COMP 3225

Natural Language Processing

3. Regular Expressions

Les Carr

lac@soton.ac.uk

University of Southampton

Tartan? It's a regular pattern with repeats and skips. Don't judge me.

What

- A regular expression is a language for describing strings of symbols
  - in this case, individual characters
- Embedded in search functions
  - grep-style command line functions
  - word processor “find” dialogues
  - programming language string processing functions
- Also, convenient for specifying pattern-based parsing
  - e.g. lexical token processing for NLP

Basic Regexps

RE	Example Patterns Matched
/woodchucks/	"interesting links to woodchucks and lemurs"
/a/	"Mary Ann stopped by Mona's"
/!/	"You've left the burglar behind again!" said Nori

Figure 2.1 Some simple regex searches.

- Most characters match themselves
- Sequences of regexps match sequences of characters

- Brackets indicate a set of possible single-character matches

RE	Match	Example Patterns
/[wW]oodchuck/	Woodchuck or woodchuck	"Woodchuck"
/[abc]/	'a', 'b', *or* 'c'	"In uomini, in soldati"
/[1234567890]/	any digit	"plenty of 7 to 5"

Figure 2.2 The use of the brackets [] to specify a disjunction of characters.

RE	Match	Example Patterns Matched
/[A-Z]/	an upper case letter	"we should call it 'Drenched Blossoms' "
/[a-z]/	a lower case letter	"my beans were impatient to be hoed!"
/[0-9]/	a single digit	"Chapter 1: Down the Rabbit Hole"

Figure 2.3 The use of the brackets [] plus the dash - to specify a range.

- A dash (hypen) inside a bracket implies a character range

Basic Regexps

- A caret (as the first character of the regexp) complements the set of characters in the range

RE	Match (single characters)	Example Patterns Matched
/[^A-Z]/	not an upper case letter	"Oyfn pripetchik"
/[^Ss]/	neither 'S' nor 's'	"I have no exquisite reason for't"
/[^.]/	not a period	"our resident Djinn"
/[e^]/	either 'e' or '^'	"look up ^ now"
/a^b/	the pattern 'a^b'	"look up a^ b now"

Figure 2.4 The caret ^ for negation or just to mean ^. See below re: the backslash for escaping the period.

- Named ranges use backslash notation for common patterns: digits, alphanumerics and whitespace

RE	Expansion	Match	First Matches
\d	[0-9]	any digit	Party␣of␣5
\D	[^0-9]	any non-digit	Blue␣moon
\w	[a-zA-Z0-9_]	any alphanumeric/underscore	Daiyu
\W	[^\w]	a non-alphanumeric	!!!!
\s	[␣\r\t\n\f]	whitespace (space, tab)	
\S	[^\s]	Non-whitespace	in␣Concord

Figure 2.8 Aliases for common sets of characters.

- A dot matches any character (wildcard)

RE	Match	Example Matches
/beg.n/	any character between *beg* and *n*	begin, beg'n, begun

Figure 2.6 The use of the period . to specify any character.

- Matches can be forced to be at the beginning or end of a word or line

RE	Match
^	start of line
\$	end of line
\b	word boundary
\B	non-word boundary

Figure 2.7 Anchors in regular expressions.

Repetition and Option

RE	Match	Example Patterns Matched
/woodchucks?/	woodchuck or woodchucks	"woodchuck"
/colou?r/	color or colour	"color"

Figure 2.5 The question mark ? marks optionality of the previous expression.

- A question mark indicates optionality

Kleene * **matches zero or more occurrences of previous expression**

/Hooray!*/ matches Hooray! or Hooray!! or Hooray!!! or Hooray!!!! …
(but also Hooray with no !)

/[0-9]*/ matches integers…. like 5 or 67 or 892 or 16763298450
(but also null string because it matches 0 or more digits so /Hoo[0-9]*ray/ matches Hooray)

/[0-9][0-9]*/ matches a single digit *and then* 0 or more digits

Kleene + **matches one or more occurrences**

/[0-9]+/ matches a sequence of at least 1 digit

- Asterisk and plus indicate repetition of the previous element (0+ times vs 1+ times)

{n} **matches n occurrences of previous expression**

/Hooray!{3}/ matches Hooray!!!

{n,m} **matches n to m occurrences of previous expression**

/Hooray!{1,3}/ matches Hooray! or Hooray!! or Hooray!!! or Hooray!!!

- Braces put upper and lower bounds on repetition (either can be blank)

Greedy vs non-greedy repetition

- Both * and + are greedy in that they match as much text as possible
  - Hello there! It's nice to see you! Goodbye! OK.
  - /[A-Z].*! / matches a capital letter *and all the characters up to* the end of a sentence
  - But there are three choices of sentence ending exclamation mark here, and * will consume as many characters as possible in its match
    - Hello there! It's nice to see you! Goodbye!
  - /[A-Z].*?! / is nongreedy repetition that uses the *? operator instead
    - Hello there!

