Python et RE

Les expressions rationnelles (régulières) en Python

Contenu

What is Regular Expression and how is it used?

https://www.analyticsvidhya.com/blog/2015/06/regular-expression-python/

Simply put, regular expression is a sequence of character(s) mainly used to find and replace patterns in a string or file. As I mentioned before, they are supported by most of the programming languages like python, perl, R, Java and many others. So, learning them helps in multiple ways (more on this later).

Regular expressions use two types of characters:

a) Meta characters: As the name suggests, these characters have a special meaning, similar to * in wild card.

b) Literals (like a,b,1,2…)

In Python, we have module "**re**" that helps with regular expressions. So you need to import library **re** before you can use regular expressions in Python.

```
Use this code --> Import re
```

The most common uses of regular expressions are:

- Search a string (search and match)
- Finding a string (findall)
- Break string into a sub strings (split)
- Replace part of a string (sub)

Let's look at the methods that library "**re**" provides to perform these tasks.

What are various methods of Regular Expressions?

The 're' package provides multiple methods to perform queries on an input string. Here are the most commonly used methods, I will discuss:

1. re.match()
2. re.search()
3. re.findall()
4. re.split()
5. re.sub()
6. re.compile()

Let's look at them one by one.

re.match(*pattern, string*):

This method finds match if it occurs at start of the string. For example, calling match() on the string 'AV Analytics AV' and looking for a pattern 'AV' will match. However, if we look for only Analytics, the pattern will not match. Let's perform it in python now.

Code

```
import re
result = re.match(r'AV', 'AV Analytics Vidhya AV')
print result
```

Output:

```
<_sre.SRE_Match object at 0x0000000009BE4370>
```

Above, it shows that pattern match has been found. To print the matching string we'll use method group (It helps to return the matching string). Use "r" at the start of the pattern string, it designates a python raw string.

```
result = re.match(r'AV', 'AV Analytics Vidhya AV')
print result.group(0)
```

Output:

```
AV
```

Let's now find 'Analytics' in the given string. Here we see that string is not starting with 'AV' so it should return no match. Let's see what we get:

Code

```
result = re.match(r'Analytics', 'AV Analytics Vidhya AV')
print result
```

Output:

```
None
```

There are methods like start() and end() to know the start and end position of matching pattern in the string.

Code

```
result = re.match(r'AV', 'AV Analytics Vidhya AV')
print result.start()
print result.end()
```

Output:

```
0
2
```

Above you can see that start and end position of matching pattern 'AV' in the string and sometime it helps a lot while performing manipulation with the string.

re.search(*pattern, string*):

It is similar to match() but it doesn't restrict us to find matches at the beginning of the string only. Unlike previous method, here searching for pattern 'Analytics' will return a match.

Code

```
result = re.search(r'Analytics', 'AV Analytics Vidhya AV')
print result.group(0)
```

Output:

```
Analytics
```

Here you can see that, search() method is able to find a pattern from any position of the string but it only returns the first occurrence of the search pattern.

re.findall (*pattern, string*):

It helps to get a list of all matching patterns. It has no constraints of searching from start or end. If we will use method findall to search 'AV' in given string it will return both occurrence of AV. While searching a string, I would recommend you to use **re.findall()** always, it can work like re.search() and re.match() both.

Code

```
result = re.findall(r'AV', 'AV Analytics Vidhya AV')
print result
```

Output:

```
['AV', 'AV']
```

re.split(*pattern, string,* [*maxsplit=0*]):

This methods helps to split *string* by the occurrences of given *pattern*.

Code

```
result=re.split(r'y','Analytics')
result
```

Output:

```
['Anal', 'tics']
```

Above, we have split the string "Analytics" by "y". Method split() has another argument "**maxsplit**". It has default value of zero. In this case it does the maximum splits that can be done, but if we give value to maxsplit, it will split the string. Let's look at the example below:

Code

```
result=re.split(r'i','Analytics Vidhya')
print result
```

Output:

```
['Analyt', 'cs V', 'dhya'] #It has performed all the splits that can be
done by pattern "i".
```

Code

```
result=re.split(r'i','Analytics Vidhya',maxsplit=1)
result
```

Output:

```
['Analyt', 'cs Vidhya']
```

Here, you can notice that we have fixed the maxsplit to 1. And the result is, it has only two values whereas first example has three values.

re.sub(*pattern, repl, string*):

It helps to search a pattern and replace with a new sub string. If the pattern is not found, *string* is returned unchanged.

Code

```
result=re.sub(r'India','the World','AV is largest Analytics community of
India')
result
```

Output:

```
'AV is largest Analytics community of the World'
```

re.compile(*pattern, repl, string*):

We can combine a regular expression pattern into pattern objects, which can be used for pattern matching. It also helps to search a pattern again without rewriting it.

Code

```
import re
pattern=re.compile('AV')
result=pattern.findall('AV Analytics Vidhya AV')
print result
result2=pattern.findall('AV is largest analytics community of India')
print result2
```

Output:

```
['AV', 'AV']
['AV']
```

Quick Recap of various methods:

Till now, we looked at various methods of regular expression using a constant pattern (fixed characters). But, what if we do not have a constant search pattern and we want to return specific set of characters (defined by a rule) from a string? Don't be intimidated.

This can easily be solved by defining an expression with the help of pattern operators (meta and literal characters). Let's look at the most common pattern operators.

What are the most commonly used operators?

Regular expressions can specify patterns, not just fixed characters. Here are the most commonly used operators that helps to generate an expression to represent required characters in a string or file. It is commonly used in web scrapping and text mining to extract required information.

Operators	Description
.	Matches with any single character except newline '\n'.
?	match 0 or 1 occurrence of the pattern to its left
+	1 or more occurrences of the pattern to its left
*	0 or more occurrences of the pattern to its left
\w	Matches with a alphanumeric character whereas \W (upper case W) matches non alphanumeric ch
\d	Matches with digits [0-9] and /D (upper case D) matches with non-digits.
\s	Matches with a single white space character (space, newline, return, tab, form) and \S (upper case non-white space character.
\b	boundary between word and non-word and /B is opposite of /b
[..]	Matches any single character in a square bracket and [^..] matches any single character not in squa
\	It is used for special meaning characters like \. to match a period or \+ for plus sign.
^ and $	^ and $ match the start or end of the string respectively
{n,m}	Matches at least n and at most m occurrences of preceding expression if we write it as {,m} then it

	any minimum occurrence to max m preceding expression.
a| b	Matches either a or b
( )	Groups regular expressions and returns matched text
\t, \n, \r	Matches tab, newline, return

For more details on meta characters "(", ")","|" and others details , you can refer this link (https://docs.python.org/2/library/re.html).

Now, let's understand the pattern operators by looking at the below examples.

Some Examples of Regular Expressions

Problem 1: Return the first word of a given string

Solution-1 Extract each character **(**using "**\w**"**)**

Code

```
import re
result=re.findall(r'.','AV is largest Analytics community of India')
print result
```

Output:

```
['A', 'V', ' ', 'i', 's', ' ', 'l', 'a', 'r', 'g', 'e', 's', 't', ' ', 'A',
'n', 'a', 'l', 'y', 't', 'i', 'c', 's', ' ', 'c', 'o', 'm', 'm', 'u', 'n',
'i', 't', 'y', ' ', 'o', 'f', ' ', 'I', 'n', 'd', 'i', 'a']
```

Above, space is also extracted, now to avoid it use "**\w**" instead of "**.**".

