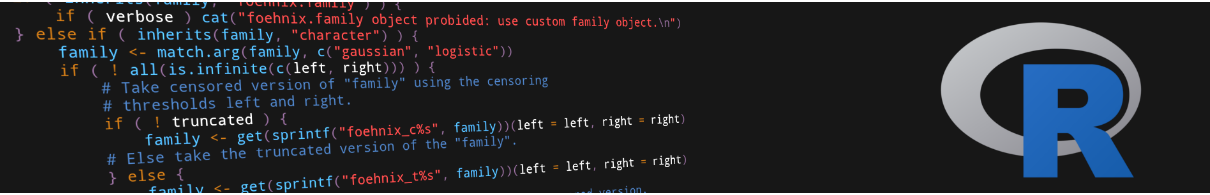
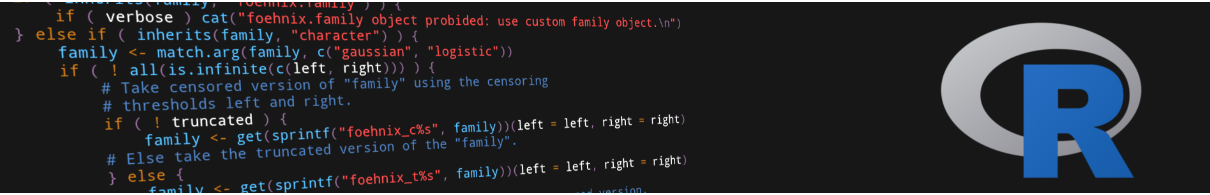


```
    if ( verbose ) cat("foehnix.family object probided: use custom family object.\n")
} else if ( inherits(family, "character") ) {
    family <- match.arg(family, c("gaussian", "logistic"))
    if ( ! all(is.infinite(c(left, right))) ) {
        # Take censored version of "family" using the censoring
        # thresholds left and right.
        if ( ! truncated ) {
            family <- get(sprintf("foehnix_c%s", family))(left = left, right = right)
        # Else take the truncated version of the "family".
        } else {
            family <- get(sprintf("foehnix_t%s", family))(left = left, right = right)
```

R programming

Fundamentals

Thorsten.Simon@uibk.ac.at



```
    if ( verbose ) cat("foehnix.family object probided: use custom family object.\n")
} else if ( inherits(family, "character") ) {
    family <- match.arg(family, c("gaussian", "logistic"))
    if ( ! all(is.infinite(c(left, right))) ) {
        # Take censored version of "family" using the censoring
        # thresholds left and right.
        if ( ! truncated ) {
            family <- get(sprintf("foehnix_c%s", family))(left = left, right = right)
        # Else take the truncated version of the "family".
        } else {
            family <- get(sprintf("foehnix_t%s", family))(left = left, right = right)
```

Data structures

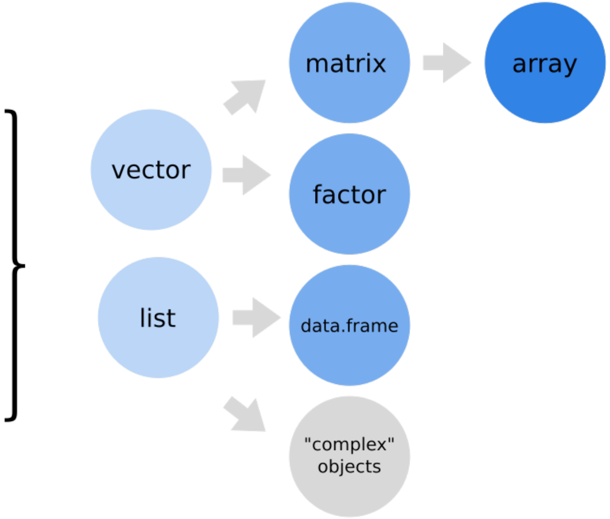
Fundamentals

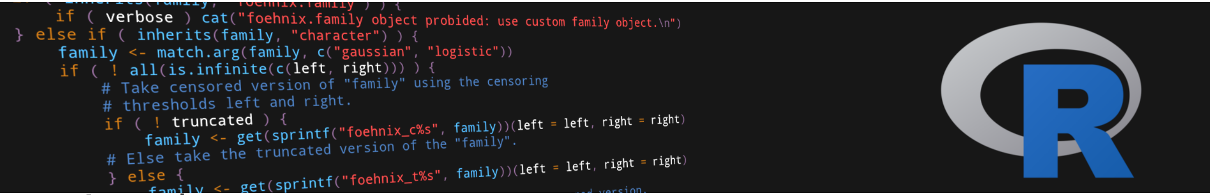
Thorsten.Simon@uibk.ac.at

Data structures

Atomic classes

- double (numeric)
- integer
- logical
- character
- complex





```
    if ( verbose ) cat("foehnix.family object probided: use custom family object.\n")
} else if ( inherits(family, "character") ) {
    family <- match.arg(family, c("gaussian", "logistic"))
    if ( ! all(is.infinite(c(left, right))) ) {
        # Take censored version of "family" using the censoring
        # thresholds left and right.
        if ( ! truncated ) {
            family <- get(sprintf("foehnix_c%s", family))(left = left, right = right)
        # Else take the truncated version of the "family".
        } else {
            family <- get(sprintf("foehnix_t%s", family))(left = left, right = right)
```

Atomic vectors

Fundamentals

Thorsten.Simon@uibk.ac.at

Overview

There are five commonly used types of atomic vectors.

