

# Working with Data

MATH 4939 – Winter 2020

Georges Monette

January 2020

## Contents

1	Introduction	6
2	Data Input —	8

2.1	From a package —	8
2.2	From vectors —	12
2.3	From a text file —	16
2.3.1	From a remote site —	16
2.3.2	Locally —	17
2.4	Excel files —	20
2.4.1	CSV files —	20
2.4.2	Reading worksheets from an Excel file —	20
2.4.3	Annoyance in ‘readxl’ —	35
2.4.4	A wrapper for ‘readxl’ —	35
2.5	Read sheets from an SPSS file —	36
2.6	Referring to variables in a data frame —	38
<b>3</b>	<b>Subsetting data frames —</b>	<b>41</b>
3.1	match: associative array —	45
3.2	recycling principle —	46
3.3	making vectors: rep and seq —	48
<b>4</b>	<b>apply and friends —</b>	<b>51</b>

4.1	lapply: do the same thing to each element of a list — . . . . .	54
4.2	Simple function — . . . . .	56
<b>5</b>	<b>Multilevel data —</b>	<b>58</b>
5.1	Extensions of apply functions — . . . . .	58
5.2	Merge examples — . . . . .	107
5.2.1	Calculate GPA — . . . . .	109
<b>6</b>	<b>The many ways of referring to variables —</b>	<b>115</b>
<b>7</b>	<b>OOP: Object-oriented programming —</b>	<b>122</b>
7.1	Creating a new class — . . . . .	129
7.1.1	Creating a class and methods for existing generics — . . .	129
<b>8</b>	<b>Data wrangling —</b>	<b>162</b>
8.1	Regular Expressions to replace strings within string variables —	162
8.1.1	Basic Regular Expressions — . . . . .	163
8.1.2	Taking a closer look at gsub — . . . . .	165
8.1.3	Common Regular Expressions — . . . . .	169

8.1.4	Quiz question —	170
8.1.5	Taking a look at regexr —	175
<b>9</b>	<b>Reshaping Data —</b>	<b>179</b>
9.1	Long form —	180
9.2	Wide form —	183
9.3	Relational data base form —	185
9.4	From Wide to Long —	188
9.5	From Long to Wide —	191
9.6	More examples —	193
9.7	Variables and years in long form —	199
9.8	Working with long data frames —	202
9.9	Reshaping categorical data —	207
9.9.1	Tabular data —	208
9.9.2	Making marginal tables —	222
9.10	Frequency data frame —	226
9.11	Individual data frame —	227
9.12	Frequency data frame with response variable on rows —	236

<b>10 Using R Script with Markdown</b>	<b>244</b>
<b>11 Attributes</b>	<b>245</b>
11.0.1 Exercises	248
<b>12 Traps and Pitfalls</b>	<b>249</b>
12.1 Factors	250
12.1.1 Transformation of factors to characters or codes	250
12.1.2 Factors transformed to character	251
12.1.3 Factors transformed to numeric	251
12.1.4 Factors operations that return a factor	251
12.1.5 Other special factor pitfalls	252
12.1.6 ‘drop’ doesn’t work with subset	252
12.2 <code>diag</code> can be tricky	253
12.3 Reading and Writing Data Files	255
12.3.1 NA as a valid value (the Namibia problem)	255
12.4 Prediction	256
12.4.1 Prediction with <code>nlme</code>	256
12.4.2 Exercises	256

<b>13 Useful Techniques and Tricks</b> —	<b>256</b>
13.1 Changing all variables to characters in a data frame — . . . . .	257
<b>References</b> —	<b>257</b>

This version rendered on January 28 2020 23:12

# 1 Introduction

These notes are a work in progress meant to supplement material in Fox and Weisberg (2019). The focus is using data to answer simple questions that only require simple tools.

There is a [collection of exercises](#) on R many of which are related to this material.

Questions, links and discussions concerning this material can take place on Piazza. There's a copy of this document on Piazza that you can edit to

- correct errors
- improve or add to the presentation
- add relevant exercises. To add exercises, precede them with a 'level 3' heading: '### Exercises'

We use the following packages in this script which you may need to install if you haven't already:

```
if(FALSE) {  
  # install these manually if you need to:  
  install.packages('haven')  
  install.packages('tidyverse') # might take a long time  
  install.packages('readxl')  
  install.packages('devtools')  
  install.packages('car')  
  install.packages('magrittr')  
  install.packages('latticeExtra')  
  install.packages('alr4')  
  devtools::install_github('gmonette/spida2')  
}
```

Play with the basic R functions listed in Hadley Wickham's chapter on vocabulary. Write a script that illustrates the use of these functions.

