Regular Expression (regex)

Jonathan Feinberg

Dept. of Informatics, Univ. of Oslo

Simula Research Laboratory

August 2014

Regular expression (regex)

Contents

- Motivation for regular expression
- Regular expression syntax
- Lots of examples on problem solving with regular expressions
- For more information on regex in Python:
  `http://uio-inf3331.github.io/resources-14/doc/texts/pub/pyregex.html`

Digression: Editor war!

Regular expression is advanced search and replace

It is strongly ecourage to learn by experimenting in a supported editor.

- Emacs
  `lmgtfy.com/?q=emacs+tutorial`
- Vim
  ```
  $ sudo apt-get install vim-gnome # or vim-gtk or vim-common
  $ vimtutor      # 30 minutes to learn basic stuff
  ```
- Sublime
  `http://www.sublimetext.com/`
- Notepad++ (Windows)
  `http://notepad-plus-plus.org/`
- textmate (Mac)
  `http://macromates.com/`

Regex search and replace in Emacs and Vim

- Emacs search:
  `C-M-s` `M-x isearch-forward-regex<enter>`
- Emacs replace:
  `C-M-%` `M-x replace-regexp<enter>`
  `# [y]es/[n]o/all[!]/cancel[<C-g>]`
- Vim highlighting:
  `:set hls<enter>` `:set hlsearch<enter>`
- Vim search:
  `/\v`
- Vim replace:
  `:%s/\v`

Show and tell

The command line wild card

- Consider a simulation code with this type of output:

```
$ ls *.tex
report.tex
```

- Here '`*`' is a wild card and can be read as:
  "Zero or more of any non-linefeed character type".
- Simple and powerful, but some what unrefined.

Regular expression introduction

- Classical wild card split into two components:
  - '`.`': Any charracter
  - '`*`': Zero or more
- Replacate wild card with '`.*`':

```
$ grep -E ".*" unsorted_fruits
orange
pear
apple
grape
pineapple

$ grep -E ".*apple" unsorted_fruits
apple
pineapple
```

Refined search

Character		Quantifier	
`.`	Non-linefeed character	`*`	Zero or more
`\w` (`\W`)	(Not) alphabet character	`+`	One or more
`\d` (`\D`)	(Not) numerical character	`?`	Zero or one
`\s` (`\S`)	(Not) white space	`{4}`	Custom quantifier
`[A-F\d_]`	Custom character	`{2,4}`	Range quantifier
`[^abc]`	All but custom character	`{,4}`	Open ended range

Only strings ending with 'apple':

```
$ grep -E "\w+apple" unsorted_fruits
pineapple
```

Delimiters

- A text have multiple “delimiters”, i.e. structures in text which are not characters:

```
 This    is    some
^       <  >      $\n
 text   worth    reading.
^      \b   \b          $
```

- Find apples but not pineapples:

```
$ grep -E "\bapple\b" unsorted_fruits
apple
```

Inconsistent Regex

- Reserved characters that has to be “cancelled” with a ’\’ prefix in Bash/Python:

```
  ^   [   .   $   {   *   (   +   )   |   ?   <   >
 \w  \W  \d  \D  \s  \S  \b  \B
```

- Emacs’ isearch-forward-regex and replace-regex:

```
  ^   [   .   $  \{   *  \(   +  \)  \|   ?  \<  \>
 \w  \W          \s- \S- \b  \B
```

- Vim’s “magic mode”:

```
  ^  \[   .   $  \{  \*  \(  \+  \)  \|  \?  \<  \>
 \w  \W  \d  \D  \s  \S  \b  \B
```

- Vim’s “very magic mode”:

```
  ^   [   .   $   {   *   (   +   )   |   ?   <   >
 \w  \W  \d  \D  \s  \S  \b  \B
```

Regular expression in Python

- Find all words:

```
>>> import re
>>> text = "apple, apples, pineapple and appletini"
>>> print re.findall(r"\w+", text)
['apple', 'apples', 'pineapple', 'and', 'appletini']
```

- Replace words ending with “apple” with “pear”:

```
>>> re.sub(r"apple\b", "pear", text)
'pear, apples, pinepear and appletini'
```

- As always, read more in the documentation:

```
$ pydoc re
```

Extraction

- Parenthesis can be used to extract sub-strings:

```
>>> re.findall(r"\bapple(\w*)", text)
['', 's', 'tini']
```

- Multiple extraction is possible:

```
>>> re.findall(r"(\w*)apple(\w*)", text)
[('', ''), ('', 's'), ('pine', ''), ('', 'tini')]
```

- Extraction can be used in advanced substitution:

```
>>> re.sub(r"(\w*)apple(\w*)", r"\2apple\1", text)
'apple, sapple, applepine and tiniapple'
```

`\1`, `\2`, ..., `\9` represent extractions.
`\0` is the full match.