- **Logical:** `TRUE` and `FALSE`
- **Integer:** `..., -1L, 0L, 1L, 2L, ...`
- **Double:** `0.3, 1.0, -3.14, 2`
- **Character:** `"Innsbruck", "one", "5"`
- **Complex:** `0 + 0i, 1 - 1i, -1 + 1i`

All elements of an atomic vector must be of the same type.

Construction

Vectors of length one are constructed by assigning an *value* to an *object*:

```
R> dbl_var <- 4.5
R> dbl_var
```

```
[1] 4.5
```

Vectors with a length greater than one are constructed using `c()` (combine):

```
R> chr_var <- c("Good", "morning", "Innsbruck")
R> chr_var
```

```
[1] "Good"      "morning"   "Innsbruck"
```

Missing values in a vector are specified with `NA`:

```
R> chr_var <- c("Good", NA, "Innsbruck")
R> chr_var
```

```
[1] "Good"      NA          "Innsbruck"
```

Construction

Structured vectors can be constructed using `rep()`, `seq()` or the colon operator (`:`).

```
R> log_var <- rep(c(TRUE, FALSE), times = 4)
R> dbl_var <- seq(0, 1, by = 0.01)
R> int_var <- 1L:5L
```

An empty vector can be initialized using `vector`

```
R> chr_var <- vector("character", length = 5)
R> chr_var
```

```
[1] "" "" "" "" ""
```

or `logical()`, `integer()`, `...`

Construction

Some vectors with constants are built into R (check `?Constants`):

```
R> LETTERS
```

```
 [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O"
[16] "P" "Q" "R" "S" "T" "U" "V" "W" "X" "Y" "Z"
```

```
R> letters
```

```
 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o"
[16] "p" "q" "r" "s" "t" "u" "v" "w" "x" "y" "z"
```

```
R> month.abb
```

```
 [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct"
[11] "Nov" "Dec"
```

```
R> month.name
```

```
 [1] "January"   "February"  "March"     "April"     "May"
 [6] "June"      "July"      "August"    "September" "October"
[11] "November"  "December"
```

```
R> pi
```

```
[1] 3.142
```

Tests and Types

To query the length of a vector use `length()`:

```
R> length(log_var)
[1] 8
R> length(dbl_var)
[1] 101
```

To query the type of a vector use `typeof()`:

```
R> typeof(dbl_var)
[1] "double"
R> typeof(int_var)
[1] "integer"
```

Tests and Types

To test if a vector is of a specific type use `is.logical()`, `is.integer()`, `...`

```
R> is.character(chr_var)
[1] TRUE
R> is.double(int_var)
[1] FALSE
```

Note:

- `is.atomic()` tests if the object is of an atomic type.
- `is.numeric()` test if the object is interpretable as number. It returns `TRUE` for type integer and type double.

Coercion

To coerce a vector from one type to another use `as.logical()`, `as.integer()`, `...`, which works from *more* to *less flexible*:

```
R> as.logical(c(0L, 1L))
[1] FALSE  TRUE
R> as.logical(c("FALSE", "TRUE"))
[1] FALSE  TRUE
```

...and from *less* to *more flexible*:

```
R> as.character(c(0, 1))
[1] "0" "1"
R> as.character(c(FALSE, TRUE))
[1] "FALSE" "TRUE"
```

Coercion

When two vectors of different types are combined, R applies auto-coercion to the most flexible type:

```
R> c(c(-1), c(FALSE, TRUE))
```

```
[1] -1  0  1
```

```
R> c("FALSE", TRUE)
```

```
[1] "FALSE" "TRUE"
```

```
R> c("one", c(1, TRUE))
```

```
[1] "one" "1"   "1"
```

Note:

- Most mathematical functions (e.g. `sum()`, `mean()`, `+`, `log()`) coerce the input to a double or integer.
- Logical operations (`&`, `|`, `any()`, `...`) coerce to logical.
- String modulating functions (`paste()`, `cat()`, `...`) coerce to character.

Subsetting

You subset vectors using the `[` operator in combination with

- **positive integers**
- **negative integers**
- **logical vectors**
- **character vectors**

Define a vector:

```
R> x <- c(2.1, 4.2, 3.3, 5.4)
```

Subsetting

Positive integers return elements at specified positions.

```
R> x[c(3, 1)]
```

```
[1] 3.3 2.1
```

```
R> x[order(x)]          ## sort x
```

```
[1] 2.1 3.3 4.2 5.4
```

```
R> x[c(1, 1)]           ## recycle elements
```

```
[1] 2.1 2.1
```

```
R> y <- LETTERS[1:4]
R> y
```

```
[1] "A" "B" "C" "D"
```

```
R> order(x)             ## returns a integer vector with indices
```

```
[1] 1 3 2 4
```

Subsetting

Negative integers omit elements at specified positions.

```
R> x[c(-3, -1)]
```

```
[1] 4.2 5.4
```

Positive and negative integers can’t be mixed in a single subset.

