Ch. 14 – Strings with stringR Package

# **Introduction and Basics:**

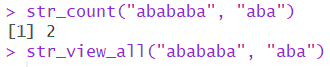
* we have already seen that base R contains many functions to work w/ strings
* however, we try to avoid them because they can be inconsistent and therefore hard to remember
  + instead we focus on **stringr functions**
* **stringr is not part of the tidyverse, because you don’t always have textual data   
  🡪 we need to load it explicitly**
* focus will be on regular expressions (**abbr. regex**):
  + regular expressions are a very powerful tool to check and filter string data and extract information from raw text
  + **regular expressions are a concise langugae for describing patterns** in desirable text and, subsequently, extracting the latter
  + basically, a regular expression is a pattern describing a certain amount of text
  + **the hard part of any regular expression is to find the pattern in a text**
  + once we get it, we can ‘open the door’ and extract as much information from the messy text as we want
  + a “match” is the piece of text, or sequence of bytes or characters, that the pattern was found to correspond to by the regex processing software (in our case R)

# Basics functions applicable to strings (basic str\_\* functions)

|  |  |  |
| --- | --- | --- |
| **Clause** | **Operation/ performance / purpose** | **Example** |
| * str\_\*: there are several stringR functions that can be applied to strings * use autocomplete function to see complete list | | |
| str\_length | * counts the number of characters in a string, incl. spaces * ***base R equivalent: nchar()*** |  |
| str\_c | * base R equivalent: paste(), except that the latter uses ws as default separator and str\_c() does not * to combine two or more character strings  > str\_c(“<string1>”, “<string2>”…) * note I: str\_c is vectorized,   + i.e. it combines each element of the character vector with the other strings   + that does not mean that we can combine the diff. elements of a character **vector itself**      * to collapse a vector of strings into a single string (= a character vector of length 1) w/ a specific separator use collapse = * **I can either insert separating string in one of the character strings or use sep = argument to control how they are separated** * objects of length 0 (e.g. FALSE) are silently dropped * this is especially useful in conjunction w/ if statement * like with most functions in R, NA values are contagious [ansteckend] 🡪 str\_c will result into NA if one entry is NA | * note: stringR combines them w/out inserting ws by default   **str\_c is a vectorized function:**     * note: R automatically recycles shorter vectors (in this case “prefix” and “suffix”) for vectors of greater length (in this case concatenated vector)   **we cannot combine elements of character vector itself with only str\_c**    **collapse= argument**   * **to collapse a vector of strings into a single string/combine elements of a character vector** > str\_c(<vector of strings>, collapse =”<separator>”)   + note: don’t confuse collapse argument with sep argument   + the latter provides only the separator for the combination of different strings, not elements of a character vector     strings  character vector  **sep argument**    **objects of length 0 are silently dropped**      **NAs are contagious** |
| str\_replace\_na | * as already been noted, like with most functions in R, NA values are contagious [ansteckend] * that means that as soon R detects NA, no subsequent operations follow * in order to avoid this we can use  > str\_replace\_na * note: does only work when combining two character vectors, not when combining character strings within one vector | * note: w/out str\_replace\_na, R only combines first element of character vector x |
| str\_sub | * subsetting a character vector * note I: negative index numbers count backwards from end * note II: if end index is out of range, stringR will just return as much as possible | General syntax:  > str\_sub(<string/vector of string>, <start by character index>, <end by character index>)    **works character-wise**    **negative index numbers**    **end index out of rage** |
| str\_to\_lower   * ***base R equivalent: tolower*** | * change text to lower case * is very useful in conjunction w/ str\_sub 🡪 one can change letters of a specific part of the string | * note: we have to assign the lower letters of the first letters, also only to the first letters, not to the whole vector 🡪 otherwise we overwrite the vector with the subset provided by str\_sub: |
| str\_to\_upper  or str\_to\_title   * ***base R equivalent: toupper*** | * change text to upper case * same as w/ str\_to\_lower very useful in conjunction w/ str\_sub |  |
| str\_sort | * sorts strings alphabetically |  |
| str\_dup | * duplicate and concatenate strings within a character vector | > str\_dup(<input character vector>, <times>)   * times: number of times to duplicate each string |

# **Matching strings with regular expression – Basics for regular expressions**

* **regular expressions provide a set of symbols to represent patterns**
* **Learnings:**
* instead of creating one long and complex regular expression, it is better to create a series of simpler ones
* supplementary symbols like +, \* or ? **always refer to the string or symbol before them in the regex**; e.g. .+ means: any character, occurring min. 1 time
* matches never overlap: e.g.





