Table of contents

---

1. Import Module

(go to top)

In [2]:
```
#import regex module
# see re documentation in last cell

import re

# x = re.compile('enter pattern here')
# y = x.search('enter tect to be match')
# z, = y.group() # print out matched geoup
```

2. Overview

(go to top)

In [5]:
```
# phone number pattern is all digits()
phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d') #raw string
mo = phoneNumRegex.search('My number is 415-555-4242.')
print('Phone number found: ' + mo.group())
```

```
Phone number found: 415-555-4242
```

3. Grouping w Parentheses

(go to top)

In [6]:
```
phoneNumRegex = re.compile(r'(\d\d\d)-(\d\d\d-\d\d\d\d)')
mo = phoneNumRegex.search('My number is 415-555-4242.')
```

In [7]:
```
mo.group()
```

Out[7]:
```
'415-555-4242'
```

In [10]:
```
mo.group(0)
```

Out[10]:
```
'415-555-4242'
```

In [17]:
```
mo.groups()
# returns a tuple of multiple values
```
Out[17]:
```
('415', '555-4242')
```

In [8]:
```
mo.group(1)
```
Out[8]:
```
'415'
```

In [9]:
```
mo.group(2)
```
Out[9]:
```
'555-4242'
```

In [19]:
```
areacode, mainNum = mo.groups()
print(areacode, 'and',  mainNum)
```
```
415 and 555-4242
```

---

4. Match parenthesis in your text

- \(and \)

(go to top)

In [20]:
```
phoneNumRegex = re.compile(r'(\(\d\d\d\))-(\d\d\d-\d\d\d\d)')
mo = phoneNumRegex.search('My number is (415)-555-4242.')
```

In [21]:
```
mo.groups()
```
Out[21]:
```
('(415)', '555-4242')
```

In [22]:
```
mo.group(1)
```
Out[22]:
```
'(415)'
```

In [23]:
```
mo.group(2)
```
Out[23]:
```
'555-4242'
```

5. Special Regex Characters

(go to top)

```
. ^ $ * + ? { } [ ] \ | ( )
```

```
to use these ad a \ in front
```

6. Matching groups with the pipe |

- The regular expression r'Batman|Tina Fey' will match either 'Batman' or 'Tina Fey'.
  When both Batman and Tina Fey occur in the searched string, the first occurrence of matching text will be returned as the Match object
  (go to top)

In [38]:
```
heroRegex = re.compile(r'Batman|Tina Fey')
```

In [39]:
```
mo1 = heroRegex.search('Batman and Tina Fey')
mo1.group()
```

Out[39]:
```
'Batman'
```

In [41]:
```
mo2 = heroRegex.search('Tina Fey and Batman')
mo2.group()
```

Out[41]:
```
'Tina Fey'
```

- You can also use the pipe to match one of several patterns as part of your regex.
  For example, say you wanted to match any of the strings 'Batman', 'Batmobile', 'Batcopter', and 'Batbat'.

In [47]:
```
batRegex = re.compile(r'Bat(man|mobile|copter|bat)')
```

In [50]:
```
mo = batRegex.search('Batmobile lost a wheel')
```

In [51]:
```
mo.group()
```

Out[51]:
```
'Batmobile'
```

In [52]:
```
mo.group(1)
```

Out[52]:
```
'mobile'
```

In [44]:
```
mo3 = heroRegex.findall('Tina Fey and Batman')
mo3
```

Out[44]:
```
['Tina Fey', 'Batman']
```

---

7. Optional Matching with Question Mark

- The ? character flags the group that precedes it as an optional part of the pattern
  That is, the regex should find a match regardless of whether that bit of text is there..

