Regular Expressions in Ruby

Vishal Telangre (Webonise Lab, Pune)

August 14, 2014

```ruby
/web/ =~ 'webonise' #=> 0
/web/.match('webonise') #=> #<MatchData "web">
```

Or

```
/\w([aeiou]b|[aeiou]s)\w/.match("Webonise")
#=> #<MatchData "Webo" 1:"eb">
/\w([aeiou]s|[aeiou]e)\w/.match("Webonise")
#=> #<MatchData "nise" 1:"is">
```

Metacharacters and Escapes

- `(, ), [, ], {, }, ., ?, +, *`
- Arbitrary Ruby expressions can be embedded into patterns with the `#{...}` construct.
- To match a backslash literally backslash-escape that: `\\\`.
- `/[ab]/` means a or b
- `[abcd]` is equivalent to `[a-d]`
- If the first character of a character class is a caret (`^`) the class is inverted: it matches any character except those named.

```
/[^a-eg-z]/.match('f') #=> #<MatchData "f">
```

`/./` - Any character except a newline.
`/./m` - Any character (the `m` modifier enables multiline mode)
`/\w/` - A word character (`[a-zA-Z0-9_]`)
`/\W/` - A non-word character (`[^a-zA-Z0-9_]`)
`/\d/` - A digit character (`[0-9]`)
`/\D/` - A non-digit character (`[^0-9]`)
`/\h/` - A hexdigit character (`[0-9a-fA-F]`)
`/\H/` - A non-hexdigit character (`[^0-9a-fA-F]`)
`/\s/` - A whitespace character: /`[\t\r\n\f]`/
`/\S/` - A non-whitespace character: /`[^\t\r\n\f]`/

Repetitions

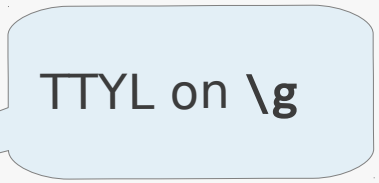
`*` - Zero or more times
`+` - One or more times
`?` - Zero or one times (optional)
`{n}` - Exactly `n` times
`{n,}` - `n` or more times
`{,m}` - `m` or less times
`{n,m}` - At least `n` and at most `m` times

Both patterns below match the string. The first uses a greedy quantifier so `.+` matches `<a><b>`; the second uses a lazy quantifier so `.+?` matches `<a>`.

```
/<.+>/.match("<a><b>")  #=> #<MatchData "<a><b>">
/<.+?>/.match("<a><b>") #=> #<MatchData "<a>">
```

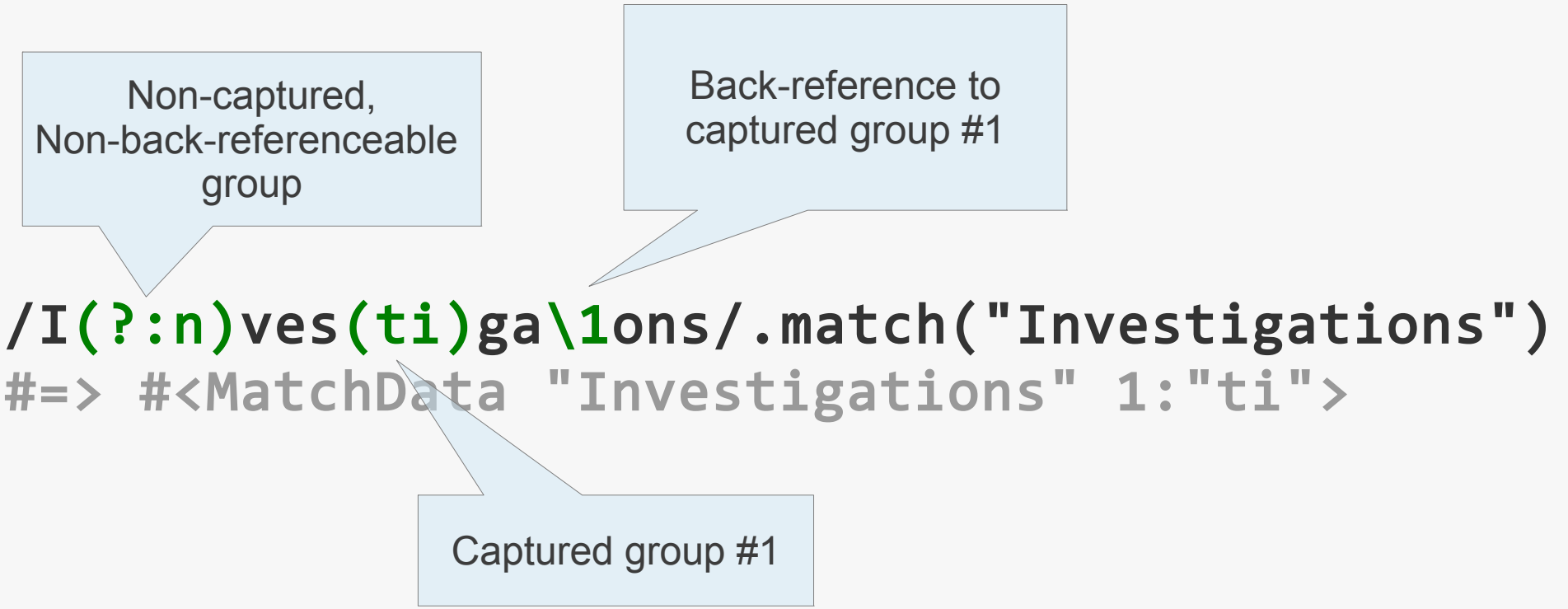
Named Groups (Capturing)

```
rx = /\A(?<name>.*)\.(?<ext>.*)\z/
"kittens.jpg".match(rx)
rx["name"] #=> kittens

"kittens.jpg" =~ rx
$~["name"] #=> "kittens"              NOTE: $~ is last match

/\$(?<dollars>\d+)\.(?<cents>\d+)/.match("$3.67")
#=> #<MatchData "$3.67" dollars:"3" cents:"67">
/\$(?<dollars>\d+)\.(?<cents>\d+)/.match("$3.67")[:dollars] #=> "3"

"kitty.jpg"[/\.(.*)\z/] #=> ".jpg"
"kitty.jpg"[/\.(.*)\z/, 1] #=> "jpg"
```




```
/\A
  (?<expr>
    (?:\d+
      | \( \g<expr> \)
    )+
  )\z
/x.match "((1(2)3)(45))"
#=> #<MatchData "((1(2)3)(45))" expr:"((1(2)3)(45))">
```

