Lecture 5: Regular Expressions

Statistical Computing, 36-350

Wednesday September 23, 2015

In our last thrilling episode

- Characters and strings
- Combining strings, splitting strings
- Basic counting with strings
- We need a way to work with *patterns* of strings

Outline

- 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:

```
strsplit(text, split=",")
```

Split entries in a data file separated by one space:

```
strsplit(text, split=" ")
```

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 unspecified number of spaces:

```
???????
```

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 what is called a **grammar**, with special symbols and rules

Grammar of regular expressions

- Every string is a valid regexp. E.g.,
  - `"fly"` matches end of `"superfly"`, `"why walk when you can fly"`
  - `"fly"` does not match `"time flies like an arrow"`, `"fruit flies like bananas"`, `"a banana flies poorly"`
- OR of two regexps is a regexp, denoted by `|`. E.g.,
  - `"fly|flies"` matches any of the above strings
- Concatenation of two regexps is a regex. E.g.,
  - `"time|fruit fly|flies"` tries to match any combination of the first term and then the second
- Parentheses define groups. E.g.,
  - `"(time|fruit) (fly|flies)"` is the same as above, but perhaps easier to read

`grep()`

`grep()` scans a character vector for matches to a regexp

`grep(pattern, x, value)`

Returns either indices of matches (when `value=FALSE`, the default), or matching strings (when `value=TRUE`)

```r
string.vec = c("time flies when you're having fun in 350",
               "time does not fly in 350, because it's not fun",
               "Flyers suck, Penguins rule")
grep("fly", string.vec) # Default is value=FALSE
```

```
## [1] 2
```

```r
grep("fly", string.vec, value=TRUE)
```

```
## [1] "time does not fly in 350, because it's not fun"
```

```r
grep("fly|flies", string.vec, value=TRUE)
```

```
## [1] "time flies when you're having fun in 350"
## [2] "time does not fly in 350, because it's not fun"
```

```r
string.vec.2 = c("time flies when you're having fun in 350",
                 "fruit flies when you throw it",
                 "a fruit fly is a beautiful creature")
grep("time|fruit fly|flies", string.vec.2, value=TRUE)
```

```
## [1] "time flies when you're having fun in 350"
## [2] "fruit flies when you throw it"
## [3] "a fruit fly is a beautiful creature"
```

```r
grep("(time|fruit) (fly|flies)", string.vec.2, value=TRUE)
```

```
## [1] "time flies when you're having fun in 350"
## [2] "fruit flies when you throw it"
## [3] "a fruit fly is a beautiful creature"
```

Special characters, character classes

- Some characters, like parentheses, are special and aren't interpreted literally
- Another example are square braces, used to indicate ranges. E.g.,
  - `[a-z]` for any lowercase character between `a` and `z`
  - `[A-Z]` for any uppercase character bewteen `A` and `Z`
  - `[0-9]` for any number between `0` and `9`
  - `[a-m]|[N-Z]|[1-5]` for any lowercase character between `a` and `m`, uppercase character between `N` and `Z`, or number between `1` and `5`
  - `[:punct:]` for punctuation marks
  - `[:space:]` for white space characters (including tabs and line breaks)
- Another example is the caret symbol, which negates what comes after it. E.g.,
  - `[^0-9]` for anything but a number between `0` and `9`
  - `[^aeiou]` for anything but a lowercase vowel
- Another example is the period `.`, which stands for any character, no brackets needed

---

```r
string.vec.3 = c("R2D2","r2d2","RJD2","RT85")
grep("[A-Z][0-9]", string.vec.3, value=TRUE)
```

```
## [1] "R2D2" "RJD2" "RT85"
```

```r
grep("[A-Z][0-9][A-Z][0-9]", string.vec.3, value=TRUE)
```

```
## [1] "R2D2"
```

```r
grep("[A-Z|a-z][0-9][A-Z|a-z][0-9]", string.vec.3, value=TRUE)
```

```
## [1] "R2D2" "r2d2"
```

```r
grep("[A-Z][^0-9][^0-9][0-9]", string.vec.3, value=TRUE)
```

```
## [1] "RJD2"
```

Note that this kind of logic is going to get tedious with longer expressions, unless we have a way of specifying that repeated patterns are OK . . .

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,}` means "n or more times"
- `{n,m}` means "between n and m times (inclusive)"

```
string.vec.4 = c("R2D2",
                 "r2d2",
                 "R2D2 was much less annoyting that C3PO")
grep("([A-Z][0-9])+", string.vec.4, value=TRUE)
```

```
## [1] "R2D2"
## [2] "R2D2 was much less annoyting that C3PO"
```

```
grep("([A-Z|a-z][0-9])+", string.vec.4, value=TRUE)
```

```
## [1] "R2D2"
## [2] "r2d2"
## [3] "R2D2 was much less annoyting that C3PO"
```

```
grep("([A-Z|a-z][0-9])+.*C3PO", string.vec.4, value=TRUE)
```

```
## [1] "R2D2 was much less annoyting that C3PO"
```

```
grep("([A-Za-z][0-9])+.*C3PO", string.vec.4, value=TRUE)
```

```
## [1] "R2D2 was much less annoyting that C3PO"
```

