RegExpr: Review & Wrapup;

Lecture 13b
Larry Ruzzo

Outline

More regular expressions & pattern matching:

- groups
- substitute
- greed

RegExpr Syntax

They're strings

Most punctuation is special; needs to be escaped by backslash (e.g., "\." instead of ".") to get non-special behavior

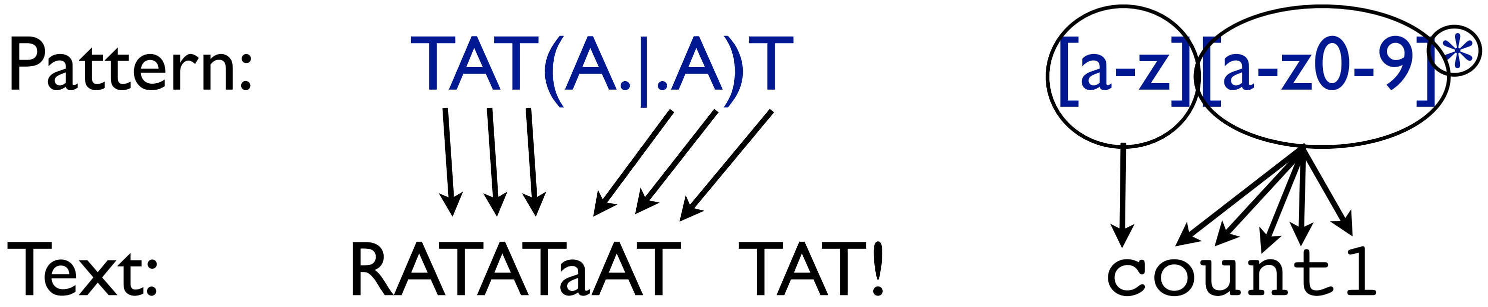
So, "raw" string literals (r′C:\new.txt′) are generally recommended for regexprs

- Unless you double your backslashes judiciously

RegExpr Semantics, I

RexExprs are patterns; they “match” sequences of characters

Patterns "Match" Text



RegExpr Semantics, I
Characters

RexExprs are patterns; they “match” sequences of characters

Letters, digits (& escaped punctuation like ‘\.’) match only themselves, just once

```
r'TATAAT'     'ACGTTATAATGGTATAAT'
```

RegExpr Semantics, 2
Character Groups

"not"; only at start of char group

Character *groups* [abc], [a-zA-Z], [^0-9] also match single characters, any of the characters in the group.

$\texttt{r'T[AG][\^{}GC].T'\ 'ACGT\underline{TGTAAT}GG\underline{TATnCT}'}$

letter group shortcuts

`.` (just a dot) matches any letter (except newline)

```
\s spaces   [ \t\n\r\f\v]
\d digits   [0-9]
\w “word” chars [a-zA-Z0-9_]
\S non-spaces [^ \t\n\r\f\v]
\D non-digits [^0-9]
\W non-word chars [^a-zA-Z0-9_]
```

(but `LOCALE`, `UNICODE` matter)

RegExpr Semantics, 3: Concatenation, Or, Grouping

Parens group subexpressions (& alter reporting)

If R, S are RegExprs, then

- RS matches the *concatenation* of strings matched by R, S individually
- R|S matches the *union* – either R or S

`r'TAT(A.|.A)T'` $'\underline{\texttt{TATCAT}}\texttt{G}\underline{\texttt{TATACT}}\texttt{CC}\overbrace{\texttt{TATCCT}}^{?}'$

`r'(A|G)(A|G)'` matches any of `AA AG GA GG`

RegExpr Semantics, 4 Repetition

If R is a RegExpr, then

R*	matches 0 or more consecutive strings (independently) matching R
R+	1 or more
R{n}	exactly n
R{m,n}	any number between m and n, inclusive
R?	0 or 1

Beware precedence (* > concat > |; use parens if needed)

`r'TAT(A.|.A)*T'` ‘$\underline{\texttt{TATCAT}}\texttt{G}\underline{\texttt{TATAC}}\underbrace{\underline{\texttt{TAT}}\texttt{CAC}\underline{\texttt{T}}}_{?}\underline{\texttt{ATT}}$’

RegExprs in Python

By default

- Case sensitive, line-oriented (\n treated specially)
- Matching is generally "greedy": Finds longest version of earliest starting match
- Next "findall()" match will *not* overlap

```
r".+\.py"  "Two files: hw3.py and upper.py."
            └────────────────────────────┘

r"\w+\.py" "Two files: hw3.py and UPPER.py."
                       └────┘     └──────┘
```

Python Mechanics

`re.match(pat, str)`
matches only at front of string

`re.search(pat,str)`
matches anywhere in string

Return "match" objects or "None"

`re.findall(pat,str)`
finds all (nonoverlapping) matches

Returns list of strings

Many others (split, substitute,...)

