The Command Line

Matthew Bender

CMSC Command Line Workshop

October 16, 2015

Section 1

The Unix Philosophy

The Unix Philisophy

- Programs should do 1 thing and do it well

The Unix Philisophy

- Programs should do 1 thing and do it well
- Programs should be able to be combined and composed with each other

The Unix Philisophy

- Programs should do 1 thing and do it well
- Programs should be able to be combined and composed with each other
- Programs should handle text streams, because text is the universal interface

The Unix Philisophy

- Programs should do 1 thing and do it well
- Programs should be able to be combined and composed with each other
- Programs should handle text streams, because text is the universal interface
- Example: combine the `fgrep`, `sort`, and `uniq` commands to print lines containing `72.30.61.37`, without duplicates
- `$ fgrep "72.30.61.37" server.log | sort | uniq`

Programs as Text Filters

- Good programs will read in text from `stdin`, operate on it, and write text to `stdout`

Programs as Text Filters

- Good programs will read in text from `stdin`, operate on it, and write text to `stdout`
- Example: `rev` reads in lines on `stdin`, and write the reverse of each line on `stdout`

Programs as Text Filters

- Good programs will read in text from `stdin`, operate on it, and write text to `stdout`
- Example: `rev` reads in lines on `stdin`, and write the reverse of each line on `stdout`
- These programs are combined with `|` , which takes the `stdout` of one command and sends it to the `stdin` of another

Programs as Text Filters

- Good programs will read in text from `stdin`, operate on it, and write text to `stdout`
- Example: `rev` reads in lines on `stdin`, and write the reverse of each line on `stdout`
- These programs are combined with `|` , which takes the `stdout` of one command and sends it to the `stdin` of another
- These programs generally will also accept a file argument and read from that instead (and sometimes multiple files)

Programs as Text Filters

- Good programs will read in text from `stdin`, operate on it, and write text to `stdout`
- Example: `rev` reads in lines on `stdin`, and write the reverse of each line on `stdout`
- These programs are combined with `|` , which takes the `stdout` of one command and sends it to the `stdin` of another
- These programs generally will also accept a file argument and read from that instead (and sometimes multiple files)
- `$ rev file.txt`, `$ rev < file.txt`, and `$ cat file.txt | rev` will all do the same thing

Example Text Filters

The following can either read from `stdin` or a given file:

Example Text Filters

The following can either read from `stdin` or a given file:

- `cat`: prints contents of all given files on `stdout`, or prints `stdin` to `stdout` (add `-n` flag to number lines)

Example Text Filters

The following can either read from `stdin` or a given file:

- `cat`: prints contents of all given files on `stdout`, or prints `stdin` to `stdout` (add `-n` flag to number lines)
- `tac`: reverses the order of the lines on `stdin`, but not the actual lines

Example Text Filters

The following can either read from `stdin` or a given file:

- `cat`: prints contents of all given files on `stdout`, or prints `stdin` to `stdout` (add `-n` flag to number lines)
- `tac`: reverses the order of the lines on `stdin`, but not the actual lines
- `rev`: reverses the lines of `stdin`, but not the order of them

Example Text Filters

The following can either read from `stdin` or a given file:

- `cat`: prints contents of all given files on `stdout`, or prints `stdin` to `stdout` (add `-n` flag to number lines)
- `tac`: reverses the order of the lines on `stdin`, but not the actual lines
- `rev`: reverses the lines of `stdin`, but not the order of them
- `sort`: sorts the lines of input, by default in string order
  - `-f`: ignore case
  - `-n`: sort numerically
  - `-r`: reverse sort

Example Text Filters

The following can either read from `stdin` or a given file:

- `cat`: prints contents of all given files on `stdout`, or prints `stdin` to `stdout` (add `-n` flag to number lines)
- `tac`: reverses the order of the lines on `stdin`, but not the actual lines
- `rev`: reverses the lines of `stdin`, but not the order of them
- `sort`: sorts the lines of input, by default in string order
  - `-f`: ignore case
  - `-n`: sort numerically
  - `-r`: reverse sort
- `shuf`: randomly permute order of lines

Example Text Filters

The following can either read from `stdin` or a given file:

- `cat`: prints contents of all given files on `stdout`, or prints `stdin` to `stdout` (add `-n` flag to number lines)
- `tac`: reverses the order of the lines on `stdin`, but not the actual lines
- `rev`: reverses the lines of `stdin`, but not the order of them
- `sort`: sorts the lines of input, by default in string order
  - `-f`: ignore case
  - `-n`: sort numerically
  - `-r`: reverse sort
- `shuf`: randomly permute order of lines
- `head -N`: print first `N` lines (default 10)
- `tail -N`: print last `N` lines (default 10)

More Text Filters

Not every text filter necessarily just modifies its input:
`wc` prints the number of lines, words, and characters of its input.

