Regular Expressions

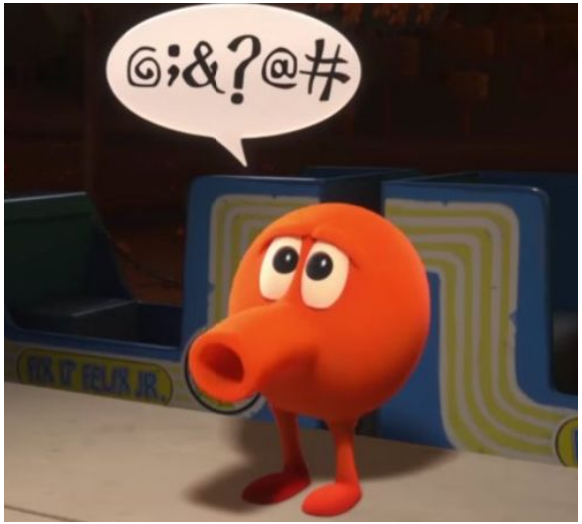
~/IFT383/Module-01/Lecture-01

Module 1 Objectives

- Explain how to use the Unix command interpreter
- Create and use regular expressions with grep
- Manipulate the basic rules of Regular Expressions
- Apply basic filters to search, edit, and reformat files containing structured lists
- Read lines from the beginning or end of a file
- Extract fields from files and sort files
- Use sed to edit and print an input stream, search and substitute strings, insert and delete lines of text

Agenda

- Introduction
- Meta Characters
- Escape Sequences
- Anchors
  - Line
  - Word
- Character Classes
- Quantifiers (repetition)
- Grouping
- Practice
- Commands and some UNIX shell review



Qbert
“Q*Bert”
Gottlieb - 1982
image from “Wreck-It Ralph”
Disney - 2012

Demo files

If you would like to follow along with the demo;

mkdir IFT383
cd ./IFT383
git clone https://cjingers@bitbucket.org/cjingers/ift383-files.git

Introduction

- Regular Expressions (RegEx) enable searching text by finding characters that match a pattern, rather that literal characters
- These patterns are defined using special combinations of characters, that can be interpreted by a scripting engine or programming runtimes
- Example; “85295”, “85212”, “92377”, “92376” ...
  - Literal series: “85295” <- the specific ZIP code of 85295
  - Pattern: [0-9]{5} <- digits 0 through 9 repeated exactly 5 times
    - [0-9] is a character class, by itself it matches a single character
    - {5} is one type of quantifier; there are many more!

Meta Characters

- Metacharacters are the scaffolding that define the variability of a regular expression

Metacharacter	Description
.	Matches any single character (except newline)
^	Anchor: beginning of line
$	Anchor: end of line
\<	Anchor: beginning of word
\>	Anchor: end of word
[ ]	Character class
[^]	Negated character class
|	Alternation; OR operator
{} ? + *	Quantifiers
()	Capture group
\	Escape (quote) character

Escape Character

- In order to use a metacharacter as a literal character in your pattern, you must use the escape character “\” followed by a single metacharacter to be used literally
  - Example: “I found $5 in the dryer!”
    - “$[0-9]” would fail to match “$5”
    - “\$[0-9]” would match “$5”
- You can also escape the escape character
  - Example: “C:\Windows\System32”
    - “[A-Z]:\\” would match “C:\”
  - Example: “\\asurite.ad.asu.edu\sysvol$\policy”
    - “\\\\[^\\]+” matches “\\asurite.ad.asu.edu”

Anchors (line)

- The metacharacters ^ and $ used individually, constrain the pattern to match at the beginning or end of a line of text
  - Example: “the dog looked longingly at the fresh bread.”
    - “(the)” would match BOTH instances
    - “^(the)” would only match “the” at the beginning of the line
  - Example: “7001 E Williams Field Rd. Mesa, AZ 85212”
    - “[0-9]+” would match “7001” and “85212”
    - “^[0-9]+” would match “7001” only
      - Note that “^[0-9]+” behaves differently than “[^0-9]+”
    - “[0-9]+$” would match “85212”
- You can use both metacharacters to match the entire line
  - Example: “a line” “a 2nd line”
    - “^A[a-z]+$” matches “a line” but not “a 2nd line”