Code

```
result=re.findall(r'\w','AV is largest Analytics community of India')
print result
```

Output:

```
['A', 'V', 'i', 's', 'l', 'a', 'r', 'g', 'e', 's', 't', 'A', 'n', 'a', 'l',
'y', 't', 'i', 'c', 's', 'c', 'o', 'm', 'm', 'u', 'n', 'i', 't', 'y', 'o',
'f', 'I', 'n', 'd', 'i', 'a']
```

Solution-2 Extract each word **(**using "*****" or "**+**"**)**

Code

```
result=re.findall(r'\w*','AV is largest Analytics community of India')
print result
```

Output:

```
['AV', '', 'is', '', 'largest', '', 'Analytics', '', 'community', '', 'of',
'', 'India', '']
```

Again, it is returning space as a word because "*****" returns zero or more matches of pattern to its left. Now to remove spaces we will go with "**+**".

Code

```
result=re.findall(r'\w+','AV is largest Analytics community of India')
print result
```

Output:

```
['AV', 'is', 'largest', 'Analytics', 'community', 'of', 'India']
```

Solution-3 Extract each word **(**using "^"**)**

Code

```
result=re.findall(r'^\w+','AV is largest Analytics community of India')
print result
```

Output:

```
['AV']
```

If we will use "$" instead of "^", it will return the word from the end of the string. Let's look at it.

Code

```
result=re.findall(r'\w+$','AV is largest Analytics community of India')
print result
```

Output:

['India']

Problem 2: Return the first two character of each word

Solution-1 Extract consecutive two characters of each word, excluding spaces (using "**\w**")

Code

```
result=re.findall(r'\w\w','AV is largest Analytics community of India')
print result
```

```
Output:
```

```
['AV', 'is', 'la', 'rg', 'es', 'An', 'al', 'yt', 'ic', 'co', 'mm', 'un',
'it', 'of', 'In', 'di']
```

Solution-2 Extract consecutive two characters those available at start of word boundary (using "**\b**")

Code

```
result=re.findall(r'\b\w.','AV is largest Analytics community of India')
print result
```

Output:

```
['AV', 'is', 'la', 'An', 'co', 'of', 'In']
```

Problem 3: Return the domain type of given email-ids

To explain it in simple manner, I will again go with a stepwise approach:

Solution-1 Extract all characters after "@"

Code

```
result=re.findall(r'@\w+','abc.test@gmail.com, xyz@test.in,
test.first@analyticsvidhya.com, first.test@rest.biz')
print result
```

Output: `['@gmail', '@test', '@analyticsvidhya', '@rest']`

Above, you can see that ".com", ".in" part is not extracted. To add it, we will go with below code.

```
result=re.findall(r'@\w+.\w+','abc.test@gmail.com, xyz@test.in,
test.first@analyticsvidhya.com, first.test@rest.biz')
print result
```

Output:

```
['@gmail.com', '@test.in', '@analyticsvidhya.com', '@rest.biz']
```

Solution – 2 Extract only domain name using "( )"

Code

```
result=re.findall(r'@\w+.(\w+)','abc.test@gmail.com, xyz@test.in,
test.first@analyticsvidhya.com, first.test@rest.biz')
print result
```

Output:

```
['com', 'in', 'com', 'biz']
```

Problem 4: Return date from given string

Here we will use "**\d**" to extract digit.

Solution:

Code

```
result=re.findall(r'\d{2}-\d{2}-\d{4}','Amit 34-3456 12-05-2007, XYZ 56-
4532 11-11-2011, ABC 67-8945 12-01-2009')
print result
```
Output:
```
['12-05-2007', '11-11-2011', '12-01-2009']
```

If you want to extract only year again parenthesis "( )" will help you.

Code

```
result=re.findall(r'\d{2}-\d{2}-(\d{4})','Amit 34-3456 12-05-2007, XYZ 56-
4532 11-11-2011, ABC 67-8945 12-01-2009')
print result
```
Output:
```
['2007', '2011', '2009']
```

Problem 5: Return all words of a string those starts with vowel

Solution-1 Return each words

Code

```
result=re.findall(r'\w+','AV is largest Analytics community of India')
print result
```

Output:
```
['AV', 'is', 'largest', 'Analytics', 'community', 'of', 'India']
```

Solution-2 Return words starts with alphabets (using [])

Code

```
result=re.findall(r'[aeiouAEIOU]\w+','AV is largest Analytics community of
India')
print result
```

Output:
```
['AV', 'is', 'argest', 'Analytics', 'ommunity', 'of', 'India']
```
Above you can see that it has returned "argest" and "ommunity" from the mid of words. To drop these two, we need to use "\b" for word boundary.

Solution- 3

Code

```
result=re.findall(r'\b[aeiouAEIOU]\w+','AV is largest Analytics community
of India')
print result
```

Output:
```
['AV', 'is', 'Analytics', 'of', 'India']
```

In similar ways, we can extract words those starts with constant using "^" within square bracket.

Code

```
result=re.findall(r'\b[^aeiouAEIOU]\w+','AV is largest Analytics community
of India')
print result
```

Output:

```
[' is', ' largest', ' Analytics', ' community', ' of', ' India']
```

Above you can see that it has returned words starting with space. To drop it from output, include space in square bracket[].

Code

```
result=re.findall(r'\b[^aeiouAEIOU ]\w+','AV is largest Analytics community
of India')
print result
```

Output:

```
['largest', 'community']
```

Problem 6: Validate a phone number (phone number must be of 10 digits and starts with 8 or 9)

We have a list phone numbers in list "li" and here we will validate phone numbers using regular

Solution

Code

```
import re
li=['9999999999','999999-999','99999x9999']
for val in li:
 if re.match(r'[8-9]{1}[0-9]{9}',val) and len(val) == 10:
     print 'yes'
 else:
     print 'no'
```

Output:

```
yes
no
no
```

Problem 7: Split a string with multiple delimiters

Solution

Code

```
import re
line = 'asdf fjdk;afed,fjek,asdf,foo' # String has multiple delimiters
(";",",","," ").
result= re.split(r'[;,\s]', line)
print result
```

Output:

```
['asdf', 'fjdk', 'afed', 'fjek', 'asdf', 'foo']
```

We can also use method **re.sub()** to replace these multiple delimiters with one as space " ".

Code

```
import re
line = 'asdf fjdk;afed,fjek,asdf,foo'
result= re.sub(r'[;,\s]',' ', line)
print result
```

Output:

```
asdf fjdk afed fjek asdf foo
```

Problem 8: Retrieve Information from HTML file

I want to extract information from a HTML file (see below sample data). Here we need to extract information available between <td> and </td> except the first numerical index. I have assumed here that below html code is stored in a string **str**.

Sample HTML file (str)

```
<tr align="center"><td>1</td> <td>Noah</td> <td>Emma</td></tr>
<tr align="center"><td>2</td> <td>Liam</td> <td>Olivia</td></tr>
<tr align="center"><td>3</td> <td>Mason</td> <td>Sophia</td></tr>
<tr align="center"><td>4</td> <td>Jacob</td> <td>Isabella</td></tr>
<tr align="center"><td>5</td> <td>William</td> <td>Ava</td></tr>
<tr align="center"><td>6</td> <td>Ethan</td> <td>Mia</td></tr>
<tr align="center"><td>7</td> <td HTML>Michael</td> <td>Emily</td></tr>
```

Solution:

Code

```
result=re.findall(r'<td>\w+</td>\s<td>(\w+)</td>\s<td>(\w+)</td>',str)
print result
```

Output:

```
[('Noah', 'Emma'), ('Liam', 'Olivia'), ('Mason', 'Sophia'), ('Jacob',
'Isabella'), ('William', 'Ava'), ('Ethan', 'Mia'), ('Michael', 'Emily')]
```

You can read html file using library urllib2 (see below code).

Code

```
import urllib2
response = urllib2.urlopen('')
html = response.read()
```

Using Regular Expressions in Python

https://regexone.com/references/python

If you need a refresher on how Regular Expressions work, check out our Interactive Tutorial first!

