Data Manipulation

Arthur Allignol

arthur.allignol@uni-ulm.de

Table of Contents

Factors

- Factor are variables in R that take on a limited number of different values
  - Categorical variables
  - Ordinal variables
- Factors are useful for statistical modelling as ordinal variables should be treated differently than continuous variables
- Factors are also useful for statistical report generation. Think SAS labels

Factors

- Factors are stored internally as numeric values
- A corresponding set of characters is used for displaying

```
aa <- factor(c("cats", "dogs", "apples"))
aa

## [1] cats   dogs   apples
## Levels: apples cats dogs

as.integer(aa)

## [1] 2 3 1
```

Factor Creation

- Factors are created using the `factor` function
- The `levels` argument permits to control the order
- The `labels` argument is used to change the levels' names
- `ordered = TRUE` creates an ordered factor (ordinal variable)

```r
set.seed(21324)
data <- sample(c(1, 2, 3), 10, TRUE)
f0 <- factor(data)
f1 <- factor(data, levels = c(2, 3, 1))
f2 <- factor(data, labels = c("I", "II", "III"))
f3 <- factor(data, levels = c(2, 3, 1),
             labels = c("II", "III", "I"))
```

```
table(f0)

## f0
## 1 2 3
## 4 3 3
```

```
table(f1)

## f1
## 2 3 1
## 3 3 4
```

```
table(f2)

## f2
##   I  II III
##   4   3   3
```

```
table(f3)

## f3
##  II III   I
##   3   3   4
```

Factors

- The `levels()` function can be used to change the labels once a factor has been created

```
levels(f0) <- c("I", "II", "III")
f0

##  [1] II  I   III III III I   I   II  I   II
## Levels: I II III
```

- The reference level of a factor can be changed using the `relevel` function

```
f0 <- relevel(f0, "II")
f0

##  [1] II  I   III III III I   I   II  I   II
## Levels: II I III
```

When Factors Are a PITA

```
set.seed(423423)
ff <- factor(sample(1:4, 10, TRUE))
```

```
mean(ff)

## Warning in mean.default(ff):  argument is not numeric or logical:
returning NA

## [1] NA

ff + 10

## Warning in Ops.factor(ff, 10):  '+' not meaningful for factors

##  [1] NA NA NA NA NA NA NA NA NA NA

c(ff, 10) # Not a factor anymore

##  [1]  1  2  1  1  2  1  3  1  2  3 10
```

When Factors Are a PITA

```
(a <- factor(sample(letters, 10, replace = TRUE)))

##  [1] a u o j i h d g n d
## Levels: a d g h i j n o u

(b <- factor(sample(letters, 10, replace = TRUE)))

##  [1] t y k g v k p b d d
## Levels: b d g k p t v y
```

```
c(a, b)

##  [1] 1 9 8 6 5 4 2 3 7 2 6 8 4 3 7 4 5 1 2 2
```

When Factors Are a PITA

```
(a <- factor(sample(letters, 10, replace = TRUE)))

##  [1] a u o j i h d g n d
## Levels: a d g h i j n o u

(b <- factor(sample(letters, 10, replace = TRUE)))

##  [1] t y k g v k p b d d
## Levels: b d g k p t v y
```

```
c(a, b)

##  [1] 1 9 8 6 5 4 2 3 7 2 6 8 4 3 7 4 5 1 2 2
```

```
factor(c(as.character(a), as.character(b)))

##  [1] a u o j i h d g n d t y k g v k p b d d
## Levels: a b d g h i j k n o p t u v y
```

Factors

- Pros
  - Needed for modelling categorical variable
  - Memory efficient, i.e., factors only need to store values as integer and the unique levels as character strings
  - Nice output

```
table(factor(c(1, 2, 3),
             labels = c("Healthy", "Diseased", "Dead")))

##
##  Healthy Diseased     Dead
##        1        1        1
```

- Cons
  - Require to be cautious for some data manipulation
- I'd recommend reading data using the option `stringsAsFactors=FALSE` and transform variables into factors as needed

Dates

R provides several options to deal with dates, which is a challenging problem, i.e., time zones, daylight savings, leap second, . . .

