Regular Expression

Regular Expression

WHAT A special text string for describing a search pattern

For searching, replacing, and parsing text with complex patterns of characters

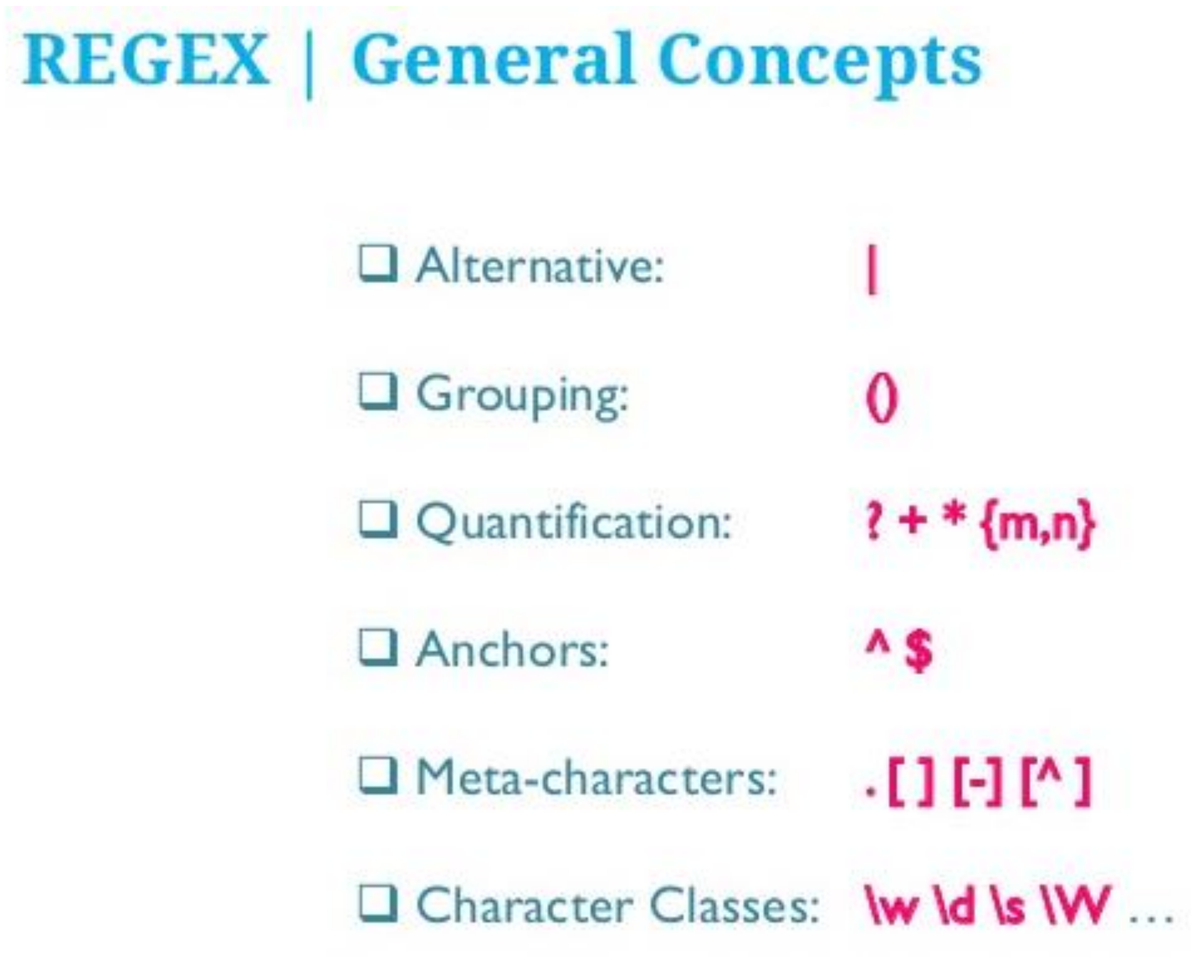
WHY Processes large amounts of text over and over again / Extremely fast

