

COM6115: Text Processing

Regular Expressions in Python

Extended presentation

Provides a more detailed account of regular expressions facilities in Python and their use

Mark Hepple

Department of Computer Science
University of Sheffield

What are Regular Expressions?

- A regular expression (**regex**) is a *pattern* that describes a *set of strings*
 - ◇ a *matching* process tests if a given string matches the pattern
 - ◇ may also then *modify* the string
 - e.g. by *substituting* a substring, or *splitting* it into substrings
- Regular expressions are a powerful programming tool, used widely in computing / text processing
 - ◇ found (e.g.) in Perl, Tcl/Tk, Java, grep, sed, awk, vi, emacs, lex
 - ◇ *but* note that regex syntax varies
- Many applications — some random examples:
 - ◇ strip the html out of a set of web pages
 - ◇ extract comment blocks from a program (e.g. to build documentation)
 - ◇ check a document for doubled words (“the the”, “here here”)

- Simplest example of a regex is a *literal pattern*
 - ◇ most characters just match against themselves
 - ◇ likewise, most char sequences form a regex to match against identical char sequence in a string
 - ◇ *but* some chars have special behaviour: *metachars*
- For example, string "pen":
 - ◇ as a regex, matches any string containing *substring* pen
 - e.g. "the pen broke"
 - e.g. "what is epenthesis?"

Simple Patterns — Example: Python

- Python provides extensive regex facilities
 - ◇ not in basic language — must import module "re"
 - ◇ can do regex matching using module functions 'directly'

- Example:

```
import sys, re
with open(sys.argv[1], 'r') as infs:
    for line in infs:
        if re.search('pen', line):
            print(line, end='')
```

- ◇ **search** scans for *first* substring matching regex *anywhere* in string
- ◇ if finds match, returns a *match object*, else returns None (=False)

Simple Patterns — Example: Python (ctd)

- When a regex is to be used many times, is better (i.e. faster) to *compile* a regex *object*

- Example:

```
import sys, re
penRE = re.compile('pen')
with open(sys.argv[1], 'r') as infs:
    for line in infs:
        if penRE.search(line):
            print(line, end='')
```

- Assigning object to a *well-named variable* gives more-readable code
e.g. having regexes for 'word', 'URL', etc

Alternatives in Regexes and Groupings

- To specify that one of several options are permitted in a match, separate them by a *vertical bar* (or 'pipe')

◇ *Example:* regex "car|bike|train" matches any of:

```
carnation  
motorbike  
detraining
```

- Can *group* parts of a pattern, using *parentheses*

◇ *Example:* regex "(e|i)nquir(e|y|ing)" matches any of:

```
enquiry  
inquiring  
enquire
```

Quantifiers

- *Quantifiers* allow you to specify that you want *some number* of occurrences of a (sub)pattern
- Following quantifiers based on *Kleene* notation:

*	zero or more
+	one or more
?	zero or one occurrences (i.e. optional)

- ◇ indicate something about the *immediately preceding* item in pattern
- Example: does regex "ab*d?e" match ...
 - ◇ abde ✓
 - ◇ bde ✗
 - ◇ abd ✗
 - ◇ aeeee ✓
 - ◇ abbbbbbbbbbde ✓
 - ◇ abdde ✗

Quantifiers (contd)

- *Counts* of possible repetitions can be specified with notation:

$\{3, 12\}$	at least 3 and at most 12 occurrences
$\{3, \}$	at least 3 occurrences
$\{, 12\}$	between zero and 12 occurrences
$\{3\}$	exactly 3 occurrences

- *Example*: matching "a{5}b{1,4}c{2,}" ...

◇ aaaaabcc	✓
◇ aaaaabc	×
◇ aaaaabcccc	✓
◇ aaaabcc	×
◇ aaaaabbbbbcc	×

Grouping with parentheses

- Any part of a regex can be *grouped with parentheses* and then treated as a *unit*
- Example:* matching "c(ab)*(de)+" ...

◇ ghcabdemn	✓
◇ ghcabbdemn	×
◇ ghcababdemn	✓
◇ ghcabababababababdemn	✓
◇ ghcabaddemn	×
◇ ghcdededededemn	✓
◇ ghcbdemn	×
◇ ghcdemn	✓

Character classes

- Use *square brackets* to indicate a character class
e.g. class `[abcde]` will match a *single* char, provided it is one of those listed
- *Example*: matching regex `"c[ad]r"`

◇ car	✓
◇ cdr	✓
◇ cadr	✗
◇ caddddr	✗

- *Example*: matching regex `"c[ad]*r"`

◇ car	✓
◇ cdr	✓
◇ caaaadr	✓
◇ caaadaaar	✓
◇ caaars	✓

Character classes (contd)

- Can specify char **ranges** using a hyphen, e.g.
 - ◇ [A-Z] upper case roman alphabet
 - ◇ [a-z] lower case roman alphabet
 - ◇ [A-Za-z] upper and lower case letters
 - ◇ [0-9] digits 0..9
 - ◇ [A-F] upper case letters A..F
- But *be careful* – best to stick to ranges with *clear semantics*
 - ◇ *OR* may get *unexpected behaviour*, e.g.
 - ◇ [A-z] valid *but* \neq [A-Za-z]
 - ◇ [a-Z] *not* valid
 - ◇ [F-A] *not* valid

Character classes (contd)

- *Example:* regex "[a-d][m-z][0-9]*"
 - ◇ matches any string containing any letter between a and d
 - ◇ followed by any letter between m and z
 - ◇ followed by zero or more digits in the range 0 to 9

Hence, this regex matches:

- ◇ bm3405 ✓
- ◇ dx19 ✓
- ◇ Thiscontainsav3440andsoqualifies ✓
 - *but* where is the *first* match within the string?

