Formal definition of the Regular language

The collection of regular languages over an alphabet Σ is defined recursively as follows:

- The empty language $\emptyset$ is a regular language.
- For each $a \in \Sigma$ (a belongs to Σ), the singleton language $\{a\}$ is a regular language.
- If A is a regular language, A* (Kleene star) is a regular language. Due to this, the empty string language $\{\varepsilon\}$ is also regular.
- If A and B are regular languages, then $A \cup B$ (union) and $A \bullet B$ (concatenation) are regular languages.
- No other languages over Σ are regular.

Example of the Regular language

The following rules describe a formal language L over the alphabet $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, =\}$:

- Every nonempty string that does not contain "+" or "=" and does not start with "0" is in L.
- The string "0" is in L.
- A string containing "=" is in L if and only if there is exactly one "=", and it separates two valid strings of L.
- A string containing "+" but not "=" is in L if and only if every "+" in the string separates two valid strings of L.
- No string is in L other than those implied by the previous rules.

Under these rules, the string "23+4=555" is in L,
but the string "=234=+" is not.

Example of the Regular language

- $L = \Sigma^*$, the set of all words over Σ;
- $L = \{a\}^* = \{an\}$, where n ranges over the natural numbers and "an" means "a" repeated n times (this is the set of words consisting only of the symbol "a");
- the set of syntactically correct programs in a given programming language
- the set of maximal strings of alphanumeric ASCII characters on this line, i.e., the set {the, set, of, maximal, strings, alphanumeric, ASCII, characters, on, this, line, i, e}

Regular expressions describe regular languages in formal language theory.

Formal definition of the Regular expression

Regular expressions consist of constants, which denote sets of strings, and operator symbols, which denote operations over these sets. Given a finite alphabet Σ, the following constants are defined as regular expressions:

- (empty set) $\emptyset$ denoting the set $\emptyset$.
- (empty string) ε denoting the set containing only the "empty" string, which has no characters at all.
- (literal character) a in Σ denoting the set containing only the character a.

Formal definition of the Regular expression

Given regular expressions R and S, the following operations over them are defined to produce regular expressions:

- (concatenation) (RS) denotes the set of strings that can be obtained by concatenating a string accepted by R and a string accepted by S (in that order). For example, let R denote {"ab", "c"} and S denote {"d", "ef"}. Then, (RS) denotes {"abd", "abef", "cd", "cef"}.
- (alternation) (R|S) denotes the set union of sets described by R and S. For example, if R describes {"ab", "c"} and S describes {"ab", "d", "ef"}, expression (R|S) describes {"ab", "c", "d", "ef"}.

Formal definition of the Regular expression

Given regular expressions R and S, the following operations over them are defined to produce regular expressions:

- (Kleene star) (R*) denotes the smallest superset of the set described by R that contains ε and is closed under string concatenation. This is the set of all strings that can be made by concatenating any finite number (including zero) of strings from the set described by R. For example, if R denotes {"0", "1"}, (R*) denotes the set of all finite binary strings (including the empty string). If R denotes {"ab", "c"}, (R*) denotes {ε, "ab", "c", "abab", "abc", "cab", "cc", "abcab", ... }.

A regular expression (shortened as regex or regexp) is a sequence of characters that specifies a search pattern in text.

Usually, such patterns are used by string-searching algorithms for "find" or "find and replace" operations on strings, or for input validation.

Regular expressions in Python

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

Regular expressions in Python

.

(Dot.) In the default mode, this matches any character except a newline. If the DOTALL flag has been specified, this matches any character including a newline.

Regular expressions in Python

^

(Caret.) Matches the start of the string.

Regular expressions in Python

$

Matches the end of the string.

Regular expressions in Python

*

Causes the resulting RE to match 0 or more repetitions of the preceding RE, as many repetitions as are possible. ab* will match ‘a’, ‘ab’, or ‘a’ followed by any number of ‘b’s.

+

Causes the resulting RE to match 1 or more repetitions of the preceding RE. ab+ will match ‘a’ followed by any non-zero number of ‘b’s; it will not match just ‘a’.

?

Causes the resulting RE to match 0 or 1 repetitions of the preceding RE. ab? will match either ‘a’ or ‘ab’.

Regular expressions in Python

*?, +?, ??

The '*', '+', and '?' qualifiers are all greedy; they match as much text as possible. Adding ? after the qualifier makes it perform the match in non-greedy or minimal fashion; as few characters as possible will be matched.

Regular expressions in Python

{m}

Specifies that exactly m copies of the previous RE should be matched

Regular expressions in Python

{m,n} or {m,n}?

Causes the resulting RE to match from m to n repetitions of the preceding RE, attempting to match as many repetitions as possible (without ?).

Regular expressions in Python

[ ]

Used to indicate a set of characters. In a set:

- Characters can be listed individually, e.g. [amk] will match 'a', 'm', or 'k'.
- Ranges of characters can be indicated by giving two characters and separating them by a '-', for example [a-z], [0-5][0-9], [0-9A-Fa-f
- Special characters lose their special meaning inside sets. For example, [(+*)] will match any of the literal characters '(', '+', '*', or ‘)’
- Characters that are not within a range can be matched by complementing the set. If the first character of the set is '^', all the characters that are not in the set will be matched.

Regular expressions in Python: how to use

```
>>> import re
>>> p = re.compile('ab*')
>>> p
re.compile('ab*')
```

Regular expressions in Python: how to use

Method/Attribute	Purpose
match()	Determine if the RE matches at the beginning of the string.
search()	Scan through a string, looking for any location where this RE matches.
findall()	Find all substrings where the RE matches, and returns them as a list.
finditer()	Find all substrings where the RE matches, and returns them as an iterator.

Regular expressions in Python: how to use

```
>>> m = p.match('tempo')
>>> m
<re.Match object; span=(0, 5), match='tempo'>
```

Method/Attribute	Purpose
group()	Return the string matched by the RE
start()	Return the starting position of the match
end()	Return the ending position of the match
span()	Return a tuple containing the (start, end) positions of the match

Regular expressions in Python: how to use

```
>>> m = p.match('tempo')
>>> m
<re.Match object; span=(0, 5), match='tempo'>
```

Method/Attribute	Purpose
group()	Return the string matched by the RE
start()	Return the starting position of the match
end()	Return the ending position of the match
span()	Return a tuple containing the (start, end) positions of the match

Regular expressions in Python: how to use

```
>>> m = p.search('::: message'); print(m)
<re.Match object; span=(4, 11), match='message'>
>>> m.group()
'message'
>>> m.span()
(4, 11)
```

Regular expressions in Python: how to use

```
>>> p = re.compile(r'\d+')
>>> p.findall('12 drummers drumming, 11 pipers piping,
10 lords a-leaping')
['12', '11', '10']
```

Regular expressions in Python: how to use

```
>>> iterator = p.finditer('12 drummers drumming, 11 ... 10 ...')
>>> iterator
<callable_iterator object at 0x...>
>>> for match in iterator:
...     print(match.span())
...
(0, 2)
(22, 24)
(29, 31)
```