A

**Array**

Creating an array ***14***

**Definition** ***14***

Erzeugen von Arrays ***14***

Naming rows and columns 15

as.() Function ***8***

Assessing data (elements or subsequences) of an object/Filtering

data frames ***24***

data frames

access values only (double brackets or $) ***26***

lists ***18***

assess value only (double brackets or $) ***19***

matrices ***11***

vectors ***3***

Vectors

Assess values only (double bracket) ***5***

C

c() Function (zusammenfügen von Objekten) ***1***

colMeans() Function

arithmetic mean of each column of a matrix ***13***

**cumsum() Function**

**cum. sum over vector elements** ***8***

D

**Data Frame**

**Definition** ***22***

Erzeugen eines Data Frames ***22***

Hinzufügen einer Reihe (rbind) ***27***

**Hinzufügen einer Spalte (cbind)** ***26***

lesen eines data frames aus einer Datei ***24***

Data Frames

Anzahl Reihen & Spalten in data frame ***22***

diag () Function

show diagonal of a matrix ***14***

diff() Function

difference between elements in a vector ***8***

dim () Function

Dimension of a matrix ***13***

dimnames

Naming rows and columns of an array ***15***

Naming rows and columns of matrix ***10***

drop=FALSE ***12***

duplicates () Functions (detect duplicates) ***8***

F

from to Function (Serie konsekutiver ganzer Zahlen) ***1***

G

General Operations

Operations with basic objects ***1***

**Arrays** ***14***

**Data Frames** ***22***

**Listen** ***17***

**Matrizen** ***9***

Vektoren ***1***

L

length () Function **6**

**Listen (lists)**

**Definition** ***17***

Erzeugen von Listen ***17***

Assign new value(s) to a list element ***20***

convert a list into a vector ***17***

convert a vector to a list ***17***

remove list elements ***21***

M

Making sets ***33***

intersect() ***33***

setdiff() 33

union() ***33***

**Matrizen (matrices)**

**Definition** ***9***

Erzeugen von Matrizen ***9***

Naming rows and columns ***10***

Nützliche Funktionen für Matrizen ***13***

Rechnen mit Matrizen ***11***

N

ncol (# col. in data frame) ***22***

ncol() Function

Number of columns in a matrix ***13***

nrow (# rows in data frame) ***22***

nrow() Function

Number of rows in a matrix ***13***

numeric() Function (Nullvektor) ***2***

P

prod() Function

product over elements in a vector ***8***

R

rep () Function (wdh. von x n-mal) ***1***, ***3***

rev() Function

Umkehren der Zahlenreihenfolge eines Vektors ***8***

rownames() and columnnames() Function ***10***

**rowSums() Function**

**Sum of each row of a matrix** ***13***

S

seq () & length.out() Function (Zahlenreihe best. Länge, gleicher Abstand) ***2***

seq () Function (Zahlenreihe nach best. Intervall) ***2***

solve() Function

Lösung der Matrixgleichung A\*x=b ***13***

**sort () Function** ***7***

ascending order ***7***

descending order ***7***

**sort () function & rules** ***8***

sum () Function

sum over vector elements ***8***

T

t() Function

Transpose a matrix ***13***

Transpose a vector ***8***

V

Vektoren (vectors)

Definition ***1***

Erzeugen von Vektoren ***1***

nicht-numerische

Booleschen Ausdrücke ***2***

Character Strings ***2***

rep() Function ***3***

numerische ***1***

c() Function (concatenate) 1

from to Function 1

numeric() Function (Nullvektor) 2

rep() Function ***1***

seq () Function ***2***

seq() Function

seq () Function & length out () Function 2

Konvertierung von Vektoren in anderen Datentyp (converting, also called coercion) ***8***

