re – Regular Expressions

Purpose:	Searching within and changing text using formal patterns.
Available In:	1.5 and later

Regular expressions are text matching patterns described with a formal syntax. The patterns are interpreted as a set of instructions, which are then executed with a string as input to produce a matching subset or modified version of the original. The term "regular expressions" is frequently shortened to as "regex" or "regexp" in conversation. Expressions can include literal text matching, repetition, pattern-composition, branching, and other sophisticated rules. A large number of parsing problems are easier to solve with a regular expression than by creating a special-purpose lexer and parser.

Regular expressions are typically used in applications that involve a lot of text processing. For example, they are commonly used as search patterns in text editing programs used by developers, including vi, emacs, and modern IDEs. They are also an integral part of Unix command line utilities such as sed, grep, and awk. Many programming languages include support for regular expressions in the language syntax (Perl, Ruby, Awk, and Tcl). Other languages, such as C, C++, and Python supports regular expressions through extension libraries.

There are multiple open source implementations of regular expressions, each sharing a common core syntax but with different extensions or modifications to their advanced features. The syntax used in Python's `re` module is based on the syntax used for regular expressions in Perl, with a few Python-specific enhancements.

Note: Although the formal definition of "regular expression" is limited to expressions that describe regular languages, some of the extensions supported by `re` go beyond describing regular languages. The term

Page Contents

Navigation

Table of Contents
Previous: StringIO and cStringIO – Work with text buffers using file-like API
Next: struct – Working with Binary Data

This Page

Show Source

Examples

The output from all the example programs from PyMOTW has been generated with Python 2.7.8, unless otherwise noted. Some of the features described here may not be available in earlier versions of Python.

"regular expression" is used here in a more general sense to mean any expression that can be evaluated by Python's `re` module.

Finding Patterns in Text

The most common use for `re` is to search for patterns in text. This example looks for two literal strings, `'this'` and `'that'`, in a text string.

```
import re

patterns = [ 'this', 'that' ]
text = 'Does this text match the pa

for pattern in patterns:
    print 'Looking for "%s" in "%s"

    if re.search(pattern,  text):
        print 'found a match!'
    else:
        print 'no match'
```

`search()` takes the pattern and text to scan, and returns a **`Match`** object when the pattern is found. If the pattern is not found, **`search()`** returns `None`.

```
$ python re_simple.py

Looking for "this" in "Does this te
Looking for "that" in "Does this te
```

The **`Match`** object returned by **`search()`** holds information about the nature of the match, including the original input string, the regular expression used, and the location within the original string where the pattern occurs.

```
import re

pattern = 'this'
text = 'Does this text match the pa

match = re.search(pattern, text)

s = match.start()
e = match.end()

print 'Found "%s" in "%s" from %d t
    (match.re.pattern, match.string
```

If you are looking for examples that work under Python 3, please refer to the PyMOTW-3 section of the site.

The **start()** and **end()** methods give the integer indexes into the string showing where the text matched by the pattern occurs.

```
$ python re_simple_match.py

Found "this" in "Does this text mat
```

Compiling Expressions

re includes module-level functions for working with regular expressions as text strings, but it is usually more efficient to *compile* the expressions your program uses frequently. The **compile()** function converts an expression string into a **RegexObject**.

```
import re

# Pre-compile the patterns
regexes = [ re.compile(p) for p in


            ]
text = 'Does this text match the pa

for regex in regexes:
    print 'Looking for "%s" in "%s"

    if regex.search(text):
        print 'found a match!'
    else:
        print 'no match'
```

The module-level functions maintain a cache of compiled expressions, but the size of the cache is limited and using compiled expressions directly means you can avoid the cache lookup overhead. By pre-compiling any expressions your module uses when the module is loaded you shift the compilation work to application startup time, instead of a point where the program is responding to a user action.

```
$ python re_simple_compiled.py

Looking for "this" in "Does this te
Looking for "that" in "Does this te
```

Multiple Matches

So far the example patterns have all used **search()** to look for single instances of literal

text strings. The **`findall()`** function returns all of the substrings of the input that match the pattern without overlapping.

```
import re

text = 'abbaaabbbbaaaaa'

pattern = 'ab'

for match in re.findall(pattern, te
    print 'Found "%s"' % match
```

There are two instances of `ab` in the input string.

```
$ python re_findall.py

Found "ab"
Found "ab"
```

`finditer()` returns an iterator that produces **`Match`** instances instead of the strings returned by **`findall()`**.

```
import re

text = 'abbaaabbbbaaaaa'

pattern = 'ab'

for match in re.finditer(pattern, t
    s = match.start()
    e = match.end()
    print 'Found "%s" at %d:%d' % (
```

This example finds the same two occurrences of `ab`, and the **`Match`** instance shows where they are in the original input.

```
$ python re_finditer.py

Found "ab" at 0:2
Found "ab" at 5:7
```

Pattern Syntax

Regular expressions support more powerful patterns than simple literal text strings. Patterns can repeat, can be anchored to different logical locations within the input, and can be expressed in compact forms that don't require every literal character be present in the pattern. All of these features are used by combining literal text values with

metacharacters that are part of the regular expression pattern syntax implemented by `re`. The following examples will use this test program to explore variations in patterns.

```
import re

def test_patterns(text, patterns=[]
    """Given source text and a list
    matches for each pattern within
    them to stdout.
    """
    # Show the character positions (
    print
    print ''.join(str(i/10 or ' ') +
    print ''.join(str(i%10) for i in
    print text

    # Look for each pattern in the t
    for pattern in patterns:
        print
        print 'Matching "%s"' % patt
        for match in re.finditer(pat
            s = match.start()
            e = match.end()
            print '  %2d : %2d = "%
                (s, e-1, text[s:e])
    return

if __name__ == '__main__':
    test_patterns('abbaaabbbbaaaaa'
```

The output of **`test_patterns()`** shows the input text, including the character positions, as well as the substring range from each portion of the input that matches the pattern.

```
$ python re_test_patterns.py


          11111
012345678901234
abbaaabbbbaaaaa

Matching "ab"
   0 :  1 = "ab"
   5 :  6 = "ab"
```

Repetition

There are five ways to express repetition in a pattern. A pattern followed by the metacharacter `*` is repeated zero or more times (allowing a pattern to repeat zero times means it does not need to appear at all to match). Replace the `*` with `+` and the pattern must appear at least once. Using `?` means the pattern appears zero or one time. For a specific number of occurrences, use `{m}` after the pattern, where

m is replaced with the number of times the pattern should repeat. And finally, to allow a variable but limited number of repetitions, use `{m,n}` where *m* is the minimum number of repetitions and *n* is the maximum. Leaving out *n* (`{m,}`) means the value appears at least *m* times, with no maximum.

```
from re_test_patterns import test_p

test_patterns('abbaaabbbbaaaaa',
              [ 'ab*',     # a foll
                'ab+',     # a foll
                'ab?',     # a foll
                'ab{3}',   # a foll
                'ab{2,3}', # a foll
                ])
```