Note that we didn't have to explicitly write `|` inside the square bracket to denote the OR

Quantifier scope and anchoring

- By default, quantifiers apply to last character; use parentheses to have it match more

  ```
  grep("ha{2,}", "haha", value=TRUE)
  ```

  ```
  ## character(0)
  ```

```r
grep("(ha){2,}", "haha", value=TRUE)
```

```
## [1] "haha"
```

- $ means a pattern can only match at the end of a line or string
- ^ means (outside of braces) the beginning of a line or string

```r
grep("[a-z]$", "I like lasers", value=TRUE)
```

```
## [1] "I like lasers"
```

```r
grep("[a-z]$", "I like LASERS", value=TRUE)
```

```
## character(0)
```

There is much more

There are many more special characters, rules for anchoring, etc., so if you are interested, go out and read more about regular expressions

WHENEVER I LEARN A NEW SKILL I CONCOCT ELABORATE FANTASY SCENARIOS WHERE IT LETS ME SAVE THE DAY.
OH NO! THE KILLER MUST HAVE FOLLOWED HER ON VACATION!
BUT TO FIND THEM WE'D HAVE TO SEARCH THROUGH 200 MB OF EMAILS LOOKING FOR SOMETHING FORMATTED LIKE AN ADDRESS!
IT'S HOPELESS!
EVERYBODY STAND BACK.
I KNOW REGULAR EXPRESSIONS.
PERL!
TAP TAP

But if not, what we've covered in this lecture should suffice for this course

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 regular string

Abe Lincoln text example from last time:

```r
linc = readLines("http://www.stat.cmu.edu/~ryantibs/statcomp/lectures/lincoln.txt")
linc = paste(linc, collapse=" ")
```

Splitting on pure spaces gives weird results:

```r
linc.words1 = strsplit(linc, split=" ")[[1]]
head(sort(table(linc.words1)))
```

```
## linc.words1
##       -    "the    "Woe absorbs  accept achieve 
##       1       1       1       1       1       1
```

Splitting on any number of spaces or punctuation marks is better:

```r
linc.words2 = strsplit(linc, split="([[:space:]]|[[:punct:]])+")[[1]]
head(sort(table(linc.words2)))
```

```
## linc.words2
## absorbs  accept achieve against  agents     aid 
##       1       1       1       1       1       1
```

```r
head(sort(table(linc.words2), decreasing=TRUE))
```

```
## linc.words2
## the  to and  of  it war 
##  55  26  24  22  12  12
```

Note that R requires double brackets for special character classes like `[[:punct:]]` and `[[:space]]` (these are called POSIXs)

Example: extracting earthquake locations

Catalog of earthquakes of magnitude 6+ between 2002 and 2012 is up at http://www.stat.cmu.edu/~ryantibs/statcomp/lectures/anss.html

```
<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
```

Suppose we want to extract just the data

Notice: every line of data begins with a date, YYYY/MM/DD

```r
anss = readLines("http://www.stat.cmu.edu/~ryantibs/statcomp/lectures/anss.html",
                 warn=FALSE)
date.pattern = "^[0-9]{4}/[0-9]{2}/[0-9]{2}"
head(grep(pattern=date.pattern, x=anss))
```

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

```r
head(grep(pattern=date.pattern, x=anss, value=TRUE))
```

```
## [1] "2002/01/01 10:39:06.82,-55.2140,-129.0000,10.00,6.00,Mw,78,,,1.07,NEI,2002010140"
## [2] "2002/01/01 11:29:22.73,6.3030,125.6500,138.10,6.30,Mw,236,,,0.90,NEI,2002010140"
## [3] "2002/01/02 14:50:33.49,-17.9830,178.7440,665.80,6.20,Mw,215,,,1.08,NEI,2002010240"
## [4] "2002/01/02 17:22:48.76,-17.6000,167.8560,21.00,7.20,Mw,427,,,0.90,NEI,2002010240"
## [5] "2002/01/03 07:05:27.67,36.0880,70.6870,129.30,6.20,Mw,431,,,0.87,NEI,2002010340"
## [6] "2002/01/03 10:17:36.30,-17.6640,168.0040,10.00,6.60,Mw,386,,,1.14,NEI,2002010340"
```

Finding non-matches

Use the `invert` option:

```r
grep(pattern=date.pattern, x=anss, value=TRUE, invert=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>"
```

`grepl()`

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

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

```
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE
## [12]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
```

