re — Regular Expressions

Purpose: Searching within and changing text using formal patterns.

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 “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, support regular expressions through extension libraries.

Multiple open source implementations of regular expressions exist, 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 “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. The `search()` function 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`.

Each `Match` object 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.

```
# re_simple_match.py

import re

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

match = re.search(pattern, text)

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

print('Found "{}"\nin "{}"\nfrom {} to {} ("{}")'.format(
    match.re.pattern, match.string, s, e, text[s:e]))
```

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

```
$ python3 re_simple_match.py

Found "this"
in "Does this text match the pattern?"
from 5 to 9 ("this")
```

Compiling Expressions

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

```
# re_simple_compiled.py

import re

# Precompile the patterns
regexes = [
    re.compile(p)
    for p in ['this', 'that']
]
text = 'Does this text match the pattern?'

print('Text: {!r}\n'.format(text))

for regex in regexes:
    print('Seeking "{}" ->'.format(regex.pattern),
          end=' ')

    if regex.search(text):
        print('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 avoids the overhead associated with cache lookup. Another advantage of using compiled expressions is that by precompiling all of the expressions when the module is loaded, the compilation work is shifted to application start time, instead of occurring at a point where the program may be responding to a user action.

```
$ python3 re_simple_compiled.py

Text: 'Does this text match the pattern?'

Seeking "this" -> match!
Seeking "that" -> no match
```

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.

```
# re_findall.py

import re

text = 'abbaaabbbbaaaaa'

pattern = 'ab'

for match in re.findall(pattern, text):
    print('Found {!r}'.format(match))
```

This example input string includes two instances of ab.

```
$ python3 re_findall.py

Found 'ab'
Found 'ab'
```

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

```
# re_finditer.py

import re

text = 'abbaaabbbbaaaaa'

pattern = 'ab'
```

```
for match in re.finditer(pattern, text):
    s = match.start()
    e = match.end()
    print('Found {!r} at {:d}:{:d}'.format(
        text[s:e], s, e))
```

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

```
$ python3 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 do not require every literal character to be present in the pattern. All of these features are used by combining literal text values with *meta-characters* that are part of the regular expression pattern syntax implemented by re.

```
# re_test_patterns.py

import re


def test_patterns(text, patterns):
    """Given source text and a list of patterns, look for
    matches for each pattern within the text and print
    them to stdout.
    """
    # Look for each pattern in the text and print the results
    for pattern, desc in patterns:
        print("'{}' ({})\n".format(pattern, desc))
        print("  '{}'".format(text))
        for match in re.finditer(pattern, text):
            s = match.start()
            e = match.end()
            substr = text[s:e]
            n_backslashes = text[:s].count('\\')
            prefix = '.' * (s + n_backslashes)
            print("  {}'{}'".format(prefix, substr))
        print()
    return


if __name__ == '__main__':
    test_patterns('abbaaabbbbaaaaa',
                  [('ab', "'a' followed by 'b'"),
                   ])
```

The following examples will use test_patterns() to explore how variations in patterns change the way they match the same input text. The output shows the input text and the substring range from each portion of the input that matches the pattern.

```
$ python3 re_test_patterns.py

'ab' ('a' followed by 'b')

  'abbaaabbbbaaaaa'
  'ab'
  .....'ab'
```

Repetition

There are five ways to express repetition in a pattern. A pattern followed by the meta-character * is repeated zero or more times (allowing a pattern to repeat zero times means it does not need to appear at all to match). If the * is replaced with +, 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 the number of times the pattern should repeat. 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 must appear at least m times, with no maximum.

```
# re_repetition.py

from re_test_patterns import test_patterns

test_patterns(
    'abbaabbba',
    [('ab*', 'a followed by zero or more b'),
     ('ab+', 'a followed by one or more b'),
     ('ab?', 'a followed by zero or one b'),
     ('ab{3}', 'a followed by three b'),
     ('ab{2,3}', 'a followed by two to three b')],
)
```

There are more matches for ab* and ab? than ab+.

```
$ python3 re_repetition.py

'ab*' (a followed by zero or more b)

  'abbaabbba'
  'abb'
  ...'a'
  ....'abbb'
  ........'a'

'ab+' (a followed by one or more b)

  'abbaabbba'
  'abb'
  ....'abbb'

'ab?' (a followed by zero or one b)

  'abbaabbba'
  'ab'
  ...'a'
  ....'ab'
  ........'a'

'ab{3}' (a followed by three b)

  'abbaabbba'
  ....'abbb'

'ab{2,3}' (a followed by two to three b)

  'abbaabbba'
  'abb'
  ....'abbb'
```

When processing a repetition instruction, re will usually 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 ?.

```
# re_repetition_non_greedy.py

from re_test_patterns import test_patterns

test_patterns(
    'abbaabbba',
    [('ab*?', 'a followed by zero or more b'),
     ('ab+?', 'a followed by one or more b'),
     ('ab??', 'a followed by zero or one b'),
     ('ab{3}?', 'a followed by three b'),
     ('ab{2,3}?', 'a followed by two to three b')],
)
```

Disabling greedy consumption of the input for any of the patterns where zero occurrences of b are allowed means the

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

```
$ python3 re_repetition_non_greedy.py

'ab*?' (a followed by zero or more b)

  'abbaabbba'
  'a'
  ...'a'
  ....'a'
  ........'a'

'ab+?' (a followed by one or more b)

  'abbaabbba'
  'ab'
  ....'ab'

'ab??' (a followed by zero or one b)

  'abbaabbba'
  'a'
  ...'a'
  ....'a'
  ........'a'

'ab{3}?' (a followed by three b)

  'abbaabbba'
  ....'abbb'

'ab{2,3}?' (a followed by two to three b)

  'abbaabbba'
  'abb'
  ....'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.