length() Function ***6***

Logische Operationen mit Vektoren ***6***

Löschen & Ergänzen von Vektorelementen ***6***

Rechnen mit Vektoren ***7***

Präfix-Form ***7***

Transponieren ***8***

Überschreiben von Vektorelementen ***5***

Weitere nützliche Funktionen ***7***

|  |  |
| --- | --- |
| **Operationen mit Basisobjekten** |  |
| **Learnings** | * always keep in mind that Vectors are basically the building blocks of every following R object   + thereby, a lot of functions that are applicable to vectors, might be applicable to matrices and further on applicable to data frames   + think of it more in sense of the snowball-effect   + for example:   # of rows in a matrix: >nrow(<MATRIX>) 🡪 # of rows in a data frame: >nrow(<DATAFRAME>) |
| **Vectors** | * Fast alle Objekte in R werden intern durch Vektoren repräsentiert * Vectors are **one of the building blocks of all R objects 🡪** the other R objects are built upon vectors * Vectors store data in one dimension; it is just a row of values of the **same atomic datatype** * **Variablen im Sinne der Statistik werden in R als Vektoren dargestellt** * für die einzelnen Elemente werden in Übereinstimmung mit ihrer Position im Vektor **ganzzahlige Indizes (integer indices)** hinterlegt, **beginnend mit 1**   + **sie können so auch angesprochen werden** * in R sind auch **einfache Zahlen Skalare** (Vektoren der Länge 1) und werden entsprechend so behandelt (siehe ‚Abfrage von Elementen eines Vektors‘) |
| Erzeugen von Vektoren | * allg. ist c() eine wichtige Funktion   + c steht für **„concatenate“** (zusammenfügen)   + it is a generic function combining its arguments to form a vector   + note: all arguments are coerced to a common type which is the type of the returned value, and **all attributes except names are removed** |
| Numerische Vektoren | **Numerische Vektoren (numeric)/ganzzahlige Vektoren (integer):**   * > ‚Name des Vektors‘<- c(‚Object1‘, ‚Object2‘,...)   + z.B.: 🡪 * > c(x,y) mit x und y als Vektoren # verbindet 2 bereits erstellte Vektoren   + z.B.: * weitere mögliche Funktionen: * > rep (‘x’, ‘times n’) #wdh. von x n-mal      * > ‚from‘:‘to‘ # erstellt Serie konsekutiver, ganzer Zahlen   + > 1:3  # ergibt 1,2,3   + > 3:1  # ergibt 3,2,1   + note: > 1 + 1:3  # ergibt 2,3,4  🡪 : hat höhere Priorität als + 🡪 es wird erst Vektor erstellt und dann 1 auf jedes Element addiert * **Verknüpfung from:to & rep:**      * + with each-argument:      * > seq(‚from‘, ‚to‘, ‚by‘= (-)‘Intervall‘)      * + note: by = argument is default and therefore, can be left out      * **Verknüpfung von rep & seq:**      * **Verknüpfung seq & length.out**      * > numeric(‚length‘): erstellt einen Null-Vektor von bestimmter Länge |
| Selecting and filtering vectors | **Assessing (i.e. selecting and filtering) elements/subsequences of a vector/array/matrix/list with  1) integer Index/Indizes  2) rules/relational operators  3) boolean variables/logical vectors 4) characters**  **1. Vectors:**  **Summary:**  x[2], x[-2], x[c(2,3,4)], x[x\*2>10], x[‘a‘], x[[‘a‘]]  **1.1 Subset to a new vector** 1) Selecting with **integer Index/Indices:** <Vektorname>[<Indizes>]   * for example:     negative index excludes the respective value  „gebe den Wert des Vektors wieder, für den gilt, dass er an 2. Stelle im Vektor auftritt“   * define “Index”-variable to use for filtering:     2) Filtering with **rules/relational operators**:  <Vectorname>[<Vectorname><Rule>]   * for example: * relational expression results logical vector, that R then uses to filter out values for which vector has value TRUE * **beachte: innerhalb der Regeln muss nochmals auf den Vektor bezuggenommen werden**; ansonsten ist die Regel nicht eindeutig definiert   3) Select with **boolean variables/logical vectors**:   * for example:     **4) Select with character strings/vector for named vectors: <name of vector>[”<name of element>”]**   * for example:       **1.2 Assess values only:**  [[„<name of element>“]]   * for example:   **# note that [[]] can only extract one value of a vector element at a time**    **1.3 Definieren des subset-/filter-Vektors als eigenständiger, neuer Vektor:**   * mit [] (behält Namen einzelner Elemente des Vektors bei):      * mit [[]] (Namen werden nicht beibehalten) and only one element: |