Notice how many more matches there are for `ab*` and `ab?` than `ab+`.

```
$ python re_repetition.py


          11111
012345678901234
abbaaabbbbaaaaa

Matching "ab*"
   0 :  2 = "abb"
   3 :  3 = "a"
   4 :  4 = "a"
   5 :  9 = "abbbb"
  10 : 10 = "a"
  11 : 11 = "a"
  12 : 12 = "a"
  13 : 13 = "a"
  14 : 14 = "a"

Matching "ab+"
   0 :  2 = "abb"
   5 :  9 = "abbbb"

Matching "ab?"
   0 :  1 = "ab"
   3 :  3 = "a"
   4 :  4 = "a"
   5 :  6 = "ab"
  10 : 10 = "a"
  11 : 11 = "a"
  12 : 12 = "a"
  13 : 13 = "a"
  14 : 14 = "a"

Matching "ab{3}"
   5 :  8 = "abbb"

Matching "ab{2,3}"
   0 :  2 = "abb"
   5 :  8 = "abbb"
```

The normal processing for a repetition instruction is to consume as much of the input as possible while matching the pattern. This so-called *greedy* behavior may result in fewer individual matches, or the matches may include more of the input text than intended. Greediness can be turned off by following the repetition instruction with `?`.

```
from re_test_patterns import test_p

test_patterns('abbaaabbbbaaaaa',
              [ 'ab*?',     # a fol
                'ab+?',     # a fol
                'ab??',     # a fol
                'ab{3}?',   # a fol
                'ab{2,3}?', # a fol
                ])
```

Disabling greedy consumption of the input for any of the patterns where zero occurences of `b` are allowed means the matched substring does not include any `b` characters.

```
$ python re_repetition_non_greedy.p


          11111
012345678901234
abbaaabbbbaaaaa

Matching "ab*?"
   0 :  0 = "a"
   3 :  3 = "a"
   4 :  4 = "a"
   5 :  5 = "a"
  10 : 10 = "a"
  11 : 11 = "a"
  12 : 12 = "a"
  13 : 13 = "a"
  14 : 14 = "a"

Matching "ab+?"
   0 :  1 = "ab"
   5 :  6 = "ab"

Matching "ab??"
   0 :  0 = "a"
   3 :  3 = "a"
   4 :  4 = "a"
   5 :  5 = "a"
  10 : 10 = "a"
  11 : 11 = "a"
  12 : 12 = "a"
  13 : 13 = "a"
  14 : 14 = "a"

Matching "ab{3}?"
   5 :  8 = "abbb"

Matching "ab{2,3}?"
```

```
   0 :  2 = "abb"
   5 :  7 = "abb"
```

Character Sets

A *character set* is a group of characters, any one of which can match at that point in the pattern. For example, `[ab]` would match either `a` or `b`.

```
from re_test_patterns import test_p

test_patterns('abbaaabbbbaaaaa',
              [ '[ab]',    # either
                'a[ab]+',  # a foll
                'a[ab]+?', # a foll
                ])
```

The greedy form of the expression, `a[ab]+`, consumes the entire string because the first letter is `a` and every subsequent character is either `a` or `b`.

```
$ python re_charset.py


          11111
012345678901234
abbaaabbbbaaaaa

Matching "[ab]"
   0 :  0 = "a"
   1 :  1 = "b"
   2 :  2 = "b"
   3 :  3 = "a"
   4 :  4 = "a"
   5 :  5 = "a"
   6 :  6 = "b"
   7 :  7 = "b"
   8 :  8 = "b"
   9 :  9 = "b"
  10 : 10 = "a"
  11 : 11 = "a"
  12 : 12 = "a"
  13 : 13 = "a"
  14 : 14 = "a"

Matching "a[ab]+"
   0 : 14 = "abbaaabbbbaaaaa"

Matching "a[ab]+?"
   0 :  1 = "ab"
   3 :  4 = "aa"
   5 :  6 = "ab"
  10 : 11 = "aa"
  12 : 13 = "aa"
```

A character set can also be used to exclude specific characters. The special marker `^` means to look for characters not in the set following.

```
from re_test_patterns import test_p

test_patterns('This is some text --
              [ '[^-. ]+',  # seque
                ])
```

This pattern finds all of the substrings that do not contain the characters `-`, `.`, or a space.

```
$ python re_charset_exclude.py


          111111111122222222233333
012345678901234567890123456789012345
This is some text -- with punctuati

Matching "[^-. ]+"
   0 :  3 = "This"
   5 :  6 = "is"
   8 : 11 = "some"
  13 : 16 = "text"
  21 : 24 = "with"
  26 : 36 = "punctuation"
```

As character sets grow larger, typing every character that should (or should not) match becomes tedious. A more compact format using *character ranges* lets you define a character set to include all of the contiguous characters between a start and stop point.

```
from re_test_patterns import test_p

test_patterns('This is some text --
              [ '[a-z]+',      # se
                '[A-Z]+',      # se
                '[a-zA-Z]+',   # se
                '[A-Z][a-z]+', # on
                ])
```

Here the range `a-z` includes the lower case ASCII letters, and the range `A-Z` includes the upper case ASCII letters. The ranges can also be combined into a single character set.

```
$ python re_charset_ranges.py


          111111111122222222233333
012345678901234567890123456789012345
This is some text -- with punctuati

Matching "[a-z]+"
   1 :  3 = "his"
   5 :  6 = "is"
   8 : 11 = "some"
  13 : 16 = "text"
  21 : 24 = "with"
```

```
  26 : 36 = "punctuation"

Matching "[A-Z]+"
   0 :  0 = "T"

Matching "[a-zA-Z]+"
   0 :  3 = "This"
   5 :  6 = "is"
   8 : 11 = "some"
  13 : 16 = "text"
  21 : 24 = "with"
  26 : 36 = "punctuation"

Matching "[A-Z][a-z]+"
   0 :  3 = "This"
```

As a special case of a character set the metacharacter dot, or period (`.`), indicates that the pattern should match any single character in that position.

```
from re_test_patterns import test_p

test_patterns('abbaaabbbbaaaaa',
              [ 'a.',   # a followe
                'b.',   # b followe
                'a.*b', # a followe
                'a.*?b', # a follow
                ])
```

