Lecture 5:
Pattern Matching

FAES BIOF 309 Introduction to Python

Christopher Coletta

Covered in this lecture

- Definition
- Workflow for using regexs in Python
- Discussion of metacharacters
- Greedy vs. lazy
- Backrefernces

Definition

Regular expression

From Wikipedia, the free encyclopedia

In computing, a **regular expression** is a specific pattern that provides concise and flexible means to "match" (specify and recognize) strings of text, such as particular characters, words, or patterns of characters. Common abbreviations for "regular expression" include **regex** and **regexp**.

The concept of regular expressions was first popularized by utilities provided with Unix distributions, in particular the editor ed and the filter grep.[*citation needed*] A regular expression is written in a formal language that can be interpreted by a regular expression processor, which is a program that either serves as a parser generator or examines text and identifies parts that match the provided specification.

Example usages

- Grabbing the text inside any HTML tag

  `<(\w+)\b[^>]*>(.*?)</\1>`

- Trimming whitespace from the beginning or end of a string

  `^\s+(.*?)\s+$`

- Checking format of a date string (US-style, a.k.a Month-Day-Year)

  `^[01]?\d[-/][0123]?\d[-/]\d\d\d\d$`

Bioinformatics example

First base	Second base: U	Second base: C	Second base: A	Second base: G	Third base
U	UUU, UUC — Phenylalanine F UUA, UUG — Leucine L	UCU, UCC, UCA, UCG — Serine S	UAU, UAC — Tyrosine Y UAA Stop codon UAG Stop codon	UGU, UGC — Cysteine C UGA Stop codon UGG Tryptophan W	U C A G
C	CUU, CUC, CUA, CUG — Leucine L	CCU, CCC, CCA, CCG — Proline P	CAU, CAC — Histidine H CAA, CAG — Glutamine Q	CGU, CGC, CGA, CGG — Arginine R	U C A G
A	AUU, AUC, AUA — Isoleucine I AUG M Methionine start codon	ACU, ACC, ACA, ACG — Threonine T	AAU, AAC — Asparagine N AAA, AAG — Lysine K	AGU, AGC — Serine S AGA, AGG — Arginine R	U C A G
G	GUU, GUC, GUA, GUG — Valine V	GCU, GCC, GCA, GCG — Alanine A	GAU, GAC — Aspartic acid D GAA, GAG — Glutamic acid E	GGU, GGC, GGA, GGG — Glycine G	U C A G

Goal: find one one regular expression that evaluates to true for any Leucine codon.

Regexps aren't just in Python!

- Built into Perl, Ruby, awk, Tcl
- Part of standard library in Python, Java, .NET (C++ only since 2011)
- 3rd party support only in C
- Many text editors, command line utilities have regexp support
  - grep, sed, vim
- Different regexp dialects, but most (including Python) use the Perl dialect

Online `re` learning resources

- Python re documentation
  - http://docs.python.org/2/library/re.html
  - Read this one for homework!
- Python regex HOWTO
  - http://docs.python.org/2/howto/regex.htm
- Quickstarts and full tutorial
  - http://www.regular-expressions.info/
- `import re; help( re )`

Regexps are powerful, but expensive

- Try to use build in `str` methods first
  - ex: `'na' in 'banana'` returns `True`
  - or: `'capital'.replace( 'tal', 'tol' )`
- If you use same regex often, you can compile it for speed:

Terminal Shell Edit View Window Help

Terminal — bash — 86×13

```
chris@NIA-LG-01778617 ~
$ python -m timeit "'hello' in 'hello world'"
10000000 loops, best of 3: 0.0956 usec per loop

chris@NIA-LG-01778617 ~
$ python -m timeit -s "import re" "re.match('hello', 'hello world'}"
1000000 loops, best of 3: 1.94 usec per loop

chris@NIA-LG-01778617 ~
$ python -m timeit -s "import re; h=re.compile('hello'}" "h.match('hello world'}"
1000000 loops, best of 3: 0.489 usec per loop
```

