

Lecture 5: Character Strings, Regular Expressions, and Web Scraping.

STAT GR5206 *Statistical Computing & Introduction to Data Science*

Gabriel Young
Columbia University

June 2, 2017

Course Notes

- Midterm next Thursday (06/08/2017).
- Midterm details will be discussed next Monday.

Check Yourself (Warm Up)

Tasks: Filtering Data

Use the built-in `iris` dataset:

- How many of the iris are species `versicolor` and have a petal width of less than or equal to 1.2?
- What is the mean petal length of the `setosa` species `iris`? (Do this with filtering and with `tapply()`.)
- Make a table of `iris` species for only those `iris` with sepal width greater than or equal to 3.0.
- Use the `ifelse()` command to create a new variable `Versicolor` that's an indicator variable (1 if the `iris` species is `versicolor` and 0 otherwise). Use the `table()` function to check your result.

Check Yourself (Warm Up)

Tasks: Plotting Data

Use the built-in `iris` dataset:

- Use the `hist()` function to plot a histogram of iris sepal width. Label the axes properly.
- Create a scatterplot of iris sepal width vs. iris sepal length. Color the points by whether or not the species is versicolor.
- Create side-by-side boxplots of iris petal length for each species.

Character Strings and String Operations

Textual Data

Analyzing textual data is common in machine learning and data science.

Textual Data Sources

- Classifying and analyzing tweets from Twitter.
- Answering, does this email belong in the spam filter?
- Processing and comparing survey responses.
- Working with character data such as names, birthdays, or addresses.

Textual Data

Analyzing textual data is common in machine learning and data science.

Textual Data Sources

- Classifying and analyzing tweets from Twitter.
- Answering, does this email belong in the spam filter?
- Processing and comparing survey responses.
- Working with character data such as names, birthdays, or addresses.

R contains many functions for manipulating character data a few of which we study today.

Definition

- A **character** is a symbol in a written language - anything you can enter on a keyboard.
Examples: 'Q', '*', '+', 'd', 'x', ' ', '{' etc.
- A **string** is a sequence of characters.
Examples: 'Columbia University', 'cat, squirrel, hedgehog', etc.

Definition

- A **character** is a symbol in a written language - anything you can enter on a keyboard.

Examples: 'Q', '*', '+', 'd', 'x', ' ', '{' etc.

- A **string** is a sequence of characters.

Examples: 'Columbia University', 'cat, squirrel, hedgehog', etc.

- Both are type character in R.

```
> mode("d")
```

```
[1] "character"
```

```
> mode("cat, squirrel")
```

```
[1] "character"
```

- Both can go into scalars, vectors, matrices, lists, or dataframes.
- In R strings are denoted by quotation marks.

Whitespace

As noted above, whitespace ' ' is considered a character and multiple spaces ' ' a string.

```
> mode(" ")
```

```
[1] "character"
```

```
> nchar(" "); nchar("  "); nchar("")
```

```
[1] 1
```

```
[1] 2
```

```
[1] 0
```

length = length of vector
nchar = length of string

Whitespace

As noted above, whitespace ' ' is considered a character and multiple spaces ' ' a string.

```
> mode(" ")
```

```
[1] "character"
```

```
> nchar(" "); nchar("  "); nchar("")
```

```
[1] 1
```

```
[1] 2
```

```
[1] 0
```

Special Characters

- Quotes within a string: \"
- Tab: \t
- New Line: \n

→ treat as
special character type

Strings as Elements of a Vector

If strings are elements of an object,

- `length()` reports the number of strings in the object, not the number of characters in the string. **Vector**
- `nchar()` reports the number of character values in a string.
- `nchar()` is vectorized, like most R functions.

Strings as Elements of a Vector

```
> length("cat, squirrel, hedgehog")
```

```
| [1] 1
```

```
> length(c("cat", "squirrel", "hedgehog"))
```

①

②

③

```
| [1] 3
```

```
> nchar("cat", "squirrel", "hedgehog") # Not 25
```

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21

↳ not including
Quotes

```
| [1] 23
```

```
> nchar(c("cat", "squirrel", "hedgehog"))
```

```
| [1] 3 8 8
```

Printing Strings

- Can be displayed when their name is typed or using `print()`.
- Often want to use `cat()` to print character strings directly.
- `cat()` coerces its argument to strings, so can be useful when printing warnings.

Printing Strings

- Can be displayed when their name is typed or using `print()`.
- Often want to use `cat()` to print character strings directly.
- `cat()` coerces its argument to strings, so can be useful when printing warnings.

```
> print("cat, squirrel")
```

↳ keeps whole string

```
[1] "cat, squirrel"
```

```
> cat("cat, squirrel")
```

↳ strips off " "

```
cat, squirrel
```

```
> x <- 6
```

```
> y <- 7
```

```
> cat("I have", x, "cats and", y, "hedgehogs as pets.")
```

I have 6 cats and 7 hedgehogs as pets.

