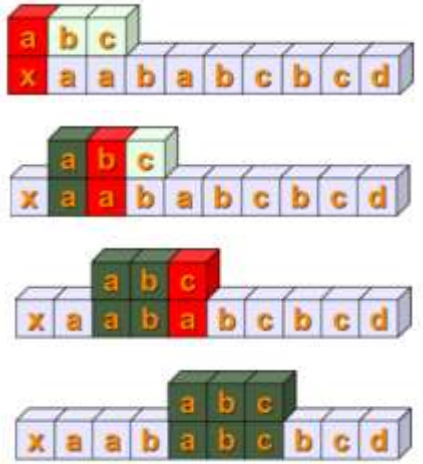
Regular Expressions

Match vs. Search

```
'abc' in 'xaababcbcd'
```



`search()` scans through the string to see if any substring matches

```
>>> import re
>>> s = re.search('cat',"A cat and a rat can't be friends.")
>>> print s
<_sre.SRE_Match object at 0x000000000306AE68>
>>> s1 = re.search('cow',"A cat and a rat can't be friends.")
>>> print s1
None
```

`match()` determines if pattern matches at the beginning of a string

```
>>> m = re.match('.*cat.*',"A cat and a rat can't be friends.")
>>> print m
<_sre.SRE_Match object at 0x000000000306AE00>
```

`match()`/`search()` returns
matched object if matched
`None`, otherwise

String Pattern Matching

- `|` means "or": `(aa|bb)` matches `aa` or `bb`
- `[abc]` matches `a`, `b`, or `c`
  - or simply `[a-c]`
  - equivalent to `(a|b|c)`
  - `[abc]+` matches `a`, `b`, `c`, `aa`, `ab`, `ac`, `ba`, `bb`, `bc`, `ca`, `cb`, `cc`, ...
- `[^5]` matches any char except `5`
  - `[^0-9]` matches any char except a digit
- `a[bcd]*` matches many more than `a(bcd)*`

Regex Pattern	Strings Matched
`at|home`	`at`, `home`
`r2d2|c3po`	`r2d2`, `c3po`
`bat|bet|bit`	`bat`, `bet`, `bit`

Regex Pattern
`z.[0-9]`
`[r-u][env-y][us]`
`[^aeiou]`
`[^\t\n]`
`["-a]`

String Pattern Matching

- *: repeating 0 or more times
  - ab*d matches ad, abd, abbd, ...
  - a(bcd)*d matches ad, abcdd, abcdbcdd, ...
- +: repeating 1 or more times
  - ab+d matches abd, abbd, ...
  - a(bcd)+d matches abcdd, abcdbcdd, ...
- ?: 0 or 1 times
- {n}: n times
- {m, n}: from m to n times

Regex Pattern
[dn]ot?
0?[1-9]
[0-9]{15,16}
</?[^>]+>
[KQRBNP][a-h][1-8]-[a-h][1-8]

Special Characters

- characters with special meanings
  - `. ^ $ * + ? { } [ ] \ | ( )`
- `\(ab\)` matches `(ab)`
- `.` matches any single character
  - `.*` matches any string
- `\\` matches `\`
- `^` matches the beginning of a line or string
  - not the `^` inside char-classes `[^...]`
- `$` matches the end of a line or string
  - `a[bcd]*b$` does not match string `abcbd`

Q: What character class is described by
`[a-zA-Z_][a-zA-Z_0-9]*` ?
Q: regexpr to describe Korean post code?

Regex Pattern
^From
/bin/tcsh$
^Subject: hi$

Predefined Character Classes

`\d`	Matches any decimal digit; equivalent to the set `[0-9]`.
`\D`	The complement of `\d`. It matches any non-digit character; equivalent to the set `[^0-9]`.
`\s`	Matches any whitespace character; equivalent to `[ \t\n\r\f\v]`.
`\S`	The complement of `\s`. It matches any non-whitespace character; equiv. to `[^ \t\n\r\f\v]`.
`\w`	Matches any alphanumeric character; equivalent to `[a-zA-Z0-9_]`. With LOCALE, it will match the set `[a-zA-Z0-9_]` plus characters defined as letters for the current locale.
`\W`	Matches the complement of `\w`.
`\b`	Matches the empty string, but only at the start or end of a word.
`\B`	Matches the empty string, but not at the start or end of a word.

