Regular Expressions

STAT 133

Gaston Sanchez

`github.com/ucb-stat133/stat133-fall-2016`

Datasets

You'll have some sort of (raw) data to work with

tabular

non-tabular

Data

- Much of the data we deal with are given to us as plain text
- The data are merely represented by their text form
- Sometimes the data are easily interpreted

Toy Data (tabular layout)

name	gender	height
Leia Skywalker	female	1.50
Luke Skywalker	male	1.72
Han Solo	male	1.80

Typically we get data formed of strings and numeric values

Comma Delimited (csv)

```
name,gender,height,weight,jedi,species,weapon
Luke Skywalker,male,1.72,77,jedi,human,lightsaber
Leia Skywalker,female,1.50,49,no_jedi,human,blaster
Obi-Wan Kenobi,male,1.82,77,jedi,human,lightsaber
Han Solo,male,1.80,80,no_jedi,human,blaster
R2-D2,male,0.96,32,no_jedi,droid,unarmed
C-3PO,male,1.67,75,no_jedi,droid,unarmed
Yoda,male,0.66,17,jedi,yoda,lightsaber
Chewbacca,male,2.28,112,no_jedi,wookiee,bowcaster
```

However ...

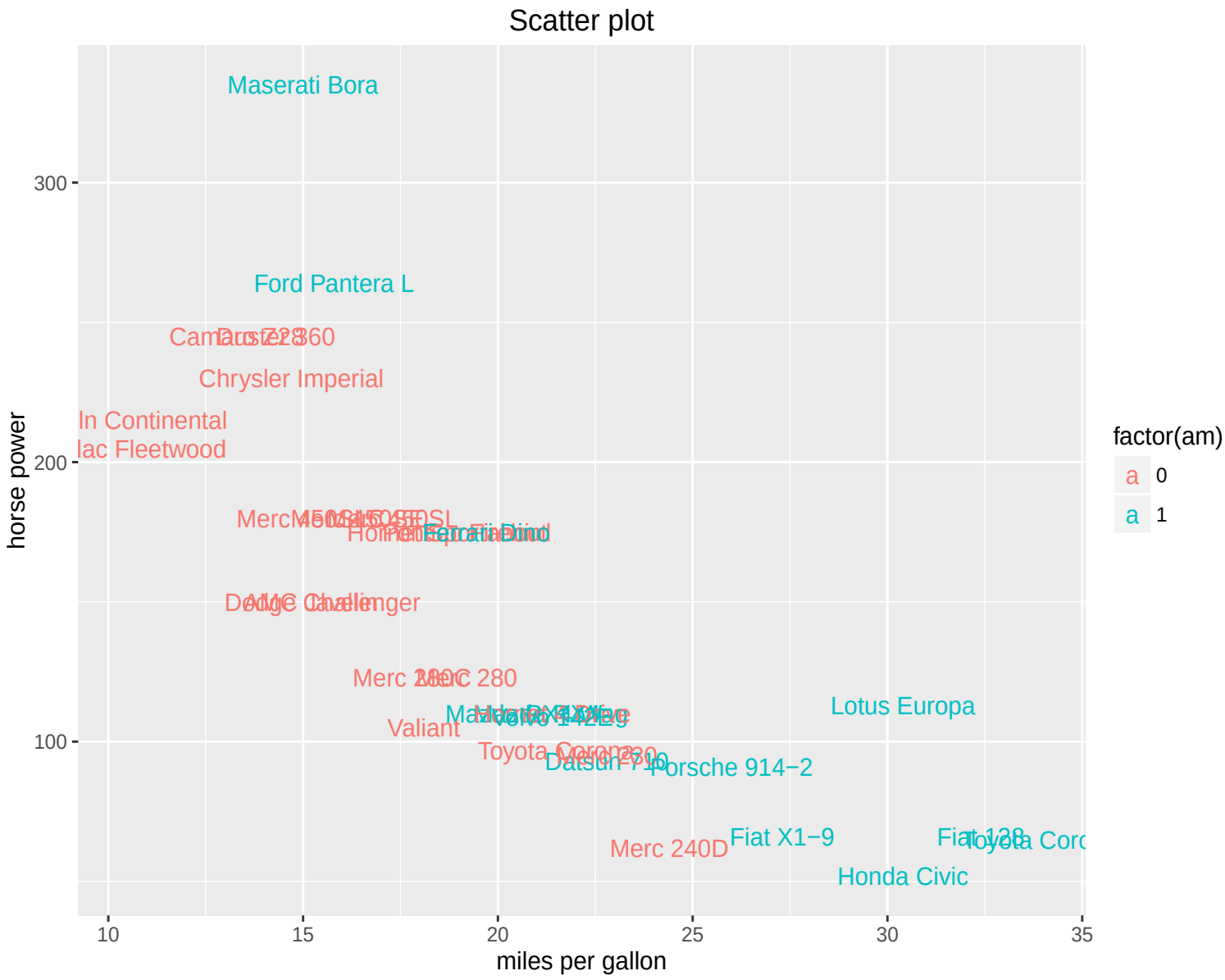
- There are many examples of more complex situations
- It is not uncommon to deal with data that are not as easily interpreted
- And thus the text must be processed to create values of interest

