|  |  |
| --- | --- |
| **Introduction** | |
| Nicht-numerische Vektoren  Character (string) vectors (mainly) with base R | * z.B. character-Vektoren und logical-Vektoren * lassen sich grundsätzlich genauso wie numerische Vektoren ausdrücken und aussprechen   **1. Character Vector:**   * character vectors are used to store text in R * **don’t confuse string w/ character vector:**    + a character vector is a **group/vector of strings**   + a string is a group of 2 or more characters (i.e. single letter or symbol)   + **multiple strings are often stored in a character vector**, **which one can create w/ c()** * in R a string is always introduced and closed by “ or ‘   + in general, the use of “ is recommended   + except, when you want to use “ in the string itself   + when one forgets to close a string, R will display a +      * **R provides a variety of built-in functions to deal w/ character vectors; many of them also perform vectorized operations, i.e. they can process numerous string values in one step**   **Create character vector using “ or ‘**    🡪  note:     * R treats those 2 cases above differently * in the first case, it is treated as one character string * in the second case, it is treated as two separate ones, that are coerced into one character ***vector***   **2. Boolsche Ausdrücke:**  > ‘name des Vektors’ <- c(TRUE, FALSE)  🡪 > ‘name des Vektors’  [1] TRUE FALSE  3. Gemischt:  > ‚name des Vektors‘ <- c("a", 2)  or > ‚name des Vektors‘ <- c("a", 2, TRUE)  🡪 > ‚name‘  [1] "a" "2" "TRUE"  > ‚name‘ <- c(2, TRUE)  🡪 > ‚name‘  [1] 2 1   * **die letzten beiden Beispiele zeigen, dass sich die Typen in Vektoren nicht mischen lassen:**   + also bei characters und anderen Typen: Zuweisung des Typs ‚character‘ auf alle Elemente   + bei numerisch und logical: Zuweisung des Typs numerical auf alle Elemente   🡪 A vector always inherits the “superset” datatype of its components  everything gets coerced to integer  everything gets coerced to double  everything gets coerced to character   * **rep-Funktion** bei nicht-numerischen Vektoren anwenden: |

**Basic operators for strings**

|  |  |  |
| --- | --- | --- |
| **Clause** | **Operational purpose and Performance** | **Syntax and Example** |
| \ | * Backslash is used to start an escape sequence * without further specification, \ allows one to include “ ” or ‘ ’ in a string itself   + ***one escapes the built-in functionality of “ ” resp. ‘ ‘*** as introducing and ending a character string * note: when printing the backlashes will be presented as well   + this is, however, only in the printed version   + beware: **the printed representation of a string is not the same as the string itself** * to see the raw contents of a string, use: writeLines() or cat() |  |
| “ ‘ “ ‘ “ ‘ | * handy alternative for backslash: use either “ ” or ‘ ’ for introducing / ending character string  and use the opposite for inside quote | Note: when using ‘ ’ for character string, R automatically inserts backslashes before “ |
| **for a full list of all \\* commands see: > ?”’”** | | |
| \\ | * if I want to state \ in the string, I have to code \\ |  |
| \n | * inserts new line before character is printed | Syntax: “\n<character>” |
| \t | * inserts tab before character is printed * syntax: | Syntax: “\t<character>” |
| \unnnn | * this is a way of writing non-English characters * the <nnnn> stand for Uni-Code |  |