Usually this pattern is then used by ***string searching*** algorithms

For find or find and replace operations on strings, or for ***input validation***

BUT, There is a ***learning curve***

General Concepts

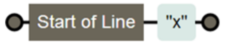
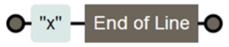
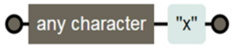
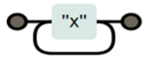
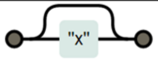
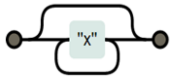
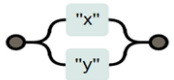
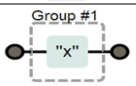
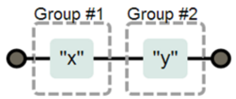
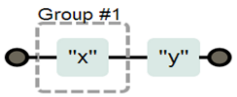
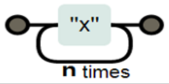
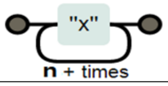
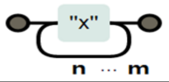


REGEX | General Concepts

❑ Alternative:	|
❑ Grouping:	()
❑ Quantification:	? + * {m,n}
❑ Anchors:	^ $
❑ Meta-characters:	. [ ] [-] [^]
❑ Character Classes:	\w \d \s \W …

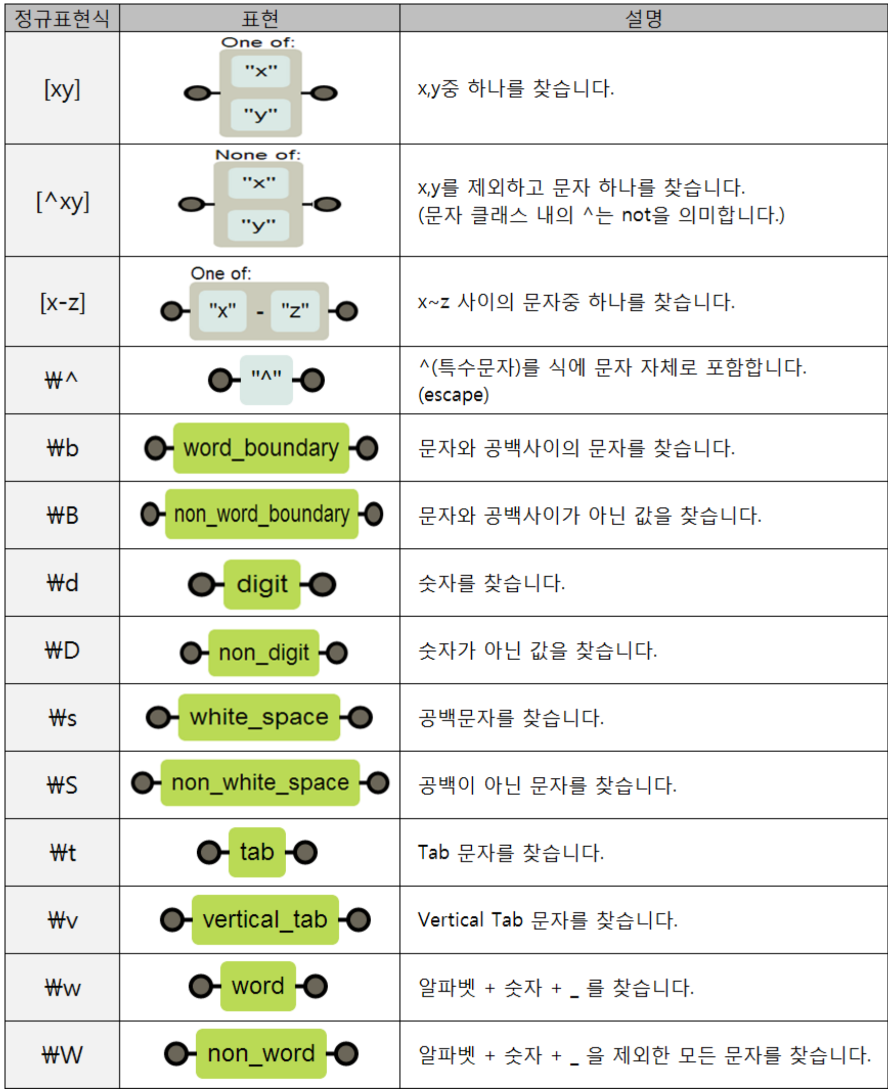
Meta Characters

.	Any character
*	Zero, one or more
+	One or more
?	Zero or one
{}	Specified number of occurrences