Anchors (word)

- \< and \> work similarly to ^ and $. Rather than anchoring to line boundaries, \< and \> match to word boundaries.
- A ‘word’ is a contiguous series of letters, numbers and underscores “_”
- A ‘word boundary’ is the demarcation between a word character to a non-word character
  - Example: “The bill for playing billiards was $10”
    - “bill\>” will match only the first “bill”
    - “bill” and “\<bill” will match both
  - Example “amber lamb am slam, bam!”
    - “\<am\>” matches “am” but not “amber”, “lamb”, “slam” or “bam!”
- **Instructors note**: In my experience, this notation is less common outside of UNIX
  - Example: Microsoft PowerShell and Oracle Java use \b as a ‘word boundary character class
  - “\<am\>” translates to; “\bam\b”

Character Classes

- Specifies a group of characters that can satisfy a match
- Character classes are defined by enclosing characters within brackets; [ ]
  - Example: “quaint, quant quandary que”
    - “\<[aeiouq]+\>” will match “que”
    - “\<[quaint]+\>” will match “quaint” and “quant”
- Specify a range of characters using a dash “-” character
  - Example: “abcdefghijklmnop”
    - “[a-f]+” will match “abcdef”
- Invert a character set using a ^ (carrot) symbol at the beginning of the class
  - Example “abcdefghijklmnop”
    - “[^a-f]+” will match “ghijklmnop”

Predefined Character CLasses

- There are several predefined character classes for convenience
- Many GNU/Linux tools support both older UNIX style, and more modern syntax
  - You may encounter both styles in the wild

UNIX-style	Modern style	Similar to	Description
[[:lower:]]	\w (includes; upper, lower, digits and _)	[a-z]	Lowercase letters
[[:upper:]]		[A-Z]	Uppercase letters
[[:alpha:]]		[a-zA-Z]	Upper and lowercase letters
[[:alnum:]]		[a-zA-Z0-9]	Letters and numbers
[[:digit:]]	\d	[0-9]	numbers
[[:punct:]]			Quotes, commas, period
[[:blank:]]	\s, \t, \n		whitespace

Quantifiers

- A quantifier specifies the number of times the preceding element should be evaluated
  - Example: “1,0{3},0{3}” would match “1,000,000”
- These elements may be characters, character sets or capture groups

Syntax	Description
?	Element is optional and matched at most once. (0 or 1 times)
*	Element is matches 0 or more times
+	Element is matched 1 or more times
{n}	Exactly n times
{n,}	n or more times
{n,m}	N to m times

Grouping

- Grouping is denoted by parenthesis; ()
- Groups can be associated with quantifiers
  - Example: “There is a fee for the coffee”
    - “\b(cof)?fee\b” would match both underlined words
    - Recall “\b(cof)?fee\b” is equivalent to “\<(cof)?fee\>”
    - ? matches 0 or 1 times
- BASH and other scripting languages also support ‘capturing groups’ and named groups
  - Example in PowerShell using "(\d{3})(\d{3})(\d{4})"

```
PS E:\> ([RegEx]"(\d{3})(\d{3})(\d{4})").match("4807271007").groups[1..3] | ft

Success Name Captures Index Length Value
------- ---- -------- ----- ------ -----
   True 1    {1}          0      3 480
   True 2    {2}          3      3 727
   True 3    {3}          6      4 1007
```

Let's put it all together!

Image source: https://xkcd.com/208/

Challenge 1: The prequels were bad

Write a single regular expression that will match the lines which are underlined, but **NOT** the others.

