Regular Expressions, II

Lecture 12b
Larry Ruzzo

Outline

- Some efficiency tidbits
- More regular expressions & pattern matching

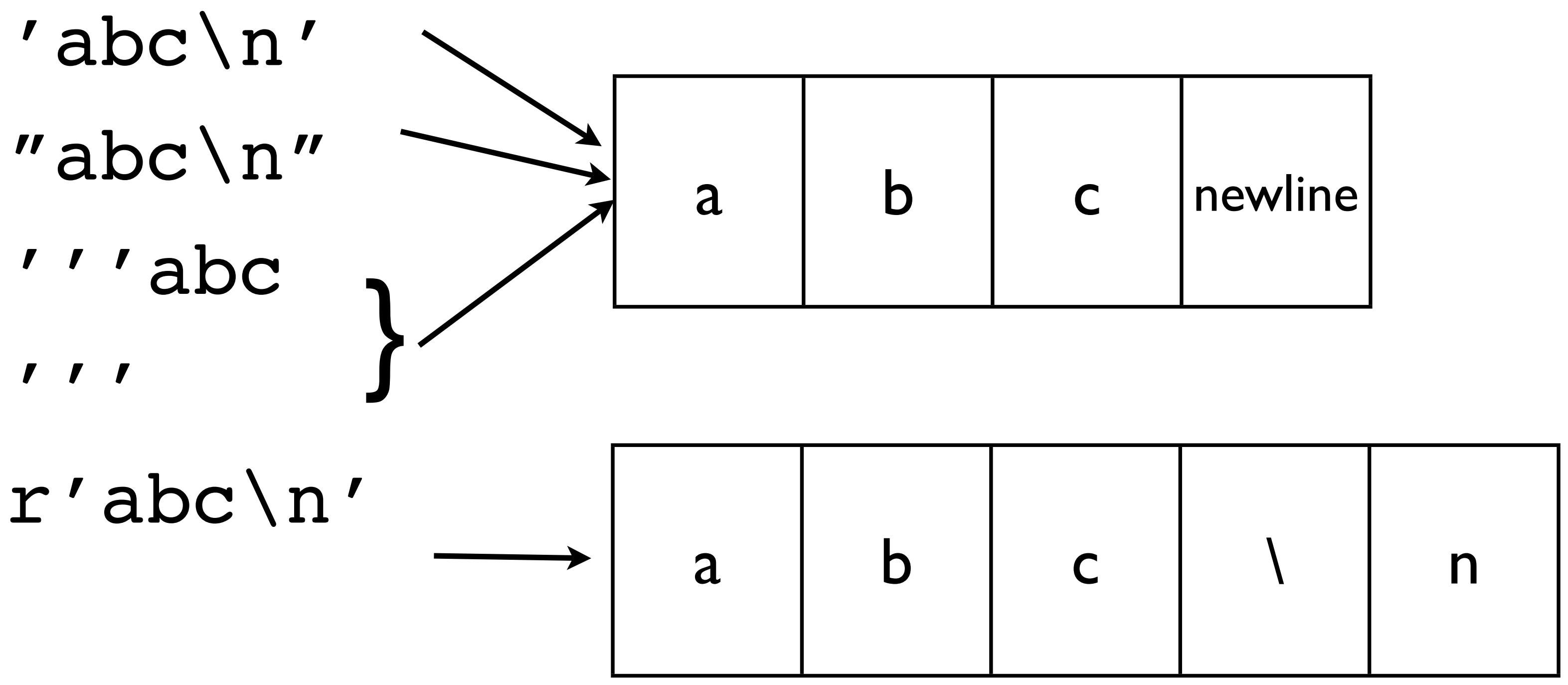
Time and Memory Efficiency

- Avoid premature optimization; get a *working* solution, even if big & slow
    - yes, wrong answers might as well be fast, but...
- Simple tricks may suffice
    - e.g., one line or one chromosome at a time
- Measure, don't guess
    - even professionals are notoriously bad at predicting the bottlenecks
- Then attack the bottleneck; "80-20 rule"

