Regular Expressions (Regex)

Regular Expression, or *regex* or *regexp* in short, is extremely and amazingly powerful in searching and manipulating text strings, particularly in processing text files. One line of regex can easily replace several dozen lines of programming codes.

Regex is supported in all the scripting languages (such as Perl, Python, PHP, and JavaScript); as well as general purpose programming languages such as Java; and even word processors such as Word for searching texts. Getting started with regex may not be easy due to its geeky syntax, but it is certainly worth the investment of your time.

1. Regex By Examples

This section is meant for those who need to refresh their memory. For novices, go to the next section to learn the syntax, before looking at these examples.

1.1 Regex Syntax Summary

- **Character**: All characters, except those having special meaning in regex, matches themselves. E.g., the regex `x` matches substring `"x"`; regex `9` matches `"9"`; regex `=` matches `"="`; and regex `@` matches `"@"`.
- **Special Regex Characters**: These characters have special meaning in regex (to be discussed below): `.`, `+`, `*`, `?`, `^`, `$`, `(`, `)`, `[`, `]`, `{`, `}`, `|`, `\`.
- **Escape Sequences (`\char`)**:
  - To match a character having special meaning in regex, you need to use a escape sequence prefix with a backslash (`\`). E.g., `\.` matches `"."`; regex `\+` matches `"+"`; and regex `\(` matches `"("`.
  - You also need to use regex `\\` to match `"\"` (back-slash).
  - Regex recognizes common escape sequences such as `\n` for newline, `\t` for tab, `\r` for carriage-return, `\nnn` for a up to 3-digit octal number, `\xhh` for a two-digit hex code, `\uhhhh` for a 4-digit Unicode, `\uhhhhhhhh` for a 8-digit Unicode.
- **A Sequence of Characters (or String)**: Strings can be matched via combining a sequence of characters (called sub-expressions). E.g., the regex `Saturday` matches `"Saturday"`. The matching, by default, is case-sensitive, but can be set to case-insensitive via *modifier*.
- **OR Operator (`|`)**: E.g., the regex `four|4` accepts strings `"four"` or `"4"`.
- **Character class (or Bracket List)**:
  - `[...]`: Accept ANY ONE of the character within the square bracket, e.g., `[aeiou]` matches `"a"`, `"e"`, `"i"`, `"o"` or `"u"`.
  - `[.-.]` (Range Expression): Accept ANY ONE of the character in the *range*, e.g., `[0-9]` matches any digit; `[A-Za-z]` matches any uppercase or lowercase letters.
  - `[^...]`: NOT ONE of the character, e.g., `[^0-9]` matches any non-digit.
  - Only these four characters require escape sequence inside the bracket list: `^`, `-`, `]`, `\`.
- **Occurrence Indicators (or Repetition Operators)**:
  - `+`: one or more (`1+`), e.g., `[0-9]+` matches one or more digits such as `'123'`, `'000'`.
  - `*`: zero or more (`0+`), e.g., `[0-9]*` matches zero or more digits. It accepts all those in `[0-9]+` plus the empty string.

TABLE OF CONTENTS (HIDE)

  - `?`: zero or one (optional), e.g., `[+-]?` matches an optional `"+"`, `"-"`, or an empty string.
  - `{m,n}`: `m` to `n` (both inclusive)
  - `{m}`: exactly `m` times
  - `{m,}`: `m` or more (`m+`)
- **Metacharacters**: matches a character
  - `.` (dot): ANY ONE character except newline. Same as `[^\n]`
  - `\d`, `\D`: ANY ONE digit/non-digit character. Digits are `[0-9]`
  - `\w`, `\W`: ANY ONE word/non-word character. For ASCII, word characters are `[a-zA-Z0-9_]`
  - `\s`, `\S`: ANY ONE space/non-space character. For ASCII, whitespace characters are `[ \n\r\t\f]`
- **Position Anchors**: does not match character, but position such as start-of-line, end-of-line, start-of-word and end-of-word.
  - `^`, `$`: start-of-line and end-of-line respectively. E.g., `^[0-9]$` matches a numeric string.
  - `\b`: boundary of word, i.e., start-of-word or end-of-word. E.g., `\bcat\b` matches the word `"cat"` in the input string.
  - `\B`: Inverse of \b, i.e., non-start-of-word or non-end-of-word.
  - `\<`, `\>`: start-of-word and end-of-word respectively, similar to `\b`. E.g., `\<cat\>` matches the word `"cat"` in the input string.
  - `\A`, `\Z`: start-of-input and end-of-input respectively.
- **Parenthesized Back References**:
  - Use parentheses `( )` to create a back reference.
  - Use `$1`, `$2`, ... (Java, Perl, JavaScript) or `\1`, `\2`, ... (Python) to retreive the back references in sequential order.
- **Laziness (Curb Greediness for Repetition Operators)**: `*?`, `+?`, `??`, `{m,n}?`, `{m,}?`

1.2 Example: Numbers `[0-9]+` or `\d+`

1. A regex (*regular expression*) consists of a sequence of *sub-expressions*. In this example, `[0-9]` and `+`.
2. The `[...]`, known as *character class* (or *bracket list*), encloses a list of characters. It matches any SINGLE character in the list. In this example, `[0-9]` matches any SINGLE character between 0 and 9 (i.e., a digit), where dash (`-`) denotes the *range*.
3. The `+`, known as *occurrence indicator* (or *repetition operator*), indicates one or more occurrences (`1+`) of the previous sub-expression. In this case, `[0-9]+` matches one or more digits.
4. A regex may match a portion of the input (i.e., substring) or the entire input. In fact, it could match zero or more substrings of the input (with global modifier).
5. This regex matches any numeric substring (of digits 0 to 9) of the input. For examples,
   a. If the input is `"abc123xyz"`, it matches substring `"123"`.
   b. If the input is `"abcxyz"`, it matches nothing.
   c. If the input is `"abc00123xyz456_0"`, it matches substrings `"00123"`, `"456"` and `"0"` (three matches).