(go to top)

In [53]:
```
batRegex = re.compile(r'Bat(wo)?man')
```

In [54]:
```
mo1 = batRegex.search('The Adventures of Batman')
mo1.group()
```
Out[54]:
```
'Batman'
```

In [58]:
```
mo2 = batRegex.search('The Adventures of Batwoman')
mo2.group()
```
Out[58]:
```
'Batwoman'
```

____
____

In [59]:
```
phoneRegex = re.compile(r'(\d\d\d-)?\d\d\d-\d\d\d\d')
```

In [60]:
```
mo1 = phoneRegex.search('My number is 415-555-4242')
mo1.group()
```
Out[60]:
```
'415-555-4242'
```

In [62]:
```
mo1.group(1)
```
Out[62]:
```
'415-'
```

In [61]:
```
mo2 = phoneRegex.search('My number is 555-4242')
mo2.group()
```
Out[61]:
```
'555-4242'
```

---

8. Zero or more matching with star *

(go to top)

In [63]:
```
batRegex = re.compile(r'Bat(wo)*man')
```

In [64]:
```
mo1 = batRegex.search('The Adventures of Batman')
mo1.group()
```
Out[64]:
```
'Batman'
```

In [65]:
```
mo2 = batRegex.search('The Adventures of Batwoman')
mo2.group()
```
Out[65]:
```
'Batwoman'
```

In [70]:
```
mo2.group(1)
```
Out[70]:
```
'wo'
```

In [68]:
```
mo3 = batRegex.search('The Adventures of Batwowowowoman')
mo3.group()
```
Out[68]:
```
'Batwowowowoman'
```

In [69]:
```
mo3.group(1)
```
Out[69]:
```
'wo'
```

In [6]:
```
#match any and evry possible string
```

In [4]:
```
batRegex = re.compile(r'.*')
```

In [5]:
```
mo1 = batRegex.search('The Adventures of Batman')
mo1.group()
```
Out[5]:
```
'The Adventures of Batman'
```

9. One or more matching with plus +

- While * means “match zero or more,” the + (or plus) means “match one or more.”
  Unlike the star, which does not require its group to appear in the matched string, the group preceding a plus must appear at least once.
  It is not optional.

(go to top)

In [25]:
```
batRegex = re.compile(r'Bat(wo)+man')
```

```
In [77]: mo1 = batRegex.search('The Adventures of Batwoman')
         mo1.group()

Out[77]: 'Batwoman'
```

```
In [76]: mo2 = batRegex.search('The Adventures of Batwowowoman')
         mo2.group()

Out[76]: 'Batwowowoman'
```

- The regex Bat(wo)+man will not match the string 'The Adventures of Batman', because at least one wo is required by the plus sign.

```
In [78]: mo3 = batRegex.search('The Adventures of Batman')
         mo3.group()

---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-78-a63afd945b67> in <module>
      1 mo3 = batRegex.search('The Adventures of Batman')
----> 2 mo3.group()

AttributeError: 'NoneType' object has no attribute 'group'
```

10. Match Specific Repetitions with Braces

(go to top)

- For example, the regex (Ha){3} will match the string 'HaHaHa', but it will not match 'HaHa', since the latter has only two repeats of the (Ha) group.
  Instead of one number, you can specify a range by writing a minimum, a comma, and a maximum in between the braces. For example, the regex (Ha){3,5} will match 'HaHaHa', 'HaHaHaHa', and 'HaHaHaHaHa'.
  You can also leave out the first or second number in the braces to leave the minimum or maximum unbounded. For example, (Ha){3,} will match three or more instances of the (Ha) group, while (Ha){,5} will match zero
  to five instances. Braces can help make your regular expressions shorter. These two regular expressions match identical patterns:

In [81]:
```python
haRegex = re.compile(r'(Ha){3}')
```

In [83]:
```python
mo1 = haRegex.search('HaHaHa')
mo1.group()
```

Out[83]:
```
'HaHaHa'
```

In [89]:
```python
mo2 = haRegex.search('Ha')
mo2 == None
```

Out[89]:
```
True
```

In [91]:
```python
mo2.group()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call
last)
<ipython-input-91-fe964b823f27> in <module>
----> 1 mo2.group()

AttributeError: 'NoneType' object has no attribute 'group'
```