- `as.Date` handles dates without time
- The **chron** package handles dates and times, but without support for time zones
- The `POSIXct` and `POSIXlt` allow for dates and times with control for time zones
- The **lubridate** packages is supposed to facilitate the use of dates and times in R

Rule of thumb: Use the simplest technique possible. If you only have dates, use `as.Date`

Dates

as.Date

- `as.Date` accepts a variety of input style through the `format` argument
- Default is `yyyy-mm-dd`

```r
as.Date("2014-06-12")
```

```
## [1] "2014-06-12"
```

```r
as.Date("12.6.2014", format = "%d.%m.%Y")
```

```
## [1] "2014-06-12"
```

```r
as.Date("12 June 14", format = "%d %B %y")
```

```
## [1] "2014-06-12"
```

See `?strptime` for a complete list of format symbols

Dates

as.Date

- Internally, dates are stored as the number of days since January 1, 1970
- as.numeric can be used to convert a date to its numeric form

```r
as.integer(as.Date("2014-06-12"))

## [1] 16233
```

- The weekdays and months functions can be used to extract the dates' components
- Calculation on dates: See ?Ops.Date. Addition, subtraction, logical operations (==, <, ...) are available

Table of Contents

Introduction

Character: A symbol in a written language, e.g, letters, numbers, punctuation marks, space, newlines, . . .

String: A sequence of character bound together

Note that R does not distinguish between character and string

```
test <- "a" # or 'a'
test2 <- "apple" # or 'apple'
class(test)

## [1] "character"

class(test2)

## [1] "character"
```

Substrings

The `substr` permits to extract and/or replace substrings

```
# Extract
my_string <- "cats don't like dogs"
substr(my_string, start = 6, stop = 15)

## [1] "don't like"
```

```
# Works with vectors
my_vector <- c("cats", "dogs", "apple")
substr(my_vector, 2, 2)

## [1] "a" "o" "p"
```

Split Strings into Vectors

The `strsplit` function permits to split a string into a list containing multiple strings based on a given delimiter

```
another_string <- "cats, dogs and apples"
strsplit(another_string, split = ",")

## [[1]]
## [1] "cats"             " dogs and apples"

strsplit(another_string, split = " ")[[1]]

## [1] "cats,"  "dogs"   "and"    "apples"
```

```
yet_another_string <- "walk into a bar"
strsplit(c(another_string, yet_another_string), split = " ")

## [[1]]
## [1] "cats,"  "dogs"   "and"    "apples"
##
## [[2]]
## [1] "walk" "into" "a"    "bar"
```

Build Strings from Multiple Parts

The paste function

The `paste` function combines multiple strings into a single strings. The `sep` and `collapse` arguments control the separation.

```
paste(c("cats", "dogs", "apple"), collapse = "|") # BUT

## [1] "cats|dogs|apple"

paste(c("cats", "dogs", "apple"), sep = "|")

## [1] "cats"  "dogs"  "apple"

# collapse permits to concatenate strings from a single vector
```

```
paste("cats", "dogs", "apple", sep = "|")

## [1] "cats|dogs|apple"
```

Search and Replace

R provides several functions for searching and replacing text

`grep`	Search for `pattern` in a vector x and return the indices of matches or matching string (`value = TRUE`)
`grepl`	As grep but returns a logical vector
`regexpr`	Return character position of the first match as well as length of the match. -1 is returned if no match
`gregexpr`	As `regexpr` but reports all matches
`regexec`	Comparable to `regexpr` but returns a list
`sub`	Finds pattern in text and replaces first match with specified string
`gsub`	As `sub` but replaces all matches

Simple Matching

```
l <- c("apple", "banana", "grape", "10", "green.pepper")
grep(pattern = "a", x = l)

## [1] 1 2 3

grep(pattern = "a", x = l, value = TRUE)

## [1] "apple"  "banana" "grape"

grepl("a", l)

## [1]  TRUE  TRUE  TRUE FALSE FALSE
```

Table of Contents

1. Factors and Dates
2. Text Processing
3. Data Manipulation
4. Data Aggregation
5. Data Reshaping
6. Combining and Merging

Lists

Lists are the most general R object.

```
(ll <- list(a = 1:3, b = month.name[1:5], c = c(TRUE, FALSE),
            d = data.frame(y = rnorm(5), x = rbinom(5, 1, .5))))
```

```
## $a
## [1] 1 2 3
##
## $b
## [1] "January"  "February" "March"    "April"    "May"
##
## $c
## [1]  TRUE FALSE
##
## $d
##            y x
## 1 -1.1065764 0
## 2  1.6957258 0
## 3 -1.0641906 1
## 4 -0.0415854 1
## 5  0.8534742 0
```

Lists

```
class(ll[[4]]); class(ll[["d"]]); class(ll$d)

## [1] "data.frame"
## [1] "data.frame"
## [1] "data.frame"

class(ll[4])

## [1] "list"
```

```
ll[c(1, 3)]

## $a
## [1] 1 2 3
##
## $c
## [1]  TRUE FALSE
```