- `-l`: print lines only
- `-w`: print words only
- `-c`: print characters(bytes) only

`bc` - **b**asic **c**alculator - read math expressions and write their value

grep

grep is a tool that takes a regular expression as argument and outputs all lines matching it.

grep has its origins in the text editor ed - the g/re/p command would print all lines matching the regex re

grep

grep is a tool that takes a regular expression as argument and outputs all lines matching it.
grep has its origins in the text editor ed - the g/re/p command would print all lines matching the regex re

- −v Invert match - print all lines not matching the given regex.

grep

grep is a tool that takes a regular expression as argument and outputs all lines matching it.
grep has its origins in the text editor ed - the g/re/p command would print all lines matching the regex re

- −v Invert match - print all lines not matching the given regex.
- −n Number lines - preceed each line with its line number

grep

grep is a tool that takes a regular expression as argument and outputs all lines matching it.
grep has its origins in the text editor ed - the g/re/p command would print all lines matching the regex re

- −v Invert match - print all lines not matching the given regex.
- −n Number lines - preceed each line with its line number
- −o Only match - print only the matched part of each line, not the whole line

grep

grep is a tool that takes a regular expression as argument and outputs all lines matching it.
grep has its origins in the text editor ed - the g/re/p command would print all lines matching the regex re

- −v Invert match - print all lines not matching the given regex.
- −n Number lines - preceed each line with its line number
- −o Only match - print only the matched part of each line, not the whole line
- −i Ignore case

grep

grep is a tool that takes a regular expression as argument and outputs all lines matching it.
grep has its origins in the text editor ed - the g/re/p command would print all lines matching the regex re

- −v Invert match - print all lines not matching the given regex.
- −n Number lines - preceed each line with its line number
- −o Only match - print only the matched part of each line, not the whole line
- −i Ignore case
- −q Quiet - produce no output, just set exit code based on if there was a match

grep

grep is a tool that takes a regular expression as argument and outputs all lines matching it.
grep has its origins in the text editor ed - the g/re/p command would print all lines matching the regex re

- −v Invert match - print all lines not matching the given regex.
- −n Number lines - preceed each line with its line number
- −o Only match - print only the matched part of each line, not the whole line
- −i Ignore case
- −q Quiet - produce no output, just set exit code based on if there was a match
- −A N, −B N, −C N After/Before/Context - print N lines after/before/both around matching lines

grep and Regular Expressions

The grep command accepts a regex as an argument, and prints only lines matching that argument to stdout

grep has 4 different regex modes:

grep and Regular Expressions

The grep command accepts a regex as an argument, and prints only lines matching that argument to stdout

grep has 4 different regex modes:

- Fixed string: `grep -F pattern` will match `pattern` exactly as a string, with no character having special meaning

grep and Regular Expressions

The grep command accepts a regex as an argument, and prints only lines matching that argument to stdout

grep has 4 different regex modes:

- Fixed string: `grep -F pattern` will match `pattern` exactly as a string, with no character having special meaning
- Basic (BRE): `grep pattern` matches with most characters matching themselves, but `. [ ] ^ $ *` all have special meanings (escape them with a \ to match them match themselves)

grep and Regular Expressions

The grep command accepts a regex as an argument, and prints only lines matching that argument to stdout

grep has 4 different regex modes:

- Fixed string: grep -F pattern will match pattern exactly as a string, with no character having special meaning
- Basic (BRE): grep pattern matches with most characters matching themselves, but . [ ] ^ $ * all have special meanings (escape them with a \ to match them match themselves)
- Extended (ERE): grep -E pattern does the same as BRE, but . [ ] | ^ $? * + { } ( ) are all metacharacters

`grep` and Regular Expressions

The `grep` command accepts a regex as an argument, and prints only lines matching that argument to `stdout`

`grep` has 4 different regex modes:

- Fixed string: `grep -F pattern` will match `pattern` exactly as a string, with no character having special meaning
- Basic (BRE): `grep pattern` matches with most characters matching themselves, but `. [ ] ^ $ *` all have special meanings (escape them with a \ to match them match themselves)
- Extended (ERE): `grep -E pattern` does the same as BRE, but `. [ ] | ^ $ ? * + { } ( )` are all metacharacters
- Perl (PCRE): `grep -P pattern` uses Perl-compatable regexes, look at the man page for `pcresyntax` and `pcrepattern` for more details.

grep and Regular Expressions

The grep command accepts a regex as an argument, and prints only lines matching that argument to stdout

grep has 4 different regex modes:

- Fixed string: grep -F pattern will match pattern exactly as a string, with no character having special meaning
- Basic (BRE): grep pattern matches with most characters matching themselves, but . [ ] ^ $ * all have special meanings (escape them with a \ to match them match themselves)
- Extended (ERE): grep -E pattern does the same as BRE, but . [ ] | ^ $? * + { } ( ) are all metacharacters
- Perl (PCRE): grep -P pattern uses Perl-compatable regexes, look at the man page for pcresyntax and pcrepattern for more details.
- fgrep and egrep are short for grep -F and grep -E, but the former usage is deprecated and the latter is preferred.

