Introduction to R Programming

Data Wrangling

Pedro Fonseca

02 Maio 2020

Built in datasets

R has many built in datasets. For a complete list see:

```
library(help = "datasets")
```

In this lecture we will use some of these datasets, namely:

- airquality
- USArrests

For information about a specific dataset see, for example:

```
?airquality
```

The `head()` and `tails()` functions

`head()` shows the first rows of a dataframe. `tail()` shows the last rows. Both `head()` and `tails()` print six rows by default.

```
nrow(airquality)
```

```
## [1] 153
```

```
head(airquality)
```

```
##   Ozone Solar.R Wind Temp Month Day
## 1    41     190  7.4   67     5   1
## 2    36     118  8.0   72     5   2
## 3    12     149 12.6   74     5   3
## 4    18     313 11.5   62     5   4
## 5    NA      NA 14.3   56     5   5
## 6    28      NA 14.9   66     5   6
```

The head() and tails() functions

```r
head(airquality, n = 2)
```

```
##   Ozone Solar.R Wind Temp Month Day
## 1    41     190  7.4   67     5   1
## 2    36     118  8.0   72     5   2
```

```r
tail(airquality, n = 3)
```

```
##     Ozone Solar.R Wind Temp Month Day
## 151    14     191 14.3   75     9  28
## 152    18     131  8.0   76     9  29
## 153    20     223 11.5   68     9  30
```

Built in constants

R also provides some built in constants and vectors:

```r
head(letters)
```

```
## [1] "a" "b" "c" "d" "e" "f"
```

```r
tail(letters)
```

```
## [1] "u" "v" "w" "x" "y" "z"
```

```r
head(LETTERS)
```

```
## [1] "A" "B" "C" "D" "E" "F"
```

```r
tail(LETTERS)
```

```
## [1] "U" "V" "W" "X" "Y" "Z"
```

For a complete list of built in constants see `?Constants`

The `subset()` function

`subset()` is a generic function that can be used to subset data frames using logical conditions.

The subset() function

```
df <-  data.frame(
  name = c("Yu","Matt","Jane","Tim", "Dave", "Marie"),
  inc = c(6, 1, 2, NA, 5, 9),
  gender = factor(c("F", "M","F","M", "M", "F")),
  state = factor(c("AZ","KS", NA, "CA","FL", "MA")))

df
```

```
##    name inc gender state
## 1    Yu   6      F    AZ
## 2  Matt   1      M    KS
## 3  Jane   2      F  <NA>
## 4   Tim  NA      M    CA
## 5  Dave   5      M    FL
## 6 Marie   9      F    MA
```

The subset() function

```r
subset(df, inc > 4)
```

```
##    name inc gender state
## 1    Yu   6      F    AZ
## 5  Dave   5      M    FL
## 6 Marie   9      F    MA
```

```r
subset(df, name == "Marie")
```

```
##    name inc gender state
## 6 Marie   9      F    MA
```

```r
subset(df, inc > 4 & name != "Marie")
```

```
##   name inc gender state
## 1   Yu   6      F    AZ
## 5 Dave   5      M    FL
```

The subset() function

```r
subset(df, inc > 4 & name != "Marie" & gender == "F")
```

```
##   name inc gender state
## 1   Yu   6      F    AZ
```

```r
subset(df, inc >= 2 & state %in% c("MA", "FL", "CA"))
```

```
##    name inc gender state
## 5  Dave   5      M    FL
## 6 Marie   9      F    MA
```

The subset() function

You can use the select argument to choose columns:

```
subset(df, inc == 5, select = c(state, name))
```

```
##   state name
## 5    FL Dave
```

```
subset(df, inc == 5, select = c(1, 4))
```

```
##   name state
## 5 Dave    FL
```

```
subset(df, inc == 5, select = inc:state)
```

```
##   inc gender state
## 5   5      M    FL
```

The subset() function

And also to drop columns:

```
subset(df, inc == 5, select = -c(state, name))
```

```
##   inc gender
## 5   5      M
```

```
subset(df, inc == 5, select = -c(state, name, gender))
```

```
##   inc
## 5   5
```

The subset() function

You can use subset() to filter out missing data with respect to specific variables:

```
subset(df, !is.na(state), select = c(name, inc))
```

```
##    name inc
## 1    Yu   6
## 2  Matt   1
## 4   Tim  NA
## 5  Dave   5
## 6 Marie   9
```

The subset() function

```
subset(df, !is.na(inc) & !is.na(state),
       select = c(name, inc, state))
```

```
##    name inc state
## 1    Yu   6    AZ
## 2  Matt   1    KS
## 5  Dave   5    FL
## 6 Marie   9    MA
```

The `subset()` function

`subset()`:

- Also works with vectors, matrices and lists.
- Doesn't drop dimensions (by default).