Python supports regular expressions through the standard python library `re` which is bundled with every Python installation. While this library isn't completely PCRE compatible, it supports the majority of common use cases for regular expressions.

Raw Python strings

When writing regular expression in Python, it is recommended that you use raw strings instead of regular Python strings. Raw strings begin with a special prefix (`r`) and signal Python not to interpret backslashes and special metacharacters in the string, allowing you to pass them through directly to the regular expression engine.

This means that a pattern like `"\n\w"` will not be interpreted and can be written as `r"\n\w"` instead of `"\\n\\w"` as in other languages, which is much easier to read.

Matching a string

The `re` package has a number of top level methods, and to test whether a regular expression matches a specific string in Python, you can use `re.search()`. This method either returns `None` if the pattern doesn't match, or a `re.MatchObject` with additional information about which part of the string the match was found.

Note that this method stops after the first match, so this is best suited for testing a regular expression more than extracting data.

Method

```
matchObject = re.search(pattern, input_str, flags=0)
```

Example

```
import re # Lets use a regular expression to match a date string. Ignore # the output since we are just testing if the regex matches. regex = r"([a-zA-Z]+) (\d+)" if re.search(regex, "June 24"): # Indeed, the expression "([a-zA-Z]+) (\d+)" matches the date string      # If we want, we can use the MatchObject's start() and end() methods # to retrieve where the pattern matches in the input string, and the # group() method to get all the matches and captured groups. match = re.search(regex, "June 24")      # This will print [0, 7), since it matches at the beginning and end of the # string print "Match at index %s, %s" % (match.start(), match.end())      # The groups contain the matched values. In particular: # match.group(0)
```

```
always returns the fully matched string # match.group(1) match.group(2),
... will return the capture # groups in order from left to right in the
input string # match.group() is equivalent to match.group(0)      # So this
will print "June 24" print "Full match: %s" % (match.group(0)) # So this
will print "June" print "Month: %s" % (match.group(1)) # So this will print
"24" print "Day: %s" % (match.group(2)) else: # If re.search() does not
match, then None is returned print "The regex pattern does not match. :("
```

Capturing groups

Unlike the `re.search()` method above, we can use `re.findall()` to perform a global search over the whole input string. If there are capture groups in the pattern, then it will return a list of all the captured data, but otherwise, it will just return a list of the matches themselves, or an empty list if no matches are found.

If you need additional context for each match, you can use `re.finditer()` which instead returns an iterator of `re.MatchObjects` to walk through. Both methods take the same parameters.

Method

```
matchList = re.findall(pattern, input_str, flags=0)

matchList = re.finditer(pattern, input_str, flags=0)
```

Example

```
import re # Lets use a regular expression to match a few date strings.
regex = r"[a-zA-Z]+ \d+" matches = re.findall(regex, "June 24, August 9,
Dec 12") for match in matches: # This will print: # June 24 # August 9 #
Dec 12 print "Full match: %s" % (match) # To capture the specific months of
each date we can use the following pattern regex = r"([a-zA-Z]+) \d+"
matches = re.findall(regex, "June 24, August 9, Dec 12") for match in
matches: # This will now print: # June # August # Dec print "Match month:
%s" % (match) # If we need the exact positions of each match regex = r"([a-
zA-Z]+) \d+" matches = re.finditer(regex, "June 24, August 9, Dec 12") for
match in matches: # This will now print: # 0 7 # 9 17 # 19 25 # which
corresponds with the start and end of each match in the input string print
"Match at index: %s, %s" % (match.start(), match.end())
```

Finding and replacing strings

Another common task is to find and replace a part of a string using regular expressions, for example, to replace all instances of an old email domain, or to swap the order of some text. You can do this in Python with the `re.sub()` method.

The optional `count` argument is the exact number of replacements to make in the input string, and if this is value is less than or equal to zero, then every match in the string is replaced.

Method

```
replacedString = re.sub(pattern, replacement_pattern, input_str, count, flags=0)
```

Example

```
import re # Lets try and reverse the order of the day and month in a date # string. Notice how the replacement string also contains metacharacters # (the back references to the captured groups) so we use a raw # string for that as well. regex = r"([a-zA-Z]+) (\d+)" # This will reorder the string and print: # 24 of June, 9 of August, 12 of Dec print re.sub(regex, r"\2 of \1", "June 24, August 9, Dec 12")
```

`re` Flags

In the Python regular expression methods above, you will notice that each of them also take an optional `flags` argument. Most of the available flags are a convenience and can be written into the into the regular expression itself directly, but some can be useful in certain cases.

- `re.IGNORECASE` makes the pattern case insensitive so that it matches strings of different capitalizations
- `re.MULTILINE` is necessary if your input string has newline characters (\n) and allows the start and end metacharacter (^ and $ respectively) to match at the beginning and end of each line instead of at the beginning and end of the whole input string
- `re.DOTALL` allows the dot (.) metacharacter match all characters, including the newline character (\n)

Compiling a pattern for performance

In Python, creating a new regular expression pattern to match many strings can be slow, so it is recommended that you compile them if you need to be testing or extracting information from many input strings using the same expression. This method returns a `re.RegexObject`.

```
regexObject = re.compile(pattern, flags=0)
```

The returned object has exactly the same methods as above, except that they take the input string and no longer require the pattern or flags for each call.

```
import re # Lets create a pattern and extract some information with it regex = re.compile(r"(\w+) World") result = regex.search("Hello World is the easiest") if result: # This will print: # 0 11 # for the start and end of the match print result.start(), result.end() # This will print: # Hello # Bonjour # for each of the captured groups that matched for result in regex.findall("Hello World, Bonjour World"): print result # This will substitute "World" with "Earth" and print: # Hello Earth print regex.sub(r"\1 Earth", "Hello World")
```

Links

For more information about using regular expressions in Python, please visit the following links:

- Python Documentation for Regular Expressions
- Python Compatible Regex Tester

Python Regular Expressions

https://developers.google.com/edu/python/regular-expressions

Regular expressions are a powerful language for matching text patterns. This page gives a basic introduction to regular expressions themselves sufficient for our Python exercises and shows how regular expressions work in Python. The Python "re" module provides regular expression support.

In Python a regular expression search is typically written as:

```
  match = re.search(pat, str)
```

The re.search() method takes a regular expression pattern and a string and searches for that pattern within the string. If the search is successful, search() returns a match object or None otherwise. Therefore, the search is usually immediately followed by an if-statement to test if the search succeeded, as shown in the following example which searches for the pattern 'word:' followed by a 3 letter word (details below):

```
str = 'an example word:cat!!'
match = re.search(r'word:\w\w\w', str)
# If-statement after search() tests if it succeeded
  if match:
    print 'found', match.group() ## 'found word:cat'
  else:
    print 'did not find'
```

The code `match = re.search(pat, str)` stores the search result in a variable named "match". Then the if-statement tests the match -- if true the search succeeded and match.group() is the matching text (e.g. 'word:cat'). Otherwise if the match is false (None to be more specific), then the search did not succeed, and there is no matching text.

The 'r' at the start of the pattern string designates a python "raw" string which passes through backslashes without change which is very handy for regular expressions (Java needs this feature badly!). I recommend that you always write pattern strings with the 'r' just as a habit.

Basic Patterns

The power of regular expressions is that they can specify patterns, not just fixed characters. Here are the most basic patterns which match single chars:

- a, X, 9, < -- ordinary characters just match themselves exactly. The meta-characters which do not match themselves because they have special meanings are: . ^ $ * + ? { [ ] \ | ( ) (details below)
- . (a period) -- matches any single character except newline '\n'
- \w -- (lowercase w) matches a "word" character: a letter or digit or underbar [a-zA-Z0-9_]. Note that although "word" is the mnemonic for this, it only matches a single word char, not a whole word. \W (upper case W) matches any non-word character.