For instance ...

- e.g. when numeric values are embedded into text
- e.g. numeric values not in a regular or simple format
- e.g. numbers in an HTML table
- e.g. data in non-delimited-field formats

Text Everywhere

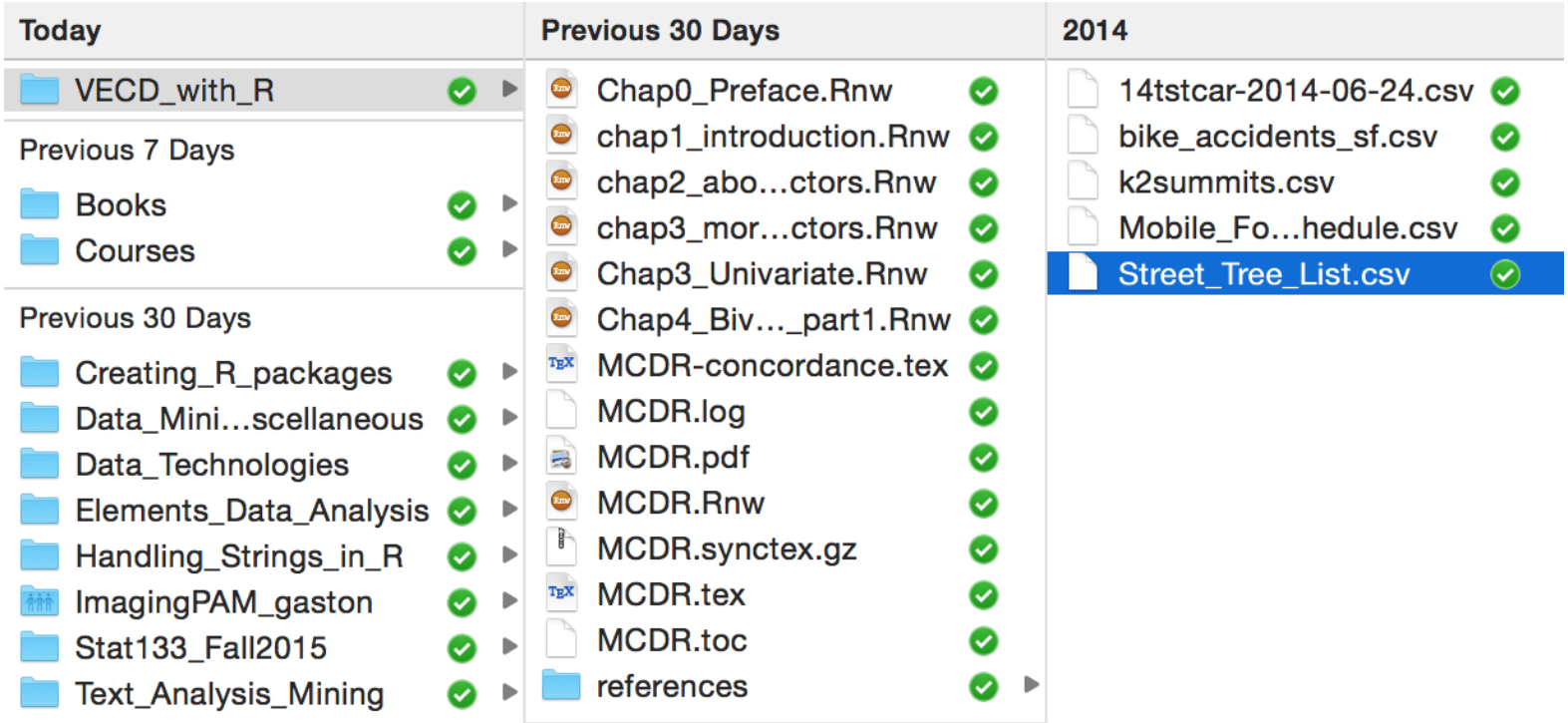
Text in plots



Text in scripts

```
# ====================================================
# Stat133: Lab 2
# Description: Basics of data frames
# Data: Star Wars characters
# ====================================================

# load "readr
library("readr")

# read data using read_csv()
sw <- read_csv("~/stat133/datasets/starwarstoy.csv")

# use str() to get information about the data frame structure
str(sw)

# use summary() to get some descriptive statistics
summary(sw)

# convert column 'gender' as a factor
sw$gender <- factor(sw$gender)
```

Text: names of files and directories



Wikipedia Table

#	Time	Name	Nationality	Date	Meet	Location
1	22:48.4	Henry Taylor	Great Britain	Jul 25, 1908	Olympic Games	London, United Kingdom
2	22:00.0	George Hodgson	Canada	Jul 10, 1912	Olympic Games	Stockholm, Sweden
3	21:35.3	Arne Borg	Sweden	Jul 8, 1923	-	Gothenburg, Sweden
4	21:15.0	Arne Borg	Sweden	Jan 30, 1924	-	Sydney, Australia
5	21:11.4	Arne Borg	Sweden	Jul 13, 1924	-	Paris, France

https://en.wikipedia.org/wiki/World_record_progression_1500_metres_freestyle

Wikipedia Table

```
<table class="wikitable sortable" style="font-size: 95%;">
<tr>
<th>#</th>
<th style="width:4em" class="unsortable">Time</th>
<th class="unsortable"></th>
<th>Name</th>
<th>Nationality</th>
<th>Date</th>
<th>Meet</th>
<th>Location</th>
<th style="width:2em" class="unsortable">Ref</th>
</tr>
<tr>
<td style="text-align:center">1</td>
