Regular Expressions

Getting and Cleaning Data

A string is a sequence of characters, letters, numbers or symbols.

```
stringA <- "This sentence is a string."
objectA <- c( "This sentence is a string.",
              "Here's another string",
              "Here's a third string" )
```

Regular expressions ("regexps") are used to describe patterns within strings.

```r
first_string <- "The quick brown fox jumps over the lazy dog."
second_string <- "There are 8 planets in our solar system:
  Mercury, Venus, Earth, Mars, Jupiter, Saturn, Uranus, and Neptune.
  There used to be a 9th planet: Pluto."
```

```r
str_view_all(string, pattern)
```

```
str_view_all(second_string, "e")
```

```
There are 8 planets in our solar system:
Mercury, Venus, Earth, Mars, Jupiter,
Saturn, Uranus, and Neptune. There used
to be a 9th planet: Pluto.
```

```
str_view_all(second_string, "8")
```

```
There are 8 planets in our solar system:
Mercury, Venus, Earth, Mars, Jupiter,
Saturn, Uranus, and Neptune. There used to
be a 9th planet: Pluto.
```

```
str_view_all(second_string, "2")
```

```
There are 8 planets in our solar system:
Mercury, Venus, Earth, Mars, Jupiter,
Saturn, Uranus, and Neptune. There used to
be a 9th planet: Pluto.
```

```r
# Search for any letter
str_view_all(first_string, "[[:alpha:]]")
```

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

```r
str_view_all(second_string, "[[:alpha:]]")
```

```
There are 8 planets in our solar system:
Mercury, Venus, Earth, Mars, Jupiter,
Saturn, Uranus, and Neptune. There used to
be a 9th planet: Pluto.
```

```r
# Search our string for upper and lower case characters or any number.
str_view_all(first_string, "[[:upper:]]")
```

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

```r
str_view_all(first_string, "[[:lower:]]")
```

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

```r
str_view_all(second_string, "[[:digit:]]")
```

```
    There are 8 planets in our solar system:
    Mercury, Venus, Earth, Mars, Jupiter, Saturn,
    Uranus, and Neptune. There used to be a 9th
    planet: Pluto.
```

```
# Search our string for spaces and/or tabs
str_view_all(first_string, "[[:space:]]")
```

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

```
str_view_all(second_string, "[[:space:]]")
```

```
  There are 8 planets in our solar system: Mercury,
  Venus, Earth, Mars, Jupiter, Saturn, Uranus, and
  Neptune. There used to be a 9th planet: Pluto.
```

```
str_view_all(first_string, "[[:blank:]]")
```

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

```r
# Search our string for punctuation
str_view_all(first_string, "[[:punct:]]")
```

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

```r
str_view_all(second_string, "[[:punct:]]")
```

```
  There are 8 planets in our solar system: Mercury,
  Venus, Earth, Mars, Jupiter, Saturn, Uranus, and
  Neptune. There used to be a 9th planet: Pluto.
```

```r
# Search our string for any character but newline
str_view_all(first_string, ".")
```

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

```r
str_view_all(second_string, ".")
```

```
There are 8 planets in our solar system:
Mercury, Venus, Earth, Mars, Jupiter,
Saturn, Uranus, and Neptune. There used to
be a 9th planet: Pluto.
```

```r
# Search our string for any vowel
str_view_all(first_string, "[aeiou]")
```

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

```
# Search our string for all letters between a and m
str_view_all(second_string, "[a-m]")
```

```
There are 8 planets in our solar system: Mercury,
Venus, Earth, Mars, Jupiter, Saturn, Uranus, and
Neptune. There used to be a 9th planet: Pluto.
```

```
# Search our string for NOT vowels
str_view_all(first_string, "[^aeiou]")
```

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

```r
# Search the first string for NOT vowels OR spaces OR punctuation
str_view_all(first_string, "[^aeiou | [[:space:]] | [[:punct:]] ]")
```

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

```r
names <-c("Keisha McDonald", "Mohammed Smith",
          "Jane Doe", "Mathieu Person")

# Search for strings beginning with M.
str_view_all(names, "^M")
```

```
    Keisha McDonald
    Mohammed Smith
    Jane Doe
    Mathieu Person
```

```r
# Search for periods
str_view_all(second_string, "\\.")
```

```
There are 8 planets in our solar system: Mercury,
Venus, Earth, Mars, Jupiter, Saturn, Uranus, and
Neptune. There used to be a 9th planet: Pluto.
```

```r
# Create a new string with some special characters in it
crazy_string <- "This string is cRaZy!?! :)"

# Search it for !, ?, and )
str_view_all(crazy_string, "\\!")
```

    This string is cRaZy!?! :)

```r
str_view_all(crazy_string, "\\?")
```

    This string is cRaZy!?! :)

```r
str_view_all(crazy_string, "\\)")
```

    This string is cRaZy!?! :)