11. Greedy or nah?

- Since (Ha){3,5} can match three, four, or five instances of Ha in the string 'HaHaHaHaHa', you may wonder why the Match object’s call to group() in the previous brace example returns 'HaHaHaHaHa' instead of the shorter possibili- ties. After all, 'HaHaHa' and 'HaHaHaHa' are also valid matches of the regular expression (Ha){3,5}.
  Python’s regular expressions are greedy by default, which means that in ambiguous situations they will match the longest string possible. The non- greedy (also called lazy) version of the braces, which matches the shortest string possible, has the closing brace followed by a question mark.
  Enter the following into the interactive shell, and notice the differ- ence between the greedy and non-greedy forms of the braces searching the same string:

(go to top)

```
In [92]: greedyRegex = re.compile(r'(Ha){3,5}')
```

```
In [93]: mo1 = greedyRegex.search('HaHaHaHaHa')
         mo1.group()

Out[93]: 'HaHaHaHaHa'
```

```
In [94]: mo1.group(1)

Out[94]: 'Ha'
```

```
In [62]: nonGreedyRegex = re.compile(r'(Ha){3,5}?')
```

```
In [63]: mo2 = nonGreedyRegex.search('HaHaHaHaHa  HaHaHa')
         mo2.group()

Out[63]: 'HaHaHa'
```

12. `findall()` method

(go to top)

```
In [115]: phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d')
```

```
In [116]: mo = phoneNumRegex.search('Cell: 415-555-999 Work: 212-555-0000')
          mo.group()

Out[116]: '212-555-0000'
```


```
In [117]: phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d')
```

```
In [118]: mo1 = phoneNumRegex.findall('Cell: 415-555-9999 Work: 212-555-0000')
          mo1

Out[118]: ['415-555-9999', '212-555-0000']
```

If there are groups in the regular expression, then findall() will return a list of tuples.
Each tuple represents a found match, and its items are the matched strings for each group in the regex.
To see findall() in action, enter the following into the interactive shell (notice that the regular expres- sion being compiled now has groups in parentheses):

```
In [119]: phoneNumRegex = re.compile(r'(\d\d\d)-(\d\d\d)-(\d\d\d\d)')
```

```
In [120]: mo2 = phoneNumRegex.findall('Cell: 415-555-9999 Work: 212-555-0000')
          mo2

Out[120]: [('415', '555', '9999'), ('212', '555', '0000')]
```

```
In [ ]:
```

13. Character Classes

(go to top)

`\d Any numeric digit from 0 to 9.

\D Any character that is not a numeric digit from 0 to 9.

\w Any letter, numeric digit, or the underscore character. (Think of this as matching “word” characters.)

\W Any character that is not a letter, numeric digit, or the underscore character.

\s Any space, tab, or newline character. (Think of this as matching “space” characters.)

\S Any character that is not a space, tab, or newline.`

```
 Character classes are nice for shortening regular expressions. The
char- acter class [0-5] will match only the numbers 0 to 5; this is
much shorter than typing (0|1|2|3|4|5). Note that while \d matches
digits and \w matches digits, letters, and the underscore, there is
no shorthand character class that matches only letters. (Though you
can use the [a-zA-Z] character class, as explained next.)
```

In [124]:
```
xmasRegex = re.compile(r'\d+\s\w+')
xmasRegex.findall('12 drummers, 11 pipers, 10 lords, 9 ladies, 8 maids
['12 drummers', '11 pipers', '10 lords', '9 ladies', '8 maids', '7 swa
```

Out[124]:
```
['12 drummers',
 '11 pipers',
 '10 lords',
 '9 ladies',
 '8 maids',
 '7 swans',
 '6 geese',
 '5 rings',
 '4 birds',
 '3 hens',
 '2 doves',
 '1 partridge']
```

14. Make Your own Character Classes

- There are times when you want to match a set of characters but the short- hand character classes (\d, \w, \s, and so on) are too broad.
  You can define your own character class using square brackets.
  For example, the character class [aeiouAEIOU] will match any vowel,
  both lowercase and uppercase. Enter the following into the interactive shell:

(go to top)