```
# re_charset.py

from re_test_patterns import test_patterns

test_patterns(
    'abbaabbba',
    [('[ab]', 'either a or b'),
     ('a[ab]+', 'a followed by 1 or more a or b'),
     ('a[ab]+?', 'a followed by 1 or more a or b, not greedy')],
)
```

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.

```
$ python3 re_charset.py

'[ab]' (either a or b)

  'abbaabbba'
  'a'
  .'b'
  ..'b'
  ...'a'
  ....'a'
  .....'b'
  ......'b'
  .......'b'
```

```
  ........'a'

'a[ab]+' (a followed by 1 or more a or b)

  'abbaabbba'
  'abbaabbba'

'a[ab]+?' (a followed by 1 or more a or b, not greedy)

  'abbaabbba'
  'ab'
  ...'aa'
```

A character set can also be used to exclude specific characters. The carat (^) means to look for characters that are not in the set following the carat.

```
# re_charset_exclude.py

from re_test_patterns import test_patterns

test_patterns(
    'This is some text -- with punctuation.',
    [('[^-. ]+', 'sequences without -, ., or space')],
)
```

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

```
$ python3 re_charset_exclude.py

'[^-. ]+' (sequences without -, ., or space)

  'This is some text -- with punctuation.'
  'This'
  .....'is'
  ........'some'
  ............'text'
  ....................'with'
  ........................'punctuation'
```

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

```
# re_charset_ranges.py

from re_test_patterns import test_patterns

test_patterns(
    'This is some text -- with punctuation.',
    [('[a-z]+', 'sequences of lowercase letters'),
     ('[A-Z]+', 'sequences of uppercase letters'),
     ('[a-zA-Z]+', 'sequences of letters of either case'),
     ('[A-Z][a-z]+', 'one uppercase followed by lowercase')],
)
```

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

```
$ python3 re_charset_ranges.py

'[a-z]+' (sequences of lowercase letters)

  'This is some text -- with punctuation.'
  .'his'
  .....'is'
  ........'some'
  ............'text'
  ....................'with'
  ........................'punctuation'
```

```
'[A-Z]+' (sequences of uppercase letters)

  'This is some text -- with punctuation.'
  'T'

'[a-zA-Z]+' (sequences of letters of either case)

  'This is some text -- with punctuation.'
  'This'
  .....'is'
  ........'some'
  ............'text'
  ....................'with'
  ........................'punctuation'

'[A-Z][a-z]+' (one uppercase followed by lowercase)

  'This is some text -- with punctuation.'
  'This'
```

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

```
# re_charset_dot.py

from re_test_patterns import test_patterns

test_patterns(
    'abbaabbba',
    [('a.', 'a followed by any one character'),
     ('b.', 'b followed by any one character'),
     ('a.*b', 'a followed by anything, ending in b'),
     ('a.*?b', 'a followed by anything, ending in b')],
)
```

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

```
$ python3 re_charset_dot.py

'a.' (a followed by any one character)

  'abbaabbba'
  'ab'
  ...'aa'

'b.' (b followed by any one character)

  'abbaabbba'
  .'bb'
  .....'bb'
  .......'ba'

'a.*b' (a followed by anything, ending in b)

  'abbaabbba'
  'abbaabbb'

'a.*?b' (a followed by anything, ending in b)

  'abbaabbba'
  'ab'
  ...'aab'
```

Escape Codes

An even more compact representation uses escape codes for several predefined character sets. The escape codes recognized by re are listed in the table below.

Regular Expression Escape Codes

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 difficult-to-read expressions. Using *raw* strings, which are created by prefixing the literal value with r, eliminates this problem and maintains readability.

```
# re_escape_codes.py

from re_test_patterns import test_patterns

test_patterns(
    'A prime #1 example!',
    [(r'\d+', 'sequence of digits'),
     (r'\D+', 'sequence of non-digits'),
     (r'\s+', 'sequence of whitespace'),
     (r'\S+', 'sequence of non-whitespace'),
     (r'\w+', 'alphanumeric characters'),
     (r'\W+', 'non-alphanumeric')],
)
```

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

```
$ python3 re_escape_codes.py

'\d+' (sequence of digits)

  'A prime #1 example!'
  '.........'1'

'\D+' (sequence of non-digits)

  'A prime #1 example!'
  'A prime #'
  '.........' example!'