`regexpr()` and `regmatches()`

- `regexpr()` returns location of first match in the target string, plus attributes like length of matching substring
- A location of `-1` means no match; it does not return the text of the match
- `regmatches()` takes the output of `regexpr()`, and teturns the matching text

```r
date.regexpr = regexpr(pattern=date.pattern, text=anss)
head(regmatches(m=date.regexpr, x=anss))
```

```
## [1] "2002/01/01" "2002/01/01" "2002/01/02" "2002/01/02" "2002/01/03"
## [6] "2002/01/03"
```

```r
head(grep(pattern=date.pattern, x=anss, value=TRUE))
```

```
## [1] "2002/01/01 10:39:06.82,-55.2140,-129.0000,10.00,6.00,Mw,78,,,1.07,NEI,2002010140"
## [2] "2002/01/01 11:29:22.73,6.3030,125.6500,138.10,6.30,Mw,236,,,0.90,NEI,2002010140"
## [3] "2002/01/02 14:50:33.49,-17.9830,178.7440,665.80,6.20,Mw,215,,,1.08,NEI,2002010240"
## [4] "2002/01/02 17:22:48.76,-17.6000,167.8560,21.00,7.20,Mw,427,,,0.90,NEI,2002010240"
## [5] "2002/01/03 07:05:27.67,36.0880,70.6870,129.30,6.20,Mw,431,,,0.87,NEI,2002010340"
## [6] "2002/01/03 10:17:36.30,-17.6640,168.0040,10.00,6.60,Mw,386,,,1.14,NEI,2002010340"
```

Notice the difference?

More complex example: earthquake coordinates

```r
one.geo.pattern = paste("-?[0-9]+\\.[0-9]{4}")
pair.geo.pattern = paste(rep(one.geo.pattern,2), collapse=",")
coords.matches = regexpr(pattern=pair.geo.pattern, text=anss)
coords = regmatches(m=coords.matches,x=anss)
head(coords)
```

```
## [1] "-55.2140,-129.0000" "6.3030,125.6500"    "-17.9830,178.7440"
## [4] "-17.6000,167.8560"  "36.0880,70.6870"    "-17.6640,168.0040"
```

```r
coords.pairs = strsplit(coords,",") # Break apart latitude and longitude
head(coords.pairs)
```

```
## [[1]]
## [1] "-55.2140"  "-129.0000"
##
## [[2]]
## [1] "6.3030"   "125.6500"
```

```
##
## [[3]]
## [1] "-17.9830" "178.7440"
##
## [[4]]
## [1] "-17.6000" "167.8560"
##
## [[5]]
## [1] "36.0880" "70.6870"
##
## [[6]]
## [1] "-17.6640" "168.0040"
```

```
coords.vec = unlist(coords.pairs) # Unlist into a vector
head(coords.vec)
```

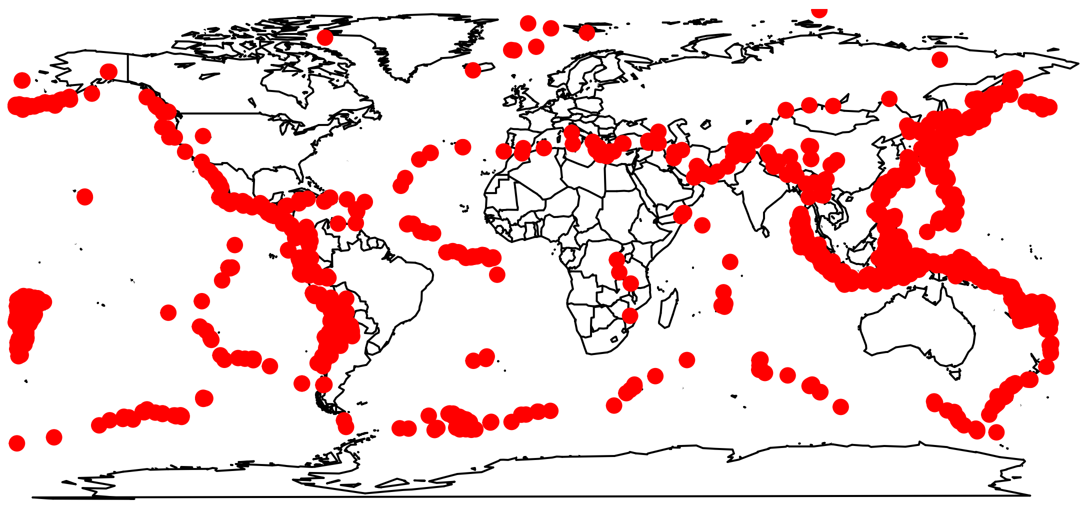
```
## [1] "-55.2140"  "-129.0000" "6.3030"    "125.6500"  "-17.9830"  "178.7440"
```

```
coords.mat = matrix(coords.vec, ncol=2, byrow=TRUE) # Reshape into a matrix
head(coords.mat)
```

```
##      [,1]       [,2]
## [1,] "-55.2140" "-129.0000"
## [2,] "6.3030"   "125.6500"
## [3,] "-17.9830" "178.7440"
## [4,] "-17.6000" "167.8560"
## [5,] "36.0880"  "70.6870"
## [6,] "-17.6640" "168.0040"
```

```
colnames(coords.mat) = c("Latitude","Longitude")
```

```
library(maps)
map("world")
points(x=coords.mat[,"Longitude"], y=coords.mat[,"Latitude"],
       pch=19, col="red")
```



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