In the logical expressions that indicate which rows to keep, missing values are taken as `FALSE`.

Modifying columns with `transform()`

`transform()` can be used to modify the columns of a data frame:

```
transform(df, state = paste0(state, "-US"))
```

```
##    name inc gender state
## 1    Yu   6      F AZ-US
## 2  Matt   1      M KS-US
## 3  Jane   2      F NA-US
## 4   Tim  NA      M CA-US
## 5  Dave   5      M FL-US
## 6 Marie   9      F MA-US
```

Modifying columns with `transform()`

Let's change how the levels of the `gender` factor are displayed:

```r
transform(df, gender = factor(
  gender, labels = c("Female", "Male")))
```

```
##    name inc gender state
## 1    Yu   6 Female    AZ
## 2  Matt   1   Male    KS
## 3  Jane   2 Female  <NA>
## 4   Tim  NA   Male    CA
## 5  Dave   5   Male    FL
## 6 Marie   9 Female    MA
```

Modifying columns with transform()

Now let's express inc in euros:

```
transform(df, inc = ifelse(is.na(inc), NA,
                           paste0(inc * 1000, "€")
                           )
         )
```

```
##    name   inc gender state
## 1    Yu 6000€      F    AZ
## 2  Matt 1000€      M    KS
## 3  Jane 2000€      F  <NA>
## 4   Tim  <NA>      M    CA
## 5  Dave 5000€      M    FL
## 6 Marie 9000€      F    MA
```

Create columns with transform()

Transform() can also be used to create new variables.

Let's create a variable with income in the logarithmic scale:

```r
transform(df, logInc = log(inc))
```

```
##    name inc gender state   logInc
## 1    Yu   6      F    AZ 1.7917595
## 2  Matt   1      M    KS 0.0000000
## 3  Jane   2      F  <NA> 0.6931472
## 4   Tim  NA      M    CA        NA
## 5  Dave   5      M    FL 1.6094379
## 6 Marie   9      F    MA 2.1972246
```

Create columns with transform()

Now lets standardize the income column:

```
standardize <- function(x){
  z <- (x-mean(x, na.rm = TRUE))/sd(x, na.rm = TRUE)
  round(z, 2)
  }
```

```
transform(df, norm_inc = standardize(inc))
```

```
##    name inc gender state norm_inc
## 1    Yu   6      F    AZ     0.44
## 2  Matt   1      M    KS    -1.12
## 3  Jane   2      F  <NA>    -0.81
## 4   Tim  NA      M    CA       NA
## 5  Dave   5      M    FL     0.12
## 6 Marie   9      F    MA     1.37
```

The USArrests dataset

Now consider the USArrests dataset, with arrests per 100.000 residents in the US for murder, assault and rape.

```r
head(USArrests)
```

```
##            Murder Assault UrbanPop Rape
## Alabama      13.2     236       58 21.2
## Alaska       10.0     263       48 44.5
## Arizona       8.1     294       80 31.0
## Arkansas      8.8     190       50 19.5
## California    9.0     276       91 40.6
## Colorado      7.9     204       78 38.7
```

The USArrests dataset

```r
USArrests_short <- USArrests[1:4, -3]

USArrests_short
```

```
##          Murder Assault Rape
## Alabama    13.2     236 21.2
## Alaska     10.0     263 44.5
## Arizona     8.1     294 31.0
## Arkansas    8.8     190 19.5
```

Row and column sums

```r
colSums(USArrests_short)
```

```
##  Murder Assault    Rape
##    40.1   983.0   116.2
```

```r
rowSums(USArrests_short)
```

```
##  Alabama   Alaska  Arizona Arkansas
##    270.4    317.5    333.1    218.3
```

Row and column means

```
colMeans(USArrests_short)
```

```
##  Murder Assault    Rape
##  10.025 245.750  29.050
```

```
rowMeans(USArrests_short)
```

```
##   Alabama    Alaska   Arizona  Arkansas
##  90.13333 105.83333 111.03333  72.76667
```

The `apply()` function

To collapse data frames across rows or columns using functions other than the sum and the mean we can use apply():

```
apply(X, MARGIN, FUN, ...)
```

- X is a data frame
- `MARGIN = 1` for rows, `MARGIN = 2` for columns
- FUN is a function
- `...` are optional arguments to pass to FUN