Combining dot with repetition can result in very long matches, unless the non-greedy form is used.

```
$ python re_charset_dot.py


          11111
012345678901234
abbaaabbbbaaaaa

Matching "a."
   0 :  1 = "ab"
   3 :  4 = "aa"
   5 :  6 = "ab"
  10 : 11 = "aa"
  12 : 13 = "aa"

Matching "b."
   1 :  2 = "bb"
   6 :  7 = "bb"
   8 :  9 = "bb"

Matching "a.*b"
   0 :  9 = "abbaaabbbb"

Matching "a.*?b"
   0 :  1 = "ab"
   3 :  6 = "aaab"
```

Escape Codes

An even more compact representation uses escape codes for several pre-defined character sets. The escape codes recognized by `re` are:

Code	Meaning
`\d`	a digit
`\D`	a non-digit
`\s`	whitespace (tab, space, newline, etc.)
`\S`	non-whitespace
`\w`	alphanumeric
`\W`	non-alphanumeric

Note: Escapes are indicated by prefixing the character with a backslash (`\`). Unfortunately, a backslash must itself be escaped in normal Python strings, and that results in expressions that are difficult to read. Using *raw* strings, created by prefixing the literal value with `r`, for creating regular expressions eliminates this problem and maintains readability.

```
from re_test_patterns import test_p

test_patterns('This is a prime #1 e
              [ r'\d+', # sequence
                r'\D+', # sequence
                r'\s+', # sequence
                r'\S+', # sequence
                r'\w+', # alphanume
                r'\W+', # non-alpha
                ])
```

These sample expressions combine escape codes with repetition to find sequences of like characters in the input string.

```
$ python re_escape_codes.py


          11111111112222222
012345678901234567890123456
This is a prime #1 example!

Matching "\d+"
  17 : 17 = "1"

Matching "\D+"
   0 : 16 = "This is a prime #"
  18 : 26 = " example!"

Matching "\s+"
   4 :  4 = " "
   7 :  7 = " "
```

```
   9 :  9 = " "
  15 : 15 = " "
  18 : 18 = " "

Matching "\S+"
   0 :  3 = "This"
   5 :  6 = "is"
   8 :  8 = "a"
  10 : 14 = "prime"
  16 : 17 = "#1"
  19 : 26 = "example!"

Matching "\w+"
   0 :  3 = "This"
   5 :  6 = "is"
   8 :  8 = "a"
  10 : 14 = "prime"
  17 : 17 = "1"
  19 : 25 = "example"

Matching "\W+"
   4 :  4 = " "
   7 :  7 = " "
   9 :  9 = " "
  15 : 16 = " #"
  18 : 18 = " "
  26 : 26 = "!"
```

To match the characters that are part of the regular expression syntax, escape the characters in the search pattern.

```
from re_test_patterns import test_p

test_patterns(r'\d+ \D+ \s+ \S+ \w+
              [ r'\\d\+',
                r'\\D\+',
                r'\\s\+',
                r'\\S\+',
                r'\\w\+',
                r'\\W\+',
                ])
```

These patterns escape the backslash and plus characters, since as metacharacters both have special meaning in a regular expression.

```
$ python re_escape_escapes.py


          1111111111222
01234567890123456789012
\d+ \D+ \s+ \S+ \w+ \W+

Matching "\\d\+"
   0 :  2 = "\d+"

Matching "\\D\+"
   4 :  6 = "\D+"

Matching "\\s\+"
   8 : 10 = "\s+"

Matching "\\S\+"
```

```
  12 : 14 = "\S+"

Matching "\\w\+"
  16 : 18 = "\w+"

Matching "\\W\+"
  20 : 22 = "\W+"
```

Anchoring

In addition to describing the content of a pattern to match, you can also specify the relative location in the input text where the pattern should appear using *anchoring* instructions.

Code	Meaning
^	start of string, or line
$	end of string, or line
\A	start of string
\Z	end of string
\b	empty string at the beginning or end of a word
\B	empty string not at the beginning or end of a word

```
from re_test_patterns import test_p

test_patterns('This is some text --
              [ r'^\w+',      # word
                r'\A\w+',     # word
                r'\w+\S*$',   # word
                r'\w+\S*\Z', # word
                r'\w*t\w*',   # word
                r'\bt\w+',    # 't'
                r'\w+t\b',    # 't'
                r'\Bt\B',     # 't',
                ])
```

The patterns in the example for matching words at the beginning and end of the string are different because the word at the end of the string is followed by punctuation to terminate the sentence. The pattern `\w+$` would not match, since `.` is not considered an alphanumeric character.

```
$ python re_anchoring.py


          111111111122222222223333
012345678901234567890123456789012
This is some text -- with punctuati

Matching "^\w+"
   0 :  3 = "This"
```

```
Matching "\A\w+"
   0 :  3 = "This"

Matching "\w+\S*$"
  26 : 37 = "punctuation."

Matching "\w+\S*\Z"
  26 : 37 = "punctuation."

Matching "\w*t\w*"
  13 : 16 = "text"
  21 : 24 = "with"
  26 : 36 = "punctuation"

Matching "\bt\w+"
  13 : 16 = "text"

Matching "\w+t\b"
  13 : 16 = "text"

Matching "\Bt\B"
  23 : 23 = "t"
  30 : 30 = "t"
  33 : 33 = "t"
```

Constraining the Search

In situations where you know in advance that only a subset of the full input should be searched, you can further constrain the regular expression match by telling `re` to limit the search range. For example, if your pattern must appear at the front of the input, then using **`match()`** instead of **`search()`** will anchor the search without having to explicitly include an anchor in the search pattern.

```
import re

text = 'This is some text -- with pu
pattern = 'is'

print 'Text   :', text
print 'Pattern:', pattern

m = re.match(pattern, text)
print 'Match  :', m
s = re.search(pattern, text)
print 'Search :', s
```

Since the literal text `is` does not appear at the start of the input text, it is not found using **`match()`**. The sequence appears two other times in the text, though, so **`search()`** finds it.

```
$ python re_match.py

Text   : This is some text -- with
```

```
Pattern: is
Match  : None
Search : <_sre.SRE_Match object at (
```

The **`search()`** method of a compiled regular expression accepts optional *start* and *end* position parameters to limit the search to a substring of the input.

```
import re

text = 'This is some text -- with pu
pattern = re.compile(r'\b\w*is\w*\b

print 'Text:', text
print

pos = 0
while True:
    match = pattern.search(text, pos
    if not match:
        break
    s = match.start()
    e = match.end()
    print '  %2d : %2d = "%s"' % \
        (s, e-1, text[s:e])
    # Move forward in text for the n
    pos = e
```

This example implements a less efficient form of **`iterall()`**. Each time a match is found, the end position of that match is used for the next search.

```
$ python re_search_substring.py

Text: This is some text -- with punc

   0 :  3 = "This"
   5 :  6 = "is"
```