**Operations with strings**

|  |  |  |
| --- | --- | --- |
| **Clause/function** | **Operational purpose** | **Syntax and Example** |
| **cat()** | * **cat-function** concatenates and **prints** objects * it converts its arguments to character vectors, concatenates [verknüpfen] them to a single vector, appends a given sep= string to each element and outputs it * **note: cat function does not create a character vector w/ indices 🡪 it just creates a message** | **Print message w/ default separator (whitespace)**     * note: * besides strings, pre-specified vectors can be integrated in message * base R appends ws as default separator after each element   **Print message w/ specific separator: add sep= argument**     * most of the time you will write whitespace manually already in the message * to avoid that one has double ws one can specify > sep = “”; then R will only take ws as provided in the character strings   **Use “ or ‘ as characters in message:**   * there are 2 options to escape the built-in functionality of both symbols as marking the beginning and ending of a string   + use \: \ is used to make special characters:   + use ‘ to introduce character vector instead of “:   + other way around: use “ to introduce character vector: * if one wants to use \ use \\ |
| **message ()** | * same as cat-function, but text appears in red and sep = “ “ is not needed to avoid double ws | * message() does not have the default of adding ws as separator |
| **paste()**   * ***stringr equivalent: str\_c(), except that the latter does not use ws as default seperator*** | * concatenates strings to one character vectors * uses ws as default separator * sep = function available * cat vs. paste: * cat only prints strings to console * paste returns character vector for further use * with multi-element character vectors, paste works element-wise | **default:**    **sep =**    **no separator at all with:**    **or**    **cat() vs. paste ()**    **multi-element vectors**    **collapse /concatenate multi-element vectors:**     * collapse = works equal to sep = : it specifies how both character vectors are concatenated        * note: when using special characters like \“ (to create “ separator” or \n (to create new line as separator) we need to use cat (hence, print function for character vectors) to print the string in its intended format   + otherwise, the text representation in the console will contain the special character as we have defined it |
| **tolower**  **toupper**  ***stringR equivalents: str\_to\_lower***  ***str\_to\_upper*** | * changes to small letters / capitals respectively * both functions are vectorized, i.e. providing a character vector, still each vector element will be transformed * functions might be handy in conjunction with created functions 🡪 **one can bypass case-sensitivity** | * note: transformation is done letter-wise     **bypass case-sensitivity using tolower() or toupper**     * we can now provide type in any form written: everything will be transformed to small letters and considered in the function * would have also worked with > toupper; then “type” must be specified with capital letters in the function, e.g. == “ADD” |
| **nchar()** | * ***stringr equivalent: str\_length()*** * **i.e.: number characters** * **function is vectorized** * counts the # of characters of each element of character vector * **nchar() vs. length:** * **nchar() counts # characters per vector element** * **length counts # of vector elements** * nchar() and length() are handy to check validity of provided information, hence is information a valid string | **check validity of strings w/ nchar() in conjunction with function:**       * this function checks whether names provided by students are valid, hence >= letters, before it gets stored in database |
| **trimws()** | * used to delete leading and/or prevailing white spaces * trimws() is overall useful to decrease noise of data and clean it | * > trimws(<Variable>): deletes leading and trailing (folgend) whitespace (default) * > trimws(<Variable>, “left“/“right“): deletes only leading (left) or trailing (right) whitspace     **clean data w/ trimws() in conjunction with function:**     * here we avoid that string gets characterized as “name” just because there are whitespaces  🡪 makes function more robust to noisy data * e.g. “ P” would have nchar() >= 2, just because 2 whitespaces have been added before the character |
| **substr()** | * function is vectorized * likewise, to numeric vectors, strings in a character vector can get subsetted * substr() and nchar() | * character vectr * index of first character to get subsetted * index of last character to get subsetted * note: whitespace counts as element as well     **nchar() and substr():**   * both functions in conjunction can be useful if e.g. character vector elements are of different length (here days are one and 2 digits long)      * subsets from element 5 until the end of each character vector element (calculated with nchar())   **overwrite with substr()** |
| **strsplit()** | * **splits texts by specific separators** * is a vectorized function and the splits each element of character vector separately * function returns a list * **Usability:** * for many variables, the length is not fixed; e.g. names come w/ a variety of length * in this case to use the function substr() and subset by index, is relatively complicated (one solution was to use it in conjunction w/ nchar()) * it is better to: * 1) split the texts and make each part accessible * 2) create a new data frame * 3) extract the necessary columns | General syntax:  > strsplit(<character vector>, **split** = “<separator by which to split>”)     * note again: * strsplit () returns a list * list has as many layers as there were elements within the character vector * there are as many elements per layer as there are elements after the split   **possible command of creating new data frame out of list** (building upon example above)    create transposed data frame; t(<object>) creates matrix, thus, we have to use as.data.frame again    provide suitable column names    provide suitable data type  **easier: use ldply() for creating data frame out of list:**  split whole string into its single characters: split =”” |
| **sprintf()** | * concatenating text with paste() becomes harder to read as the format gets long * sprintf() is clearer in that regard, as the function works with a **template** * this template consists of several placeholders, representing input arguments to appear in the string | **General syntax:**  sprintf(“<template>”, <input arguments>)  **General syntax of template:**  “%<symbols>, …, %<symbols>”  **Building blocks of template:**   * **%d:** placeholder for **digits** * **%s:** placeholder for **strings** * **%f:** placeholder for **double precision value**   **most commonly used syntaxes:**  placeholder exemplary input argument provided    **\***  **\*****\***  **\***  **\* overall number of digits**  **\*\* number of decimal places**  **Example sprintf()**      string as comb. of placeholders and  characters  input arguments  - wenn wir hier nicht „1:“ schreiben nimmt R hier automatisch nur die letzte Zeilennummer   * note: * the whole string consists of a **combination of characters and placeholders** * **input for placeholders has to be provided after the “ and comma** * the sep = “\n” argument refers to cat() function |