Quantifiers

- ? : 0 or 1
- + : 1 or more
- * : 0 or more

- {n} : exactly n times
- {n,} : n or more times
- {n,m} : between n and m times

```r
# Identify any time the letter m shows up one or more times
str_view_all(names, "m+")
```

```
  Keisha McDonald
  Mohammed Smith
  Jane Doe
  Mathieu Person
```

```r
# Identify single m's
str_view_all(names, "m{1}")
```

```
  Keisha McDonald
  Mohammed Smith
  Jane Doe
  Mathieu Person
```

```r
# Identify where m shows up exactly two times in a row
str_view_all(names, "m{2}")
```

```
  Keisha McDonald
  Mohammed Smith
  Jane Doe
  Mathieu Person
```

```r
# Identify any time 'mm' shows up one or more times
str_view_all(names, "mm+")
```

```
  Keisha McDonald
  Mohammed Smith
  Jane Doe
  Mathieu Person
```

Summarizing: Regular Expressions

Getting and Cleaning Data

Basic Regular Expressions in R
Cheat Sheet

Character Classes

[[:digit:]] or \\d	Digits; `[0-9]`
\\D	Non-digits; `[^0-9]`
[[:lower:]]	Lower-case letters; `[a-z]`
[[:upper:]]	Upper-case letters; `[A-Z]`
[[:alpha:]]	Alphabetic characters; `[A-z]`
[[:alnum:]]	Alphanumeric characters `[A-z0-9]`
\\w	Word characters; `[A-z0-9_]`
\\W	Non-word characters
[[:xdigit:]] or \\x	Hexadec. digits; `[0-9A-Fa-f]`
[[:blank:]]	Space and tab
[[:space:]] or \\s	Space, tab, vertical tab, newline, form feed, carriage return
\\S	Not space; `[^[:space:]]`
[[:punct:]]	Punctuation characters; `!"#$%&'()*+,-./:;<=>?@[]^_`{|}~`
[[:graph:]]	Graphical characters; `[[:alnum:][:punct:]]`
[[:print:]]	Printable characters; `[[:alnum:][:punct:]\\s]`
[[:cntrl:]] or \\c	Control characters; `\n, \r` etc.

Special Metacharacters

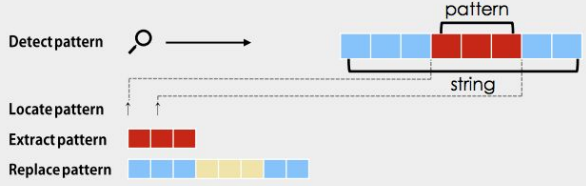
\n	New line
\r	Carriage return
\t	Tab
\v	Vertical tab
\f	Form feed

Lookaraounds and Conditionals*

(?=)	Lookahead (requires PERL = TRUE), e.g. `(?=yx)`: position followed by 'xy'
(?!)	Negative lookahead (PERL = TRUE); position NOT followed by pattern
(?<=)	Lookbehind (PERL = TRUE), e.g. `(?<=yx)`: position following 'xy'
(?<!)	Negative lookbehind (PERL = TRUE); position NOT following pattern
?(if)then	If-then-condition (PERL = TRUE); use lookaheads, optional char. etc in if-clause
?(if)then|else	If-then-else-condition (PERL = TRUE)

*see, e.g. http://www.regular-expressions.info/lookaround.html http://www.regular-expressions.info/conditional.html

Functions for Pattern Matching




```
> string <- c("Hiphopopotamus", "Rhymenoceros", "time for bottomless lyrics")
> pattern <- "t.m"
```

Detect Patterns

grep(pattern, string)
`[1] 1 3`

grep(pattern, string, value = TRUE)
`[1] "Hiphopopotamus"`
`[2] "time for bottomless lyrics"`

grepl(pattern, string)
`[1] TRUE FALSE TRUE`

stringr::str_detect(string, pattern)
`[1] TRUE FALSE TRUE`

Locate Patterns

regexpr(pattern, string)
find starting position and length of first match

gregexpr(pattern, string)
find starting position and length of all matches

stringr::**str_locate(string, pattern)**
find starting and end position of first match

stringr::**str_locate_all(string, pattern)**
find starting and end position of all matches

Split a String using a Pattern

strsplit(string, pattern) or stringr::**str_split(string, pattern)**

Extract Patterns

regmatches(string, regexpr(pattern, string))
extract first match `[1] "tam" "tim"`

regmatches(string, gregexpr(pattern, string))
extract all matches, outputs a list
`[[1]] "tam" [[2]] character(0) [[3]] "tim" "tom"`

stringr::**str_extract(string, pattern)**
extract first match `[1] "tam" NA "tim"`

stringr::**str_extract_all(string, pattern)**
extract all matches, outputs a list

stringr::**str_extract_all(string, pattern, simplify = TRUE)**
extract all matches, outputs a matrix

stringr::**str_match(string, pattern)**
extract first match + individual character groups

stringr::**str_match_all(string, pattern)**
extract all matches + individual character groups