HINT: Use quantifiers

HINT2: another solution would be to use an OR condition with several | (pipe) characters

Rocky V
Rocky IV
Star Wars: Episode I
Star Wars: Episode II
Star Wars: Episode III
Star Wars: Episode IV: Christmas Special
Star Wars: Episode IV
Star Wars: Episode V
Star Wars: Episode VI

Solution 1

“Star Wars: Episode I?VI?”

A more complex solution;

"^Star [^IV]+I?VI?$"

Others exist!

Rocky V
Rocky IV
Star Wars: Episode I
Star Wars: Episode II
Star Wars: Episode III
Star Wars: Episode IV: Christmas Special
Star Wars: Episode IV
Star Wars: Episode V
Star Wars: Episode VI

Challenge 2: How many lights?

Write a single regular expression to match the first three lines, but not the last line.

Do not use a quantifier or OR; can you accomplish the goal using a character class?

There are FOUR lights!
There are four lights.
There are 4 lights.
There are 3 lights...

Solution 2

There are [FOURfour4]+ lights.

(FOUR|four|4)

There are FOUR lights!
There are four lights.
There are 4 lights.
There are 3 lights...

Challenge 3: AD host names

You are writing a script to assign host names to a fleet of new web servers. You want to ensure the new host names are compatible with your Active Directory environment.

Write a regular expression that will match a valid AD/NETBIOS-style host name

1 to 15 characters length
Illegal characters are \ / : * ? " < > | , .
Also, no spaces

Valid names:
webhost001
webhost002
sparkyproxydx_2

Invalid names
that_server_that_is_broken
pickle server
batman2000?
sadserver:-(

Challenge 3: AD host names

^[^\\/:*\?"<>\|,\.]{1,15}$

Valid names:
webhost001
webhost002
sparkyproxydx_2

Invalid names
that_server_that_is_broken
pickle server
batman2000?
sadserver:-(

Commands and shell concepts

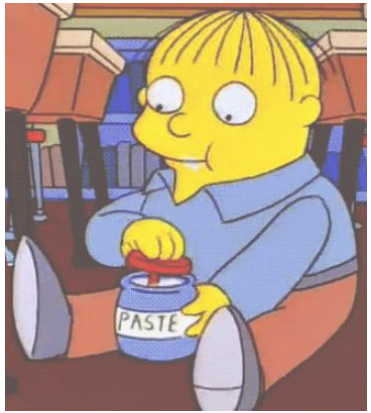
Cut

- Selects columns of data from a file
  - The file must be formatted with a delimiter
    - Example; CSV
- These columns are evaluated line-by-line
- The default delimiter is \t (tab)
- Most common usage;
  - cut -d”<delimiter>” -f<field expression>
  - <delimiter>
    - The character that separates the columns
  - <field expression>
    - a,b a,b,c a,c n-m

Cut Man
“Mega Man”
Capcom - 1987

Paste

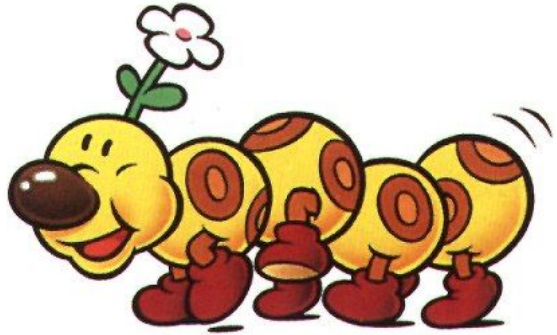
- Works line-by-line to combine multiple files into sequential fields
- If **cat** works vertically; **paste** works horizontally
- common syntax;
  - paste -d<delimiter> [file1] [file2] [fileN] …
    - <delimiter>
      - character to use for separating fields
      - default is \t (tab)
- Can be used in conjunction with cut to reorder fields in a file



Ralph Wiggum
“The Simpsons”

Head and Tail

- Outputs lines relative to the head or tail of a file
- Examples;
  - head file1
    - first 10 lines of file1
  - tail file2
    - last 10 lines of file2
  - head -n -5
    - everything except first 5 lines
  - tail -n -10
    - Everything except last 10 lines



