Lecture 7

Ashwini Vaidya

11/5

Regular expressions are a "generalized pattern language" (Friedl) that is used for powerful text processing. In the simplest example, when we type a character like '*' in a command in the UNIX terminal:

```
ls *.txt
```

We get a list of all files that have the extension '.txt'. In this expression, the '*' means "match anything" and hence the expression implies, list any file that starts with any name and ends with '.txt'

If we use the search function in a text editor to look for a word like 'in', then the editor is likely to give you all instances of 'in', even when it's found in words like 'finish' or 'pin'. Most likely, the editor is performing a type of greedy search using 'in'–and doesn't understand the human idea of word 'in'.

Metacharacters and literals are the two major types of regexes. Note that conventions followed for literals are slightly different across programming languages. But by and large the metacharacters have the same (or similar) meanings.

To use regular expressions in python, you need to import the regex package :

```
import re
```

In Python, regular expressions are usually written using `r'<expression>'`. E.g.

```
line='This cat is in the room'
patt= re.compile(r'^cat')
result = patt.match(line)
```

```
(where line is the string that we're checking for the pattern).

OR
result = re.match(patt, line)
```

Table 1: Metacharacters

Character	Explanation
.	Any ONE character except a newline
$*$	Zero or more instances of a character
$+$	One or more instances of a character
^	A character at the beginning of a line
$	A character at the end of a line
?	0 or 1 (only) characters
|	or–matches either expression it separates
(..)	used to limit scope of — (and some other uses)

Two types of regular expressions ^ and '$' which signify the start and end of a sentence, and will be used to search for sequences in particular positions. What if I try and match '^in$' or simply '^$'

Date matching is a useful use case for some of these expressions, e.g. July fourth or Jul 4 or 4. It's possible to say r'(Jul|July)\s(fourth|4th|4)'. Also possible to say r'(July?)', because it matches 0 or 1 of the preceding character. Further, it's possible to simplify the entire expression further by '(Jul|July)\s(fourth|4(th)?)'

$*$ and $+$ (also ?) are quantifiers because they regulate how many times an expression can be matched. Especially useful while matching spaces. E.g. in an HTML tag like $\langle ahref \rangle$, it's possible to get one or more spaces. An expression like r'$\langle +href \rangle$' will match one or more spaces after 'a'. On the other hand r'¡a *href¿' will match $\langle ahref \rangle$ as well, because the criteria is 0 or more.

```
>>> x = re.search("brown","The cat is brown")
>>> x
<_sre.SRE_Match object at 0x110327d98>
>>> x = re.findall("brown","The cat is brown")
>>> x
```

Table 2: Literals

Literals	Explanation
\b	A sequence of alphanumeric characters separated by space
\d	A decimal digit, equivalent to the set [0-9] when Unicode is not specified
\s	Any whitespace character including space, tab or newline
\w	Any alphanumeric character and underscore [a-zA-Z_]

```
['brown']
>>>
```

Character class

We might want to be able to find all examples of 'The' and 'the' in a sentence, but instead of performing two separate searches, we can use a character class eg. '[Th]e'. This might also be useful in catching spelling differences between British and American English e.g. r'gr[ae]y'

Character classes can also be ranges e.g. if I want to search for words like 'H1', 'H2','H3', I can use a range r'H[1-3]' or r'H[1-10]'

Inside a character class, the rules for regex are also a bit different, so if I want to exclude a certain character, e.g. 'H1' but include others, it's possible to write 'H[^1]'– here the symbol ^ has a completely different meaning (note it is literally interpreted if it's not at the beginning of the line).

Special characters lose their special meaning inside sets. For example, [(+*)] will match any of the literal characters '(', '+', '*', or ')'.

```
r"M[ae][iy]er"
```

Groups

"Parentheses can "remember" text matched by the subexpression they enclose".(Friedl, pg 19).

For cases like the double word problem, e.g. instances of 'the the' occurring as typos in a text, it's possible to write r'(the the)' (what's wrong with this?– 'the theory' will also be caught by the regex). A better way to write it would be r'\bthe\b \bthe\b'

But can we search for doubling errors irrespective of the word that's doubled? Not possible unless each word is individually specified. To do this, we need the expression to 'remember' what it's already seen and this is accomplished via backreferencing or grouping

```
>>> x = re.search(r"([A-Za-z]+)\s+\1","The cat is brown brown")
>>> x.group()
'brown brown'
>>> x = re.search(r"([A-Za-z]+)\s+\1","The the man is in the room",re.I)
>>> x.group()
'The the'
>>> x.group(1)
'The'
>>> x.group(0)
'The the'
>>> x.group(2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: no such group
```

References

Mastering Regular Expressions by Jeffrey Friedl

http://www.python-course.eu/re.php