   Take note that this regex matches number with leading zeros, such as `"000"`, `"0123"` and `"0001"`, which may not be desirable.
6. You can also write `\d+`, where `\d` is known as a *metacharacter* that matches any digit (same as `[0-9]`). There are more than one ways to write a regex! Take note that many programming languages (C, Java, JavaScript, Python) use backslash `\` as the prefix for escape sequences (e.g., `\n` for newline), and you need to write `"\\d+"` instead.

1.3 Code Examples (Python, Java, JavaScript, Perl, PHP)

Code Example in Python

See "Python's `re` module for Regular Expression" for full coverage.

Python supports Regex via module `re`. Python also uses backslash (`\`) for escape sequences (i.e., you need to write `\\` for `\`, `\\d` for `\d`), but it supports raw string in the form of `r'...'`, which ignore the interpretation of escape sequences - great for writing regex.

```
# Test under the Python Command-Line Interpreter
$ python3
......
>>> import re   # Need module 're' for regular expression

# Try find: re.findall(regexStr, inStr) -> matchedSubstringsList
# r'...' denotes raw strings which ignore escape code, i.e., r'\n' is '\'+'n'
>>> re.findall(r'[0-9]+', 'abc123xyz')
['123']   # Return a list of matched substrings
>>> re.findall(r'[0-9]+', 'abcxyz')
[]
>>> re.findall(r'[0-9]+', 'abc00123xyz456_0')
['00123', '456', '0']
>>> re.findall(r'\d+', 'abc00123xyz456_0')
['00123', '456', '0']

# Try substitute: re.sub(regexStr, replacementStr, inStr) -> outStr
>>> re.sub(r'[0-9]+', r'*', 'abc00123xyz456_0')
'abc*xyz*_*'

# Try substitute with count: re.subn(regexStr, replacementStr, inStr) -> (outStr, count)
>>> re.subn(r'[0-9]+', r'*', 'abc00123xyz456_0')
('abc*xyz*_*', 3)   # Return a tuple of output string and count
```

Code Example in Java

See "Regular Expressions (Regex) in Java" for full coverage.

Java supports Regex in package `java.util.regex`.

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

public class TestRegexNumbers {
   public static void main(String[] args) {

      String inputStr = "abc00123xyz456_0";  // Input String for matching
      String regexStr = "[0-9]+";            // Regex to be matched

      // Step 1: Compile a regex via static method Pattern.compile(), default is case-sensitive
      Pattern pattern = Pattern.compile(regexStr);
      // Pattern.compile(regex, Pattern.CASE_INSENSITIVE);  // for case-insensitive matching

      // Step 2: Allocate a matching engine from the compiled regex pattern,
      //         and bind to the input string
      Matcher matcher = pattern.matcher(inputStr);

      // Step 3: Perform matching and Process the matching results
      // Try Matcher.find(), which finds the next match
      while (matcher.find()) {
         System.out.println("find() found substring \"" + matcher.group()
               + "\" starting at index " + matcher.start()
               + " and ending at index " + matcher.end());
      }

      // Try Matcher.matches(), which tries to match the ENTIRE input (^...$)
      if (matcher.matches()) {
         System.out.println("matches() found substring \"" + matcher.group()
               + "\" starting at index " + matcher.start()
               + " and ending at index " + matcher.end());
      } else {
         System.out.println("matches() found nothing");
      }

      // Try Matcher.lookingAt(), which tries to match from the START of the input (^...)
      if (matcher.lookingAt()) {
         System.out.println("lookingAt() found substring \"" + matcher.group()
               + "\" starting at index " + matcher.start()
               + " and ending at index " + matcher.end());
      } else {
         System.out.println("lookingAt() found nothing");
      }

      // Try Matcher.replaceFirst(), which replaces the first match
      String replacementStr = "**";
      String outputStr = matcher.replaceFirst(replacementStr); // first match only
      System.out.println(outputStr);

      // Try Matcher.replaceAll(), which replaces all matches
      replacementStr = "++";
      outputStr = matcher.replaceAll(replacementStr); // all matches
      System.out.println(outputStr);
   }
}
```

The output is:

```
find() found substring "00123" starting at index 3 and ending at index 8
find() found substring "456" starting at index 11 and ending at index 14
find() found substring "0" starting at index 15 and ending at index 16
matches() found nothing
lookingAt() found nothing
abc**xyz456_0
abc++xyz++_++
```

Code Example in Perl

See "Regular Expression (Regex) in Perl" for full coverage.

Perl makes extensive use of regular expressions with many built-in syntaxes and operators. In Perl (and JavaScript), a regex is delimited by a pair of forward slashes (default), in the form of `/regex/`. You can use built-in operators:

- `m/regex/modifier` or `/regex/modifier`: Match against the *regex*. `m` is optional.
- `s/regex/replacement/modifier`: Substitute matched substring(s) by the replacement.

In Perl, you can use single-quoted non-interpolating string `'....'` to write regex to disable interpretation of backslash (`\`) by Perl.