정규표현식	표현	설명
^x		문자열이 x로 시작합니다.
x$		문자열이 x로 끝납니다.
.x		임의의 한 문자를 표현합니다. (x가 마지막으로 끝납니다.)
x+		x가 1번이상 반복합니다.
x?		x가 존재하거나 존재하지 않습니다.
x*		x가 0번이상 반복합니다.
x|y		x 또는 y를 찾습니다. (or연산자를 의미합니다.)
(x)		( )안의 내용을 캡쳐하며, 그룹화 합니다.
(x)(y)		그룹화 할 때, 자동으로 앞에서부터 1번부터 그룹 번호를 부여해서 캡쳐합니다. 결과값에 그룹화한 Data가 배열 형식으로 그룹번호 순서대로 들어갑니다.
(x)(?:y)		캡쳐하지 않는 그룹을 생성할 경우 ?:를 사용합니다. 결과값 배열에 캡쳐하지 않는 그룹은 들어가지 않습니다.
x{n}		x를 n번 반복한 문자를 찾습니다.
x{n,}		x를 n번이상 반복한 문자를 찾습니다.
x{n,m}		x를 n번이상 m번이하 반복한 문자를 찾습니다.

Meta Characters

[]	Check any single character
[-]	Range of characters
[^]	Negation
^	Beginning of string
$	End of string
()	Grouping

