `$1` and `$3`:

```
awk '{print "Line",NR,"has",NF,"columns"}' <file>
```

awk: simple math

If columns contain numerical data, it is possible to do simple mathematical operations on them

If your file has four columns, two sets of x,y coordinates marking the vertices of a straight line:

```
<x1> <y1> <x2> <y2>
```

we could find the centers of the lines like so:

```
awk '{print ($1+$3)/2, ($2+$4)/2}' <file>
```

and the lengths of the lines by:

```
awk '{print sqrt(($3-$1)^2+($4-$2)^2)}' <file>
```

awk: built in math functions

`awk` has several useful mathematical functions built in:

`sin`, `cos`, `atan2`, `sqrt`, `exp`, `log`

The trig functions expect data in radians, so maybe some conversions may be necessary:

```
awk '{print sin($1*3.14159/180)}' <file>
```

```
awk '{print cos($1*3.14159/180)}' <file>
```

the previous two assume that column 1 contains angles in degrees. The next assumes two pairs of x,y points again:

```
awk '{print atan2($3-$1,$4-$2)*180/3.14159}' <file>
```

what does that do?

awk: if-statements

awk is actually a fully-fledged scripting language that can be used for much more complex purposes

One consequence of this is that it has the capability of conditional statements (i.e. if statements):

```
    awk '{if (NR==5) print $0}' <file>
```

this prints the 5th line of the file.

If-statements can also lead to the execution of multiple commands:

```
    awk '{if (NR==5){print "Line",NR,"is:"; print $0}}'
<file>
```

(note that all brackets, curly or otherwise are closed)

awk: relational operators

Essential for if-statements are the relational operators – ways of comparing two quantities:

`==` – is equal to

`!=` – is not equal to

`>` – greater than

`<` – less than

`>=` – greater than or equal to

`<=` – less than or equal to

Conditional operators can also be used, i.e.

`&&` – and

`||` – or

awk: complex ‘if’s

Some examples of more complex if-statements:

```
awk '{if ((NR>=2)&&(NF==6)) print $0}' <file>
```

will ignore the first line of ‘file’ and any lines that do not have 6 columns

```
awk '{if (($4=="good")||($4=="excellent")) print $0}'
  <file>
```

will print all lines that have ‘good’ or ‘excellent’ in column 4

```
awk '{if (($4=="good")||($4=="excellent")) print $0; else
  print "bad"}' <file>
```

will print all lines that have ‘good’ or ‘excellent’ in column 4 in full; lines that do not have ‘good’ or ‘excellent’ will return ‘bad’

awk: variables

awk can also use user-defined variables – which means that you can simplify mathematical expressions somewhat, i.e.

```
awk '{a=$3-$1; b=$4-$2; print "gradient:",b/a,
  "inclination:",atan2(b,a)*180/3.14159}' <file>
```

awk: BEGIN and END

If you need to set initial values of a variable, you can do that with the 'BEGIN' part of an awk statement – these are things that are executed before awk runs line-by-line through the input file.

Similarly, the 'END' portion of the command is executed after awk has gone through all of the lines of the file.

An example:

```
awk 'BEGIN{total=0}{total=total+$4}END{print total/NR}'
  <file>
```

What do you think this does?

Other useful commands

`wc` – counts lines, words and characters in a text file

`gmt info` – a GMT command that finds the maximum and minimum values of each column in a file (where possible) and does a line count

Handling multiple files

So far we have seen (I hope) that awk is sophisticated tool for dealing with data in a single text file. Often, though, data comes in multiple files. How best to handle that?

Two general approaches come to mind:

1) Aggregate all the files together and deal with them all at once with awk
2) Deal with each file separately with awk

Aggregating files

We have already met the command `cat` that can be used for concatenation of files

```
cat *.dat > all_files.dat
```

would aggregate all of the '.dat' files in the present directory, in the order they are listed, into a file called 'all_files.dat'. You can then manipulate this file with awk, as before.

