Object and Data validation using Regular Expression

SE306: Object Oriented Concept II

What is Regular Expression

- A regular expression, regex or regexp is a formal language in theoretical computer science and software engineering.
- It is a sequence of characters that define a search pattern.
- The concept arose in the 1950s when the American mathematician Stephen Cole Kleene formalized the description of a regular language.

Patterns

- The pattern is a single character or a metacharacter (with its special meaning), or a regular character (with its literal meaning) for matching standard textual syntax.
- For example, in the regex a. *a* is a literal character which matches just '**a**' and **.** is a meta character which matches every character except a newline. Therefore, this regex would match for example 'a ' or 'ax' or 'a0'.

Boolean "or"

- A vertical bar separates alternatives.
- For example, Selim|Saeed can match “Selim" or “Saeed".

Grouping

- Parentheses are used to define the scope and precedence of the operators (among other uses).
- For example, BSSE|MSSE and (B|M)SSE are equivalent patterns which both describe the set of "BSSE" or "MSSE".

Quantification

?	The question mark indicates zero or one occurrences of the preceding element. For example, colou?r matches both "color" and "colour".
*****	The asterisk indicates *zero or more* occurrences of the preceding element. For example, ab*c matches "ac", "abc", "abbc", "abbbc", and so on.
+	The plus sign indicates *one or more* occurrences of the preceding element. For example, ab+c matches "abc", "abbc", "abbbc", and so on, but not "ac"
{n}	The preceding item is matched exactly *n* times. Example: a{3} matches "aaa"
{min,}	The preceding item is matched *min* or more times. Example: a{3, } matches "aaa" or "aaaa" or more
{min, max}	The preceding item is matched at least *min* times, but not more than *max* times.

RE in Java

The java.util.regex package primarily consists of the following three classes:

- **Pattern Class:** To create a pattern, you must first invoke one of its public static compile() methods, which will then return a Pattern object.
- **Matcher Class:** A Matcher object is the engine that interprets the pattern and performs match operations against an input string.
- **PatternSyntaxException:** A PatternSyntaxException object is an unchecked exception that indicates a syntax error in a regular expression pattern.

Basic Java RE Code

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

{
   String line = "Welcome BSSE11 Batch";
   String pattern = "S{2,8}";

    // Create a Pattern object
   Pattern r = Pattern.compile(pattern);

   // Now create matcher object.
   Matcher m = r.matcher(line);

   System.out.println( m.find() ? "Found value: " + line : "NO MATCH“ );
}
```

Regex Code

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

class Main {
  public static void main(String[] args) {
  String line = "Welcome BSSE11 Batch";
  String pattern = "S{2,8}";

  // Create a Pattern object
  Pattern r = Pattern.compile(pattern);

  // Now create matcher object.
  Matcher m = r.matcher(line);

  System.out.println( m.find() ? "Found value: " + line : "NO MATCH" );
  }
}
```

Doing it in Java, I

- First, you must *compile* the pattern

```
import java.util.regex.*;
Pattern p = Pattern.compile("[a-z]+");
```

- Next, you must create a *matcher* for a specific piece of text by sending a message to your pattern

```
Matcher m = p.matcher("Now is the time");
```

- Points to notice:
  - Pattern and Matcher are both in java.util.regex
  - Neither Pattern nor Matcher has a public constructor; you create these by using methods in the Pattern class
  - The matcher contains information about *both* the pattern to use *and* the text to which it will be applied

Doing it in Java, II

- Now that we have a matcher m,
  - m.matches() returns true if the pattern matches the entire text string, and false otherwise
  - m.lookingAt() returns true if the pattern matches at the beginning of the text string, and false otherwise
  - m.find() returns true if the pattern matches any part of the text string, and false otherwise
    - If called again, m.find() will start searching from where the last match was found
    - m.find() will return true for as many matches as there are in the string; after that, it will return false
    - When m.find() returns false, matcher m will be *reset* to the beginning of the text string (and may be used again)

Finding what was matched

- *After a successful match,* m.start() will return the index of the first character matched
- *After a successful match*, m.end() will return the index of the last character matched, *plus one*
- If no match was attempted, or if the match was unsuccessful, m.start() and m.end() will throw an IllegalStateException
  - This is a RuntimeException, so you don't have to catch it

RE Syntax

.	Matches any single character (many applications exclude newlines,
[ ]	Matches a single character that is contained within the brackets. For example, [abc] matches "a", "b", or "c". [a-z] specifies a range which matches any lowercase letter from "a" to "z".
[^]	Matches a single character that is not contained within the brackets. For example, [^a-z]matches any single character that is not a lowercase letter from "a" to "z".
$	Matches the ending position of the string.
( )	A marked subexpression is also called a block or capturing group.
n	Matches what the *n*th marked subexpression matched, where *n* is a digit from 1 to 9

RE Examples

- .at matches any three-character string ending with "at", including "hat", "cat", and "bat".
- [^b]at matches all strings matched by .at except "bat".
- [^hc]at matches all strings matched by .at other than "hat" and "cat".
- [hc]at$ matches "hat" and "cat", but only at the end of the string or line.
- \[.\] matches any single character surrounded by "[" and "]" since the brackets are escaped, for example: "[a]" and "[b]".
- s.* matches s followed by zero or more characters, for example: "s" and "saw" and "seed".