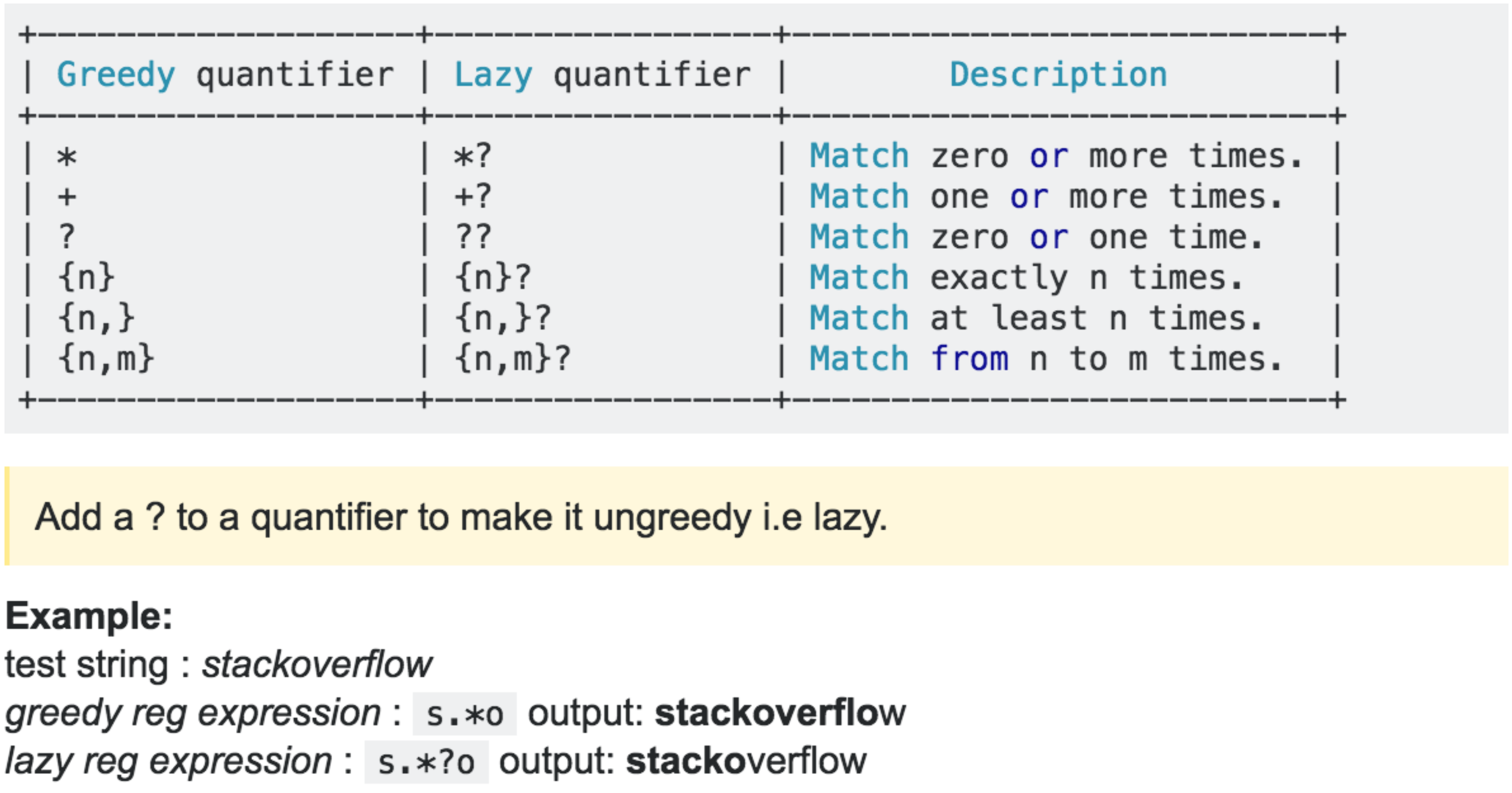


정규표현식	표현	설명
[xy]	One of: "x" "y"	x,y중 하나를 찾습니다.
[^xy]	None of: "x" "y"	x,y를 제외하고 문자 하나를 찾습니다. (문자 클래스 내의 ^는 not을 의미합니다.)
[x-z]	One of: "x" - "z"	x~z 사이의 문자중 하나를 찾습니다.
₩^	"^"	^(특수문자)를 식에 문자 자체로 포함합니다. (escape)
₩b	word_boundary	문자와 공백사이의 문자를 찾습니다.
₩B	non_word_boundary	문자와 공백사이가 아닌 값을 찾습니다.
₩d	digit	숫자를 찾습니다.
₩D	non_digit	숫자가 아닌 값을 찾습니다.
₩s	white_space	공백문자를 찾습니다.
₩S	non_white_space	공백이 아닌 문자를 찾습니다.
₩t	tab	Tab 문자를 찾습니다.
₩v	vertical_tab	Vertical Tab 문자를 찾습니다.
₩w	word	알파벳 + 숫자 + _ 를 찾습니다.
₩W	non_word	알파벳 + 숫자 + _ 을 제외한 모든 문자를 찾습니다.