Section 2

Regular Expressions

Regular Expressions

Regular expressions (regex for short) are ways to match certain parts of text, in which certain characters can have special meanings

For example, `[a-z]{4,8}` will match any lowercase letter, 4 to 8 times in a row

The regex `^\s*$` will match any line containing only whitespace

Regexes can come in multiple "flavors", aka which characters have what meanings.

Basic Regular Expressions (BRE): the .

The . metacharacter will match any character

Print all lines with an a, then any char, then b, then any char, then c:

```
$ grep ’a.b.c’ words.txt
barbecue
drawback
```

etc...

Print all lines with an M followed by a .:

```
$ grep ’M\.’  words.txt
Y.M.C.A
```

etc...

Basic Regular Expressions (BRE): character classes

Use `[` and `]` to define a character class. This will match any character inside it.

Print all words with "`bl<vowel>z`":

```
$ grep 'bl[aeiou]z' words.txt
ablaze
blizzard
```

etc...

```
$ grep '[abc][abc][abc][abc]' words.txt
cabbage
tabacco
```

etc...

Basic Regular Expressions (BRE): character classes

We can add ranges to this, instead of listing each individual character:

```
$ grep '[a-d][e-h][i-l][m-p][q-t]' words.txt
chins
ocelot
```

etc...

Look for anything resembling a hex digit: (e.g. 0x3f)

```
$ grep '0x[0-9A-Fa-f][0-9A-Fa-f]' file.txt
```

Basic Regular Expressions (BRE): character classes

If the first character is a ˆ, then the character class is negated:

```
$ grep '[ˆaeiouy][ˆaeiouy][ˆaeiouy][ˆaeiouy]'
patchwork
thoughts
```

etc...

'i' before 'e' except after c?

```
$ grep 'cie' words.txt
$ grep '[ˆc]ei' words.txt
```

Basic Regular Expressions (BRE): character classes

The `\w` means match any alpha-numeric character, and `\W` matches the opposite.

Similarly, `\s` matches any whitespace, and `\S` matches the opposite.

`\b` matches any word boundary, and `\B` matches not at a word boundary.

`as\b` will match all words ending in `as` - even if the next character is whitespace, or a period, or dash, etc. It will not match things like `mast`.

Basic Regular Expressions (BRE): anchors

The ^ and $ characters match the beginning and ending of a line, respectively.

```
$ grep ’^abc’ words.txt
abcess
$ grep ’az$’ words.txt
spaz
```

How many 18-letter words start with 'a' and end with 'y'?

```
$ grep ’^a................y$’ words.txt
antidemocratically
```

Extended Regular Expressions (ERE): |

The `-E` flag gives us access to Extended Regular Expressions, with more metacharacters.

The should also be accessable by escaping them in BRE.

`patt1|patt2` will match `patt1` or `patt2`:

```
$ grep -E ’abc|xyz’ words.txt
abcess
hydroxyzine
```

This works with any regex pattern:

```
$ grep -E ’x...x|z[aeiou]z’ words.txt
exotoxin
pizazz
```

Extended Regular Expressions (ERE): ?

The ? matches either the previous token or the empty string, a.k.a. it makes a token optional:

```
$ grep -E '^abc?e' words.txt
abcess
abettor
```

Note how it makes a whole character class optional:

```
$ grep -E 'od[aeiou]?d' words.txt
goddess
wooded
```

Extended Regular Expressions (ERE): ⋆ and +

⋆ will match any number of the previous token, + will match one or more (⋆ is also available in BRE):

All words with no vowels:

```
$ grep −E ’ˆ[ˆaeiou]+$’ words.txt
crypt
```

Which words contain all the vowels in order?

```
$ grep −E ’a.⋆e.⋆i.⋆o.⋆u’ words.txt
haemoglobinous
```

How would you modify it to have only those 5 vowels?

Extended Regular Expressions (ERE): ranges

You can also specify a range after a token: {n} matches it exactly n times, {n,} matches n or more times, {,n} matches up to n times, and {n,m} matches n to m times:

All 20-letter words:

```
$ grep -E '^.{20}$' words.txt
```

All words containing 4 or more vowels in a row:

```
$ grep -E '[aeiou]{4,}' words.txt
```

Extended Regular Expressions (ERE): Grouping and Backreferences

Parentheses can be used for grouping: `(abc)def` is the same as `abcdef`, but `ab(cd|ef)gh` matched `abcdgh` or `abefgh`.

Parentheses also store their capture in a *backreference*, which can be referred to later in the regex with `\N`, where `N` is the number of the backreference.

All words containing the same 3-character string twice:

```
$ grep -E '(.{3}).*\1'
```

All words with the same first and last 3 characters, but reverse:

```
$ grep -E '^(.)(.)(.).*\3\2\1$'
```