You can
change
Quantity w/
Cat

Printing Strings

```
> print("cat, \n squirrel")
```

```
| [1] "cat, \n squirrel"
```

```
> cat("cat, \nsquirrel")
```

```
cat,  
squirrel
```

```
> print("In R, an \"array\" is a multi-dimension matrix.")
```

```
| [1] "In R, an \"array\" is a multi-dimension matrix."
```

```
> cat("A group of hedgehogs is called an \"array\".")
```

```
| A group of hedgehogs is called an "array".
```

} difference when
using "\n"

Check Yourself

Task

Use `print()` and `cat()` to print the following in R:

`"Columbia\tUniversity"`

How many characters are in the above? Why?

Substrings

- A **substring** is a smaller string taken from a larger string, but it is itself still a string.
- Note that we can't use regular subsetting options like `[[]]` or `[]` because a string isn't a vector or list.
- The `substr()` function can extract or change values of parts of strings.

Substrings

The call `substr(string, start = , stop =)` returns the substring from character position `start` to `stop` in the given `string`.

Substrings

inclusive

The call `substr(string, start = , stop =)` returns the substring from character position `start` to `stop` in the given string.

```
> phrase <- "Christmas Bonus"  
> substr(phrase, start = 8, stop = 12)
```

```
[1] "as Bo"
```

```
> substr(phrase, start = 13, stop = 13) <- "g"  
> phrase
```

Insert into string

```
[1] "Christmas Bogus"
```

Substrings

~~vectorized~~ substr() vectorizes

```
> fav_animals <- c("cat", "squirrel", "hedgehog")
> substr(fav_animals, start = 1, stop = 2)
```

```
[1] "ca" "sq" "he"
```

length of character string (4)

```
> substr(fav_animals, nchar(fav_animals)-1,
+         nchar(fav_animals))
```

```
[1] "at" "el" "og"
```

```
> substr(fav_animals, start = 4, stop = 4)
```

```
[1] "" "i" "g"
```

Dividing Strings into Vectors

`strsplit(string, split =)` divides its input string at the appearances of the pattern passed to `split`.

```
> todo <- "Lecture, Lab, Homework"  
> strsplit(todo, split = ",")
```

split a character string however you want

```
[[1]]  
[1] "Lecture"     "Lab"           "Homework"
```

```
> strsplit(todo, split = ", ")
```

becomes vectorized

```
[[1]]  
[1] "Lecture"    "Lab"          "Homework"
```

Dividing Strings into Vectors

`strsplit(string, split =)` divides its input string at the appearances of the pattern passed to `split`.

```
> todo <- "Lecture, Lab, Homework"  
> strsplit(todo, split = ",")
```

```
[[1]]  
[1] "Lecture"     "Lab"           "Homework"
```

```
> strsplit(todo, split = ", ")
```

```
[[1]]  
[1] "Lecture"    "Lab"          "Homework"
```

Note that the output of `strsplit()` is a list. Why?

Dividing Strings into Vectors

```
> todo <- "Lecture, Lab, Homework"  
> strsplit(c(todo, "Midterm, Final"), split = ",")
```

```
[[1]]  
[1] "Lecture"      "Lab"           "Homework"  
  
[[2]]  
[1] "Midterm"      "Final"
```

split by

① > doesn't combine b/c not some logical

②

The pattern is recycled over the elements of an input vector.

Check Yourself

Tasks

- Make a vector of three elements which are “Columbia”, “slumber party”, and “sugarplum”. Make a call to `substr()` that returns the “lum” from each element of the vector. The output should be
“lum” “lum” “lum”
- Use `strsplit()` on the vector you created splitting on “lum”. Output should be a list of length three.

substr → create 2 vectors for starting locations & ending locations
↳ can also be used for logicals!

