1 ICL/Regular Expressions/2005-10-12

2 Overview of REs

2.1 Introduction

Overview
Goals:

- a basic idea of the formal background for REs
- an ability to write small Python programs that do useful things with REs

Motivation

Task: To search for strings using (partially specified) patterns

Why:

- validate data fields (dates, email addresses, URLs)
- filter text (spam, disallowed web sites)
- identify particular strings in a text (token boundaries for tokenization)
- convert the output of one processing component into the format required for a second component (`rabbit_NN` → `<word pos=''NN''>rabbit</word>`)

The Basic Idea

- Regular expressions form a language for expressing patterns.
- The language can be stated as a formal algebra.
- Recognizers for RE can be efficiently implemented.
- 'Regular expression' also a term for a pattern that is constructed using the language.
- Every pattern specifies a set of strings.
- Text string: a sequence of letters, numerals, spaces, tabs, punctuation, . . .

Initital Examples

	Pattern	Matches
concatenation	**abc**	abc
disjunction	**a** | **b**	a, b
	(**a** | **bb**) **d**	ad, bbd
closure	**a***	ϵ, a, aa, aaa, aaaa, . . .
	c(**a** | **bb**)*	c, ca, cbb, cabb, caa, cbbbb, . . .

Two Types of RE

Literals Every normal text character is an RE, and denotes itself.

Metacharacters Special characters which allow you to specify various sets of strings.

Example—Kleene star (*)

- **a** denotes *a*
- **a*** denotes ϵ (empty string), *a*, *aa*, *aaa*, . . .

2.2 Formal Background to REs

Preliminaries: Operations on Sets of Strings
Let Σ be a finite set of symbols and let Σ^* be the set of all strings (including the empty string) over Σ. Suppose L, L_1, L_2 are subsets of Σ^*.

- The *union* of L_1, L_2, denoted $L_1 \cup L_2$, is the set of strings x such that $x \in L_1$ or $x \in L_2$.
- The *concatenation* of L_1, L_2, denoted L_1L_2, is the set of strings xy such that $x \in L_1$ and $y \in L_2$.
- The *Kleene closure* of L, denoted L^*, is the set of strings constructed by concatenating any number of strings from L. L^* contains ϵ, the empty string.
- The *positive closure* of L, denoted L^+, is the same as L^* but without ϵ.

Examples
Let $L_1 = \{a, b\}$ and $L_2 = \{c\}$. Then

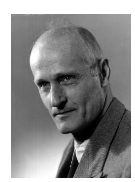
- $L_1 \cup L_2 = \{a, b, c\}$
- $L_1L_2 = \{ac, bc\}$
- $\{a, b\}^* = \{\epsilon, a, b, aa, bb, ab, ba, \ldots\}$
- $\{a, b\}^+ = \{a, b, aa, bb, ab, ba, \ldots\}$

Formal Definition of Regular Expressions
Regular expressions over a finite alphabet Σ:

1. ϵ is a regular expression and denotes the set $\{\epsilon\}$.
2. For each a in Σ, a is a regular expression and denotes the set $\boldsymbol{\{a\}}$.
3. If $\boldsymbol{r}$ and $\boldsymbol{s}$ are regular expressions denoting the sets $\boldsymbol{R}$ and $\boldsymbol{S}$ respectively, then
   - $\boldsymbol{(r \mid s)}$ is a regular expression denoting $\boldsymbol{R} \cup \boldsymbol{S}$.
   - $\boldsymbol{(rs)}$ is a regular expression denoting $\boldsymbol{RS}$.
   - $\boldsymbol{(r^*)}$ is a regular expression denoting $\boldsymbol{R^*}$.

Recognizers

- A recognizer for a language is a program that takes as input a string x and answers "yes" if x is a sentence of the language and "no" otherwise.
- We can think of this program as a machine which only emits two possible responses to its input.



Finite State Automata

- A Finite State Automaton (FSA) is an abstract finite machine.
- Regular expressions can be viewed as a way to describe a Finite State Automaton (FSA)

- Kleene's theorem (1956): FSA and RE describe the same languages:
  - Any regular expression can be implemented as an FSA.
  - Any FSA can be described by a regular expression.
- Regular languages are those that can be recognized by FSAs (or characterized by a regular expression).