Subscripting Data Frames

```
set.seed(4234234)
(df <- data.frame(x = c(rnorm(3), NA, 3),
                 y = c(NA, rexp(2, 0.01), NA, 3)))
```

```
##            x         y
## 1  1.7547348        NA
## 2 -0.3676785 108.34508
## 3 -1.5529115  85.43826
## 4         NA        NA
## 5  3.0000000   3.00000
```

```
df$x
```

```
## [1]  1.7547348 -0.3676785 -1.5529115         NA  3.0000000
```

```
df[, "x", drop = FALSE]
```

```
##            x
## 1  1.7547348
## 2 -0.3676785
## 3 -1.5529115
```

Subscripting Data Frames

```
df[df$y > 10, ]
```

```
##                x         y
## NA            NA        NA
## 2    -0.3676785 108.34508
## 3    -1.5529115  85.43826
## NA.1          NA        NA
```

```
df[!is.na(df$y) & df$y > 10, ]
```

```
##             x         y
## 2 -0.3676785 108.34508
## 3 -1.5529115  85.43826
```

```
subset(df, y > 10)
```

```
##             x         y
## 2 -0.3676785 108.34508
## 3 -1.5529115  85.43826
```

Subscripting Data Frames

Order a data frame

```
df[order(df$x), ]
```

```
##            x         y
## 3 -1.5529115  85.43826
## 2 -0.3676785 108.34508
## 1  1.7547348        NA
## 5  3.0000000   3.00000
## 4         NA        NA
```

Table of Contents

Data Aggregation

- For simple tabulation and cross-tabulation, the `table`, `ftable` and `xtabs` functions are available
- For more complex tasks, the available functions can be classified into two groups
  - Functions that operate on arrays and/or lists (e.g., `*apply`, `sweep`)
  - Functions oriented towards data frames (e.g., `aggregate`, `by`)

The `table` function

```
data(iris)

head(iris)

##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

The table function

```
table(iris$Species)
```

```
## 
##     setosa versicolor  virginica 
##         50         50         50 
```

```
table(iris$Species, iris$Petal.Length > 6)
```

```
## 
##              FALSE TRUE
##   setosa        50    0
##   versicolor    50    0
##   virginica     41    9
```

```
as.data.frame(table(iris$Species, iris$Petal.Length > 6))
```

```
##         Var1  Var2 Freq
## 1     setosa FALSE   50
## 2 versicolor FALSE   50
## 3  virginica FALSE   41
## 4     setosa  TRUE    0
## 5 versicolor  TRUE    0
```

Road Map for Aggregation

Three things to consider

1. How are the groups that divide the data defined?
2. What is the nature of the data to be operated on?
3. What is the desired end result

Groups Defined as Lists Elements

`sapply` or `lapply` are the appropriate functions

- `lapply` always returns a list
- `sapply` tries to “simplify” the output

Groups Defined as Lists Elements

`sapply` or `lapply` are the appropriate functions

- `lapply` always returns a list
- `sapply` tries to “simplify” the output

```
myList <- list()
for (i in 1:4) {
    myList[[i]] <- rnorm(n = 3 * i)
}
myList
```

```
## [[1]]
## [1]  0.3429579 -0.3193258  0.7808710
##
## [[2]]
## [1]  1.16866312  0.01419804  0.45813283 -0.43180622  0.34224696 -1.30745260
##
## [[3]]
## [1]  1.4005004 -1.7575754 -0.2415508  1.0928182 -1.1926425  1.8645074
## [7] -0.3128976  0.8755070 -1.9690762
##
## [[4]]
##  [1]  1.1415283 -1.1269112 -2.2914106  0.9855559 -1.4959317  2.2773178
##  [7]  0.5160894 -0.5481570 -0.6098171 -1.4302086  1.2207910 -1.7708338
```

Groups Defined as Lists Elements

Both for `lapply` and `sapply`, the first argument is a list, the second argument is a function
Third, fourth, . . . arguments are further arguments for the function that is applied

```
lapply(myList, length)

## [[1]]
## [1] 3
##
## [[2]]
## [1] 6
##
## [[3]]
## [1] 9
##
## [[4]]
## [1] 12
```

```
sapply(myList, length)

## [1]  3  6  9 12
```

Groups Defined as Lists Elements

```
myList[[2]][c(3, 5)] <- NA
sapply(myList, mean)

## [1]  0.26816768          NA -0.02671217 -0.26099896
```

```
sapply(myList, mean, na.rm = TRUE)

## [1]  0.26816768 -0.13909942 -0.02671217 -0.26099896
```

```
sapply(myList, quantile, probs = c(0.25, 0.75), na.rm = TRUE)

##          [,1]       [,2]      [,3]      [,4]
## 25% 0.0118160 -0.6507178 -1.192642 -1.446639
## 75% 0.5619144  0.3028143  1.092818  1.024549
```

Groups Defined as Lists Elements