How to measure space?

```
>>> dir('')
['__add__', …, '__sizeof__', … 'split', …, 'strip', …]
>>> help(''.__sizeof__)
Help on built-in function __sizeof__:
__sizeof__(...)
    S.__sizeof__() -> size of S in memory, in bytes
>>> (''.__sizeof__(),'a'.__sizeof__(),'ab'.__sizeof__())
(40, 41, 42)
>>> dir()
['__builtins__', '__doc__', ..., 'fh', 'x', 'y', ‘z’]
>>>
```

Strings Again



Only Skin Deep

```
>>> 'ab'
'ab'
>>> "ab"
'ab'
>>> '''ab'''
'ab'
>>> r'ab'
'ab'
>>> r"ab"
'ab'
>>> r'''ab
... '''
'ab\n'
>>> 'ab' == "ab" == '''ab''' == r'ab' == r"ab"
True
```

These are *external*, not internal, differences, hopefully convenient for data entry.

Internally, a string is a string.

Regular Expressions

- A language for simple patterns, based on 4 simple primitives
  - match single letters
  - this OR that
  - this FOLLOWED BY that
  - this REPEATED 0 or more times
- A specific syntax (fussy, and varies among pgms...)
- A library of utilities to deal with them
- Key features: Search, replace, dissect

RegExprs in Python

http://docs.python.org/library/re.html

Simple RegExpr Testing

```
>>> import re
>>> str1 = 'what foot or hand fell fastest'
>>> re.findall(r'f[a-z]*', str1)
['foot', 'fell', 'fastest']

>>> str2 = "I lack e's successor"
>>> re.findall(r'f[a-z]*',str2)
[]
```

Definitely recommend trying this with examples to follow, & more

Returns list of all matching substrings.

Exercise: change it to find strings starting with f and ending with t

Exercise: In honor of the winter Olympics, “-ski-ing”

Download & save war_and_peace.txt

Write py program to read it line-by-line, use re.findall to see whether current line contains one or more proper names ending in “...ski”; print each (but don’t print [])

Mine begins:

```
['Bolkonski']
['Bolkonski']
['Bolkonski']
['Bolkonski']
['Bolkonski']
['Razumovski']
['Razumovski']
['Bolkonski']
['Spasski']
...
['Nesvitski', 'Nesvitski']
```

-ski solution

```
import re

fh = open('war_and_peace.txt')
for line in fh:
  hits=re.findall('[A-Z][a-z]*ski',line)
  if hits != []:
    print hits
```

RegExpr Syntax

They’re strings

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

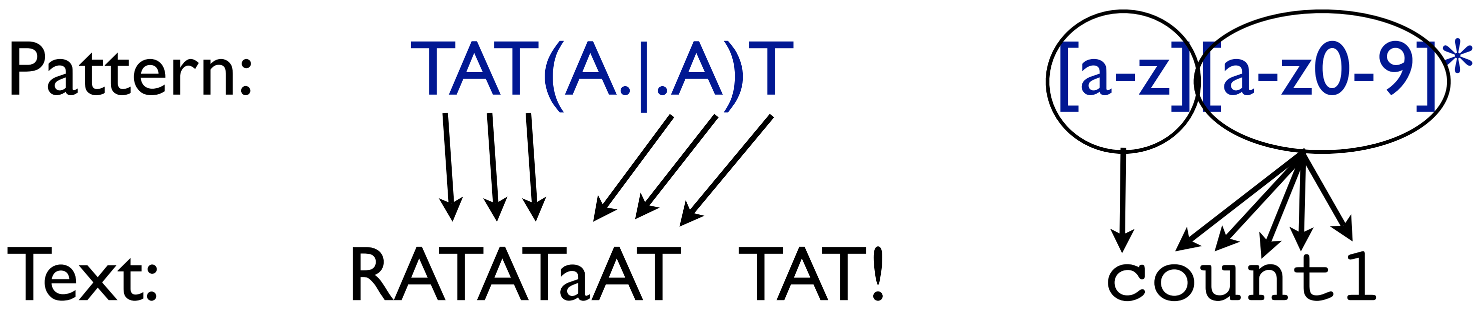
So, “raw” string literals (r'C:\new.txt') are generally recommended for regexprs