- \b -- boundary between word and non-word
- \s -- (lowercase s) matches a single whitespace character -- space, newline, return, tab, form [\n\r\t\f]. \S (upper case S) matches any non-whitespace character.
- \t, \n, \r -- tab, newline, return
- \d -- decimal digit [0-9] (some older regex utilities do not support but \d, but they all support \w and \s)
- ^ = start, $ = end -- match the start or end of the string
- \ -- inhibit the "specialness" of a character. So, for example, use \. to match a period or \\ to match a slash. If you are unsure if a character has special meaning, such as '@', you can put a slash in front of it, \@, to make sure it is treated just as a character.

Basic Examples

Joke: what do you call a pig with three eyes? piiig!

The basic rules of regular expression search for a pattern within a string are:

- The search proceeds through the string from start to end, stopping at the first match found
- All of the pattern must be matched, but not all of the string
- If `match = re.search(pat, str)` is successful, match is not None and in particular match.group() is the matching text

```
  ## Search for pattern 'iii' in string 'piiig'.
  ## All of the pattern must match, but it may appear anywhere.
  ## On success, match.group() is matched text.
  match = re.search(r'iii', 'piiig') =>  found, match.group() == "iii"
  match = re.search(r'igs', 'piiig') =>  not found, match == None

  ## . = any char but \n
  match = re.search(r'..g', 'piiig') =>  found, match.group() == "iig"

  ## \d = digit char, \w = word char
  match = re.search(r'\d\d\d', 'p123g') =>  found, match.group() == "123"
  match = re.search(r'\w\w\w', '@@abcd!!') =>  found, match.group() ==
"abc"
```

Repetition

Things get more interesting when you use + and * to specify repetition in the pattern

- + -- 1 or more occurrences of the pattern to its left, e.g. 'i+' = one or more i's
- * -- 0 or more occurrences of the pattern to its left
- ? -- match 0 or 1 occurrences of the pattern to its left

Leftmost & Largest

First the search finds the leftmost match for the pattern, and second it tries to use up as much of the string as possible -- i.e. + and * go as far as possible (the + and * are said to be "greedy").

Repetition Examples

```
  ## i+ = one or more i's, as many as possible.
  match = re.search(r'pi+', 'piiig') =>  found, match.group() == "piii"

  ## Finds the first/leftmost solution, and within it drives the +
  ## as far as possible (aka 'leftmost and largest').
  ## In this example, note that it does not get to the second set of i's.
  match = re.search(r'i+', 'piigiiii') =>  found, match.group() == "ii"

  ## \s* = zero or more whitespace chars
  ## Here look for 3 digits, possibly separated by whitespace.
  match = re.search(r'\d\s*\d\s*\d', 'xx1 2   3xx') =>  found,
match.group() == "1 2   3"
  match = re.search(r'\d\s*\d\s*\d', 'xx12  3xx') =>  found, match.group()
== "12  3"
  match = re.search(r'\d\s*\d\s*\d', 'xx123xx') =>  found, match.group() ==
"123"

  ## ^ = matches the start of string, so this fails:
  match = re.search(r'^b\w+', 'foobar') =>  not found, match == None
  ## but without the ^ it succeeds:
  match = re.search(r'b\w+', 'foobar') =>  found, match.group() == "bar"
```

Emails Example

Suppose you want to find the email address inside the string 'xyz alice-b@google.com purple monkey'. We'll use this as a running example to demonstrate more regular expression features. Here's an attempt using the pattern r'\w+@\w+':

```
  str = 'purple alice-b@google.com monkey dishwasher'
  match = re.search(r'\w+@\w+', str)
  if match:
    print match.group()  ## 'b@google'
```

The search does not get the whole email address in this case because the \w does not match the '-' or '.' in the address. We'll fix this using the regular expression features below.

Square Brackets

Square brackets can be used to indicate a set of chars, so [abc] matches 'a' or 'b' or 'c'. The codes \w, \s etc. work inside square brackets too with the one exception that dot (.) just means a literal dot. For the emails problem, the square brackets are an easy way to add '.' and '-' to the set of chars which can appear around the @ with the pattern r'[\w.-]+@[\w.-]+' to get the whole email address:

```
  match = re.search(r'[\w.-]+@[\w.-]+', str)
  if match:
    print match.group()  ## 'alice-b@google.com'
```

(More square-bracket features) You can also use a dash to indicate a range, so [a-z] matches all lowercase letters. To use a dash without indicating a range, put the dash last, e.g. [abc-]. An up-hat (^) at the start of a square-bracket set inverts it, so [^ab] means any char except 'a' or 'b'.

Group Extraction

The "group" feature of a regular expression allows you to pick out parts of the matching text. Suppose for the emails problem that we want to extract the username and host separately. To do this, add parenthesis ( ) around the username and host in the pattern, like this: r'([\w.-]+)@([\w.-]+)'. In this case, the parenthesis do not change what the pattern will match, instead they establish logical "groups" inside of the match text. On a successful search, match.group(1) is the match text corresponding to the 1st left parenthesis, and match.group(2) is the text corresponding to the 2nd left parenthesis. The plain match.group() is still the whole match text as usual.

```
  str = 'purple alice-b@google.com monkey dishwasher'
  match = re.search('([\w.-]+)@([\w.-]+)', str)
  if match:
    print match.group()   ## 'alice-b@google.com' (the whole match)
    print match.group(1)  ## 'alice-b' (the username, group 1)
    print match.group(2)  ## 'google.com' (the host, group 2)
```

A common workflow with regular expressions is that you write a pattern for the thing you are looking for, adding parenthesis groups to extract the parts you want.

findall

findall() is probably the single most powerful function in the re module. Above we used re.search() to find the first match for a pattern. findall() finds *all* the matches and returns them as a list of strings, with each string representing one match.

```
  ## Suppose we have a text with many email addresses
  str = 'purple alice@google.com, blah monkey bob@abc.com blah dishwasher'

  ## Here re.findall() returns a list of all the found email strings
  emails = re.findall(r'[\w\.-]+@[\w\.-]+', str) ## ['alice@google.com',
'bob@abc.com']
  for email in emails:
    # do something with each found email string
    print email
```

findall With Files

For files, you may be in the habit of writing a loop to iterate over the lines of the file, and you could then call findall() on each line. Instead, let findall() do the iteration for you -- much better! Just feed the whole file text into findall() and let it return a list of all the matches in a single step (recall that f.read() returns the whole text of a file in a single string):

```
  # Open file
  f = open('test.txt', 'r')
  # Feed the file text into findall(); it returns a list of all the found
strings
  strings = re.findall(r'some pattern', f.read())
```

findall and Groups

The parenthesis ( ) group mechanism can be combined with findall(). If the pattern includes 2 or more parenthesis groups, then instead of returning a list of strings, findall() returns a list of

tuples. Each tuple represents one match of the pattern, and inside the tuple is the group(1), group(2) .. data. So if 2 parenthesis groups are added to the email pattern, then findall() returns a list of tuples, each length 2 containing the username and host, e.g. ('alice', 'google.com').

```
str = 'purple alice@google.com, blah monkey bob@abc.com blah dishwasher'
tuples = re.findall(r'([\w\.-]+)@([\w\.-]+)', str)
print tuples  ## [('alice', 'google.com'), ('bob', 'abc.com')]
for tuple in tuples:
  print tuple[0]  ## username
  print tuple[1]  ## host
```

Once you have the list of tuples, you can loop over it to do some computation for each tuple. If the pattern includes no parenthesis, then findall() returns a list of found strings as in earlier examples. If the pattern includes a single set of parenthesis, then findall() returns a list of strings corresponding to that single group. (Obscure optional feature: Sometimes you have paren ( ) groupings in the pattern, but which you do not want to extract. In that case, write the parens with a ?: at the start, e.g. (?:) and that left paren will not count as a group result.)

