Lecture 5, Regular Expressions

36-350

10 September 2014

In Our Last Thrilling Episode

- Characters and strings
- Matching strings, splitting on strings, counting strings
- We need a ways to compute with patterns of strings

Agenda

- Patterns of strings: regular expressions
- Grammar of regular expressions
- Splitting, searching, replacing
- Capture groups

Why We Need String Patterns

Split entries in a data file separate by commas:

```r
strsplit(some_text, split=",")
```

Split entries in a data file separated by one space:

```r
strsplit(some_text, split= " ")
```

```r
=== Split entries in a data file separated by a comma, then a space:
strsplit(text, split=", ")
```

Split entries in a data file separated by a comma, then optionally some spaces:

```r
???????
```

Regular Expressions

- We need a language for telling R about patterns of strings
- The most basic such language is that of regular expressions
- Regular expressions match sets of strings
- Start with string constants, and build up by allowing “this and then that”, “either this or that”, “repeat this”
- These rules get expressed in a grammar, with special symbols
Grammar of Regular Expressions

- Every string is a valid regexp
  - `fly` matches end of `fruitfly`, `why walk when you can fly`
  - `does not match time flies like an arrow; fruit flies like a banana; a banana flies poorly`
- OR of two regexps is a regexp, write with `|`
  - `fly|flies`
- Concatenation of two regexps is a regexp
  - `time|fruit fly|flies`
- Parentheses create groups: `(time|fruit) (fly|flies)`

Escaping, Ranges

- **Escape** special characters with a leading `\` to match them
- Use braces `[]` to indicate character ranges
  - `[a-z]`, `[0-9]`, many pre-named ones like `[:punct:]` for punctuation marks
- **Negate** a character range with a leading `^`
  - `[^aeiou]` = anything except a lower-case vowel
- The period `.` stands for any character, no brackets needed

Quantifiers in Regexps

How often? - `+` after a regexp means “1 or more times” - `*` means “0 or more times” - `?` means “0 or 1 times” (optional, once) - `{n}` means “exactly n times” - `{n,m}` means “between n and m times (inclusive)”

Some redundancy, e.g., can fake `+` with `*`

Quantifier Scope

- By default, quantifiers are “greedy”, match as many repetitions as they can
- Following a quantifier by `?` makes it match as few as possible
  - `\[.+\]` matches all of `[i][j]`, but `\[.+?\]` just matches `[i]`
- By default, quantifiers apply to last character; use parentheses
  - `H(TT)+` vs. `(HH|TT)+`

Anchoring

- `$` means a pattern can only match at the beginning of a line or string
- `^` means (outside of braces) the end of a line or string
- `< and >` anchor to beginning or ending of words
- `\b` anchors boundary (beginning or ending) of words, `\B` anywhere else
- e.g. `[a-z,]`$ matches lines ending in a lower-case letter or comma
- e.g., `\B[A-Z]` matches capital letters not at the beginning or ending of a word
Back-References

- Use \1, \2, etc., to refer to whatever matched the 1st, 2nd, etc. parenthesized sub-expression
- The matching strings are captures, capture-groups or captured strings
- [HT]+ matches any sequence of heads and tails
- ( [HT]+ ) \1 matches any sequence of heads and tails that exactly repeats

Self-Referentially

- Regular expressions are strings
- ., a regexp can be stored in a character variable
- regexps can be built up and changed using string-manipulating functions

Splitting on a Regexp

- **strsplit** will take a regexp as its *split* argument
- Splits a string into new strings at each instance of the regexp, just like it would if *split* were a string

```r
=== Last time:

```r
a12 <- readLines("http://www.stat.cmu.edu/~cshalizi/statcomp/14/lectures/04/al2.txt")
a12 <- paste(a12, collapse=" ")
a12.words1 <- strsplit(a12, split=" ")
```r

```r
=== Weird results (e.g., punctuation marks as parts of wordss)

```r
head(sort(table(a12.words1)))
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
Closer examination shows there’s still a problem: “men’s” → “men”, “s”

Handle possessives: look for any number of white spaces, or at least one punctuation mark followed by at least one space

```r
al2.words3 <- strsplit(al2, split="\s+|[[:punct:]]+\[[:space:]]+")[[1]]
```

grep() and grepl()

grep() scans a character vector for matches to a regexp
returns either indices of matches, or matching strings
grep(x, pattern, value)

Example: scanning data files

ANSS.csv.html catalogs earthquakes of magnitude 6+, 1/1/2002–1/1/2012

```
<HTML><HEAD><TITLE>NCEDC_Search_Results</TITLE></HEAD><BODY>Your search parameters are:<ul>
<li>catalog=ANSS
<li>start_time=2002/01/01,00:00:00
<li>end_time=2012/01/01,00:00:00
<li>minimum_magnitude=6.0
<li>maximum_magnitude=10
<li>event_type=E
</ul>
<pre>
DateTime,Latitude,Longitude,Depth,Magnitude,MagType,NbStations,Gap,Distance,RMS,Source,EventID
2002/01/01 10:39:06.82,-55.2140,-129.0000,10.00,6.00,Mw,78,,,1.07,NEI,2002010140
</pre>
```

Now: extract just the data, not the search parameters and so forth