```
mySummary <- function(x, na.rm = FALSE) {
    data.frame(
        Mean = mean(x, na.rm = na.rm),
        SD = sd(x, na.rm = na.rm),
        Min = min(x, na.rm = na.rm),
        Max = max(x, na.rm = na.rm))
}

sapply(myList, mySummary, na.rm = TRUE)

##      [,1]       [,2]       [,3]        [,4]
## Mean 0.2681677  -0.1390994 -0.02671217 -0.260999
## SD   0.5538984  1.030286   1.411432    1.446369
## Min  -0.3193258 -1.307453  -1.969076   -2.291411
## Max  0.780871   1.168663   1.864507    2.277318
```

Groups Defined by Rows or Columns of a Matrix/Array

In this case, the `apply` function is the logical choice.
The `apply` function requires three arguments

- the array/matrix on which to operate
- An index telling `apply` which dimension to operate on (1 on rows; 2 on columns, `c(1, 2)` on both
- The function to use
- Optionally further arguments to be used by the function that we want to apply

```
apply(iris[, 1:4], 2, mean)

## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##     5.843333     3.057333     3.758000     1.199333
```

Groups Defined by Rows or Columns of a Matrix/Array

```
apply(iris[, 1:4], 2, mySummary)

## $Sepal.Length
##       Mean        SD Min Max
## 1 5.843333 0.8280661 4.3 7.9
##
## $Sepal.Width
##       Mean        SD Min Max
## 1 3.057333 0.4358663   2 4.4
##
## $Petal.Length
##    Mean       SD Min Max
## 1 3.758 1.765298   1 6.9
##
## $Petal.Width
##       Mean        SD Min Max
## 1 1.199333 0.7622377 0.1 2.5
```

Groups Based on One or More Grouping Variables

A very common operation

A lot of choice in base R + a couple of additional packages that facilitates these operations

- `aggregate`
- `tapply`, `by`
- *split-apply-combine* strategy
  - `split`, `lapply`, `do.call`
  - **plyr**, **dplyr** package
  - ...

Groups Based on One or More Grouping Variables

aggregate

A natural choice for data summaries of several variables

- First argument: A formula
  - LHS: Variables to “summarise”
  - RHS: Grouping variables
- Second argument: A data frame
- Third argument: Function to apply
- . . . ; Further arguments for FUN

```r
iris$Petal.Length.f <- factor(iris$Petal.Length > 4.8,
                              levels = c(FALSE, TRUE),
                              labels = c("Small petals", "Big petals"))
aggregate(cbind(Sepal.Length, Sepal.Width) ~ Species + Petal.Length.f,
          data = iris, FUN = mean)
```

```
##      Species Petal.Length.f Sepal.Length Sepal.Width
## 1     setosa   Small petals     5.006000    3.428000
## 2 versicolor   Small petals     5.889130    2.765217
## 3  virginica   Small petals     5.700000    2.766667
## 4 versicolor     Big petals     6.475000    2.825000
## 5  virginica     Big petals     6.644681    2.987234
```

Groups Based on One or More Grouping Variables

Split-Apply-Combine

Term coined by Hadley Wickham (author of the **ggplot2**, **plyr**, **reshape**, **dplyr**, . . . , packages)

Split	Divide the problem into smaller pieces
Apply	Work on each pieces independently
Combine	Recombine the pieces

A common problem for both programming and data analysis; many implementations

- In base R: `split()`, `*apply()`, `do.call()`
- R-packages: **plyr**, **doBy**, **dplyr**, **data.table** (to some extent)

Split-Apply-Combine

Base R

- Split by species

```
s_iris <- split(iris, iris$Species)

## s_iris is a list with number of items
## equal to the number of levels of iris$Species
length(s_iris) == length(levels(iris$Species))

## [1] TRUE
```

- Apply a function to each item of the list

```
s_means <- lapply(s_iris, function(x) colMeans(x[1:4]))
s_means[[1]]

## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##        5.006        3.428        1.462        0.246
```

- Combine

```
(res <- do.call(rbind, s_means))

##            Sepal.Length Sepal.Width Petal.Length Petal.Width
## setosa            5.006       3.428        1.462       0.246
## versicolor        5.936       2.770        4.260       1.326
## virginica         6.588       2.974        5.552       2.026
```

The **dplyr** package

The **dplyr** package proposes a “grammar of data manipulation”, i.e., it implements “verbs” useful for data manipulation.

`select`	column subset (select variables)
`filter`	row subset (⇔ `subset` in base R)
`mutate`	add new/modify rows
`summarise`	summary statistics
`arrange`	re-order the rows
`do`	arbitrary action

- **dplyr** supports data.frames, data.tables (see later) as well as data bases
- Operations can be chained using a pipe operator

The **dplyr** package

Compute the mean sepal width by species for flower whose petal length is longer than 4.8