2.3 Extensions of Basic REs

Metacharacters
NB. Different sets of metacharacters and notations used by different 'host languages' (e.g., Unix grep, GNU emacs, Perl, Java, Python, etc.). Cf. Jurafsky & Martin, Appendix A)

Disjunction: |

Wild card: .

Optionality: ?

Quantification: * and +

Choice: [Mm] [**0123456789**]

Ranges: [**a-z**] [**0-9**]

Negation: [^**Mm**] (only when '^' occurs immediately after '[')

Special Backslash Sequences

- Standard escape sequences
  `\t`: tab
  `\n`: newline
- Abbreviatory forms

`\d`: digit (i.e., numeral)	`\D`: non-digit
`\s`: 'whitespace' ([`\t\n`])	`\S`: non-whitespace
`\w`: 'alphanumeric' ([a-zA-Z0-9])	`\W`: non-alphanumeric

- `\` is a general escape character; e.g., `\.` is not a wildcard, but matches a period, `.`
- If you want to use `\` in a string, it has to be escaped: `\\`

Anchors
(Also: zero-width characters)

- Anchors don't match strings in the text, instead
- they match positions in the text.
  `^`: matches beginning of line (or text)
  `$`: matches end of line (or text)
  `\b`: matches word boundary (i.e., a location with `\w` on one side but not the other)

3 REs in Python

3.1 Examples with re_show

Wildcard

```
>>> from nltk_lite.utilities import re_show
>>> s = '''BP has agreed to sell
... it's petrochemicals unit for $5.1bn.'''
>>> re_show('...', s)
{BP }{has}{ ag}{ree}{d t}{o s}{ell}
{it'}{s p}{etr}{och}{emi}{cal}{s u}{nit}{ fo}{r $}{5.1}{bn.}
```

```
>>> re_show('.a..', s)
BP {has }agreed to sell
it's petrochemi{cals} unit for $5.1bn.
```

Wildcards with Quantifiers

```
>>> re_show('s.*l', s)
BP ha{s agreed to sell}
it'{s petrochemical}s unit for $5.1bn.
```

```
>>> re_show('B.*P', s)
{BP} has agreed to sell
it's petrochemicals unit for $5.1bn.
```

```
>>> re_show('B.+P', s)
BP has agreed to sell
it's petrochemicals unit for $5.1bn.
```

Disjunction

```
>>> re_show('has|it', s)
BP {has} agreed to sell
{it}'s petrochemicals un{it} for $5.1bn.
```

```
>>> re_show('has | it', s)
BP {has }agreed to sell
it's petrochemicals unit for $5.1bn.
```

```
>>> re_show('(e|l)+', s)
BP has agr{ee}d to s{ell}
it's p{e}troch{e}mica{l}s unit for $5.1bn.
```

Zero Width Characters

```
>>> re_show('l', s)
BP has agreed to se{l}{l}
it's petrochemica{l}s unit for $5.1bn.
```

```
>>> re_show('l$', s)
BP has agreed to sel{l}
it's petrochemicals unit for $5.1bn.
```

```
>>> re_show('i', s)
BP has agreed to sell
{i}t's petrochem{i}cals un{i}t for $5.1bn.
```

```
>>> re_show('^i', s)
BP has agreed to sell
{i}t's petrochemicals unit for $5.1bn.
```

Escaping Special Characters

```
>>> re_show('.', s)
{B}{P}{ }{h}{a}{s}{ }{a}{g}{r}{e}{e}{d}...
```

```
>>> re_show('\.', s)
BP has agreed to sell
it's petrochemicals unit for $5{.}1bn{.}
```

```
>>> re_show('$', s)
BP has agreed to sell{}
it's petrochemicals unit for $5.1bn.{}
```

```
>>> re_show('\$', s)
BP has agreed to sell
it's petrochemicals unit for {$}5.1bn.
```

Metacharacters and Negated Ranges

```
>>> re_show('\w',s)
{B}{P} {h}{a}{s} {a}{g}{r}{e}{e}{d} ...
```

```
>>> re_show('\d',s)
BP has agreed to sell
it's petrochemicals unit for ${5}.{1}bn.
```

```
>>> re_show('[^a-z\s]',s)
{B}{P} has agreed to sell
it{'}s petrochemicals unit for {$}{5}{.}{1}bn{.}
```

```
>>> re_show('[^\w]',s)
BP{ }has{ }agreed{ }to{ }sell{
}it{'}s{ }petrochemicals{ }unit{ }for{ }{$}5{.}1bn{.}
```

