Regular Expressions: Matching

Regular Expressions: Introduction

"Regexes" define a declarative language used to match patterns in text, and are used for text validation/inspection, and text extraction.

Previously, we have had limited tools for inspecting text:

In the case of *fixed-width* text, we have been able to use a **slice**.

```
line = '19340903  3.4 0.9'
year = line[0:4]                # year == 1934
```

In the case of *delimited* text, we have been able to use **split()**

```
line = '19340903,3.4,0.9'
els = line.split(',')

mkt_rf = els[1]                  # 3.4
```

In the case of *formatted* text, there is no obvious way to do it.

Regular Expressions: Preview

This regex pattern matches elements of a web server log line in a single statement, without resorting to complicated splitting, slicing or string inspection methods.

The bolded portions show log line elements that we wish to extract; we are doing so using the parentheses.

```
import re

log_line = '66.108.19.165 - - [09/Jun/2003:19:56:33 -0400] "GET /~jjk265/cd.jpg HTTP/1.1" 200 175449'

reg = re.search(r'(\d{2,3}\.\d{2,3}\.\d{2,3}\.\d{2,3}) - - \[(\d\d\/\w{3}\/\d{4}):(\d\d:\d\d:\d\d) (\-?\d\d\d\d
print type(reg)

print reg.group(1)   # 66.108.19.165
print reg.group(2)   # 09/Jun/2003
print reg.group(3)   # 19:56:33
print reg.group(4)   # -0400
```

Patterns look for various *classes* of text (for example, numbers or letters), as well as literal characters, in various *quantities*, reading through consecutive characters of the text.

Reading from left to right, the pattern (shown in the **r''** string) says this:

```
  2-3 digits, a period, 2-3 digits, a period, 2-3 digits, a period, 2-3 digits,
  followed by a space, dash, space, dash,
  followed by an open square bracket, 2 digits, forward slash, 3 word characters, forward slash, 4 digits,
  followed by a colon, 2 digits, colon, 2 digits, colon, 2 digits, space
  followed by a dash and 4 digits.
```

The parentheses identify text to be extracted. You can see the text that was extracted in the output.

The *re* module and *re.search()* function

re.search() returns **True** if the pattern matches the text.

```
import re          # import the regex library

if re.search(r'~jjk265', line):
    print line                          # prints any line with the characters ~jjk265
```

re.search() takes two arguments: the *string pattern*, and the string to be searched. Normally used in an **if** expression, it will evaluate to **True** if the pattern matched.

```
# weblog contains string lines like this:
  '66.108.19.165 - - [09/Jun/2003:19:56:33 -0400] "GET /~jjk265/cd.jpg HTTP/1.1" 200 175449'
  '66.108.19.165 - - [09/Jun/2003:19:56:44 -0400] "GET /~dbb212/mysong.mp3 HTTP/1.1" 200 175449'
  '66.108.19.165 - - [09/Jun/2003:19:56:45 -0400] "GET /~jjk265/cd2.jpg HTTP/1.1" 200 175449'

# script snippet:
for line in weblog.readlines():
  if re.search(r'~jjk265', line):
    print line                      # prints 2 of the above lines
```

"not" for negating a search

not is used to negate a search: "if the pattern does not match".

search file

```
  '66.108.19.165 - - [09/Jun/2003:19:56:33 -0400] "GET /~jjk265/cd.jpg HTTP/1.1" 200 175449'
  '66.108.19.165 - - [09/Jun/2003:19:56:44 -0400] "GET /~dbb212/mysong.mp3 HTTP/1.1" 200 175449'
  '66.108.19.165 - - [09/Jun/2003:19:56:45 -0400] "GET /~jjk265/cd2.jpg HTTP/1.1" 200 175449'
```

code

```
for line in weblog.readlines():
    if not re.search(r'~jjk265', line):
        print line                      # prints 1 of the above lines -- the one without jjk265
```

The re Vocabulary

These terms, used in combination, cover most needs in composing text matching patterns.