The apply() function

Apply functions over the columns of `USArrests_short`:

```
apply(USArrests_short, 2, mean)
```

```
##  Murder Assault    Rape
##  10.025 245.750  29.050
```

```
apply(USArrests_short, 2, median)
```

```
##  Murder Assault    Rape
##     9.4   249.5    26.1
```

```
apply(USArrests_short, 2, sd)
```

```
##    Murder   Assault      Rape
##  2.257395 44.078528 11.479402
```

The apply() function

Apply functions the rows of `USArrests_short`:

```
apply(USArrests_short, 1, max)
```

```
##  Alabama   Alaska  Arizona Arkansas
##      236      263      294      190
```

```
apply(USArrests_short, 1, min)
```

```
##  Alabama   Alaska  Arizona Arkansas
##     13.2     10.0      8.1      8.8
```

```
apply(USArrests_short, 1, var)
```

```
##  Alabama   Alaska  Arizona Arkansas
## 15973.81 18823.58 25238.70 10336.36
```

apply() vs for loop

Loop over the columns of `USArrests_short`:

```
res <- vector()

for(i in 1:ncol(USArrests_short)){

  res[i] <- mean(USArrests_short[[i]], na.rm = TRUE)
  names(res)[i] <- names(USArrests_short)[i]
}

res
```

```
##   Murder Assault    Rape
##   10.025 245.750  29.050
```

apply() vs for loop

Loop over the rows of USArrests_short:

```
res <- vector()

for(j in 1:nrow(USArrests_short)){

  res[j] <- max(USArrests_short[j, ],na.rm = TRUE)
  names(res)[j] <- rownames(USArrests_short)[j]
}

res
```

```
##   Alabama    Alaska   Arizona Arkansas
##       236       263       294      190
```

apply() with ...

Now let's see an example that requires using dot-dot-dot (...).

Try to use apply() to compute the means of the first four columns of the airquality dataset:

```r
head(airquality)
```

```
##   Ozone Solar.R Wind Temp Month Day
## 1    41     190  7.4   67     5   1
## 2    36     118  8.0   72     5   2
## 3    12     149 12.6   74     5   3
## 4    18     313 11.5   62     5   4
## 5    NA      NA 14.3   56     5   5
## 6    28      NA 14.9   66     5   6
```

apply() with ...

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

```
##     Ozone   Solar.R      Wind      Temp
##        NA        NA  9.957516 77.882353
```

We get `NA` for the first two columns. Why?

```
sum(is.na(airquality$Ozone))
```

```
## [1] 37
```

```
sum(is.na(airquality$Solar.R))
```

```
## [1] 7
```

apply() with ...

Problem: Some columns have NAs.

Solution: use ... to pass the na.rm argument to mean():

```
apply(airquality[, 1:4], 2, mean, na.rm = TRUE)
```

```
##      Ozone    Solar.R       Wind       Temp
##  42.129310 185.931507   9.957516  77.882353
```

The `lapply()` function

Syntax: `lapply(X, FUN, ...)`

`lapply()` is similar to `apply()` but:

- X is a vector (atomic or list)
- There is no MARGINS argument
- Always returns a list

`lapply()` returns a list of the same length as X, each element of which is the result of applying FUN to the corresponding element of X.

The lapply() function

```r
lapply(c(1:3), log, base = 10)
```

```
## [[1]]
## [1] 0
##
## [[2]]
## [1] 0.30103
##
## [[3]]
## [1] 0.4771213
```

The `lapply()` function

```r
A <- matrix(1:10, ncol = 5)
B <- matrix(c(1, 5, 7, -1), ncol = 4)
C <- matrix(letters[1:4], ncol = 2)

my_list <- list(A, B, C)
```

The `lapply()` function

```
my_list
```

```
## [[1]]
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    3    5    7    9
## [2,]    2    4    6    8   10
##
## [[2]]
##      [,1] [,2] [,3] [,4]
## [1,]    1    5    7   -1
##
## [[3]]
##      [,1] [,2]
## [1,] "a"  "c"
## [2,] "b"  "d"
```

The lapply() function

Every element of my_list, except the last, contain a numerical matrix. Sum the elements of each of those matrices:

```
lapply(my_list[-3], sum)
```

```
## [[1]]
## [1] 55
##
## [[2]]
## [1] 12
```

The `lapply()` function

Extract the element in position (1, 2) from each matrix:

```
lapply(my_list,"[", 1, 2)
```

```
## [[1]]
## [1] 3
##
## [[2]]
## [1] 5
##
## [[3]]
## [1] "c"
```