Building Strings from Multiple Parts

`paste()` combines strings into one long string and is very flexible.

```
> paste("cat", "squirrel", "hedgehog")
```

```
[1] "cat squirrel hedgehog"
```

```
> paste("cat", "squirrel", "hedgehog", sep = " ", ")
```

```
[1] "cat, squirrel, hedgehog"
```

```
> paste(c("cat", "squirrel", "hedgehog"), 1:3)
```

```
[1] "cat 1"      "squirrel 2" "hedgehog 3"
```

```
> paste(c("cat", "squirrel", "hedgehog"), 1:2)
```

```
[1] "cat 1"      "squirrel 2" "hedgehog 1"
```

Building Strings from Multiple Parts

```
> paste(c("cat", "squirrel", "hedgehog"), "(", 1:3, ")")
```

```
[1] "cat ( 1 )"      "squirrel ( 2 )" "hedgehog ( 3 )"
```

```
> paste(c("cat", "squirrel", "hedgehog"), "(", 1:3, ")",
+       sep = "")
```

```
[1] "cat(1)"      "squirrel(2)" "hedgehog(3)"
```

```
> paste(c("cat", "squirrel", "hedgehog"), " (", 1:3, ")",
+       sep = "")
```

```
[1] "cat (1)"      "squirrel (2)" "hedgehog (3)"
```

Exercise

What happens when you pass a vector to the `sep` argument?

Condensing Multiple Strings

The `paste()` function can also condense multiple strings using the `collapse` argument.

```
> paste(c("cat", "squirrel", "hedgehog"), " (", 1:3, ")",
+       sep = "", collapse = "; ")
```

```
| [1] "cat (1); squirrel (2); hedgehog (3)"
```

Check Yourself

Task

Use `paste()` with its first input being `c("Columbia", "slumber party", "sugarplum")` along with the `sep` and `collapse` arguments to create the following string:

 *seperation*

`"Columbia [3-5]; slumber party [2-4]; sugarplum [7-9]"`.

Searching Strings

Honor Code Example

The file "HonorCode.txt" contains Columbia University's Honor Code:

"Students should be aware that academic dishonesty (for example, plagiarism, cheating on an examination, or dishonesty in dealing with a faculty member or other University official) or the threat of violence or harassment are particularly serious offenses and will be dealt with severely under Dean's Discipline..."

Searching Strings

Code example.

Searching Strings

Honor Code Example

```
> HC <- readLines("HonorCode.txt") → outputs a vector of  
> length(HC) character strings
```

```
[1] 43
```

```
> head(HC, 5)
```

```
[1] "Students should be aware that academic dishonesty (for e  
[2] "examination, or dishonesty in dealing with a faculty mem  
[3] "the threat of violence or harassment are particularly se  
[4] "with severely under Dean's Discipline."  
[5] ""
```

HC is a vector with one element per line of text in the Honor Code.

Searching Strings

The `grep(pattern, x)` function searches for a specified substring given by pattern in a vector x of strings.

Honor Code Example

→ case sensitive

```
> grep("students", HC)
```

```
[1] 6 15 23 30
```

→ where you can find "students" (line#)

```
> grep("Students", HC)
```

```
[1] 1 19 33
```

→ logical

```
> head(grep1("students", HC), 15)
```

$\text{vec} \leftarrow \text{c("B", "C", "B")}$

$\text{vec} == "B" \rightarrow \text{c(T, F, T)}$

$\text{which}(\text{vec} == "B") \rightarrow \text{c(1, 3)}$

similar to `which`

```
[1] FALSE FALSE FALSE FALSE FALSE TRUE FALSE FALSE FALSE
```

```
[10] FALSE FALSE FALSE FALSE FALSE TRUE
```

Searching Strings

Honor Code Example

```
> grep("students", HC)
```

```
[1] 6 15 23 30
```

```
> HC[grep("students", HC)]
```

6

- [1] "Graduate students are expected to exhibit the high level
- [2] "In practical terms,¹⁵ students must not cheat on examination
- [3] "Graduate²³ students are responsible for proper citation and
- [4] "All incoming doctoral and master's³⁶ students in the Arts

Searching Strings

Using functions we've learned today, let's make HC a vector with each element a word of the Honor Code (instead of a line of text). Of course, could do this with `scan()`.

Honor Code Example

```
> HC <- paste(HC, collapse = " ") # One long string
> HC.words <- strsplit(HC, split = " ")[[1]] # List output
> head(HC.words, 10)    ↳ produces a list w/ 1 element but you need
                           to extract it in order to do operations
```

[1]	"Students"	"should"	"be"	"aware"
[5]	"that"	"academic"	"dishonesty"	"(for"
[9]	"example,"	"plagiarism,"		

Searching Strings

We can count words using `table()`.

Honor Code Example

```
> word_count <- table(HC.words) → we can see individual words  
> word_count <- sort(word_count, decreasing = TRUE)  
> head(word_count, 10)
```

HC.words

and	of	or	the	to	in	from	a	is
23	17	16	13	11	10	8	6	5

↓

most frequent words that show up in honor code.

Searching Strings

We can count words using `table()`.