| Überschreiben von Vektorelementen | * ein Objekt kann ganz einfach überschrieben werden, wenn dem Objektnamen einfach ein anderes Element zugewiesen wird   z.B.:  > objekt1 <- 1.043  🡪 > objekt1 <- 3 # überschreibt die erste Definition von 1.043 auf 3   * **Mit Hilfe des Index**, z.B. bei Vektoren:   > x[2] <- 3 # überschreibt zweites Element des Vektors auf 3 > x[2:4] <- c(1,2,3) # überschreibt Elemente 2-4   * **Mit Hilfe von character strings**: * we can use **Boolean Variables** to overwrite as well: > x[c(TRUE, false, true, false)] <- c(2,3) 🡪 # ersetzt den ersten und dritten Wert * auch auf das Überschreiben können **Regeln** angewendet werden:        * **beachte**: auch, wenn Vektoren bzw. Objekten Namen gegeben wurden, werden stets die dahinterstehenden Werte, nicht aber die Namen überschrieben! |
| Löschen & Ergänzen von Vektorelementen | **Löschen von Vektorelementen: - und <-**    **Alternativ:**  Länge des Vektors festsetzen = Elemente löschen  > length(‚Name d. Vektors‘) = ‚desired length‘    **Ergänzen von Vektorelementen**   * > ‘name of vector’ [‘Indexnummer an dessen Stelle Element hinzuge. werden soll‘] = ‚Wert‘ * for example:      * note: sofern Index gewählt wird der eine ‚Lücke‘ zu dem bisherigem höchsten Index darstellt, wird der Rest mit NA aufgefüllt: |
| Relationale Operationen mit Vektoren | Two vectors can be easily compared:   * Comparison is done element-wise * Shorter vector is repeated until length fits * for example:             works with character vectors equivalently: |
| Arithmetische Operatoren und Vektoren | * bei Vektoren wirken arith. Operatoren **auf jede Komponente der Vektoren** * z.B. für Vektoren x, y ist > x\*y ein Vektor, der **komponentenweise** das Produkt enthält   Beispiel:  > x=c(1,2,3)  > y=rep(2,3)  > x\*y  [1] 2 4 6        Rest  ganzzahlige Division     * scheinbar ist das Layout des „linken“ Vektors maßgeblich * Operatoren können auch in ***Präfix-Form*** aufgerufen werden   z.B. >“+“(x, y); note: hier kann sowohl „ als auch ‚ Apostroph verwendet werden     * in Situationen, in denen Operanden nicht die gleiche Länge haben, wird der kürzere zyklisch wiederholt * z.B. |
| Weitere nützliche Funktionen auf Vektoren | **Sort: sort()**  > sort (<Name d. Vektors>, decreasing = TRUE): descending order  > sort (<Name d. Vektors>): ascending order (by default)      **Sortieren & relationale Operatoren/Regeln:** |
| **Remove duplicates:**  > unique (<Name d. Vektors>) |
| **Detect duplicates:**  > duplicates(<Name d. Vektors>): results in logical vector |
| **Sum over elements in a vector:**  > sum(<Name d. Vektors>) |
| **Cumulative sum over elements in a vector:**  > cumsum(<Name d. Vektors>) |
| **Revert (umkehren):**  > rev(<Name d. Vektors>) |
| **Product over elements in a vector:**  > prod(<Name d. Vektors>) |
| **Difference between elements in a vector**  > diff(<Name d. Vektors>) |
| **For logical vectors only:**  **> which (): provides the “TRUE” – indices of a logical vectors**   * also useful in conjunction w/ other functions like is.na that yield a logical vector |
| **Transponieren:**  > t(‚Vektor‘) |
| Konvertierung von Vektoren in anderen Datentyp (Converting Vectors)  ALSO CALLED COERCION [Zwang] | * > as.‘type of class‘(<Objekt>) * convert character vector into numeric vector: > as.numeric(<Objekt>)   + z.B.:    + this might be useful if one wants to run calculations:      * This can be done for any type of class:   + as.logical()   + as.character   + as.integer … * there are, however, some constraints on the conversion of vectors:   + see > showMethods(coerce) for an overview of possible coercions   + characters like letters can of course not be converted into numerics   + when converting a numeric vector into logical:     - the rule is, as usual, that 0 corresponds to FALSE and all non-zero (also negative) numbers will produce TRUE   + each kind of vector can be converted to a character vector |