```
#!/usr/bin/env perl
use strict;
use warnings;

my $inStr = 'abc00123xyz456_0';  # input string
my $regex = '[0-9]+';            # regex pattern string in non-interpolating string

# Try match /regex/modifiers (or m/regex/modifiers)
```

```
my @matches = ($inStr =~ /$regex/g);  # Match $inStr with regex with global modifier
                                      # Store all matches in an array
print "@matches\n";   # Output: 00123 456 0

while ($inStr =~ /$regex/g) {
   # The built-in array variables @- and @+ keep the start and end positions
   #   of the matches, where $-[0] and $+[0] is the full match, and
   #   $-[n] and $+[n] for back references $1, $2, etc.
   print substr($inStr, $-[0], $+[0] - $-[0]), ', ';   # Output: 00123, 456, 0,
}
print "\n";

# Try substitute  s/regex/replacement/modifiers
$inStr =~ s/$regex/**/g;    # with global modifier
print "$inStr\n";           # Output: abc**xyz**_**
```

Code Example in JavaScript

See "Regular Expression in JavaScript" for full coverage.

In JavaScript (and Perl), a regex is delimited by a pair of forward slashes, in the form of `/.../`. There are two sets of methods, issue via a `RegEx` object or a `String` object.

```
<!DOCTYPE html>
<!-- JSRegexNumbers.html -->
<html lang="en">
<head>
<meta charset="utf-8">
<title>JavaScript Example: Regex</title>
<script>
var inStr = "abc123xyz456_7_00";

// Use RegExp.test(inStr) to check if inStr contains the pattern
console.log(/[0-9]+/.test(inStr));  // true

// Use String.search(regex) to check if the string contains the pattern
// Returns the start position of the matched substring or -1 if there is no match
console.log(inStr.search(/[0-9]+/));  // 3

// Use String.match() or RegExp.exec() to find the matched substring,
//   back references, and string index
console.log(inStr.match(/[0-9]+/));  // ["123", input:"abc123xyz456_7_00", index:3, length:"1"]
console.log(/[0-9]+/.exec(inStr));   // ["123", input:"abc123xyz456_7_00", index:3, length:"1"]

// With g (global) option
console.log(inStr.match(/[0-9]+/g));  // ["123", "456", "7", "00", length:4]

// RegExp.exec() with g flag can be issued repeatedly.
// Search resumes after the last-found position (maintained in property RegExp.lastIndex).
var pattern = /[0-9]+/g;
var result;
while (result = pattern.exec(inStr)) {
   console.log(result);
   console.log(pattern.lastIndex);
      // ["123"],  6
      // ["456"], 12
      // ["7"],   14
      // ["00"],  17
}
```