Honor Code Example

```
> word_count <- table(HC.words)
> word_count <- sort(word_count, decreasing = TRUE)
> head(word_count, 10)
```

HC.words

and	of	or	the	to	in	from	a	is
23	17	16	13	11	10	8	6	5

This is much more simple than the strategy we used a few weeks ago to produce the same result!

Searching Strings

Honor Code Example

```
> word_count <- table(HC.words)
> word_count <- sort(word_count, decreasing = TRUE)
> head(word_count, 10)
```

HC.words

and	of	or	the	to	in	" "	from	a	is
23	17	16	13	11	10	8	6	5	5



Some undesirable things are happening here...

- The null string "" is the seventh most common word.
- Punctuation and capitalization are messing up our counts.
("students" vs. "Students" before).

Searching Strings

Honor Code Example

```
> head(word_count, 10)
```

HC.words

and	of	or	the	to	in	from	a	is
23	17	16	13	11	10	8	6	5

Some undesirable things are happening here...

```
> tail(word_count, 10)
```

HC.words

vital	which	words	work	work;	works.	world
1	1	1	1	1	1	1
write	Writing	your				
1	1	1				

→ words that do not show up frequently

Check Yourself

Task

Use `grep()` to search over `names(word_count)` to find the number of words in the word count vector that have semi-colons in them. Hint: ";" should be one of your arguments to `grep`. Using your result print the words that `grep()` finds.

Functions for Character Data

Next

We want to search for text patterns instead of text constants. We can do this with regular expressions.

Summary

- `nchar()`: Finds the length of a string.
- `substring()`: Extracts substrings and substitutes.
- `strsplit()`: Turns strings into vectors.
- `paste()`: Turns vectors into a string.
- `grep()`: Search for patterns in a string.

Regular Expressions

Why Do We Need Regular Expressions

```
> fav_animals <- "cat,squirrel, hedgehog, octopus"
```

Why Do We Need Regular Expressions

```
> fav_animals <- "cat,squirrel, hedgehog,   octopus"
```

```
> strsplit(fav_animals, split = ",")
```

} because there
is a space
here
output:

```
[[1]]
```

```
[1] "cat"           "squirrel"      " hedgehog"     "   octopus"
```

Why Do We Need Regular Expressions

```
> fav_animals <- "cat,squirrel, hedgehog, octopus"
```

```
> strsplit(fav_animals, split = ",")
```

```
[[1]]
```

```
[1] "cat"           "squirrel"     " hedgehog"    " octopus"
```

```
> strsplit(fav_animals, split = " ")
```

```
[[1]]
```

```
[1] "cat,squirrel," "hedgehog,"      ""
```

```
[4] ""                  "octopus"
```

> not desirable

Why Do We Need Regular Expressions

```
> fav_animals <- "cat,squirrel, hedgehog, octopus"
```

```
> strsplit(fav_animals, split = ", ")
```

```
[[1]]
```

```
[1] "cat,squirrel" "hedgehog"      "octopus"
```

We need regular expressions to avoid
this type of output

Why Do We Need Regular Expressions

```
> fav_animals <- "cat,squirrel, hedgehog, octopus"
```

```
> strsplit(fav_animals, split = ", ")
```

```
[[1]]
```

```
[1] "cat,squirrel" "hedgehog"      "octopus"
```

Need to split the entries by a comma and *optionally* some space.

Why Do We Need Regular Expressions

It's not just annoying we can't do these things, there are a lot of examples where we have to do such manipulations.

- When scraping data from a webpage, will need to get rid of formatting instructions buried in the source of the webpage.
- Names may be preceded by titles such as Mr., Mrs., or Dr. that we aren't interested in using.

Regular Expressions

- **Regular expressions** are a method of expressing patterns in character strings.
- Used to match sets of strings or patterns of strings in R.
- Can express ideas like match "*this* and then *that*", "either *this* or *that*", "*this* repeated some number of times".
- **Regular expressions** are rules expressed in a grammar with special symbols.

Rules for Regular Expressions

1. Every string is a regular expression. *(technically)*

- "cat" matches "categorize" and "dogs and cats".
- "cat" does not match "Dog is man's best friend" and "work doggedly".



Rules for Regular Expressions

1. Every string is a regular expression.
 - "cat" matches "categorize" and "dogs and cats".
 - "cat" does not match "Dog is man's best friend" and "work doggedly".
2. Can represent OR with a vertical bar |.
 - "cat|dog|Dog" matches all of the above.

Rules for Regular Expressions

1. Every string is a regular expression.
 - "cat" matches "categorize" and "dogs and cats".
 - "cat" does not match "Dog is man's best friend" and "work doggedly".
2. Can represent OR with a vertical bar |.
 - "cat|dog|Dog" matches all of the above.
3. Precede special characters like | with a backslash \ to match exactly.
 - "A\|b" matches "P(A|b)".
 - "A|b" matches twice in "Alabama" and twice in "blueberry".