Greedy vs. Lazy

Greedy Tries to find the last possible match

Lazy Tries to find the first possible match

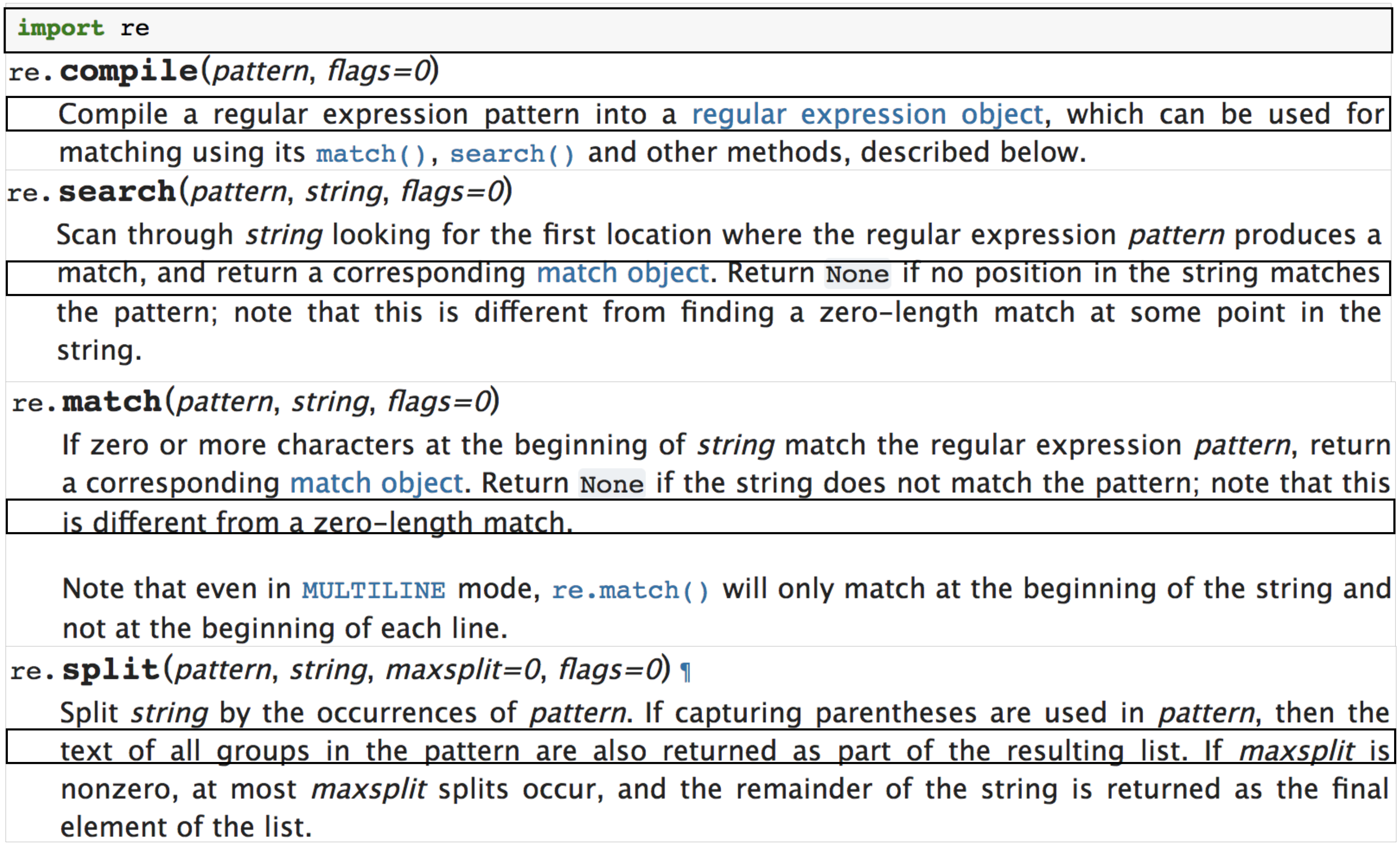


```
+-------------------+-----------------+------------------------------+
| Greedy quantifier | Lazy quantifier |        Description           |
+-------------------+-----------------+------------------------------+
| *                 | *?              | Match zero or more times.    |
| +                 | +?              | Match one or more times.     |
| ?                 | ??              | Match zero or one time.      |
| {n}               | {n}?            | Match exactly n times.       |
| {n,}              | {n,}?           | Match at least n times.      |
| {n,m}             | {n,m}?          | Match from n to m times.     |
+-------------------+-----------------+------------------------------+
```

Add a ? to a quantifier to make it ungreedy i.e lazy.