```

// String.replace(regex, replacement):
console.log(inStr.replace(/\d+/, "**"));    // abc**xyz456_7_00
console.log(inStr.replace(/\d+/g, "**"));   // abc**xyz**_**_**
</script>
</head>
<body>
  <h1>Hello,</h1>
</body>
</html>
```

Code Example in PHP

[TODO]

1.4 Example: Full Numeric Strings `^[0-9]+$` or `^\d+$`

1. The leading `^` and the trailing `$` are known as *position anchors*, which match the start and end positions of the line, respectively. As the result, the entire input string shall be matched fully, instead of a portion of the input string (substring).
2. This regex matches any non-empty numeric strings (comprising of digits 0 to 9), e.g., "`0`" and "`12345`". It does not match with "" (empty string), "`abc`", "`a123`", "`abc123xyz`", etc. However, it also matches "`000`", "`0123`" and "`0001`" with leading zeros.

1.5 Example: Positive Integer Literals `[1-9][0-9]*|0` or `[1-9]\d*|0`

1. `[1-9]` matches any character between 1 to 9; `[0-9]*` matches zero or more digits. The `*` is an *occurrence indicator* representing zero or more occurrences. Together, `[1-9][0-9]*` matches any numbers without a leading zero.
2. `|` represents the OR operator; which is used to include the number `0`.
3. This expression matches "`0`" and "`123`"; but does not match "`000`" and "`0123`" (but see below).
4. You can replace `[0-9]` by metacharacter `\d`, but not `[1-9]`.
5. We did not use *position anchors* `^` and `$` in this regex. Hence, it can match any parts of the input string. For examples,
   a. If the input string is "`abc123xyz`", it matches the substring `"123"`.
   b. If the input string is `"abcxyz"`, it matches nothing.
   c. If the input string is `"abc123xyz456_0"`, it matches substrings `"123"`, `"456"` and `"0"` (three matches).
   d. If the input string is `"0012300"`, it matches substrings: `"0"`, `"0"` and `"12300"` (three matches)!!!

1.6 Example: Full Integer Literals `^[+-]?[1-9][0-9]*|0$` or `^[+-]?[1-9]\d*|0$`

1. This regex match an Integer literal (for entire string with the *position anchors*), both positive, negative and zero.
2. `[+-]` matches either `+` or `-` sign. `?` is an *occurrence indicator* denoting 0 or 1 occurrence, i.e. optional. Hence, `[+-]?` matches an optional leading `+` or `-` sign.
3. We have covered three occurrence indicators: `+` for one or more, `*` for zero or more, and `?` for zero or one.

1.7 Example: Identifiers (or Names) `[a-zA-Z_][0-9a-zA-Z_]*` or `[a-zA-Z_]\w*`

1. Begin with one letters or underscore, followed by zero or more digits, letters and underscore.
2. You can use *metacharacter* `\w` for a word character `[a-zA-Z0-9_]`. Recall that *metacharacter* `\d` can be used for a digit `[0-9]`.

1.8 Example: Image Filenames `^\w+\.(gif|png|jpg|jpeg)$`

1. The *position anchors* `^` and `$` match the beginning and the ending of the input string, respectively. That is, this regex shall match the entire input string, instead of a part of the input string (substring).

2. `\w+` matches one or more word characters (same as `[a-zA-Z0-9_]+`).
3. `\.` matches the dot `(.)` character. We need to use `\.` to represent `.` as `.` has special meaning in regex. The `\` is known as the escape code, which restore the original literal meaning of the following character. Similarly, `*`, `+`, `?` (occurrence indicators), `^`, `$` (position anchors) have special meaning in regex. You need to use an escape code to match with these characters.
4. `(gif|png|jpg|jpeg)` matches either "`gif`", "`png`", "`jpg`" or "`jpeg`". The `|` denotes "OR" operator. The parentheses are used for grouping the selections.
5. The *modifier* `i` after the regex specifies case-insensitive matching (applicable to some languages like Perl and JavaScript only). That is, it accepts "`test.GIF`" and "`TesT.Gif`".

1.9 Example: Email Addresses `^\w+([.-]?\w+)*@\w+([.-]?\w+)*(\.\w{2,3})+$`

1. The *position anchors* `^` and `$` match the beginning and the ending of the input string, respectively. That is, this regex shall match the entire input string, instead of a part of the input string (substring).
2. `\w+` matches 1 or more word characters (same as `[a-zA-Z0-9_]+`).
3. `[.-]?` matches an optional character `.` or `-`. Although dot (`.`) has special meaning in regex, in a character class (square brackets) any characters except `^`, `-`, `]` or `\` is a literal, and do not require escape sequence.
4. `([.-]?\w+)*` matches 0 or more occurrences of `[.-]?\w+`.
5. The sub-expression `\w+([.-]?\w+)*` is used to match the username in the email, before the `@` sign. It begins with at least one word character `[a-zA-Z0-9_]`, followed by more word characters or `.` or `-`. However, a `.` or `-` must follow by a word character `[a-zA-Z0-9_]`. That is, the input string cannot begin with `.` or `-`; and cannot contain "`..`", "`--`", "`.-`" or "`-.`". Example of valid string are "`a.1-2-3`".
6. The `@` matches itself. In regex, all characters other than those having special meanings matches itself, e.g., `a` matches `a`, `b` matches `b`, and etc.
7. Again, the sub-expression `\w+([.-]?\w+)*` is used to match the email domain name, with the same pattern as the username described above.
8. The sub-expression `\.\w{2,3}` matches a `.` followed by two or three word characters, e.g., "`.com`", "`.edu`", "`.us`", "`.uk`", "`.co`".
9. `(\.\w{2,3})+` specifies that the above sub-expression could occur one or more times, e.g., "`.com`", "`.co.uk`", "`.edu.sg`" etc.

Exercise: Interpret this regex, which provide another representation of email address: `^[\w\-\.\+]+\@[a-zA-Z0-9\.\-]+\.[a-zA-z0-9]{2,4}$`.

1.10 Example: Swapping Words using Parenthesized Back-References `^(\S+)\s+(\S+)$` and `$2 $1`

1. The `^` and `$` match the beginning and ending of the input string, respectively.
2. The `\s` (lowercase `s`) matches a whitespace (blank, tab `\t`, and newline `\r` or `\n`). On the other hand, the `\S+` (uppercase `S`) matches anything that is NOT matched by `\s`, i.e., non-whitespace. In regex, the uppercase metacharacter denotes the *inverse* of the lowercase counterpart, for example, `\w` for word character and `\W` for non-word character; `\d` for digit and `\D` or non-digit.
3. The above regex matches two words (without white spaces) separated by one or more whitespaces.
4. Parentheses `()` have two meanings in regex:
    a. to group sub-expressions, e.g., `(abc)*`
    b. to provide a so-called *back-reference* for capturing and extracting matches.
5. The parentheses in `(\S+)`, called *parenthesized back-reference*, is used to extract the matched substring from the input string. In this regex, there are two `(\S+)`, match the first two words, separated by one or more whitespaces `\s+`. The two matched words are extracted from the input string and typically kept in special variables `$1` and `$2` (or `\1` and `\2` in Python), respectively.
6. To swap the two words, you can access the special variables, and print "`$2 $1`" (via a programming language); or substitute operator "`s/(\S+)\s+(\S+)/$2 $1/`" (in Perl).

Code Example in Python

Python keeps the parenthesized back references in `\1`, `\2`, ... Also, `\0` keeps the entire match.

```
$ python3
>>> re.findall(r'^(\S+)\s+(\S+)$', 'apple orange')
[('apple', 'orange')]     # A list of tuples if the pattern has more than one back references
# Back references are kept in \1, \2, \3, etc.
>>> re.sub(r'^(\S+)\s+(\S+)$', r'\2 \1', 'apple orange')    # Prefix r for raw string which ignores escape
'orange apple'
>>> re.sub(r'^(\S+)\s+(\S+)$', '\\2 \\1', 'apple orange')   # Need to use \\ for \ for regular string
'orange apple'
```

Code Example in Java

Java keeps the parenthesized back references in `$1`, `$2`, ...

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

public class TestRegexSwapWords {
   public static void main(String[] args) {
      String inputStr = "apple orange";
      String regexStr = "^(\\S+)\\s+(\\S+)$";  // Regex pattern to be matched
      String replacementStr = "$2 $1";         // Replacement pattern with back references

      // Step 1: Allocate a Pattern object to compile a regex
      Pattern pattern = Pattern.compile(regexStr);

      // Step 2: Allocate a Matcher object from the Pattern, and provide the input
      Matcher matcher = pattern.matcher(inputStr);

      // Step 3: Perform the matching and process the matching result
      String outputStr = matcher.replaceFirst(replacementStr); // first match only
      System.out.println(outputStr);   // Output: orange apple
   }
}
```

