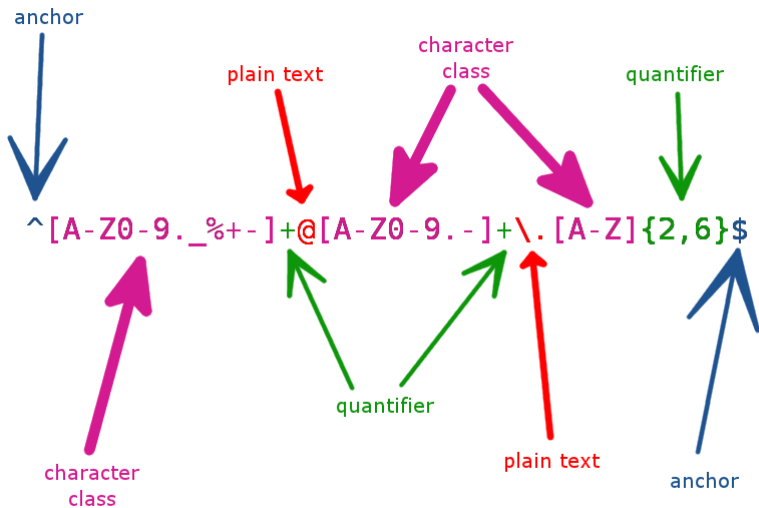


Regular Expressions Primer

Jeremy Stephens
Computer Systems Analyst
Department of Biostatistics

December 18, 2015



What are they?

Regular expressions are a way to describe patterns in text.

Why use them?

- ▶ To find stuff

```
haystack <- c("abcdef", "anbeceddelfe", "abcdef")  
grep("n.e.e.d.l.e", haystack) #=> [1] 2
```

Why use them?

- ▶ To find stuff

```
haystack <- c("abcdef", "anbeceddelfe", "abcdef")  
grep("n.e.e.d.l.e", haystack) #=> [1] 2
```

- ▶ To remove cruft

```
x <- c("123", "123 oz", "123 ounces")  
sub("\\D+$", "", x) #=> [1] "123" "123" "123"
```

Why use them?

- ▶ To find stuff

```
haystack <- c("abcdef", "anbecedelfe", "abcdef")  
grep("n.e.e.d.l.e", haystack) #=> [1] 2
```

- ▶ To remove cruft

```
x <- c("123", "123 oz", "123 ounces")  
sub("\\D+$", "", x) #=> [1] "123" "123" "123"
```

- ▶ To extract data

```
x <- "The Predators lost in overtime with 4:43 left on the clock."  
gsub("^.(\\d{1,2}:\\d{2}).+$", "\\1", x) #=> "4:43"
```


Don't Panic!

Things you (might) need to know

There are 3 main regular expression standards:

- ▶ POSIX Basic Regular Expressions
- ▶ POSIX Extended Regular Expressions (R uses this by default)
- ▶ Perl-based Regular Expressions (R has support for this as well)

Using regular expressions in R

- ▶ `grep('pattern', text, ...)`
- ▶ `sub('pattern', replacement, text, ...)`
- ▶ `regexpr('pattern', text, ...)`

Using regular expressions in other languages

- ▶ Ruby:

```
text =~ /pattern/
```

- ▶ Javascript:

```
text.match(/pattern/)
```

- ▶ Python:

```
re.match('pattern', text)
```

The simplest regular expression

Plain text!

```
grep("test", c("foo", "bar", "test")) #=> [1] 3
```

The pattern "test" is a perfectly valid regular expression.

Metacharacters

A "metacharacter" in a regular expression is a character with special meaning.

Your first metacharacter: dot

A "dot" (or period) means match any one character.

```
grep("foo.bar", c("fooxbar", "foo bar", "foobar", "foo12bar")) #=> [1] 1 2
```

In the above example, "foo.bar" is the regular expression.

Your first metacharacter: dot (cont.)

Using multiple wildcards:

```
grep("foo..bar", c("foo12bar", "foo123bar")) #=> [1] 1  
grep("foo...bar", c("foo12bar", "foo123bar")) #=> [1] 2
```

Variable-length matching: plus

Use `+` to indicate **one or more**.

```
grep("foo.+bar", c("fooxbar", "foo bar", "foobar", "foo12bar")) #=> [1] 1 2 4
```


Variable-length matching: plus (cont.)

Using `+` works on normal characters too:

```
grep("a+rgh", c("argh", "aaargh", "aaaaaaaargh", "ugh")) #=> [1] 1 2 3
```

Quantifiers

Quantifiers are metacharacters that describe "how many", like the + character.

Matching zero or more: asterisk

Use * to indicate **zero or more**.

```
grep("foo.*bar", c("fooxbar", "foo bar", "foobar", "foo12bar")) #=> [1] 1 2 3 4
```

Matching zero or one: question mark

Use `?` to indicate **zero or one**.

```
grep("abc?def", c("abdef", "abcdef", "abccdef")) #=> [1] 1 2
```

Matching n times: curly braces

Use `{}` to indicate **exactly n times**.

```
grep("10{6}", c("1000", "1000000")) => [1] 2
```

Matching n to m times: curly braces

You can also use `{}` to indicate a range.

```
grep("10{2,4}1", c("101", "1001", "10001", "100001", "1000001")) #=> [1] 2 3 4
```

Intermission



Anchor down

Consider the following:

```
grep("10{3}", c("100", "1000", "10000")) #=> [1] 2 3
```