The `lapply()` function

Extract the first row from each matrix:

```
lapply(my_list,"[", 1 , )
```

```
## [[1]]
## [1] 1 3 5 7 9
##
## [[2]]
## [1]  1  5  7 -1
##
## [[3]]
## [1] "a" "c"
```

The `lapply()` function

Extract the 2nd column from each matrix:

```r
lapply(my_list,"[", , 2)
```

```
## [[1]]
## [1] 3 4
##
## [[2]]
## [1] 5
##
## [[3]]
## [1] "c" "d"
```

lapply() vs for loop

Extract the 2nd column from each matrix:

```r
res <- vector(mode = "list")

for(i in seq_along(my_list)){
  res[[i]] <- my_list[[i]][, 2]
}
res
```

```
## [[1]]
## [1] 3 4
##
## [[2]]
## [1] 5
##
## [[3]]
## [1] "c" "d"
```

The sapply() function

The sapply() function:

- Works like lapply(), but simplifies the output to the most elementary data structure that is possible.
- Returns vectors or matrices.

```
sapply(my_list[-3], sum)
```

```
## [1] 55 12
```

```
sapply(my_list,"[", 1, 2)
```

```
## [1] "3" "5" "c"
```

The sapply() function

```r
set.seed(123)

our_list <- list(
  w = 1:6,
  x = sample(1:5, 4, replace = TRUE),
  y = matrix(sample(1:100, 9), nrow = 3),
  z = sample(1:10, 3, replace = TRUE)
)

str(our_list)
```

```
## List of 4
##  $ w: int [1:6] 1 2 3 4 5 6
##  $ x: int [1:4] 3 3 2 2
##  $ y: int [1:3, 1:3] 43 14 25 90 91 69 96 57 92
##  $ z: int [1:3] 9 9 3
```

The sapply() function

```
sapply(our_list, max)
```

```
##  w  x  y  z
##  6  3 96  9
```

```
sapply(our_list, min)
```

```
##  w  x  y  z
##  1  2 14  3
```

```
sapply(our_list, class)
```

```
##         w          x          y          z
## "integer" "integer"  "matrix" "integer"
```

The sapply() function

How many numbers are there inside each element of our_list?

```
sapply(our_list, length)
```

```
## w x y z
## 6 4 9 3
```

The sapply() function

```
set.seed(123)

our_list_2 <- list(
  w = 1:6,
  x = sample(1:5, 4, replace = TRUE),
  y = airquality
)
```

```
sapply(our_list_2, class)
```

```
##             w            x            y
##     "integer"    "integer" "data.frame"
```

```
dim(airquality)
```

```
## [1] 153   6
```

How many numbers are there inside each element of `our_list_2`?

The sapply() function

The length of a data frame is the number of columns, and hence `sapply(our_list_2, length)` won't do the trick.

```r
our_fun <- function(x){
  if(class(x) == "data.frame"){
    nrow(x) * ncol(x)
  }else{
      length(x)
  }
}

sapply(our_list_2, our_fun)
```

```
##   w   x   y
##   6   4 918
```

sapply() vs for loop

The same but with a for loop:

```
res <- vector()

for(i in seq_along(our_list_2)){

  res[i] <- our_fun(our_list_2[[i]])
  names(res)[i] <- names(our_list_2)[i]
}

res
```

```
##   w   x   y
##   6   4 918
```

The `mapply()` function

`mapply()` is a generalization of `sapply()`. It applies a multivariate function over multiple vectors of arguments.

The mapply() function

Suppose we want 3 samples of different sizes from a Normal$(0, 1)$ distribution:

```r
set.seed(123)

rnorm(n = 1)
```

```
## [1] -0.5604756
```

```r
rnorm(n = 2)
```

```
## [1] -0.2301775  1.5587083
```

```r
rnorm(n = 3)
```

```
## [1] 0.07050839 0.12928774 1.71506499
```

The mapply() function

The same result can be obtained more compactly with `mapply()`:

```r
set.seed(123)

sample_size <- 1:3
mapply(FUN = rnorm, n = sample_size)
```

```
## [[1]]
## [1] -0.5604756
##
## [[2]]
## [1] -0.2301775  1.5587083
##
## [[3]]
## [1] 0.07050839 0.12928774 1.71506499
```

The mapply() function

Since we only iterated over one vector, we could have used sapply():

```
set.seed(123)

sample_size <- 1:3
sapply(FUN = rnorm, X = sample_size)
```

```
## [[1]]
## [1] -0.5604756
##
## [[2]]
## [1] -0.2301775  1.5587083
##
## [[3]]
## [1] 0.07050839 0.12928774 1.71506499
```