- Unless you double your backslashes judiciously

RegExpr Semantics, I Characters

RexExprs are patterns; they “match” sequences of characters

Patterns “Match” Text



RegExpr Semantics, 1 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

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

```
r'T[AG][^GC].T' 'ACGTTGTAATGGTATnCT'
```

Matching any of a set of individual letters

- Square brackets mean that any of the listed characters will do
- `[ab]` means either "a" or "b"
- You can also give a range:
- `[a-d]` means "a" "b" "c" or "d"
- Negation: caret means "not"

```
[^a-d]    # anything but a, b, c or d
```

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

You can group subexpressions with parens

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'` $'\underline{\texttt{TATCAT}}\texttt{G}\underline{\texttt{TATACT}}\texttt{CC}\overset{?}{\overbrace{\texttt{TATCCT}}}'$

`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 > |)

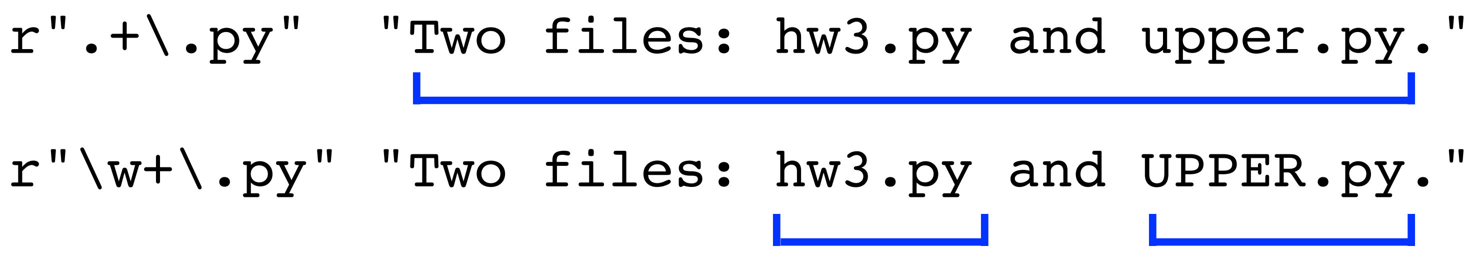
$$\texttt{r'TAT(A.|.A)*T' '}\underline{\texttt{TATCAT}}\texttt{G}\underline{\texttt{TATAC}}\overbrace{\underline{\texttt{TAT}}\texttt{CAC}\underline{\texttt{T}}}^{?}\underline{\texttt{ATT}}\texttt{'}$$

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

`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.match(pat,str)
match.groups()
('09', '03', '1988')
```

“digit” ≡ [0-9]

Many more options; see Python docs...

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. Useful if your code 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 3

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 3

```
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
```

Basics of regexp construction

- Letters and numbers match themselves
- Normally case sensitive
- Watch out for punctuation–most of it has special meanings!

Wild cards

- ”.” means ”any character”
- If you really mean ”.” you must use a backslash
- WARNING:
  - backslash is special in Python strings
  - It’s special again in regexps
  - This means you need too many backslashes
  - We will use ”raw strings” instead
  - Raw strings look like `r"ATCGGC"`

Using . and backslash

- To match file names like "hw3.pdf" and "hw5.txt":

```
hw.\....
```

Zero or more copies

- The asterisk repeats the previous character 0 or more times
- "ca*t" matches "ct", "cat", "caat", "caaat" etc.
- The plus sign repeats the previous character 1 or more times
- "ca+t" matches "cat", "caat" etc. but not "ct"

Repeats

- Braces are a more detailed way to indicate repeats
- A{1,3} means at least one and no more than three A's
- A{4,4} means exactly four A's

simple testing