Subsetting

Logical vectors select elements where the corresponding logical value is `TRUE`.

```
R> x[c(TRUE, TRUE, FALSE, FALSE)]

[1] 2.1 4.2

R> x[x > 4]

[1] 4.2 5.4

R> x[y == "B"]

[1] 4.2
```

Subsetting

Character vectors to return elements with matching names (Names will be introduced later along with attributes):

```
R> names(x) <- y
R> x
```

```
  A   B   C   D
2.1 4.2 3.3 5.4
```

```
R> x[c("A", "D")]
```

```
  A   D
2.1 5.4
```

```
R> x[c("A", "A", "A", "D", "D", "D")]
```

```
  A   A   A   D   D   D
2.1 2.1 2.1 5.4 5.4 5.4
```

Subsetting and assignment

All subsetting operators can be combined with assignment <- to modify selected values of the input vector.

```
R> x <- 1:5
R> x[c(1, 2)] <- c(-999, -999)
R> x
```

```
[1] -999 -999    3    4    5
```

Example: Expanding abbreviations

You have a character vector `x` with abbreviations of the months:

```
R> set.seed(111)
R> x <- sample(month.abb, size = 1000, replace = TRUE)
R> head(x)
```

```
[1] "Nov" "Apr" "Mar" "Sep" "Nov" "May"
```

Expand the abbreviations to the full names:

```
[1] "November"  "April"     "March"     "September" "November"
[6] "May"
```

Hint: `month.name` and `month.abb` give you the full names and abbreviations of the months, respectively.

Example: Expanding abbreviations

Solution using names attribute and character subsetting:

```
R> names(month.name) <- month.abb
R> y <- month.name[x]
R> head(y)
```

```
        Nov         Apr         Mar         Sep         Nov 
 "November"     "April"     "March" "September"  "November" 
        May 
      "May" 
```

Alternatively (and likely more common) using `match()`ing and integer subsetting:

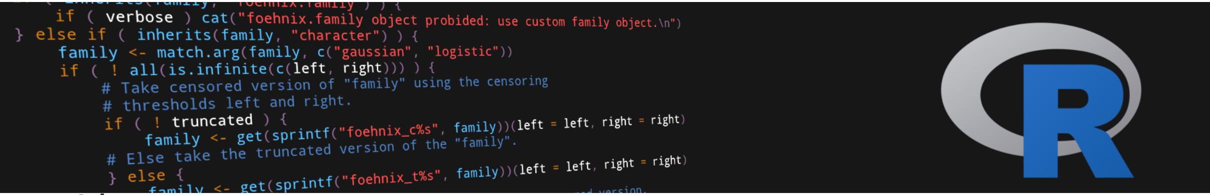
```
R> y <- month.name[match(x, month.abb)]
R> head(y)
```

```
        Nov         Apr         Mar         Sep         Nov 
 "November"     "April"     "March" "September"  "November" 
        May 
      "May" 
```

Excursus: Arithmetics

`+`, `-`, `*`, `/`, `^`, `**`, `%%`, `%/%` work for vectors:

```
R> 1:3 + 3:1
[1] 4 4 4
R> 1:3 + 2                # single values are recylcled
[1] 3 4 5
R> 2^(0:4)
[1]  1  2  4  8 16
R> (0:5)^2
[1]  0  1  4  9 16 25
R> 1:10 %% 3              # modulo
 [1] 1 2 0 1 2 0 1 2 0 1
R> 1:10 %/% 3             # integer division
 [1] 0 0 1 1 1 2 2 2 3 3
```



```
    if ( verbose ) cat("foehnix.family object probided: use custom family object.\n")
} else if ( inherits(family, "character") ) {
    family <- match.arg(family, c("gaussian", "logistic"))
    if ( ! all(is.infinite(c(left, right))) ) {
        # Take censored version of "family" using the censoring
        # thresholds left and right.
        if ( ! truncated ) {
            family <- get(sprintf("foehnix_c%s", family))(left = left, right = right)
        # Else take the truncated version of the "family".
        } else {
```

Lists

Fundamentals

Thorsten.Simon@uibk.ac.at

Lists

- *Lists* are the other basic data structure in R.
- A list is a vector in which the elements can be of different types.
- A single element of a list could be again a list, which can lead to a *recursive* structure.
- `list()` is used to consruct a list:

```
R> x <- list(1L:100L, dbl_var, list("recursive_element", log_var))
R> str(x)
```

```
List of 3
 $ : int [1:100] 1 2 3 4 5 6 7 8 9 10 ...
 $ : num [1:101] 0 0.01 0.02 0.03 0.04 0.05 0.06 0.07 0.08 0.09 ...
 $ :List of 2
  ..$ : chr "recursive_element"
  ..$ : logi [1:8] TRUE FALSE TRUE FALSE TRUE FALSE ...
```

`str()` helps to examine the structure of a list.

Tests and Types

A list has a length and is of type ‘list‘, which ca be directly tested with ‘is.list()‘:

```
R> length(x)
```

```
[1] 3
```

```
R> typeof(x)
```

```
[1] "list"
```

```
R> is.list(x)
```

```
[1] TRUE
```

`as.list()` coerces a vector to a list.

```
R> y <- 1:3
R> str(as.list(y))
```

```
List of 3
 $ : int 1
 $ : int 2
 $ : int 3
```

Tests and Types

An empty list can be initialized with `list()`. Use `vector()` to initialize a list of predefined length:

```
R> new_list <- list()
R> str(new_list)
```

```
 list()
```

```
R> new_list_with_predefined_length <- vector("list", length = 3)
R> str(new_list_with_predefined_length)
```

```
List of 3
 $ : NULL
 $ : NULL
 $ : NULL
```

where the elements of the new list contain R’s null object `NULL`.

