Slides from INF3331 lectures - Regular expressions

Ola Skavhaug, Joakim Sundnes and Hans Petter Langtangen

Dept. of Informatics, Univ. of Oslo

&

Simula Research Laboratory

September 2012

Regular expressions

Contents

- Motivation for regular expression
- Regular expression syntax
- Lots of examples on problem solving with regular expressions
- Many examples related to scientific computations

More info

- Ch. 8.2 in the course book
- Regular Expression HOWTO for Python (see `doc.html`)
- perldoc perlrequick (intro), perldoc perlretut (tutorial), perldoc perlre (full reference)
- “Text Processing in Python” by Mertz (Python syntax)
- “Mastering Regular Expressions” by Friedl (Perl syntax)
- Note: the core syntax is the same in Perl, Python, Ruby, Tcl, Egrep, Vi/Vim, Emacs, ..., so books about these tools also provide info on regular expressions

Motivation

- Consider a simulation code with this type of output:

```
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
...
```

- You want to make two graphs:
  - iterations vs t
  - eps vs t
- How can you extract the relevant numbers from the text?

Regular expressions

- Some structure in the text, but `line.split()` is too simple (different no of columns/words in each line)
- Regular expressions constitute a powerful language for formulating structure and extract parts of a text
- Regular expressions look cryptic for the novice
- regex/regexp: abbreviations for regular expression

Specifying structure in a text

```
t=6.386  a: 1.0 1.1525   6 iterations and eps=2.22433E-06
```

- Structure: t=, number, 2 blanks, a:, some numbers, 3 blanks, integer, ' iterations and eps=', number
- Regular expressions constitute a language for specifying such structures
- Formulation in terms of a regular expression:

```
t=(.*)\s{2}a:.*\s+(\d+) iterations and eps=(.*)
```

Dissection of the regex

- A regex usually contains special characters introducing freedom in the text:

```
t=(.*)\s{2}a:.*\s+(\d+) iterations and eps=(.*)
```

```
t=6.386  a: 1.0 1.1525   6 iterations and eps=2.22433E-06
```

```
.           any character
.*          zero or more . (i.e. any sequence of characters)
(.*)        can extract the match for .* afterwards
\s          whitespace (spacebar, newline, tab)
\s{2}       two whitespace characters
a:          exact text
.*          arbitrary text
\s+         one or more whitespace characters
\d+         one or more digits (i.e. an integer)
(\d+)       can extract the integer later
iterations and eps=     exact text
```

Using the regex in Python code

```
pattern = \
r"t=(.*)\s{2}a:.*\s+(\d+) iterations and eps=(.*)"

t = []; iterations = []; eps = []

# the output to be processed is stored in the list of lines

for line in lines:

    match = re.search(pattern, line)

    if match:
        t.append          (float(match.group(1)))
        iterations.append(int   (match.group(2)))
        eps.append        (float(match.group(3)))
```

Result

- Output text to be interpreted:

```
t=2.5  a: 1 6 -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 1.15   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
```

- Extracted Python lists:

```
t = [2.5, 4.25, 5.0, 6.386, 8.05]
iterations = [12, 6, 2, 6, 3]
eps = [1.38756e-05, 2.22433e-05, 3.78796e-05,
       2.22433e-06, 9.11111E-04]
```

Another regex that works

- Consider the regex

  ```
  t=(.*)\s+a:.*\s+(\d+)\s+.*=(.*)
  ```

  compared with the previous regex

  ```
  t=(.*)\s{2}a:.*\s+(\d+) iterations and eps=(.*)
  ```

- Less structure
- How 'exact' does a regex need to be?
- The degree of preciseness depends on the probability of making a wrong match

Failure of a regex

- Suppose we change the regular expression to

```
t=(.*)\s+a:.*(\d+).*=(.*)
```

- It works on most lines in our test text but not on

```
t=2.5  a: 1 6 -2   12 iterations and eps=1.38756E-05
```

- 2 instead of 12 (iterations) is extracted
  (why? see later)
- Regular expressions constitute a powerful tool, but you need to develop understanding and experience