```
iris %>% group_by(Species) %>% filter(Petal.Length > 4.8) %>%
  summarise(mean_width = mean(Petal.Width))

## Source: local data frame [2 x 2]
##
##      Species mean_width
##       (fctr)      (dbl)
## 1 versicolor   1.575000
## 2  virginica   2.042553
```

The **data.table** package

The **data.table** package enhances the base data.frame. The package offers (extremely) fast

- subset
- grouping
- update
- joints (merging)

A `data.table` inherits from `data.frame`, i.e., it is compatible with R functions and packages that only accept `data.frame`.

The **data.table** package

The general syntax is

```
dt[i, j, by]
```

`i` permits to select rows (A bit like `subset`)

`j` permits to update/create columns. Extremely flexible (maybe too much?)

`by` permits to “group by”

Additionally, data.tables can be *keyed* by one or more variables, leading to

- ordered data
- faster merging by keyed variables

The **data.table** package

Subset rows in `i`

```
require(data.table)
(dt_iris <- data.table(iris, key = "Species"))

##      Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
##   1:          5.1         3.5          1.4         0.2    setosa
##   2:          4.9         3.0          1.4         0.2    setosa
##   3:          4.7         3.2          1.3         0.2    setosa
##   4:          4.6         3.1          1.5         0.2    setosa
##   5:          5.0         3.6          1.4         0.2    setosa
##  ---
## 146:          6.7         3.0          5.2         2.3 virginica
## 147:          6.3         2.5          5.0         1.9 virginica
## 148:          6.5         3.0          5.2         2.0 virginica
## 149:          6.2         3.4          5.4         2.3 virginica
## 150:          5.9         3.0          5.1         1.8 virginica
##      Petal.Length.f
##   1:   Small petals
##   2:   Small petals
##   3:   Small petals
##   4:   Small petals
##   5:   Small petals
```

The **data.table** package

Subset rows in `i`

```
dt_iris[Species == "versicolor" & Petal.Length > 4.8]

##    Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
## 1:          6.9         3.1          4.9         1.5 versicolor
## 2:          6.3         2.5          4.9         1.5 versicolor
## 3:          6.7         3.0          5.0         1.7 versicolor
## 4:          6.0         2.7          5.1         1.6 versicolor
##    Petal.Length.f
## 1:     Big petals
## 2:     Big petals
## 3:     Big petals
## 4:     Big petals
```

The **data.table** package

Select columns in j

```
dt_iris[1:3, Species]

## [1] setosa setosa setosa
## Levels: setosa versicolor virginica

dt_iris[Species == "versicolor" & Petal.Length > 4.8,
        list(Species, Petal.Length)]

##        Species Petal.Length
## 1: versicolor          4.9
## 2: versicolor          4.9
## 3: versicolor          5.0
## 4: versicolor          5.1
```

The **data.table** package

Compute in `j`: As long as `j-expressions` returns a list, each element of the list will be converted to a column

```
dt_iris[, mean(Petal.Length)]

## [1] 3.758

dt_iris[Species == "versicolor", mean(Petal.Length)]

## [1] 4.26
```

The **data.table** package

```
dt_iris[Species == "versicolor", list(mean = mean(Petal.Length),
        sd = sd(Petal.Length))]

##    mean       sd
## 1: 4.26 0.469911

## With a use defined function
myFun <- function(x) {
    list(mean = mean(x),
         sd = sd(x))
}
dt_iris[Species == "versicolor", myFun(Petal.Length)]

##    mean       sd
## 1: 4.26 0.469911
```

The **data.table** package

Group by using by

```
dt_iris[, .N, by = list(Species, Petal.Length.f)]

##       Species Petal.Length.f  N
## 1:     setosa   Small petals 50
## 2: versicolor   Small petals 46
## 3: versicolor     Big petals  4
## 4:  virginica     Big petals 47
## 5:  virginica   Small petals  3

dt_iris[, myFun(Sepal.Length), by = list(Species, Petal.Length.f)]

##       Species Petal.Length.f     mean        sd
## 1:     setosa   Small petals 5.006000 0.3524897
## 2: versicolor   Small petals 5.889130 0.5012111
## 3: versicolor     Big petals 6.475000 0.4031129
## 4:  virginica     Big petals 6.644681 0.5955664
## 5:  virginica   Small petals 5.700000 0.7000000
```

The **data.table** package

Reorder the last output by `Species` and `Petal.Length.f`

```
tmp <- dt_iris[, myFun(Sepal.Length), by = list(Species, Petal.Length.f)]
(setkeyv(tmp, c("Species", "Petal.Length.f")))