1.11 Example: HTTP Addresses `^http:\/\/\S+(\/\S+)*(\/)?$`

1. Begin with `http://`. Take note that you may need to write `/` as `\/` with an escape code in some languages (JavaScript, Perl).
2. Followed by `\S+`, one or more non-whitespaces, for the domain name.
3. Followed by `(\/\S+)*`, zero or more "/...", for the sub-directories.
4. Followed by `(\/)?`, an optional (0 or 1) trailing `/`, for directory request.

1.12 Example: Regex Patterns in AngularJS

The following rather complex regex patterns are used by AngularJS in JavaScript syntax:

```javascript
var ISO_DATE_REGEXP = /^\d{4,}-[01]\d-[0-3]\dT[0-2]\d:[0-5]\d:[0-5]\d\.\d+(?:[+-][0-2]\d:[0-5]\d|Z)$/;

var URL_REGEXP = /^[a-z][a-z\d.+-]*:\/*(?:[^:@]+(?::[^@]+)?@)?(?:[^\s:/?#]+|\[[a-f\d:]+])(?::\d+)?(?:\/[^?#]*)?(?:\?[^#]*)?(?:#.*)?$/i;

var EMAIL_REGEXP = /^(?=.{1,254}$)(?=.{1,64}@)[-!#$%&'*+/0-9=?A-Z^_`a-z{|}~]+(\.[-!#$%&'*+/0-9=?A-Z^_`a-z{|}~]+)*@[A-Za-z0-9]([A-Za-z0-9-]{0,61}[A-Za-z0-9])?(\.[A-Za-z0-9]([A-Za-z0-9-]{0,61}[A-Za-
// Match both uppercase and lowercase letters, single-quote but not double-quote

var NUMBER_REGEXP = /^\s*(-|\+)?(\d+|(\d*(\.\d*)))([eE][+-]?\d+)?\s*$/;

var DATE_REGEXP = /^(\d{4,})-(\d{2})-(\d{2})$/;

var DATETIMELOCAL_REGEXP = /^(\d{4,})-(\d\d)-(\d\d)T(\d\d):(\d\d)(?::(\d\d)(\.\d{1,3})?)?$/;

var WEEK_REGEXP = /^(\d{4,})-W(\d\d)$/;

var MONTH_REGEXP = /^(\d{4,})-(\d\d)$/;

var TIME_REGEXP = /^(\d\d):(\d\d)(?::(\d\d)(\.\d{1,3})?)?$/;
```

1.13 Example: Sample Regex in Perl

```
s/^\s+//        # Remove leading whitespaces (substitute with empty string)
s/\s+$//        # Remove trailing whitespaces
s/^\s+.*\s+$//  # Remove leading and trailing whitespaces
```

2. Regular Expression (Regex) Syntax

A Regular Expression (or Regex) is a *pattern* (or *filter*) that describes a set of strings that matches the pattern. In other words, a regex *accepts* a certain set of strings and *rejects* the rest.

A regex consists of a sequence of characters, metacharacters (such as `.`, `\d`, `\D`, `\s`, `\S`, `\w`, `\W`) and operators (such as `+`, `*`, `?`, `|`, `^`). They are constructed by combining many smaller sub-expressions.

2.1 Matching a Single Character

The fundamental building blocks of a regex are patterns that match a *single* character. Most characters, including all letters (`a-z` and `A-Z`) and digits (`0-9`), match itself. For example, the regex `x` matches substring `"x"`; `z` matches `"z"`; and `9` matches `"9"`.

Non-alphanumeric characters without special meaning in regex also matches itself. For example, `=` matches `"="`; `@` matches `"@"`.

2.2 Regex Special Characters and Escape Sequences

Regex's Special Characters

These characters have special meaning in regex (I will discuss in detail in the later sections):

- metacharacter: dot (`.`)
- bracket list: `[ ]`
- position anchors: `^`, `$`
- occurrence indicators: `+`, `*`, `?`, `{ }`
- parentheses: `( )`
- or: `|`
- escape and metacharacter: backslash (`\`)

Escape Sequences

The characters listed above have special meanings in regex. To match these characters, we need to prepend it with a backslash (`\`), known as *escape sequence*. For examples, `\+` matches `"+"`; `\[` matches `"["`; and `\.` matches `"."`.

Regex also recognizes common escape sequences such as `\n` for newline, `\t` for tab, `\r` for carriage-return, `\nnn` for a up to 3-digit octal number, `\xhh` for a two-digit hex code, `\uhhhh` for a 4-digit Unicode, `\uhhhhhhhh` for a 8-digit Unicode.