Example:
test string : *stackoverflow*
greedy reg expression : `s.*o` output: **stackoverflow**
lazy reg expression : `s.*?o` output: **stacko**verflow

re



```python
import re
```

re.**compile**(*pattern*, *flags=0*)

Compile a regular expression pattern into a regular expression object, which can be used for matching using its `match()`, `search()` and other methods, described below.

re.**search**(*pattern*, *string*, *flags=0*)

Scan through *string* looking for the first location where the regular expression *pattern* produces a match, and return a corresponding match object. Return `None` if no position in the string matches the pattern; note that this is different from finding a zero-length match at some point in the string.

re.**match**(*pattern*, *string*, *flags=0*)

If zero or more characters at the beginning of *string* match the regular expression *pattern*, return a corresponding match object. Return `None` if the string does not match the pattern; note that this is different from a zero-length match.

Note that even in `MULTILINE` mode, `re.match()` will only match at the beginning of the string and not at the beginning of each line.

re.**split**(*pattern*, *string*, *maxsplit=0*, *flags=0*) ¶

Split *string* by the occurrences of *pattern*. If capturing parentheses are used in *pattern*, then the text of all groups in the pattern are also returned as part of the resulting list. If *maxsplit* is nonzero, at most *maxsplit* splits occur, and the remainder of the string is returned as the final element of the list.

```python
data = """
park 800905-1049118
kim  700905-1059119
"""

result = []
for line in data.split("\n"):
    word_result = []
    for word in line.split(" "):
        if len(word) == 14 and word[:6].isdigit() and word[7:].isdigit():
            word = word[:6] + "-" + "*******"
        word_result.append(word)
    result.append(" ".join(word_result))
print("\n".join(result))
```

```python
data = """
park 800905-1049118
kim  700905-1059119
"""

pat = re.compile("(\d{6})[-]\d{7}")
print(pat.sub("\g<1>-*******", data))
```

```
p = re.compile('Crow|Servo')
m = p.match('CrowHello')
print(m)
```

```
print(re.search('^Life', 'Life is too short'))
print(re.search('^Life', 'My Life'))
```

```
print(re.search('short$', 'Life is too short'))
print(re.search('short$', 'Life is too short, you need python'))
```

```
p = re.compile('(ABC)+')
m = p.search('ABCABCABC OK?')
print(m.group())
```

```
p = re.compile(r'\bclass\b')
print(p.search('no class at all'))
print(p.search('one subclass is'))
print(p.search('the declassified algorithm'))
```

```
p = re.compile(r'\Bclass\B')
print(p.search('no class at all'))
print(p.search('one subclass is'))
print(p.search('the declassified algorithm'))
```

```
p = re.compile(r"\w+\s+\d+[-]\d+[-]\d+")
m = p.search("park 010-1234-1234")
print(m)

p = re.compile(r"(\w+)\s+\d+[-]\d+[-]\d+")
m = p.search("park 010-1234-1234")
print(m.group(1))
```

`Match.`**group**([*group1*, ...])

Returns one or more subgroups of the match. If there is a single argument, the result is a single string; if there are multiple arguments, the result is a tuple with one item per argument. Without arguments, *group1* defaults to zero (the whole match is returned). If a *groupN* argument is zero, the corresponding return value is the entire matching string; if it is in the inclusive range [1..99], it is the string matching the corresponding parenthesized group. If a group number is negative or larger than the number of groups defined in the pattern, an `IndexError` exception is raised. If a group is contained in a part of the pattern that did not match, the corresponding result is `None`. If a group is contained in a part of the pattern that matched multiple times, the last match is returned.

method	목적
group()	매치된 문자열을 리턴한다.
start()	매치된 문자열의 시작 위치를 리턴한다.
end()	매치된 문자열의 끝 위치를 리턴한다.
span()	매치된 문자열의 (시작, 끝) 에 해당되는 튜플을 리턴한다.

```python
m = re.match(r"(\w+) (\w+)", "Isaac Newton, physicist")
print(m.group(0))
print(m.group(1))
print(m.group(2))
print(m.group(1, 2))
```

```python
p = re.compile(r"(\w+) (\w+)")
m = p.search("Isaac Newton, physicist")
print(m.group())

p.sub("\g<2> \g<1>", "Isaac Newton, physicist")
```

regularexpressions

Anchors

^	Start of line +
\A	Start of string +
$	End of line +
\Z	End of string +
\b	Word boundary +
\B	Not word boundary +
\<	Start of word
\>	End of word