##       Species Petal.Length.f     mean        sd
## 1:     setosa   Small petals 5.006000 0.3524897
## 2: versicolor   Small petals 5.889130 0.5012111
## 3: versicolor     Big petals 6.475000 0.4031129
## 4:  virginica   Small petals 5.700000 0.7000000
## 5:  virginica     Big petals 6.644681 0.5955664
```

Update/create column with the := operator

The := operator adds or update columns by reference

```
dt_iris[, new_variable := Petal.Length + 1]
```

Table of Contents

Data Reshaping

An important operation in R

- Most R functions expect their input (usually data frames) to be arranged in particular ways
- It is the responsibility of the user to ensure that the data are in the appropriate form
- For instance, data for multiple groups are organised as columns, with a column for each group
- Most R functions expect values to be in **one** column with an additional column specifying the groups

Long versus Wide Format

Useful concept for, e.g., *longitudinal studies*, in which a patient may have several measurements over time

Wide If all the measurements for a single individual are in the same row, the data are said to be **wide**

id	visit1	visit2
1	90	95
2	80	78

Long If each measurement is in a different row, the data are said to be in the **long** format

id	visit	measure
1	1	90
1	2	95
2	1	80
2	2	78

Most data sets are delivered in the wide format, modelling is done in the long format

The `reshape` Function

The `reshape` function performs the long → wide and wide → long transformations

- Motivated by longitudinal data (repeated measurements)
- Very flexible function (maybe too much)
- Google very useful for using this function

The `reshape` Function

Wide → Long Transformation

As an example, consider a data set on US personal expenditure

```
usp
```

```
##                    type  X1940  X1945 X1950 X1955 X1960
## 1     Food and Tobacco 22.200 44.500 59.60  73.2 86.80
## 2 Household Operation 10.500 15.500 29.00  36.5 46.20
## 3  Medical and Health  3.530  5.760  9.71  14.0 21.10
## 4       Personal Care  1.040  1.980  2.45   3.4  5.40
## 5   Private Education  0.341  0.974  1.80   2.6  3.64
```

The `reshape` Function

Wide → Long Transformation

Useful arguments for wide to long transformations

- `varying`: names of sets of variables in the wide format that correspond to single variables in long format. Can be a list of names (see later)
- `v.names`: The name we wish to give the variable containing these values in our long dataset
- `timevar`: The name we wish to give the variable describing the different times or metrics
- `times`: the values this variable will have
- `idvar`: Values describing the different individuals
- `direction`: Character string indicating the direction of the transformation; either `"wide"` or `"long"`
- `times`, `split`, `sep`

The reshape Function

Wide → Long Transformation

```
rr2 <- reshape(usp, varying = list(names(usp)[-1]), idvar = "type",
               times = seq(1940, 1960, 5), v.names = "expenditure",
               direction = "long")
head(rr2)
```

```
##                                         type time expenditure
## Food and Tobacco.1940       Food and Tobacco 1940      22.200
## Household Operation.1940 Household Operation 1940      10.500
## Medical and Health.1940   Medical and Health 1940       3.530
## Personal Care.1940             Personal Care 1940       1.040
## Private Education.1940     Private Education 1940       0.341
## Food and Tobacco.1945       Food and Tobacco 1945      44.500
```

The reshape Function

Wide → Long Transformation

```
rr3 <- reshape(usp, varying = names(usp)[-1], idvar = "type",
               times = seq(1940, 1960, 5), v.names = "expenditure",
               direction = "long")
head(rr3)

##                                          type time expenditure
## Food and Tobacco.1940       Food and Tobacco 1940      22.200
## Household Operation.1940 Household Operation 1940      10.500
## Medical and Health.1940   Medical and Health 1940       3.530
## Personal Care.1940             Personal Care 1940       1.040
## Private Education.1940     Private Education 1940       0.341
## Food and Tobacco.1945       Food and Tobacco 1945      44.500
```

The reshape Function

Wide → Long Transformation

```
rr3 <- reshape(usp, varying = names(usp)[-1], idvar = "type",
               times = seq(1940, 1960, 5), v.names = "expenditure",
               direction = "long")
head(rr3)

##                                           type time expenditure
## Food and Tobacco.1940       Food and Tobacco 1940      22.200
## Household Operation.1940 Household Operation 1940      10.500
## Medical and Health.1940   Medical and Health 1940       3.530
## Personal Care.1940             Personal Care 1940       1.040
## Private Education.1940     Private Education 1940       0.341
## Food and Tobacco.1945       Food and Tobacco 1945      44.500
```

Specifying a vector of names in `varying` now works because we also specify how the resulting variable should be named (`v.names`)

The reshape Function

Wide → Long Transformation

The sep argument is sometimes useful to help reshape automagically find the v.names