<td style="text-align:right; padding-left:0.5em; padding-right:0.5em;">22:48.4</td>
<td style="font-size:smaller"></td>
<td><span class="nowrap"><span class="sortkey">Taylor, Henry</span><span class="vcard"><span
class="fn"><a href="/wiki/Henry_Taylor_(swimmer)" title="Henry Taylor (swimmer)">Henry
Taylor</a></span></span></span></td>
<td><span class="flagicon"><img alt=""
src="//upload.wikimedia.org/wikipedia/en/thumb/a/ae/Flag_of_the_United_Kingdom.svg/23px-
Flag_of_the_United_Kingdom.svg.png" width="23" height="12" class="thumbborder"
srcset="//upload.wikimedia.org/wikipedia/en/thumb/a/ae/Flag_of_the_United_Kingdom.svg/35px-
Flag_of_the_United_Kingdom.svg.png 1.5x,
//upload.wikimedia.org/wikipedia/en/thumb/a/ae/Flag_of_the_United_Kingdom.svg/46px-
Flag_of_the_United_Kingdom.svg.png 2x" data-file-width="1200" data-file-height="600" /> 
</span><a href="/wiki/United_Kingdom" title="United Kingdom">Great Britain</a></td>
<td style="text-align:center"><span class="sortkey"
style="display:none;speak:none">000000001908-07-25-0000</span><span style="white-
space:nowrap">Jul 25, 1908</span></td>
<td><a href="/wiki/Swimming_at_the_1908_Summer_Olympics" title="Swimming at the 1908 Summer
```

Example: XML Data

Toy Data (XML format)

```
<subject>
  <name>
    <first>Luke</first>
    <last>Skywalker</last>
  </name>
  <gender>male</gender>
  <height>1.72</height>
</subject>
<subject>
  <name>
    <first>Leia</first>
    <last>Skywalker</last>
  </name>
  <gender>female</gender>
  <height>1.50</height>
</subject>
```

Extracting Data

- Sometimes we must extract the elements of interest from the text content
- The extraction is done by identifying the patterns where the values occur