Dissecting Matches with Groups

Searching for pattern matches is the basis of the powerful capabilities provided by regular expressions. Adding *groups* to a pattern lets you isolate parts of the matching text, expanding those capabilities to create a parser. Groups are defined by enclosing patterns in parentheses (`(` and `)`).

```
from re_test_patterns import test_pa

test_patterns('abbaaabbbbaaaaa',
              [ 'a(ab)',    # 'a' fo
```

```
                'a(a*b*)',  # 'a' f
                'a(ab)*',   # 'a' f
                'a(ab)+',   # 'a' f
                ])
```

Any complete regular expression can be converted to a group and nested within a larger expression. All of the repetition modifiers can be applied to a group as a whole, requiring the entire group pattern to repeat.

```
$ python re_groups.py


          11111
012345678901234
abbaaabbbbaaaaa

Matching "a(ab)"
   4 :  6 = "aab"

Matching "a(a*b*)"
   0 :  2 = "abb"
   3 :  9 = "aaabbbb"
  10 : 14 = "aaaaa"

Matching "a(ab)*"
   0 :  0 = "a"
   3 :  3 = "a"
   4 :  6 = "aab"
  10 : 10 = "a"
  11 : 11 = "a"
  12 : 12 = "a"
  13 : 13 = "a"
  14 : 14 = "a"

Matching "a(ab)+"
   4 :  6 = "aab"
```

To access the substrings matched by the individual groups within a pattern, use the **`groups()`** method of the **`Match`** object.

```
import re

text = 'This is some text -- with p

print text
print

for pattern in [ r'^(\w+)',
                 r'(\w+)\S*$',
                 r'(\bt\w+)\W+(\w+)
                 r'(\w+t)\b',
                 ]:
    regex = re.compile(pattern)
    match = regex.search(text)
    print 'Matching "%s"' % pattern
    print '  ', match.groups()
    print
```

`Match.groups()` returns a sequence of strings in the order of the group within the expression that matches the string.

```
$ python re_groups_match.py

This is some text -- with punctuati

Matching "^(\w+)"
   ('This',)

Matching "(\w+)\S*$"
   ('punctuation',)

Matching "(\bt\w+)\W+(\w+)"
   ('text', 'with')

Matching "(\w+t)\b"
   ('text',)
```

If you are using grouping to find parts of the string, but you don't need all of the parts matched by groups, you can ask for the match of only a single group with **`group()`**.

```
import re

text = 'This is some text -- with pu

print 'Input text            :', te

# word starting with 't' then anoth
regex = re.compile(r'(\bt\w+)\W+(\w+
print 'Pattern               :', re

match = regex.search(text)
print 'Entire match          :', ma
print 'Word starting with "t":', ma
print 'Word after "t" word   :', ma
```

Group `0` represents the string matched by the entire expression, and sub-groups are numbered starting with `1` in the order their left parenthesis appears in the expression.

```
$ python re_groups_individual.py

Input text            : This is som
Pattern               : (\bt\w+)\W+
Entire match          : text -- wit
Word starting with "t": text
Word after "t" word   : with
```

Python extends the basic grouping syntax to add *named groups*. Using names to refer to groups makes it easier to modify the pattern over time, without having to also modify the

code using the match results. To set the name of a group, use the syntax `(P?<name>pattern)`.

```
import re

text = 'This is some text -- with pu

print text
print

for pattern in [ r'^(?P<first_word>\
                 r'(?P<last_word>\w+
                 r'(?P<t_word>\bt\w+
                 r'(?P<ends_with_t>\
                 ]:
    regex = re.compile(pattern)
    match = regex.search(text)
    print 'Matching "%s"' % pattern
    print '  ', match.groups()
    print '  ', match.groupdict()
    print
```

Use **`groupdict()`** to retrieve the dictionary mapping group names to substrings from the match. Named patterns are included in the ordered sequence returned by **`groups()`**, as well.

```
$ python re_groups_named.py

This is some text -- with punctuati

Matching "^(?P<first_word>\w+)"
   ('This',)
   {'first_word': 'This'}

Matching "(?P<last_word>\w+)\S*$"
   ('punctuation',)
   {'last_word': 'punctuation'}

Matching "(?P<t_word>\bt\w+)\W+(?P<
   ('text', 'with')
   {'other_word': 'with', 't_word':

Matching "(?P<ends_with_t>\w+t)\b"
   ('text',)
   {'ends_with_t': 'text'}
```

An updated version of **`test_patterns()`** that shows the numbered and named groups matched by a pattern will make the following examples easier to follow.

```
import re

def test_patterns(text, patterns=[]
    """Given source text and a list
    matches for each pattern within
    them to stdout.
    """
    # Show the character positions
    print
```

```
    print ''.join(str(i/10 or ' ') 
    print ''.join(str(i%10) for i i
    print text

    # Look for each pattern in the 
    for pattern in patterns:
        print
        print 'Matching "%s"' % pat
        for match in re.finditer(pa
            s = match.start()
            e = match.end()
            print '  %2d : %2d = "%
                (s, e-1, text[s:e])
            print '    Groups:', ma
            if match.groupdict():
                print '    Named gr
            print
    return
```

Since a group is itself a complete regular expression, groups can be nested within other groups to build even more complicated expressions.

```
from re_test_patterns_groups import

test_patterns('abbaaabbbbaaaaa',
              [r'a((a*)(b*))', # 'a
               ])
```

In this case, the group `(a*)` matches an empty string, so the return value from **`groups()`** includes that empty string as the matched value.

```
$ python re_groups_nested.py


          11111
012345678901234
abbaaabbbbaaaaa

Matching "a((a*)(b*))"
   0 :  2 = "abb"
    Groups: ('bb', '', 'bb')

   3 :  9 = "aaabbbb"
    Groups: ('aabbbb', 'aa', 'bbbb'

  10 : 14 = "aaaaa"
    Groups: ('aaaa', 'aaaa', '')
```

Groups are also useful for specifying alternative patterns. Use `|` to indicate that one pattern or another should match. Consider the placement of the `|` carefully, though. The first expression in this example matches a sequence of `a`

followed by a sequence consisting entirely of a single letter, `a` or `b`. The second pattern matches `a` followed by a sequence that may include *either* `a` or `b`. The patterns are similar, but the resulting matches are completely different.

```
from re_test_patterns_groups import

test_patterns('abbaaabbbbaaaaa',
              [r'a((a+)|(b+))', # '
               r'a((a|b)+)',    # '
               ])
```

When an alternative group is not matched, but the entire pattern does match, the return value of **`groups()`** includes a `None` value at the point in the sequence where the alternative group should appear.

```
$ python re_groups_alternative.py


          11111
012345678901234
abbaaabbbbaaaaa

Matching "a((a+)|(b+))"
   0 :  2 = "abb"
    Groups: ('bb', None, 'bb')

   3 :  5 = "aaa"
    Groups: ('aa', 'aa', None)

  10 : 14 = "aaaaa"
    Groups: ('aaaa', 'aaaa', None)


Matching "a((a|b)+)"
   0 : 14 = "abbaaabbbbaaaaa"
    Groups: ('bbaaabbbbaaaaa', 'a')
```

