python™ 2 programming

Advanced Regular Expressions

Advanced Regular Expressions

- Review
- Regular expression quoting
- Modifiers
- Side Effect Variables and capturing
- Minimal matches
- Multi-line matching
- Global matches
- Look-around assertions
- Substitution with interpolation
- Substitution with expressions

Regular expression objects

- `import re` to use them
- We can search or match
  - **search** matches a pattern - like conventional RE's
  - **match** matches the entire string
- Both return a MatchObject, or None (False)

```
testy = 'The quick brown fox jumps over the lazy dog'

m = re.search(r"(quick|slow).*(fox|camel)", testy)
if m:
    print('Matched',m.groups())
    print('Starting at', m.start())
    print('Ending at', m.end())
```

```
Matched ('quick', 'fox')
Starting at 4
Ending at 19
```

Elementary extended RE meta-characters

`.`	match any single character	Character Classes
`[a-zA-Z]`	match any char in the `[…]` set	
`[^a-zA-Z]`	match any char *not* in the `[…]` set	
`^`	match beginning of text	Anchors
`$`	match end of text	
x`?`	match 0 or 1 occurrences of *x*	Quantifiers
x`+`	match 1 or more occurrences of *x*	
x`*`	match 0 or more occurrences of *x*	
x`{`*m*`,`*n*`}`	match between *m* and *n* *x*'s	
`abc`	match `abc`	
`abc|xyz`	match `abc` or `xyz`	Alternation

Compiled Res and re objects

- **We can compile, for efficiency**

```
reobj = re.compile (r"(I).*(\1)")
for line in file:
    m = reobj.match(line)
    if m:
        print(m.string[m.start():m.end()])
```

- **We can also compile to obtain an re object**
  - **Method call parameters are different !**

Flags

- **Change the behaviour of the match**

Long name	Short	RE	
re.ASCII	re.A	(?a)	Class shortcuts do not include Unicode
re.IGNORECASE	re.I	(?i)	Case insensitive match
re.LOCALE	re.L	(?L)	Class shortcuts are locale sensitive
re.MULTILINE	re.M	(?m)	^ and $ match start and end of *line*
re.DOTALL	re.S	(?s)	. also matches a new-line
re.VERBOSE	re.X	(?x)	Whitespace is ignored, allow comments

  - **May be embedded in the RE**
  - **May be specified as an optional argument to**
    - re.search, re.match, and re.compile
    - Multiple flags may be combined

```
m = re.search(r'(?im)^john',name)
m = re.search(r'^john',name,re.IGNORECASE|re.MULTILINE)
```

Global matches

- **re.findall**
  - **Returns a list of matches or groups**

```
str='/dev/sd3d 135398 69683 52176 57% /home/stuff'
nums = re.findall(r'\b\d+\b', str)
print(nums)
```

```
['135398', '69683', '52176', '57']
```

- **re.finditer**
  - **Returns an iterator to a match object**

```
str='/dev/sd3d 135398 69683 52176 57% /home/stuff'

for num in re.finditer(r'\b(\d+)\b', str):
    print(num.groups())
```

```
('135398',)
('69683',)
('52176',)
('57',)
```

Back-references

- Python allows ‘back-references’
  - To create self-referencing regular expressions
- Indicated by *n*, which represents the ‘*n*th’ set of parentheses
- Back-references may be used in the replacement in sub and subn

```
str='copyright 2005-2006'
print(re.sub(
   r'((19|20)[0-9]{2})-((19|20)[0-9]{2})',r'\1-2010',
   str))
```

```
copyright 2005-2010
```

Non-capturing groups

- **A parenthesis group creates group values**
  - **Incurs some run-time overhead for keeping track of this**

```
drink = 'A bottle of Miller'

pattern = \
    'A (glass|bottle|barrel) of (Bud|Miller|Coors)'
m = re.search (pattern, drink)
if m: print(m.groups())
```

```
('bottle', 'Miller')
```