Alternatives and Groups

- The infix pipe operator | indicates alternatives between matched elements.
  - /cat|dog/
    - matches cat or dog
  - /I am a (cat|dog) person/
    - matches I am a cat person or I am a dog person
  - /I am a cat|dog person/
    - Matches either I am a cat or dog person
- The pipe operator has parsing priority lower than sequence
  - the () parenthesis operator groups a single element
  - Useful to extend scope of the * and + and ? operators

Example of integer parsing with groups

- \d+
  - 1236598710745672
  - An unlimited run of digits

- [\d,]+
  - 1,234,567,890 *but also* 1234,,56
  - An unlimited run of digits interspersed with commas

- \d{1,3},(\d{3},)*\d{3}
  - 12,345,678,901
  - (Either 1,2 or 3 digits followed by a comma), followed by (zero or more groups-of-three-digits-with-a-comma-at-the-end), followed by (a group-of-three-digits-without-a-comma)
    - but will match a correct subsequence of an incorrect string ,1,234,56711
    - and won't match a number less than 1,000
    - *Left as an exercise for the reader*

Escaping Special Meaning

RE	Match	First Patterns Matched
*	an asterisk "*"	"K*A*P*L*A*N"
\.	a period "."	"Dr. Livingston, I presume"
\?	a question mark	"Why don't they come and lend a hand?"
\n	a newline	
\t	a tab	

Figure 2.10 Some characters that need to be backslashed.

- To recover the normal meaning of a special character, precede it with a backslash
  - That includes a backslash

- But, in some implementations, the special meaning of a character is SWITCHED ON by the backslash
  - Read the Manual *e.g.* sed & vim use \(\) for capture groups (see next slide)

Capture Groups

- Using parentheses for grouping also captures the substring matched to a numbered register
  - /I am a (\w+) person, do you like \1s\?/
  - matches I am a cat person, do you like cats?
  - Or I am a dog person, do you like dogs?
  - but not I am a cat person, do you like dogs?
- NB If you want to specifically forbid a group's capture, use the non-capturing group indicator ?: after the opening parenthesis

Lookaround Assertions

- A lookaround assertion is a pattern that successfully matches but does not consume the matched text
  - Lookahead checks that a pattern matches in front of the current position
  - Lookbehind checks behind the current position
  - Negative look(ahead|behind) checks the pattern DOESN'T match

(?=foo)	Lookahead for foo
(?<=foo)	Lookbehind for foo
(?!foo)	Negative Lookahead for foo
(?<!foo)	Negative Lookbehind for foo

Lookahead example

- Check for multiple constraints on a password
  - Between 8 and 16 letters
    - ^\w{6,10}$
  - Contain at least one digit
    - \D*\d

You could use .*\d but don't forget that * is greedy

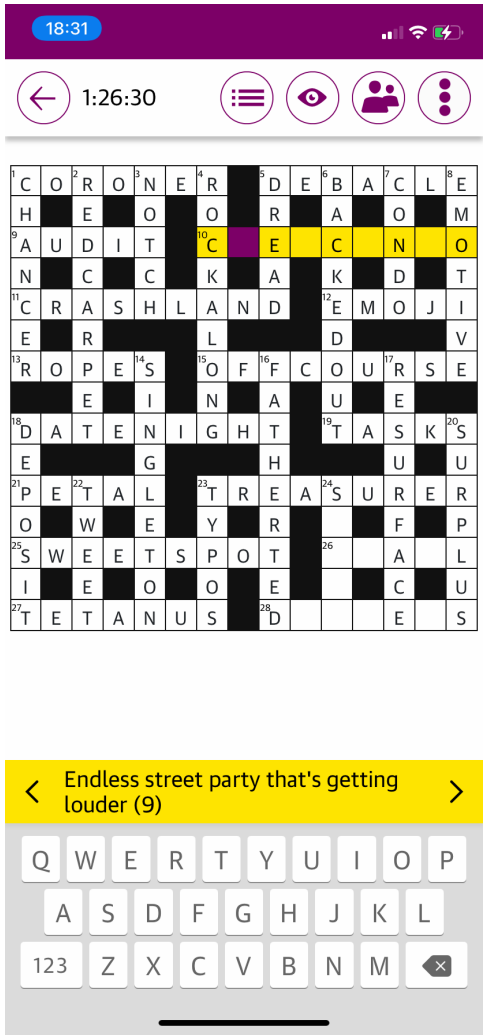
  - and three upper case letters
    - (?:[^A-Z]*[A-Z]){3}
- And then match the whole password
    - .*

(?=^\w{8,16}$)(?=\D*\d)(?=(?:[^A-Z]*[A-Z]){3}).*

Break: Solve This Crossword Clue

- Using
  - regular expressions
  - the UNIX dictionary
    - /usr/share/dict/words

- What regular expression do you need to use?