|  |  |
| --- | --- |
| **Matrizen (matrices)** | * a matrix is basically a vector that stores data in 2 dimensions, rather than in only one * thus: what applies to vectors is most likely to apply to matrices as well   + e.g. each type of vector (for example logical or numerical) has its matrix version * if one does not specify rows and columns, matrix will always be regarded as vector in R |
| Erzeugen von Matrizen (matrices) | * **Create a matrix > matrix ()** * spaltenweise [by column] **(default)**   > matrix (c(‘numbers to insert in the matrix’, ncol = ‘number of columns’))   * for example:   alternativ: > matrix (c(‘numbers to insert in the matrix’, ncol = ‘number of columns’, byrow=FALSE)   * for example: * zeilenweise [by row]:  > matrix (c(‘numbers to insert in the matrix’), ncol = ‘number of columns’, byrow=TRUE) * for example: * diagonale Matrix:   > diag (<number to be displayed in diagonal>, nrow = <number of rows>, ncol=<number of col.>)   * for example:        * **Naming rows and columns** * > ‘Code zum Erstellen der Matrix (s.o.)’, dimnames = list(c(<Namen der Zeilen>, <Namen der Spalten>) 🡪 es müssen immer erst die Zeilennamen dann die Spaltennamen angegeben werden (default) * for example: * Alternative: use row names () and column names () after the matrix is created: * for example: |
| Rechnen mit Matrizen  (matrices) | * same as w/ vectors, all operators perform calculations **element-wise**  * there is only 1 matrix-specific operator: %\*% (Matrixprodukt) * for example:   🡪 Zeile Matrix1 \* Spalte Matrix 2 (e.g. 1\*1 + 4\*2 + 7\* 3 = 30) |
| Selecting and filtering | **Assessing (i.e. selecting and filtering) elements/subsequences of a vector/array/matrix/list with  1) integer Index/Indizes  2) rules/relational operators  3) boolean variables/logical vectors 4) characters**  **2. Subsetting matrices:**  **Summary:**  m1[2,3], m1[1,], m1[,1], m1[1:2,], m1[,1:2], m1[,3,drop=false], m1[1],m1[’r1’]  **Subsettting to a vector:**  **1) Select with 2-dimensional Index:** <Matrixname>[<Row Index>,<Column Index>]  for example    # access element in first row and second column     * same as w/ vectors: neg. numbers exclude rows or columns respectively     note: if one wants to keep the data structure *matrix* when accessing a complete column, rather than getting a vector:  use > drop = FALSE  for example:      **Subsetting to a matrix:** for example    # access elements in row 1-2 and column 2-3    **2) select with one-dimensional Index:** <Matrixname>[<Index>]   * as matrix is a collection of vectors, we can also use one-dimensional accessor * resulting subset is always a vector * gem. der Vektorschreibweise, setzt **R bei Matrizen Indizes spaltenweise**       **3) filter with rules/relational operators:** <Matrixname>[<Regel>]     * note: auch hier gilt, dass innerhalb der Regel nochmals auf die Matrix Bezug genommen werden muss   **4) select with character strings/vector for named matrix (e.g. dimnames):**    **5) extracting values:** <Matrixname>[[<Row Index>, <Column Index>]] |
| Nützliche Funktionen für Matrizen | Transponieren von Matrix:  > t(‘Name of Matrix’) |
| **Solution x of equation A\*x = b**  > solve (‘Name of Matrix’, b) |
| **Arithmetic mean of each column of a matrix**  colMeans(‘Name of Matrix’) |
| **Sum of each row of a matrix**  rowSums(‘Name of Matrix’) |
| **Number of rows in a matrix**  > nrow(‘Name of Matrix’)  **Number of columns in a matrix**  > ncol(‘Name of Matrix’) |
| **Dimension of a matrix**  > dim(‘Name of Matrix’) |
| **Diagonal of a matrix**  > diag(‘Name of Matrix’) |