Extracting Data

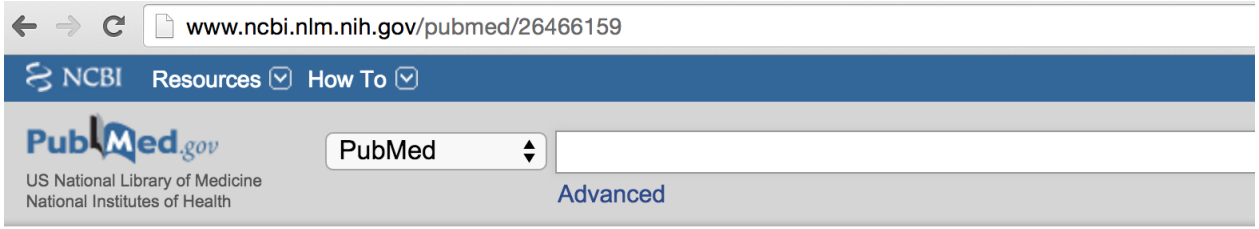
- A different example occurs when text itself makes up the data
- Speech
- Lyrics
- Email messages
- Abstract
- *etc*

Example: Speech

Text of President Barack Obama's State of the Union address, as provided by the White House:

> *Mr. Speaker, Mr. Vice President, members of Congress, distinguished guests and fellow Americans:*
>
> *Last month, I went to Andrews Air Force Base and welcomed home some of our last troops to serve in Iraq. Together, we offered a final, proud salute to the colors under which more than a million of our fellow citizens fought– and several thousand gave their lives.*

Example: Abstract



Abstract

Send to:

PLoS One. 2015 Oct 14;10(10):e0138805. doi: 10.1371/journal.pone.0138805. eCollection 2015.

Quantification of Hydroxylated Polybrominated Diphenyl Ethers (OH-BDEs), Triclosan, and Related Compounds in Freshwater and Coastal Systems.

Kerrigan JF[1], Engstrom DR[2], Yee D[3], Sueper C[4], Erickson PR[5], Grandbois M[6], McNeill K[5], Arnold WA[1].

Author information

Abstract

Hydroxylated polybrominated diphenyl ethers (OH-BDEs) are a new class of contaminants of emerging concern, but the relative roles of natural and anthropogenic sources remain uncertain. Polybrominated diphenyl ethers (PBDEs) are used as brominated flame retardants, and they are a potential source of OH-BDEs via oxidative transformations. OH-BDEs are also natural products in marine systems. In this study, OH-BDEs were measured in water and sediment of freshwater and coastal systems along with the anthropogenic wastewater-marker compound triclosan and its photoproduct dioxin, 2,8-dichlorodibenzo-p-dioxin. The 6-OH-BDE 47 congener and its brominated dioxin (1,3,7-tribromodibenzo-p-dioxin) photoproduct were the only OH-BDE and brominated dioxin detected in surface sediments from San Francisco Bay, the anthropogenically impacted coastal site, where levels increased along a north-south gradient. Triclosan, 6-OH-BDE 47, 6-OH-BDE 90, 6-OH-BDE 99, and (only once) 6'-OH-BDE 100 were detected in two sediment cores from San Francisco Bay. The occurrence of 6-OH-BDE 47 and 1,3,7-tribromodibenzo-p-dioxin sediments in Point Reyes National Seashore, a marine system with limited anthropogenic impact, was generally lower than in San Francisco Bay surface sediments. OH-BDEs were not detected in freshwater lakes. The spatial and temporal trends of triclosan, 2,8-dichlorodibenzo-p-dioxin, OH-BDEs, and brominated dioxins observed in this study suggest that the dominant source of OH-BDEs in these systems is likely natural production, but their occurrence may be enhanced in San Francisco Bay by anthropogenic activities.

Example: Web Log

Web log example

```
123.123.123.123 - - [26/Apr/2000:00:23:48 -0400]
"GET /pics/wpaper.gif HTTP/1.0" 200 6248
"http://www.jafsoft.com/asctortf/"
"Mozilla/4.05 (Macintosh; I; PPC)"
```

```
123.123.123.123 - - [26/Apr/2000:00:23:47 -0400]
"GET /asctortf/ HTTP/1.0" 200 8130
"http://search.netscape.com/Computers/Data_Formats"
"Mozilla/4.05 (Macintosh; I; PPC)"
```

Web log data

- The information in the log has a lot of structure
- e.g. the date always appears in square brackets
- However, the information is not consistently separated by the same characters
- Nor is it placed consistently in the same columns in the file

Web log example

Web log content structure:

```
ppp931.on.bellglobal.com
- -
[26/Apr/2000:00:16:12 -0400]
"GET /download/windows/asctab31.zip HTTP/1.0"
200
1540096
"http://www.htmlgoodies.com/downloads/freeware/15.html"
"Mozilla/4.7 [en]C-SYMPA  (Win95; U)"
```