Rules for Regular Expressions

When I say 'matches', I mean in R:

```
> grep("cat|dog", c("categorize", "work doggedly"))
```

[1] 1 2

matches

matches

```
> grep("A|b", c("Alabama", "blueberry", "work doggedly"))
```

[1] 1 2



not matching 3rd element of vector

Rules for Regular Expressions

1. Indicate sets of characters with brackets [] .

- "[a-z]" matches any lower case letters.
- "[[:punct:]]" matches all punctuation marks.

2. The caret ^ negates a character range when in the leading position.

- "[^aeiou]" matches any characters except lower-case vowels.

3. The period . stands for any character and doesn't need brackets.

- "c..s" matches "cats", "class", "c88s", "c s", etc.

↳ anything in between c & s

Rules for Regular Expressions

Quantifiers can be used to tell “how often” an expression occurs.

Quantifier	Description (Match if the expression is ...)
+	Repeated one or more times.
*	Repeated zero or more times.
?	Repeated zero or one times.
{n}	Repeated exactly n times.
{n, }	Repeated n or more times.
{n, m}	Repeated between n and m times.

Rules for Regular Expressions

Quantifiers can be used to tell “how often” an expression occurs.

Quantifier	Description (Match if the expression is ...)
+	Repeated one or more times.
*	Repeated zero or more times.
?	Repeated zero or one times.
{n}	Repeated exactly n times.
{n, }	Repeated n or more times.
{n, m}	Repeated between n and m times.

Notes

- Quantifiers apply to the last character before they appear.
- Any valid expression can be enclosed in parentheses for grouping.

Rules for Regular Expressions

Quantifiers can be used to tell "how often" an expression occurs.

Quantifier	Description (Match if the expression is ...)
+	Repeated one or more times.
*	Repeated zero or more times.
?	Repeated zero or one times.
{n}	Repeated exactly n times.
{n, }	Repeated n or more times.
{n, m}	Repeated between n and m times.

Examples

- "[0-9] [0-9] [a-zA-Z]+" matches strings with two digits followed by one or more letters.
Annotations: 'digit' with arrows pointing to the first two '[0-9]' sequences; 'letters' with an arrow pointing to the third sequence; 'capital' with an arrow pointing to the 'Z' in 'a-zA-Z'; a red bracket groups the three sequences.
- "(abc){3}" matches three consecutive occurrences of "abc".
Annotation: A red backslash character '\ for special characters' is written above the '{3}'.
- "abc{3}" matches "abccc".
Annotation: Three green arrows point from the 'abc' in the regular expression to the three 'abc' characters in the result string.

Rules for Regular Expressions

Quantifiers can be used to tell "how often" an expression occurs.

Quantifier	Description (Match if the expression is ...)
+	Repeated one or more times.
*	Repeated zero or more times.
?	Repeated zero or one times.
{n}	Repeated exactly n times.
{n, }	Repeated n or more times.
{n, m}	Repeated between n and m times.

Examples *must be 2* maybe a period (but doesn't need to be there).

- "M[rs] [rs]?\.?" matches "Mr", "Ms", "Mrs", "Mr.", "Ms.", "Mrs.". *maybe 3rd letter? (but doesn't need to be there).*
- The above also matches "Mrr", "Msr", "Mss", "Mrr.", "Msr.", "Mss." (and nothing else).

Rules for Regular Expressions

not like matrix subsetting

1. The dollar sign \$ means that a pattern only matches at the end of a line.
 - "[a-z,]\$" matches strings ending in lower-case letters or a comma.
2. The caret ^ outside of brackets means that a pattern only matches at the beginning of a line.
 - "[^A-Z]" matches strings not beginning with capital letters.

beginning different meaning carats

Regular Expressions in R

- Many R functions we've already seen take regular expressions as their arguments.
 - `strsplit()` can use a regular expression to divide a string into a vector.
 - `grep()` can search for patterns represented by regular expressions in a string.

Regular Expressions in R

Honor Code Example

Without regular expressions we get weird results when we try to count words:

```
> head(word_count, 10)
```

HC.words

and	of	or	the	to	in	from	a	is
23	17	16	13	11	10	8	6	5

```
> tail(word_count, 10)
```

HC.words

vital	which	words	work	work;	works.	world
1	1	1	1	1	1	1
write	Writing	your				
1	1	1				

Regular Expressions in R

Honor Code Example

```
> HC <- readLines("HonorCode.txt")
> length(HC)
```

```
[1] 43
```

```
> head(HC, 5)
```

```
[1] "Students should be aware that academic dishonesty (for e
[2] "examination, or dishonesty in dealing with a faculty mem
[3] "the threat of violence or harassment are particularly se
[4] "with severely under Dean's Discipline."
[5] ""
```

HC is a vector with one element per line of text in the Honor Code.