'\s+' (sequence of whitespace)

  'A prime #1 example!'
  ' ' '
  '.......' '
  '.........' '

'\S+' (sequence of non-whitespace)

  'A prime #1 example!'
  'A'
  ..'prime'
  '........'#1'
  '..........'example!'

'\w+' (alphanumeric characters)

  'A prime #1 example!'
  'A'
  ..'prime'
  '.........'1'
```

```
  ..........'example'

'\W+' (non-alphanumeric)

  'A prime #1 example!'
  .' '
  '.......' #'
  '.........' '
  '..................'!'
```

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

```
# re_escape_escapes.py

from re_test_patterns import test_patterns

test_patterns(
    r'\d+ \D+ \s+',
    [(r'\\.\+', 'escape code')],
)
```

The pattern in this example escapes the backslash and plus characters, since both are meta-characters and have special meaning in a regular expression.

```
$ python3 re_escape_escapes.py

'\\.\+' (escape code)

  '\d+ \D+ \s+'
  '\d+'
  '.....'\D+'
  '..........'\s+'
```

Anchoring

In addition to describing the content of a pattern to match, the relative location can be specified in the input text where the pattern should appear by using *anchoring* instructions. the table below lists valid anchoring codes.

Regular Expression Anchoring Codes

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

```
# re_anchoring.py

from re_test_patterns import test_patterns

test_patterns(
    'This is some text -- with punctuation.',
    [(r'^\w+', 'word at start of string'),
     (r'\A\w+', 'word at start of string'),
     (r'\w+\S*$', 'word near end of string'),
     (r'\w+\S*\Z', 'word near end of string'),
     (r'\w*t\w*', 'word containing t'),
     (r'\bt\w+', 't at start of word'),
     (r'\w+t\b', 't at end of word'),
     (r'\Bt\B', 't, not start or end of word')],
)
```

The patterns in the example for matching words at the beginning and the 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

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.

```
$ python3 re_anchoring.py

'^\w+' (word at start of string)

  'This is some text -- with punctuation.'
  'This'

'\A\w+' (word at start of string)

  'This is some text -- with punctuation.'
  'This'

'\w+\S*$' (word near end of string)

  'This is some text -- with punctuation.'
  ........................'punctuation.'

'\w+\S*\Z' (word near end of string)

  'This is some text -- with punctuation.'
  ........................'punctuation.'

'\w*t\w*' (word containing t)

  'This is some text -- with punctuation.'
  ............'text'
  ..................'with'
  ........................'punctuation'

'\bt\w+' (t at start of word)

  'This is some text -- with punctuation.'
  ............'text'

'\w+t\b' (t at end of word)

  'This is some text -- with punctuation.'
  ............'text'

'\Bt\B' (t, not start or end of word)

  'This is some text -- with punctuation.'
  ......................'t'
  ............................'t'
  ..............................'t'
```

Constraining the Search

In situations where it is known in advance that only a subset of the full input should be searched, the regular expression match can be further constrained by telling `re` to limit the search range. For example, if the 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.

```
# re_match.py

import re

text = 'This is some text -- with punctuation.'
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.

```
$ python3 re_match.py

Text   : This is some text -- with punctuation.
Pattern: is
Match  : None
Search : <re.Match object; span=(2, 4), match='is'>
```

The `fullmatch()` method requires that the entire input string match the pattern.

```
# re_fullmatch.py

import re

text = 'This is some text -- with punctuation.'
pattern = 'is'

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

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

Here `search()` shows that the pattern does appear in the input, but it does not consume all of the input so `fullmatch()` does not report a match.

```
$ python3 re_fullmatch.py

Text       : This is some text -- with punctuation.
Pattern    : is
Search     : <re.Match object; span=(2, 4), match='is'>
Full match : None
```

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.

```
# re_search_substring.py

import re

text = 'This is some text -- with punctuation.'
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} = "{}"'.format(
        s, e - 1, text[s:e]))
    # Move forward in text for the next search
    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.

```
$ python3 re_search_substring.py

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

```
   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 isolates parts of the matching text, expanding those capabilities to create a parser. Groups are defined by enclosing patterns in parentheses.

```
# re_groups.py

from re_test_patterns import test_patterns

test_patterns(
    'abbaaabbbbaaaaa',
    [('a(ab)', 'a followed by literal ab'),
     ('a(a*b*)', 'a followed by 0-n a and 0-n b'),
     ('a(ab)*', 'a followed by 0-n ab'),
     ('a(ab)+', 'a followed by 1-n ab')],
)
```

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.

```
$ python3 re_groups.py

'a(ab)' (a followed by literal ab)

  'abbaaabbbbaaaaa'
  ....'aab'

'a(a*b*)' (a followed by 0-n a and 0-n b)

  'abbaaabbbbaaaaa'
  'abb'
  ...'aaabbbb'
  ..........'aaaaa'

'a(ab)*' (a followed by 0-n ab)

  'abbaaabbbbaaaaa'
  'a'
  ...'a'
  ....'aab'
  ..........'a'
  ...........'a'
  ............'a'
  .............'a'
  ..............'a'

'a(ab)+' (a followed by 1-n ab)

  'abbaaabbbbaaaaa'
  ....'aab'
```