Code Example in Python

```
$ python3
>>> import re   # Need module 're' for regular expression
# Try find: re.findall(regexStr, inStr) -> matchedStrList
# r'...' denotes raw strings which ignore escape code, i.e., r'\n' is '\'+'n'
>>> re.findall(r'a', 'abcabc')
['a', 'a']
>>> re.findall(r'=', 'abc=abc')   # '=' is not a special regex character
['=']
>>> re.findall(r'\.', 'abc.com')  # '.' is a special regex character, need regex escape sequence
['.']
```

```
>>> re.findall('\\.', 'abc.com')  # You need to write \\ for \ in regular Python string
['.']
```

Code Example in JavaScript

[TODO]

Code Example in Java

[TODO]

2.3 Matching a Sequence of Characters (String or Text)

Sub-Expressions

A regex is constructed by combining many smaller *sub-expressions* or *atoms*. For example, the regex `Friday` matches the string "`Friday`". The matching, by default, is case-sensitive, but can be set to case-insensitive via modifier.

2.4 OR (`|`) Operator

You can provide *alternatives* using the "OR" operator, denoted by a vertical bar '`|`'. For example, the regex `four|for|floor|4` accepts strings "`four`", "`for`", "`floor`" or "`4`".

2.5 Bracket List (Character Class) `[...]`, `[^...]`, `[.-.]`

A *bracket expression* is *a list of characters* enclosed by `[ ]`, also called *character class*. It matches ANY ONE character in the list. However, if the first character of the list is the caret (`^`), then it matches ANY ONE character NOT in the list. For example, the regex `[02468]` matches a single digit `0`, `2`, `4`, `6`, or `8`; the regex `[^02468]` matches any single character other than `0`, `2`, `4`, `6`, or `8`.

Instead of listing all characters, you could use a *range expression* inside the bracket. A range expression consists of two characters separated by a hyphen (`-`). It matches any single character that sorts between the two characters, inclusive. For example, `[a-d]` is the same as `[abcd]`. You could include a caret (`^`) in front of the range to *invert* the matching. For example, `[^a-d]` is equivalent to `[^abcd]`.

Most of the special regex characters lose their meaning inside bracket list, and can be used as they are; except `^`, `-`, `]` or `\`.

- To include a `]`, place it first in the list, or use escape `\]`.
- To include a `^`, place it anywhere but first, or use escape `\^`.
- To include a `-` place it last, or use escape `\-`.
- To include a `\`, use escape `\\`.
- No escape needed for the other characters such as `.`, `+`, `*`, `?`, `(`, `)`, `{`, `}`, and etc, inside the bracket list
- You can also include metacharacters (to be explained in the next section), such as `\w`, `\W`, `\d`, `\D`, `\s`, `\S` inside the bracket list.

Name Character Classes in Bracket List (For Perl Only?)

Named (POSIX) classes of characters are pre-defined within bracket expressions. They are:

- `[:alnum:]`, `[:alpha:]`, `[:digit:]`: letters+digits, letters, digits.
- `[:xdigit:]`: hexadecimal digits.
- `[:lower:]`, `[:upper:]`: lowercase/uppercase letters.
- `[:cntrl:]`: Control characters
- `[:graph:]`: printable characters, except space.
- `[:print:]`: printable characters, include space.
- `[:punct:]`: printable characters, excluding letters and digits.
- `[:space:]`: whitespace

For example, `[[:alnum:]]` means `[0-9A-Za-z]`. (Note that the square brackets in these class names are part of the symbolic names, and must be included in addition to the square brackets delimiting the bracket list.)

2.6 Metacharacters `.`, `\w`, `\W`, `\d`, `\D`, `\s`, `\S`

A *metacharacter* is a symbol with a special meaning inside a regex.

- The metacharacter dot (`.`) matches any single character except newline `\n` (same as `[^\n]`). For example, `...` matches any 3 characters (including alphabets, numbers, whitespaces, but except newline); `the..` matches "`there`", "`these`", "`the  `", and so on.
- `\w` (word character) matches any single letter, number or underscore (same as `[a-zA-Z0-9_]`). The uppercase counterpart `\W` (non-word-character) matches any single character that doesn't match by `\w` (same as `[^a-zA-Z0-9_]`).
- In regex, the uppercase metacharacter is always the *inverse* of the lowercase counterpart.
- `\d` (digit) matches any single digit (same as `[0-9]`). The uppercase counterpart `\D` (non-digit) matches any single character that is not a digit (same as `[^0-9]`).
- `\s` (space) matches any single whitespace (same as `[ \t\n\r\f]`, blank, tab, newline, carriage-return and form-feed). The uppercase counterpart `\S` (non-space) matches any single character that doesn't match by `\s` (same as `[^ \t\n\r\f]`).

Examples:

```
\s\s      # Matches two spaces
\S\S\s    # Two non-spaces followed by a space
\s+       # One or more spaces
\S+\s\S+  # Two words (non-spaces) separated by a space
```

2.7 Backslash (`\`) and Regex Escape Sequences

Regex uses backslash (`\`) for two purposes:

1. for *metacharacters* such as `\d` (digit), `\D` (non-digit), `\s` (space), `\S` (non-space), `\w` (word), `\W` (non-word).
2. to escape special regex characters, e.g., `\.` for `.`, `\+` for `+`, `\*` for `*`, `\?` for `?`. You also need to write `\\` for `\` in regex to avoid ambiguity.
3. Regex also recognizes `\n` for newline, `\t` for tab, etc.

Take note that in many programming languages (C, Java, Python), backslash (`\`) is also used for escape sequences in string, e.g., `"\n"` for newline, `"\t"` for tab, and you also need to write `"\\"` for `\`. Consequently, to write regex pattern `\\` (which matches one `\`) in these languages, you need to write `"\\\\"` (two levels of escape!!!). Similarly, you need to write `"\\d"` for regex metacharacter `\d`. This is cumbersome and error-prone!!!