Example

```
String line = “tusar0805iitdu";
String pattern =”[a-z]+";
int count = 0;

Pattern r = Pattern.compile(pattern);
Matcher m = r.matcher(line);

while(m.find()) {
    count++;
    System.out.println("Match number "+count);
    System.out.println("start(): "+m.start());
    System.out.println("end(): "+m.end());
    System.out.println(line);
 }
```

RE Syntax

\w	Matches the word characters.
\W	Matches the nonword characters.
\s	Matches the whitespace. Equivalent to [\t\n\r\f].
\S	Matches the nonwhitespace.
\d	Matches the digits. Equivalent to [0-9].
\D	Matches the nondigits.
\A	Matches the beginning of the string.
\Z	Matches the end of the string. If a newline exists, it matches just before newline.
\z	Matches the end of the string.

RE Syntax

\b	Matches the word boundaries when outside the brackets. Matches the backspace (0x08) when inside the brackets.
\n, \t	Matches newlines, carriage returns, tabs, etc.
\G	Matches the point where the last match finished.
\n	Back-reference to capture group number "n".
^abc$	start / end of the string
**\. * **	escaped special characters

Replace in Java

```
String REGEX = "dog";
String INPUT = "The dog says meow. " + "All dogs say meow.";
String REPLACE = "cat";


Pattern p = Pattern.compile(REGEX);
Matcher m = p.matcher(INPUT);
INPUT = m.replaceAll(REPLACE);
System.out.println(INPUT);
```

Additional methods

- If m is a matcher, then
  - m.replaceFirst(*replacement*) returns a new String where the first substring matched by the pattern has been replaced by *replacement*
  - m.replaceAll(r*eplacement*) returns a new String where every substring matched by the pattern has been replaced by *replacement*
  - m.find(*startIndex*) looks for the next pattern match, starting at the specified index
  - m.reset() resets this matcher
  - m.reset(*newText*) resets this matcher and gives it new text to examine (which may be a String, StringBuffer, or CharBuffer)

RE in Python

Regular Expressions in Python

- Regular expressions are a powerful string manipulation tool
- All modern languages have similar library packages for regular expressions
- Use regular expressions to:
  - Search a string (`search` and `match`)
  - Replace parts of a string (`sub`)
  - Break strings into smaller pieces (`split`)

Search and Match

- The two basic functions are **re.search** and **re.match**
  - Search looks for a pattern anywhere in a string
  - Match looks for a match staring at the beginning
- Both return *None* (logical false) if the pattern isn't found and a "match object" instance if it is

```
>>> import re
>>> pat = "a*b"
>>> re.search(pat,"fooaaabcde")
<_sre.SRE_Match object at 0x809c0>
>>> re.match(pat,"fooaaabcde")
>>>
```

Q: What's a match object?

- A: an instance of the match class with the details of the match result

```
>>> r1 = re.search("a*b","fooaaabcde")
>>> r1.group()  # group returns string
  matched
'aaab'
>>> r1.start()  # index of the match start
3
>>> r1.end()    # index of the match end
7
>>> r1.span()   # tuple of (start, end)
(3, 7)
```

What got matched?

- Here's a pattern to match simple email addresses

  \w+@(\w+\.)+(com|org|net|edu)

```
>>> pat1 = "\w+@(\w+\.)+(com|org|net|edu)"
>>> r1 = re.match(pat,"finin@cs.umbc.edu")
>>> r1.group()
'finin@cs.umbc.edu'
```

- We might want to extract the pattern parts, like the email name and host

What got matched?

- We can put parentheses around groups we want to be able to reference

```
>>> pat2 = "(\w+)@((\w+\.)+(com|org|net|edu))"
>>> r2 = re.match(pat2,"finin@cs.umbc.edu")
>>> r2.group(1)
'finin'
>>> r2.group(2)
'cs.umbc.edu'
>>> r2.groups()
r2.groups()
('finin', 'cs.umbc.edu', 'umbc.', 'edu')
```

- Note that the 'groups' are numbered in a preorder traversal of the forest

More re functions

- re.split() is like split but can use patterns

```
>>> re.split("\W+", “This... is a test,
  short and sweet, of split().”)
['This', 'is', 'a', 'test', 'short’,
  'and', 'sweet', 'of', 'split’, ‘’]
```

- re.sub substitutes one string for a pattern

```
>>> re.sub('(blue|white|red)', 'black', 'blue
  socks and red shoes')
'black socks and black shoes’
```

- re.findall() finds al matches

```
>>> re.findall("\d+”,"12 dogs,11 cats, 1 egg")
['12', '11', ’1’]
```

Compiling regular expressions

- If you plan to use a re pattern more than once, compile it to a re object
- Python produces a special data structure that speeds up matching

```
>>> capt3 = re.compile(pat3)
>>> cpat3
<_sre.SRE_Pattern object at 0x2d9c0>
>>> r3 = cpat3.search("finin@cs.umbc.edu")
>>> r3
<_sre.SRE_Match object at 0x895a0>
>>> r3.group()
'finin@cs.umbc.edu'
```

Pattern object methods

Pattern objects have methods that parallel the re functions (e.g., match, search, split, findall, sub), e.g.:

```
>>> p1 = re.compile("\w+@\w+\.+com|org|net|edu")
>>> p1.match("steve@apple.com").group(0)
'steve@apple.com'
>>> p1.search(”Email steve@apple.com today.").group(0)
'steve@apple.com’
>>> p1.findall("Email steve@apple.com and bill@msft.com now.")
['steve@apple.com', 'bill@msft.com’]
>>> p2 = re.compile("[.?!]+\s+")
>>> p2.split("Tired? Go to bed!   Now!! ")
['Tired', 'Go to bed', 'Now', ’ ']
```

email address

sentence boundary

Assignment

- Check the user account name validation (First Name and Last Name)
- Create password protection RE validation; at least 8 character and combination of uppercase, lowercase and digit.
- Check the phone number and email ID