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

```
# re_groups_match.py

import re

text = 'This is some text -- with punctuation.'

print(text)
print()

patterns = [
    (r'^(\w+)', 'word at start of string'),
```

```
    (r'^(\w+)', 'word at start of string'),
    (r'(\w+)\S*$', 'word at end, with optional punctuation'),
    (r'(\bt\w+)\W+(\w+)', 'word starting with t, another word'),
    (r'(\w+t)\b', 'word ending with t'),
]

for pattern, desc in patterns:
    regex = re.compile(pattern)
    match = regex.search(text)
    print("'{}' ({})\n".format(pattern, desc))
    print('  ', match.groups())
    print()
```

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

```
$ python3 re_groups_match.py

This is some text -- with punctuation.

'^(\w+)' (word at start of string)

   ('This',)

'(\w+)\S*$' (word at end, with optional punctuation)

   ('punctuation',)

'(\bt\w+)\W+(\w+)' (word starting with t, another word)

   ('text', 'with')

'(\w+t)\b' (word ending with t)

   ('text',)
```

To ask for the match of a single group, use the `group()` method. This is useful when grouping is being used to find parts of the string, but some of the parts matched by groups are not needed in the results.

```
# re_groups_individual.py

import re

text = 'This is some text -- with punctuation.'

print('Input text            :', text)

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

match = regex.search(text)
print('Entire match          :', match.group(0))
print('Word starting with "t":', match.group(1))
print('Word after "t" word   :', match.group(2))
```

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

```
$ python3 re_groups_individual.py

Input text            : This is some text -- with punctuation.
Pattern               : (\bt\w+)\W+(\w+)
Entire match          : text -- with
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)`.

```
# re_groups_named.py

import re

text = 'This is some text -- with punctuation.'

print(text)
print()

patterns = [
    r'^(?P<first_word>\w+)',
    r'(?P<last_word>\w+)\S*$',
    r'(?P<t_word>\bt\w+)\W+(?P<other_word>\w+)',
    r'(?P<ends_with_t>\w+t)\b',
]

for pattern in patterns:
    regex = re.compile(pattern)
    match = regex.search(text)
    print("'{}'".format(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.

```
$ python3 re_groups_named.py

This is some text -- with punctuation.

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

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

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

'(?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.

```
# re_test_patterns_groups.py

import re


def test_patterns(text, patterns):
    """Given source text and a list of patterns, look for
    matches for each pattern within the text and print
    them to stdout.
    """
    # Look for each pattern in the text and print the results
    for pattern, desc in patterns:
        print('{!r} ({})\n'.format(pattern, desc))
        print('  {!r}'.format(text))
        for match in re.finditer(pattern, text):
            s = match.start()
            e = match.end()
            prefix = ' ' * (s)
            print(
                '  {}{!r}{} '.format(prefix
```

```
                                    text[s:e],
                                    ' ' * (len(text) - e)),
                end=' ',
            )
            print(match.groups())
            if match.groupdict():
                print('{}{}'.format(
                    ' ' * (len(text) - s),
                    match.groupdict()),
                )
        print()
    return
```

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

```
# re_groups_nested.py

from re_test_patterns_groups import test_patterns

test_patterns(
    'abbaabbba',
    [(r'a((a*)(b*))', 'a followed by 0-n a and 0-n b')],
)
```

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

```
$ python3 re_groups_nested.py

'a((a*)(b*))' (a followed by 0-n a and 0-n b)

  'abbaabbba'
  'abb'        ('bb', '', 'bb')
    'aabbb'    ('abbb', 'a', 'bbb')
         'a'   ('', '', '')
```

Groups are also useful for specifying alternative patterns. Use the pipe symbol (`|`) to separate two patterns and indicate that either pattern should match. Consider the placement of the pipe 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.

```
# re_groups_alternative.py

from re_test_patterns_groups import test_patterns

test_patterns(
    'abbaabbba',
    [(r'a((a+)|(b+))', 'a then seq. of a or seq. of b'),
     (r'a((a|b)+)', 'a then seq. of [ab]')],
)
```

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.

```
$ python3 re_groups_alternative.py

'a((a+)|(b+))' (a then seq. of a or seq. of b)

  'abbaabbba'
  'abb'        ('bb', None, 'bb')
    'aa'       ('a', 'a', None)

'a((a|b)+)' (a then seq. of [ab])

  'abbaabbba'
  'abbaabbba'  ('bbaabbba', 'a')
```

Defining a group containing a subpattern is also useful in cases where the string matching the subpattern is not part of what should be extracted from the full text. These kinds of groups are called *non-capturing*. Non-capturing groups can be used to describe repetition patterns or alternatives, without isolating the matching portion of the string in the value returned. To create a non-capturing group, use the syntax `(?:pattern)`.

```
# re_groups_noncapturing.py

from re_test_patterns_groups import test_patterns

test_patterns(
    'abbaabbba',
    [(r'a((a+)|(b+))', 'capturing form'),
     (r'a((?:a+)|(?:b+))', 'noncapturing')],
)
```

In the following example, compare the groups returned for the capturing and non-capturing forms of a pattern that matches the same results.