Character Classes

\c	Control character
\s	White space
\S	Not white space
\d	Digit
\D	Not digit
\w	Word
\W	Not word
\xhh	Hexadecimal character hh
\Oxxx	Octal character xxx

POSIX Character Classes

[:upper:]	Upper case letters
[:lower:]	Lower case letters
[:alpha:]	All letters
[:alnum:]	Digits and letters
[:digit:]	Digits
[:xdigit:]	Hexadecimal digits
[:punct:]	Punctuation
[:blank:]	Space and tab
[:space:]	Blank characters
[:cntrl:]	Control characters
[:graph:]	Printed characters
[:print:]	Printed characters and spaces
[:word:]	Digits, letters and underscore

Assertions

?=	Lookahead assertion +
?!	Negative lookahead +
?<=	Lookbehind assertion +
?!= or ?<!	Negative lookbehind +
?>	Once-only Subexpression
?()	Condition [if then]
?()|	Condition [if then else]
?#	Comment

Note Items marked + should work in most regular expression implementations.

Sample Patterns

([A-Za-z0-9-]+)	Letters, numbers and hyphens
(\d{1,2}\/\d{1,2}\/\d{4})	Date (e.g. 21/3/2006)
([^\s]+(?=\.(jpg|gif|png))\.\2)	jpg, gif or png image
(^[1-9]{1}$\|^[1-4]{1}[0-9]{1}$|^50$)	Any number from 1 to 50 inclusive
(#?([A-Fa-f0-9]){3}(([A-Fa-f0-9]){3})?)	Valid hexadecimal colour code
((?=.*\d)(?=.*[a-z])(?=.*[A-Z]).{8,15})	8 to 15 character string with at least one upper case letter, one lower case letter, and one digit (useful for passwords).
(\w+@[a-zA-Z_]+?\.[a-zA-Z]{2,6})	Email addresses
(\<(/?[^\>]+)\>)	HTML Tags

Note *These patterns are intended for reference purposes and have not been extensively tested. Please use with caution and test thoroughly before use.*

Quantifiers

*	0 or more +
*?	0 or more, ungreedy +
+	1 or more +
+?	1 or more, ungreedy +
?	0 or 1 +
??	0 or 1, ungreedy +
{3}	Exactly 3 +
{3,}	3 or more +
{3,5}	3, 4 or 5 +
{3,5}?	3, 4 or 5, ungreedy +

Special Characters

\	Escape Character +
\n	New line +
\r	Carriage return +
\t	Tab +
\v	Vertical tab +
\f	Form feed +
\a	Alarm
[\b]	Backspace
\e	Escape
\N{name}	Named Character

String Replacement (Backreferences)

$n	nth non-passive group
$2	"xyz" in /^(abc(xyz))$/
$1	"xyz" in /^(?:abc)(xyz)$/
$`	Before matched string
$'	After matched string
$+	Last matched string
$&	Entire matched string
$_	Entire input string
$$	Literal "$"

Ranges

.	Any character except new line (\n) +
(a|b)	a or b +
(...)	Group +
(?:...)	Passive Group +
[abc]	Range (a or b or c) +
[^abc]	Not a or b or c +
[a-q]	Letter between a and q +
[A-Q]	Upper case letter + between A and Q +
[0-7]	Digit between 0 and 7 +
\n	nth group/subpattern +

Note *Ranges are inclusive.*

Pattern Modifiers

g	Global match
i	Case-insensitive
m	Multiple lines
s	Treat string as single line
x	Allow comments and white space in pattern
e	Evaluate replacement
U	Ungreedy pattern

Metacharacters (must be escaped)

^	[	.
$	{	*
(	\	+
)	|	?
<	>	