Alternatively, you can pipe the output of `cat` directly into `awk`:

```
cat *.dat | awk 'BEGIN{total=0}
  {total=total+$4}END{print total/NR}'
```

Dealing with multiple files

A more sophisticated approach would be to write a script to go through each file in turn

A simple way of doing this is a for loop in bash:

```
#!/bin/bash

# simple script to loop through three files

for infile in 20100901.dat 20100902.dat 20100903.dat

do

    awk 'BEGIN{total=0}{total=total+$4}
      END{print total/NR}' $infile

done
```

Dealing with multiple files

For loops in bash can also read filenames directly, which is really useful:

```
#!/bin/bash

# simple script to loop through all the files

for infile in *.dat

do

    awk 'BEGIN{total=0}{total=total+$4}
      END{print total/NR}' $infile

done
```

Variables in bash

A for loop defines a variable (in the case of the previous example, 'infile'), and gives that variable each of the values/strings that follow 'in', at each step

To call the variable after it is defined, you place a '$' sign in front, i.e.

```
for blah in value1 value2 …

do

     echo $blah

done
```

You can define your own variables without running a loop – simply declare them (no spaces though!):

```
wah=45

woo='hoo'
```

Backwards quotes

One useful way of generating information for a variable is to use the ‘backwards quotes’ syntax

If a shell command is surrounded by backwards quotes, its output is written to a string, not to standard output

```
echo `ls *.dat`
```

would write the list as a sequence. We can use this to populate the terms in a foreach loop if you want to use some simple manipulations on the list, i.e.

```
for infile in `ls *.dat | head -3`

do

     …

done
```

Redirect and append

As well as redirecting standard output to a file, we can append it to a file. The difference is that redirecting (>) overwrites any file that is there and appending (>>) just adds to it.

In other words,

```
for blah in <filenames> ; do

  awk '{<commands>}' $blah > $outputfile

done
```

would rewrite the output file named by the variable $outputfile at each timestep. If the awk statement were altered like this:

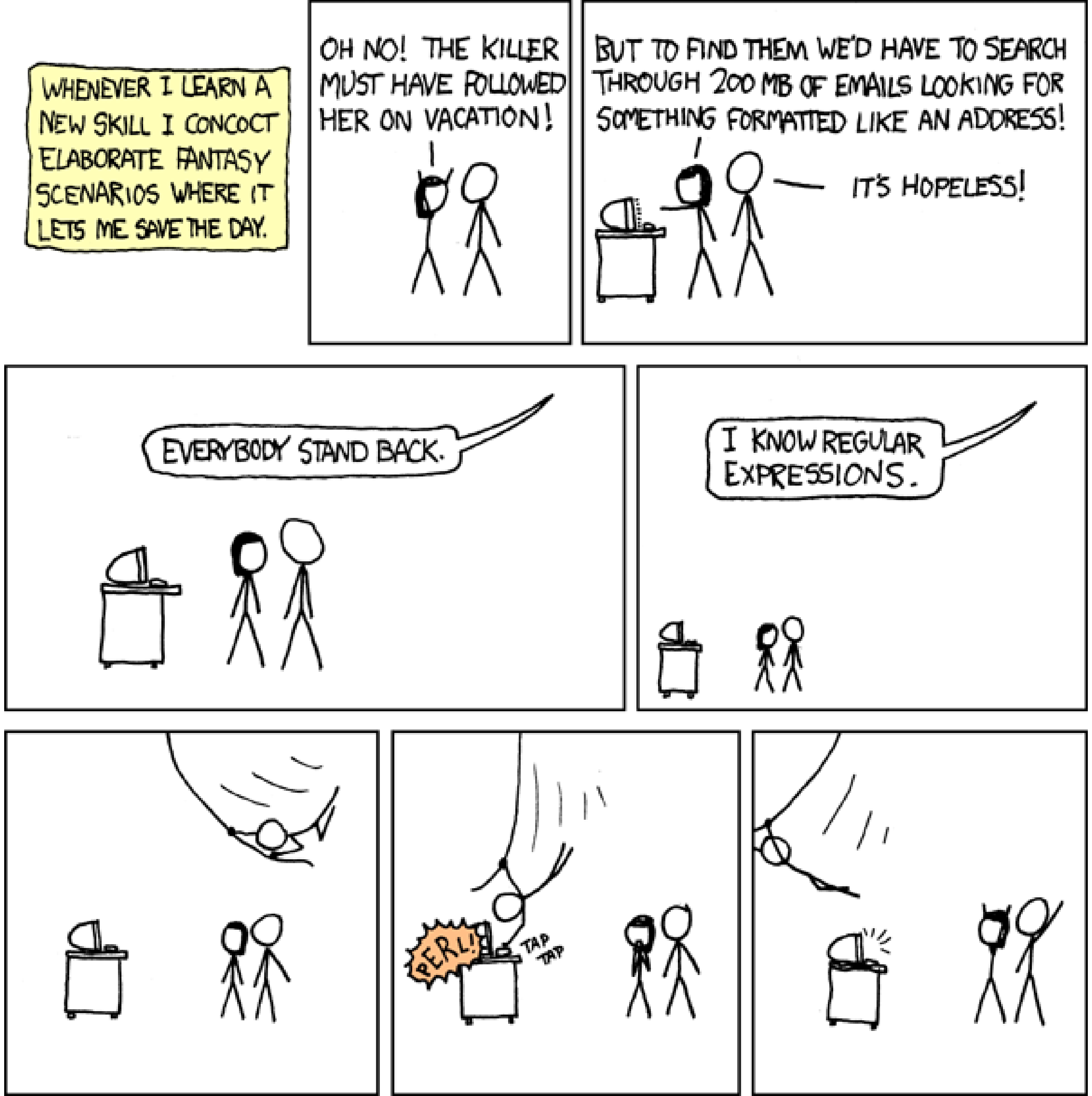
```
  awk '{<commands>}' $ blah >> $outputfile
```