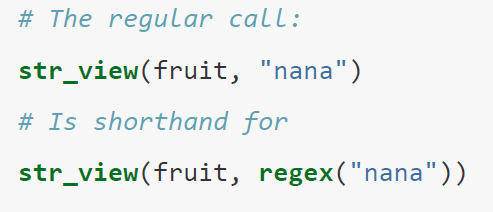
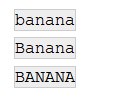
* regex for “any word”: [^ ]+ (i.e. no ws, for min 1 time)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Clause** | | | **Operation / performance /purpose** | **Syntax and examples** | |
| str\_view()  baseR equivalent: | | | **shows** the **first match in viewer** | General syntax:  str\_view(<character vector>,“<regexp>”)    you can see the interim result of the match in the Viewer Pane: | |
| str\_view (<character vector>,“<.regexp.>”) | | | **shows** **all matches in viewer** |  | |
|  | | |  |  | |
| **Symbol** | **String representation** | | |
| **Anchors** | * can be used to “anchor” the regex match at a certain position | | |
| **^ [caret]** | * is used to match the start of a string | | |
| **$** | * is used to match the end of a string | | |
| **combine “^<string>” and “$ <string>”** | * to force a regular expression to only match a specific complete string 🡪 nothing more and nothing less       see in contrast: | | |
| * **note I: ^ and $ avoid partial matching (which is the default of regular expressions otherwise)**   + partial matching means, if any part of the pattern matches the text, the whole text is considered to match the pattern | | | |
| **\b** | * represents and matches word boundaries = the transitions between a word characters (i.e. letters, digits and underscore) and a non-word character (e.g. whitespace) | | |
| **\w** | * represents and matches a **word** **character (i.e. a letter)** | | |
| **\w+** | * symbol represents **and matches one or more consecutive word characters (i.e. a string) [see repititions]** | | |
| **\s** | * symbol represents and matches a **whitespace** character (e.g. space, tab, newline) | | |
| **\d** | * symbol represents and matches any **digit** character | | |
| **\d+** | * symbol represents and matches **one or more consecutive digit characters [see repititions]** | | |
| * note: to create the regular expressions **\\* we have to state \\\* in the string representation** * this is due to \ has already the built-in functionality of being an escape character * hence, we need to “escape the escape” by using \\\* to create \\* as a regular expression | | | |
| **:** | * **symbol represents and matches exactly the symbol occuring after a word** | | |
| **.** | * **represents and matches any character** | | |
| **“\\.”** | * allows one to represent and match . in literal sense in regexp * reason why we have to use double \ is the following:   + in regexp the dot already has a built-in functionality as representing any character   + so we need to escape this functionality by using backlash, same as we do with strings (e.g. \”)   + but exactly the last point poses a problem: \. could be interpreted as a representation of an escape in the string as well; this is exactly how R interprets \.:   + so to not merely represent the escaping in strings, but to actually create an escape for the regular expression (built-in functionality of .), we have to use \\ | | |
| **\\\\** | * allows one to use \ in a literal sense in regexp * argumentation:   + if we would want to use \ in strings, we would code \\, to escape the built-in functionality of \   + furthermore, in regexp, \ is already used to introduce regexp   + if we want to represent \, we actually represent \\, which would make \\\ and to escape the build-in functionality of \ in regexp, we need four times backlashes | | |
| **Character class** | * with a character class, also called a character set, you can tell the regex engine to match only one out of several characters * the characters one wants to be matched are put into square brackets | | |
| **[xyz...]** | * symbol represents and macthes **x, y or z** * can be done with infinite amount of random letters * beachte die Reihenfolge in denen die Buchstaben in der Klammer genannt werden spielt **keine Rolle** | | |
| **[^abc…]** | * ***negated character class*** * symbol represents and matches anything except a, b or c * can be done with infinite amount of random letter   e.g. negated character class to find words containing only consonants:     * note also: we add the + sign here, as R then searches throughout the whole string and does not output TRUE as soon as it has found one match | | |
| **Alternatives to character class and negated character class** |  | | |
| **!** | e.g. find words containing only consonants | | |
| **use of logical operator OR |** | * can be handy if one wants to consider e.g. singular and plurar, British and American English etc. * e.g. abc | d..f will represent and match ‘abc’ **or** a string like ‘dewf’     a more complex example:   * **note: use parentheses to make the OR expression a little clearer to see** | | |
| **Repetitions** | * controlling how many times a pattern matches | | |
| **?** | * pattern matches **0 or 1 times / at most 1 times** * note I: does **“partial matching**”:       **? and [xyz]**       * note II: R takes the first match and stops when condition “pattern matches at most 1 times” is not fulfilled anymore | | |
| **\*** | * pattern matches **at least 0 times** * note: R takes the first match it finds here and stops match search after that   **\* and [xyz]**    🡪 R stops match search after the first C it finds, as the criterion of “pattern matches at least 0 times” is already fulfilled | | |
| **+** | * pattern matches **at least 1 times** * note I: does partial matching:       **+ and [xyz]**           * note II: R outputs every match it finds | | |
| * **note: ?, + and \* are used as supplements after a symbol or string; to make it clearer to which characters they apply use parentheses** * e.g. banan(a)+ | | | |
| **{n}** | * pattern matches exactly n times | | |
| **{n, }** | * pattern matches n or more times | | |
| **{n,m}** | * pattern matches at least n and at most m times | | |
| * by default {n, } and {n,m} will match the longest string possible * one can change this default so that they will match the smallest string possible by adding **? 🡪 R will then only consider the min requirements**       this also holds for **+** | | | |