Anchor Characters and the Boundary Character	$, ^, \b
Character Classes	\w, \d, \s, \W, \S, \D
Custom Character Classes	[aeiou], [a-zA-Z]
The Wildcard	.
Quantifiers	+, *, ?
Custom Quantifiers	{2,3}, {2,}, {2}
Groupings	(parentheses groups)

Patterns can match anywhere, but must match on consecutive characters

Read the string from left to right, looking for the first place the pattern matches on consecutive characters.

```
import re

str1 = 'hello there'
str2 = 'why hello there'
str3 = 'hel lo'

if re.search(r'hello', str1):  print 'matched'   # matched
if re.search(r'hello', str2):  print 'matched'   # matched
if re.search(r'hello', str3):  print 'matched'   # does not match
```

Note that 'hello' matches at the start of the first string and the middle of the second string. But it doesn't match in the third string, even though all the characters we are looking for are there. This is because the space in **str3** is unaccounted for - always remember - matches take place on *consecutive characters*.

Anchors and Boundary

Boundary characters **$** and **^** require that the pattern match starts at the beginning of the string or ends at the end of the string.

This program lists only those files in the directory that end in '.txt':

```
import os, re
for filename in os.listdir(r'/path/to/directory'):
    if re.search(r'\.txt$', filename):     # look for '.txt' at end of filename
        print filename
```

This program prints all the lines in the file that don't begin with a hash mark:

```
for text_line in open(r'/path/to/file.py'):
    if not re.search(r'^#', text_line):        # look for '#' at start of filename
        print text_line
```

When they are used as anchors, we will always expect **^** to appear at the start of our pattern, and **$** to appear at the end.

Character Classes

A *character class* is a special pattern entity can match on any of a group of characters: any of the "digit" class (0-9), any of the "word" class (letters, numbers and underscore), etc.

```
user_input = raw_input('please enter a single-digit integer: ')
if not re.search(r'^\d$', user_input):
    exit('bad input:  exiting...')
```

\d	[0-9]	(Digits)
\w	[a-zA-Z0-9_]	(Word characters -- letters, numbers or underscores)
\s	[\n\t]	('Whitespace' characters -- spaces, newlines, or tabs)

So a **\d** will match on a **5**, **9**, **3**, etc.; a **\w** will match on any of those, or on **a**, **Z**, _ (underscore).

Keep in mind that although they match on any of several characters, a single instance of a character class matches on only one character. For example, a **\d** will match on a single number like '5', but it won't match on both characters in '55'. To match on 55, you could say **\d\d**.

Built-in Character Class: digits

The **\d** character class matches on any digit.

This example lists only those files with names formatted with a particular syntax -- YYYY-MM-DD.txt:

```
import re
dirlist = ('.', '..', '2010-12-15.txt', '2010-12-16.txt', 'testfile.txt')
for filename in dirlist:
    if re.search(r'^\d\d\d\d-\d\d-\d\d\.txt$', filename):
        print filename
```

Here's another example, validation: this regex uses the pattern ^\d\d\d\d$ to check to see that the user entered a four-digit year:

```
import re
answer = raw_input("Enter your birth year in the form YYYY\n")
if re.search(r'^\d\d\d\d$', answer):
    print "Your birth year is ", answer
else:
    print "Sorry, that was not YYYY"
```

Built-in Character Class: "word" characters

The **\w** character class matches on any number, letter or underscore.

In this example, we require the user to enter a username with any word characters:

```
username = raw_input()
if not re.search(r'^\w\w\w\w\w$', username):
    print "use five numbers, letters, or underscores\n"
```

As you can see, the anchors force the match to start at the start of the string and end at the end of the string -- thus matching only on the whole string.

Built-in Character Classes: spaces

The **\s** character class matches on a space, a newline (**\n**) or a tab (**\t**).

This program searches for a space anywhere in the string and if it finds it, the match is successful - which means the input isn't successful:

```
new_password = raw_input()
if re.search(r'\s', new_password):
    print "password must not contain spaces"
```

Note in particular that the regex pattern **\s** is not anchored anywhere. So the regex will match if a space occurs anywhere in the string.

You may also reflect that we treat spaces pretty roughly - always stripping them off. They're always causing problems! And they're invisible, too, and still get in the way. What a nuisance.

Inverse Character Classes

Each built-in has a corresponding "uppercased" character class that matches on *anything but* the original class.