Replace Patterns

sub(pattern, replacement, string)
replace first match

gsub(pattern, replacement, string)
replace all matches

stringr::**str_replace(string, pattern, replacement)**
replace first match

stringr::**str_replace_all(string, pattern, replacement)**
replace all matches

Character Classes and Groups

.	Any character except \n
|	Or, e.g. `(a|b)`
[...]	List permitted characters, e.g. `[abc]`
[a-z]	Specify character ranges
[^...]	List excluded characters
(...)	Grouping, enables back referencing using \\N where N is an integer

Anchors

^	Start of the string
$	End of the string
\\b	Empty string at either edge of a word
\\B	NOT the edge of a word
\\<	Beginning of a word
\\>	End of a word

Quantifiers

*	Matches at least 0 times
+	Matches at least 1 time
?	Matches at most 1 time; optional string
{n}	Matches exactly n times
{n,}	Matches at least n times
{n,m}	Matches between n and m times

General Modes

By default R uses *extended* regular expressions. You can switch to *PCRE regular expressions* using `PERL = TRUE` for base or by wrapping patterns with `perl()` for stringr.

All functions can be used with literal searches using `fixed = TRUE` for base or by wrapping patterns with `fixed()` for stringr.

All base functions can be made case insensitive by specifying `ignore.cases = TRUE`.

Escaping Characters

Metacharacters (`. * +` etc.) can be used as literal characters by escaping them. Characters can be escaped using `\\` or by enclosing them in `\\Q...\\E`.

Case Conversions

Regular expressions can be made case insensitive using `(?i)`. In backreferences, the strings can be converted to lower or upper case using `\\L` or `\\U` (e.g. `\\L\\1`). This requires `PERL = TRUE`.

Greedy Matching

By default the asterisk * is greedy, i.e. it always matches the longest possible string. It can be used in lazy mode by adding `?`, i.e. `*?`.

Greedy mode can be turned off using `(?U)`. This switches the syntax, so that `(?U)a*` is lazy and `(?U)a*?` is greedy.

Note

Regular expressions can conveniently be created using e.g. the packages rex or rebus.

Updated: 10/18

https://github.com/rstudio/cheatsheets/raw/master/regex.pdf

Need to Know

Pattern arguments in stringr are interpreted as regular expressions *after any special characters have been parsed.*

In R, you write regular expressions as *strings*, sequences of characters surrounded by quotes (**""**) or single quotes(**''**).

Some characters cannot be represented directly in an R string . These must be represented as **special characters**, sequences of characters that have a specific meaning., e.g.

Special Character	Represents
\\	\
\"	"
\n	new line

Run **?"'"** to see a complete list

Because of this, whenever a \ appears in a regular expression, you must write it as \\ in the string that represents the regular expression.

Use **writeLines()** to see how R views your string after all special characters have been parsed.

```
writeLines("\\.")
# \.

writeLines("\\ is a backslash")
# \ is a backslash
```

INTERPRETATION

Patterns in stringr are interpreted as regexs To change this default, wrap the pattern in one of:

regex(pattern, ignore_case = FALSE, multiline = FALSE, comments = FALSE, dotall = FALSE, ...) Modifies a regex to ignore cases, match end of lines as well of end of strings, allow R comments within regex's , and/or to have . match everything including \n.
str_detect("i", regex("i", TRUE))

fixed() Matches raw bytes but will miss some characters that can be represented in multiple ways (fast). *str_detect("\u0130", fixed("i"))*

coll() Matches raw bytes and will use locale specific collation rules to recognize characters that can be represented in multiple ways (slow). *str_detect("\u0130", coll("i", TRUE, locale = "tr"))*

boundary() Matches boundaries between characters, line_breaks, sentences, or words. *str_split(sentences, boundary("word"))*

Regular Expressions -

Regular expressions, or *regexps*, are a concise language for describing patterns in strings.