Subsetting

Subsetting a list with [works the same way as subsetting an atomic vector, i.e., with positive and negative integers, logical and character vectors. Subsetting with [always returns a list, potentially only with one element.

```
R> x <- list(given = "Jan-Bernd", family = "Schmitt", born = 1976)
R> str(x[c(TRUE, TRUE, FALSE)])
```

```
List of 2
 $ given : chr "Jan-Bernd"
 $ family: chr "Schmitt"
```

```
R> str(x["born"])
```

```
List of 1
 $ born: num 1976
```

Subsetting

`[[` pulls out the element of the list, and works with positive integers and character vectors.

```
R> str(x[[1]])
```

```
 chr "Jan-Bernd"
```

```
R> str(x[["born"]])
```

```
 num 1976
```

`$` is a short-cut for `[[`.

```
R> x$family
```

```
[1] "Schmitt"
```

```
R> x$born
```

```
[1] 1976
```

Subsetting

If [[gets an vector of length greater than one, it applies the indices/names recursively.

```
R> x <- list(full = list(given = "Jan-Bernd", family = "Schmitt"), born = 1976)
R> str(x)
```

```
List of 2
 $ full:List of 2
  ..$ given : chr "Jan-Bernd"
  ..$ family: chr "Schmitt"
 $ born: num 1976
```

```
R> str(x[[c(1, 2)]])
```

```
 chr "Schmitt"
```

```
R> str(x[[c("full", "family")]])
```

```
 chr "Schmitt"
```

Attributes

One application of lists is that one can set *attributes* of an object. (The elements of the list must be *named*.)

```
R> y <- 1:3
R> attributes(y) <- list(my_attr = "my_vector")
R> y
```

```
[1] 1 2 3
attr(,"my_attr")
[1] "my_vector"
```

- Use `attributes()` to set or access all attributes.
- Use `attr(x, which)` to set or access individual attributes.

Attributes

```
R> attr(y, "my_attr")
```

```
[1] "my_vector"
```

```
R> attr(y, "my_attr") <- "your_vector"
R> str(attributes(y))
```

```
List of 1
 $ my_attr: chr "your_vector"
```

Attributes

There are special attributes of a vector:

- **Names**: A character vector giving each element a name. Names must be the same length as the vector.
- **Dimensions**: A integer vector used to turn vectors into matrices or arrays.
- **Class**: A character vector used to implement the S3 object system.

Each of these attributes has a specific accessor function to get and set values. When working with these attributes use `names(x)`, `dim(x)` and `class(x)`, not `attr(x, "names")`, etc.

Names

Set and access names of a vector.

```
R> y <- 1:3
R> names(y) <- c("A", "B", "C")
R> y
```

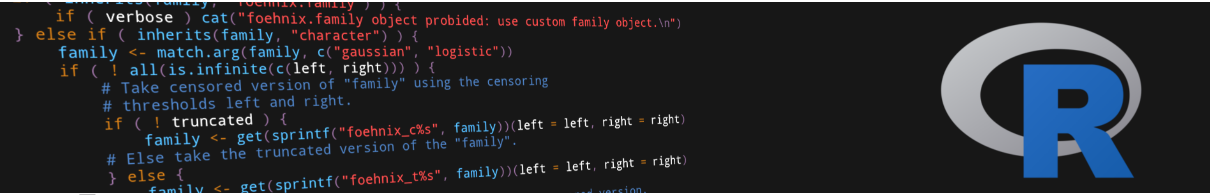
```
A B C
1 2 3
```

```
R> names(y)
```

```
[1] "A" "B" "C"
```

Alternative ways to set names are:

```
R> y <- c(A = 1L, B = 2L, C = 3L)
R> y <- setNames(1:3, c("A", "B", "C"))
R> ## Not recommended:
R> y <- 1:3
R> attr(y, "names") <- c("A", "B", "C")
```



```
    if ( verbose ) cat("foehnix.family object probided: use custom family object.\n")
} else if ( inherits(family, "character") ) {
    family <- match.arg(family, c("gaussian", "logistic"))
    if ( ! all(is.infinite(c(left, right))) ) {
        # Take censored version of "family" using the censoring
        # thresholds left and right.
        if ( ! truncated ) {
            family <- get(sprintf("foehnix_c%s", family))(left = left, right = right)
        # Else take the truncated version of the "family".
        } else {
            family <- get(sprintf("foehnix_t%s", family))(left = left, right = right)
```

Factors

Fundamentals

Thorsten.Simon@uibk.ac.at

Excursus: Categorical data

Categorical data is used to describe categories. Sometimes this type of data is also to as *qualitative*. It is always discrete and be of one of the following kinds:

- **Binary**: Two possible outcomes, e.g. Occurrence of an event such as landfall of a hurricane (yes/no).
- **Multinomial**: Three or more possible outcomes which are mutually exclusive and exhaustive (and *non-ordered*, e.g. field of study (natural sciences / social sciences / engineering)
- **Ordered**: Three or more possible outcomes which are mutually exclusive and exhaustive (and *ordered*, e.g. Do you agree with the political program of the ruling party (strongly agree / agree / neutral / disagree / strongly disagree)?