Built-in Character Classes & Negation

- Some common char classes have *predefined* names:

.	matches <i>any</i> char
\d	abbreviates [0-9]
\w	abbreviates [A-Za-z0-9_]
\s	abbreviates [\f\t\n\r]

- To *negate* a char class, put the “carat” sign ^ at the start
 - ◇ matches anything *except* chars indicated

[^abc]	matches everything but abc
[^\d\s]	does not match digits or white space
\D	negation of \d
\W	negation of \w
\S	negation of \s

- *Anchors* tie matching to appear at certain positions:
 - ◇ `^` matches the *beginning* of the string
 - ◇ `$` matches the *end* of the string
 - ◇ `\b` matches at word *boundary* (between `\w` and `\W`)
 - *but* see slide on *raw strings* before using `\b` in Python
- *Examples:*

<code>"^author"</code>	match strings starting with author
<code>"[^0-9]"</code>	negation
<code>"^[0-9]"</code>	start anchor (string must begin with digit)
<code>">>\$"</code>	looks for » at end of string
<code>"\bfu(n nny)"</code>	matches fun, funny, not refund

Raw Strings and Literal Metacharacters

- Some special chars, such as `\b`, present a problem in Python
 - ◇ normally, in strings, `\b` means “backspace”
 - ◇ to get Python to handle `\b` correctly in regexes, must mark it as a *raw string*
 - ◇ a *raw string* is preceded by `r`

e.g. `funRE = re.compile(r"\bfu(n|nny)")`

- Can use `\` to ‘escape’ a metacharacter back to its original meaning
 - e.g. might actually want to look for occurrences of `^` in text
 - ◇ in that case, use `"\^"`

Extracting matched parts

- Often want to get hold of the *part* of a string that matched against a regex, or against specific *sub-parts* of regex
 - ◇ need a means to *identify* the relevant sub-parts
- Common method:
 - ◇ use *parenthesised portions* of regex, called *groups*
 - ◇ identify the different groups *numerically*
 - ◇ count 'open' brackets in from left-hand side of regex (starting with 1)

e.g. the groups within the regex `' ([A-Za-z]+)(ed|ing) '` are:

group 1: `(([A-Za-z]+)(ed|ing))`

group 2: `([A-Za-z]+)`

group 3: `(ed|ing)`

Extracting matched parts (ctd)

- In Python, a successful regex match returns a *match object*:
 - ◇ object stores information about the match
 - i.e. of matching substrings and their spans
 - ◇ info accessible using object's methods: `group`, `groups`, `span`
- *Example:*

```
>>> sent = "I have baked a cake!"
>>> m = re.search(' ([a-z]+)(ed|ing)) ',sent)
>>> m
<sre.SRE_Match object at 0x1081b5030>
>>> m.group()      # returns substring for overall regex match
' baked '
>>> m.span()       # returns start/end indices for full match
(6, 13)
>>> m.group(1)     # returns substring for group 1
'baked'
>>> m.span(1)      # returns start/end indices for group 1
(7, 12)
```

Extracting matched parts (ctd)

- *Example continued ...:*

```
>> sent = "I have baked a cake!"
>> m = re.search(' ([a-z]+)(ed|ing)) ',sent)
....
>> m.group(2)    # substring for group 2
'bak'
>> m.group(3)    # substring for group 3
'ed'
>> m.group(4)    # no group 4 - throws an error
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: no such group
>> m.groups()    # list of matches for groups 1 upwards
('baked', 'bak', 'ed')
>> m.group(0)    # same as m.group()
' baked '
```

Finding Multiple Regex Matches

- Python provides methods for finding multiple matches at same time
- Method `findall` returns list of matches, as
 - ◇ list of matches for full regex (group 0s), if regex has no groups (i.e. parenthesised sub-parts)
 - ◇ list of n-tuples of group matches (for groups 1+), if regex has groups
- *Example:*

```
»» str = 'al bill 22 chad 333 dave eric 55'
»» mm = re.findall('[a-z]+ \d+',str)
»» mm
['bill 22', 'chad 333', 'eric 55']
»» mm = re.findall('([a-z]+) (\d+)',str)
»» mm
[('bill', '22'), ('chad', '333'), ('eric', '55')]
»»
```

Finding Multiple Regex Matches (ctd)