The mapply() function

But what if we want to sample from normal distributions with different means, while still having samples of different sizes?

```r
set.seed(123)

rnorm(n = 1, mean = 5)
```

```
## [1] 4.439524
```

```r
rnorm(n = 2, mean = 10)
```

```
## [1]  9.769823 11.558708
```

```r
rnorm(n = 3, mean = -3)
```

```
## [1] -2.929492 -2.870712 -1.284935
```

The mapply() function

In this case we need to iterate over two vectors, one for sample sizes and one for means:

```r
set.seed(123)

sample_size <- 1:3
mu <- c(5, 10, -3)
mapply(rnorm, n = sample_size, mean = mu)
```

```
## [[1]]
## [1] 4.439524
##
## [[2]]
## [1]  9.769823 11.558708
##
## [[3]]
## [1] -2.929492 -2.870712 -1.284935
```

The mapply() function

Now suppose we also want each sample to have a different standard deviation:

```r
set.seed(123)

rnorm(n = 1, mean = 5, sd = 1)
```

```
## [1] 4.439524
```

```r
rnorm(n = 2, mean = 10, sd = 3)
```

```
## [1]  9.309468 14.676125
```

```r
rnorm(n = 3, mean = -3, sd = 5)
```

```
## [1] -2.647458 -2.353561  5.575325
```

The mapply() function

```
set.seed(123)

sample_size <- 1:3
mu <- c(5, 10, -3)
sigma <- c(1, 3, 5)

mapply(rnorm, mean = mu, sd = sigma, n = sample_size)
```

```
## [[1]]
## [1] 4.439524
##
## [[2]]
## [1]  9.309468 14.676125
##
## [[3]]
## [1] -2.647458 -2.353561  5.575325
```

The mapply() function

Now suppose we wanted our results with two decimal places only:

```
set.seed(123)

results <- mapply(rnorm, mean = mu, sd = sigma,
                  n = sample_size)
```

```
sapply(results, FUN = round, 2)
```

```
## [[1]]
## [1] 4.44
##
## [[2]]
## [1]  9.31 14.68
##
## [[3]]
## [1] -2.65 -2.35  5.58
```

mapply() vs for loop

```r
set.seed(123)

sample_size <- 1:3
mu <- c(5, 10,-3)
sigma <- c(1, 3, 5)

res <- vector(mode = "list")

for (i in 1:3) {
  res[[i]] <- round(rnorm(mean = mu[i],
                          sd = sigma[i],
                          n = sample_size[i]),
                    2)
}
```

mapply() vs for loop

```r
res
```

```
## [[1]]
## [1] 4.44
##
## [[2]]
## [1]  9.31 14.68
##
## [[3]]
## [1] -2.65 -2.35  5.58
```

Relational models

- Sometimes our tables are related to other tables.
- It is often necessary to complement one table with information from another table, or to cross information between tables.
- We usually join tables by using one or more variables that are present in both tables as a key to match rows from one table to the other.

A simple relational model

```
set.seed(1)

Sales <-data.frame(
  Product = sample(c("Toaster", "Radio", "TV"),
                   size = 7, replace = TRUE),
  CustomerID =c(rep("1_2019", 2),
              paste(2:3, "2019", sep = "_"),
              paste(1:3, "2020", sep = "_")))

Sales$Price <- round(ifelse(
  Sales$Product == "TV", rnorm(1, 400, 20),
  ifelse(Sales$Product == "Toaster",
         rnorm(1, 40, 2), rnorm(1, 35, 2))))
```

A simple relational model

```r
set.seed(1)

Clients <-data.frame(
  CustomerID =c(paste(2:4, "2019", sep = "_"),
                paste(1:2, "2020", sep = "_")),
  State = sample(c("CA", "AZ", "IL", "MA"),
                size = 5, replace = TRUE))
```

A simple relational model

Table 1: Sales

Product	CustomerID	Price
Toaster	1_2019	38
TV	1_2019	407
Toaster	2_2019	38
Radio	3_2019	36
Toaster	1_2020	38
TV	2_2020	407
TV	3_2020	407

Table 2: Clients

CustomerID	State
2_2019	CA
3_2019	MA
4_2019	IL
1_2020	CA
2_2020	AZ

Joining tables

- `CustomerID` is present in both tables and uniquely identifies each row of the `Clients` table. We can therefore use it as a key to match rows from one table to another.
- In R this can be done with the `merge()` function.