# Regular expressions

|  |  |  |
| --- | --- | --- |
| **regular expressions**  **and grep() / str\_matches** | * 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 * **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 * regular expressions solve problems using 2 steps: * 1) find a pattern that matches a text (with grep () [baseR] or str\_match() [stringr]) * 2) group the patterns to extract the information needed (with ())   grep()  str\_match() and grouping() | **Vorab:**   * **if you want to learn more about regular expression see:** <http://regexone.com/> or <http://www.regular-expressions.info/> * to test patterns interactively online, see: <http://www.regexr.com/> * for documentation of regular expressions call: regex() * **regular expressions provide a set of symbols to represent patterns: see list of table regular expressions**   **Supplements to above stated symbols (Ren, p. 199)**    refer to preceding items only (i.e. no stand-alone symbols)   * **note III:** if the pattern is very long it might be more practical to save it as a vector itself:        * **note IV: separators are stated as well** * general process:   (1. read text from file)  2a) use grep() function to find matches  2b) use str\_match, regular expression and () to extract necessary information   * 1. note: grep () and str\_match only take character vectors as input  1. coerce to data frame and assign suitabel names   **1. Read text to character vector:**  **Text in text file**: **read text from text file with readLine()** **(automatically results in character vector)**    **Text in csv file**      **Assign column names so it is easier to refer to it in str\_match**    **2. Find matches**  **2.a grep() function**   * searches for matches to argument pattern (expressed through regular expression) **within each element** **of a character vector**   **Syntax:**  **> grep(pattern, x, ignore.case = FALSE, value = FALSE, invert = FALSE)**   * pattern: pattern to look for; general syntax: **“^**<pattern represented by symbols>**$”** * ignore.case:   + if FALSE, the pattern matching is case sensitive (**default)**   + if TRUE, case is ignored during matching * value:   + if FALSE, a vector containing the (integer) indices of the matches determined by grep is returned **(default)**,   + if TRUE, a character vector containing the matching elements themselves is returned * invert:   + logical: if TRUE return indices or values for elements that do not match   **Example:**  **I. w/out setting “value” to TRUE**  1) create temporary vector with indices provided by grep()    extracting elements of character vector  **🡪** returns indices of matching text in character vector   * **don’t forget “ “**   2) use indices as a result of grep() to extract information **> <df>$<column>[matches]**    **Step 1 and 2a) can also be done in one single step:**     * **Funktion wird von innen nach außen gelöst:**    + **zunächst ergeben sich die Indizes der übereinstimmenden Elemente im Character-Vector,**   + **dann werden sie auf den gleichen Vektor angewendet um sie entsprechend aus dem df zu extrahieren**   **II. w/ setting “value” TRUE**     * **In both options I and II the result is a character vector, that can be coerced to a data frame;** * **however, this data frame will always be of one row only, containing all the information** * **thus, the data frame would need to be separated afterwards**   **3. Coerce to data frame**    **2.b str\_match() and grouping with ()**  Syntax str\_match:  > str\_match(string, pattern)   * string: Input vector. Either a character vector, or something coercible to one * pattern: pattern to look for; form: **“^**<pattern represented by symbols>**$”** * **result: character matrix**   + **first column is the complete match,**   + **followed by one column for each capture group**   **1. find matches and group matches of interest:**  **if we already have character vector:**    **if we have data frame: extract values of row first**  **2. coerce character matrix to data frame and keep only columns needed** |