the awk output would be added to the end of the file

A trial example

I have 29 files, each containing earthquake data from the southern California catalog for a single day in September 2010.

Make a file containing the catalog entries for the largest earthquake for each day…


WHENEVER I LEARN A NEW SKILL I CONCOCT ELABORATE FANTASY SCENARIOS WHERE IT LETS ME SAVE THE DAY.
OH NO! THE KILLER MUST HAVE FOLLOWED HER ON VACATION!
BUT TO FIND THEM WE'D HAVE TO SEARCH THROUGH 200 MB OF EMAILS LOOKING FOR SOMETHING FORMATTED LIKE AN ADDRESS!
IT'S HOPELESS!
EVERYBODY STAND BACK.
I KNOW REGULAR EXPRESSIONS.
PERL!
TAP TAP

Regular expressions

A regular expression, or ‘regexp’, is a sequence of characters that define a search pattern.

They are commonly used for ‘find and replace’ functions; word processors and text editors commonly have a regexp processor built-in.

Several scripting languages and shell commands can interpret regexp natively, including Perl, awk, grep and sed

A standardized format for regexp was established in 1992 as part of POSIX (Portable Operating System Interface), an IEEE standard

grep

“global regular expression print”

```
    grep '<regexp>' <file(s)>
```

writes each line in 'file(s)' that satisfies the regular expression 'regexp'

The quotes are necessary, otherwise the regexp will be interpreted by the shell before grep can!

Basic regular expressions (BREs)

^ (caret) => match a character at the start of a line

```
grep '^A' <file(s)>
```

will return every line that starts with an ‘A’

Basic regular expressions (BREs)

$ => match a character at the end of a line

```
grep 'e.$' <file(s)>
```

will return every line that ends with ‘e.’

Basic regular expressions (BREs)

[ ] (square brackets) => match a specific character or one of a set

```
grep 't[ou]' <file(s)>
```

will return every line that contains a ‘t’ followed by ‘o’ or ‘u’

Basic regular expressions (BREs)

[^] (caret in square brackets) => does not match a specific character or one of a set

```
grep 't[^ou]' <file(s)>
```

will return every line that contains a ‘t’ that is not followed by ‘o’ or ‘u’

sed

sed is another very useful text-handling command – it can be used to match regexps and then make substitutions, dumping the result to standard output. It is very useful for reformatting files on the fly… The basic syntax

```
sed 's/<regexp>/<replacement>/g'
                <file>
```

takes every instance of ‘regexp’ and substitutes it with ‘replacement’

sed

An example: take every example of ‘yes’ in the text and replace it with ‘no’:

```
sed 's/yes/no/g' <infile>
```

sed

The `d` command deletes a matching line

```
    sed '/yes/d' <infile>
```

deletes every line containing ‘yes’

If you just want to remove every instance of a word, then

```
    sed 's/yes//g' <infile>
```

replaces every ‘yes’ with nothing

Formatting your files

Once you have seen how frustrating it can be to have to read an irregularly formatted file, you might want to start producing your own regularly formatted files!