```
>>> import re
>>> string = 'what foot or hand fell fastest'
>>> re.findall(r'f[a-z]*', string)
['foot', 'fell', 'fastest']
```

Practice problem 1

- Write a regexp that will match any string that starts with "hum" and ends with "001" with any number of characters, including none, in between
- (Hint: consider both "." and "*")

Practice problem 2

- Write a regexp that will match any Python (.py) file.
- There must be at least one character before the "."
- ".py" is not a legal Python file name
- (Imagine the problems if you imported it!)

All of these objects! What can they do?

Functions offered by a Pattern object:

- `match()`–does it match the beginning of my string? Returns None or a match object
- `search()`–does it match anywhere in my string? Returns None or a match object
- `findall()`–does it match anywhere in my string? Returns a list of strings (or an empty list)
- Note that `findall()` does NOT return a Match object!

All of these objects! What can they do?

Functions offered by a Match object:

- `group()`–return the string that matched
  `group()`–the whole string
  `group(1)`–the substring matching 1st parenthesized sub-pattern
  `group(1,3)`–tuple of substrings matching 1st and 3rd parenthesized sub-patterns

- `start()`–return the starting position of the match

- `end()`–return the ending position of the match

- `span()`–return (start,end) as a tuple

A practical example

Does this string contain a legal Python filename?

```
import re
myrule = re.compile(r".+\.py")
mystring = "This contains two files, hw3.py and uppercase.py."
mymatch = myrule.search(mystring)
print mymatch.group()
This contains two files, hw3.py and uppercase.py
# not what I expected!  Why?
```

Matching is greedy

- My regexp matches "hw3.py"
- Unfortunately it also matches "This contains two files, hw3.py"
- And it even matches "This contains two files, hw3.py and uppercase.py"
- Python will choose the longest match
- I could break my file into words first
- Or I could specify that no spaces are allowed in my match

A practical example

Does this string contain a legal Python filename?

```
import re
myrule = re.compile(r"[^ ]+\.py")
mystring = "This contains two files, hw3.py and uppercase.py."
mymatch = myrule.search(mystring)
print mymatch.group()
hw3.py
allmymatches = myrule.findall(mystring)
print allmymatches
['hw3.py','uppercase.py']
```

Practice problem 3

- Create a regexp which detects legal Microsoft Word file names
- The file name must end with ".doc" or ".DOC"
- There must be at least one character before the dot.
- We will assume there are no spaces in the names
- Print out a list of all the legal file names you find
- Test it on testre.txt (on the web site)

More challenge? or ".docx" or ".DOCX"

Practice problem 4

- Create a regexp which detects legal Microsoft Word file names that do not contain any numerals (0 through 9)
- Print out the start location of the first such filename you encounter
- Test it on testre.txt

Practice problem

- Create a regexp which detects legal Microsoft Word file names that do not contain any numerals (0 through 9)
- Print out the “base name”, i.e., the file name after stripping of the .doc extension, of each such filename you encounter. Hint: use parenthesized sub patterns.
- Test it on testre.txt

Practice problem 1 solution

Write a regexp that will match any string that starts with "hum" and ends with "001" with any number of characters, including none, in between

```
myrule = re.compile(r"hum.*001")
```

Practice problem 2 solution

Write a regexp that will match any Python (.py) file.

```
myrule = re.compile(r".+\.py")

# if you want to find filenames embedded in a bigger
# string, better is:
myrule = re.compile(r"[^ ]+\.py")
# this version does not allow whitespace in file names
```

Practice problem 3 solution

Create a regexp which detects legal Microsoft Word file names, and use it to make a list of them

```
import sys
import re
filename = sys.argv[1]
filehandle = open(filename,"r")
filecontents = filehandle.read()
myrule = re.compile(r"[^ ]+\.[dD][oO][cC]")
matchlist = myrule.findall(filecontents)
print matchlist
```

Practice problem 4 solution

Create a regexp which detects legal Microsoft Word file names which do not contain any numerals, and print the location of the first such filename you encounter

```
import sys
import re
filename = sys.argv[1]
filehandle = open(filename,"r")
filecontents = filehandle.read()
myrule = re.compile(r"[^ 0-9]+\.[dD][oO][cC]")
match = myrule.search(filecontents)
print match.start()
```

Regular expressions summary

- The `re` module lets us use regular expressions
- These are fast ways to search for complicated strings
- They are not essential to using Python, but are very useful
- File format conversion uses them a lot
- Compiling a regexp produces a Pattern object which can then be used to search
- Searching produces a Match object which can then be asked for information about the match