Python

regular expressions

Regular Expressions

- Regular expressions are a powerful string manipulation tool
- All modern languages have similar library packages for 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'`

Regular Expressions Syntax

- "." matches any single character
- Parentheses can be used for grouping

  "(abc)+" matches `'abc'`, `'abcabc'`, `'abcabcabc'`, etc.
- *x*|*y* matches *x* or *y*

  "this|that" matches `'this'` and `'that'`, but not `'thisthat'`.

Regular Expression Syntax

- *x** matches zero or more *x*'s
  "a*" matches `''`, `'a'`, `'aa'`, etc.
- *x*+ matches one or more *x*'s
  "a+" matches `'a'`,`'aa'`,`'aaa'`, etc.
- *x*? matches zero or one *x*'s
  "a?" matches `''` or `'a'` .
- *x{m, n}* matches *i x*'s, where *m*≤*i*≤ *n*
  "a{2,3}" matches `'aa'` or `'aaa'`

Regular Expression Syntax

- "\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
- "^" matches the beginning of the string; "$" matches the end of the string
- "\b" matches a word boundary; "\B" matches position that is not a word boundary

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 staring 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")
>>>
```

Q: What's a match object?

- A: 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?

- Here's a pattern to match simple email addresses

  \w+@(\w+\.)+(com|org|net|edu)

```
>>> pat1 = "\w+@(\w+\.)+(com|org|net|edu)"
>>> r1 = re.match(pat,"finin@cs.umbc.edu")
>>> r1.group()
'finin@cs.umbc.edu'
```

- We might want to extract the pattern parts, like the email name and host

What got matched?

- We can put parentheses around groups we want to be able to reference

```
>>> pat2 = "(\w+)@((\w+\.)+(com|org|net|edu))"
>>> r2 = re.match(pat2,"finin@cs.umbc.edu")
>>> r2.group(1)
'finin'
>>> r2.group(2)
'cs.umbc.edu'
>>> r2.groups()
r2.groups()
('finin', 'cs.umbc.edu', 'umbc.', 'edu')
```

- Note that the 'groups' are numbered in a preorder traversal of the forest

What got matched?

- We can 'label' the groups as well…

```
>>> pat3 ="(?P<name>\w+)@(?P<host>(\w+\.)+(com|
org|net|edu))"
>>> r3 = re.match(pat3,"finin@cs.umbc.edu")
>>> r3.group('name')
'finin'
>>> r3.group('host')
'cs.umbc.edu'
```

- And reference the matching parts by the labels

More re functions

- re.split() is like split but can use patterns

```
>>> re.split("\W+", "This... is a test,
  short and sweet, of split().")
['This', 'is', 'a', 'test', 'short',
  'and', 'sweet', 'of', 'split', '']
```

- re.sub substitutes one string for a pattern

```
>>> re.sub('(blue|white|red)', 'black', 'blue
 socks and red shoes')
'black socks and black shoes'
```

- re.findall() finds al matches

```
>>> re.findall("\d+","12 dogs,11 cats, 1 egg")
['12', '11', '10']
```

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

```
>>> capt3 = re.compile(pat3)
>>> cpat3
<_sre.SRE_Pattern object at 0x2d9c0>
>>> r3 = cpat3.search("finin@cs.umbc.edu")
r3 = cpat3.search("finin@cs.umbc.edu")
>>> r3
<_sre.SRE_Match object at 0x895a0>
>>> r3.group()
r3.group()
'finin@cs.umbc.edu'
```

Pattern object methods

- There are methods defined for a pattern object that parallel the regular expression functions, e.g.,
  - match
  - search
  - split
  - findall
  - sub

Example: pig latin

- Rules
  - If word starts with consonant(s)
    - — Move them to the end, append “ay”
  - Else word starts with vowel(s)
    - — Keep as is, but add “zay
  - How might we do this?

piglatin.py

```
import re
pat = r'\b([bcdfghjklmnpqrstvwxyz]+)(\w+)\b'
cpat = re.compile(pat)

def piglatin(string):
  return " ".join( [piglatin1(w) for w in string.split()] )
```

piglatin.py

```
def piglatin1(word):
    match = cpat.match(word)
    if match:
        consonants = match.group(1)
        rest = match.group(2)
        return rest + consonents + “ay”
    else:
        return word + "zay"
```