“Match” Objects

Retain info about exactly where the pattern matched, and how.

Of special note, *if your pattern contains parenthesized groups*, you can see what, if anything, matched each group, within the context of the overall match.

```
str= 'My birthdate is 09/03/1988'
pat = r'[bB]irth.* (\d{2})/(\d{2})/(\d{4})'
match = re.search(pat,str)
match.groups()
('09', '03', '1988')
```

“digit” ≡ [0-9]

Many more options; e.g., match.start, match.end; see Python docs...

Match object methods

group() entire matching string

group(0) ditto

group(1) string matching 1st paren group

group(1,3) tuple of strings matching 1st & 3rd

start(...) location of start of match

end(...) location of end of match

span(...) return (start,end) locations as a tuple

Pattern Objects & "Compile"

Compile: assemble, e.g., a report, from various sources

```
 mypat = re.compile(pattern[,flags])
```

Preprocess the pattern to make pattern matching fast. Always happens. Do it yourself if you will do *repeated* searches with the same pattern. (Optional flags can modify defaults, e.g., case-sensitive matching, etc.)

Then use:

```
 mypat.{match,search,findall,...}(string)
```

Exercise 1

Suppose “filenames” are upper or lower case letters or digits, starting with a letter, followed by a period (“.”) followed by a 3 character extension (again alphanumeric). Scan a list of lines or a file, and print all “filenames” in it, with*out* their extensions. Hint: use paren groups.

Solution 1

```
import sys
import re

filehandle = open(sys.argv[1],"r")
filecontents = filehandle.read()
myrule = re.compile(
   r"([a-zA-Z][a-zA-Z0-9]*)\.[a-zA-Z0-9]{3}")
#Finds skidoo.bar amidst 23skidoo.barber; ok?
match = myrule.findall(filecontents)
print match
```

Exercise 2

Find & print all email addresses in, say, the course home page

```
ruzzo@cs.washington.edu
jht@u.washington.edu
obama2@whitehouse.gov
word@word.word.word.word.dom,
```

(where dom is 2-3 letters or digits, e.g. , ".edu", ".ru")

Solution 2

```
import re
page=open('index.html').read()
emailpat = r'\w+@\w[\w.]*\.\w{2,3}'
re.findall(emailpat,page)
```

```
  ['jht@u.washington.edu','jht@u.washington.edu']
```

NB: '\w' after @ avoids matching `a@.xyz`, but unfortunately allows `a@b....xyz`. Part of the general art of using Reg Exps is taste in how loose/rigid to make your patterns. `r'\w+@(\w+\.)+\w{2,3}'` is better, pattern-wise, but the parens change what findall reports. (try it...) See "`(?: ... )`" for a better way.

Substitute

A very handy RegExp feature is the ability to *substitute*, one string for another

```
>>> re.sub('dog','cat','dogfish')
'catfish'
>>> pat = r'(\w)(\w+)'
>>> rep =  r'\2\1ay'
>>> re.sub(pat,rep, "Hello World!")
'elloHay orldWay!'
```

text matching the 2nd paren group

text matching the 1st paren group

Exercise 3

In the course home page, *replace* any

```
anyname@u.washington.edu
```

email addresses by the shorter equivalent

```
anyname@uw.edu
```

Avoid picking up non-email addresses, like

```
^$#@(&*%$!!*@u.washington.edu!
```

Solution 3

```
import re
page=open('index.html').read()
atupat = r'(\w)@u.washington.edu(\W)'
re.sub(atupat, r'\1@uw.edu\2', page)
```

better (also works at end of string):

match at word boundary

```
atupat = r'(\w@)u.washington.edu\b'
re.sub(atupat, r'\1uw.edu', page)
```

Exercise 4

Greedy matching is often what you want, but sometimes not.

E.g., find all images in the course home page

```
<img src="foo.png" ...></p>
```

The "obvious" `r'<img.*>'` may run past the matching '>'. (Try it!) Fixes:

- read the regexp docs for "non-greedy" matching, *or*
- think of something to use instead of .* so you don't gobble extra angle brackets.

Solution 4

```
import re
page=open('index.html').read()
re.findall(r'<img.*>',page)          ←No
re.findall(r'<img[^>]*>',page) ←Yes
```

```
['<img src="data/athelvetica84.png" height=13 align=bottom>',
 '<img src="http://healthlinks.washington.edu/images/lock.gif">',
 '<img\n src="http://healthlinks.washington.edu/images/lock.gif">',
 '<img src="data/athelvetica84.png" height=14 align=bottom>']
```

RegExp Summary

Search for/replace complex patterns

Not essential, but convenient

Pattern: a string; “compiled” to a pattern object

Use raw strings (or many backslashes)

findall returns list of (matching) strings; other functions usually return “match objects”