Python regexp workflow

1. Compile the expression into a pattern match object
   - ex: `p = re.compile( 'hello' )`
2. Make method calls on that object with input string as argument
   - `p.match()` – Determine if the RE matches at the beginning of the string
   - `p.search()` – Scan through a string, looking for any location where this RE matches
   - `p.findall()` – Find all substrings where the RE matches, and returns them as a list
   - `p.finditer()` – Find all substrings where the RE matches, and returns them as an iterator
   - `p.sub(`*<replacement>, <string>*`)` – Find all the matches for a pattern, and replace them with a different string
   - ex: `m = p.search( 'hello world' )`
3. Retval from step 2 is a match object or `None` if no match
   - `m.group()` – Return the string matched by the RE
   - `m.start()` – Return the starting position of the match
   - `m.end()` – Return the ending position of the match
   - `m.span()` – Return a tuple containing the (start, end) positions of the match

Simple characters

- Regular expressions can contain both special and ordinary characters.
- Most ordinary characters, like "A", "a", or "0", are the simplest regular expressions; they simply match themselves
- You can concatenate ordinary characters, so `last` matches the string `'last'`.

Fourteen metacharacters

- `[` and `]`
- `\`
- `.`
- `|`
- `^` and `$`
- `*`, `+`, `?`, `{` and `}`
- `(` and `)`

Character Classes: [and]

- Denotes a set of possible character matches
  - ex `'gr[ea]y'` will match “grey” or “gray”
- If you specify ^ as first character inside brackets, it negates the characters inside
  - ex: `'[^GAT]'` matches any character BUT G, A, and T
- Can specify a range of alphanumeric characters using the dash –
  - ex: `'[a-zA-Z]'` matches any upper or lowercase letter
- Within square brackets, most metacharacters are interpreted as literal characters instead of metacharacters

\

1. Defines shorthand character sets:
   - `\w` - any word character
   - `\W` - any non-word character
   - `\d` - any digit
   - `\D` - any non-digit
   - `\s` - any whitespace (space, tab, newline)
   - `\S` - any non-whitespace
   - `\b` – any word boundary
2. Escapes other metacharacters:
   - `\\` - matches a backslash
   - `\.` – matches a dot

Metacharacters, cont.

`Or: |`

- Separates alternate possibilities.
- Can combine two regexps into one

`The dot: .`

- Matches any character EXCEPT for line breaks

Anchors: ^ and $

- ^ matches beginning of string
- $ matches the end of the string
- $ will match the end of any line IF you compile your regexp with the `re.MULTILINE`
  - `p = re.compile( 'er$', re.MULTILINE )`
  - `query_string = "Peter\nPiper"`
  - `p.findall( query_string )` will match twice

Quantification: `*` , `+`, `?`, and `{m,n}`

`*` matches 0 to ∞ times

- ex: `'ab*c'` matches "ac", "abc", "abbc", "abbbc", etc.

`+` matches 1 to ∞ times

- ex: `'ab+c'` matches "abc", "abbc", "abbbc", and so on, but not "ac"

`?` matches 0 or 1 times

- ex: `'colou?r'` matches "color" or "colour"

`{m,n}` means there must be at least *m* repetitions, and at most *n.*

Grouping: (and)

- Look for characters that repeat as a unit
  - ex: `re.search('(na)+', 'banana')` matches 'nana'

Back-referencing

- Use parens to group characters, then use \1 to refer to the first group, \2 to the second, etc.

```
>>> p = re.compile('(\b\w+)\s+\1')
>>> p.search('Paris in the the spring').group()
'the the'
```

Greedy vs. Lazy regexs

- By default, all regexs are greedy.
  - The matching engine goes as far as it can at first, and if no match is found it will then progressively back up and retry the rest of the RE again and again.
- Make a regexp lazy by appending the `?` metacharacter
  - ex: `'[na]+'` matches "anana" in "banana" but:
  - `p.search('[na]+?', "banana")` stops at the first a

In-class exercises

- http://regex.sketchengine.co.uk/extra_regexps.html