Accelerating Information Technology Innovation

http://aiti.mit.edu

Kenya Summer 2011
Lecture 09 – Regular Expressions

MISTI

Regular Expressions

- Regular expressions are a powerful string manipulation tool
- Specify the rules for which you can match strings
- REs are compiled into a series of bytecodes which are then executed by a regex engine
- All modern languages have similar library packages for regular expressions

Examples of Regular Expressions

- "What text is embedded in the <H3> tag?"
- Strings that are valid email addresses.
- Parsing text to extract information

Regular Expressions

- Use regular expressions to:
  - Search a string (`search` and `match`)
  - Replace parts of a string (`sub`)
  - Break stings into smaller pieces (`split`)

Regular Expression Python Syntax

- Most characters match themselves

  The regular expression “test” matches the string `'test'`, and only that string

- [x] matches any *one* of a list of characters

  “[abc]” matches `'a'`, `'b'`, or `'c'`

- [^x] matches any *one* character that is not included in *x*

  “[^abc]” matches any single character *except* `'a'`, `'b'`, or `'c'`

Anchors (Position Characters)

- Anchors allow you to designate where a match can occur
  - ^ → match to beginning of String
    - Example:
      - Pattern: "^[Aa] [Rr]ose"
      - "A Rose is a rose is a rose."
  - $ → match at end of String
    - Example
      - Pattern: "rose$"
      - "A Rose is a rose is a rose"

Anchors (Position Characters)

- \b matches at word boundary:
  - Pattern "\brose" matches "rose" "rosemary", but not "primrose"

Repetition Operators

- Repetition operators allow us to denote that a (sub)pattern may repeat
  - * → zero or more repetitions
    - Example: "0*\d" matches "05" "5" "0006"
  - + → One or more repetitions
    - Example: "de+r" matches "deer" "deeer" "der" not "debr"
  - ? → exactly zero or 1 occurrence
    - Example "de[ae]?r" matches "der" "deer" "dear" not "debr" "deeer"
  - {m, n} → at least *m* occurrences and at most *n* occurances

    “a{2,3}” matches ”aa” or ”aaa" but not ”a" or ”aaaa"

Grouping

- Just like math expressions you can group subpatterns using ( )
  - Pattern "(word)+" matches "word" "wordword" "wordwordword" not "" "wordd"

Example: Valid Email Address

- aiti@mit.edu
  - one or more word characters
  - Followed by '@'
  - Followed by word characters that has to have at least one '.' somewhere
    - Since '.' is an operator in a RE, we need to escape it

Example: Valid Email Address

(\w)+@\w+(\.\w)+

Escaping

- If you want one of the RE reserved characters to appear in your pattern you must escape it:
  - \. → literal . in pattern
  - * → literal * in pattern
  - \ { } + ( ) are the others you must escape

Alternation

- | denotes logical OR operation
- Examples:
  - Pattern "soda|juice" matches "soda" "juice" "soda water", not "water"
  - "\w+@[\w\.]*\.(net|gov|edu)"
    - Good or bad RE for emails?
- | has lowest precedence (applied last)
  - Use ( ) to avoid confusion

Pairs

- “\d” matches any digit; “\D” matches any non-digit
- “\s” matches any whitespace character; “\S” matches any non-whitespace character
- “\w” matches any alphanumeric character; “\W” matches any non-alphanumeric character
- “\b” matches a word boundary; “\B” matches position that is not a word boundary
  A word boundary is a position that changes from a word character to a non-word character, or vice versa.

Search and Match

- The two basic functions are **re.search** and **re.match**
  - Search looks for a pattern anywhere in a string
  - Match looks for a match starting at the beginning
- Both return **None** if the pattern is not found (logical false) and a "match object" if it is

```
>>> pat = "a*b"
>>> import re
>>> re.search(pat,"fooaaabcde")
<_sre.SRE_Match object at 0x809c0>
>>> re.match(pat,"fooaaabcde")
>>>
```

Match object

An instance of the match class with the details of the match result

```
>>>pat = "a*b"
>>>r1 = re.search(pat,"fooaaabcde")
>>>r1.group()  # group returns string matched
'aaab'
>>>r1.start()  # index of the match start
3
>>>r1.end()    # index of the match end
7
>>>r1.span()   # tuple of (start, end)
(3, 7)
```

What got matched?

- We can ‘label’ the groups as well…
  (?P<*label*>*regular expression*)

```
>>> pat ="(?P<name>\w+)@(?P<host>(\w+\.)+(com|org|
   net|edu))"
>>> r2 = re.match(pat,"aiti@mit.edu")
>>> r2.group('name')
'aiti'
>>> r2.group('host')
'mit.edu’
```

Compiling regular expressions

- If you plan to use a re pattern more than once, compile it to a re object
- Python produces a special data structure that speeds up matching

```
>>> pat ="(?P<name>\w+)@(?P<host>(\w+\.)+(com|org|net|
   edu))"
>>> compiled= re.compile(pat)
>>> compiled
<_sre.SRE_Pattern object at 0x2d9c0>
>>> r3 = compiled.search("aiti@mit.edu")
>>> r3
<_sre.SRE_Match object at 0x895a0>
>>> r3.group()
'aiti@mit.edu'
```

Substitution

```
>>> import re
>>> address = 'Ole Sangale Road'
>>> re.sub(’Road$', 'RD.',
  address)
’Ole Sangale RD.'
```

Examples

```
>>> re.sub('\d', 'x', 'a_b - 12')
'a_b - xx'
>>> re.sub('\D', 'x', 'a_b - 12')
'xxxxxxx12'
>>> re.sub('\s', 'x', 'a_b - 12')
'a_bx-x12'
>>> re.sub('\S', 'x', 'a_b - 12')
'xxx x xx'
>>> re.sub('\w', 'x', 'a_b - 12')
'xxx - xx'
>>> re.sub('\W', 'x', 'a_b - 12')
'a_bxxx12'
```

Questions?