```
$ python3 re_groups_noncapturing.py

'a((a+)|(b+))' (capturing form)

  'abbaabbba'
  'abb'        ('bb', None, 'bb')
    'aa'       ('a', 'a', None)

'a((?:a+)|(?:b+))' (noncapturing)

  'abbaabbba'
  'abb'        ('bb',)
    'aa'       ('a',)
```

Search Options

Option flags are used to change the way the matching engine processes an expression. The flags can be combined using a bitwise OR operation, then 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 uppercase and lowercase characters.

```
# re_flags_ignorecase.py

import re

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

print('Text:\n  {!r}'.format(text))
print('Pattern:\n  {}'.format(pattern))
print('Case-sensitive:')
for match in with_case.findall(text):
    print('  {!r}'.format(match))
print('Case-insensitive:')
for match in without_case.findall(text):
    print('  {!r}'.format(match))
```

Since the pattern includes the literal `T`, if `IGNORECASE` is not set, the only match is the word `This`. When case is ignored, `text` also matches.

```
$ python3 re_flags_ignorecase.py

Text:
  'This is some text -- with punctuation.'
Pattern:
  \bT\w+
```

```
  \b\w'
Case-sensitive:
  'This'
Case-insensitive:
  'This'
  'text'
```

Input with Multiple Lines

Two flags affect how searching in multi-line input works: `MULTILINE` and `DOTALL`. 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.

```
# re_flags_multiline.py

import re

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

print('Text:\n  {!r}'.format(text))
print('Pattern:\n  {}'.format(pattern))
print('Single Line :')
for match in single_line.findall(text):
    print('  {!r}'.format(match))
print('Multline    :')
for match in multiline.findall(text):
    print('  {!r}'.format(match))
```

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.

```
$ python3 re_flags_multiline.py

Text:
  'This is some text -- with punctuation.\nA second line.'
Pattern:
  (^\w+)|(\w+\S*$)
Single Line :
  ('This', '')
  ('', 'line.')
Multline    :
  ('This', '')
  ('', 'punctuation.')
  ('A', '')
  ('', 'line.')
```

`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 the dot to match newlines as well.

```
# re_flags_dotall.py

import re

text = 'This is some text -- with punctuation.\nA second line.'
pattern = r'.+'
no_newlines = re.compile(pattern)
dotall = re.compile(pattern, re.DOTALL)

print('Text:\n  {!r}'.format(text))
print('Pattern:\n  {}'.format(pattern))
print('No newlines :')
for match in no_newlines.findall(text):
    print('  {!r}'.format(match))
print('Dotall      :')
for match in dotall.findall(text):
    print('  {!r}'.format(match))
```

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

```
$ python3 re_flags_dotall.py

Text:
  'This is some text -- with punctuation.\nA second line.'
Pattern:
  .+
No newlines :
  'This is some text -- with punctuation.'
  'A second line.'
Dotall      :
  'This is some text -- with punctuation.\nA second line.'
```

Unicode

Under Python 3, `str` objects use the full Unicode character set, and regular expression processing on a `str` assumes that the pattern and input text are both Unicode. The escape codes described earlier are defined in terms of Unicode by default. Those assumptions mean that the pattern `\w+` will match both the words “French” and “Français”. To restrict escape codes to the ASCII character set, as was the default in Python 2, use the `ASCII` flag when compiling the pattern or when calling the module-level functions `search()` and `match()`.

```
# re_flags_ascii.py

import re

text = u'Français złoty Österreich'
pattern = r'\w+'
ascii_pattern = re.compile(pattern, re.ASCII)
unicode_pattern = re.compile(pattern)

print('Text    :', text)
print('Pattern :', pattern)
print('ASCII   :', list(ascii_pattern.findall(text)))
print('Unicode :', list(unicode_pattern.findall(text)))
```

The other escape sequences (`\W`, `\b`, `\B`, `\d`, `\D`, `\s`, and `\S`) are also processed differently for ASCII text. Instead of consulting the Unicode database to find the properties of each character, `re` uses the ASCII definition of the character set.

```
$ python3 re_flags_ascii.py

Text    : Français złoty Österreich
Pattern : \w+
ASCII   : ['Fran', 'ais', 'z', 'oty', 'sterreich']
Unicode : ['Français', 'złoty', 'Österreich']
```

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 an expression increases, it will be more work to keep 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 comments and extra whitespace to be embedded in the pattern.

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`, or `.edu`.

```
# re_email_compact.py

import re

address = re.compile('[\w\d.+-]+@([\w\d.]+\.)+(com|org|edu)')

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

```
]

for candidate in candidates:
    match = address.search(candidate)
    print('{:<30}  {}'.format(
        candidate, 'Matches' if match else 'No match')
    )
```

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

```
$ python3 re_email_compact.py

first.last@example.com          Matches
first.last+category@gmail.com   Matches
valid-address@mail.example.com  Matches
not-valid@example.foo           No match
```

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

```
# re_email_verbose.py

import re

address = re.compile(
    '''
    [\w\d.+-]+       # username
    @
    ([\w\d.]+\.)+    # domain name prefix
    (com|org|edu)    # TODO: support more top-level domains
    ''',
    re.VERBOSE)

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

for candidate in candidates:
    match = address.search(candidate)
    print('{:<30}  {}'.format(
        candidate, 'Matches' if match else '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.