|  |  |
| --- | --- |
| **Array** | * arrays are the R data objects that can store data in more than 2 dimensions * it is a natural extension to matrices in its number of dimensions   + an array is a vector that is represented and accessible in a specified number of dimensions (mostly > 2 dimensions)   + array w/ 2 dimension is of course a matrix; array w/ > 2 dimensions can be pictured as a cube w/ n different layers * for example: if we create an array of dimension (2,3,4) then it creates 4 rectangular matrices each with 2 rows and 3 columns |
| Erzeugen von Arrays | * an array takes vectors as input and uses the dim parameter to create an array * **create an array:** > array (c(‘numbers to be included in the array), dim=c(‘number of rows’, ‘number of columns’, ‘number of matrices’)) * der ‘Array’ wird wie folgt gebildet: je nach Anzahl der festgelegten Zeilen und Spalten werden die Zahlen über die Layer/Matrizen verteilt * examples:       # wdh. des Vektors über die Matrizen   * **out of given vectors:** > array (c(‘Vector1’,…,’Vector n’), dim=c(‘number of rows’, ‘number of columns’, ‘number of matrices’)) * for example:      * **naming rows and columns** > dimnames (‘name of array’) <- list (c(‘name of row1’,…, ‘row n’), c(‘name of column 1’, …,‘column n’), c(‘name of matrix 1’, …,’name of matrix 2’) 🡪 note: es macht Sinn, dass hier zunächst den Zeilen Namen gegeben werden müssen, dann den Spalten; denn grundsätzlich erstellt R eine Matrix spaltenweise und muss daher zunächst für jede Zeile die Namen kennen * for example:       dimnames can also be set directly in array definition: |
| Selecting and filtering | **Assessing (i.e. selecting and filtering) elements/subsequences of a vector/array/matrix/list with  1) integer Index/Indizes  2) rules/relational operators  3) boolean variables/logical vectors 4) characters**  **3. array:** <array>[<row index>,<column index>, <layer index>]   * **principle for subsetting a matrix also apply to subsetting an array**   for example:    **1) Select with multidimensional Index:**  only specifying row index: yields all columns and layers of a row    **- note: R directly coerces to matrix**  only specifying column index: yields all rows and layers of that column  only specifying layer index: yields all rows and columns in that layer    specifying with multiple indizes:    **2) select w/ one-dimensional index:**   * treats array as vector       **3) select with rules:**    **4) select with character strings for named array**     * **note:** when selecting w/ character strings we have to define all dimensions; otherwise, we will get following error |