Factors

A *factor* is a vector representing categorical data.
Factors are built on top of integer vectors using two attributes: *class* and *levels*.

```
R> group <- factor(c("A", "O", "B", "AB", "A", "O", "O", "A", "A"))
R> group
```

```
[1] A  O  B  AB A  O  O  A  A
Levels: O A AB B
```

```
R> class(group)
```

```
[1] "factor"
```

```
R> levels(group)
```

```
[1] "O"  "A"  "AB" "B"
```

```
R> attributes(group)
```

```
$levels
[1] "O"  "A"  "AB" "B"

$class
[1] "factor"
```

Factors

Althought not often needed in practice, the integer vector could be obtained by coercion:

```
R> typeof(group)
[1] "integer"
R> str(group)
 Factor w/ 4 levels "O","A","AB","B": 2 1 4 3 2 1 1 2 2
R> as.integer(group)
[1] 2 1 4 3 2 1 1 2 2
R> attributes(as.integer(group))
NULL
```

Construction

You can specify the levels when not all possible values are included in the vector.

```
R> group <- factor(
+     c("A", "O", "B", "A", "O", "O", "A", "A"),
+     levels = c("A", "B", "AB", "O")
+   )
R> group
```

```
[1] A O B A O O A A
Levels: A B AB O
```

You can't assign values not included in the levels to the factor:

```
R> group[9] <- "AB"
R> group
```

```
[1] A  O  B  A  O  O  A  A  AB
Levels: A B AB O
```

```
R> group[10] <- "AC"    ## gives a warning (suppressed here)
R> group
```

```
 [1] A    O    B    A    O    O    A    A    AB   <NA>
Levels: A B AB O
```

Construction

You construct a factor from a atomic vector by specifying the arguments levels and labels

```
R> x <- rep(1:12, times = 2)
R> factor(x, levels = 1:12, labels = month.abb)
```

```
 [1] Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec Jan Feb Mar
[16] Apr May Jun Jul Aug Sep Oct Nov Dec
Levels: Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
```

Duplicated values in labels can be used to map different values of x to the same factor level.

```
R> factor(x, levels = 1:12, labels = rep(paste0("Q", 1:4), each = 3))
```

```
 [1] Q1 Q1 Q1 Q2 Q2 Q2 Q3 Q3 Q3 Q4 Q4 Q4 Q1 Q1 Q1 Q2 Q2 Q2 Q3 Q3
[21] Q3 Q4 Q4 Q4
Levels: Q1 Q2 Q3 Q4
```

Construction

Construction of a categorical variable (factor) from a countinuos variable (numeric).

```
R> x <- c(8.3, 10, 9.2, 1.3, 8.2, 3.7, 6.2, 6.8, 4.8, 8.7)
R> x <- cut(x, breaks = c(0, 3, 7, 10), labels = c("low", "mid", "high"))
R> x
```

```
 [1] high high high low  high mid  mid  mid  mid  high
Levels: low mid high
```

As these levels are *ordered* it makes sense to coerce it to an ordered factor:

```
R> x <- as.ordered(x)
R> x
```

```
 [1] high high high low  high mid  mid  mid  mid  high
Levels: low < mid < high
```

```
R> class(x)
```

```
[1] "ordered" "factor"
```

Construction

Sometimes it is helpful to condition your data analysis on the interaction of two factors.

```
R> group
```

```
 [1] A    O    B    A    O    O    A    A    AB   <NA>
Levels: A B AB O
```

```
R> sex <- factor(c("f", "m", "f", "f", "f", "f", "f", "f", "m", "m"))
R> interaction(sex, group, sep = ":")
```

```
 [1] f:A  m:O  f:B  f:A  f:O  f:O  f:A  f:A  m:AB <NA>
Levels: f:A m:A f:B m:B f:AB m:AB f:O m:O
```

Excursus: Generic Functions

A factor is of class `factor`. There are *generic functions* as `print()`, `summary()`, `plot()`, `...`, that produce output tailored to the input, e.g. `summary()` of a `factor` produces a `table` rather than a quantile statistics:

```
R> summary(group)
```

```
   A    B   AB    0 NA's 
   4    1    1    3    1 
```

```
R> summary(dbl_var)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
   0.00    0.25    0.50    0.50    0.75    1.00 
```

Example: Explore categorial data with tables

Take the *Chicken Weights by Feed Type* dataset

```
R> feed <- chickwts$feed           # the $ extracts variables from a data.frame
R> weight <- chickwts$weight
```

- Explore the variables using `summary()` and `class()`.
- Give probabilities how many chicks are light-weightd ($weight \leq 200$), medium ($200 < weight \leq 300$) or well-fed ($weight > 300$).
- Give these probabilities conditioned on the type of feeding.

```
R> summary(feed)      ## class factor

   casein horsebean   linseed  meatmeal   soybean sunflower 
       12        10        12        11        14        12 

R> summary(weight)    ## class numeric

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
    108     204     258     261     324     423 
```