List of special regex characters

```
.         # any single character except a newline
^         # the beginning of the line or string
$         # the end of the line or string
*         # zero or more of the last character
+         # one or more of the last character
?         # zero or one of the last character

[A-Z]     # matches all upper case letters
[abc]     # matches either a or b or c
[^b]      # does not match b
[^a-z]    # does not match lower case letters
```

Context is important

```
.*     # any sequence of characters (except newline)
[.*]   # the characters . and *

^no    # the string 'no' at the beginning of a line
[^no]  # neither n nor o

A-Z    # the 3-character string 'A-Z' (A, minus, Z)
[A-Z]  # one of the chars A, B, C, ..., X, Y, or Z
```

More weird syntax...

- The OR operator:

```
(eg|le)gs  # matches eggs or legs
```

- Short forms of common expressions:

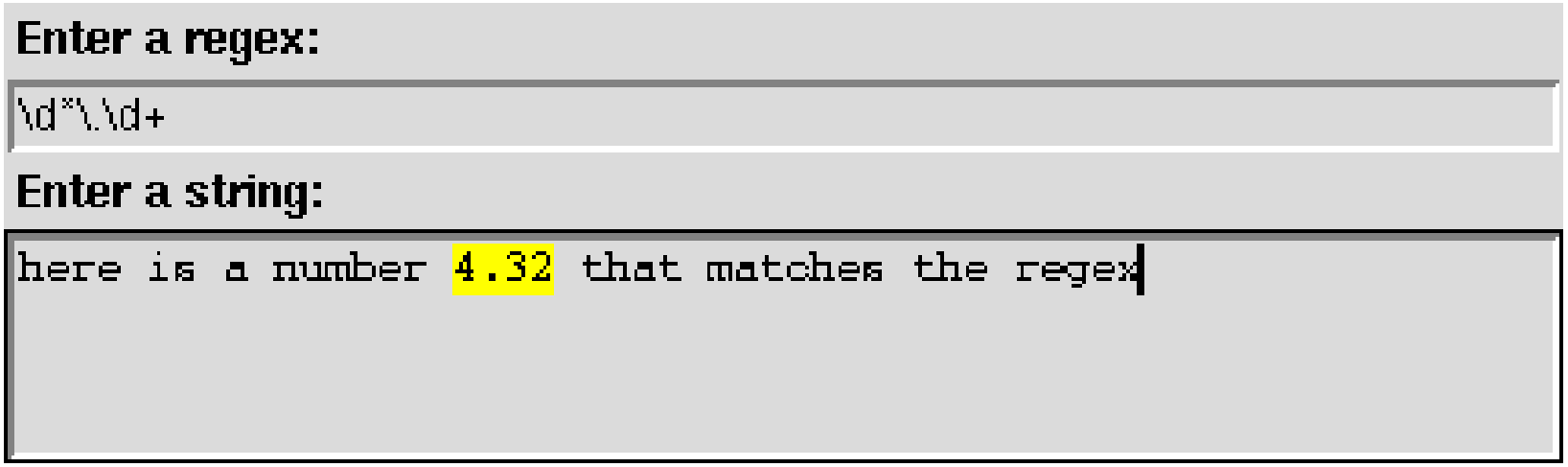
```
\n      # a newline
\t      # a tab
\w      # any alphanumeric (word) character
        # the same as [a-zA-Z0-9_]
\W      # any non-word character
        # the same as [^a-zA-Z0-9_]
\d      # any digit, same as [0-9]
\D      # any non-digit, same as [^0-9]
\s      # any whitespace character: space,
        # tab, newline, etc
\S      # any non-whitespace character
\b      # a word boundary, outside [] only
\B      # no word boundary
```

Quoting special characters

```
\.      # a dot
\|      # vertical bar
\[      # an open square bracket
\)      # a closing parenthesis
\*      # an asterisk
\^      # a hat
\/      # a slash
\\      # a backslash
\{      # a curly brace
\?      # a question mark
```

GUI for regex testing

src/tools/regexdemo.py:



The part of the string that matches the regex is high-lighted

Regex for a real number

- 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 regex

- 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 equivalently (\d is [0-9])
-?[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 + ')'
```

- Note: first test on the most complicated regex in OR expressions

Groups (in introductory example)

- 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:

```
match = re.search(pattern, line)
if match:
    time = float(match.group(1))
    iter = int  (match.group(2))
    eps  = float(match.group(3))
```

- The complete match is group 0 (here: the whole line)

Regex for an interval

- Aim: extract lower and upper limits of an interval:

```
[ -3.14E+00, 29.6524]
```

- Structure: bracket, real number, comma, real number, bracket, with embedded whitespace

Easy start: integer limits

- Regex for real numbers is a bit complicated
- Simpler: integer limits

```
pattern = r'\[\d+,\d+\]'
```

  but this does must be fixed for embedded white space or negative numbers a la

```
[ -3   , 29  ]
```

- Remedy:

```
pattern = r'\[\s*-?\d+\s*,\s*-?\d+\s*\]'
```

- Introduce groups to extract lower and upper limit:

```
pattern = r'\[\s*(-?\d+)\s*,\s*(-?\d+)\s*\]'
```

Testing groups

In an interactive Python shell we write

```
>>> pattern = r'\[\s*(-?\d+)\s*,\s*(-?\d+)\s*\]'
>>> s = "here is an interval: [ -3, 100] ..."
>>> m = re.search(pattern, s)
>>> m.group(0)
[ -3, 100]
>>> m.group(1)
-3
>>> m.group(2)
100
>>> m.groups()   # tuple of all groups
('-3', '100')
```

Named groups

- Many groups? inserting a group in the middle changes other group numbers...
- Groups can be given *logical names* instead
- Standard group notation for interval:

```
# apply integer limits for simplicity: [int,int]
\[\s*(-?\d+)\s*,\s*(-?\d+)\s*\]
```

- Using named groups:

```
\[\s*(?P<lower>-?\d+)\s*,\s*(?P<upper>-?\d+)\s*\]
```

- Extract groups by their names:

```
match.group('lower')
match.group('upper')
```

Regex for an interval; real limits

- Interval with general real numbers:

```
real_short = r'\s*(-?(\d+(\.\d*)?|\d*\.\d+)([eE][+\-]?\d+)?)\s*'
interval = r"\[" + real_short + "," + real_short + r"\]"
```

- Example:

```
>>> m = re.search(interval, '[-100,2.0e-1]')
>>> m.groups()
('-100', '100', None, None, '2.0e-1', '2.0', '.0', 'e-1')
```

i.e., lots of (nested) groups; only group 1 and 5 are of interest

Handle nested groups with named groups

- Real limits, previous regex resulted in the groups

```
('-100', '100', None, None, '2.0e-1', '2.0', '.0', 'e-1')
```

- Downside: many groups, difficult to count right
- Remedy 1: use named groups for the outer left and outer right groups:

```
real1 = \
 r"\s*(?P<lower>-?(\d+(\.\d*)?|\d*\.\d+)([eE][+\-]?\d+)?)\s*"
real2 = \
 r"\s*(?P<upper>-?(\d+(\.\d*)?|\d*\.\d+)([eE][+\-]?\d+)?)\s*"
interval = r"\[" + real1 + "," + real2 + r"\]"
...
match = re.search(interval, some_text)
if match:
    lower_limit = float(match.group('lower'))
    upper_limit = float(match.group('upper'))
```

Simplify regex to avoid nested groups

- Remedy 2: reduce the use of groups
- Avoid nested OR expressions (recall our first tries):

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

- Cost: (slightly) less general and safe regex

Extracting multiple matches (1)

- `re.findall` finds all matches (`re.search` finds the first)

```
>>> r = r"\d+\.\d*"
>>> s = "3.29 is a number, 4.2 and 0.5 too"
>>> re.findall(r,s)
['3.29', '4.2', '0.5']
```

- Application to the interval example:

```
lower, upper = re.findall(real, '[-3, 9.87E+02]')
# real: regex for real number with only one group!
```

Extracting multiple matches (1)

- If the regex contains groups, `re.findall` returns the matches of all groups - this might be confusing!