Web log data

- IP address: `ppp931.on.bellglobal.com`
- Username etc: `"- -"`
- Timestamp: `"[26/Apr/2000:00:16:12 -0400]"`
- Access request:
  `"GET /download/windows/asctab31.zip HTTP/1.0"`
- Result status code: `"200"`
- Bytes transferred: `"1540096"`
- Referrer URL:
  `"http://www.htmlgoodies.com/downloads/freeware/15.html"`
- User Agent: `"Mozilla/4.7 [en]C-SYMPA (Win95; U)"`

Spam Filtering

Anatomy of an email message

- Three parts:
  - header
  - body
  - attachments (optional)
- Like regular mail, the header is the envelope and the body is the letter
- Plain text

Spam Filtering

Email header

- date, sender, and subject
- message id
- who are the carbon-copy recipients
- return path

Example Email Header

```
Date: Mon, 29 Jun 2015 22:16:19 -0800 (PST)
From: doe@email.edu
X-X-Sender: smith@email.net
To: Txxxx Uxxx <txxxx@uclink.berkeley.edu>
Subject: Re: prof: did you receive my hw?
In-Reply-To: <web-569552@calmail-st.berkeley.edu>
MIME-Version: 1.0
Content-Type: TEXT/PLAIN; charset=US-ASCII
Status: O
X-Status:
X-Keywords:
X-UID: 9079
```

Motivation

Motivation

- So far we have seen some basic and intermediate functions for handling and working with text in R
- These functions are very useful functions and they allows us to do many interesting things.
- However, if we truly want to unleash the power of strings manipulation, we need to talk about *regular expressions*.

Motivation

```
##      name   gender   height   weight      born
## 1    Luke     male    1.72m     77gr    19BBY
## 2    Leia   Female    1.50m     49gr    19BBY
## 3   R2-D2     male    0.96m     32gr    33BBY
## 4   C-3PO     MALE    1.67m     75gr   112BBY
```

Motivation

```
##      name   gender   height   weight      born
## 1    Luke     male    1.72m     77gr    19BBY
## 2    Leia   Female    1.50m     49gr    19BBY
## 3   R2-D2     male    0.96m     32gr    33BBY
## 4   C-3PO     MALE    1.67m     75gr   112BBY
```

It is not uncommon to have datasets that need some cleaning and preprocessing

Motivation

Some common tasks

- finding pieces of text or characters that meet a certain pattern
- extracting pieces of text in non-standard formats
- transforming text into a uniform format
- resolving inconsistencies
- substituting certain characters
- splitting a piece of text into various parts

Regular Expressions

Regular Expressions

- A **regular expression** is a special text string for describing a certain amount of text.
- This "certain amount of text" receives the formal name of **pattern**.
- A regular expression is a *pattern that describes a set of strings*.
- It is common to abbreviate the term "regular expression" as **regex**

Regular Expressions

Simply put, working with regular expressions is nothing more than **pattern matching**

Regular Expressions

- Regex patterns consist of a combination of alphanumeric characters as well as special characters.
  - e.g. `[a-zA-Z0-9_.]*`
- A regex pattern can be as simple as a single character
- But it can also be formed by several characters with a more complex structure

About Regex

- "Regular Expressions" is not a full programming language
- Regex have a special syntax and instructions that you must learn
- Regular expressions are supported in a variety of forms on almost every computing platform
- R has functions and packages designed to work with regular expressions

Basic Concepts

Regular expressions are constructed from 3 things:

- **Literal characters** are matched only by the character itself
- A **character class** is matched by any single member of the specified class
- **Modifiers** operate on literal characters, character classes, or combinations of the two.

Literal Characters

Consider the following text:

```
The quick brown fox jumps over the lazy dog
```

A basic regex can be something as simple as `fox`. The characters `fox` match the word "fox" in the sentence above.

Special Characters

Consider this other text:

```
One. Two. Three. And four* and Five!
```

Not all characters are matched literally. There are some characters that have a special meaning in regular expressions: `.` or `*` are some of these special characters.

Special Characters

Metacharacters

- The simplest form of regular expressions are those that match a single character
- Most characters, including all letters and digits, are regular expressions that match themselves
- For example, the pattern `"1"` matches the number 1
- The pattern `"blu"` matches the set of letters “blu”.
- However, there are some special characters that don’t match themselves
- These special characters are known as **metacharacters**.