Inner join

The inner join returns only rows that have matching values in both tables:

```
merge(x = Sales, y = Clients,
  by = "CustomerID")
```

```
##   CustomerID Product Price State
## 1     1_2020 Toaster    38    CA
## 2     2_2019 Toaster    38    CA
## 3     2_2020      TV   407    AZ
## 4     3_2019   Radio    36    MA
```

Natural join

A natural join is an inner join where the joining attributes are defined as having equal names, so they need not be stated explicitly:

```
merge(x = Sales, y = Clients)
```

```
##   CustomerID Product Price State
## 1     1_2020 Toaster    38    CA
## 2     2_2019 Toaster    38    CA
## 3     2_2020      TV   407    AZ
## 4     3_2019   Radio    36    MA
```

Left join

To includes all the rows of x and only those from y that match use `all.x = TRUE`:

```
merge(x = Sales, y = Clients,
  by = "CustomerID",
  all.x = TRUE)
```

```
##   CustomerID Product Price State
## 1     1_2019 Toaster    38  <NA>
## 2     1_2019      TV   407  <NA>
## 3     1_2020 Toaster    38    CA
## 4     2_2019 Toaster    38    CA
## 5     2_2020      TV   407    AZ
## 6     3_2019   Radio    36    MA
## 7     3_2020      TV   407  <NA>
```

Right join

To include all the rows of y and only those from x that match use `all.y = TRUE`:

```
merge(x = Sales, y = Clients,
  by = "CustomerID",
  all.y = TRUE)
```

```
##   CustomerID Product Price State
## 1     1_2020 Toaster    38    CA
## 2     2_2019 Toaster    38    CA
## 3     2_2020      TV   407    AZ
## 4     3_2019   Radio    36    MA
## 5     4_2019    <NA>    NA    IL
```

Full outer join

To keep all rows from both tables use `all = TRUE`.

```
merge(x = Sales, y = Clients,
  by = "CustomerID",
  all = TRUE)
```

```
##   CustomerID Product Price State
## 1     1_2019 Toaster    38  <NA>
## 2     1_2019      TV   407  <NA>
## 3     1_2020 Toaster    38    CA
## 4     2_2019 Toaster    38    CA
## 5     2_2020      TV   407    AZ
## 6     3_2019   Radio    36    MA
## 7     3_2020      TV   407  <NA>
## 8     4_2019    <NA>    NA    IL
```

Cross Join

Cartesian product of the two tables. The output has `nrow(x) * nrow(y)` rows and `ncol(x) + ncol(y)` columns.

```
merge(x = Sales, y = Clients,
      by = NULL)
```

Joining tables

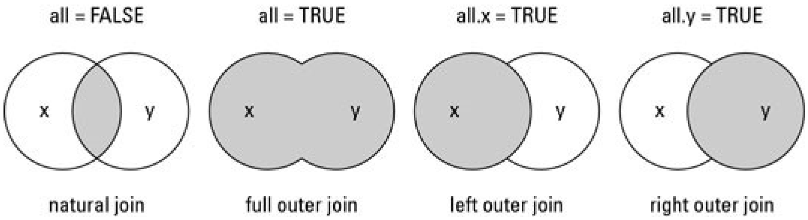

Figure 1: Join operations with `Merge()`

Joining tables

- If the merging key is a combination of more than one column, you can provide a vector to `by`.
- If the columns used as key have different names in different tables, we need to use `by.x` and `by.y` instead of `by`.
- If no `by` argument is provided, the tables are merged on the columns with names they both have.
- `all.x`, `all.y`, and `all` are set to FALSE by default. This is why the default join is the natural join.

The `sqldf` package

- You can run SQL queries in R using the `sqldf` package.
- SQL queries must be provided to the `sqldf()` function as strings.

```r
library(sqldf)
```

Inner join with sqldf

```
sqldf("SELECT CustomerID, Product, Price, State
       FROM Sales
       JOIN Clients
       USING(CustomerID)
       ORDER BY CustomerID")
```

```
##   CustomerID Product Price State
## 1     1_2020 Toaster    38    CA
## 2     2_2019 Toaster    38    CA
## 3     2_2020      TV   407    AZ
## 4     3_2019   Radio    36    MA
```

Left join with sqldf

```r
sqldf("SELECT CustomerID, Product, Price, State
       FROM Sales
       LEFT JOIN Clients
       USING(CustomerID)
       ORDER BY CustomerID")
```

```
##   CustomerID Product Price State
## 1     1_2019 Toaster    38  <NA>
## 2     1_2019      TV   407  <NA>
## 3     1_2020 Toaster    38    CA
## 4     2_2019 Toaster    38    CA
## 5     2_2020      TV   407    AZ
## 6     3_2019   Radio    36    MA
## 7     3_2020      TV   407  <NA>
```

Cross join with sqldf

```r
sqldf("SELECT *
       FROM Sales
       CROSS JOIN Clients
       ORDER BY CustomerID")
```

Subgroup summaries with `aggregate()`

The `aggregate()` function can be used to compute subgroup summary statistics.