In [130]:
```
vowelRegex = re.compile(r'[aeiouAEIOU]')
```

In [131]:
```
vowelRegex.findall('Robocop eats baby food')
```

Out[131]:
```
['o', 'o', 'o', 'e', 'a', 'a', 'o', 'o']
```

In [135]:
```
vowelRegex2 = re.compile(r'[a-zA-Z0-9]')
```

In [137]:
```
vowelRegex2.findall('Robocop eats baby food 345')
```

Out[137]:
```
['R',
 'o',
 'b',
 'o',
 'c',
 'o',
 'p',
 'e',
 'a',
 't',
 's',
 'b',
 'a',
 'b',
 'y',
 'f',
 'o',
 'o',
 'd',
 '3',
 '4',
 '5']
```

- Note that inside the square brackets, the normal regular expression symbols are not interpreted as such.
  This means you do not need to escape the ., *, ?, or () characters with a preceding backslash.
  For example, the character class [0-5.] will match digits 0 to 5 and a period.
  You do not need to write it as [0-5\.].
  By placing a caret character (^) just after the character class’s opening bracket, you can make a negative character class.
  A negative character class will match all the characters that are not in the character class.
  For example:

In [138]:
```
consonantRegex = re.compile(r'[^aeiouAEIOU]')
consonantRegex.findall('RoboCop eats baby food. BABY FOOD.')
```

Out[138]:
```
['R',
 'b',
 'C',
 'p',
 ' ',
 't',
 's',
 ' ',
 'b',
 'b',
 'y',
 ' ',
 'f',
 'd',
 '.',
 ' ',
 'B',
 'B',
 'Y',
 ' ',
 'F',
 'D',
 '.']
```

15. Caret and Dollar Sign Characters $ ^

(go to top)

You can also use the caret symbol (^) at the start of a regex to indicate that a match must occur at the beginning of the searched text.
Likewise, you can put a dollar sign ($) at the end of the regex to indicate the string must end with this regex pattern.
And you can use the ^ and $ together to indicate that the entire string must match the regex—that is, it's not enough for a match to be made on some subset of the string.
For example, the r'^Hello' regular expression string matches strings that begin with 'Hello'.

```
In [4]: beginsWithHello = re.compile(r'^Hello')
```

```
In [6]: beginsWithHello.search('Hello World')
```

```
Out[6]: <re.Match object; span=(0, 5), match='Hello'>
```

```
In [13]: mo = beginsWithHello.search('Hello World')
```

```
In [14]: mo
```

```
Out[14]: <re.Match object; span=(0, 5), match='Hello'>
```

```
In [16]: endsWithNumber = re.compile(r'\d$')
```

```
In [18]: endsWithNumber.search('Your number is 42')
```

```
Out[18]: <re.Match object; span=(16, 17), match='2'>
```

```
In [21]: endsWithNumber.search('Your number is 42 is') == None
```

```
Out[21]: True
```

The r'^\d+$' regular expression string matches strings that both begin and end with one or more numeric characters

```
In [39]: strBeginsEndsWithNum = re.compile(r'^\d+$')
```

```
In [40]: strBeginsEndsWithNum.search('123456789')
```

```
Out[40]: <re.Match object; span=(0, 9), match='123456789'>
```

---

16. Wildcard Character .

(go to top)

The . (or dot) character in a regular expression is called a wildcard and will match any character except for a newline. For example, enter the following into the interactive shell:

```
In [18]: atRegex = re.compile(r'.at')
```

```
In [21]: mo = atRegex.findall('The cat in the hat sat on the flat mat ate attac
```

```
In [22]: mo
```

```
Out[22]: ['cat', 'hat', 'sat', 'lat', 'mat', ' at', ' at', 'mat']
```