# Matching strings with regular expressions – Tools

|  |  |  |
| --- | --- | --- |
| **Clause** | **Operation / Performance / Purpose** | **Syntax and Example** |
| str\_detect() | * simply used to detect whether a **character vector** matches a pattern * **returns a logical vector** | > str\_detect(<character vector>, “<regular expression>”)    remember from dplyr: **logical vectors are useful in conjunction with aggregation functions like sum or mean**:    **str\_detect and logical subsetting:** |
| str\_subset() | * alternative to logical subsetting w/ str\_detect | > str\_subset(<input vector>, <regex>)   * input vector: Either a character vector, or something coercible to one * output: character vector |
| str\_detect & filter | * most of the times we will have text in one column of a data frame **(hence, character vector is a column in df)** * thus, **we want to use dplyr’s filter instead** | > df %>% filter(str\_detect(<column in df containing the text>, <regex>) |
| str\_count | * detects **how many** matches there are in a character vector, instead of if it has a match in general (as str\_detect does) | **str\_count vs. str\_detect**    **str\_count and mutate** |
| str\_extract | * extracts **first** matches between pattern and elements of character vector | > str\_extract(<input vector>, <regex>)   * input vector: character vector or vector coercible to character vector * output of str\_extract will be a **character vector** containing all **first** matches * note: str\_extract only extracts the first matches it finds in each character vector element |
| str\_extract\_all | * extracts **all matches** between pattern and elements of character vector | > str\_extract\_all(<input vector>, <regex>, [ggf.] simplify = TRUE)   * input vector: character vector or vector coercible to character vector * output of str\_extract will be a **list** containing **all** matches * note: if one adds simplify = TRUE, R will output a matrix with all matches,  in which it expands short matches to the same length as the longest |
| **Grouped matches** | |  |
| () | * to determine a group | * marking specific groups in regex, allows one to later on to view/subset(extract those parts separately * e.g. regex with 2 groups marked: |
| str\_subset and () |  | * output: character vector containing the whole elements matching both groups |
| str\_extract and () |  | * output: character vector containing only the strings matching both groups; the matches for each character vector element are presented in one element again: |
| str\_match and () | * for grouped matches | * output: matrix containing only the strings matching both groups   + matrix will always contain several columns:     - strings matching all groups as one element,     - + as many more columns as we have groups: containing the string matches with a single group * for more details see baseR character vectors |
| str\_match\_all and () |  | * str\_match only outputs the first matches it finds for each character vector element * if you want to find all matches for each character vector element, use str\_match\_all * **output: list** |
| alternative: tidyr’s extract function | * if data is in tibble it is often easier to use extract() function | * it generally works like str\_match * but it requires to name the matches, which are then placed in new columns of the old tibble * > extract (<data frame>, <col>, <regex>, remove = FALSE) * col: column name(s); if multiple names: concatenate as usual * remove: if remove is = TRUE, the input character vector from which matches are taken, will no longer be part of the new tibble |
| **Replacing matches** | |  |
| str\_replace | * replace matches with ne strings | > str\_replace(<input vector>, <regex>, <replacement>)   * input vector: character vector or vector coercible to character vector * output: character vector * note: will only replace the first match |
| str\_replace\_all |  | > str\_replace\_all(<input vector>, <regex>, <replacement>)   * input vector: character vector or vector coercible to character vector * output: character vector * will replace all matches it finds     **multiple replacements at once by supplying a vector**   * only possible w/ str\_replace\_all * note: all elements of the supplied vector are applied to all elements in original character vector      * here every digit gets replaced by 0, followed by any character being replaced by b; furthermore, every e is replaced by “two”   as usual one can define the replacement vector separately:    **flip the order with \\:**     * **here the order of the second and third match is reversed** |
| **Splitting** | |  |
| str\_split |  | > str\_split(<input vector>, <regex as separator>, n=<number of pieces>, simplify = FALSE)   * input vector: character vector or vector coercible to character vector * n: determines the maximum number of splits per element of character vector   + by default n = infinite * simplify:   + if TRUE: output is a list of character vectors   + if FALSE: output is a character matrix * note: the separator will not be displayed anymore in the output   **with simplify = FALSE vs. simplify = TRUE**    **with maximum number of splits:**     * note: as soon as n is reached elements of character vector are not getting split anymore |
| str\_locate  and  str\_locate\_all | * give starting and ending position (index) of each match * particularly useful, when none of the patterns applied do what one wants | > str\_locate(<input vector>, <regex>)  > str\_locate\_all(<input vector>, <regex>)   * input vector: character vector, or vector coercible to character vector * output:   + for str\_locate: integer matrix   + for str\_locate\_all: list of integer matrices |

# Additional arguments to regex:

* when one uses a regex that is a string, the latter is automatically wrapped into a call to: regex()  
  
* normally, we therefore do not have to state regex as such
* however, regex() functions comes with handy supplements:
  + ignore\_case = TRUE   
    🡪 no case-sensitivity anymore  
     
  + multiline = TRUE  
    🡪 allows ^ and $ to match each of multiple lines