|  |  |
| --- | --- |
| **Lists** | * lists are able to combine different objects (of potentially different types; even other lists) within one object * lists are an ordered sequence of those different objects   + lists are therefore also called recursive |
| Erzeugen von Listen | * > list() * for example:      * convert a vector to a list: as.list()   + for example:   + can be useful to then add other objects to that list * **convert a list into a vector:** > unlist (‘name of list’) * for example: |
| Namen der Listenelemente | * as usual with names() function:  > names(<list>) <- (<Name Listelement1>, …<Name Listelement n>)      * namen entfernen: names(<list>) <- NULL |
| Selecting and filtering | **Assessing (i.e. selecting and filtering) elements/subsequences of a vector/array/matrix/list with  1) integer Index/Indizes  2) rules/relational operators  3) boolean variables/logical vectors 4) characters**  **4. lists:**   * list access is overall **very similar to vector access**   **3.1 Subset to a new list:**  **1) select with integer Index/Indices:** <name of list>[<Index des Listenelementes>]    # access 1st element from list    # access 1st-2nd element from list    **2) select with boolean variables/logical vectors:**    **3) select with character strings/vectors** [<name of list element>]      **3.1 Extracting values:  🡪 getting a vector**  **Using $**  > <name of list>$<name of list element>[<Index/Rule>]  for example:    **# note that $ can only extract one list element at a time**  **also: no “ ” needed**      **Extracting with rules**    🡪 again: name of list has to be mentioned in rule as well  **Using** [[“<name of list element>”]]  **# note that [[]] can only extract one list element at a time**    **Using** [[<name of list element/index list element>]] [[<Index>]] 🡪 the first double bracket defines index of list element; **the second defines the index within the element**  **# note that double-[[]] can only extract one element of a list element at a time (see error message)**    Creating new vector with extracted values:  For example: |
| Löschen & Ergänzen von Listenelementen | * assign a new value to a list element or add a completely new element:   + ‘name of list’$’name of new object’<-‘value’        * assign multiple new values at once to a list element:   + ‘name of list’[c(“‘name of object 1’”,…,”’name of object n’”] <- list(‘name of object1’ = ’new value for object1’,…,’name of object n’=’new value for object n’) 🡪 statt eines neuen Wertes kann nat. auch wie mit der c-Funktion ein Vektor mit mehreren Einträgen definiert werden   + for example:      * beachte, wenn die Reihenfolge eingehalten wird kann auf die nochmalige Zuweisung des Variablennamens zu den neuen Werten verzichtet werden:      * remove one or multiple element(s) from a list: NULL   ‘name of list’$’name of object to be removed’/**[**c(“ ’name of object 1’ “, “ ‘name of object2’ “…)**]** <- NULL |
| List vs. Vector |  |
| Unlist | > unlist()   * note: when unlisting a list w/ different data types, R chooses the data type that is able to serve as common denominator to coerce the other ones (as usual): |
| **Data Frame (Tabelle)** | * Explicit package for working w/ data frames: dplyr * ein data frame ist eine Liste, die Vektoren der gleichen Länge enthält * represents **a dataset w/ a number of rows (=observations) and columns (=variables),** where the columns are not necessarily of the same data type  🡪 **most common form of dataset** * very useful when working w/ observations of different datatypes (Rintro Part 1, slide 40)   + **this is what you are usually working on if you have 3rd party data** * for example: |
| Anzahl d. Reihen (=Beobachtungen) und Spalten (= Variablen) | * nrow(<DATAFRAME>) * ncol(<DATAFRAME>) |
| Erzeugen eines Data Frames | > <name> <- data.frame(Ojekt1, Objekt2,...)   * beachte: sollen im dataframe die einzelnen Objekte/dataframe-Elemente benannt werden, kann die Zuweisung des Attributes **nur über das Gleichzeichen** erfolgen * for example:     or    **Erzeugen eines data frames aus Liste oder Matrix:**   * > data.frame(<Name Liste/Matrix>)  oder >as.data.frame(<Name Liste/Matrix>)       **Erzeugen eines data frames mit > expand.grid**   * erstellt ein data frame basierend auf allen möglichen Kombinationen der Vektoren und/oder Faktoren (siehe data frames und factors) angegeben:     **add or change column and row names with colnames() and rownames()**   * > colnames/rownames(<df>) <- c(<Name of row/col.1>…<Name of row/col. n>)     **lesen des generierten data frames:**  > print (<name dataframe>) alternativ: Klammer vor und nach data frame  **lesen eines data frames aus einer Datei:**  > read.table (‚Dateiname‘) |