RE Workflow and Debug

Regular expression patterns pack a lot of meaning into just a few characters , but they are so dense, you can spend a lot of time debugging your patterns. Set up your runtime so you can run a pattern and print what it matches easily, for example by running it on a small test text and printing the result of findall(). If the pattern matches nothing, try weakening the pattern, removing parts of it so you get too many matches. When it's matching nothing, you can't make any progress since there's nothing concrete to look at. Once it's matching too much, then you can work on tightening it up incrementally to hit just what you want.

Options

The re functions take options to modify the behavior of the pattern match. The option flag is added as an extra argument to the search() or findall() etc., e.g. re.search(pat, str, re.IGNORECASE).

- IGNORECASE -- ignore upper/lowercase differences for matching, so 'a' matches both 'a' and 'A'.
- DOTALL -- allow dot (.) to match newline -- normally it matches anything but newline. This can trip you up -- you think .* matches everything, but by default it does not go past the end of a line. Note that \s (whitespace) includes newlines, so if you want to match a run of whitespace that may include a newline, you can just use \s*
- MULTILINE -- Within a string made of many lines, allow ^ and $ to match the start and end of each line. Normally ^/$ would just match the start and end of the whole string.

Greedy vs. Non-Greedy (optional)

This is optional section which shows a more advanced regular expression technique not needed for the exercises.

Suppose you have text with tags in it: foo and so on

Suppose you are trying to match each tag with the pattern '(<.*>)' -- what does it match first?

The result is a little surprising, but the greedy aspect of the .* causes it to match the whole 'foo and so on' as one big match. The problem is that the .* goes as far as is it can, instead of stopping at the first > (aka it is "greedy").

There is an extension to regular expression where you add a ? at the end, such as .*? or .+?, changing them to be non-greedy. Now they stop as soon as they can. So the pattern '(<.*?>)' will get just '' as the first match, and '' as the second match, and so on getting each <..> pair in turn. The style is typically that you use a .*?, and then immediately its right look for some concrete marker (> in this case) that forces the end of the .*? run.

The *? extension originated in Perl, and regular expressions that include Perl's extensions are known as Perl Compatible Regular Expressions -- pcre. Python includes pcre support. Many command line utils etc. have a flag where they accept pcre patterns.

An older but widely used technique to code this idea of "all of these chars except stopping at X" uses the square-bracket style. For the above you could write the pattern, but instead of .* to get all the chars, use [^>]* which skips over all characters which are not > (the leading ^ "inverts" the square bracket set, so it matches any char not in the brackets).

Substitution (optional)

The re.sub(pat, replacement, str) function searches for all the instances of pattern in the given string, and replaces them. The replacement string can include '\1', '\2' which refer to the text from group(1), group(2), and so on from the original matching text.

Here's an example which searches for all the email addresses, and changes them to keep the user (\1) but have yo-yo-dyne.com as the host.

```
  str = 'purple alice@google.com, blah monkey bob@abc.com blah dishwasher'
  ## re.sub(pat, replacement, str) -- returns new string with all
replacements,
  ## \1 is group(1), \2 group(2) in the replacement
  print re.sub(r'([\w\.-]+)@([\w\.-]+)', r'\1@yo-yo-dyne.com', str)
  ## purple alice@yo-yo-dyne.com, blah monkey bob@yo-yo-dyne.com blah
dishwasher
```

Exercise

To practice regular expressions, see the Baby Names Exercise.

Use Regular Expressions with Python – Regex support

http://www.regular-expressions.info/python.html

Python's re Module

Python is a high level open source scripting language. Python's built-in "re" module provides excellent support for regular expressions, with a modern and complete regex flavor. The only significant features missing from Python's regex syntax are atomic grouping, possessive quantifiers, and Unicode properties.

The first thing to do is to import the regexp module into your script with `import re`.

Regex Search and Match

Call `re.search(regex, subject)` to apply a regex pattern to a subject string. The function returns `None` if the matching attempt fails, and a `Match` object otherwise. Since `None` evaluates to `False`, you can easily use `re.search()` in an `if` statement. The `Match` object stores details about the part of the string matched by the regular expression pattern.

You can set regex matching modes by specifying a special constant as a third parameter to `re.search()`. `re.I` or `re.IGNORECASE` applies the pattern case insensitively. `re.S` or `re.DOTALL` makes the dot match newlines. `re.M` or `re.MULTILINE` makes the caret and dollar match after and before line breaks in the subject string. There is no difference between the single-letter and descriptive options, except for the number of characters you have to type in. To specify more than one option, "or" them together with the | operator: `re.search("^a", "abc", re.I | re.M)`.

By default, Python's regex engine only considers the letters A through Z, the digits 0 through 9, and the underscore as "word characters". Specify the flag `re.L` or `re.LOCALE` to make `\w` match all characters that are considered letters given the current locale settings. Alternatively, you can specify `re.U` or `re.UNICODE` to treat all letters from all scripts as word characters. The setting also affects word boundaries.

Do not confuse `re.search()` with `re.match()`. Both functions do exactly the same, with the important distinction that `re.search()` will attempt the pattern throughout the string, until it finds a match. `re.match()` on the other hand, only attempts the pattern at the very start of the string. Basically, `re.match("regex", subject)` is the same as `re.search("\Aregex", subject)`. Note that `re.match()` does *not* require the regex to match the entire string. `re.match("a", "ab")` will succeed.

Python 3.4 adds a new `re.fullmatch()` function. This function only returns a `Match` object if the regex matches the string entirely. Otherwise it returns `None`. `re.fullmatch("regex",`

`subject)` is the same as `re.search("\Aregex\Z", subject)`. This is useful for validating user input. If `subject` is an empty string then `fullmatch()` evaluates to `True` for any regex that can find a zero-length match.

To get all matches from a string, call `re.findall(regex, subject)`. This will return an array of all non-overlapping regex matches in the string. "Non-overlapping" means that the string is searched through from left to right, and the next match attempt starts beyond the previous match. If the regex contains one or more capturing groups, `re.findall()` returns an array of tuples, with each tuple containing text matched by all the capturing groups. The overall regex match is *not* included in the tuple, unless you place the entire regex inside a capturing group.

More efficient than `re.findall()` is `re.finditer(regex, subject)`. It returns an iterator that enables you to loop over the regex matches in the subject string: `for m in re.finditer(regex, subject)`. The for-loop variable `m` is a `Match` object with the details of the current match.

Unlike `re.search()` and `re.match()`, `re.findall()` and `re.finditer()` do not support an optional third parameter with regex matching flags. Instead, you can use global mode modifiers at the start of the regex. E.g. "(?i)regex" matches `regex` case insensitively.

Strings, Backslashes and Regular Expressions

The backslash is a metacharacter in regular expressions, and is used to escape other metacharacters. The regex `\\` matches a single backslash. `\d` is a single token matching a digit.

Python strings also use the backslash to escape characters. The above regexes are written as Python strings as `"\\\\"` and `"\\w"`. Confusing indeed.

Fortunately, Python also has "raw strings" which do not apply special treatment to backslashes. As raw strings, the above regexes become `r"\\"` and `r"\w"`. The only limitation of using raw strings is that the delimiter you're using for the string must not appear in the regular expression, as raw strings do not offer a means to escape it.

You can use \n and \t in raw strings. Though raw strings do not support these escapes, the regular expression engine does. The end result is the same.