Subgroup summaries with aggregate()

```
my_df <- data.frame(
  age = c(22, 36, 21, 39, 33, 45, 34, 59),
  smoker = factor(c("no", "yes", "no", "no", "yes",
                    "no", "yes", "yes")),
  child = factor(c("no", "yes", "no", "no", "yes",
                   "yes", "no", "yes")),
  income = c(0.8, 1.8, 1.6, 1.5, 2.3, 1.4, 1.8, 1.5),
  stringsAsFactors = FALSE)
```

Subgroup summaries with aggregate()

```
my_df
```

```
##   age smoker child income
## 1  22     no    no    0.8
## 2  36    yes   yes    1.8
## 3  21     no    no    1.6
## 4  39     no    no    1.5
## 5  33    yes   yes    2.3
## 6  45     no   yes    1.4
## 7  34    yes    no    1.8
## 8  59    yes   yes    1.5
```

Subgroup summaries with aggregate()

On average, do people with children earn more than people without children?

```
aggregate(formula = income ~ child,
          data = my_df,
          FUN = mean)
```

```
##   child income
## 1    no  1.425
## 2   yes  1.750
```

Subgroup summaries with aggregate()

```
aggregate(
  x = my_df["income"],
  by = list(child = my_df$child),
  FUN = mean)
```

```
##   child income
## 1    no  1.425
## 2   yes  1.750
```

Subgroup summaries with aggregate()

On average, do people who smoke earn more than people who don't?

```r
aggregate(income ~ smoker, my_df, mean)
```

```
##   smoker income
## 1     no  1.325
## 2    yes  1.850
```

Subgroup summaries with aggregate()

```
aggregate(
  my_df["income"],
  list(smoker = my_df$smoker),
  mean)
```

```
##   smoker income
## 1     no  1.325
## 2    yes  1.850
```

Subgroup summaries with aggregate()

Is the median income higher for smokers or non-smokers?

```r
aggregate(income ~ smoker, my_df, median)
```

```
##   smoker income
## 1     no   1.45
## 2    yes   1.80
```

Subgroup summaries with aggregate()

```r
aggregate(
  my_df["income"],
  list(smoker = my_df$smoker),
  median)
```

```
##   smoker income
## 1     no   1.45
## 2    yes   1.80
```

Subgroup summaries with aggregate()

What is the lowest income for someone with children? And without?

```r
aggregate(income ~ child, my_df, min)
```

```
##   child income
## 1    no    0.8
## 2   yes    1.4
```

Subgroup summaries with aggregate()

```
aggregate(
  my_df["income"],
  list(child = my_df$child),
  min)
```

```
##   child income
## 1    no    0.8
## 2   yes    1.4
```

Subgroup summaries with aggregate()

Is the average age of people with children higher than that of people without children?

```
aggregate(age ~ child, my_df, mean)
```

```
##   child   age
## 1    no 29.00
## 2   yes 43.25
```

Subgroup summaries with aggregate()

```
aggregate(
  my_df["age"],
  list(child = my_df$child),
  mean)
```

```
##   child   age
## 1    no 29.00
## 2   yes 43.25
```

Subgroup summaries with aggregate()

Is the median age of smokers higher than that of non-smokers?

```
aggregate(age ~ smoker, my_df, median)
```

```
##   smoker  age
## 1     no 30.5
## 2    yes 35.0
```

Subgroup summaries with aggregate()

```r
aggregate(
  my_df["age"],
  list(smoker = my_df$smoker),
  median)
```

```
##   smoker  age
## 1     no 30.5
## 2    yes 35.0
```

Subgroup summaries with aggregate()

Compare the age of the younger person with children with the age of the younger person without children:

```
aggregate(age ~ child, my_df, min)
```

```
##   child age
## 1    no  21
## 2   yes  33
```

Subgroup summaries with aggregate()

```r
aggregate(
  my_df["age"],
  list(child = my_df$child),
  min)
```

```
##   child age
## 1    no  21
## 2   yes  33
```

Subgroup summaries with aggregate()

What is the age of the older smoker?

```
subset(
  aggregate(age ~ smoker, my_df, max),
  smoker == "yes",
  select = "age"
  )
```

```
##   age
## 2  59
```