- **`(?:)` parenthesis group without back-references**
  - **Incurs lower run-time overhead**

```
drink = 'A bottle of Miller'

pattern = \
    'A (?:glass|bottle|barrel) of (?:Bud|Miller|Coors)'
m = re.search (pattern, drink)
if m: print(m.groups())
```

```
()
```

Other match values

- `start()`: list of starting offsets of each match
  - First element gives the offset to the start of the first match
- `end()` : list of ending offsets+1 of each match
  - First element gives the offset+1 to the end of the first match
- lastindex : index of last group matched

```
#      0123456789012345678901234567890123456789012345
txt = '2 456 first 3456 second third 98765 fourth 123'
m = re.search(r'(\d+) ([a-z]+) (\d+)',txt)
if m:
   print(m.groups())
   print([m.start(i) for i in range(1,m.lastindex+1)])
   print([m.end(i)   for i in range(1,m.lastindex+1)])
```

```
('456', 'first', '3456')
[2, 6, 12]
[5, 11, 16]
```

Named captures

- Give names to captures, not just boring `m.group()`…

`(?P<name>RE-pattern)`

- Capture names and values are in `m.groupdict()`

```
df = '/dev/sd3d  135398  69683  52176  57%  /home/stuff'
m = re.search('''
        ^(?P<fs>\S+)   \s+ (?:\d+)       \s+ (?:\d+) \s+
         (?P<avail>\d+)\s+ (?P<cap>\d+)% \s+ (?P<mnt>\S+)
              ''', df, re.X)
if m:
    gd = m.groupdict()
    print("{O[mnt]} ({O[fs]}) has {O[avail]} ({O[cap]}%) free".
          format(gd))
```

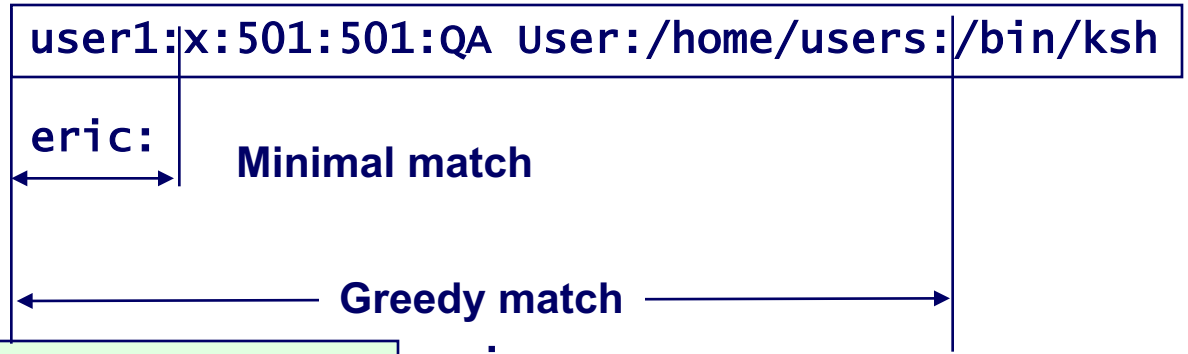
```
/home/stuff (/dev/sd3d) has 52176 (57%) free
```

Minimal matches

- **A minimal match ends with a question mark**
  - **Match as few times as possible**

`+?`	match one or more times
`*?`	match zero or more times
`??`	match 0 or 1 times, preferably 0
`{n}?`	match *n* times (same as `{n}`)
`{n,}?`	match at least *n* times, stop as soon as possible
`{n,m}?`	match *n* to *m* times, stop as soon as possible

```
re.sub('.*?:','eric:',passwd,1)
```




```
re.sub('.*:','eric:',passwd,1)
```

Multi-line matches

- Python allows searching a pattern across multiple lines
  - Just search a text which contains embedded "\n" characters
- However, Python is normally cautious :
  - `.` character class does not match newline
  - `.*` will only go until the end of the line
- `re.S` flag treats "\n" as a normal character
  - `.` matches any character, including newline
  - `.*` will match until the end of the search text
  - `\s` and `\S` shortcuts are not affected