Metacharacters

The metacharacters in *Extended Regular Expressions* are:

```
.  \  |  (  )  [  {  $  *  +  ?
```

To use a metacharacter symbol as a literal character, we need to **escape** them

Metacharacters and how to escape them in R

Metacharacter	Literal meaning	Escape in R
.	the period or dot	\\.
$	the dollar sign	\\$
*	the asterisk or star	*
+	the plus sign	\\+
?	the question mark	\\?
|	the vertical bar or pipe symbol	\\\|
\	the backslash	\\\\
^	the caret	\\^
[	the opening bracket	\\[
]	the closing bracket	\\]
{	the opening brace	\\{
}	the closing brace	\\}
(	the opening parenthesis	\\(
)	the closing parenthesis	\\)

Character Classes

- A *character class* or *character set* is a list of characters enclosed by square brackets `[ ]`
- Character sets are used to match **only one** of several characters.
- e.g. the regex character class `[aA]` matches any lower case letter a or any upper case letter `A`.
- character classes including the caret `^` at the beginning of the list indicates that the regular expression matches any character **NOT** in the list
- A dash symbol `-` (not at the beginning) is used to indicate ranges: e.g. `[0-9]`

Character Classes

```r
# some string
transport <- c("car", "bike", "plane", "boat")

# look for 'e' or 'i'
grep(pattern = "[ei]", transport, value = TRUE)

## [1] "bike"  "plane"
```

Character Classes

Some (Regex) Character Classes

Anchor	Description
`[aeiou]`	match any one lower case vowel
`[AEIOU]`	match any one upper case vowel
`[0123456789]`	match any digit
`[0-9]`	match any digit (same as previous class)
`[a-z]`	match any lower case ASCII letter
`[A-Z]`	match any upper case ASCII letter
`[a-zA-Z0-9]`	match any of the above classes
`[^aeiou]`	match anything other than a lowercase vowel
`[^0-9]`	match anything other than a digit

POSIX Classes

Closely related to the regex character classes we have what is known as *POSIX character classes*. In R, POSIX character classes are represented with expressions inside double brackets `[[ ]]`.

POSIX Classes

POSIX Character Classes in R

Class	Description
`[[:lower:]]`	Lower-case letters
`[[:upper:]]`	Upper-case letters
`[[:alpha:]]`	Alphabetic characters (`[[:lower:]]` and `[[:upper:]]`)
`[[:digit:]]`	Digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
`[[:alnum:]]`	Alphanumeric characters (`[[:alpha:]]` and `[[:digit:]]`)
`[[:blank:]]`	Blank characters: space and tab
`[[:cntrl:]]`	Control characters
`[[:punct:]]`	Punctuation characters: ! " # % & ' ( ) * + , - . / : ;
`[[:space:]]`	Space characters: tab, newline, vertical tab, form feed, carriage return, and space
`[[:xdigit:]]`	Hexadecimal digits: 0-9 A B C D E F a b c d e f
`[[:print:]]`	Printable characters (`[[:alpha:]]`, `[[:punct:]]` and space)
`[[:graph:]]`	Graphical characters (`[[:alpha:]]` and `[[:punct:]]`)

Special Sequences

Sequences define, no surprinsingly, sequences of characters which can match.

There are shorthand versions (or anchors) for commonly used sequences

Anchor Sequences in R

Anchor	Description
`\\d`	match a digit character
`\\D`	match a non-digit character
`\\s`	match a space character
`\\S`	match a non-space character
`\\w`	match a word character
`\\W`	match a non-word character
`\\b`	match a word boundary
`\\B`	match a non-(word boundary)
`\\h`	match a horizontal space
`\\H`	match a non-horizontal space
`\\v`	match a vertical space
`\\V`	match a non-vertical space

Quantifiers

- Another important set of regex elements are the so-called *quantifiers*.
- These are used when we want to match a **certain number** of characters that meet certain criteria.
- Quantifiers specify how many instances of a character, group, or character class must be present in the input for a match to be found.