Defining a group containing a sub-pattern is also useful in cases where the string matching the sub-pattern is not part of what you want to extract from the full text. These groups are called *non-capturing*. To create a non-capturing group, use the syntax `(?:pattern)`.

```
from re_test_patterns_groups import

test_patterns('abbaaabbbbaaaaa',
              [r'a((a+)|(b+))',
               r'a((?:a+)|(?:b+))',
               ])
```

Compare the groups returned for the capturing and non-capturing forms of a pattern that matches the same results.

```
$ python re_groups_non_capturing.py


          11111
012345678901234
abbaaabbbbaaaaa

Matching "a((a+)|(b+))"
   0 :  2 = "abb"
    Groups: ('bb', None, 'bb')

   3 :  5 = "aaa"
    Groups: ('aa', 'aa', None)

  10 : 14 = "aaaaa"
    Groups: ('aaaa', 'aaaa', None)


Matching "a((?:a+)|(?:b+))"
   0 :  2 = "abb"
    Groups: ('bb',)

   3 :  5 = "aaa"
    Groups: ('aa',)

  10 : 14 = "aaaaa"
    Groups: ('aaaa',)
```

Search Options

You can change the way the matching engine processes an expression using option flags. The flags can be combined using a bitwise or operation, and passed to **`compile()`**, **`search()`**, **`match()`**, and other functions that accept a pattern for searching.

Case-insensitive Matching

`IGNORECASE` causes literal characters and character ranges in the pattern to match both upper and lower case characters.

```
import re

text = 'This is some text -- with pu
pattern = r'\bT\w+'
with_case = re.compile(pattern)
without_case = re.compile(pattern,

print 'Text            :', text
print 'Pattern         :', pattern
print 'Case-sensitive  :', with_cas
print 'Case-insensitive:', without_
```

Since the pattern includes the literal `T`, without setting **`IGNORECASE`** the only match is the word `This`. When case is ignored, `text` also matches.

```
$ python re_flags_ignorecase.py

Text            : This is some text
Pattern         : \bT\w+
Case-sensitive  : ['This']
Case-insensitive: ['This', 'text']
```

Input with Multiple Lines

There are two flags that effect how searching in multi-line input works. The **`MULTILINE`** flag controls how the pattern matching code processes anchoring instructions for text containing newline characters. When multiline mode is turned on, the anchor rules for `^` and `$` apply at the beginning and end of each line, in addition to the entire string.

```
import re

text = 'This is some text -- with p
pattern = r'(^\w+)|(\w+\S*$)'
single_line = re.compile(pattern)
multiline = re.compile(pattern, re.

print 'Text        :', repr(text)
print 'Pattern     :', pattern
print 'Single Line :', single_line.
print 'Multline    :', multiline.fi
```

The pattern in the example matches the first or last word of the input. It matches `line.` at the end of the string, even though there is no newline.

```
$ python re_flags_multiline.py

Text        : 'This is some text --
Pattern     : (^\w+)|(\w+\S*$)
Single Line : [('This', ''), ('', '
Multline    : [('This', ''), ('', '
```

`DOTALL` is the other flag related to multiline text. Normally the dot character `.` matches everything in the input text except a newline character. The flag allows dot to match newlines as well.

```
import re
```

```
text = 'This is some text -- with p
pattern = r'.+'
no_newlines = re.compile(pattern)
dotall = re.compile(pattern, re.DOT

print 'Text        :', repr(text)
print 'Pattern     :', pattern
print 'No newlines :', no_newlines.
print 'Dotall      :', dotall.finda
```

Without the flag, each line of the input text matches the pattern separately. Adding the flag causes the entire string to be consumed.

```
$ python re_flags_dotall.py

Text        : 'This is some text --
Pattern     : .+
No newlines : ['This is some text -
Dotall      : ['This is some text -
```

Unicode

Under Python 2, **`str`** objects use the ASCII character set, and regular expression processing assumes that the pattern and input text are both ASCII. The escape codes described earlier are defined in terms of ASCII by default. Those assumptions mean that the pattern `\w+` will match the word "French" but not "Français", since the `ç` is not part of the ASCII character set. To enable Unicode matching in Python 2, add the **`UNICODE`** flag when compiling the pattern.

```
import re
import codecs
import sys

# Set standard output encoding to U
sys.stdout = codecs.getwriter('UTF-

text = u'Français złoty Österreich'
pattern = ur'\w+'
ascii_pattern = re.compile(pattern)
unicode_pattern = re.compile(patter

print 'Text    :', text
print 'Pattern :', pattern
print 'ASCII   :', u', '.join(ascii
print 'Unicode :', u', '.join(unico
```

The other escape sequences (`\W`, `\b`, `\B`, `\d`, `\D`, `\s`, and `\S`) are also processed differently for Unicode text. Instead of assuming the members

of the character set identified by the escape sequence, the regular expression engine consults the Unicode database to find the properties of each character.

```
$ python re_flags_unicode.py

Text    : Français złoty Österreich
Pattern : \w+
ASCII   : Fran, ais, z, oty, sterre
Unicode : Français, złoty, Österrei
```

Note: Python 3 uses Unicode for all strings by default, so the flag is not necessary.

Verbose Expression Syntax

The compact format of regular expression syntax can become a hindrance as expressions grow more complicated. As the number of groups in your expression increases, you will have trouble keeping track of why each element is needed and how exactly the parts of the expression interact. Using named groups helps mitigate these issues, but a better solution is to use *verbose mode* expressions, which allow you to add comments and extra whitespace.