```
anss <- readLines("http://www.stat.cmu.edu/~cshalizi/statcomp/14/lectures/05/ANSS.csv.html", warn=FALSE)
head(grep(x=anss,pattern="^[0-9]{4}/[0-9]{2}/[0-9]{2}\d{4}\d{6,6}]\d{2,2}))
```

```
## [1] 11 12 13 14 15 16
```

Getting the value of the matches
Storing a regexp in a variable

```r
initial_date <- "^[0-9]{4}/[0-9]{2}/[0-9]{2}\$"
all.equal(grep(x=anss,pattern="^[0-9]{4}/[0-9]{2}/[0-9]{2}\$"),
grep(x=anss,pattern=initial_date))
```

## [1] TRUE

Finding non-matches

The invert option:

```r
grep(x=anss,pattern=initial_date,invert=TRUE,value=TRUE)
```

## [1] "<HTML><HEAD><TITLE>NCEDC_Search_Results</TITLE></HEAD><BODY>Your search parameters are:<ul>
## [2] "<li>catalog=ANSS"
## [3] "<li>start_time=2002/01/01,00:00:00"
## [4] "<li>end_time=2012/01/01,00:00:00"
## [5] "<li>minimum_magnitude=6.0"
## [6] "<li>maximum_magnitude=10"
## [7] "<li>event_type=E"
## [8] "</ul>
## [9] "<pre>
## [10] "DateTime,Latitude,Longitude,Depth,Magnitude,MagType,NbStations,Gap,Distance,RMS,Source,EventID"
## [11] "</pre>"
## [12] "</BODY></HTML>"

```r
grepl()
```

When you just want a Boolean vector saying where the matches are:

```r
grepl(x=anss,pattern=initial_date)[1:20]
```

## [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE...
## [20] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE True
More information about the matches

- `regexpr()` returns location of first match in the target string, plus attributes like length of matching substring
- `gregexpr()` returns a list of this for all matches
- A location of -1 means no match
- Neither returns the text of the match

Getting the matching text

- `regmatches()` takes the output of `regexpr()` or `gregexpr()` and a string, and returns the matching strings
- Why separate `regexpr()` from `regmatches`?
  - Lets us do things like count the number or length of matches with less work
  - Lets us see what text in one file corresponds to matching locations in another file

Example: Extracting earthquake locations

Get the (latitude, longitude) pair for each earthquake:

```r
one_geo_coord <- paste("-?[0-9]+\.[0-9]{4}\")
pair_geo_coords <- paste(rep(one_geo_coord,2),collapse=",")
have_coords <- grepl(x=anss,pattern=pair_geo_coords)
coord.matches <- gregexpr(pattern=pair_geo_coords,text=anss[have_coords])
coords <- regmatches(x=anss[have_coords],m=coord.matches)
```

```r
coord.matches[1]
```

```r
## [[1]]
## [1] 24
## attr("match.length")
## [1] 18
## attr("useBytes")
## [1] TRUE
```

`useBytes`: The default is to assume the ASCII encoding of characters for English, 1 character per byte. Other alphabets need longer encodings and forcing `useBytes=FALSE`

```r
head(coords)
```

```r
## [[1]]
## [1] "-55.2140,-129.0000"
## [2]
```
Earthquake coordinates (cont’d)

You thought we’d forgotten data frames, didn’t you?

```r
coords <- do.call(c,coords) # De-list-ify to vector
coord.pairs <- strsplit(coords,",") # Break apart latitude and longitude
coord.df <- do.call(rbind, coord.pairs) # De-list-ify to array
coord.df <- apply(coord.df,2,as.numeric) # Character to numeric
coord.df <- as.data.frame(coord.df)
colnames(coord.df) <- c("Latitude","Longitude")
```

```r
head(coord.df)
```

```r
## Latitude Longitude
## 1 -55.214 -129.00
## 2  6.303  125.65
## 3 -17.983  178.74
## 4 -17.600  167.86
## 5  36.088   70.69
## 6 -17.664  168.00
```

```r
library(maps)
map("world")
points(x=coord.df$Longitude, y=coord.df$Latitude, pch=19, col="red")
```
Replacements

Assigning to `regmatches()` changes the matched string, just like `substr()`

`sub()` and `gsub()` work like `regexpr()` and `gregexpr()`, but with an extra `replace` argument

`sub()` produces a new string, assigning to `regmatches()` modifies the original one
Really, assigning to `regmatches()` creates a new string, destroys the old one, and assigns the new string the old name

Summary

- Regexps are text patterns built up from strings by alternation and repetition
- Mastering the syntax of regexps lets us scan text for complicated patterns
- Many string-based functions work with regexps as well
- Special functions exist to scan vectors for matches, to extract regexp matches, and to do substitutions