Regular Expressions in R

Honor Code Example

```
> HC <- paste(HC, collapse = " ") # One long string  
> HC.words <- strsplit(HC, split=" ")[[1]] # Last Time always  
> HC.words <- strsplit(HC, split="(\s|[[:punct:]])+" )[[1]] outputs a list  
> head(HC.words, 10)  
[1] "Students"      "should"       "be"           "aware"        " "  
[5] "that"          "academic"     "dishonesty"    "for"          " "  
[9] "example"       "plagiarism"
```

① escape \n any punctuation

② apply special character []

③ []

- Splits at blocks of *only* whitespace and/or punctuation.
- Regular expression is enclosed in quotation marks.
- "\s" is a special character like "\n" or "\t".

↳ space

↳ new line

↳ tab

so you have to extract the elements using [] to operate on them.

Regular Expressions in R

Honor Code Example

In the previous we have the following problem: university's splits to university and s.

Exercise

Check that `split = "\\\s+|([[:punct:]]+|[[:space:]]+)"` gives us what we want: *either* any number of white spaces *or* at least one punctuation mark followed by at least one space.

```
help(regex)
```

Earthquakes Example

- Recall that the `grep()` functions search a character string for a specified pattern.
- Now we can use regular expressions to specify that pattern.
- We're going to practice using some data from the web.

Earthquakes Example

Code Example.

Earthquakes Example

```
> quakes <- readLines("NCEDC_Search_Results.html",
+                         warn = FALSE)
> head(quakes)
```

```
[1] "<HTML><HEAD><TITLE>NCEDC_Search_Results</TITLE></HEAD><BODY>Yo
[2] "<li>catalog=ANSS"
[3] "<li>start_time=2002/01/01,00:00:00"
[4] "<li>end_time=2016/01/01,00:00:00"
[5] "<li>minimum_magnitude=6.0"
[6] "<li>maximum_magnitude=10"
```

We suppress a superfluous warning about end-of-line character in the `readLines()` call.

Earthquakes Example

```
> tail(quakes)
```

```
[1] "2015/12/19 02:10:53.36 -18.3819 169.3857 10.00 6.00  
[2] "2015/12/20 18:47:35.53 3.6384 117.6310 6.93 6.10  
[3] "2015/12/24 19:44:03.13 -55.7550 -123.1158 12.28 6.20  
[4] "2015/12/25 19:14:47.17 36.4872 71.1308 206.00 6.30  
[5] "</PRE>"  
[6] "</BODY></HTML>"
```

Every earthquake of magnitude 6 on the Richter scale from January 1, 2002 until January 1, 2016.

Earthquakes Example

```
> quakes[8:15]
```

```
[1] "</ul>"  
[2] "<PRE>"  
[3] "Date           Time            Lat        Lon    Depth   Mag Magt N  
[4] "-----"  
[5] "2002/01/01 10:39:06.82 -55.2140 -129.0000 10.00  6.00 Mw  2  
[6] "2002/01/01 11:29:22.73   6.3030  125.6500 138.10  6.30 Mw  2  
[7] "2002/01/02 14:50:33.49 -17.9830  178.7440 665.80  6.20 Mw  2  
[8] "2002/01/02 17:22:48.76 -17.6000  167.8560  21.00  7.20 Mw  4
```

Earthquakes Example

```
> quakes[8:15]
```

```
[1] "</ul>"  
[2] "<PRE>"  
[3] "Date           Time           Lat        Lon    Depth   Mag  Magt  N  
[4] "-----"  
[5] "2002/01/01 10:39:06.82 -55.2140 -129.0000 10.00  6.00  Mw  2  
[6] "2002/01/01 11:29:22.73   6.3030  125.6500 138.10  6.30  Mw  2  
[7] "2002/01/02 14:50:33.49 -17.9830  178.7440 665.80  6.20  Mw  2  
[8] "2002/01/02 17:22:48.76 -17.6000  167.8560  21.00  7.20  Mw  4
```

Tasks

- Get rid of the first few lines of HTML formatting code and search parameters.
- Actual data begins on line 12. Headers on line 11.
- Strategy: all data lines begin with a date in format YYYY/MM/DD.