2.8 Occurrence Indicators (Repetition Operators): `+`, `*`, `?`, `{m}`, `{m,n}`, `{m,}`

A regex sub-expression may be followed by an *occurrence indicator* (aka *repetition operator*):

- `?`: The preceding item is optional and matched at most once (i.e., occurs 0 or 1 times or optional).
- `*`: The preceding item will be matched zero or more times, i.e., `0+`
- `+`: The preceding item will be matched one or more times, i.e., `1+`
- `{m}`: The preceding item is matched exactly m times.
- `{m,}`: The preceding item is matched m or more times, i.e., `m+`
- `{m,n}`: The preceding item is matched at least m times, but not more than n times.

For example: The regex `xy{2,4}` accepts "`xyy`", "`xyyy`" and "`xyyyy`".

2.9 Modifiers

You can apply modifiers to a regex to tailor its behavior, such as global, case-insensitive, multiline, etc. The ways to apply modifiers differ among languages.

In Perl, you can attach *modifiers* after a regex, in the form of `/.../`*`modifiers`*. For examples:

```
m/abc/i     # case-insensitive matching
m/abc/g     # global (Match ALL instead of match first)
```

In Java, you apply modifiers when compiling the regex `Pattern`. For example,

```
Pattern p1 = Pattern.compile(regex, Pattern.CASE_INSENSITIVE);  // for case-insensitive matching
Pattern p2 = Pattern.compile(regex, Pattern.MULTILINE);         // for multiline input string
Pattern p3 = Pattern.compile(regex, Pattern.DOTALL);            // Dot (.) matches all characters including newline
```

The commonly-used modifer modes are:

- Case-Insensitive mode (or `i`): case-insensitive matching for letters.
- Global (or `g`): match All instead of first match.
- Multiline mode (or `m`): affect `^`, `$`, `\A` and `\Z`. In multiline mode, `^` matches start-of-line or start-of-input; `$` matches end-of-line or end-of-input, `\A` matches start-of-input; `\Z` matches end-of-input.
- Single-line mode (or `s`): Dot (`.`) will match all characters, including newline.
- Comment mode (or `x`): allow and ignore embedded comment starting with `#` till end-of-line (EOL).
- more...

2.10 Greediness, Laziness and Backtracking for Repetition Operators

Greediness of Repetition Operators `*`, `+`, `?`, `{m,n}`: The repetition operators are *greedy operators*, and by default grasp as many characters as possible for a match. For example, the regex `xy{2,4}` try to match for "`xyyyy`", then "`xyyy`", and then "`xyy`".

Lazy Quantifiers `*?`, `+?`, `??`, `{m,n}?`, `{m,}?`, : You can put an extra `?` after the repetition operators to curb its greediness (i.e., stop at the shortest match). For example,

```
input = "The <code>first</code> and <code>second</code> instances"
regex = <code>.*</code> matches "<code>first</code> and <code>second</code>"
But
regex = <code>.*?</code> produces two matches: "<code>first</code>" and "<code>second</code>"
```

Backtracking: If a regex reaches a state where a match cannot be completed, it backtracks by unwinding one character from the greedy match. For example, if the regex `z*zzz` is matched against the string "`zzzz`", the `z*` first matches "`zzzz`"; unwinds to match "`zzz`"; unwinds to match "`zz`"; and finally unwinds to match "`z`", such that the rest of the patterns can find a match.

Possessive Quantifiers `*+`, `++`, `?+`, `{m,n}+`, `{m,}+`: You can put an extra `+` to the repetition operators to disable backtracking, even it may result in match failure. e.g, `z++z` will not match "`zzzz`". This feature might not be supported in some languages.

2.11 Position Anchors `^`, `$`, `\b`, `\B`, `\<`, `\>`, `\A`, `\Z`

Positional anchors DO NOT match actual character, but matches *position* in a string, such as start-of-line, end-of-line, start-of-word, and end-of-word.

- `^` and `$`: The `^` matches the start-of-line. The `$` matches the end-of-line excluding newline, or end-of-input (for input not ending with newline). These are the most commonly-used position anchors. For examples,

```
ing$          # ending with 'ing'
^testing 123$  # Matches only one pattern. Should use equality comparison instead.
^[0-9]+$       # Numeric string
```

- `\b` and `\B`: The `\b` matches the boundary of a word (i.e., start-of-word or end-of-word); and `\B` matches inverse of `\b`, or non-word-boundary. For examples,

```
\bcat\b        # matches the word "cat" in input string "This is a cat."
               # but does not match input "This is a catalog."
```

- `\<` and `\>`: The `\<` and `\>` match the start-of-word and end-of-word, respectively (compared with `\b`, which can match both the start and end of a word).
- `\A` and `\Z`: The `\A` matches the start of the input. The `\Z` matches the end of the input.
  They are different from `^` and `$` when it comes to matching input with multiple lines. `^` matches at the start of the string and after each line break, while `\A` only matches at the start of the string. `$` matches at the end of the string and before each line break, while `\Z` only matches at the end of the string. For examples,

```
$ python3
# Using ^ and $ in multiline mode
>>> p1 = re.compile(r'^.+$', re.MULTILINE)  # . for any character except newline
>>> p1.findall('testing\ntesting')
['testing', 'testing']
>>> p1.findall('testing\ntesting\n')
['testing', 'testing']
   # ^ matches start-of-input or after each line break at start-of-line
   # $ matches end-of-input or before line break at end-of-line
   # newlines are NOT included in the matches

# Using \A and \Z in multiline mode
>>> p2 = re.compile(r'\A.+\Z', re.MULTILINE)
>>> p2.findall('testing\ntesting')
[]    # This pattern does not match the internal \n
>>> p3 = re.compile(r'\A.+\n.+\Z', re.MULTILINE)  # to match the internal \n
>>> p3.findall('testing\ntesting')
['testing\ntesting']
>>> p3.findall('testing\ntesting\n')
[]    # This pattern does not match the trailing \n
   # \A matches start-of-input and \Z matches end-of-input
```