Subgroup summaries with aggregate()

```r
subset(
  aggregate(
    my_df["age"],
    list(smoker = my_df$smoker),
    max),
  smoker == "yes",
  select = "age")
```

```
##   age
## 2  59
```

Subgroup summaries with aggregate()

We can divide our subgroups further into more subgroups:

```r
aggregate(income ~ smoker + child, my_df, mean)
```

```
##   smoker child   income
## 1     no    no 1.300000
## 2    yes    no 1.800000
## 3     no   yes 1.400000
## 4    yes   yes 1.866667
```

Subgroup summaries with aggregate()

```
aggregate(
  my_df["income"],
  list(smoker = my_df$smoker,
       child = my_df$child),
  mean)
```

```
##   smoker child   income
## 1     no    no 1.300000
## 2    yes    no 1.800000
## 3     no   yes 1.400000
## 4    yes   yes 1.866667
```

Subgroup summaries with aggregate()

On average, do parents who smoke earn more than parents who don't smoke?

```
subset(
  aggregate(income ~ smoker + child, my_df, mean),
  child == "yes",
  select = c(smoker, income)
  )
```

```
##   smoker   income
## 3     no 1.400000
## 4    yes 1.866667
```

Subgroup summaries with aggregate()

```
subset(
  aggregate(
    my_df["income"],
    list(smoker = my_df$smoker,
         child = my_df$child),
    mean),
  child == "yes",
  select = c(smoker, income)
  )
```

```
##   smoker   income
## 3     no 1.400000
## 4    yes 1.866667
```

Subgroup summaries with `aggregate()`

Is the median age of parents who smoke higher than that of parents who don't smoke?

```
subset(
  aggregate(age ~ smoker + child, my_df, median),
  child == "yes",
  select = c(smoker, age)
  )
```

```
##   smoker age
## 3     no  45
## 4    yes  36
```

Subgroup summaries with aggregate()

```r
subset(
  aggregate(
    my_df["age"],
    list(smoker = my_df$smoker,
         child = my_df$child),
    median),
  child == "yes",
  select = c(smoker, age)
  )
```

```
##   smoker age
## 3     no  45
## 4    yes  36
```

Subgroup summaries with sqldf()

On average, do people with children earn more than people without children?

```
sqldf(
  "SELECT child, AVG(income) as income
  FROM my_df
  GROUP BY child"
)
```

```
##   child income
## 1    no  1.425
## 2   yes  1.750
```

Subgroup summaries with sqldf()

On average, do people who smoke earn more than people who don't?

```
sqldf(
  "SELECT smoker, AVG(income) as income
  FROM my_df
  GROUP BY smoker"
)
```

```
##   smoker income
## 1     no  1.325
## 2    yes  1.850
```

Subgroup summaries with sqldf()

What is the lowest income for someone with children? And without?

```
sqldf(
  "SELECT child, min(income) as income
  FROM my_df
  GROUP BY child"
)
```

```
##   child income
## 1    no    0.8
## 2   yes    1.4
```

Subgroup summaries with `sqldf()`

Is the average age of people with children higher than that of people without children?

```
sqldf(
  "SELECT child, AVG(age) as age
  FROM my_df
  GROUP BY child"
)
```

```
##   child   age
## 1    no 29.00
## 2   yes 43.25
```

Subgroup summaries with sqldf()

Compare the age of the younger person with children with the age of the younger person without children:

```
sqldf(
  "SELECT child, min(age) as age
  FROM my_df
  GROUP BY child"
)
```

```
##   child age
## 1    no  21
## 2   yes  33
```

Subgroup summaries with `sqldf()`

What is the age of the older smoker?

```
sqldf(
  "SELECT max(age) as age
  FROM my_df
  GROUP BY smoker
  HAVING smoker = 'yes'
  "
)
```

```
##   age
## 1  59
```

Subgroup summaries with aggregate()

We can divide our subgroups further into more subgroups:

```
sqldf(
  "SELECT smoker, AVG(income) as income
  FROM my_df
  GROUP BY child, smoker
  "
)
```

```
##   smoker   income
## 1     no 1.300000
## 2    yes 1.800000
## 3     no 1.400000
## 4    yes 1.866667
```

Subgroup summaries with sqldf()

On average, do parents who smoke earn more than parents who don't smoke?

```
sqldf(
  "SELECT smoker, AVG(income) as income
  FROM my_df
  GROUP BY child, smoker
  HAVING child = 'yes'
  "
)
```

```
##   smoker   income
## 1     no 1.400000
## 2    yes 1.866667
```