```
$ python3 re_email_verbose.py

first.last@example.com          Matches
first.last+category@gmail.com   Matches
valid-address@mail.example.com  Matches
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 >).

```
# re_email_with_name.py

import re

address = re.compile(
    '''

    # A name is made up of letters, and may include "."
    # for title abbreviations and middle initials.
    ((?P<name>
       ([\w.,]+\s+)*[\w.,]+)
```

```
        \s*
        # Email addresses are wrapped in angle
        # brackets < >, but only if a name is
        # found, so keep the start bracket in this
        # group.
        <
    )? # the entire name is optional

    # The address itself: username@domain.tld
    (?P<email>
      [\w\d.+-]+       # username
      @
      ([\w\d.]+\.)+    # domain name prefix
      (com|org|edu)    # limit the allowed top-level domains
    )

    >? # optional closing angle bracket
    ''',
    re.VERBOSE)

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

for candidate in candidates:
    print('Candidate:', candidate)
    match = address.search(candidate)
    if match:
        print('  Name :', match.groupdict()['name'])
        print('  Email:', match.groupdict()['email'])
    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.

```
$ python3 re_email_with_name.py

Candidate: first.last@example.com
  Name : None
  Email: first.last@example.com
Candidate: first.last+category@gmail.com
  Name : None
  Email: first.last+category@gmail.com
Candidate: valid-address@mail.example.com
  Name : None
  Email: valid-address@mail.example.com
Candidate: not-valid@example.foo
  No match
Candidate: First Last <first.last@example.com>
  Name : First Last
  Email: first.last@example.com
Candidate: No Brackets first.last@example.com
  Name : None
  Email: first.last@example.com
Candidate: First Last
  No match
Candidate: First Middle Last <first.last@example.com>
  Name : First Middle Last
  Email: first.last@example.com
Candidate: First M. Last <first.last@example.com>
```

```
  Name : First M. Last
  Email: first.last@example.com
Candidate: <first.last@example.com>
  Name : None
  Email: first.last@example.com
```

Embedding Flags in Patterns

In situations where flags cannot be added when compiling an expression, such as when a pattern is passed as an argument to a library function that will compile it later, the flags can be embedded inside the expression string itself. For example, to turn case-insensitive matching on, add `(?i)` to the beginning of the expression.

```
# re_flags_embedded.py

import re

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

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

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

```
$ python3 re_flags_embedded.py

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

The abbreviations for all of the flags are listed in the table below.

Regular Expression Flag Abbreviations

Flag	Abbreviation
ASCII	a
IGNORECASE	i
MULTILINE	m
DOTALL	s
VERBOSE	x

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

Looking Ahead or Behind

In many cases, 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 marked as optional. Realistically, the brackets should be paired, and the expression should match only if both are present, or neither is. This modified version of the expression uses a *positive look ahead* assertion to match the pair. The look ahead assertion syntax is `(?=pattern)`.

```
# re_look_ahead.py
```

```
import re

address = re.compile(
    '''
    # A name is made up of letters, and may include "."
    # for title abbreviations and middle initials.
    ((?P<name>
       ([\w.,]+\s+)*[\w.,]+
     )
```

```
    (
     \s+
    ) # name is no longer optional

    # LOOKAHEAD
    # Email addresses are wrapped in angle brackets, but only
    # if both are present or neither is.
    (?= (<.*>$)       # remainder wrapped in angle brackets
        |
        ([^<].*[^>]$) # remainder *not* wrapped in angle brackets
      )

    <? # optional opening angle bracket

    # The address itself: username@domain.tld
    (?P<email>
      [\w\d.+-]+       # username
      @
      ([\w\d.]+\.)+    # domain name prefix
      (com|org|edu)    # limit the allowed top-level domains
    )

    >? # optional closing angle bracket
    ''',
    re.VERBOSE)

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

for candidate in candidates:
    print('Candidate:', candidate)
    match = address.search(candidate)
    if match:
        print('  Name :', match.groupdict()['name'])
        print('  Email:', match.groupdict()['email'])
    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 either the remainder of the string is wrapped with a pair of angle brackets, or there is not a mismatched bracket; either both of or neither of the brackets is present. 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.

```
$ python3 re_look_ahead.py

Candidate: First Last <first.last@example.com>
  Name : First Last
  Email: first.last@example.com
Candidate: No Brackets first.last@example.com
  Name : No Brackets
  Email: first.last@example.com
Candidate: Open Bracket <first.last@example.com
  No match
Candidate: Close Bracket first.last@example.com>
  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 the `noreply` mailing addresses commonly used by automated systems.

