Advanced Unix and Regular Expressions

James Curran and Tara Murphy

19th February 2007
Outline

1. Checkup
2. Matching
3. Regex
4. grep and sed
Do you remember what these do?
Do you remember what these do?

- cat, less
Do you remember what these do?

- cat, less
- head, tail
Do you remember what these do?

- cat, less
- head, tail
- grep
Do you remember what these do?

- cat, less
- head, tail
- grep
- wc
Do you remember what these do?

- cat, less
- head, tail
- grep
- wc
- sort, uniq
Do you remember what these do?

- cat, less
- head, tail
- grep
- wc
- sort, uniq
- most of these are *line oriented commands*
What about these ones?
What about these ones?

- cut, paste
What about these ones?

- cut, paste
- awk (and gawk)
What about these ones?

- cut, paste
- awk (and gawk)
- sort (again)
What about these ones?

- cut, paste
- awk (and gawk)
- sort (again)
- these ones are *column oriented commands*
And these ones?

- `tr`
And these ones?

- tr
- ed
And these ones?

- `tr`
- `ed`
- `sed`
We can use grep to match lines containing a given string:

```bash
1 % grep 'cs1' /etc/passwd
```

and we can return lines which don’t match using `-v`:

```bash
1 % grep -v 'cs1' /etc/passwd
```

If we want further constraints we can add more greps:

```bash
1 % grep 'cs1' /etc/passwd | grep '2006' | grep 'James'
```

what if we want to connect these constraints in some way?
Solving crosswords

- I have a crossword puzzle which I can’t finish
  ![Crossword Clue](image)
- and being a programmer I want the computer to do the work
- one step would be to find all the words containing k, d and g
  ```
  % grep k /usr/dict/words | grep d | grep g | sort -u
  Fredericksburg
  Hodgkin
  acknowledge
  acknowledgeable
  ...
  % grep k /usr/dict/words | grep d | grep g | sort -u | wc
  26 26 286
  ```
- hmm, still too many words to read!
Solving crosswords in Python

- we could write a Python program to find the answer:

```
>>> for line in open('/usr/dict/words'):
...     if len(line) > 6 and line[0] == 'k' and 
...     line[3] == 'd' and line[6] == 'g':
...         print line,
...
kindergarten
```

- but this seems like too much work
- what about AWK?
We need a language for describing text

- such pattern languages already exist:
  - e.g. filenames with wildcards: *.txt, note?ad.exe
  - e.g. wildcards in search in Word
- even hangman (not the gallows bit)
We need a language for describing text

- such pattern languages already exist:
  - e.g. filenames with wildcards: *.txt, note?ad.exe
  - e.g. wildcards in search in Word
- even hangman (not the gallows bit)
- Argh, but we need more power!
Regular expression describe strings

- *Regular expressions* are very expressive pattern language
- often abbreviated to *RE, regex, regexp*
- `grep` can interpret regular expressions:

```
1  % grep 'k..d..g.....' /usr/dict/words
2  kindergarten
3  %
```

- the dot (.) is a placeholder that represents *any* character
- `grep` prints any line which *contains* a match:

```
1  % grep 'k..g' /usr/dict/words | sort -u
2  Chungking
3  Mekong
4  Nanking
5  Peking
6  ...
```
We can force where matches start and end

- the *caret* or *hat* (`^`) matches the beginning of line
  ```
  % grep '^
dog' /usr/dict/words | sort -u
  dog
dogbane
dogberry
dogfish
  ...
  ```

- while dollars (`$`) matches the end of line
  ```
  % grep 'to$' /usr/dict/words | sort -u
  Alberto
  Callisto
  Caputo
  ...
  ```
We can force where matches start and end

- the caret and dollars can be used together:

```
% grep '^t....o$' /usr/dict/words | sort -u
  tattoo
termo
tomato
tupelo
tuxedo
%
```

- use dots to start/end the match later/earlier:

```
% grep '^x.' /usr/dict/words | sort -u
  Exeter
  Exxon
  Oxford
  Oxnard
  ...
```
Kleene star matches zero or more times

- what if we don’t know how many characters to match
- the star (*) matches the previous char\(^1\) zero or more times

```
% grep '^	.*$' /usr/dict/words | sort -u
```

```
taboo
tallyho
tango
```

- it is called the *Kleene star* operator
- the plus (+) matches one or more times
- but it requires the \(-E\) option to \texttt{grep}

\[\text{and not all versions of grep support \texttt{\(-E\)}}\]

\(^{1}\text{not true! This is a deliberate lie – for now}\)
Matching against alternatives

• sometimes we want to match one regular expression or another
• the pipe (|) matches one regular expression or another
• again it requires the -E option to `grep`
• or use `egrep` (which is identical to `grep -E`):

```bash
% egrep '^(dog|cat)' /usr/dict/words | sort -u

 cat
catabolic
cataclysm
...
dog
dogbane
```
Character sets represent many alternatives compactly

- to find all words starting with vowels we could go:
  ```
  % egrep '^[aeiou]' /usr/dict/words
  ```

- this is clumsy, and what if we wanted all consonants instead?
- regular expressions abbreviate this using character sets:
  ```
  % grep '^[[aeiou]]' /usr/dict/words
  ```

- the square brackets ([ and ] ) delimit the character set
- it can match any character inside the character set
Character sets represent ranges compactly

- if we wanted all lowercase characters we would need to write:
  ```bash
  % grep '^[abcdefghijklmnopqrstuvwxyz]' /usr/dict/words
  ```
- this is long and error prone
Character sets represent ranges compactly

- if we wanted all lowercase characters we would need to write:
  1. `% grep '^[abcdefghijklmnopqrstuvwxyz]' /usr/dict/words`
- this is long and error prone
- did you noticed I missed `n`?
Character sets represent ranges compactly

- if we wanted all lowercase characters we would need to write:
  
  ```
  % grep '^[abcdefghijklmnopqrstuvwxyz]' /usr/dict/words
  ```

- this is long and error prone

- did you noticed I missed n?

- to abbreviate this further, character sets support *ranges*:
  
  ```
  % grep '^[a-z]' /usr/dict/words
  ```

- so we can match Python variable names with:
  
  ```
  % grep '^[a-zA-Z_][a-zA-Z0-9_]*$' /usr/dict/words
  ```
Character sets also support set complements

- the *complement* of a set is all of the elements *not* in the set
- here this means all of the *characters not in the set*
- the caret (^) is used as the *first* character inside the set:

  ```
  1  % grep '^[^a-zA-Z]' /usr/dict/words
  2  10th
  3  1st
  4  2nd
  5  3rd
  6  ...
  ```

- the caret now means two things in regular expressions
grep’s command line arguments

- `-c` count the number of matches in a file
- `-F` fixed strings
- `-h` print no filenames on output
- `-i` case insensitive
- `-l` print only the filename of a match
- `-L` print only the filename of a non-match
- `-r` recursive search
- `-v` lines that don’t match
sed is a stream editor

- sed is a powerful language for streams editing
- it allows regular expressions to be used for modifying rather than just matching
- greedy matching
grouping and substitution

- reusing the whole match with &
- parentheses and back references
advanced sed and grep

- using multiple sed/greps together
- sed programs (operators like global and insensitive etc)