```
rr5 <- reshape(usp, varying = names(usp)[-1], idvar = "type",
               sep = "",
               direction = "long")
head(rr5)

##                                          type time      X
## Food and Tobacco.1940       Food and Tobacco 1940 22.200
## Household Operation.1940 Household Operation 1940 10.500
## Medical and Health.1940   Medical and Health 1940  3.530
## Personal Care.1940             Personal Care 1940  1.040
## Private Education.1940     Private Education 1940  0.341
## Food and Tobacco.1945       Food and Tobacco 1945 44.500
```

The reshape Function

Wide → Long Transformation

Reshape()’d data have additional attributes so that the inverse transformation is easy

```
reshape(rr2)
```

```
##                                         type  X1940  X1945 X1950 X1955
## Food and Tobacco.1940       Food and Tobacco 22.200 44.500 59.60  73.2
## Household Operation.1940 Household Operation 10.500 15.500 29.00  36.5
## Medical and Health.1940   Medical and Health  3.530  5.760  9.71  14.0
## Personal Care.1940             Personal Care  1.040  1.980  2.45   3.4
## Private Education.1940     Private Education  0.341  0.974  1.80   2.6
##                          X1960
## Food and Tobacco.1940    86.80
## Household Operation.1940 46.20
## Medical and Health.1940  21.10
## Personal Care.1940        5.40
## Private Education.1940    3.64
```

The reshape Function

Long → Wide Transformation

```r
longdat <- data.frame(id = as.integer(mapply(rep, 1:3, 3)),
                      visit = rep(1:3, 3),
                      x = rnorm(9), y = rnorm(9))
longdat
```

```
##   id visit           x          y
## 1  1     1 -0.38732169 -1.9495348
## 2  1     2  0.82067539  1.9460650
## 3  1     3 -0.49831634  0.8050138
## 4  2     1 -0.80859026  1.0224776
## 5  2     2 -1.05940918  1.3847261
## 6  2     3 -0.01233044  1.3253185
## 7  3     1  0.84289345 -1.1346494
## 8  3     2  1.56222152  0.3455885
## 9  3     3 -1.44585913 -1.8670554
```

The `reshape` Function

Long → Wide Transformation

Arguments needed (beside the data set to reshape)

- `idvar`: names of variable that define the experimental units
- `v.names`: Variables that are used to create the multiple variables in the wide format
- `timevar` identifies the “time” variable for the repeated measurements
- `direction`: `"long"` or `"wide"`

The reshape Function

Long → Wide Transformation

```
widedat <- reshape(longdat, idvar = "id", v.names = c("x", "y"),
                   timevar = "visit", direction = "wide")
widedat
```

```
##   id        x.1       y.1        x.2       y.2         x.3        y.3
## 1  1 -0.3873217 -1.949535  0.8206754 1.9460650 -0.49831634  0.8050138
## 4  2 -0.8085903  1.022478 -1.0594092 1.3847261 -0.01233044  1.3253185
## 7  3  0.8428935 -1.134649  1.5622215 0.3455885 -1.44585913 -1.8670554
```

The reshape Function

Long → Wide Transformation

Wide to long transformation again easy from the reshape()’d data

```
reshape(widedat)
```

```
##     id visit           x          y
## 1.1  1     1 -0.38732169 -1.9495348
## 2.1  2     1 -0.80859026  1.0224776
## 3.1  3     1  0.84289345 -1.1346494
## 1.2  1     2  0.82067539  1.9460650
## 2.2  2     2 -1.05940918  1.3847261
## 3.2  3     2  1.56222152  0.3455885
## 1.3  1     3 -0.49831634  0.8050138
## 2.3  2     3 -0.01233044  1.3253185
## 3.3  3     3 -1.44585913 -1.8670554
```

Table of Contents

Combining Data Frames

At the most basic level, two or more data frames can be combined by rows using `rbind`, or by columns using `cbind`

rbind Data frames must have the same number of columns

cbind The data must have the same number of rows

```
d1 <- data.frame(x = rnorm(4), y = sample(letters, 4, replace = FALSE))
d2 <- data.frame(x = rnorm(4), y = sample(letters, 4, replace = FALSE))
d1


##            x y
## 1 -2.1928785 s
## 2  0.7911268 u
## 3 -1.1763532 l
## 4  1.2151311 b
```

Combining Data Frames

cbind

```
cbind(d1, d2)

##            x y           x y
## 1 -2.1928785 s -0.09692773 m
## 2  0.7911268 u -1.88955642 l
## 3 -1.1763532 l  0.50479514 b
## 4  1.2151311 b  1.15345513 e
```

Duplicate column names are not detected

Combining Data Frames

cbind