`[\s,.]` matches any white spaces, `","` , or `"."`
`\b` means word-boundary (zero-length):
`\b\w+\b` matches a single word (actually `\b\w+` is enough)

Regex Pattern	Strings Matched
`\w+-\d+`	Alphanumeric string and number separated by a hyphen
`[A-Za-z]\w*`	Alphabetic first character; additional characters (if present) can be alphanumeric (almost equivalent to the set of valid Python identifiers [see exercises])
`\d{3}-\d{3}-\d{4}`	American-format telephone numbers with an area code prefix, as in 800-555-1212
`\w+@\w+\.com`	Simple e-mail addresses of the form *XXX@YYY.com*

Regex Pattern	Strings Matched
`\d+(\.\d*)?`	Strings representing simple floating-point numbers; that is, any number of digits followed optionally by a single decimal point and zero or more numeric digits, as in "0.004," "2," "75.," etc.
`(Mr?s?\. )?[A-Z][a-z]* [A-Za-z-]+`	First name and last name, with a restricted first name (must start with uppercase; lowercase only for remaining letters, if any), the full name, prepended by an optional title of "Mr.," "Mrs.," "Ms.," or "M.," and a flexible last name, allowing for multiple words, dashes, and uppercase letters

Matching is Greedy

- by "matching" we mean matching the beginning portion of a string
  - `a(bc)+` matches the underlined part in abcbcd
- greedy search with backtracking
- `a(bcd)*b` matches abcdb, abcdbcd, abcd
- try match the pattern `a[bcd]*b` with string abcbd

1	a	The a in the RE matches.
2	abcbd	The engine matches [bcd]*, going as far as it can, which is to the end of the string.
3	*Failure*	The engine tries to match b, but the current position is at the end of the string, so it fails.
4	abcb	Back up, so that [bcd]* matches one less character.
5	*Failure*	Try b again, but the current position is at the last character, which is a "d".
6	abc	Back up again, so that [bcd]* is only matching "bc".
6	abcb	Try b again. This time but the character at the current position is "b", so it succeeds.

Performing Match and Search

```
>>> import re

>>> re.match('[a-z]+', "")
None

>>> p = re.compile('[a-z]+')
>>> p
<_sre.SRE_Pattern object at 80c3c28>

>>> p.match("")
>>> print p.match("")
None

>>> m = p.match( 'tempo')
>>> print m
<_sre.SRE_Match object at 80c4f68>
```

```
>>> print p.match('::: message')
None
>>> m = p.search('::: message')
>>> print m
<re.MatchObject instance at 80c9650>

>>> m.group()
'message'
>>> m.span()
(4, 11)
```

compiled version is faster for repeated use

Grouping

```
>>> import re
>>> patt = '\w+@(\w+\.)?\w+\.com'
>>> re.match(patt, 'nobody@www.xxx.com').group()
'nobody@www.xxx.com'
>>>
>>> group_patt = '(\w+)@(\w+\.)?\w+\.com'
>>> p = re.compile(group_patt)
>>> m = p.match('nobody@www.xxx.com')
>>> m.group()   # entine match
'nobody@www.xxx.com'
>>> m.group(0)  # same as group()
'nobody@www.xxx.com'
>>> m.group(1)  # subgroup 1
'nobody'
>>> m.group(2)  # subgroup 2
'www.'
>>> m.groups()  # all subgroups
('nobody', 'www.')
```

Common Match Object Methods (see documentation for others)

`group(num=0)`	Return entire match (or specific subgroup *num*)
`groups (default=None)`	Return all matching subgroups in a tuple (empty if there aren't any)
`groupdict (default=None)`	Return dict containing all matching named subgroups with the names as the keys (empty if there weren't any)

findall() vs. finditer()