Consider having even column spacings, the same number formats, etc…

This can be achieved in awk using its `printf` command

awk: printf

If you want more control over how awk outputs text, the `printf` command supplies it.

`printf` uses the syntax for output of the command of the same name in the C language, namely

```
awk '{printf("text <output_type>",
<variable_name>)}' <file>
```

where ‘output_type’ is a code describing the format and ‘variable_name’ is the variable you want displayed in that format

awk: printf (output types)

The simplest codes specify the generic type of variable output, e.g.

```
awk '{printf("%f\n", <varname>)}' <file>
```

prints the value of ‘varname’ as a floating point number

```
awk '{printf("%d\n", <varname>)}' <file>
```

prints the value of ‘varname’ as an integer

```
awk '{printf("%s\n", <varname>)}' <file>
```

prints the value of ‘varname’ as a text string

‘varname’ here is a variable or a column identifier ($1, etc)

The `\n` ‘switch’ indicates ‘new line’ – just try running the command without it!

awk: printf (column width)

You can specify how many columns your output takes up, too, by adding numbers to your output code

```
awk '{printf("%8d\n", <varname>)}' <file>
```

prints the integer value of ‘varname’ within an 8 column space

```
awk '{printf("%08d\n", <varname>)}' <file>
```

prints the integer value of ‘varname’ as an 8-digit number, padding out the first columns with zeros if necessary

awk: printf (floats 1)

You can specify how many columns your output takes up, too, by adding numbers to your code

```
awk '{printf("%8.4f\n", <varname>)}'    <file>
```

prints the floating point value of ‘varname’ within an 8 column space, including 4 decimal places

Be aware that minus signs and the decimal points themselves count within the column limit. If more columns are needed, though, awk will supply them, which might ruin your column formatting

awk: printf (floats 2)

You can also specify the number of significant figures:

```
awk '{printf("%8.4g\n", <varname>)}' <file>
```

prints the floating point value of ‘varname’ within an 8 column space, including 4 significant figures

```
awk '{printf("%8.4e\n", <varname>)}' <file>
```

prints the four decimal places in exponential format

awk: printf (justification)

A minus sign at the beginning forces left justification (default is right), i.e.

```
awk '{printf("%-8f\n", <varname>)}'      <file>
```

prints the floating point value of ‘varname’ on the left of an 8 column space

```
awk '{printf("%8f\n", <varname>)}' <file>
```

prints the same thing on the right of an 8 column space

awk: printf (other text)

You can specify whatever other text you want in a `printf` statement, e.g.

```
    awk '{printf("My favorite number is %f\n",
    <varname>)}'   <file>
```

and you can have as many outputs as you have variables in a comma-separated list

```
    awk '{printf("First: %8f, Second: %8f.\n",
<var1>,<var2>)}'  <file>
```

awk: printf (commas and tabs)

If you want spreadsheet-readable output, that's pretty easy to achieve. The following gives comma-separated values:

```
    awk '{printf("%f, %f, %f,\n", <var1>, <var2>, <var3>)}' <file>
```

If you want tab-separated values, the `\t` switch inserts a tab, i.e.:

```
    awk '{printf(" %f\t %f\t %f\n", <var1>, <var2>, <var3>)}' <file>
```

An exercise: kml files

Earlier, we figured out how to identify the largest earthquake of a given day from a series of data files. Now I’d like you to write this information into a kml file that can be viewed in Google Earth.

Some tips:

- kml files are ASCII files
- If you copy a Google Earth overlay and paste it into a text editor, you will see the kml for that overlay
- The header and footer of the kml can be saved as separate text files and concatenated with the location information
- You need to use formatted output for the lat-long information, as awk by default truncates floating point numbers

The 1st assignment

This will be posted on Thursday.

There will be 2 questions.

They will involve simple Unix operations, extracting or converting data from files, and scripting.

The assignment and a tarball file of test data will be posted on iLearn, under ‘Course Materials’.

The scripts used to complete the project, plus the manipulated data files should be tarballed and e-mailed to me as a single file, before the deadline.