- Method `finditer` returns an *iterator* for a sequence of *match objects*, one for each match
- *Example:*

```
»> str = 'al bill 22 chad 333 dave eric 55'
»> mm = re.finditer('[a-z]+ (\d+)',str)
»> for m in mm:
    print(m.group(1),end='')

bill chad eric
»>
```

Greedy Matching

- Matching of regexes is *greedy*, for Python (and for PERL)
 - ◇ always takes longest matching string
 - ◇ for each quantified sub-pattern, tries to match to *longest substring*
 - ◇ this is not always the behaviour you need
- *Example*: attempt to create a regex for capturing HTML tags

```
»> str = '<p><b>hello</b></p>'  
»> tag = re.compile('<.*>')  
»> m = tag.search(str)  
»> m.group()  
'<p><b>hello</b></p>'  
»>
```

- ◇ doesn't work due to greedy matching behaviour

Greedy Matching (ctd)

- Can force *non-greedy* matching of a quantifier by adding a `'?'`
 - ◇ quantifiers `*?`, `+?`, `??`, `{m,n}?` match to the *shortest* string possible for overall match
- *Example*: capturing HTML tags, again:

```
»> str = '<p><b>hello</b></p>'  
»> tag = re.compile('<.*?>')  
»> m = tag.search(str)  
»> m.group()  
'<p>'  
»> tag.findall(str)  
['<p>', '<b>', '</b>', '</p>']  
»>
```

(Note: other ways to solve this problem, e.g. with regex `'<[^>]*>'`)

Modifying Matching

- Can provide *flags* that *modify* how a regex is compiled, and thereby its behaviour in matching, e.g.

flag (shortname)	meaning
ignorecase (I)	do case-insensitive matching (treats [A-Z] same as [a-z])
multiline (M)	multiline matching – allows ^,\$ to match start/end of lines in multiline string
dotall (S)	allow "." metachar to match newlines in multiline strings

e.g.

```
>> m = re.search('[a-z]+',str,re.IGNORECASE)
>> word = re.compile('[a-z]+',re.I)
>> firstword = re.compile('^ [a-z]+',re.I|re.M)
>> str='This and that.\nAnd the other.'
>> firstword.findall(str)
['This', 'And']
```

Making substitutions

- A key operation is *substring replacement*, or *substitution*
- For *substitution* in Python, can use regex object method `sub()`
 - ◇ has args for replacement string, and string being matched against
 - ◇ optional keyword arg `count=n` sets upper limit on count of replacements made – if absent (or `count=0`), *all* occurrences replaced
 - ◇ returns string that results after replacements done
- *Example:*

```
»> names = re.compile('(Alan|Bill|Chad)')
»> str = 'Vote for Alan. Bill is clever.'
»> names.sub('Mark',str)
'Vote for Mark. Mark is clever.'
»> names.sub('Mark',str,count=1)
'Vote for Mark. Bill is clever.'
»>
```


Making substitutions (ctd)

- Alternative method `subn()`, returns (as an n-tuple) both modified string and a *count* of the replacements done
 - ◇ corresponding class level function additionally takes regex as its first arg
- Often want to use substrings from a match in constructing the replacement string — for this use *backreferences*
 - ◇ a *backreference* has form “\N” for some $N \geq 1$
 - ◇ refers to the text matched to the *Nth group* of the regex
 - ◇ *must* use a *raw string* when backrefs are present

Making substitutions (ctd)

Example:

```
»> swap = re.compile('(Anne|Abi) (likes|hates) (Bill|Bob)')
»> str = 'I heard that Anne likes Bill, today.'
»> swap.sub(r'\3 \2 \1',str)
'I heard that Bill likes Anne, today.'
»>
```

- Backrefs can also be used *within* a regex
 - ◇ indicate that some matched string must *appear again*

e.g. regex: `r'\b(\w+) \1\b'`

matches if same word appears *twice*, as in “kick the the ball”

Splitting & Joining

- Python regexes have a **split** method to split strings:
 - ◇ regex matches identify the split points
 - ◇ strings *between* the split points are returned as a list

e.g.

```
»> tokenize = re.compile('[^A-Za-z]+')
»> str = 'I said, "No - go away!'. '
»> tokenize.split(str)
['I', 'said', 'No', 'go', 'away', '']
»>
```

- Python strings are *objects* with own methods. To **join** a list of strings, call `join()` method from instance of *delimiter* string:

e.g.

```
»> tokens=['this','and','that']
»> '::'.join(tokens)
'this::and::that'
»>
```

Splitting & Joining (ctd)

- Some other Python string methods are v.useful for text processing. Where they are sufficient for your needs, have *advantages*:
 - ◇ are *simpler* to use than regexes
 - ◇ are *faster*/more efficient than regexes
 - ◇ method names provide a *clear 'semantics'* when code is read
- Some example Python string methods:
 - ◇ *string testing* methods such as:
`isupper()`, `islower()`, `isalpha()`, `isdigit()`, `isalnum()`
 - ◇ upper/lower *case conversion*: `upper()`, `lower()`, `capitalize()`
 - ◇ *splitting* on a fixed string (not regex): `split()`