```
>>> r = r"(\d+)\.\d*"
>>> s = "3.29 is a number, 4.2 and 0.5 too"
>>> re.findall(r,s)
['3', '4', '0']
```

- Application to the interval example:

```
>>> real_short = r"([+\-]?(\d+(\.\d*)?|\d*\.\d+)([eE][+\-]?\d+)?)"
>>> # recall: real_short contains many nested groups!
>>> g = re.findall(real_short, '[-3, 9.87E+02]')
>>> g
[('-3', '3', '', ''), ('9.87E+02', '9.87', '.87', 'E+02')]
>>> limits = [ float(g1) for g1, g2, g3, g4 in g ]
>>> limits
[-3.0, 987.0]
```

Making a regex simpler

- Regex is often a question of structure *and context*
- Simpler regex for extracting interval limits:

```
\[(.*),(.*)\]
```

- It works!

```
>>> l = re.search(r'\[(.*),(.*)\]',
                  ' [-3.2E+01,0.11  ]').groups()
>>> l
('-3.2E+01', '0.11  ')

# transform to real numbers:
>>> r = [float(x) for x in l]
>>> r
[-32.0, 0.11]
```

Failure of a simple regex (1)

- Let us test the simple regex on a more complicated text:

```
>>> l = re.search(r'\[(.*),(.*)\]', \
    ' [-3.2E+01,0.11  ] and [-4,8]').groups()
>>> l
('-3.2E+01,0.11  ] and [-4', '8')
```

  Regular expressions can surprise you...!

- Regular expressions are greedy, they attempt to find the longest possible match, here from `[` to the last (!) comma
- We want a shortest possible match, up to the first comma, i.e., a non-greedy match
- Add a `?` to get a non-greedy match:

```
\[(.*?),(.*?)\]
```

- Now `l` becomes

```
('-3.2E+01', '0.11  ')
```

Failure of a simple regex (2)

- Instead of using a non-greedy match, we can use

```
\[([^,]*),([^\]]*)\]
```

- Note: only the first group (here first interval) is found by `re.search`, use `re.findall` to find all

Failure of a simple regex (3)

- The simple regexes

```
\[([^,]*),([^\]]*)\]
\[(.*?),(.*?)\]
```

  are not fool-proof:

```
>>> l = re.search(r'\[([^,]*),([^\]]*)\]',
                  ' [e.g., exception]').groups()
>>> l
('e.g.', ' exception')
```

- 100 percent reliable fix: use the detailed real number regex inside the parenthesis
- The simple regex is ok for personal code

Application example

- Suppose we, in an input file to a simulator, can specify a grid using this syntax:

```
domain=[0,1]x[0,2] indices=[1:21]x[0:100]
domain=[0,15] indices=[1:61]
domain=[0,1]x[0,1]x[0,1] indices=[0:10]x[0:10]x[0:20]
```

- Can we easily extract domain and indices limits and store them in variables?

Extracting the limits

- Specify a regex for an interval with real number limits
- Use re.findall to extract multiple intervals
- Problems: many nested groups due to complicated real number specifications
- Various remedies: as in the interval examples, see fdmgrid.py
- The bottom line: a very simple regex, utilizing the surrounding structure, works well

Utilizing the surrounding structure

- We can get away with a simple regex, because of the surrounding structure of the text:

```
indices = r"\[([^:,]*):([^\]]*)\]"   # works
domain  = r"\[([^,]*),([^\]]*)\]"    # works
```

- Note: these ones do not work:

```
indices = r"\[([^:]*):([^\]]*)\]"
indices = r"\[(.*?):(.*?)\]"
```

  They match too much:

```
domain=[0,1]x[0,2] indices=[1:21]x[1:101]
       [.....................:
```

  we need to exclude commas (i.e. left bracket, anything but comma or colon, colon, anythin but right bracket)

Splitting text

- Split a string into words:

```
line.split(splitstring)
# or
string.split(line, splitstring)
```

- Split wrt a regular expression:

```
>>> files = "case1.ps, case2.ps,    case3.ps"
>>> import re
>>> re.split(r",\s*", files)
['case1.ps', 'case2.ps', 'case3.ps']

>>> files.split(", ")  # a straight string split is undesired
['case1.ps', 'case2.ps', '   case3.ps']
>>> re.split(r"\s+", "some    words   in a text")
['some', 'words', 'in', 'a', 'text']
```

- Notice the effect of this:

```
>>> re.split(r" ", "some    words   in a text")
['some', '', '', '', 'words', '', '',  'in', 'a', 'text']
```

Pattern-matching modifiers (1)

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

```
if re.search('yes', answer):
```

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

```
if re.search(r'(yes|YES)', answer):
```

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

```
if re.search(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)

```
if re.search('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.search('text with a=1.9672E-04 ')
```

Substitution

- Substitute `float` by `double`:

```
# filestr contains a file as a string
filestr = re.sub('float', 'double', filestr)
```

- In general:

```
re.sub(pattern, replacement, str)
```

- If there are groups in pattern, these are accessed by

```
\1      \2      \3      ...
\g<1>   \g<2>   \g<3>   ...

\g<lower>  \g<upper> ...
```

  in `replacement`

Example: strip away C-style comments

- C-style comments could be nice to have in scripts for commenting out large portions of the code:

```
/*
while 1:
    line = file.readline()
    ...
...
*/
```

- Write a script that strips C-style comments away
- Idea: match comment, substitute by an empty string

Trying to do something simple

- Suggested regex for C-style comments:

```
comment = r'/\*.*\*/'

# read file into string filestr
filestr = re.sub(comment, '', filestr)
```

  i.e., match everything between `/*` and `*/`

- Bad: . does not match newline
- Fix: `re.S` or `re.DOTALL` modifier makes . match newline:

```
comment = r'/\*.*\*/'
c_comment = re.compile(comment, re.DOTALL)
filestr = c_comment.sub(comment, '', filestr)
```

- OK? No!

Testing the C-comment regex (1)

Test file:

```
/*****************************************/
/* File myheader.h                       */
/*****************************************/

#include <stuff.h>  // useful stuff

class MyClass
{
  /* int r; */  float q;
  // here goes the rest class declaration
}

/* LOG HISTORY of this file:
 * $ Log: somefile,v $
 * Revision 1.2  2000/07/25 09:01:40  hpl
 * update
 *
 * Revision 1.1.1.1  2000/03/29 07:46:07  hpl
 * register new files
 *
*/
```

Testing the C-comment regex (2)

- The regex

  ```
  /\*.*\*/  with re.DOTALL (re.S)
  ```

  matches the whole file (i.e., the whole file is stripped away!)
- Why? a regex is by default greedy, it tries the longest possible match, here the whole file
- A question mark makes the regex non-greedy:

  ```
  /\*.*?\*/
  ```

Testing the C-comment regex (3)

- The non-greedy version works
- OK? Yes - the job is done, almost...

```
const char* str ="/* this is a comment */"
```

gets stripped away to an empty string...

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)
```

Files: src/py/intro/swap1.py

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)
```

Files: src/py/intro/swap2.py

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 = "-------------//-------------";
```

???

Regex debugging (1)

- The following useful function demonstrate how to extract matches, groups etc. for examination:

```
def debugregex(pattern, str):
    s = "does '" + pattern + "' match '" + str + "'?\n"
    match = re.search(pattern, str)
    if match:
        s += str[:match.start()] + "[" + \
             str[match.start():match.end()] + \
             "]" + str[match.end():]
        if len(match.groups()) > 0:
            for i in range(len(match.groups())):
                s += "\ngroup %d: [%s]" % \
                     (i+1,match.groups()[i])
    else:
        s += "No match"
    return s
```

Regex debugging (2)

- Example on usage:

```
>>> print debugregex(r"(\d+\.\d*)",
                        "a= 51.243 and b =1.45")

does '(\d+\.\d*)' match 'a= 51.243 and b =1.45'?
a= [51.243] and b =1.45
group 1: [51.243]
```