2.12 Capturing Matches via Parenthesized Back-References & Matched Variables `$1`, `$2`, ...

Parentheses `( )` serve two purposes in regex:

1. Firstly, parentheses `( )` can be used to group sub-expressions for overriding the precedence or applying a repetition operator. For example, `(abc)+` (accepts `abc`, `abcabc`, `abcabcabc`, ...) is different from `abc+` (accepts `abc`, `abcc`, `abccc`, ...).
2. Secondly, parentheses are used to provide the so called *back-references* (or *capturing groups*). A back-reference contains the *matched substring*. For examples, the regex `(\S+)` creates one back-reference `(\S+)`, which contains the first word (consecutive non-spaces) of the input string; the regex `(\S+)\s+(\S+)` creates two back-references: `(\S+)` and another `(\S+)`, containing the first two words, separated by one or more spaces `\s+`.

These back-references (or capturing groups) are stored in special variables `$1`, `$2`, ... (or `\1`, `\2`, ... in Python), where `$1` contains the substring matched the first pair of parentheses, and so on. For example, `(\S+)\s+(\S+)` creates two back-references which matched with the first two words. The matched words are stored in `$1` and `$2` (or `\1` and `\2`), respectively.

Back-references are important to manipulate the string. Back-references can be used in the substitution string as well as the pattern. For examples,

```
# Swap the first and second words separated by one space
s/(\S+) (\S+)/$2 $1/;                    # Perl
re.sub(r'(\S+)  (\S+)', r'\2 \1', inStr)  # Python

# Remove duplicate word
s/(\w+) $1/$1/;                   # Perl
re.sub(r'(\w+)  \1', r'\1', inStr)  # Python
```

2.13 (Advanced) Lookahead/Lookbehind, Groupings and Conditional

These feature might not be supported in some languages.

Positive Lookahead `(?=pattern)`

The `(?=pattern)` is known as *positive lookahead*. It performs the match, but does not capture the match, returning only the result: match or no match. It is also called *assertion* as it does not consume any characters in matching. For example, the following complex regex is used to match email addresses by AngularJS:

```
^(?=.{1,254}$)(?=.{1,64}@)[-!#$%&'*+/0-9=?A-Z^_`a-z{|}~]+(\.[-!#$%&'*+/0-9=?A-Z^_`a-z{|}~]+)*@[A-Za-z0-9]([A-Za-z0-9-]{0,61}[A-Za-z0-9])?(\.[A-Za-z0-9]([A-Za-z0-9-]{0,61}[A-Za-z0-9])?)*$
```

The first positive lookahead patterns `^(?=.{1,254}$)` sets the maximum length to 254 characters. The second positive lookahead `^(?=.{1,64}@)` sets maximum of 64 characters before the `'@'` sign for the username.

Negative Lookahead `(?!pattern)`

Inverse of `(?=pattern)`. Match if `pattern` is missing. For example, `a(?=b)` matches `'a'` in `'abc'` (not consuming `'b'`); but not `'acc'`. Whereas `a(?!b)` matches `'a'` in `'acc'`, but not `abc`.

Positive Lookbehind `(?<=pattern)`

[TODO]

Negative Lookbehind `(?<!pattern)`

[TODO]

Non-Capturing Group `(?:pattern)`

Recall that you can use Parenthesized Back-References to capture the matches. To disable capturing, use `?:` inside the parentheses in the form of `(?:pattern)`. In other words, `?:` disables the creation of a capturing group, so as not to create an unnecessary capturing group.

Example: [TODO]

Named Capturing Group `(?<name>pattern)`

The capture group can be referenced later by *`name`*.

Atomic Grouping `(>pattern)`

Disable backtracking, even if this may lead to match failure.

Conditional `(?(Cond)then|else)`

[TODO]

2.14 Unicode

The metacharacters `\w`, `\W`, (word and non-word character), `\b`, `\B` (word and non-word boundary) recongize Unicode characters.

[TODO]

3. Regex in Programming Languages

Python: See "Python `re` module for Regular Expression"

Java: See "Regular Expressions in Java"

JavaScript: See "Regular Expression in JavaScript"

Perl: See "Regular Expressions in Perl"

PHP: [Link]

C/C++: [Link]

REFERENCES & RESOURCES

1. (Python) Python's Regular Expression HOWTO @ https://docs.python.org/3/howto/regex.html (Python 3).

2. (Python) Python's re - Regular expression operations @ https://docs.python.org/3/library/re.html (Python 3).
3. (Java) Online Java Tutorial's Trail on "Regular Expressions" @ https://docs.oracle.com/javase/tutorial/essential/regex/index.html.
4. (Java) JavaDoc for `java.util.regex` Package @ https://docs.oracle.com/javase/10/docs/api/java/util/regex/package-summary.html (JDK 10).
5. (Perl) perlrequick - Perl regular expressions quick start @ https://perldoc.perl.org/perlrequick.html.
6. (Perl) perlre - Perl regular expressions @ https://perldoc.perl.org/perlre.html.
7. (JavaScript) Regular Expressions @ https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions.

Last modified: November, 2018