Groupings

- Paranthesis can also be used to to group alternatives through the ’|’ character.
- For example, find words shorter than 6 charracters, but also longer than 7:

```
>>> re.findall(r"(\w{8,}|\w{1,5})", text)
['apple', 'pineapple', 'appletini']
```

- Note: Precedence order from left to right:

```
>>> re.findall(r"(\w{1,5}|\w{8,})", text)
['apple', 'apple', 's', 'pinea', 'pple', 'apple', 'tini']
```

  Left clause must fail before right clause can be addressed.

Imprecise syntax leads to too much greed

- Finding all words starting with “a” and end with “s”:

```
>>> re.findall(r"a.*s", text)
['apple, apples']
```

  Ups!

- Quantifiers like '`*`' and '`+`' are greedy.
- Quantifiers can be made non-greedy by placing a '?' after them:

```
>>> re.findall(r"a.*?s", text)
['apples']
```

- Note 1: '?' after characters are still “zero or one”.
- Note 2: '?' is not supported in Vim. Use negative range instead: '`{-0,}`'

Larger example: extracting all the numbers!

```
t=2.5  a: 1.0 6.2 -2.2   12 iterations and eps=1.38756E-05
t=4.25  a: 1.0 1.4   6 iterations and eps=2.22433E-05
>> switching from method AQ4 to AQP1
t=5  a: 0.9   2 iterations and eps=3.78796E-05
t=6.386  a: 1.0 1.1525   6 iterations and eps=2.22433E-06
>> switching from method AQP1 to AQ2
t=8.05  a: 1.0   3 iterations and eps=9.11111E-04
...
```

- Different ways of writing real numbers:
  -3, 42.9873, 1.23E+1, 1.2300E+01, 1.23e+01
- Three basic forms:
  - integer: -3
  - decimal notation: 42.9873, .376, 3.
  - scientific notation: 1.23E+1, 1.2300E+01, 1.23e+01, 1e1

A simple approach

- Could just collect the legal characters in the three notations:

```
[0-9.Ee\-+]+
```

- Downside: this matches text like

```
12-24
24.-
--E1--
+++++
```

- How can we define precise regular expressions for the three notations?

Decimal notation regex

- Regex for decimal notation:

```
-?\d*\.\d+

# or in emacs:
-?[0-9]*\.[0-9]+
```

- Problem: this regex does not match '3.'
- The fix

```
-?\d*\.\d*
```

  is ok but matches text like '-.' and (much worse!) '.'
- Trying it on

```
'some text. 4. is a number.'
```

  gives a match for the first period!

Fix of decimal notation regex

- We need a digit before OR after the dot
- The fix:

```
-?(\d*\.\d+|\d+\.\d*)
```

- A more compact version (just "OR-ing" numbers without digits after the dot):

```
-?(\d*\.\d+|\d+\.)
```

Combining regular expressions

- Make a regex for integer or decimal notation:

  `(integer OR decimal notation)`

  using the OR operator and parenthesis:

  `-?(\d+|(\d+\.\d*|\d*\.\d+))`

- Problem: `22.432` gives a match for `22`
  (i.e., just digits? yes - `22` - match!)

Check the order in combinations!

- Remedy: test for the most complicated pattern first

```
(decimal notation OR integer)
```

```
-?((\d+\.\d*|\d*\.\d+)|\d+)
```

- Modularize the regex:

```
real_in = r'\d+'
real_dn = r'(\d+\.\d*|\d*\.\d+)'
real = '-?(' + real_dn + '|' + real_in + ')'
```

Scientific notation regex (1)

- Write a regex for numbers in scientific notation
- Typical text: `1.27635E+01`, `-1.27635e+1`
- Regular expression:

  ```
  -?\d\.\d+[Ee][+\-]\d\d?
  ```

- = optional minus, one digit, dot, at least one digit, E or e, plus or minus, one digit, optional digit

Scientific notation regex (2)

- Problem: `1e+00` and `1e1` are not handled
- Remedy: zero or more digits behind the dot, optional e/E, optional sign in exponent, more digits in the exponent (`1e001`):

```
-?\d\.?\d*[Ee][+\-]?\d+
```

Making the regex more compact

- A pattern for integer or decimal notation:

  ```
  -?((\d+\.\d*|\d*\.\d+)|\d+)
  ```

- Can get rid of an OR by allowing the dot and digits behind the dot be optional:

  ```
  -?(\d+(\.\d*)?|\d*\.\d+)
  ```