Unicode

Prior to Python 3.3, Python's re module did not support any Unicode regular expression tokens. Python Unicode strings, however, have always supported the `\uFFFF` notation. Python's re module can use Unicode strings. So you could pass the Unicode string `u"\u00E0\\d"` to the re module to match `à` followed by a digit. The backslash for `\d` was escaped, while the one for `\u` was not. That's because `\d` is a regular expression token, and a regular expression backslash needs to be escaped. `\u00E0` is a Python string token that shouldn't be escaped. The string `u"\u00E0\\d"` is seen by the regular expression engine as `à\d`.

If you did put another backslash in front of the `\u`, the regex engine would see `\u00E0\d`. If you use this regex with Python 3.2 or earlier, it will match the literal text `u00E0` followed by a digit instead.

To avoid any confusion about whether backslashes need to be escaped, just use Unicode raw strings like `ur"\u00E0\d"`. Then backslashes don't need to be escaped. Python does interpret Unicode escapes in raw strings.

In Python 3.0 and later, strings are Unicode by default. So the `u` prefix shown in the above samples is no longer necessary. Python 3.3 also adds support for the `\uFFFF` notation to the regular expression engine. So in Python 3.3, you can use the string `"\\u00E0\\d"` to pass the regex `\u00E0\d` which will match something like `à0`.

Search and Replace

`re.sub(regex, replacement, subject)` performs a search-and-replace across subject, replacing all matches of regex in subject with replacement. The result is returned by the `sub()` function. The subject string you pass is not modified.

If the regex has capturing groups, you can use the text matched by the part of the regex inside the capturing group. To substitute the text from the third group, insert `\3` into the replacement string. If you want to use the text of the third group followed by a literal three as the replacement, use `\g<3>3`. `\33` is interpreted as the 33rd group. It is an error if there are fewer than 33 groups. If you used named capturing groups, you can use them in the replacement text with `\g<name>`.

The `re.sub()` function applies the same backslash logic to the replacement text as is applied to the regular expression. Therefore, you should use raw strings for the replacement text, as I did in the examples above. The `re.sub()` function will also interpret \n and \t in raw strings. If you want `c:\temp` as a replacement, either use `r"c:\\temp"` or `"c:\\\\temp"`. The 3rd backreference is `r"\3"` or `"\\3"`.

Splitting Strings

`re.split(regex, subject)` returns an array of strings. The array contains the parts of subject between all the regex matches in the subject. Adjacent regex matches will cause empty strings to appear in the array. The regex matches themselves are not included in the array. If the regex contains capturing groups, then the text matched by the capturing groups is included in the array. The capturing groups are inserted between the substrings that appeared to the left and right of the regex match. If you don't want the capturing groups in the array, convert them into non-capturing groups. The `re.split()` function does not offer an option to suppress capturing groups.

You can specify an optional third parameter to limit the number of times the subject string is split. Note that this limit controls the number of splits, not the number of strings that will end up in the array. The unsplit remainder of the subject is added as the final string to the array. If there are no capturing groups, the array will contain `limit+1` items.

Match Details

`re.search()` and `re.match()` return a Match object, while `re.finditer()` generates an iterator to iterate over a Match object. This object holds lots of useful information about the regex match. I will use `m` to signify a Match object in the discussion below.

`m.group()` returns the part of the string matched by the entire regular expression. `m.start()` returns the offset in the string of the start of the match. `m.end()` returns the offset of the character beyond the match. `m.span()` returns a 2-tuple of `m.start()` and `m.end()`. You can use the `m.start()` and `m.end()` to slice the subject string: `subject[m.start():m.end()]`.

If you want the results of a capturing group rather than the overall regex match, specify the name or number of the group as a parameter. `m.group(3)` returns the text matched by the third capturing group. `m.group('groupname')` returns the text matched by a named group 'groupname'. If the group did not participate in the overall match, `m.group()` returns an empty string, while `m.start()` and `m.end()` return -1.

If you want to do a regular expression based search-and-replace without using `re.sub()`, call `m.expand(replacement)` to compute the replacement text. The function returns the replacement string with backreferences etc. substituted.

Regular Expression Objects

If you want to use the same regular expression more than once, you should compile it into a regular expression object. Regular expression objects are more efficient, and make your code more readable. To create one, just call `re.compile(regex)` or `re.compile(regex, flags)`. The flags are the matching options described above for the `re.search()` and `re.match()` functions.

The regular expression object returned by `re.compile()` provides all the functions that the re module also provides directly: `search()`, `match()`, `findall()`, `finditer()`, `sub()` and `split()`. The difference is that they use the pattern stored in the regex object, and do not take the regex as the first parameter. `re.compile(regex).search(subject)` is equivalent to `re.search(regex, subject)`.

Python learn : Chapter 11 Regular expressions

http://www.pythonlearn.com/html-007/cfbook012.html

So far we have been reading through files, looking for patterns and extracting various bits of lines that we find interesting. We have been using string methods like `split` and `find` and using lists and string slicing to extract portions of the lines.

This task of searching and extracting is so common that Python has a very powerful library called **regular expressions** that handles many of these tasks quite elegantly. The reason we have not introduced regular expressions earlier in the book is because while they are very powerful, they are a little complicated and their syntax takes some getting used to.

Regular expressions are almost their own little programming language for searching and parsing strings. As a matter of fact, entire books have been written on the topic of regular expressions. In this chapter, we will only cover the basics of regular expressions. For more detail on regular expressions, see:

`http://en.wikipedia.org/wiki/Regular_expression`

`http://docs.python.org/library/re.html`

The regular expression library must be imported into your program before you can use it. The simplest use of the regular expression library is the `search()` function. The following program demonstrates a trivial use of the search function.

```
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search('From:', line) :
        print line
```

We open the file, loop through each line and use the regular expression `search()` to only print out lines that contain the string ``From:''. This program does not use the real power of regular expressions since we could have just as easily used `line.find()` to accomplish the same result.

The power of the regular expressions comes when we add to special characters to the search string that allow us to more precisely control which lines match the string. Adding these special characters to our regular expression allow us to do sophisticated matching and extraction while writing very little code.

For example, the caret character is uses in regular expressions to match ``the beginning'' of a line. We could change our application to only match lines where ``From:'' was at the beginning of the line as follows:

```
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search('^From:', line) :
        print line
```

Now we will only match lines that *start with* the string ``From:". This is still a very simple example that we could have done equivalently with the `startswith()` method from the string library. But it serves to introduce the notion that regular expressions contain special action characters that give us more control as to what will match the regular expression.

11.1 Character matching in regular expressions

There are a number of other special characters that let us build even more powerful regular expressions. The most commonly used special character is the period character which matches any character.

In the following example, the regular expression ``F..m:" would match any of the strings ``From:", ``Fxxm:", ``F12m:", or ``F!@m:" since the period characters in the regular expression match any character.

```
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search('^F..m:', line) :
        print line
```

This is particularly powerful when combined with the ability to indicate that a character can be repeated any number of times using the ``*" or ``+" characters in your regular expression. These special characters mean that instead of matching a single character in the search string they match zero-or-more in the case of the asterisk or one-or-more of the characters in the case of the plus sign.

We can further narrow down the lines that we match using a repeated **wild card** character in the following example:

```
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search('^From:.+@', line) :
        print line
```

The search string ``^From:.+@" will successfully match lines that start with ``From:" followed by one or more characters ``.+" followed by an at-sign. So this will match the following line:

```
From: stephen.marquard @uct.ac.za
```
You can think of the ``.+" wildcard as expanding to match all the characters between the colon character and the at-sign.

```
From:.+ @
```
It is good to think of the plus and asterisk characters as ``pushy". For example the following string would match the last at-sign in the string as the ``.+" pushes outwards as shown below:

```
From: stephen.marquard@uct.ac.za, csev@umich.edu, and
cwen @iupui.edu
```
It is possible to tell an asterisk or plus-sign not to be so ``greedy" by adding another character. See the detailed documentation for information on turning off the greedy behavior.