Multi-line matches

- Normally, ^ and $ mean: start and end of search text
- For multiple-line matches, this may be inconvenient
  - You may want to search the start and end of each line in the text
- `re.M` matches ^ and $ for each line within search text
  - Can be combined with `re.S` option

```
all = open('names.txt').read()

m = re.search(r'(^dpm|^james)',all,re.I)
if m:
    print("File starts with",m.groups())

m = re.search(r'(^dpm|^james)',all,re.I|re.M)
if m:
    print("A line in the file starts with",m.groups())
```

```
Fred
DPM
Clive
Tom
Dick
Harry
```

```
A line in the file starts with ('DPM',)
```

Alternatives to ^ and $

- \A matches beginning of string
  - Will not match multiple times with /m
- \Z matches end of string, or before a "\n"
  - The "\n" is optional
- \z matches end of string
  - The real end-of-string, the "\n" is significant

Comments in Regular Expressions

- **Delimit comments with (?#*comment*)**

```
re.search(
  r'\d{3}(?# 3 digits)\s(?# space)\d{3,5}(?# 3-5 digits)',
  txt)
```

- **The `re.X` flag allows comments in REs**
  - **White-space in the regular expression is discarded**
  - **Comments until end-of-line allowed**

```
re.search(
    '''
      \d{3}       # 3 digits
      \s          # space
      \d{3,5}     # 3-5 digits
    ''',
    txt, re.X)
```

Lambda in re.sub

- **The re.sub method can take a function as a replacement**
  - **Passes a match object to the function**
  - **The return value is the value substituted**

```
import re
codes = {}

names = ['zero','wun','two','tree','fower','fife','six','seven',
         'ait','niner','alpha','bravo','charlie','delta','echo',
         'foxtrot','golf','hotel','india','juliet','kilo','lima',
         'mike','november','oscar','papa','quebec','romeo',
         'sierra','tango','uniform','victor','whisky','xray',
         'yankee','zulu']

for key in (range(0,10)):
    codes[str(key)] = names[key]
for key in (range(ord('A'),ord('Z')+1)):
    codes[chr(key)] = names[key - ord('A')+10]

reg = 'WG07 OKD'

result = re.sub(r'(\w)',
                lambda m: codes[m.groups()[0]]+' ', reg)
```

Look-around assertions

- **Do not consume the pattern, or capture**
  - **Look-ahead: Positive: (?=*pattern*) Negative: (?!*pattern*)**
  - **Look-behind: Positive: (?<=*pattern*) Negative: (?<!*pattern*)**

- **Without look-arounds**

```
var = '<h1>This is a header</h1>'

m = re.search(r'<([hH]\d)>.*</\1>',var)
print("Matched: ",m.group())
```

```
Matched: <h1>This is a header</h1>
```

- **With look-arounds**

```
var = '<h1>This is a header</h1>'

m = re.search(r'(?<=<([hH]\d)>).*(?=</\1>)',var)
print("Matched: ",m.group())
```

```
Matched: This is a header
```

Another example

- **Subtract 1 from the 2nd field for:**
  - `.log` **files where the user (3rd field) is** `root`
  - `.dat` **files where the type (4th field) is** `system`

```
for line in open('log.txt'):
   nline = re.sub(
     '(?:(?<=\.log,)(\d+)(?=,root,)) |
      (?:(?<=\.dat,)(\d+)(?=.*,system$))',
     lambda m: str(int(m.group(1))-1) if m.group(1) \
               else str(int(m.group(2))-1),
     line,re.X)

   print(nline, end = "")
```

```
accounts.dat,2,user12,system
apache.log,1682,apache,daemon
var.log,23,root,user
profile.dat,57,home,user
payroll.dat,887,prd,system
```

→

```
accounts.dat,1,user12,system
apache.log,1682,apache,daemon
var.log,22,root,user
profile.dat,57,home,user
payroll.dat,886,prd,system
```

Summary

- Modifiers alter the effect of the RE
- Side effect variables and capturing can be controlled
- Minimal matches use a ? after the quantifier
- Multi-line matching uses re.M and re.S
- Look-around assertions
- Substitution with interpolation
- Substitution with expressions