MATCH CHARACTERS

see <- function(rx) str_view_all("abc ABC 123\t.!?\\(){}\n", rx)

string (type this)	regexp (to mean this)	matches (which matches this)	example	
	a (etc.)	a (etc.)	see("a")	abc ABC 123 .!?\(){}
\\.	\.	.	see("\\.")	abc ABC 123 .!?\(){}
\\!	\!	!	see("\\!")	abc ABC 123 .!?\(){}
\\?	\?	?	see("\\?")	abc ABC 123 .!?\(){}
\\\\	\\	\	see("\\\\")	abc ABC 123 .!?\(){}
\\(	\(	(	see("\\(")	abc ABC 123 .!?\(){}
\\)	\)	)	see("\\)")	abc ABC 123 .!?\(){}
\\{	\{	{	see("\\{")	abc ABC 123 .!?\(){}
\\}	\}	}	see("\\}")	abc ABC 123 .!?\(){}
\\n	\n	new line (return)	see("\\n")	abc ABC 123 .!?\(){}
\\t	\t	tab	see("\\t")	abc ABC 123 .!?\(){}
\\s	\s	any whitespace (**\S** *for non-whitespaces*)	see("\\s")	abc ABC 123 .!?\(){}
\\d	\d	any digit (**\D** *for non-digits*)	see("\\d")	abc ABC 123 .!?\(){}
\\w	\w	any word character (**\W** *for non-word chars*)	see("\\w")	abc ABC 123 .!?\(){}
\\b	\b	word boundaries	see("\\b")	abc ABC 123 .!?\(){}
	[:digit:] [1]	digits	see("[:digit:]")	abc ABC 123 .!?\(){}
	[:alpha:] [1]	letters	see("[:alpha:]")	abc ABC 123 .!?\(){}
	[:lower:] [1]	lowercase letters	see("[:lower:]")	abc ABC 123 .!?\(){}
	[:upper:] [1]	uppercase letters	see("[:upper:]")	abc ABC 123 .!?\(){}
	[:alnum:] [1]	letters and numbers	see("[:alnum:]")	abc ABC 123 .!?\(){}
	[:punct:] [1]	punctuation	see("[:punct:]")	abc ABC 123 .!?\(){}
	[:graph:] [1]	letters, numbers, and punctuation	see("[:graph:]")	abc ABC 123 .!?\(){}
	[:space:] [1]	space characters (i.e. \s)	see("[:space:]")	abc ABC 123 .!?\(){}
	[:blank:] [1]	space and tab (but not new line)	see("[:blank:]")	abc ABC 123 .!?\(){}
	.	every character except a new line	see(".")	abc ABC 123 .!?\(){}

[1] Many base R functions require classes to be wrapped in a second set of [ ], e.g. **[[:digit:]]**

[:space:] ↵ new line

[:blank:] space tab

.

[:graph:]

[:punct:]

. , : ; ? ! \ | / ` = * + - ^ _ ~ " ' [ ] { } ( ) < > @ # $

[:alnum:]

[:digit:]

0 1 2 3 4 5 6 7 8 9

[:alpha:]

[:lower:]

a b c d e f g h i j k l m n o p q r s t u v w x z

[:upper:]

A B C D E F G H I J K L M N O P Q R S T U V W X Z

ALTERNATES

alt <- function(rx) str_view_all("abcde", rx)

regexp	matches	example	
ab|d	or	alt("ab|d")	abcde
[abe]	one of	alt("[abe]")	abcde
[^abe]	anything but	alt("[^abe]")	abcde
[a-c]	range	alt("[a-c]")	abcde

ANCHORS

anchor <- function(rx) str_view_all("aaa", rx)

regexp	matches	example	
^a	start of string	anchor("^a")	aaa
a$	end of string	anchor("a$")	aaa

LOOK AROUNDS

look <- function(rx) str_view_all("bacad", rx)

regexp	matches	example	
a(?=c)	followed by	look("a(?=c)")	bacad
a(?!c)	not followed by	look("a(?!c)")	bacad
(?<=b)a	preceded by	look("(?<=b)a")	bacad
(?<!b)a	not preceded by	look("(?<!b)a")	bacad

QUANTIFIERS

quant <- function(rx) str_view_all(".a.aa.aaa", rx)

regexp	matches	example	
a?	zero or one	quant("a?")	.a.aa.aaa
a*	zero or more	quant("a*")	.a.aa.aaa
a+	one or more	quant("a+")	.a.aa.aaa
a{n}	exactly **n**	quant("a{2}")	.a.aa.aaa
a{n, }	**n** or more	quant("a{2,}")	.a.aa.aaa
a{n, m}	between **n** and **m**	quant("a{2,4}")	.a.aa.aaa

GROUPS

ref <- function(rx) str_view_all("abbaab", rx)

Use parentheses to set precedent (order of evaluation) and create groups

regexp	matches	example	
(ab|d)e	sets precedence	alt("(ab|d)e")	abcde

Use an escaped number to refer to and duplicate parentheses groups that occur earlier in a pattern. Refer to each group by its order of appearance

string (type this)	regexp (to mean this)	matches (which matches this)	example (the result is the same as ref("abba"))	
\\1	\1 (etc.)	first () group, etc.	ref("(a)(b)\\2\\1")	abbaab

https://github.com/rstudio/cheatsheets/raw/master/strings.pdf