11.2 Extracting data using regular expressions

If we want to extract data from a string in Python we can use the `findall()` method to extract all of the substrings which match a regular expression. Let's use the example of wanting to extract anything that looks like an e-mail address from any line regardless of format. For example, we want to pull the e-mail addresses from each of the following lines:

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16
2008
Return-Path: <postmaster@collab.sakaiproject.org>
          for <source@collab.sakaiproject.org>;
Received: (from apache@localhost)
Author: stephen.marquard@uct.ac.za
```
We don't want to write code for each of the types of lines, splitting and slicing differently for each line. This following program uses `findall()` to find the lines with e-mail addresses in them and extract one or more addresses from each of those lines.

```
import re
s = 'Hello from csev@umich.edu to cwen@iupui.edu about
the meeting @2PM'
lst = re.findall('\S+@\S+', s)
print lst
```
The `findall()` method searches the string in the second argument and returns a list of all of the strings that look like e-mail addresses. We are using a two-character sequence that matches a non-whitespace character (S).

The output of the program would be:

```
['csev@umich.edu', 'cwen@iupui.edu']
```
Translating the regular expression, we are looking for substrings that have at least one non-whitespace character, followed by an at-sign, followed by at least one more non-white space characters. Also, the ``S+" matches as many non-whitespace characters as possible (this is called ``**greedy**'' matching in regular expressions).

The regular expression would match twice (csev@umich.edu and cwen@iupui.edu) but it would not match the string ``@2PM" because there are no non-blank characters *before* the at-sign. We can use this regular expression in a program to read all the lines in a file and print out anything that looks like an e-mail address as follows:

```
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    x = re.findall('\S+@\S+', line)
    if len(x) > 0 :
        print x
```

We read each line and then extract all the substrings that match our regular expression. Since `findall()` returns a list, we simple check if the number of elements in our returned list is more than zero to print only lines where we found at least one substring that looks like an e-mail address.

If we run the program on `mbox.txt` we get the following output:

```
['wagnermr@iupui.edu']
['cwen@iupui.edu']
['<postmaster@collab.sakaiproject.org>']
['<200801032122.m03LMFo4005148@nakamura.uits.iupui.edu>
']
['<source@collab.sakaiproject.org>;']
['<source@collab.sakaiproject.org>;']
['<source@collab.sakaiproject.org>;']
['apache@localhost)']
['source@collab.sakaiproject.org;']
```

Some of our E-mail addresses have incorrect characters like ``<" or ``;" at the beginning or end. Let's declare that we are only interested in the portion of the string that starts and ends with a letter or a number.

To do this, we use another feature of regular expressions. Square brackets are used to indicate a set of multiple acceptable characters we are willing to consider matching. In a sense, the ``S" is asking to match the set of ``non-whitespace characters". Now we will be a little more explicit in terms of the characters we will match.

Here is our new regular expression:

```
[a-zA-Z0-9]\S*@\S*[a-zA-Z]
```

This is getting a little complicated and you can begin to see why regular expressions are their own little language unto themselves. Translating this regular expression, we are looking for substrings that start with a *single* lowercase letter, upper case letter, or number ``[a-zA-Z0-9]" followed by zero or more non blank characters ``S*", followed by an at-sign, followed by zero or more non-blank characters ``S*" followed by an upper or lower case letter. Note that we switched from ``+" to ``*" to indicate zero-or-more non-blank characters since ``[a-zA-Z0-9]" is already one non-blank character. Remember that the ``*" or ``+" applies to the single

character immediately to the left of the plus or asterisk.

If we use this expression in our program, our data is much cleaner:

```
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    x = re.findall('[a-zA-Z0-9]\S+@\S+[a-zA-Z]', line)
    if len(x) > 0 :
        print x

â€¦
['wagnermr@iupui.edu']
['cwen@iupui.edu']
['postmaster@collab.sakaiproject.org']
['200801032122.m03LMFo4005148@nakamura.uits.iupui.edu']
['source@collab.sakaiproject.org']
['source@collab.sakaiproject.org']
['source@collab.sakaiproject.org']
['apache@localhost']
```

Notice that on the ``source@collab.sakaiproject.org" lines, our regular expression eliminated two letters at the end of the string (``>;"). This is because when we append ``[a-zA-Z]" to the end of our regular expression, we are demanding that whatever string the regular expression parser finds, it must end with a letter. So when it sees the ``>" after ``sakaiproject.org>;" it simply stops at the last ``matching" letter it found (i.e. the ``g" was the last good match).

Also note that the output of the program is a Python list that has a string as the single element in the list.

11.3 Combining searching and extracting

If we want to find numbers on lines that start with the string ``X-" such as:

```
X-DSPAM-Confidence: 0.8475
X-DSPAM-Probability: 0.0000
```

We don't just want any floating point numbers from any lines. We only to extract numbers from lines that have the above syntax.

We can construct the following regular expression to select the lines:

```
^X-.*: [0-9.]+
```

Translating this, we are saying, we want lines that start with ``X-" followed by zero or more characters ``.*" followed by a colon (``:") and then a space. After the space we are looking for one or more characters that are either a digit (0-9) or a period ``[0-9.]+". Note that in between the square braces, the period matches an actual period (i.e. it is not a wildcard between the square brackets).

This is a very tight expression that will pretty much match only the lines we are interested in as follows:

```
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search('^X\S*: [0-9.]+', line) :
        print line
```

When we run the program, we see the data nicely filtered to show only the lines we are looking for.

```
X-DSPAM-Confidence: 0.8475
X-DSPAM-Probability: 0.0000
X-DSPAM-Confidence: 0.6178
X-DSPAM-Probability: 0.0000
```

But now we have to solve the problem of extracting the numbers using `split`. While it would be simple enough to use `split`, we can use another feature of regular expressions to both search and parse the line at the same time.

Parentheses are another special character in regular expressions. When you add parentheses to a regular expression they are ignored when matching the string, but when you are using `findall()`, parentheses indicate that while you want the whole expression to match, you only are interested in extracting a portion of the substring that matches the regular expression.

So we make the following change to our program:

```
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    x = re.findall('^X\S*: ([0-9.]+)', line)
    if len(x) > 0 :
        print x
```

Instead of calling `search()`, we add parentheses around the part of the regular expression that represents the floating point number to indicate we only want `findall()` to give us back the floating point number portion of the matching string.

The output from this program is as follows:

```
['0.8475']
['0.0000']
['0.6178']
['0.0000']
['0.6961']
['0.0000']
```

```
..
```
The numbers are still in a list and need to be converted from strings to floating point but we have used the power of regular expressions to both search and extract the information we found interesting.

As another example of this technique, if you look at the file there are a number of lines of the form:

```
Details:
http://source.sakaiproject.org/viewsvn/?view=rev&rev=39
772
```
If we wanted to extract all of the revision numbers (the integer number at the end of these lines) using the same technique as above, we could write the following program:

```
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    x = re.findall('^Details:.*rev=([0-9.]+)', line)
    if len(x) > 0:
        print x
```
Translating our regular expression, we are looking for lines that start with ``Details:', followed by any any number of characters ``.*" followed by ``rev=" and then by one or more digits. We want lines that match the entire expression but we only want to extract the integer number at the end of the line so we surround ``[0-9]+" with parentheses.

When we run the program, we get the following output:

```
['39772']
['39771']
['39770']
['39769']
...
```
Remember that the ``[0-9]+" is ``greedy" and it tries to make as large a string of digits as possible before extracting those digits. This ``greedy" behavior is why we get all five digits for each number. The regular expression library expands in both directions until it counters a non-digit, the beginning, or the end of a line.

Now we can use regular expressions to re-do an exercise from earlier in the book where we were interested in the time of day of each mail message. We looked for lines of the form:

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16
2008
```
And wanted to extract the hour of the day for each line. Previously we did this with two calls to `split`. First the line was split into words and then we pulled out the fifth word and split it again on the colon character to pull out the two characters we were interested in.