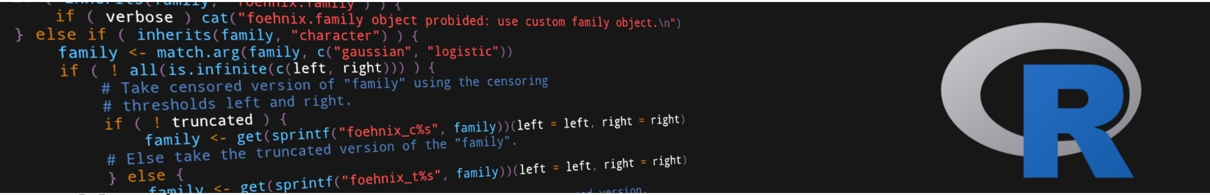
Example: Explore categorial data with tables

```
R> weight <- cut(weight,
+    breaks = c(0, 200, 300, 500),
+    labels = c("light", "medium", "well-fed")
+  )
R> prop.table(table(weight))

weight
   light   medium well-fed
  0.2394   0.3944   0.3662

R> print(prop.table(table(weight, feed), 2), digits = 2)

          feed
weight     casein horsebean linseed meatmeal soybean sunflower
  light     0.000     0.800   0.333    0.091   0.286     0.000
  medium    0.333     0.200   0.583    0.455   0.500     0.250
  well-fed  0.667     0.000   0.083    0.455   0.214     0.750
```



```
    if ( verbose ) cat("foehnix.family object probided: use custom family object.\n")
} else if ( inherits(family, "character") ) {
    family <- match.arg(family, c("gaussian", "logistic"))
    if ( ! all(is.infinite(c(left, right))) ) {
        # Take censored version of "family" using the censoring
        # thresholds left and right.
        if ( ! truncated ) {
            family <- get(sprintf("foehnix_c%s", family))(left = left, right = right)
        # Else take the truncated version of the "family".
        } else {
            family <- get(sprintf("foehnix_t%s", family))(left = left, right = right)
```

Matrices

Fundamentals

Thorsten.Simon@uibk.ac.at

Matrices

Multi-dimensional data is represented by

- **matrices** for two dimensions
- **arrays** for three and more dimensions.

Matrices and arrays are vectors with an additional attribute `dim` for the dimensions.

As a matrix is based on a vector, all elements must be of the same type.

```
R> m <- matrix(1:6, ncol = 3, nrow = 2)
R> m
```

```
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

Construction

Access number of rows and columns, and manipulate names of rows and columns:

```
R> dim(m)
```

```
[1] 2 3
```

```
R> c(nrow(m), ncol(m))
```

```
[1] 2 3
```

```
R> length(m)
```

```
[1] 6
```

```
R> rownames(m) <- c("A", "B")
R> colnames(m) <- c("a", "b", "c")
R> m
```

```
  a b c
A 1 3 5
B 2 4 6
```

Combining

`c()` generalises to

- `cbind()` for column binding, and
- `rbind()` for row binding.

```
R> m1 <- matrix(1:4, nrow = 2)
R> m1
```

```
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

```
R> m2 <- matrix(15:20, nrow = 2)
R> m2
```

```
     [,1] [,2] [,3]
[1,]   15   17   19
[2,]   16   18   20
```

```
R> cbind(m1, m2)
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3   15   17   19
[2,]    2    4   16   18   20
```

Subsetting

You can subset a matrix with

- two vectors
- a single vector
- a matrix

Here is a matrix:

```
R> m <- outer(1:3, 1:3, FUN = paste, sep = ",")
R> colnames(m) <- LETTERS[1:3]
R> rownames(m) <- paste0("r", 1:3)
R> m
```

```
   A     B     C    
r1 "1,1" "1,2" "1,3"
r2 "2,1" "2,2" "2,3"
r3 "3,1" "3,2" "3,3"
```

Subsetting

Using two vectors, the first for the rows, the second for the columns:

```
R> m[c(1,3), ]  ## blank subsetting
```

```
   A     B     C    
r1 "1,1" "1,2" "1,3"
r3 "3,1" "3,2" "3,3"
```

Blank subsetting lets you keep all rows or columns.

```
R> m[c(TRUE, FALSE, TRUE), c("C", "B")]
```

```
   C     B    
r1 "1,3" "1,2"
r3 "3,3" "3,2"
```

Different types of vectors can be combined.

Subsetting

As a matrix is basically a vector, you can subset it using a single vector:

```
R> m[rep(c(TRUE, FALSE, TRUE), 3)]
[1] "1,1" "3,1" "1,2" "3,2" "1,3" "3,3"
R> m[c(2, 8)]
[1] "2,1" "2,3"
```

Note that `[` simplifies the result to the lowest possible dimension by default.

Subsetting

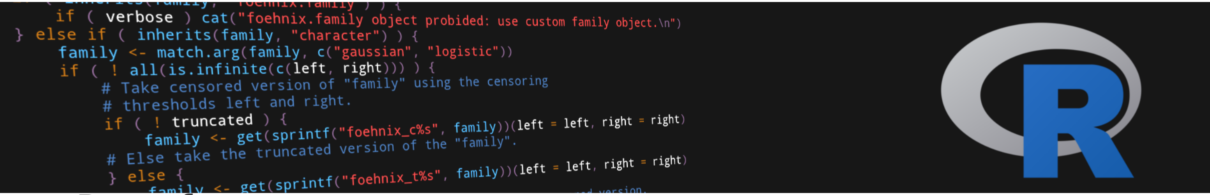
You can subset a matrix with a matrix, where each row in the *indexing* matrix specifies the position of one value. The first column corresponds to the rows, the second column correspond to the columns.

```
R> select <- matrix(c(2, 1, 3, 2, 1, 3), ncol = 2, byrow = TRUE)
R> select
```

```
     [,1] [,2]
[1,]    2    1
[2,]    3    2
[3,]    1    3
```

```
R> m[select]
```

```
[1] "2,1" "3,2" "1,3"
```



```
    if ( verbose ) cat("foehnix.family object probided: use custom family object.\n")
} else if ( inherits(family, "character") ) {
    family <- match.arg(family, c("gaussian", "logistic"))
    if ( ! all(is.infinite(c(left, right))) ) {
        # Take censored version of "family" using the censoring
        # thresholds left and right.
        if ( ! truncated ) {
            family <- get(sprintf("foehnix_c%s", family))(left = left, right = right)
        # Else take the truncated version of the "family".
        } else {
            family <- get(sprintf("foehnix_t%s", family))(left = left, right = right)
```

Data frames

Fundamentals

Thorsten.Simon@uibk.ac.at

Data frames

- A data frame is the common way to work with data in R.
- A data frame is a list with equal-length vectors, and additional attributes *names*, *class* and *row.names*.
- This allows one to bind variables of different type together, of which the elements with the same index are observed jointly.
- Thus a data frame is a 2-dim structure, but shares properties of both matrix and list:
- `names()` returns the same as `colnames()`.
- `length()` returns the same as `ncol()`.