Matching Everything with Dot-Star

Sometimes you will want to match everything and anything.
For example, say you want to match the string 'First Name:', followed by any and all text, followed by 'Last Name:', and then followed by anything again.
You can use the dot-star (.*) to stand in for that “anything.”
Remember that the dot character means “any single character except the newline,” and the star character means “zero or more of the preceding character.”

```
In [52]: nameRegex = re.compile(r'First Name: (.*) Last Name: (.*)')
```

```
In [53]: mo = nameRegex.search('First Name: Al Last Name: Sweigart')
```

```
In [54]: mo.group()
Out[54]: 'First Name: Al Last Name: Sweigart'
```

```
In [56]: mo.group(1)
Out[56]: 'Al'
```

```
In [58]: mo.group(2)
Out[58]: 'Sweigart'
```

```
---------------------------------------------------------------------
------------------------------------------
```

```
The dot-star uses greedy mode: It will always try to match as much
text as possible.
To match any and all text in a non-greedy fashion, use the dot,
star, and question mark (.*?).
Like with braces, the question mark tells Python to match in a non-
greedy way.
```

```
In [61]: nongreedyRegex = re.compile(r'<.*?>')
         mo = nongreedyRegex.search('<To serve man> for dinner.>')
```

```
In [64]: mo.group()
Out[64]: '<To serve man>'
```

```
In [67]: greedyRegex = re.compile(r'<.*>')
         mo = greedyRegex.search('<To serve man> for dinner.>')
         mo.group()
Out[67]: '<To serve man> for dinner.>'
```

```
---------------------------------------------------------------------
------------------------------------------
```

Matching Newlines with the Dot Character

The dot-star will match everything except a newline.
By passing re.DOTALL as the second argument to re.compile(), you can make the dot character match all characters, including the newline character.

In [77]:
```
noNewLineRegex = re.compile('.*')
```

In [78]:
```
mo = noNewLineRegex.search('Serve the public trust.\nProtect the innoc
```

In [79]:
```
mo.group()
```

Out[79]:
```
'Serve the public trust.'
```

In [80]:
```
NewLineRegex = re.compile('.*', re.DOTALL)
```

In [81]:
```
mo1 = NewLineRegex.search('Serve the public trust.\nProtect the innoce
```

In [82]:
```
mo1.group()
```

Out[82]:
```
'Serve the public trust.\nProtect the innocent. \nUphold the law.'
```

---

17. Recap

(go to top)

- The ? matches zero or one of the preceding group.
- The * matches zero or more of the preceding group.
- The + matches one or more of the preceding group.
- The {n} matches exactly n of the preceding group.
- The {n,} matches n or more of the preceding group.
- The {,m} matches 0 to m of the preceding group.
- The {n,m} matches at least n and at most m of the preceding group.
- {n,m}? or *? or +? performs a non-greedy match of the preceding group.
- ^spam means the string must begin with spam.
- spam$ means the string must end with spam.
- The . matches any character, except newline characters.
- \d, \w, and \s match a digit, word, or space character, respectively.
- \D, \W, and \S match anything except a digit, word, or space character, respectively.
- [abc] matches any character between the brackets (such as a, b, or c).
- [^abc] matches any character that isn't between the brackets.

---

18. Case-Insensitive Matching

sometimes you care only about matching the letters without worrying whether they're uppercase or lowercase. To make your regex case- insensitive, you can pass re.IGNORECASE or re.I as a second argument to re.compile().

(go to top)

In [84]:
```
robocop = re.compile(r'robocop', re.I)
```

In [85]:
```
robocop.search('Robocop is part man, part machine, all cop.').group()
```

Out[85]:
```
'Robocop'
```

In [86]:
```
robocop.search('ROBOCOP protects the innocent.').group()
```
Out[86]:
```
'ROBOCOP'
```

In [87]:
```
robocop.search('Al, why does your programming book talk about robocop
```
Out[87]:
```
'robocop'
```

19. Substitute Strings withthe `sub()` method

- Regular expressions can not only find text patterns but can also substitute new text in place of those patterns. The sub() method for Regex objects is passed two arguments..

(go to top)

In [13]:
```
namesRegex = re.compile(r'Agent \w+') # match  'Agent space (any word
```
In [14]:
```
mo = namesRegex.sub('CENSORED', 'Agent Alice gave the secret documents
```
In [15]:
```
mo
```
Out[15]:
```
'CENSORED gave the secret documents to CENSORED.'
```