Anchoring to the end: dollar sign

Use \$ to indicate **anchoring at the end**.

```
grep("10{3}$", c("100", "1000", "10000")) #=> [1] 2
```

Anchoring to the beginning: caret

Use `^` to indicate **anchoring at the beginning**.

```
grep("^10{3}", c("1000", "abc1000")) #=> [1] 1
```

Using both anchors

You can use both `^` and `$` to anchor at both ends.

```
grep("10{3}", c("abc1000", "1000", "10000")) #=> [1] 1 2 3
grep("10{3}$", c("abc1000", "1000", "10000")) #=> [1] 1 2
grep("^10{3}$", c("abc1000", "1000", "10000")) #=> [1] 2
```

Matching groups of characters: brackets

Use `[]` to indicate a **group of characters**, also known as a **character class**.

```
grep("ab[cd ef]", c("abc", "abd", "abe", "abf", "abg")) #=> [1] 1 2 3 4
```

Matching groups of characters: brackets (cont.)

You can also use `[]` with a range of characters.

```
grep("ab[c-f]", c("abc", "abd", "abe", "abf", "abg")) #=> [1] 1 2 3 4
```

Matching groups of characters: brackets (cont.)

Using `[]` with multiple ranges:

```
grep("ab[c-fC-F]", c("abc", "abC", "abf", "abF", "abg")) #=> [1] 1 2 3 4
```

Matching groups of characters: brackets (cont.)

Mix and match ranges and characters:

```
grep("ab[c-fC-F123]", c("abc", "abF", "ab1", "ab2")) #=> [1] 1 2 3 4
```

Context in character classes

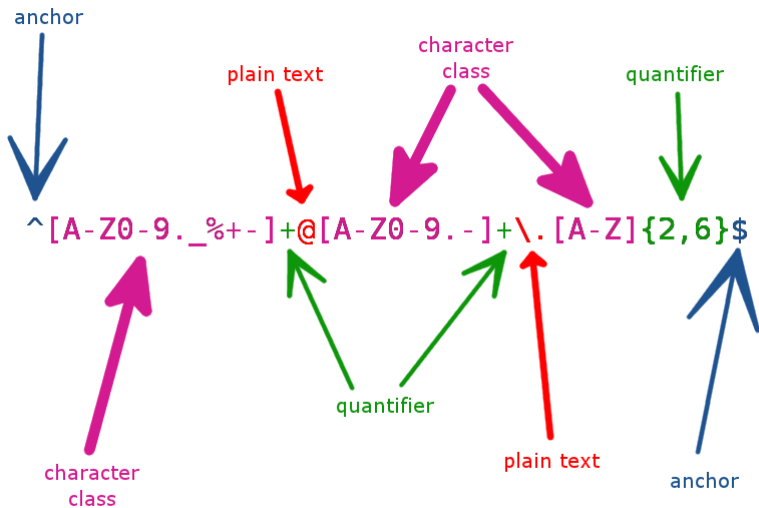
Most metacharacters lose their meaning inside character classes:

```
grep("[.+*]", c(".", "+", "*", "x")) #=> [1] 1 2 3
grep("[a-c-]", c("a", "b", "c", "-")) #=> [1] 1 2 3 4
```


Using quantifiers with character classes

It's possible to put quantifiers on character classes:

```
grep("[a-z]+", c("123", "abcdef")) #=> [1] 2
```



Negating a character class

Use `^` inside a character class to **negate** it.

```
grep("[^a-z]+", c("123", "abcdef")) #=> [1] 1
```

Built-in character classes

Shortcut	Expanded
<code>\d</code>	<code>[0-9]</code>
<code>\w</code>	<code>[a-zA-Z0-9_]</code>
<code>\s</code>	<code>[\t\n\r\f]</code>
<code>\D</code>	<code>[^0-9]</code>
<code>\W</code>	<code>[^a-zA-Z0-9_]</code>
<code>\S</code>	<code>[^ \t\n\r\f]</code>

A note about backslashes in R

To use built-in character classes in R, you need to **escape** the backslashes.

```
grep("\\d", c("123", "abcdef")) #=> Error!  
grep("\\d", c("123", "abcdef")) #=> [1] 1
```

A note about backslashes in R (cont.)

To match a literal backslash in a regular expression in R...

```
grep("\\\\", c("123", "123\\")) #=> [1] 2  
grep("\\\\\\\\", c("123", "123\\\\")) #=> [1] 2
```

Grouping

Use () to specify a **group** to be referenced later using `\[number]`.

```
sub("foo(.+)", "\\1", c("foobar", "foobaz")) #=> [1] bar baz
```

Advanced usage: negative lookahead

Use `?!` inside a group to indicate a **negative lookahead** match.
This requires `perl=TRUE` in R functions.

```
grep("foo(?!bar).+", c("foo123", "foobar", "foo123bar"), perl=TRUE) #=> [1] 1 3
```


Advanced usage: negative lookbehind

Use `?<!` inside a group to indicate a **negative lookbehind** match. This requires `perl=TRUE` in R functions.

```
grep("(?!foo)bar", c("bar", "foobar", "123bar"), perl=TRUE) #=> [1] 1 3
```

Regular expressions on the command line

You can use regular expressions with the `grep` and `sed` commands in your terminal.

```
grep -E 'foo.+bar' *.txt  
egrep 'foo.+bar' *.txt  
sed 's/foo/bar/' *.txt
```

References

- ▶ <http://www.regular-expressions.info/>
- ▶ `man perlre` from a terminal
- ▶ `?sub` in R
- ▶ RegularExpressionPrimer on the Biostatistics Wiki

The End