Earthquakes Example

Extracting the Data

```
> date_express <- "[0-9]{4}/[0-9]{2}/[0-9]{2}"  
> head(grep(quakes, pattern = date_express))
```

```
[1] 12 13 14 15 16 17
```

↑
Which line this pattern starts on

any digit 4+
symbol in date
any digit 2+
symbol in date 2+
any digit 2+

Earthquakes Example

Extracting the Data

```
> date_express <- "[0-9]{4}/[0-9]{2}/[0-9]{2}"  
> head(grep(quakes, pattern = date_express))
```

```
[1] 12 13 14 15 16 17
```

```
> head(grep(quakes, pattern = date_express, value = TRUE))
```

```
[1] "2002/01/01 10:39:06.82 -55.2140 -129.0000 10.00 6.00"  
[2] "2002/01/01 11:29:22.73 6.3030 125.6500 138.10 6.30"  
[3] "2002/01/02 14:50:33.49 -17.9830 178.7440 665.80 6.20"  
[4] "2002/01/02 17:22:48.76 -17.6000 167.8560 21.00 7.20"  
[5] "2002/01/03 07:05:27.67 36.0880 70.6870 129.30 6.20"  
[6] "2002/01/03 10:17:36.30 -17.6640 168.0040 10.00 6.60"
```

gives actual lines

Earthquakes Example

What are we leaving behind?

```
> grep(quakes, pattern = date_express,  
+       invert = TRUE, value = TRUE)
```

↳ gives everything except pattern

```
[1] "<HTML><HEAD><TITLE>NCEDC_Search_Results</TITLE></HEAD><BODY>Y  
[2] "<li>catalog=ANSS"  
[3] "<li>start_time=2002/01/01,00:00:00"  
[4] "<li>end_time=2016/01/01,00:00:00"  
[5] "<li>minimum_magnitude=6.0"  
[6] "<li>maximum_magnitude=10"  
[7] "<li>event_type=E"  
[8] "</ul>"  
[9] "<PRE>"  
[10] "Date           Time                 Lat          Lon    Depth     Mag   Magt  
[11] "-----  
[12] "</PRE>"  
[13] "</BODY></HTML>"
```

Check Yourself

Task

- We just extracted the lines we need by noting that they all begin with a date. The lines we need also seem to all end in an event i.d. which is a 12 digit code. Use this idea with grep() to extract the lines of actual data.
- This won't work – we leave behind some data lines. What happened?
- How else could we search for the data using regular expressions?

More Commands in the grep() Family

All return information about where regular expressions are matched *in a string*.

- `grep()` returns a logical indicating a match.
- `regexpr()` returns the location of the first match with attributes like the length of the match.
- `gregexpr()` works similarly to `regexpr()`, but returns *all* matching locations. ‘g’ for global.
- `regmatches()` takes strings and the output of `regexpr()` or `gregexpr()` and returns the actual matching strings.

More Commands in the grep() Family

Examples

```
> # Is there a match?  
> grep("a[a-z]", "Alabama")
```

```
[1] 1
```

```
> # Information about the first match.  
> regexpr("a[a-z]", "Alabama")
```

```
[1] 3  
attr(,"match.length")  
[1] 2  
attr(,"useBytes")  
[1] TRUE
```

More Commands in the grep() Family

Examples

```
> # Information on all matches.  
> gregexpr("a[a-z]", "Alabama")
```

```
[[1]]  
[1] 3 5  
attr(),"match.length")  
[1] 2 2  
attr(),"useBytes")  
[1] TRUE
```

```
> # What are the matches?  
> regmatches("Alabama", gregexpr("a[a-z]", "Alabama"))
```

```
[[1]]  
[1] "ab" "am"
```

Earthquakes Example

Let's Extract the (longitude, latitude) Pairs

```
> quakes[11:15]
```

```
[1] -----  
[2] "2002/01/01 10:39:06.82 -55.2140 -129.0000 10.00 6.00  
[3] "2002/01/01 11:29:22.73 6.3030 125.6500 138.10 6.30  
[4] "2002/01/02 14:50:33.49 -17.9830 178.7440 665.80 6.20  
[5] "2002/01/02 17:22:48.76 -17.6000 167.8560 21.00 7.20
```

```
> coord_exp <- "-?[0-9]+\\. [0-9]{4}"  
> full_exp <- paste(coord_exp, "\\\\s+", coord_exp, sep = "")
```

Earthquakes Example

Let's Extract the (longitude, latitude) Pairs

```
> quakes[11:15]
```

```
[1] -----  
[2] "2002/01/01 10:39:06.82 -55.2140 -129.0000 10.00 6.00  
[3] "2002/01/01 11:29:22.73 6.3030 125.6500 138.10 6.30  
[4] "2002/01/02 14:50:33.49 -17.9830 178.7440 665.80 6.20  
[5] "2002/01/02 17:22:48.76 -17.6000 167.8560 21.00 7.20
```

```
> coord_exp <- "-?[0-9]+\\. [0-9]{4}"  
> full_exp <- paste(coord_exp, "\\\\s+", coord_exp, sep = "")
```

A negative sign zero or one times with digits 0 – 9 one or more times followed by a period and four digits.