A pattern to validate email addresses will illustrate how verbose mode makes working with regular expressions easier. The first version recognizes addresses that end in one of three top-level domains, `.com`, `.org`, and `.edu`.

```
import re

address = re.compile('[\w\d.+-]+@([

candidates = [
    u'first.last@example.com',
    u'first.last+category@gmail.com
    u'valid-address@mail.example.co
    u'not-valid@example.foo',
    ]

for candidate in candidates:
    print
    print 'Candidate:', candidate
    match = address.search(candidat
    if match:
        print '  Matches'
    else:
        print '  No match'
```

This expression is already complex. There are several character classes, groups, and repetition expressions.

```
$ python re_email_compact.py


Candidate: first.last@example.com
  Matches

Candidate: first.last+category@gmai
  Matches

Candidate: valid-address@mail.examp
  Matches

Candidate: not-valid@example.foo
  No match
```

Converting the expression to a more verbose format will make it easier to extend.

```
import re

address = re.compile(
    '''
    [\w\d.+-]+       # username
    @
    ([\w\d.]+\.)+    # domain name
    (com|org|edu)    # we should su
    ''',
    re.UNICODE | re.VERBOSE)

candidates = [
    u'first.last@example.com',
    u'first.last+category@gmail.com
    u'valid-address@mail.example.co
    u'not-valid@example.foo',
    ]

for candidate in candidates:
    print
    print 'Candidate:', candidate
    match = address.search(candidat
    if match:
        print '  Matches'
    else:
        print '  No match'
```

The expression matches the same inputs, but in this extended format it is easier to read. The comments also help identify different parts of the pattern so that it can be expanded to match more inputs.

```
$ python re_email_verbose.py


Candidate: first.last@example.com
  Matches
```

```
Candidate: first.last+category@gmai
  Matches

Candidate: valid-address@mail.examp
  Matches

Candidate: not-valid@example.foo
  No match
```

This expanded version parses inputs that include a person's name and email address, as might appear in an email header. The name comes first and stands on its own, and the email address follows surrounded by angle brackets (< and >).

```
import re

address = re.compile(
    '''

    # A name is made up of letters,
    # abbreviations and middle init
    ((?P<name>
       ([\w.,]+\s+)*[\w.,]+)
       \s*
       # Email addresses are wrappe
       # but we only want one if we
       # the start bracket in this
       <
    )? # the entire name is optiona

    # The address itself: username@
    (?P<email>
      [\w\d.+-]+       # username
      @
      ([\w\d.]+\.)+    # domain nam
      (com|org|edu)    # limit the
    )

    >? # optional closing angle bra
    ''',
    re.UNICODE | re.VERBOSE)

candidates = [
    u'first.last@example.com',
    u'first.last+category@gmail.com
    u'valid-address@mail.example.co
    u'not-valid@example.foo',
    u'First Last <first.last@exampl
    u'No Brackets first.last@exampl
    u'First Last',
    u'First Middle Last <first.last(
    u'First M. Last <first.last@exa
    u'<first.last@example.com>',
    ]

for candidate in candidates:
    print
    print 'Candidate:', candidate
    match = address.search(candidat
    if match:
        print '  Match name :', mat
        print '  Match email:', mat
```

```
    else:
        print '  No match'
```

As with other programming languages, the ability to insert comments into verbose regular expressions helps with their maintainability. This final version includes implementation notes to future maintainers and whitespace to separate the groups from each other and highlight their nesting level.

```
$ python re_email_with_name.py


Candidate: first.last@example.com
  Match name : None
  Match email: first.last@example.c

Candidate: first.last+category@gmai
  Match name : None
  Match email: first.last+category@

Candidate: valid-address@mail.examp
  Match name : None
  Match email: valid-address@mail.e

Candidate: not-valid@example.foo
  No match

Candidate: First Last <first.last@e
  Match name : First Last
  Match email: first.last@example.c

Candidate: No Brackets first.last@e
  Match name : None
  Match email: first.last@example.c

Candidate: First Last
  No match

Candidate: First Middle Last <first
  Match name : First Middle Last
  Match email: first.last@example.c

Candidate: First M. Last <first.las
  Match name : First M. Last
  Match email: first.last@example.c

Candidate: <first.last@example.com>
  Match name : None
  Match email: first.last@example.c
```

Embedding Flags in Patterns

In situations where you cannot add flags when compiling an expression, such as when you are passing a pattern to a library function that will compile it later, you can embed the flags inside the expression string itself. For example, to turn

case-insensitive matching on, add `(?i)` to the beginning of the expression.

```
import re

text = 'This is some text -- with pu
pattern = r'(?i)\bT\w+'
regex = re.compile(pattern)

print 'Text      :', text
print 'Pattern   :', pattern
print 'Matches   :', regex.findall(
```

Because the options control the way the entire expression is evaluated or parsed, they should always come at the beginning of the expression.

```
$ python re_flags_embedded.py

Text      : This is some text -- wi
Pattern   : (?i)\bT\w+
Matches   : ['This', 'text']
```

The abbreviations for all of the flags are:

Flag	Abbreviation
IGNORECASE	i
MULTILINE	m
DOTALL	s
UNICODE	u
VERBOSE	x

Embedded flags can be combined by placing them within the same group. For example, `(?imu)` turns on case-insensitive matching for multiline Unicode strings.

Looking Ahead, or Behind

There are many cases where it is useful to match a part of a pattern only if some other part will also match. For example, in the email parsing expression the angle brackets were each marked as optional. Really, though, the brackets should be paired, and the expression should only match if both are present, or neither are. This modified version of the expression uses a *positive look ahead* assertion to match the pair. The look ahead assertion syntax is `(?=pattern)`.

```
import re

address = re.compile(
    '''
    # A name is made up of letters,
    # abbreviations and middle init
    ((?P<name>
       ([\w.,]+\s+)*[\w.,]+
     )
     \s+
    ) # name is no longer optional

    # LOOKAHEAD
    # Email addresses are wrapped i
    # the brackets if they are both
    (?= (<.*>$)       # remainder w
        |
        ([^<].*[^>]$) # remainder *
      )

    <? # optional opening angle bra

    # The address itself: username@
    (?P<email>
      [\w\d.+-]+       # username
      @
      ([\w\d.]+\.)+    # domain nam
      (com|org|edu)    # limit the
    )

    >? # optional closing angle bra
    ''',
    re.UNICODE | re.VERBOSE)

candidates = [
    u'First Last <first.last@exampl
    u'No Brackets first.last@exampl
    u'Open Bracket <first.last@exam
    u'Close Bracket first.last@exam
    ]

for candidate in candidates:
    print
    print 'Candidate:', candidate
    match = address.search(candidat
    if match:
        print '  Match name :', mat
        print '  Match email:', mat
    else:
        print '  No match'
```

There are several important changes in this version of the expression. First, the name portion is no longer optional. That means stand-alone addresses do not match, but it also prevents improperly formatted name/address combinations from matching. The positive look ahead rule after the "name" group asserts that the remainder of the string is either wrapped with a pair of angle brackets, or there is not a mismatched bracket; the brackets are either both present or neither is. The look ahead is expressed as a group, but the match for a look

ahead group does not consume any of the input text, so the rest of the pattern picks up from the same spot after the look ahead matches.

```
$ python re_look_ahead.py


Candidate: First Last <first.last@e
  Match name : First Last
  Match email: first.last@example.c

Candidate: No Brackets first.last@e
  Match name : No Brackets
  Match email: first.last@example.c

Candidate: Open Bracket <first.last(
  No match

Candidate: Close Bracket first.last(
  No match
```