Wiggler
“Super Mario World”
Nintendo - 1990

Uniq

- By default; drops consecutive repeated lines
- common options
  - -c print a count of the number of duplicated lines
  - -u print only unique lines
    - This differs from the default behavior, which shows 1 instance of duplicated lines
  - -d print only duplicate lines; one for each set
- Typically used in combination with sort to make duplicated lines consecutive

Sort

- A tool for sorting lines of a file
- By default, sorts in ASCII order (0=48, A=65, a=97)
- usage;
  - sort -f<delimiter> -k<key> <files>
    - -t specifies a field delimiter; default is blank to non-blank transition
    - -k field number to use for sort
      - optionally, specify a character in the field; **-k1.n,2.n**
    - -u similar to uniq
    - -r reverse sort order
    - -f treats lowercase letters as uppercase (fold)
      - ASCII a=97 becomes a=65
    - -n expect numerical data, inclding negatifes and floating points

tr - translate

- Replaces a collection of characters with
  - usage:
    - tr <set1> <set2>
      - set1 is the set of characters to be found
      - set2 is the set of characters to replace set1
      - -s collapses series of characters matched in set 1 to a single character in set2
      -

nl - number lines

- Adds line numbers as it prints to stdout
- common usage;
  - nl <files>
    - <files> is any number of input files
    - -v starting line number
    - -i increment line numbers (default 1)

Sed - Stream EDitor

Sed

- Non-interactive text editor
  - Performs text editor style actions using commands
  - Originated during a time when computer ‘displays’ were physical printers
    - We still use the term ‘print’ today when referring to console output!
  - Allows editing a file to be scripted
- syntax

  sed [options] “[addresses] [action] [arguments]” [files]

Addresses

- sed [options] “[addresses] [action] [arguments]” [files]
- The address portion tells sed what lines to act on
  - the address can be a number, range of numbers and/or a regular expression
- Lines outside an address are passed to the output unmodified
  - suppress these with the **-n** option
- Examples
  - single line; 1
  - range; 1,10
  - regular expression; /^[0-8]/
  - RegEx range; /^[2-3]/,/^6/
    - Starts at first matching line, ends at last matching line
  - Last line; $

Common commands

- sed [options] “[addresses] [action] [arguments]” [files]

Command	Description	Example
d	delete	sed ‘1,5d’ filename
p	print	sed -n ‘10p’ filename
r	read	echo ‘This is the new first line!’ | sed ‘r myfile’ > myfile
s	substitute	sed -n ‘1,3 s/[0-9]\+/REDACTED/?’ ? can be any of number: match this number of times per line g: match all (globally) p: print the resulting line w: write to a file

Substitution (s command)

- Simple substitution
  - echo ‘Bob Builder 123-45-6789’ | sed 's/[0-9]\+-[0-9]\+-[0-9]\+/REDACTED/'
    - Bob Builder REDACTED
- Single back reference
  - echo -e "1 apple\n33 apple" | sed 's/[0-9]\{2,\} [a-z]\+/&s/'
    - & uses matched string in replacement
- capture groups (back references)
  - echo 'red car' | sed 's/\([a-z]\+\) \([a-z]\+\)/The \2 is the color \1/'
    - input = red car
    - output = The car is the color red

Getting crazy: multiple commands

- On the command line
  - use the -e option to specify multiple commands
    - sed -e ‘command1’ -e ‘command2’ infile
- From a file
  - you can specify 1 sed command on each line in a file
  - provide that file with the **-f** option
  - Example;
    - sed -f myScript.sed infile

Input and Output

- Sed operates line-by-line
  - Reads a line of the file
  - processes the provided command
  - prints the line
- Default output: standard out
  - can be piped to another command, or redirected to a file
  - cat infile | sed ‘command’ > outFile
- In-place output
  - When a file is specified, sed will replace that file with its own output on successful completion of the provided commands
  - sed -i ‘command’ infile