Earthquakes Example

```
> head(grepl(quakes, pattern = full_exp), 15)
```

```
[1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  
[10] FALSE FALSE TRUE TRUE TRUE TRUE
```

```
> coord_log <- grepl(quakes, pattern = full_exp)  
> matches <- gregexpr(pattern = full_exp,  
+                           text = quakes[coord_log])  
> head(matches, 1)
```

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

Earthquakes Example

Let's Extract the (longitude, latitude) Pairs

```
> coords <- regmatches(quakes[coord_log], matches)
> head(coords, 4)
```

```
[[1]]
[1] "-55.2140 -129.0000"
```

```
[[2]]
[1] "6.3030 125.6500"
```

```
[[3]]
[1] "-17.9830 178.7440"
```

```
[[4]]
[1] "-17.6000 167.8560"
```

Earthquakes Example

Let's get the data out of a list and put it in a two-column matrix.

Let's Extract the (longitude, latitude) Pairs

```
> coords_split <- sapply(coords, strsplit, split="\s+")
> head(coords_split, 3)
```

```
[[1]]
[1] "-55.2140"   "-129.0000"
```

```
[[2]]
[1] "6.3030"     "125.6500"
```

```
[[3]]
[1] "-17.9830"   "178.7440"
```

Earthquakes Example

Let's get the data out of a list and put it in a two-column matrix.

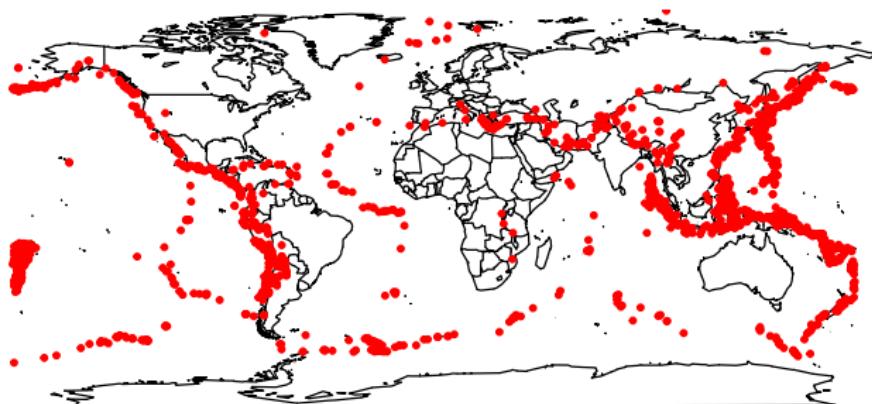
Let's Extract the (longitude, latitude) Pairs

```
> coords_mat <- matrix(unlist(coords_split), ncol = 2,  
+                                byrow = TRUE)  
> colnames(coords_mat) <- c("Latitude", "Longitude")  
> head(coords_mat)
```

	Latitude	Longitude
[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"

Earthquakes Example

```
> library(maps)
> map("world")
> points(coords_mat[, "Longitude"], coords_mat[, "Latitude"],
+         pch = 19, col = "red", cex = .5)
```



What is Web Scraping?

- We've learned about getting data in and out of R when it's structured: `read.table()`, `read.csv()`, etc.
- Often, like the last example, it's not as structured.
 - Could have metadata.
 - Non-tabular arrangement.
- In general this is true of data on the web.

Strategy

Read in line-by-line and split into a nicer format (generally requires a lot of regular expressions).

What is Web Scraping?

- Webpages are generally designed for humans to read.
- Use a computer to extract the information we actually want.
- Iterate the process.

Strategy

Take in unstructured pages, return rigidly-formatted data.

What is Web Scraping?

How do we use the computer extract the information we want?

What is Web Scraping?

How do we use the computer extract the information we want?

- Information is *somewhere* in the page source, usually in the HTML code.
- Often some sort of marker or pointer surrounding the data (again, usually HTML).
- Pick apart the HTML to leave the data using regular expressions.

What is Web Scraping?

How do we pick apart HTML code with regular expressions?

What is Web Scraping?

How do we pick apart HTML code with regular expressions?

- What *exactly* do we want from the page?
- How is the page organized? Where is the information we want located?
 - How does it show up on the webpage?
 - How is that represented in the HTML?
- Write a function to automate the information extraction.
- Now iterate over relevant pages.