3.2 Match objects in Python

Using REs in Python, 1

- Usually best to compile the RE into a PatternObject; more efficient, and it can be re-used.

  ```
  >>> import re
  >>> str = 'do you say hello or hullo?'
  >>> helloRE = re.compile('h[eu]llo')
  ```

- The resulting PatternObject has a number of methods:

findall(s): returns a list of all matches of pattern in string `s`

search(s): searches for leftmost occurrence of pattern in string `s`

match(s): tries to match pattern at the beginning of string `s`

Using REs in Python, 2

- The PatternObject method `findall` returns a list:

```
>>> helloRE.findall(str)
['hello', 'hullo']
```

- The PatternObject method `search` (and `match`) returns a MatchObject or None.
- A MatchObject has a variety of methods, but is not a string.

```
>>> m = helloRE.search(str)
>>> m
<_sre.SRE_Match object at 0x47b138>
>>> m.group() # return matched substring (sort of!)
'hello'
>>> m.end() # index of end of target
16
```

Groups

- Groups in regular expressions are captured using parentheses.

```
>>> import re
>>> str = 'do you say hello or hullo?'
>>> reGRP = re.compile('(d.)(.*)(e..)')
>>> m = reGRP.search(str)
>>> m
<_sre.SRE_Match object at 0x64390>
>>> m.groups()
('do', ' you say h', 'ell')
```

Named Groups

- Name groups captured using `(?P<name>)`:

```
FROM = re.compile("""
     ^From:          # Anchor to start of line
     \s*             # maybe some spaces
     (?P<user>\w+)   # 'user': group of word characters
     @
     (?P<domain>     # the 'domain':
     \S+)            # some non-space characters
     \s              # finally, a space character
     """,re.VERBOSE)
```

Named Groups (cont.)

```
from nltk_lite.corpus import twenty_newsgroups

for item in twenty_newsgroups.items('misc.forsale'):
    text = twenty_newsgroups.read(item)
    m = FROM.search(text)
    if m:
      print '%s is at %s' % \
      (m.group('user'), m.group('domain'))
```

```
kedz is at bigwpi.WPI.EDU
myoakam is at cis.ohio-state.edu
gt1706a is at prism.gatech.EDU
jvinson is at xsoft.xerox.com
hungjenc is at usc.edu
thouchin is at cs.umr.edu
kssimon is at silver.ucs.indiana.edu
```

Tokenization with Regular Expressions (1)

- The method `tokenize.regexp()` takes a string and a regular expression, and returns the list of substrings that match the RE

```
>>> from nltk_lite import tokenize
>>> s = "Hello.   Isn't this fun?"
>>> pat= r'\w+|[^\w\s]+'
>>> list(tokenize.regexp(s, pat))
['Hello', '.', 'Isn', "'", 't', 'this', 'fun', '?']
```

- This is a simple tokenizer that may break up things we want to keep as a single token:

```
>>> t = "That poster from the U.S.A. costs $22.50."
>>> list(tokenize.regexp(t, pat))
['That', 'poster', 'from', 'the', 'U', '.', 'S', '.',
'A', '.', 'costs', '$', '22', '.', '50', '.']
```

Tokenization with Regular Expressions (2)

- Add further components to the RE used in the tokenizer:

```
>>> import re
>>> pat2 = re.compile(r'''
...      \$?\d+(\.\d+)? # currency amounts (eg $22.50)
...    | ([A-Z]\.)+     # abbreviations (eg U.S.A.)
...    | \w+            # sequences of 'word' characters
...    | [^\w\s]+       # punctuation sequences
... ''', re.VERBOSE)
>>> list(tokenize.regexp(t, pat2))
['That', 'poster', 'from', 'the', 'U.S.A.', 'costs',
'$22.50', '.']
```

Reading

- Jurafsky & Martin, Chap 2
- NLTK Lite Tutorial: Regular Expressions available from `http://nltk.sourceforge.net/lite/doc/en/regexps.html`