- `findall()` returns a list of all substrings that matches
- `finditer()` returns an iterator of matched objects

```
>>> p = re.compile('\d+')
>>> s = '12 drummers, 11 pipers, 10 lords'
>>> p.findall()
['12', '11', '10']

>>> iterator = p.finditer(s)
>>> iterator
<callable-iterator object at 0x401
>>> for match in iterator:
...     print match.span()
...
(0, 2)
(13, 15)
(24, 26)
```

```
>>> import re
>>> patt = re.compile(r'(\w+)\s+(\d+)')
>>> s = " I teach cs 399 and cis   500."
>>>
>>> m = patt.search(s)
>>> m.group()
'cs 399'
>>> m.groups()
('cs', '399')
>>>
>>> patt.findall(s)
[('cs', '399'), ('cis', '500')]
>>>
>>> for m in p.finditer(s):
...     print m.group(), m.groups()
...
cs 399 ('cs', '399')
cis   500 ('cis', '500')
```

Back-referencing Groups

- referring to previous groups by \1, \2, ...

```
>>> q = re.compile(r'(\b\w+)\s+\1')
>>> s = "this is the  the course"

>>> q.findall(s)
['the']

>>> q.search(s).group()
'the  the'
```

r means "raw"

typo detecter :)

```
>>> p = re.compile('(a(b)c)d')
>>> m = p.match('abcd')
>>> m.group(0)
'abcd'
>>> m.group(1)
'abc'
>>> m.group(2)
'b'
```

Non-greedy Quantifier

- default matching is greedy
  - use non-greedy quantifiers ?

```
>>> s = '<html><head><title>Title</title>'

>>> print re.match('<.*>', s).group()
<html><head><title>Title</title>

>>> print re.match('<.*?>', s).group()
<html>

>>> p = re.compile('<a href=(.*)>')
>>> p.match("<a href=\"index.html\">back</a>").group(1)
'"index.html">back</a>'

>>> p = re.compile('<a href=(.*?)>')
>>> p.match("<a href=\"index.html\">back</a>").group(1)
'"index.html"'
```

Common Regular Expression Attribute

re *Module Function Only*

`compile(pattern, flags=0)`	Compile regex *pattern* with any optional *flags* and return a regex object

re *Module Functions and Regex Object Methods*

`match(pattern, string, flags=0)`	Attempt to match *pattern* to *string* with optional *flags*; return match object on success, `None` on failure
`search(pattern, string, flags=0)`	Search for first occurrence of *pattern* within *string* with optional *flags*; return match object on success, `None` on failure
`findall(pattern, string[,flags])`[a]	Look for all (non-overlapping) occurrences of *pattern* in *string*; return a list of matches
`finditer(pattern, string[, flags])`[b]	Same as `findall()`, except returns an iterator instead of a list; for each match, the iterator returns a match object
`split(pattern, string, max=0)`[c]	Split *string* into a list according to regex *pattern* delimiter and return list of successful matches, splitting at most *max* times (split all occurrences is the default)
`sub(pattern, repl, string, count=0)`c	Replace all occurrences of the regex *pattern* in *string* with *repl*, substituting all occurrences unless *count* provided (see also `subn()`, which, in addition, returns the number of substitutions made)
`purge()`	Purge cache of implicitly compiled regex patterns

Common Module Attributes (flags for most regex functions)

`re.I, re.IGNORECASE`	Case-insensitive matching
`re.L, re.LOCALE`	Matches via `\w, \W, \b, \B, \s, \S` depends on locale
`re.M, re.MULTILINE`	Respectively causes `^` and `$` to match the beginning and end of each line in target string rather than strictly the beginning and end of the entire string itself
`re.S, re.DOTALL`	The `.` normally matches any single character except `\n`; this flag says `.` should match them, too
`re.X, re.VERBOSE`	All whitespace plus # (and all text after it on a single line) are ignored unless in a character class or backslash-escaped, allowing comments and improving readability