Grouping and back-referencing



Options

`/foo/i` - Ignore case
`/foo/m` - Treat a newline as a character matched by **`.`**
`/foo/x` - Ignore whitespace and comments in the pattern
`/foo/o` - Perform **`#{}`** interpolation only once

`i`, **`m`**, and **`x`** can also be applied on the subexpression level with the **`(?on-off)`** construct, which enables options **`on`**, and disables options **`off`** for the expression enclosed by the parentheses.

```
/a(?i:b)c/.match('aBc') #=> #<MatchData "aBc">
/a(?i:b)c/.match('abc') #=> #<MatchData "abc">
```

```ruby
"Kittens are cute!".scan(/\w+/)
#=> ["Kittens", "are", "cute"]

%r(this|that)
#=> /this|that/

%r!
  (?<num>\d+) |
  (?<var>\p{L}+) |
  (?<op>[+\-*/])
!x

Regexp.new("tobe|nottobe")
#=> /tobe|nottobe/
```

Anchors

^ - Matches beginning of line
$ - Matches end of line
\A - Matches beginning of string.
\Z - Matches end of string. If string ends with a newline, it matches just before newline
\z - Matches end of string
\G - Matches point where last match finished
\b - Matches word boundaries when outside brackets; backspace (0x08) when inside brackets
(?=foo) - Positive lookahead assertion: ensures that the following characters match **foo**, but doesn't include those characters in the matched text
(?!foo) - Negative lookahead assertion: ensures that the following characters do not match **foo**, but doesn't include those characters in the matched text
(?<=foo) - Positive lookbehind assertion: ensures that the preceding characters match **foo**, but doesn't include those characters in the matched text
(?<!foo) - Negative lookbehind assertion: ensures that the preceding characters do not match **foo**, but doesn't include those characters in the matched text

Anchors

- Anchoring the pattern to the beginning of the string forces the match to start there. `real` doesn't occur at the beginning of the string, so now the match fails.

```
/\Areal/.match("surrealist") #=> nil
```

- The match below fails because although `Demand` contains `and`, the pattern does not occur at a word boundary.

```
/\band/.match("Demand") #=> nil
```

- Whereas in the following example `and` has been anchored to a non-word boundary so instead of matching the first `and` it matches from the fourth letter of `demand` instead.

```
/\Band.+/.match("Supply and demand curve")
#=> #<MatchData "and curve">
```

- The pattern below uses positive lookahead and positive lookbehind to match text appearing in `<b></b>` tags without including the tags in the match.

```
/(?<=<b>)\w+(?=<\/b>)/.match("Fortune favours the <b>bold</b>")
#=> #<MatchData "bold">
```

Subexpression Calls

```
/\A(?<paren>\(\g<paren>*\))*\z/ =~ '(())' #=> 0
```

1. Matches at the beginning of the string, i.e. before the first character.
2. Enters a named capture group called **`paren`**
3. Matches a literal **`(`**, the first character in the string
4. Calls the **`paren`** group again, i.e. recurses back to the second step
5. Re-enters the **`paren`** group
6. Matches a literal **`(`**, the second character in the string
7. Try to call **`paren`** a third time, but fail because doing so would prevent an overall successful match
8. Match a literal **`)`**, the third character in the string. Marks the end of the second recursive call
9. Match a literal **`)`**, the fourth character in the string
10. Match the end of the string

Questions?