- Such a number, followed by an optional exponent (a la `e+02`), makes up a general real number (!)

  ```
  -?(\d+(\.\d*)?|\d*\.\d+)([eE][+\-]?\d+)?
  ```

A more readable regex

- Scientific OR decimal OR integer notation:

```
-?(\d\.?\d*[Ee][+\-]?\d+|(\d+\.\d*|\d*\.\d+)|\d+)
```

or better (modularized):

```
real_in = r'\d+'
real_dn = r'(\d+\.\d*|\d*\.\d+)'
real_sn = r'(\d\.?\d*[Ee][+\-]?\d+'
real = '-?(' + real_sn + '|' + real_dn + '|' + real_in + ')'
```

Grab the groups

- Enclose parts of a regex in () to extract the parts:

```
pattern = r"t=(.*)\s+a:.*\s+(\d+)\s+.*=(.*)"
# groups:     (  )          (   )      (  )
```

  This defines three groups (t, iterations, eps)

- In Python code:

```
matches = re.findall(pattern, line)
for match in mathces:
    time = float(match[0])
    iter = int  (match[1])
    eps  = float(match[2])
```

Pattern-matching modifiers (1)

- ...also called flags in Python regex documentation
- Check if a user has written "yes" as answer:

```
re.findall('yes', answer)
```

- Problem: "YES" is not recognized; try a fix

```
re.findall(r'(yes|YES)', answer)
```

- Should allow "Yes" and "YEs" too...

```
re.findall(r'[yY][eE][sS]', answer)
```

- This is hard to read and case-insensitive matches occur frequently - there must be a better way!

Pattern-matching modifiers (2)

```
matches =  re.findall('yes', answer,  re.IGNORECASE)
# pattern-matching modifier: re.IGNORECASE
# now we get a match for 'yes', 'YES', 'Yes' ...

# ignore case:
re.I  or  re.IGNORECASE

# let ^ and $ match at the beginning and
# end of every line:
re.M  or  re.MULTILINE

# allow comments and white space:
re.X  or  re.VERBOSE

# let . (dot) match newline too:
re.S  or  re.DOTALL

# let e.g. \w match special chars (?, ?, ...):
re.L  or  re.LOCALE
```

Comments in a regex

- The `re.X` or `re.VERBOSE` modifier is very useful for inserting comments explaning various parts of a regular expression
- Example:

```
# real number in scientific notation:
real_sn = r"""
-?               # optional minus
\d\.\d+           # a number like 1.4098
[Ee][+\-]\d\d?  # exponent, E-03, e-3, E+12
"""

match = re.search(real_sn, 'text with a=1.92E-04 ',
                  re.VERBOSE)

# or when using compile:
c = re.compile(real_sn, re.VERBOSE)
match = c.findall('text with a=1.9672E-04 ')
```

Substitution example

- Suppose you have written a C library which has many users
- One day you decide that the function

  ```
  void superLibFunc(char* method, float x)
  ```

  would be more natural to use if its arguments were swapped:

  ```
  void superLibFunc(float x, char* method)
  ```

- All users of your library must then update their application codes - can you automate?

Substitution with backreferences

- You want locate all strings on the form

  `superLibFunc(arg1, arg2)`

  and transform them to

  `superLibFunc(arg2, arg1)`

- Let `arg1` and `arg2` be groups in the regex for the superLibFunc calls
- Write out

  `superLibFunc(\2, \1)`

```
# recall: \1 is group 1, \2 is group 2 in a re.sub command
```

Regex for the function calls (1)

- Basic structure of the regex of calls:

  `superLibFunc\s*\(\s*arg1\s*,\s*arg2\s*\)`

  but what should the `arg1` and `arg2` patterns look like?
- Natural start: `arg1` and `arg2` are valid C variable names

  `arg = r"[A-Za-z_0-9]+"`
- Fix; digits are not allowed as the first character:

  `arg = "[A-Za-z_][A-Za-z_0-9]*"`

Regex for the function calls (2)

- The regex

  ```
  arg = "[A-Za-z_][A-Za-z_0-9]*"
  ```

  works well for calls with variables, but we can call `superLibFunc` with numbers too:

  ```
  superLibFunc ("relaxation", 1.432E-02);
  ```

- Possible fix:

  ```
  arg = r"[A-Za-z0-9_.\-+\"]+"
  ```

  but the disadvantage is that `arg` now also matches

  ```
  .+-32skj  3.ejks
  ```

Constructing a precise regex (1)