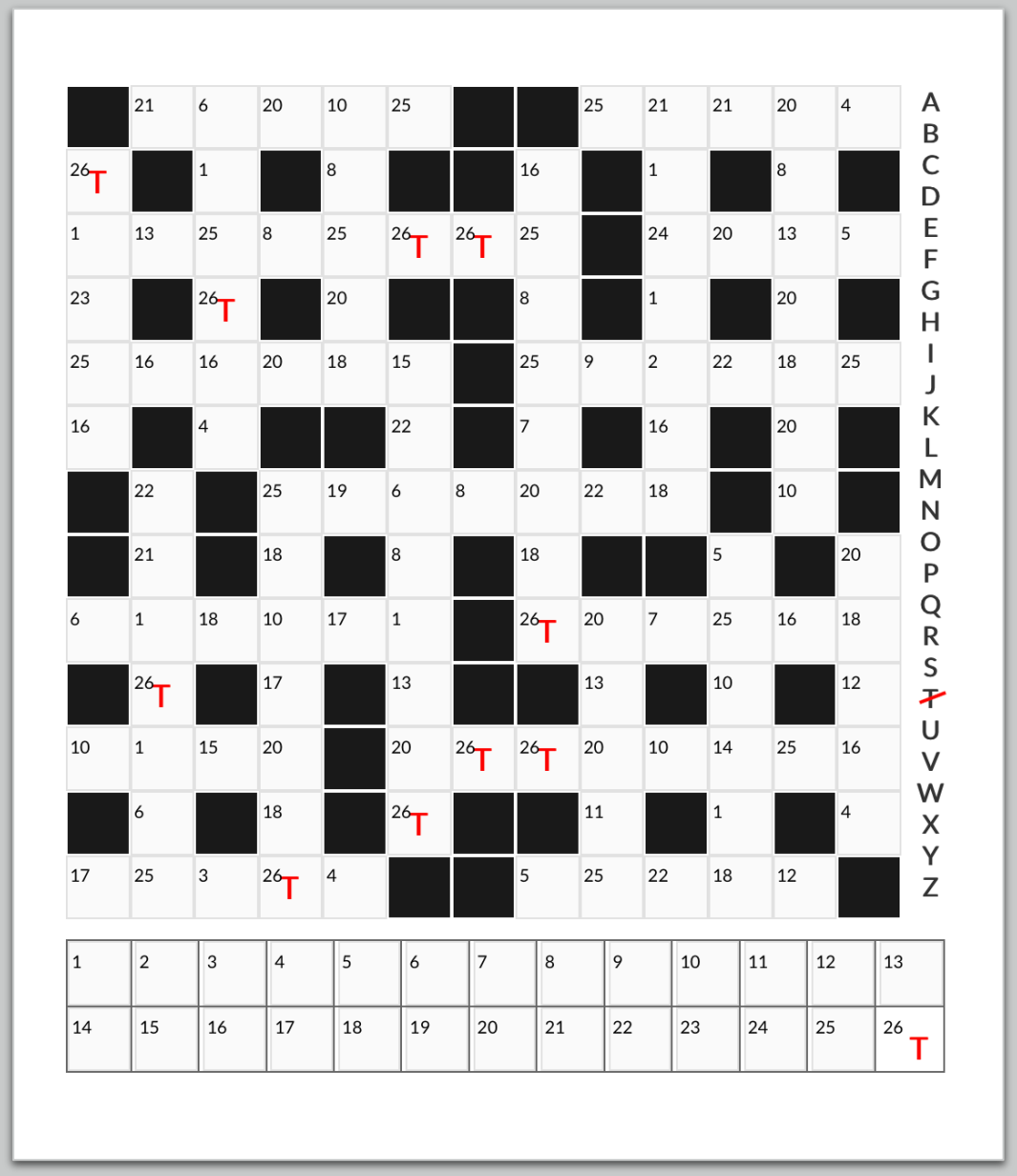


Solve This Enigma Puzzle

- Using
  - regular expressions
  - the UNIX dictionary

- The unknown letters before and after the double tt are the same as the third letter.

- What regular expression do you need to use?



History of Regexp

- Started as grammar for computer language parsing
- Kleene
- Wildcards
- Globbing
- ed / sed early UNIX
- Extended regexps, perl, TCL
  - Complexity of interpretation

How to do regexps in BASH

- grep –P ’#\w+’ filename
  - print lines that contain a match for a regexp
  - -E => extended regexps
  - -o => just print matching text
  - --color => print whole line but put matching text in a color
- sed –e ‘s/#\w+/_hashtag_/’ filename
  - stream editor – substitute matches with replacement text
  - s/find_this/replacement/

Note: GNU grep distinguishes between basic regexp, extended regexps (-E) and Perl-style regexps (-P)!