A *negative look ahead* assertion (`(?!pattern)`) says that the pattern does not match the text following the current point. For example, the email recognition pattern could be modified to ignore `noreply` mailing addresses commonly used by automated systems.

```
import re

address = re.compile(
    '''
    ^

    # An address: username@domain.t

    # Ignore noreply addresses
    (?!noreply@.*$)

    [\w\d.+-]+       # username
    @
    ([\w\d.]+\.)+    # domain name
    (com|org|edu)    # limit the al

    $
    ''',
    re.UNICODE | re.VERBOSE)

candidates = [
    u'first.last@example.com',
    u'noreply@example.com',
    ]

for candidate in candidates:
    print
    print 'Candidate:', candidate
    match = address.search(candidat
    if match:
        print '  Match:', candidate
    else:
        print '  No match'
```

The address starting `noreply` does not match the pattern, since the look ahead assertion fails.

```
$ python re_negative_look_ahead.py


Candidate: first.last@example.com
  Match: first.last@example.com

Candidate: noreply@example.com
  No match
```

Instead of looking ahead for `noreply` in the username portion of the email address, the pattern can also be written using a *negative look behind* assertion after the username is matched using the syntax `(?<!pattern)`.

```
import re

address = re.compile(
    '''
    ^

    # An address: username@domain.t

    [\w\d.+-]+       # username

    # Ignore noreply addresses
    (?<!noreply)

    @
    ([\w\d.]+\.)+    # domain name
    (com|org|edu)    # limit the al

    $
    ''',
    re.UNICODE | re.VERBOSE)

candidates = [
    u'first.last@example.com',
    u'noreply@example.com',
    ]

for candidate in candidates:
    print
    print 'Candidate:', candidate
    match = address.search(candidat
    if match:
        print '  Match:', candidate
    else:
        print '  No match'
```

Looking backwards works a little differently than looking ahead, in that the expression must use a fixed length pattern. Repetitions are allowed, as long as there is a fixed number (no wildcards or ranges).

```
$ python re_negative_look_behind.py


Candidate: first.last@example.com
  Match: first.last@example.com

Candidate: noreply@example.com
  No match
```

A *positive look behind* assertion can be used to find text following a pattern using the syntax `(?<=pattern)`. For example, this expression finds Twitter handles.

```
import re

twitter = re.compile(
    '''
    # A twitter handle: @username
    (?<=@)
    ([\w\d_]+)       # username
    ''',
    re.UNICODE | re.VERBOSE)

text = '''This text includes two Tw
One for @ThePSF, and one for the au
'''

print text
for match in twitter.findall(text):
    print 'Handle:', match
```

The pattern matches sequences of characters that can make up a Twitter handle, as long as they are preceded by an `@`.

```
$ python re_look_behind.py

This text includes two Twitter hand
One for @ThePSF, and one for the au

Handle: ThePSF
Handle: doughellmann
```

Self-referencing Expressions

Matched values can be used in later parts of an expression. For example, the email example can be updated to match only addresses composed of the first and last name of the person by including back-references to those groups. The easiest way to achieve this is by referring to the previously matched group by id number, using `\num`.

```
import re

address = re.compile(
    r'''

    # The regular name
    (\w+)               # first nam
    \s+
    (([\w.]+)\s+)?      # optional
    (\w+)               # last name

    \s+

    <

    # The address: first_name.last_
    (?P<email>
      \1                # first name
      \.
      \4                # last name
      @
      ([\w\d.]+\.)+     # domain nam
      (com|org|edu)     # limit the
    )

    >
    ''',
    re.UNICODE | re.VERBOSE | re.IG

candidates = [
    u'First Last <first.last@exampl
    u'Different Name <first.last@ex
    u'First Middle Last <first.last(
    u'First M. Last <first.last@exa
    ]

for candidate in candidates:
    print
    print 'Candidate:', candidate
    match = address.search(candidat
    if match:
        print '  Match name :', mat
        print '  Match email:', mat
    else:
        print '  No match'
```

Although the syntax is simple, creating back-references by numerical id has a couple of disadvantages. From a practical standpoint, as the expression changes, you must count the groups again and possibly update every reference. The other disadvantage is that only 99 references can be made this way, because if the id number is three digits long it will be interpreted as an octal character value instead of a group reference. On the other hand, if you have more than 99 groups in your expression you will have more serious maintenance challenges than not being able to refer to some of the groups in the expression.

```
$ python re_refer_to_group.py


Candidate: First Last <first.last@e
  Match name : First Last
  Match email: first.last@example.c

Candidate: Different Name <first.la
  No match

Candidate: First Middle Last <first
  Match name : First Last
  Match email: first.last@example.c

Candidate: First M. Last <first.las
  Match name : First Last
  Match email: first.last@example.c
```

Python's expression parser includes an extension that uses `(?P=name)` to refer to the value of a named group matched earlier in the expression.

```
import re

address = re.compile(
    '''

    # The regular name
    (?P<first_name>\w+)
    \s+
    (([\w.]+)\s+)?      # optional
    (?P<last_name>\w+)

    \s+

    <

    # The address: first_name.last_
    (?P<email>
      (?P=first_name)
      \.
      (?P=last_name)
      @
      ([\w\d.]+\.)+    # domain nam
      (com|org|edu)    # limit the
    )

    >
    ''',
    re.UNICODE | re.VERBOSE | re.IG

candidates = [
    u'First Last <first.last@exampl
    u'Different Name <first.last@ex
    u'First Middle Last <first.last(
    u'First M. Last <first.last@exa
    ]

for candidate in candidates:
    print
    print 'Candidate:', candidate
    match = address.search(candidat
    if match:
        print '  Match name :', mat
```

```
        print '  Match email:', mat
    else:
        print '  No match'
```

The address expression is compiled with the **IGNORECASE** flag on, since proper names are normally capitalized but email addresses are not.

```
$ python re_refer_to_named_group.py


Candidate: First Last <first.last@e
  Match name : First Last
  Match email: first.last@example.c

Candidate: Different Name <first.la
  No match

Candidate: First Middle Last <first
  Match name : First Last
  Match email: first.last@example.c

Candidate: First M. Last <first.las
  Match name : First Last
  Match email: first.last@example.c
```