```
# re_negative_look_ahead.py

import re

address = re.compile(
```

```
    '''
    ^

    # An address: username@domain.tld

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

    [\w\d.+-]+       # username
    @
    ([\w\d.]+\.)+    # domain name prefix
    (com|org|edu)    # limit the allowed top-level domains

    $
    ''',
    re.VERBOSE)

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

for candidate in candidates:
    print('Candidate:', candidate)
    match = address.search(candidate)
    if match:
        print('  Match:', candidate[match.start():match.end()])
    else:
        print('  No match')
```

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

```
$ python3 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 alternatively be written using a *negative look behind* assertion after the username is matched using the syntax `(?<!pattern)`.

```
# re_negative_look_behind.py

import re

address = re.compile(
    '''
    ^

    # An address: username@domain.tld

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

    # Ignore noreply addresses
    (?<!noreply)

    @
    ([\w\d.]+\.)+    # domain name prefix
    (com|org|edu)    # limit the allowed top-level domains

    $
    ''',
    re.VERBOSE)

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

```
for candidate in candidates:
    print('Candidate:', candidate)
    match = address.search(candidate)
    if match:
        print('  Match:', candidate[match.start():match.end()])
    else:
        print('  No match')
```

Looking backward 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 of them (no wildcards or ranges).

```
$ python3 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)`. In the following example, the expression finds Twitter handles.

```
# re_look_behind.py

import re

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

text = '''This text includes two Twitter handles.
One for @ThePSF, and one for the author, @doughellmann.
'''

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 `@`.

```
$ python3 re_look_behind.py

This text includes two Twitter handles.
One for @ThePSF, and one for the author, @doughellmann.

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 names 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`.

```
# re_refer_to_group.py

import re

address = re.compile(
    r'''

    # The regular name
    (\w+)               # first name
    \s+
    (([\w.]+)\s+)?      # optional middle name or initial
    (\w+)               # last name
```

```
    (\w+)                # last name

    \s+

    <

    # The address: first_name.last_name@domain.tld
    (?P<email>
      \1               # first name
      \.
      \4               # last name
      @
      ([\w\d.]+\.)+    # domain name prefix
      (com|org|edu)    # limit the allowed top-level domains
    )

    >
    ''',
    re.VERBOSE | re.IGNORECASE)

candidates = [
    u'First Last <first.last@example.com>',
    u'Different Name <first.last@example.com>',
    u'First Middle Last <first.last@example.com>',
    u'First M. Last <first.last@example.com>',
]

for candidate in candidates:
    print('Candidate:', candidate)
    match = address.search(candidate)
    if match:
        print('  Match name :', match.group(1), match.group(4))
        print('  Match email:', match.group(5))
    else:
        print('  No match')
```

Although the syntax is simple, creating back-references by numerical ID has a few disadvantages. From a practical standpoint, as the expression changes, the groups must be counted again and every reference may need to be updated. Another disadvantage is that only 99 references can be made using the standard back-reference syntax `\n`, because if the ID number is three digits long, it will be interpreted as an octal character value instead of a group reference. Of course, if there are more than 99 groups in an expression, there will be more serious maintenance challenges than simply not being able to refer to all of them.

```
$ python3 re_refer_to_group.py

Candidate: First Last <first.last@example.com>
  Match name : First Last
  Match email: first.last@example.com
Candidate: Different Name <first.last@example.com>
  No match
Candidate: First Middle Last <first.last@example.com>
  Match name : First Last
  Match email: first.last@example.com
Candidate: First M. Last <first.last@example.com>
  Match name : First Last
  Match email: first.last@example.com
```

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.

```
# re_refer_to_named_group.py

import re

address = re.compile(
    '''

    # The regular name
    (?P<first_name>\w+)
    \s+
    (([\w.]+)\s+)?      # optional middle name or initial
```

```
    (?P<last_name>\w+)

    \s+

    <

    # The address: first_name.last_name@domain.tld
    (?P<email>
      (?P=first_name)
      \.
      (?P=last_name)
      @
      ([\w\d.]+\.)+    # domain name prefix
      (com|org|edu)    # limit the allowed top-level domains
    )

    >
    ''',
    re.VERBOSE | re.IGNORECASE)

candidates = [
    u'First Last <first.last@example.com>',
    u'Different Name <first.last@example.com>',
    u'First Middle Last <first.last@example.com>',
    u'First M. Last <first.last@example.com>',
]

for candidate in candidates:
    print('Candidate:', candidate)
    match = address.search(candidate)
    if match:
        print('  Match name :', match.groupdict()['first_name'],
              end=' ')
        print(match.groupdict()['last_name'])
        print('  Match email:', match.groupdict()['email'])
    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.

```
$ python3 re_refer_to_named_group.py

Candidate: First Last <first.last@example.com>
  Match name : First Last
  Match email: first.last@example.com
Candidate: Different Name <first.last@example.com>
  No match
Candidate: First Middle Last <first.last@example.com>
  Match name : First Last
  Match email: first.last@example.com
Candidate: First M. Last <first.last@example.com>
  Match name : First Last
  Match email: first.last@example.com
```

The other mechanism for using back-references in expressions chooses a different pattern based on whether a previous group matched. The email pattern can be corrected so that the angle brackets are required if a name is present, and not required if the email address is by itself. The syntax for testing whether 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.