Construction

You can construct a data frame from scratch:

```
R> d <- data.frame(x = 1:4, y = c("a","b", "c", "d"))
R> head(d)
```

```
  x y
1 1 a
2 2 b
3 3 c
4 4 d
```

However, it’s more common to use R’s I/O infrastructure to import data which is then provided as data frame.

Note that many functions that return a data frame coerce character vectors to factors, which can be suppressed by setting `stringAsFactors = FALSE` argument.

Testing and coercion

The type of a data frame is `list`. Explicit testing is performed using `is.data.frame()`

```
R> typeof(d)
[1] "list"
R> is.data.frame(d)
[1] TRUE
```

`as.data.frame()` coerces

- a **vector** to a one-column data frame.
- a **list** to a data frame, where each element of the list will be a column in the data frame. An error occurs if the elements are not of equal length.
- a **matrix** to a data frame, where each column of the matrix will be a column in the data frame.

Combining data frames

You can use `cbind()` to add columns (variables). However, `merge()` in base R gives you more control with the `by` argument.

```
R> d2 <- data.frame(x = 4:1, z = rev(c("aa","bb", "cc", "dd")))
R> d2
```

```
  x  z
1 4 dd
2 3 cc
3 2 bb
4 1 aa
```

```
R> merge(d, d2, by = "x")
```

```
  x y  z
1 1 a aa
2 2 b bb
3 3 c cc
4 4 d dd
```

```
R> d3 <- merge(d, d2, by = "x")
```

Combining data frames

You can use `rbind()` to add rows (observations). However, number and names of columns must match. Use `dplyr::bind_rows()` to combine data frames that don't have the same columns.

```
R> dplyr::bind_rows(d, d2)
```

```
  x    y    z
1 1    a <NA>
2 2    b <NA>
3 3    c <NA>
4 4    d <NA>
5 4 <NA>   dd
6 3 <NA>   cc
7 2 <NA>   bb
8 1 <NA>   aa
```

Subsetting

When subsetting data frames with `[`, they behave like a

- **list** when you provide a single vector, or like a
- **matrix** when you provide a two vector separated with a comma.

When subsetting with `[[` or `$` the data frame behaves like a list.

Furthermore, you can subset data frames using `subset()`.

Subsetting

The first argument of `subset()` is the data frame. The second and third arguments are `subset` and `select`.

- `subset`: logical expression indicating the rows to keep (`NA` are taken as `FALSE`).
- `select`: indicating columns to select.

```
R> subset(d3, subset = x > 2, select = c("y", "z"))
```

```
  y  z
3 c cc
4 d dd
```

Note: The logical expression `subset` is evaluated in the data frame, so columns can be referred to (by name) as variables in the expression.

Subsetting and assignment

`$` or `[[` plus assignment `<-` can be used to create new columns, or to delete column with `NULL`.

```
R> d3$yz <- paste(d3$y, d3$z, sep = "-")
R> d3$z <- NULL
R> d3
```

```
  x y   yz
1 1 a a-aa
2 2 b b-bb
3 3 c c-cc
4 4 d d-dd
```

Example: Convert units

- Load the data `mtcars`:

```
R> data(mtcars)
```

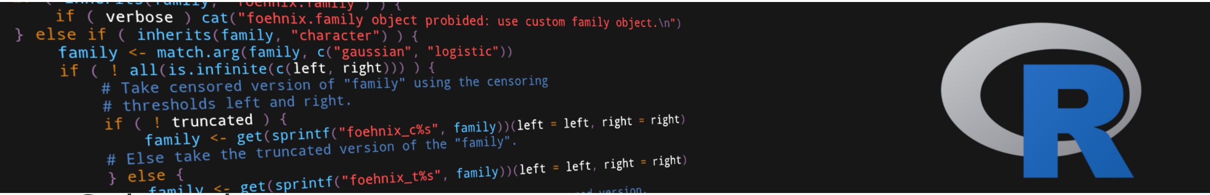
- Remove all columns, but the ones containing *miles per gallon*, *displacement* and *weight*. (Hint: Check the help page for the data *?mtcars* to find the right columns.)
- Construct a new data frame ‘autos‘ with 4 columns:
    - ‘Typ‘: Type of car.
    - ‘Verbrauch‘ in l/100 km.
    - ‘Hubraum‘ in ccm.
    - ‘Gewicht‘ in kg.
- Use these constants for converting the units:

```
R> mile <- 1.609344          # km
R> gallon <- 3.7854          # liter
R> cubic_inch <- 16.38706    # ccm
R> pound <- 0.4535924        # kg
```