Sometimes you may need to use the matched text itself as part of the substitution.
In the first argument to sub(), you can type \1, \2, \3, and so on, to mean "Enter the text of group 1, 2, 3, and so on, in the substitution."
For example, say you want to censor the names of the secret agents by showing just the first letters of their names.
To do this, you could use the regex Agent (\w)\w* and pass r'\1****' as the first argument to sub().
The \1 in that string will be replaced by whatever text was matched by group 1— that is, the (\w) group of the regular expression.

```
In [38]: agentNamesRegex = re.compile(r'Agent (\w)\w*') # match Agent then spac
```

```
In [39]: mo = agentNamesRegex.sub(r'\1****', 'Agent Alice told Agent Carol that
```

```
In [40]: mo
```

```
Out[40]: 'A**** told C**** that E**** knew B**** was a double agent.'
```

```
In [38]: agentNamesRegex = re.compile(r'Agent (\w)\w*')
```

```
In [55]: mo = agentNamesRegex.sub(r'\0', 'Agent Alice told Agent Carol that Age
```

```
In [56]: mo
```

```
Out[56]: '\x00 told \x00 that \x00 knew \x00 was a double agent.'
```

```
In [58]: mo1 = agentNamesRegex.search('Agent Alice told Agent Carol that Agent
```

```
In [59]: mo1
```

```
Out[59]: <re.Match object; span=(0, 11), match='Agent Alice'>
```

20. Managing Complex Regexes

- Regular expressions are fine if the text pattern you need to match is simple.
- But matching complicated text patterns might require long, convoluted reg- ular expressions.
- You can mitigate this by telling the re.compile() function to ignore whitespace and comments inside the regular expression string.
- This “verbose mode” can be enabled by passing the variable re.VERBOSE as the second argument to re.compile().

(go to top)

In [60]:
```
phoneRegex = re.compile(r'((\d{3}|\(\d{3}\))?(\s|-|\.)?\d{3}(\s|-|\.)\
```

In [66]:
```
# same as

phoneRegex = re.compile(r'''(
                            (\d{3}|\(\d{3}\))?           # area code
                            (\s|-|\.)?                   # seperator
                            \d{3}                        # first 3 di
                            (\s|-|\.)?
                            \d{4}                        # last 4 dig
                            (\s*(ext|x|ext.)\s*\d{2,5})?  # extension
                            )''', re.VERBOSE)
```

21. re.IGNORE, re.DOTALL, re.VERBORSE Combined

- What if you want to use re.VERBOSE to write comments in your regular expression but also want to use re.IGNORECASE to ignore capitalization?
- Unfortunately, the re.compile() function takes only a single value as its second argument.
- You can get around this limitation by combining the re.IGNORECASE, re.DOTALL, and re.VERBOSE variables using the pipe character (|), which in this context is known as the bitwise or operator.

(go to top)

In [67]:
```
someRegexValue = re.compile('foo', re.IGNORECASE | re.DOTALL)
```

In [68]:
```
someRegexValue = re.compile('foo', re.IGNORECASE | re.DOTALL | re.VERE
```

21. Phone Number & Email Extractor

(go to top)

In [ ]:

In [ ]:

In [ ]:

20. Title

(go to top)

In [ ]:

In [ ]:

```
Support for regular expressions (RE).

This module provides regular expression matching operations si
milar to
those found in Perl.  It supports both 8-bit and Unicode strin
gs; both
the pattern and the strings being processed can contain null b
ytes and
characters outside the US ASCII range.

Regular expressions can contain both special and ordinary char
acters.
```

```
Most ordinary characters, like "A", "a", or "0", are the simpl
est
regular expressions; they simply match themselves.  You can
concatenate ordinary characters, so last matches the string 'l
ast'.

The special characters are:
    "."      Matches any character except a newline.
    "^"      Matches the start of the string.
    "$"      Matches the end of the string or just before the
newline at
             the end of the string.
    "*"      Matches 0 or more (greedy) repetitions of the pre
ceding RE.
             Greedy means that it will match as many repetitio
ns as possible.
    "+"      Matches 1 or more (greedy) repetitions of the pre
ceding RE.
    "?"      Matches 0 or 1 (greedy) of the preceding RE.
    *?,+?,?? Non-greedy versions of the previous three special
characters.
    {m,n}    Matches from m to n repetitions of the preceding
RE.
    {m,n}?   Non-greedy version of the above.
    "\\"     Either escapes special characters or signals a sp
ecial sequence.
    []       Indicates a set of characters.
             A "^" as the first character indicates a compleme
nting set.
    "|"      A|B, creates an RE that will match either A or B.
    (...)    Matches the RE inside the parentheses.
             The contents can be retrieved or matched later in
the string.
    (?aiLmsux) The letters set the corresponding flags defined
below.
    (?:...)  Non-grouping version of regular parentheses.
    (?P<name>...) The substring matched by the group is access
ible by name.
    (?P=name)     Matches the text matched earlier by the grou
p named name.
    (?#...)  A comment; ignored.
    (?=...)  Matches if ... matches next, but doesn't consume
```

```
the string.
    (?!...)  Matches if ... doesn't match next.
    (?<=...) Matches if preceded by ... (must be fixed length)
.
    (?<!...) Matches if not preceded by ... (must be fixed len
gth).
    (?(id/name)yes|no) Matches yes pattern if the group with i
d/name matched,
                       the (optional) no pattern otherwise.

The special sequences consist of "\\" and a character from the
list
below.  If the ordinary character is not on the list, then the
resulting RE will match the second character.
    \number  Matches the contents of the group of the same num
ber.
    \A       Matches only at the start of the string.
    \Z       Matches only at the end of the string.
    \b       Matches the empty string, but only at the start o
r end of a word.
    \B       Matches the empty string, but not at the start or
end of a word.
    \d       Matches any decimal digit; equivalent to the set
[0-9] in
             bytes patterns or string patterns with the ASCII
flag.
             In string patterns without the ASCII flag, it wil
l match the whole
             range of Unicode digits.
    \D       Matches any non-digit character; equivalent to [^
\d].
    \s       Matches any whitespace character; equivalent to [
\t\n\r\f\v] in
             bytes patterns or string patterns with the ASCII
flag.
             In string patterns without the ASCII flag, it wil
l match the whole
             range of Unicode whitespace characters.
    \S       Matches any non-whitespace character; equivalent
to [^\s].
    \w       Matches any alphanumeric character; equivalent to
[a-zA-Z0-9_]
```

```
             in bytes patterns or string patterns with the ASC
II flag.
             In string patterns without the ASCII flag, it wil
l match the
             range of Unicode alphanumeric characters (letters
plus digits
             plus underscore).
             With LOCALE, it will match the set [0-9_] plus ch
aracters defined
             as letters for the current locale.
    \W       Matches the complement of \w.
    \\       Matches a literal backslash.

This module exports the following functions:
    match     Match a regular expression pattern to the beginn
ing of a string.
    fullmatch Match a regular expression pattern to all of a s
tring.
    search    Search a string for the presence of a pattern.
    sub       Substitute occurrences of a pattern found in a s
tring.
    subn      Same as sub, but also return the number of subst
itutions made.
    split     Split a string by the occurrences of a pattern.
    findall   Find all occurrences of a pattern in a string.
    finditer  Return an iterator yielding a Match object for e
ach match.
    compile   Compile a pattern into a Pattern object.
    purge     Clear the regular expression cache.
    escape    Backslash all non-alphanumerics in a string.

Each function other than purge and escape can take an optional
'flags' argument
consisting of one or more of the following module constants, j
oined by "|".
A, L, and U are mutually exclusive.
    A  ASCII       For string patterns, make \w, \W, \b, \B, \
d, \D
                   match the corresponding ASCII character cat
egories
                   (rather than the whole Unicode categories,
which is the
```

```
                   default).
                   For bytes patterns, this flag is the only a
vailable
                   behaviour and needn't be specified.
    I  IGNORECASE  Perform case-insensitive matching.
    L  LOCALE      Make \w, \W, \b, \B, dependent on the curre
nt locale.
    M  MULTILINE   "^" matches the beginning of lines (after a
newline)
                   as well as the string.
                   "$" matches the end of lines (before a newl
ine) as well
                   as the end of the string.
    S  DOTALL      "." matches any character at all, including
the newline.
    X  VERBOSE     Ignore whitespace and comments for nicer lo
oking RE's.
    U  UNICODE     For compatibility only. Ignored for string
patterns (it
                   is the default), and forbidden for bytes pa
tterns.
```