Note: filename globbing is not regexp!
the commandline `ls *-v?.txt`
is interpreted as `.*-v.\.txt`

How to do regexps in JavaScript

- Javascript
  - A regular expression is a bespoke object
    - A regexp literal looks like `/^[A-Z].*/`
  - String methods to use regexps
    - `“hello #awkward”.search(/#\w+/) == 6`
    - `“hello #awkward”.replace(/#\w+/,”_”) == “hello _”`

How to Do Regexps in Python (1)

```
>>> text = 'That U.S.A. poster-print costs $12.40...'
>>> pattern = r'''(?x)    # set flag to allow verbose regexps
...     ([A-Z]\.)+        # abbreviations, e.g. U.S.A.
...   | \w+(-\w+)*        # words with optional internal hyphens
...   | \$?\d+(\.\d+)?%?  # currency and percentages, e.g. $12.40, 82%
...   | \.\.\.            # ellipsis
...   | [][.,;"'?():-_`]  # these are separate tokens; includes ], [
... '''
>>> nltk.regexp_tokenize(text, pattern)
['That', 'U.S.A.', 'poster-print', 'costs', '$12.40', '...']
```

Figure 2.12 A Python trace of regular expression tokenization in the NLTK Python-based natural language processing toolkit (Bird et al., 2009), commented for readability; the `(?x)` verbose flag tells Python to strip comments and whitespace. Figure from Chapter 3 of Bird et al. (2009).

Regular Expression in Python (2)

- Regular expressions are in the ‘re’ package.
- Notation for patterns is slightly different from other languages – using **raw string** as an alternative to regular string if complex escaping is needed.

Regular String	Raw string
"ab*"	r"ab*"
"\\\\section"	r"\\section"
"\\w+\\s+\\1"	r"\w+\s+\1"

```
import re
string = 'hello 12 hi 89. Howdy 34'
pattern = '\d+'
result = re.findall(pattern, string)
print(result)
# Output: ['12', '89', '34']
new_string = re.sub(pattern, 'N', string)
# Output: hello N hi N. Howdy N
```

Regular Expression in Python (2)

- Matching a re object against a string is done in several ways.

Method/Attribute	Purpose
match()	Determine if the RE matches at the **beginning** of the string.
search()	Scan through a string, looking for any location where this RE matches.
findall()	Find all substrings where the RE matches, and returns them as a list.
finditer()	Find all substrings where the RE matches, and returns them as an *iterator*.

Advice: When to Use Regexps

- They're an excellent tool, and when **used appropriately** can save you a lot of time and effort.
  - Moreover, a good implementation used carefully should not be particularly CPU-intensive.
- They are compact and efficient representations of grammar rules

Limitations of Pattern Matching

- Any IR task has two kinds of errors:
  - false positives, strings that we incorrectly matched
  - false negatives, strings that we incorrectly missed
- Reducing the overall error rate for an application thus involves two antagonistic efforts:
  - Increasing precision (minimizing false positives)
  - Increasing recall (minimizing false negatives)
- Remember the law of diminishing returns, the 80:20 rule, the Pareto principle

Advice: When Not to Use Regexps

- **When they are too much of a good thing**
  - They can easily mushroom in complexity as you chase the last 5% of precision
- **When there are parsers.**
  - Beware HTML, XML and CSV.
  - A simple valid XML cannot be reasonably parsed with a regular expression, even if you know the schema and you know it will never change.
- **When you have better tools to do your job.**
  - What if you must search for a letter, both small and capital? If you love regular expressions, you'll use them. But isn't it easier/faster/readable to use two searches, one after another? Chances are in most languages you'll achieve better performance and make your code more readable.
- **When parsing human writing.**
  - A good example is an obscenity filter. There are plenty of ways an human can write a word, a number, a sentence and will be understood by another human, but not your regular expression. So instead of catching real obscenity, your regular expression will spend her time hurting other users.
- **When validating some types of data.**
  - For example, don't validate an e-mail address through a regular expression. In most cases, you'll do it wrong. In a rare case, you'll do it right and finish with a 6 343 characters length coding horror.
  - https://blog.codinghorror.com/regex-use-vs-regex-abuse/
- **When your code will be read. And then read again, and again and again, every time by different developers.**
  - Seriously, if I take your code and must review it or modify it, I don't want to spend a week trying to understand a twenty lines long string plenty of symb

End of Lecture Questions

- Panopto Quiz - 1 minute brainstorm for interactive questions

Please spend **1 minute** using Panopto quiz to wrie down two or three questions that you would like to have answered at the next interactive session.

Do it **right now** while its fresh.

Take a screen shot of your questions and **bring them with you** at the interactive session so you have something to ask.