While this worked, it actually results in pretty brittle code that is assuming the lines are nicely

formatted. If you were to add enough error checking (or a big try/except block) to insure that your program never failed when presented with incorrectly formatted lines, the code would balloon to 10-15 lines of code that was pretty hard to read.

We can do this far simpler with the following regular expression:

```
^From .* [0-9][0-9]:
```

The translation of this regular expression is that we are looking for lines that start with ``From " (note the space) followed by any number of characters ``.*" followed by a space followed by two digits ``[0-9][0-9]" followed by a colon character. This is the definition of the kinds of lines we are looking for.

In order to pull out only the hour using `findall()`, we add parentheses around the two digits as follows:

```
^From .* ([0-9][0-9]):
```

This results in the following program:

```
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    x = re.findall('^From .* ([0-9][0-9]):', line)
    if len(x) > 0 : print x
```

When the program runs, it produces the following output:

```
['09']
['18']
['16']
['15']
â€¦
```

11.4 Escape character

Since we use special characters in regular expressions to match the beginning or end of a line or specify wild cards, we need a way to indicate that these characters are ``normal" and we want to match the actual character such as a dollar-sign or caret.

We can indicate that we want to simply match a character by prefixing that character with a backslash. For example, we can find money amounts with the following regular expression.

```
import re
x = 'We just received $10.00 for cookies.'
y = re.findall('\$[0-9.]+',x)
```

Since we prefix the dollar-sign with a backslash, it actually matches the dollar-sign in the input string instead of matching the ``end of line" and the rest of the regular expression matches one or more digits or the period character. *Note:* In between square brackets,

characters are not ``special". So when we say ``[0-9.]", it really means digits or a period. Outside of square brackets, a period is the ``wild-card" character and matches any character. In between square brackets, the period is a period.

11.5 Summary

While this only scratched the surface of regular expressions, we have learned a bit about the language of regular expressions. They are search strings that have special characters in them that communicate your wishes to the regular expression system as to what defines ``matching" and what is extracted from the matched strings. Here are some of those special characters and character sequences:

^
Matches the beginning of the line.

$
Matches the end of the line.

.
Matches any character (a wildcard).

s
Matches a whitespace character.

S
Matches a non-whitespace character (opposite of s).

*
Applies to the immediately preceding character and indicates to match zero or more of the preceding character.

*?
Applies to the immediately preceding character and indicates to match zero or more of the preceding character in ``non-greedy mode".

+
Applies to the immediately preceding character and indicates to match zero or more of the preceding character.

+?
Applies to the immediately preceding character and indicates to match zero or more of the preceding character in ``non-greedy mode".

[aeiou]
Matches a single character as long as that character is in the specified set. In this example, it would match ``a", ``e", ``i", ``o" or ``u" but no other characters.

[a-z0-9]
You can specify ranges of characters using the minus sign. This example is a single character that must be a lower case letter or a digit.

[^A-Za-z]
When the first character in the set notation is a caret, it inverts the logic. This example matches a single character that is anything *other than* an upper or lower case character.

( )
When parentheses are added to a regular expression, they are ignored for the purpose of matching, but allow you to extract a particular subset of the matched string rather than the whole string when using `findall()`.

b
Matches the empty string, but only at the start or end of a word.

B
Matches the empty string, but not at the start or end of a word.

d
Matches any decimal digit; equivalent to the set [0-9].

D
Matches any non-digit character; equivalent to the set [^0-9].

11.6 Bonus section for UNIX users

Support for searching files using regular expressions was built into the UNIX operating system since the 1960's and it is available in nearly all programming languages in one form or another.

As a matter of fact, there is a command-line program built into UNIX called **grep** (Generalized Regular Expression Parser) that does pretty much the same as the `search()` examples in this chapter. So if you have a Macintosh or Linux system, you can try the following commands in your command line window.

```
$ grep '^From:' mbox-short.txt
From: stephen.marquard@uct.ac.za
From: louis@media.berkeley.edu
From: zqian@umich.edu
From: rjlowe@iupui.edu
```

This tells `grep` to show you lines that start with the string ``From:" in the file `mbox-short.txt`. If you experiment with the `grep` command a bit and read the documentation for `grep`, you will find some subtle differences between the regular expression support in Python and the regular expression support in `grep`. As an example, `grep` does not support the non-blank character ``S" so you will need to use the slightly more complex set notation ``[^]"- which simply means - match a character that is anything other than a space.

11.7 Debugging

Python has some simple and rudimentary built-in documentation that can be quite helpful if you need a quick refresher to trigger your memory about the exact name of a particular method. This documentation can be viewed in the Python interpreter in interactive mode.

You can bring up an interactive help system using `help()`.

```
>>> help()

Welcome to Python 2.6!  This is the online help
utility.

If this is your first time using Python, you should
definitely check out
the tutorial on the Internet at
http://docs.python.org/tutorial/.

Enter the name of any module, keyword, or topic to get
help on writing
Python programs and using Python modules.  To quit this
help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, or
topics, type "modules",
"keywords", or "topics".  Each module also comes with a
one-line summary
of what it does; to list the modules whose summaries
contain a given word
such as "spam", type "modules spam".

help> modules
```

If you know what module you want to use, you can use the `dir()` command to find the methods in the module as follows:

```
>>> import re
>>> dir(re)
[.. 'compile', 'copy_reg', 'error', 'escape',
'findall',
'finditer', 'match', 'purge', 'search', 'split',
'sre_compile',
'sre_parse', 'sub', 'subn', 'sys', 'template']
```

You can also get a small amount of documentation on a particular method using the dir command.

```
>>> help (re.search)
Help on function search in module re:
```

```
search(pattern, string, flags=0)
    Scan through string looking for a match to the
pattern, returning
    a match object, or None if no match was found.
>>>
```

The built in documentation is not very extensive, but it can be helpful when you are in a hurry or don't have access to a web browser or search engine.

11.8 Glossary

brittle code:
Code that works when the input data is in a particular format but prone to breakage if there is some deviation from the correct format. We call this ``brittle code" because it is easily broken.

greedy matching:
The notion that the ``+" and ``*" characters in a regular expression expand outward to match the largest possible string.

grep:
A command available in most UNIX systems that searches through text files looking for lines that match regular expressions. The command name stands for "Generalized Regular Expression Parser".

regular expression:
A language for expressing more complex search strings. A regular expression may contain special characters that indicate that a search only matches at the beginning or end of a line or many other similar capabilities.

wild card:
A special character that matches any character. In regular expressions the wild card character is the period character.

11.9 Exercises

Exercise 1 *Write a simple program to simulate the operation of the the `grep` command on UNIX. Ask the user to enter a regular expression and count the number of lines that matched the regular expression:*

```
$ python grep.py
Enter a regular expression: ^Author
mbox.txt had 1798 lines that matched ^Author

$ python grep.py
Enter a regular expression: ^X-
mbox.txt had 14368 lines that matched ^X-
```

```
$ python grep.py
Enter a regular expression: java$
mbox.txt had 4218 lines that matched java$
```

Exercise 2 *Write a program to look for lines of the form*

```
New Revision: 39772
```

And extract the number from each of the lines using a regular expression and the `findall()` *method. Compute the average of the numbers and print out the average.*

```
Enter file:mbox.txt
38549.7949721

Enter file:mbox-short.txt
39756.9259259
```