Not a digit: **\D**

\D matches on any character that is not a digit, including letters, underscores, punctuation, etc. This program checks for a non-digit in the user's account number:

```
account_number = raw_input()
if re.search(r'\D', account_number):
    print "account number must be all digits!"
```

Not a word character: **\W**

\W matches on any character that is not a word character, including punctuation and spaces.

```
account_number = raw_input()
if re.search(r'\W', account_number):
    print "account number must be only letters, numbers, and underscores"
```

Not a space character: **\S**

\S matches on any character that is not a space (i.e. letters, numbers, special characters, etc.)

These two regexes check for a non-space at the start and end of the string:

```
sentence = raw_input()
if re.search(r'^\S', sentence) and re.search(r'\S$', sentence):
    print "the sentence does not begin or end with a space, tab or newline."
```

Custom Character Classes

The collection of characters included in a character class can be custom-defined.

Consider this table of character classes and the list of characters they match on:

class designation	members
\d	[0123456789]
\w	[abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOOPQRSTUVWXYZ0123456789_] or [a-zA-Z0-9_]
\s	[\t\n]

In fact, the bracketed ranges can be used to create our own character classes. We simply place members of the class within the brackets and use it in the same way we might use **\d** or the others.

A custom class can contain a *range* of characters. This example looks for letters only (there is no built-in class for letters):

```
import re
input = raw_input("please enter a username, starting with a letter:  ")
if not re.search(r'^[a-zA-Z]', input):
    exit("invalid user name entered")
```

This custom class [.,;:?!] matches on any one of these punctuation characters, and this example identifies single punctuation characters and removes them:

```
import re
text_line = 'Will I?  I will.  Today, tomorrow; yesterday and before that.'
for word in text_line.split():
    while re.search(r'[.,;:?! -]$', word):
        word = word[:-1]
    print word
```

Negative Custom Character Classes

Any customer character class can be "inversed" when preceded by a carat character.

Like **\S** for **\s**, the inverse character class matches on anything *not* in the list. It is designated with a carrot just inside the open bracket:

```
import re
for text_line in open('unknown_text.txt'):
    for word in text_line.split():
        while re.search(r'[^a-zA-Z]$', word):
            word = word[:-1]
        print word
```

It would be easy to confuse the carrot at the start of a string with the carrot at the start of a custom character class -- just keep in mind that one appears at the very start of the string, and the other at the start of the bracketed list.

The Wildcard (.)

The wildcard matches on any character that is not a newline.

```
import re
username = raw_input()
if not re.match(r'^.....$', username):   # five dots here
    print "you can use any characters except newline, but there must \
    be five of them.\n"
```

We might surmise this is because we are often working with line-oriented input, with pesky newlines at the end of every line. Not matching on them means we never have to worry about stripping or watching out for newlines.)

Quantifiers: specifies how many to look for

The quantifier specifies how many of the immediately preceding character may match by the pattern.

We can say three digits (\d{3}), between 1 and 3 word characters (\w{1,3}), one or more letters [a-zA-Z]+, zero or more spaces (\s*), one or more x's (x+). Anything that matches on a character can be quantified.

+	1 or more
*	0 or more

?	0 or 1
{3,10}	between 3 and 10

In this example directory listing, we are interested only in files with the pattern **config**_ followed by an integer of any size. We know that there could be a config_1.txt, a config_12.txt, or a config_120.txt. So, we simply specify "one or more digits":

```
import re
filenames = ['config_1.txt', 'config_10.txt', 'notthis.txt', '.', '..']
wanted_files = []
for file in filenames:
    if re.search(r'^config_\d+\.txt$', file):
        wanted_files.append(file)
```

Here, we validate user input to make sure it matches the pattern for valid NYU ID. The pattern for an NYU Net ID is: two or three letters followed by one or more numbers:

```
import re
input = raw_input("please enter your net id:  ")
if not re.search(r'^[A-Za-z]{2,3}\d+$', input):
    print "that is not valid NYU Net ID!"
```

A *simple* email address is one or more word characters followed by an @ sign, followed by a period, followed by 2-4 letters:

```
import re
email_address = raw_input()
if re.search(r'^\w+@\w+\.[A-Za-z]{2,}$', email_address):
    print "email address validated"
```

Of course email addresses can be more complicated than this - but for this exercise it works well.