Quantifiers

Quantifier	Description
?	The preceding item is optional and will be matched at most once
*	The preceding item will be matched zero or more times
+	The preceding item will be matched one or more times
{n}	The preceding item is matched exactly n times
{n,}	The preceding item is matched n or more times
{n,m}	The preceding item is matched at least n times, but not more than m times

Positions

Character	Description
^	matches the start of the string
$	matches the end of the string
.	matches any single character
|	"OR" operator, matches patterns on either side of the |

Regex Tasks

Common operations

- **Identify** a match to a pattern
- **Locate** a pattern match (positions)
- **Replace** a matched pattern
- **Extract** a matched pattern

Identify a Match

Identify Matches

Functions for identifying match to a pattern:

- `grep(..., value = FALSE)`
- `grepl()`
- `str_detect()` (`"stringr"`)

grep(..., value = FALSE)

```r
# some text
text <- c("one word", "a sentence", "three two one")

# pattern
pat <- "one"

# default usage
grep(pat, text)

## [1] 1 3
```

`grepl()`

- `grepl()` is very similar to `grep()`
- The difference resides in that the output are not numeric indices, but logical
- You can think of `grepl()` as `grep`-*logical*

grepl()

```r
# some text
text <- c("one word", "a sentence", "three two one")

# pattern
pat <- "one"

# default usage
grepl(pat, text)

## [1]  TRUE FALSE  TRUE
```

str_detect() in "stringr"

```
# some text
text <- c("one word", "a sentence", "three two one")

# pattern
pat <- "one"

# default usage
str_detect(text, pat)

## [1]  TRUE FALSE  TRUE
```

Locate a Match

Locate Matches

Functions for locating match to a pattern:

- `regexpr()`
- `gregexpr()`
- `str_locate()` (`"stringr"`)
- `str_locate_all()` (`"stringr"`)

`regexpr()`

- `regexpr()` is used to find exactly where the pattern is found in a given string
- it tells us which elements of the text vector actually contain the regex pattern, and
- identifies the position of the substring that is matched by the regex pattern

regexpr()

```
# some text
text <- c("one word", "a sentence", "three two one")

# default usage
regexpr(pattern = "one", text)

## [1]  1 -1 11
## attr(,"match.length")
## [1]  3 -1  3
## attr(,"useBytes")
## [1] TRUE
```

`gregexpr()`

- `gregexpr()` does practically the same thing as `regexpr()`
- identifies where a pattern is within a string vector, by searching each element separately
- The only difference is that `gregexpr()` has an output in the form of a list

gregexpr()

```r
# some text
text <- c("one word", "a sentence", "three two one")

# default usage
gregexpr(pattern = "one", text)
```

```
## [[1]]
## [1] 1
## attr(,"match.length")
## [1] 3
## attr(,"useBytes")
## [1] TRUE
##
## [[2]]
## [1] -1
## attr(,"match.length")
## [1] -1
## attr(,"useBytes")
## [1] TRUE
##
## [[3]]
## [1] 11
## attr(,"match.length")
## [1] 3
## attr(,"useBytes")
## [1] TRUE
```

`str_locate()` in `"stringr"`

- `str_locate()` locates the position of the first occurrence of a pattern in a string
- it tells us which elements of the text vector actually contain the regex pattern, and
- identifies the position of the substring that is matched by the regex pattern

str_locate() in "stringr"

```
# some text
text <- c("one word", "a sentence", "three two one")

# default usage
str_locate(text, pattern = "one")

##      start end
## [1,]     1   3
## [2,]    NA  NA
## [3,]    11  13
```

`str_locate_all()` in `"stringr"`

- `str_locate_all()` locates the position of ALL the occurrences of a pattern in a string
- it tells us which elements of the text vector actually contain the regex pattern, and
- the output is in the form of a list with as many elements as the number of elements in the examined vector

str_locate_all() in "stringr"

```
# some text
text <- c("one word", "a sentence", "one three two one")

# default usage
str_locate_all(text, pattern = "one")

## [[1]]
##      start end
## [1,]     1   3
##
## [[2]]
##      start end
##
## [[3]]
##      start end
## [1,]     1   3
## [2,]    15  17
```

Extract a Match

Extract Matches

Functions for extracting positions of matched pattern:

- `grep(..., value = TRUE)`
- `str_extract() ("stringr")`
- `str_extract_all() ("stringr")`

Extraction with `grep(..., value = TRUE)`

- `grep(..., value = TRUE)` allows us to do basic extraction
- Actually, it will extract the entire element that matches the pattern
- Sometimes this is not exactly what we want to do (it's better to use functions of `"stringr"`)

grep(..., value = TRUE)

```r
# some text
text <- c("one word", "a sentence", "one three two one")

# extract first one
grep(pattern = "one", text, value = TRUE)

## [1] "one word"           "one three two one"
```