- Since `arg2` is a float we can make a precise regex: legal C variable name OR legal real variable format

```
arg2 = r"([A-Za-z_][A-Za-z_0-9]*|" + real + \
        "|float\s+[A-Za-z_][A-Za-z_0-9]*" + ")"
```

where `real` is our regex for formatted real numbers:

```
real_in = r"-?\d+"
real_sn = r"-?\d\.\d+[Ee][+\-]\d\d?"
real_dn = r"-?\d*\.\d+"
real = r"\s*("+ real_sn +"|"+ real_dn +"|"+ real_in +r")\s*"
```

Constructing a precise regex (2)

- We can now treat variables and numbers in calls
- Another problem: should swap arguments in a user's definition of the function:

```
void superLibFunc(char* method, float x)
```

to

```
void superLibFunc(float x, char* method)
```

Note: the argument names (`x` and `method`) can also be omitted!

- Calls and declarations of superLibFunc can be written on more than one line and with embedded C comments!
- Giving up?

A simple regex may be sufficient

- Instead of trying to make a precise regex, let us make a very simple one:

```
arg = '.+'   # any text
```

- "Any text" may be precise enough since we have the surrounding structure,

```
superLibFunc\s*(\s*arg\s*,\s*arg\s*)
```

and assume that a C compiler has checked that `arg` is a valid C code text in this context

Refining the simple regex

- A problem with `.+` appears in lines with more than one calls:

  ```
  superLibFunc(a,x);  superLibFunc(ppp,qqq);
  ```

- We get a match for the first argument equal to

  ```
  a,x);  superLibFunc(ppp
  ```

- Remedy: non-greedy regex (see later) or

  ```
  arg = r"[^,]+"
  ```

  This one matches multi-line calls/declarations, also with embedded comments (`.+` does not match newline unless the `re.S` modifier is used)

Swapping of the arguments

- Central code statements:

```
arg = r"[^,]+"
call = r"superLibFunc\s*\(\s*(%s),\s*(%s)\)" % (arg,arg)

# load file into filestr

# substutite:
filestr = re.sub(call, r"superLibFunc(\2, \1)", filestr)

# write out file again
fileobject.write(filestr)
```

Testing the code

- Test text:

```
superLibFunc(a,x);  superLibFunc(qqq,ppp);
superLibFunc ( method1, method2 );
superLibFunc(3method /* illegal name! */, method2 ) ;
superLibFunc(  _method1,method_2) ;
superLibFunc (
               method1 /* the first method we have */ ,
          super_method4 /* a special method that
                                deserves a two-line comment... */
             ) ;
```

- The simple regex successfully transforms this into

```
superLibFunc(x, a);  superLibFunc(ppp, qqq);
superLibFunc(method2 , method1);
superLibFunc(method2 , 3method /* illegal name! */) ;
superLibFunc(method_2, _method1) ;
superLibFunc(super_method4 /* a special method that
                                deserves a two-line comment... */
             , method1 /* the first method we have */ ) ;
```

- Notice how powerful a small regex can be!!
- Downside: cannot handle a function call as argument

Shortcomings

- The simple regex

  ```
  [^,]+
  ```

  breaks down for comments with comma(s) and function calls as arguments, e.g.,

  ```
  superLibFunc(m1, a /* large, random number */);
  superLibFunc(m1, generate(c, q2));
  ```

  The regex will match the longest possible string ending with a comma, in the first line

  ```
  m1, a /* large,
  ```

  but then there are no more commas ...

- A complete solution should *parse* the C code

More easy-to-read regex

- The `superLibFunc` call with comments and named groups:

```
call = re.compile(r"""
     superLibFunc  # name of function to match
     \s*           # possible whitespace
     \(            # parenthesis before argument list
     \s*           # possible whitespace
     (?P<arg1>%s)  # first argument plus optional whitespace
     ,             # comma between the arguments
     \s*           # possible whitespace
     (?P<arg2>%s)  # second argument plus optional whitespace
     \)            # closing parenthesis
     """ % (arg,arg), re.VERBOSE)

# the substitution command:
filestr = call.sub(r"superLibFunc(\g<arg2>,
                   \g<arg1>)",filestr)
```

Example

- Goal: remove C++/Java comments from source codes
- Load a source code file into a string:

```
filestr = open(somefile, 'r').read()
# note: newlines are a part of filestr
```

- Substitute comments *// some text...* by an empty string:

```
filestr = re.sub(r'//.*', '', filestr)
```

- Note: . (dot) does not match newline; if it did, we would need to say

```
filestr = re.sub(r'//[^\n]*', '', filestr)
```

Failure of a simple regex

- How will the substitution

```
filestr = re.sub(r'//[^\n]*', '', filestr)
```

  treat a line like

```
const char* heading = "------------//------------";
```

  ???