| Selecting and filtering | **Assessing (i.e. selecting and filtering) elements/subsequences of a vector/array/matrix/list with  1) integer Index/Indizes  2) rules/relational operators  3) boolean variables/logical vectors 4) characters**  **4. Data frames**  **4.1) Subset a data frame as a list**   * note: subsetting data frames by treating them as list, does not support row-selection (see data frames as a matrix) * the same principles **as for lists** apply:     **1) select with integer Index/Indices:**    # access ‚list elements‘ 1-2      **2) select with character strings/vectors**     * **note:** even if we only choose one columne, subsetting a dataframe as a list,  still results in a dataframe not in a vector   **3) select with boolean variables**    **4.2) Subset a dataframe as a matrix**   * supports both, column and row selection * the same principles **as with matrix selection** apply   **1) select with two-dimensional integer Index/Indices** <name of df>[<Row Index>,<Column Index>]      **2) character strings/vectors (i.e. row- and columnnames)**       * note: as with matrices, we will get a vector, not a data.frame anymore, if we choose only one column and subset data.frame as matrix using two-dimensional integer index * keep the result as a data frame when assessing only one column:   > <name of df> [<# of rows>,] [<name of column>] (dadurch bleiben 2 dimensionen erhalten) s   * alternative: drop=false> <name of df> [<number of rows>,”<name of column>”, drop = FALSE] * z.B.:   **3) filter with rules/relational operators**  > ‘name data frame’[name data frame‘$‘Spaltenname‘*Regel***,]**  **🡪** relational expression results in logical vector; the latter is then used to filter the data frame 🡪 the comma at the end indicates that all columns shall be included in filtered data frame  🡪 this command DOES NOT work without the comma  **🡪** again name of dataframe has to be mentioned in rule again  another example w/ explicetly stated columns:    **4.3 Extract values w/ Index or Character Strings:**    **# note that $ and [[]] can only extract one data frame element at a time**  #> <name data frame>$<Spaltenname>  **note: $ uses partial matching; hence one could type $i and R would autocomplete it to id, if (!) there does not exist any other variable starting with i** |
| Hinzufügen von Spalten, Reihen, Werten und Überschreiben | **Hinzufügen (append) einer Spalte zu einem data frame mit cbind()**   * > <name erweitertes data frame> <- cbind(<name initial data frame>, <neuer (Spalten-)Vektor>)     **Hinzufügen (append) einer Reihe zu einem data frame: rbind()**   * > <name erweitertes data frames>  <- rbind(<name initial data frame> , **data.frame**(<name of col. 1> =<new value>, …, <name of col. n> = <new value>)     **Hinzufügen einer Spalte mit $:**    **Hinzufügen einer Spalte als Kalkulation bereits existierender Spalten:**      **Setting/overwrite new values to data frame, treating it as a list:**   * Assign new values to a data frame using **$**        * Alternatively **[]** works too: * Alternatively **[[]]** works in the same way with data frames:      * [] in conjunction with list() allows **to add multiple columns**:      * **Setting new values to data frame by treating it as a matrix:** * assigning new values to a data frame by treating it as a list, leads to the same problems as when subsetting it: we can only access columns * treating a data frame as a matrix provides more flexibility, as we consider its 2-dimensionality * if we need to set values w/ more flexibility, we shall use matrix notations:   changes rows 1-3 only |
| **Data frames and factors** | * a data frame tries to use memory more efficiently and sometimes this behavior might silently lead to unexpected problems * for example, if we create a data frame containing a character vector as column, by default it will get coerced to a factor * a factor only stores the same value once so that repetitions will not cost much memory * R uses factors to represent categorical variables with fixed possible values:   + a factor is essentially an integer vector, with a pre-specified set of possible values called levels      * + 🡪 each level represents a unique, possible value of the variable/attribute   + in the above example, Name, Gender and Major are not stored as character vectors, but as Factors to save memory space by only saving the possible values [Merkmalsausprägungen]     - that means ***only the unqiue values of those character vectors*** ***are stored in the factor dictionary***     - It is reasonable that Gender is represented by a factor because it may only be either Female or Male,  so ***using two integers to represent these two values*** is more efficient than using a character vector to store all the values regardless of the repetition   + however, it may induce problems for other columns with character-type data, that are not essentially categorical variables, but that can take alsmost an infinite amount of values     - take name as an example   + now, if we want to change/set a persons name, this is what happens:   + this happens, because in the initial name dictionary of the factor, there is no word called John  🡪 we cannot set the name of the first person with name that does not exist in variable’s dictionary   + we are restricted to the current dictionary when wanting to assign a new value * simplest way to avoid this behavior:  use > stringAsFactors = False  when creating a data frame     note: if we want character vector to be converted to factor we can explitely state it, just as we did with Gender by using factor() |