```
# re_id.py

import re

address = re.compile(
    '''
    ^

    # A name is made up of letters, and may include "."
```

```
    # for title abbreviations and middle initials.
    (?P<name>
       ([\w.]+\s+)*[\w.]+
     )?
    \s*

    # Email addresses are wrapped in angle brackets, but
    # only if a name is found.
    (?(name)
      # remainder wrapped in angle brackets because
      # there is a name
      (?P<brackets>(?=(<.*>$)))
      |
      # remainder does not include angle brackets without name
      (?=([^<].*[^>]$))
     )

    # Look for a bracket only if the look-ahead assertion
    # found both of them.
    (?(brackets)<|\s*)

    # The address itself: username@domain.tld
    (?P<email>
      [\w\d.+-]+       # username
      @
      ([\w\d.]+\.)+    # domain name prefix
      (com|org|edu)    # limit the allowed top-level domains
     )

    # Look for a bracket only if the look-ahead assertion
    # found both of them.
    (?(brackets)>|\s*)

    $
    ''',
    re.VERBOSE)

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

for candidate in candidates:
    print('Candidate:', candidate)
    match = address.search(candidate)
    if match:
        print('  Match name :', match.groupdict()['name'])
        print('  Match email:', match.groupdict()['email'])
    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 to 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.

```
$ python3 re_id.py

Candidate: First Last <first.last@example.com>
  Match name : First Last
  Match email: first.last@example.com
Candidate: No Brackets first.last@example.com
  No match
Candidate: Open Bracket <first.last@example.com
  No match
Candidate: Close Bracket first.last@example.com>
  No match
Candidate: no.brackets@example.com
```

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

Modifying Strings with Patterns

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

```
# re_sub.py

import re

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

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

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

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

```
$ python3 re_sub.py

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

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

```
# re_sub_named_groups.py

import re

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

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

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

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

```
$ python3 re_sub_named_groups.py

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

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

```
# re_sub_count.py

import re

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

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

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

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

```
$ python3 re_sub_count.py

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

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

```
# re_subn.py

import re

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

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

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

The search pattern matches twice in the example.

```
$ python3 re_subn.py

Text: Make this **bold**.  This **too**.
Bold: ('Make this <b>bold</b>.  This <b>too</b>.', 2)
```

Splitting with Patterns

`str.split()` is one of the most frequently used methods for breaking apart strings to parse them. It supports only the use of 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,}`.

```
# re_paragraphs_findall.py

import re

text = '''Paragraph one
on two lines.

Paragraph two.


Paragraph three.'''

for num, para in enumerate(re.findall(r'(.+?)\n{2,}',
                                      text,
                                      flags=re.DOTALL)
                           ):
    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.

```
$ python3 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 simpler.

```
# re_split.py

import re

text = '''Paragraph one
```

```
on two lines.

Paragraph two.


Paragraph three.'''

print('With findall:')
for num, para in enumerate(re.findall(r'(.+?)(\n{2,}|$)',
                                      text,
                                      flags=re.DOTALL)):
    print(num, repr(para))
    print()

print()
print('With split:')
for num, para in enumerate(re.split(r'\n{2,}', text)):
    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.

```
$ python3 re_split.py

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

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.

```
# re_split_groups.py

import re

text = '''Paragraph one
on two lines.

Paragraph two.


Paragraph three.'''

print('With split:')
for num, para in enumerate(re.split(r'(\n{2,})', text)):
    print(num, repr(para))
    print()
```

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

```
$ python3 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

- Standard library documentation for re
- Regular Expression HOWTO – Andrew Kuchling's introduction to regular expressions for Python developers.
- Kodos – An interactive regular expression testing tool by Phil Schwartz.
- pythex – A web-based tool for testing regular expressions created by Gabriel Rodríguez. Inspired by Rubular.
- Wikipedia: Regular expression – General introduction to regular expression concepts and techniques.
- `locale` – Use the `locale` module to set the language configuration when working with Unicode text.
- `unicodedata` – Programmatic access to the Unicode character property database.

textwrap — Formatting Text Paragraphs

difflib — Compare Sequences

Quick Links

Finding Patterns in Text
Compiling Expressions
Multiple Matches
Pattern Syntax
Repetition
Character Sets
Escape Codes
Anchoring
Constraining the Search
Dissecting Matches with Groups
Search Options
Case-insensitive Matching
Input with Multiple Lines
Unicode
Verbose Expression Syntax
Embedding Flags in Patterns
Looking Ahead or Behind
Self-referencing Expressions
Modifying Strings with Patterns
Splitting with Patterns

This page was last updated 2018-12-09.

Navigation

textwrap — Formatting Text Paragraphs
difflib — Compare Sequences

The output from all the example programs from PyMOTW-3 has been generated with Python 3.7.1, unless otherwise noted. Some of the features described here may not be available in earlier versions of Python.

Looking for examples for Python 2?