Regular Expressions

Notes

Author	Faram Khambatta
Created	2014-08-15
Last updated	2014-08-15

Overview

A Regular Expression (Regex) is a group of characters contained in a string. If ^ and *$* are used the *whole* string is used to evaluate the match.

Perl

```
"Hello World" =~ /World/
```

Use *m* to change regex delimiters,

```
"Hello World" =~ m!World!
```

Metachars

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

To specify *where* in string regex should match, use *anchors* **^** and **$**

Special chars within char class [...]

-] \ ^ $

Regex Modifiers

Placed after end delimiter of regex.

i	case insensitive
g	match regex within string as many times as possible
r	non-destructive substitution

Extracting Matches and Backreferences

/(\w+)\s(\d+)/

$1	$2	← These variables are used outside the regex
\g1	\g2	← Backreferences used within the regex

Match as many times as possible

```
$a = "hello cat";
while($a =~ /(\w+)/g) {
        say $1;         # will print 'hello' then 'cat'
}
```

Search and Replace

s/regex/replacement string/modifiers

```
$a =~ s/\s(\w+)\s+/$1/g;
```

If orginal variable needs to be left as is then use modifier *r* like

```
$b = $a =~ s/regex/replacement/r;
```

Split string

```
split  /regex/,  string
```

In above, *regex* represents the delimiter on which to split the string.

e.g.

```
@words = split  /\s+/,  $someString;
```

Pre-compiled Regex

```
$var = qr/regex/;
```

Now *$var* contains compiled regex and can be used anywhere.

Quantifiers

? * + { } are normally *Greedy* and will try to match as much of the string as possible then backoff if required to match regex.

Appending **?** to above quantifiers makes them *Reluctant* i.e. they try to match as little as possible.

Possessive quantifiers are created by appending **+** and are greedy except that they don't backoff even if doing so can make the string match.

Non-Capture Groups

Groups within **(...)** are extracted as **$1, $2,** etc.

Groups within **(?: ..)** won’t be extracted.

Java

```
Pattern p = Pattern.compile(regex);
Matcher m = p.matcher(someString);
```

Metachars

*< > ( ) [ ] { } ^ $ \ - = ! | ? * . +*

(19 chars)

Predefined char classes within strings

Escape the backslash.
e.g. to use **\w** in a regex,

```
String regex = "\\w";
```

Groups

To *Backreference* a group *within* a regex, use **\1, \2,** etc.

Non-capturing groups are denoted as **(?: …)**

Literal Backslash within regex

To put a literal \ within a regex,

```
String regex = "\\\\";              // 4 backslashes to escape a single literal \
```

Methods of Pattern class

A Pattern can be compiled with flags using two argument version of *compile()*.

Flags can be used for case-insensitive matches, include \n in **. (dot)** matches, etc.

e.g.

```
Pattern p = Pattern.compile("regex", Pattern.CASE_INSENSITIVE);
```

Methods of Matcher class

There are methods to find matches, extract groups, replace, etc.

In *replace* methods, in replacement string **$1, $2,** etc refer to back references.

Regex methods of class String

matches(), split(), replaceFirst(), relaceAll()

Groovy

Pre-compiled Pattern

```
p = ~/regex/
```

Matcher

```
str = “hello world”
m = str =~ /world/          // m is a Matcher
```

Matcher evaluates to boolean true/false so **=~** can be used in *if* conditions, like in Perl.

==~ requires a match on a *whole* string. It returns a boolean (not a Matcher).

Capture Groups

m[0][0]	represents whole matched substring
m[0][1]	first capture group
m[0][2]	second capture group

The first dimension of matcher array is match instance while second dimension is the capture group number.

Non-capture groups are denoted by **(?: …)**

Back References in String.replaceAll()

If back references ($1, $2, etc) are used then enclose replacement string within **‘ ...’** and **not** within **“ … “** otherwise Groovy will think it is a G string.

e.g.

```
str.replaceAll(/regex/, ‘$1$2’)
```

Groovy String Quoting

“…“   ‘…‘   / … / $/ … /$

Use whichever is convenient depending on which literal characters need to be escaped the most. All these alternatives will interpret embedded variables except the single quotes alternative.