```
cbind(d1, d2)
```

```
##            x y           x y
## 1 -2.1928785 s -0.09692773 m
## 2  0.7911268 u -1.88955642 l
## 3 -1.1763532 l  0.50479514 b
## 4  1.2151311 b  1.15345513 e
```

Duplicate column names are not detected

```
cbind(d1, z = c(1, 2))
```

```
##            x y z
## 1 -2.1928785 s 1
## 2  0.7911268 u 2
## 3 -1.1763532 l 1
## 4  1.2151311 b 2
```

Smaller vectors/data are recycled

Combining Data Frames

rbind

For using `rbind`, names and classes of values to be joined must match

```
rbind(d1, d2)

##             x y
## 1 -2.19287849 s
## 2  0.79112680 u
## 3 -1.17635317 l
## 4  1.21513114 b
## 5 -0.09692773 m
## 6 -1.88955642 l
## 7  0.50479514 b
## 8  1.15345513 e
```

Combining Data Frames

rbind

```
d1$y <- factor(d1$y)
d2$y <- factor(d2$y)
rbind(d1, d2)

##             x y
## 1 -2.19287849 s
## 2  0.79112680 u
## 3 -1.17635317 l
## 4  1.21513114 b
## 5 -0.09692773 m
## 6 -1.88955642 l
## 7  0.50479514 b
## 8  1.15345513 e
```

It works!

Combining Data Frames

rbind

```
(d3 <- rbind(d1, data.frame(x = "X", y = 12)))

## Warning in ‘[<-.factor‘(‘*tmp*‘, ri, value = structure(c(3L, 4L, 2L, 1L, :  invalid factor
level, NA generated

##                    x    y
## 1 -2.19287849290542    s
## 2 0.791126796699532    u
## 3 -1.17635316755859    l
## 4   1.2151311364279    b
## 5                  X <NA>

sapply(d3, class)

##           x           y
## "character"    "factor"
```

```
rbind(d1, data.frame(y = "X", d = 12))

## Error in match.names(clabs, names(xi)):  names do not match previous
names
```

Merge Data Frames

For more complicated tasks, the `merge` function can be used

- The default behaviour of merge is to join together rows of the data frames based on the values of all of the variables (columns) that the data frames have in common (*natural join*)
- When called without argument, `merge` only returns rows which have observations in both data frames

```
dd1 <- data.frame(a = c(1,2,4,5,6), x = c(9,12,14,21,8))
dd2 <- data.frame(a=c(1,3,4,6),y=c(8,14,19,2))
merge(dd1, dd2)

##   a  x  y
## 1 1  9  8
## 2 4 14 19
## 3 6  8  2
```

Merge Data Frames

To change the default behaviour the arguments

- `all = TRUE`: Includes all rows (*full outer join*)
- `all.x = TRUE`: Includes all rows of the first data frame (*left outer join*)
- `all.y = TRUE`: Includes all rows of the second data frame (*right outer join*)

Merge Data Frames

- The by argument permits to specify the name of the variables that should be used for the merging.
- If the merging variables have different names in the data frames to merge, the by.x and by.y arguments can be used

```
dd1$PAT <- letters[1:5]
dd2$id <- letters[3:6]
merge(dd1, dd2, by.x = c("PAT"), by.y = c("id"))
```

```
##   PAT a.x  x a.y  y
## 1   c   4 14   1  8
## 2   d   5 21   3 14
## 3   e   6  8   4 19
```

Merge Data Frames

- The by argument permits to specify the name of the variables that should be used for the merging.
- If the merging variables have different names in the data frames to merge, the by.x and by.y arguments can be used

```
dd1$PAT <- letters[1:5]
dd2$id <- letters[3:6]
merge(dd1, dd2, by.x = c("PAT"), by.y = c("id"))

##   PAT a.x  x a.y  y
## 1   c   4 14   1  8
## 2   d   5 21   3 14
## 3   e   6  8   4 19
```

Note the new variables a.x and a.y

Merge with **dplyr**

dplyr includes some functions for merging data sets

`inner_join` Equivalent to `merge` without arguments

`left_join` Equivalent to `merge` with `all.x = TRUE`

`right_join` Equivalent to `merge` with `all.y = TRUE`

`full_join` Equivalent to `merge` with `all = TRUE`

```
left_join(dd1, dd2, by = c("PAT" = "id"))

##   a.x  x PAT a.y  y
## 1   1  9   a  NA NA
## 2   2 12   b  NA NA
## 3   4 14   c   1  8
## 4   5 21   d   3 14
## 5   6  8   e   4 19
```

Merge with **data.table**

- A `merge` function is available in the **data.table** package.
- It works in the same way as the base function