Pattern extraction with `str_extract()`

- `str_extract()` extracts the first occurrence of the matched pattern
- It will only extract the matched pattern
- If no pattern is matched, then a missing value is returned

str_extract() in "stringr"

```
# some text
text <- c("one word", "a sentence", "one three two one")

# extract first one
str_extract(text, pattern = "one")

## [1] "one" NA    "one"
```

Pattern extraction with `str_extract_all()`

- `str_extract_all()` extracts ALL the occurrences of the matched pattern
- It will only extract the matched pattern
- If no pattern is matched, then a character vector of length zero is returned
- the output is in list format

str_extract_all() in "stringr"

```
# some text
text <- c("one word", "a sentence", "one three two one")

# extract all 'one's
str_extract_all(text, pattern = "one")

## [[1]]
## [1] "one"
##
## [[2]]
## character(0)
##
## [[3]]
## [1] "one" "one"
```

Replace a Match

Replace Matches

Functions for replacing a matched pattern:

- `sub()`
- `gsub()`
- `str_replace()` (`"stringr"`)
- `str_replace_all()` (`"stringr"`)

Replace first occurrence with `sub()`

About `sub()`

- `sub()` replaces the **first** occurrence of a pattern in a given text
- if there is more than one occurrence of the pattern in each element of a string vector, only the first one will be replaced.

Replacing with sub()

```
# string
Rstring <- c("The R Foundation",
             "for Statistical Computing",
             "R is FREE software",
             "R is a collaborative project")

# substitute first 'R' with 'RR'
sub("R", "RR", Rstring)
```

```
## [1] "The RR Foundation"             "for Statistical Computing"
## [3] "RR is FREE software"           "RR is a collaborative project"
```

Replace all occurrences with `gsub()`

To replace not only the first pattern occurrence, but **all** of the occurrences we should use `gsub()` (think of it as *general* substitution)

Replacing with gsub()

```r
# string
Rstring <- c("The R Foundation",
             "for Statistical Computing",
             "R is FREE software",
             "R is a collaborative project")

# substitute all 'R' with 'RR'
gsub("R", "RR", Rstring)
```

```
## [1] "The RR Foundation"             "for Statistical Computing"
## [3] "RR is FRREE software"          "RR is a collaborative project"
```

Replacing with str_replace()

```
# string
Rstring <- c("The R Foundation",
             "for Statistical Computing",
             "R is FREE software",
             "R is a collaborative project")

# substitute first 'R' with 'RR'
str_replace(Rstring, "R", "RR")
```

```
## [1] "The RR Foundation"             "for Statistical Computing"
## [3] "RR is FREE software"           "RR is a collaborative project"
```

Replacing with str_replace_all()

```
# string
Rstring <- c("The R Foundation",
             "for Statistical Computing",
             "R is FREE software",
             "R is a collaborative project")

# substitute first 'R' with 'RR'
str_replace_all(Rstring, "R", "RR")
```

```
## [1] "The RR Foundation"             "for Statistical Computing"
## [3] "RR is FRREE software"          "RR is a collaborative project"
```

Split a string

Split a string

Another common task is *splitting* a string based on a pattern. The idea is to split the elements of a character vector into substrings according to regex matches.

Split a string

Functions for extracting positions of matched pattern:

- `strsplit()`
- `str_split() ("stringr")`

Split a string

```
# a sentence
sentence <- c("The quick brown fox")

# split into words
strsplit(sentence, " ")

## [[1]]
## [1] "The"   "quick" "brown" "fox"
```

Split a string

```
# telephone numbers
tels <- c("510-548-2238", "707-231-2440", "650-752-1300")

# split each number into its portions
strsplit(tels, "-")

## [[1]]
## [1] "510"  "548"  "2238"
##
## [[2]]
## [1] "707"  "231"  "2440"
##
## [[3]]
## [1] "650"  "752"  "1300"
```

Split a string

```
# telephone numbers
tels <- c("510-548-2238", "707-231-2440", "650-752-1300")

# split each number into its portions
str_split(tels, "-")

## [[1]]
## [1] "510"  "548"  "2238"
##
## [[2]]
## [1] "707"  "231"  "2440"
##
## [[3]]
## [1] "650"  "752"  "1300"
```