Regular Expressions 1

STAT 133

Gaston Sanchez

Department of Statistics, UC–Berkeley

gastonsanchez.com
github.com/gastonstat/stat133
Course web: gastonsanchez.com/teaching/stat133

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

Text Data:

- speeches
- lyrics
- movie scripts
- html content
- articles
- books
- ETC

State of the Union, Barack Obama 2009

Madam Speaker, Mr. Vice President, Members of Congress, the First Lady of the United States–she's around here somewhere: I have come here tonight not only to address the distinguished men and women in this great Chamber, but to speak frankly and directly to the men and women who sent us here.

I know that for many Americans watching right now, the state of our economy is a concern that rises above all others, and rightly so.

Motivation

State of the Union addresses

- How long are the speeches?
- How do the distributions of certain words change over time?
- How "different" are addresses between democrats and republicans?
- Which presidents have given "similar" speeches?

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

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.

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

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(..., value = TRUE)`
- `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()

```r
# 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"

```r
# 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")`

Pattern 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 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* substition)

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"
```

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.
- For instance, 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

Character Classes

```
# 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

Quantifiers in R

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

Regular Expressions in R

- There are two main aspects that we need to consider about regular expressions in R
- One has to do with *the functions* designed for regex pattern matching
- The other aspect has to do with *the way* regex patterns are expressed in R