| Nützliche Funktion für data frames | Arrange data frame:  > <data frame> [order(<data frame>.$<column name>, descending = T /F)] |
| > summary(<dataframe>)  🡪 shows inter alia summary statistics, class and mode, length of factor |
| **Functions** | * a function is an object you can call * basically, it is a machine with internal logic that takes a group of inputs (parameters or arguments) and returns a value as output * In fact, in R environment, **everything we use is an object**, **everything we do is a function,** and, maybe to your surprise, **all functions are still objects** * although we might sometimes call it differently, any command that needs further processing from side of R is basically a function 🡪 even <- or + are functions, taking two arguments and returning a value as output 🡪 although they are called binary operators, they are at their core functions * When we do casual, interactive data analysis, at times, we won't have to write any function on our own since the built-in functions and those provided by thousands of packages are usually enough   + we have already covered some build in functions, such as is.numeric()that takes any input and returns a logical vector, depicting whether input is numeric or not * however, if one needs to repeat one’s logic or a process in data manipulation or analysis, it might be necessary to create one’s own function |
| **Creating a function** | Basic syntax:  <name of function> <- function**(**<arguments of the function>**)** **{**<function body**}**   * the function body contains a series of expressions, considering the arguments provided before * the value of the last expression determines the value returned by the function   Basic example:    we could use this function now to calculate sums defining any kind of x and y: |
| **Set default values for a function** | * we can easily assign a value to one of the arguments and thereby establishing a default * **if the function gets called, we only have to specify the remaining arguments**:     we only have to define x in this example |
| **Calling a function** | * Once the function is defined, we can call the function just as we do in math   + we just need to provide e.g. values for the arguments: * **R process**:   + when we call the function, R will find out if there is a function named *add* in the environment   + Then, it will figure out that *add* refers to the function we just created and creates a local environment in which x takes 2 and y takes 3.   + The expression in the function body is then evaluated given the values of the arguments   + Finally, the function returns the value of that expression, 5 * note: as long as the function body, hence the mathematical expression, allows for it, the input can be anything :   + the type of inputs are not fixed prior to a function’s calling   + the function as.Date creates a Date object |
| **Generalizing a function** | * Developers often want a function to be general enough to adapt to a wide range of use cases   + thereby, we can use it for similar problems, w/out rewriting it completely * making a function more widely applicable is called ***generalization*** * we can generalize a function easily by using an if-else if statement:      1. **Define all functions arguments** needed for mathemacial expression + **a placeholder variable** (in this example *type*) 2. define the placeholder in the function body: if (*placeholder* == *specific value) {do specific calculation} else if*(*placeholder* == *specific value)(..)* 3. set {stop} argument  🡪 it is important to end the loop! 🡪 note: the text in columns is only for information purposes and does not serve any function in the if-else if statement   Again, once the function is set, we can call it providing respective function arguments:  Avoid default of R automatically proceeding by picking first matching argument of placeholder argument in if-else if statements:   * the last error message occurs because we have provided 2 entries for the “type” arguments, but only 1 value for the “type” argument in the if- else-if statements * in order to avoid that R automatically picks the first match it finds in the statement, as stated in the error message, we can restrict our if-else if statement further by **adding another if-stop statement:**        * command that only logical vectors of length one shall be taken into account 🡪 remember: with logical operators, comparisons yield TRUE or FALSE only * note: after a stop, we start w/ if statement again |

|  |  |
| --- | --- |
| **Making sets: union(), union all() (dplyr function)**  **intersect()**  **setdiff()** | 1. **union(<object A>, <object B>):** Union of two sets is everything in the two sets taken together, but counting elements only once that are common to both sets 2. **union all():** union all of 2 sets combines all values of both sets, w/out omitting redundancies:      1. **intersect(<object A>, <object B>):** intersect of 2 sets only results the overlap between to sets:      1. **setdiff(<object A>, <object B>):** setdiff of 2 sets only results the difference between object A and object B for object A or B  note: order in stating object A and B counts |