Example: Convert units

```
R> mtcars <- subset(mtcars, select = c("mpg", "disp", "wt"))
R> autos <- data.frame(
+     "Typ"         = rownames(mtcars),
+     "Verbrauch" = 1/mtcars$mpg * gallon * 1/mile * 100,
+     "Hubraum"    = mtcars$disp * cubic_inch,
+     "Gewicht"    = mtcars$wt * pound * 1000
+  )
R> head(autos)
```

```
                Typ Verbrauch Hubraum Gewicht
1         Mazda RX4     11.20    2622    1188
2     Mazda RX4 Wag     11.20    2622    1304
3        Datsun 710     10.32    1770    1052
4    Hornet 4 Drive     10.99    4228    1458
5 Hornet Sportabout     12.58    5899    1560
6           Valiant     13.00    3687    1569
```



```
    if ( verbose ) cat("foehnix.family object probided: use custom family object.\n")
} else if ( inherits(family, "character") ) {
    family <- match.arg(family, c("gaussian", "logistic"))
    if ( ! all(is.infinite(c(left, right))) ) {
        # Take censored version of "family" using the censoring
        # thresholds left and right.
        if ( ! truncated ) {
            family <- get(sprintf("foehnix_c%s", family))(left = left, right = right)
        # Else take the truncated version of the "family".
        } else {
            family <- get(sprintf("foehnix_t%s", family))(left = left, right = right)
```

Subsetting

Fundamentals

Thorsten.Simon@uibk.ac.at

Simplifying and preserving subsetting

We saw the differeence of `[` and `[[` for lists

- `[` returns one or more elemets as a list
- `[[` returns the content of one element

Thus we can distinguish between *simplifying* and *preserving* subsetting.

- **preserving** subsetting always returns an object of the same class.
- the returned object for **simplifying** subsetting varies between different inputs.

Simplifying and preserving subsetting

Operators for simplifying and preserving subsetting dependend on the input objects:

	Simplifying	Preserving
Vector	`x[[1]]`	`x[1]`
List	`x[[1]]`	`x[1]`
Factor	`x[1:4, drop = TRUE]`	`x[1:4]`
Matrix	`x[1, ]` **or** `x[, 1]`	`x[1, , drop = FALSE]` **or** `...`
Data frame	`x[, 1]` **or** `x[[1]]`	`x[, 1, drop = FALSE]` **or** `x[1]`

Simplifying and preserving subsetting

For **atomic vectors** simplifying subsetting removes names:

```
R> x <- c(a = 1, b = 2)
R> x[1]
```

```
a
1
```

```
R> x[[1]]
```

```
[1] 1
```

Simplifying and preserving subsetting

For **lists** simplifying subsetting returns the object inside the list, not a single element list:

```
R> x <- list(a = 1, b = 2)
R> str(x[1])
```

```
List of 1
 $ a: num 1
```

```
R> str(x[[1]])
```

```
 num 1
```

Simplifying and preserving subsetting

For **factors** simplifying subsetting drops unused levels:

```
R> x <- factor(c("a", "b"))
R> x[1]
```

```
[1] a
Levels: a b
```

```
R> x[1, drop = TRUE]
```

```
[1] a
Levels: a
```

Simplifying and preserving subsetting

Matrices: If any of the dimensions has length 1, simplifying subsetting drops that dimension.

```
R> m <- matrix(1:4, nrow = 2)
R> m[1, , drop = FALSE]
```

```
     [,1] [,2]
[1,]    1    3
```

```
R> m[1, ]
```

```
[1] 1 3
```

Simplifying and preserving subsetting

Data frame: If output is a single column, simplifying subsetting returns a vector instead of a data frame.

```
R> d <- data.frame(a = 1:2, b = 3:4)
R> str(d[1])
```

```
'data.frame':	2 obs. of  1 variable:
 $ a: int  1 2
```

```
R> str(d[[1]])
```

```
 int [1:2] 1 2
```

```
R> str(d[, "a", drop = FALSE])
```

```
'data.frame':	2 obs. of  1 variable:
 $ a: int  1 2
```

```
R> str(d[, "a"])
```

```
 int [1:2] 1 2
```

Subsetting and assignment

All subsetting operators can be combined with assignment <- to modify selected values of the input vector.

```
R> x <- 1:5
R> x[c(1, 2)] <- c(-999, -999)
R> x
```

```
[1] -999 -999    3    4    5
```

Example: Data handling

Load the data `gender_score.rds`:

```
R> d <- readRDS("../../data/gender_score.rds")
R> head(d, 3)
```

```
  index gender score
1    14   male   6.3
2    19   male   7.9
3    15   male   7.3
```

- Sort the rows in the data `d` by the column `index`.
- Select all rows, for which the `gender` is `male`.
- Convert the numeric `score` into a categorical score (name the column `cscore`), where all value below or equal to 3 fall into the category `low`, between 3 and 7 into `mid`, and above 7 into `high`.

Example: Data handling

```
R> ## --- sort by index ---
R> d <- d[order(d$index), ]
R> ## --- select males ---
R> d <- subset(d, gender == "male")
R> ## --- convert score to categorical score ---
R> d$cscore <- cut(d$score,
+    breaks = c(0, 3, 7, 10),
+    labels = c("low", "mid", "high"),
+    include.lowest = TRUE,
+    ordered_result = TRUE
+  )
R> head(d)
   index gender score cscore
9      1   male   5.8    mid
14     2   male   4.7    mid
5      3   male   0.3    low
4      5   male   5.9    mid
7      8   male  10.0   high
18     9   male   2.3    low
```