The other mechanism for using back-references in expressions lets you choose a different pattern based on whether or not a previous group matched. The email pattern can be corrected so that the angle brackets are required if a name is present, and not if the email address is by itself. The syntax for testing to see if a group has matched is `(?(id)yes-expression|no-expression)`, where *id* is the group name or number, *yes-expression* is the pattern to use if the group has a value and *no-expression* is the pattern to use otherwise.

```
import re

address = re.compile(
    '''
    ^

    # A name is made up of letters,
    # abbreviations and middle init
    (?P<name>
       ([\w.]+\s+)*[\w.]+
     )?
    \s*

    # Email addresses are wrapped i
    # the brackets if we found a na
    (?(name)
      # remainder wrapped in angle
      (?P<brackets>(?=(<.*>$)))
      |
```

```
      # remainder does not include
      (?=([^<].*[^>]$))
     )

    # Only look for a bracket if ou
    # of them.
    (?(brackets)<|\s*)

    # The address itself: username@
    (?P<email>
      [\w\d.+-]+       # username
      @
      ([\w\d.]+\.)+    # domain nam
      (com|org|edu)    # limit the
     )

    # Only look for a bracket if ou
    # of them.
    (?(brackets)>|\s*)

    $
    ''',
    re.UNICODE | re.VERBOSE)

candidates = [
    u'First Last <first.last@exampl
    u'No Brackets first.last@exampl
    u'Open Bracket <first.last@exam
    u'Close Bracket first.last@exam
    u'no.brackets@example.com',
    ]

for candidate in candidates:
    print
    print 'Candidate:', candidate
    match = address.search(candidat
    if match:
        print '  Match name :', mat
        print '  Match email:', mat
    else:
        print '  No match'
```

This version of the email address parser uses two tests. If the `name` group matches, then the look ahead assertion requires both angle brackets and sets up the `brackets` group. If `name` is not matched, the assertion requires the rest of the text not have angle brackets around it. Later, if the `brackets` group is set, the actual pattern matching code consumes the brackets in the input using literal patterns, otherwise it consumes any blank space.

```
$ python re_id.py


Candidate: First Last <first.last@e
  Match name : First Last
  Match email: first.last@example.c

Candidate: No Brackets first.last@e
  No match
```

```
Candidate: Open Bracket <first.last(
  No match

Candidate: Close Bracket first.last(
  No match

Candidate: no.brackets@example.com
  Match name : None
  Match email: no.brackets@example.(
```

Modifying Strings with Patterns

In addition to searching through text, `re` also supports modifying text using regular expressions as the search mechanism, and the replacements can reference groups matched in the regex as part of the substitution text. Use **`sub()`** to replace all occurances of a pattern with another string.

```
import re

bold = re.compile(r'\*{2}(.*?)\*{2}

text = 'Make this **bold**.  This *

print 'Text:', text
print 'Bold:', bold.sub(r'<b>\1</b>
```

References to the text matched by the pattern can be inserted using the `\num` syntax used for back-references above.

```
$ python re_sub.py

Text: Make this **bold**.  This **t(
Bold: Make this <b>bold</b>.  This
```

To use named groups in the substitution, use the syntax `\g<name>`.

```
import re

bold = re.compile(r'\*{2}(?P<bold_t(

text = 'Make this **bold**.  This *

print 'Text:', text
print 'Bold:', bold.sub(r'<b>\g<bol(
```

The `\g<name>` syntax also works with numbered references, and using it eliminates any ambiguity between group numbers and surrounding literal digits.

```
$ python re_sub_named_groups.py

Text: Make this **bold**.  This **t
Bold: Make this <b>bold</b>.  This
```

Pass a value to *count* to limit the number of substitutions performed.

```
import re

bold = re.compile(r'\*{2}(.*?)\*{2}

text = 'Make this **bold**.  This *

print 'Text:', text
print 'Bold:', bold.sub(r'<b>\1</b>
```

Only the first substitution is made because *count* is `1`.

```
$ python re_sub_count.py

Text: Make this **bold**.  This **t
Bold: Make this <b>bold</b>.  This
```

`subn()` works just like **`sub()`** except that it returns both the modified string and the count of substitutions made.

```
import re

bold = re.compile(r'\*{2}(.*?)\*{2}

text = 'Make this **bold**.  This *

print 'Text:', text
print 'Bold:', bold.subn(r'<b>\1</b
```

The search pattern matches twice in the example.

```
$ python re_subn.py

Text: Make this **bold**.  This **t
Bold: ('Make this <b>bold</b>.  This
```

Splitting with Patterns

`str.split()` is one of the most frequently used methods for breaking apart strings to parse them. It only supports using literal values as separators, though, and sometimes a regular

expression is necessary if the input is not consistently formatted. For example, many plain text markup languages define paragraph separators as two or more newline (`\n`) characters. In this case, **`str.split()`** cannot be used because of the "or more" part of the definition.

A strategy for identifying paragraphs using **`findall()`** would use a pattern like `(.+?)\n{2,}`.

```
import re

text = 'Paragraph one\non two lines
for num, para in enumerate(re.finda
    print num, repr(para)
    print
```

That pattern fails for paragraphs at the end of the input text, as illustrated by the fact that "Paragraph three." is not part of the output.

```
$ python re_paragraphs_findall.py

0 'Paragraph one\non two lines.'

1 'Paragraph two.'
```

Extending the pattern to say that a paragraph ends with two or more newlines, or the end of input, fixes the problem but makes the pattern more complicated. Converting to **`re.split()`** instead of **`re.findall()`** handles the boundary condition automatically and keeps the pattern simple.

```
import re

text = 'Paragraph one\non two lines

print 'With findall:'
for num, para in enumerate(re.finda
    print num, repr(para)
    print

print
print 'With split:'
for num, para in enumerate(re.split
    print num, repr(para)
    print
```

The pattern argument to **`split()`** expresses the markup specification more precisely: Two or more newline characters mark a separator point between paragraphs in the input string.

```
$ python re_split.py

With findall:
0 ('Paragraph one\non two lines.',

1 ('Paragraph two.', '\n\n\n')

2 ('Paragraph three.', '')


With split:
0 'Paragraph one\non two lines.'

1 'Paragraph two.'

2 'Paragraph three.'
```

Enclosing the expression in parentheses to define a group causes **`split()`** to work more like **`str.partition()`**, so it returns the separator values as well as the other parts of the string.

```
import re

text = 'Paragraph one\non two lines

print
print 'With split:'
for num, para in enumerate(re.split
    print num, repr(para)
    print
```

The output now includes each paragraph, as well as the sequence of newlines separating them.

```
$ python re_split_groups.py


With split:
0 'Paragraph one\non two lines.'

1 '\n\n'

2 'Paragraph two.'

3 '\n\n\n'

4 'Paragraph three.'
```

See also:

re

The standard library documentation for this module.

Regular Expression HOWTO

Andrew Kuchling's introduction to regular expressions for Python developers.

Kodos

An interactive regular expression testing tool by Phil Schwartz.

Python Regular Expression Testing Tool

A web-based tool for testing regular expressions created by David Naffziger at BrandVerity.com. Inspired by Kodos.

Wikipedia: Regular expression

General introduction to regular expression concepts and techniques.

`locale`

Use the `locale` module to set your language configuration when working with Unicode text.

`unicodedata`

Programmatic access to the Unicode character property database.