## 2 Data Input —-

### 2.1 From a package —-

The Davis data set in the 'car' package on measured versus reported height:

```
library("car") # loads car and carData packages
```

```
Loading required package: carData
```

```
class(Davis)
```

```
[1] "data.frame"
```



```
brief(Davis)
```

```
200 x 5 data.frame (195 rows omitted)
```

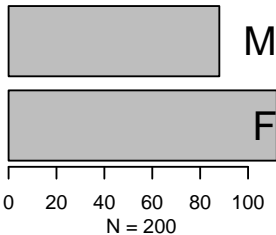
```
  sex weight height repwt repht
  [f]   [i]   [i]   [i]   [i]
1  M     77   182    77   180
2  F     58   161    51   159
3  F     53   161    54   158
. . .
199 M     90   181    91   178
200 M     79   177    81   178
```

```
library(spida2)
```

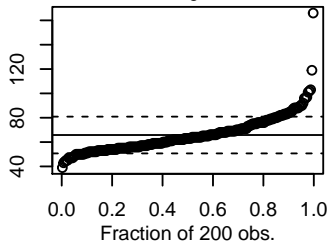
```
spida2: development branch 0.2.0.9000.
```

```
xqplot(Davis) # uniform quantiles
```

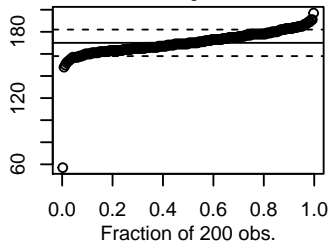
sex  
factor



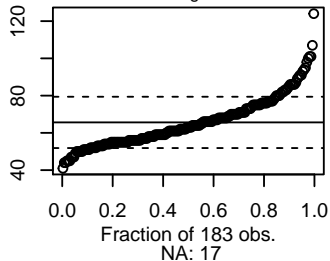
weight  
integer



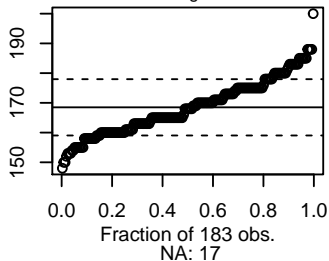
height  
integer



repwt  
integer

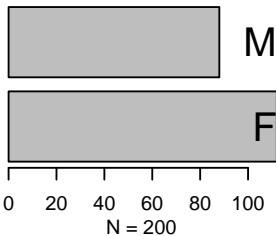


repht  
integer

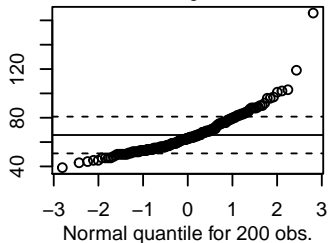


```
xqplot(Davis, ptype = 'n') # normal quantiles
```

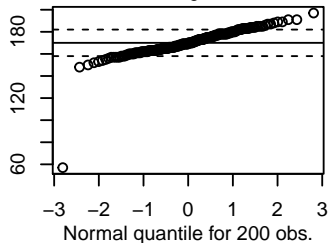
sex  
factor



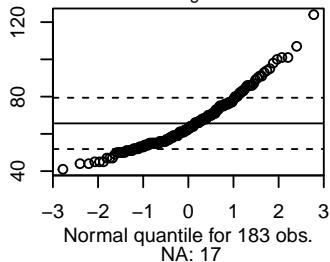
weight  
integer



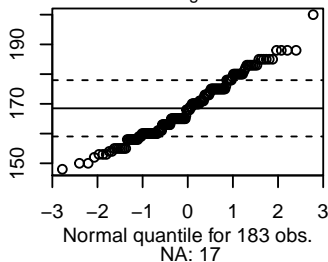
height  
integer



repwt  
integer



reph  
integer



```
# if not installed: install.packages('alr4')  
data("Challeng", package="alr4")
```

## brief(Challeng)

23 x 7 data.frame (18 rows omitted)

	temp	pres	fail	n	erosion	blowby	damage
	[i]	[i]	[i]	[i]	[i]	[i]	[i]
4/12/81	66	50	0	6	0	0	0
11/12/81	70	50	1	6	1	0	4
3/22/82	69	50	0	6	0	0	0
. . .							
11/26/85	76	200	0	6	0	0	0
1/12/86	58	200	1	6	1	0	4

## 2.2 From vectors —

```
cooperation <- c(49, 64, 37, 52, 68, 54, 61, 79, 64, 29,  
                27, 58, 52, 41, 30, 40, 39, 44, 34, 44)
```

```
(condition <- rep(c("public", "anonymous"), c(10, 10)))
```

```
[1] "public"      "public"      "public"      "public"      "public"
[6] "public"      "public"      "public"      "public"      "public"
[11] "anonymous"  "anonymous"  "anonymous"  "anonymous"  "anonymous"
[16] "anonymous"  "anonymous"  "anonymous"  "anonymous"  "anonymous"
```

```
(sex <- rep(rep(c("male", "female"), each=5), 2))
```

```
[1] "male"      "male"      "male"      "male"      "male"      "female"
[7] "female"    "female"    "female"    "female"    "male"      "male"
[13] "male"      "male"      "male"      "female"    "female"    "female"
[19] "female"    "female"
```

```
rep(5, 3)
```

```
[1] 5 5 5
```

```
rep(c(1, 2, 3), 2)
```

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

```
rep(1:3, 3:1)
```

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

```
Guyer1 <- data.frame(cooperation, condition, sex)  
brief(Guyer1)
```

```
20 x 3 data.frame (15 rows omitted)
```

	cooperation	condition	sex
	[n]	[f]	[f]
1	49	public	male
2	64	public	male
3	37	public	male
. . .			
19	34	anonymous	female
20	44	anonymous	female

```
Guyer2 <- data.frame(  
  cooperation = c(49, 64, 37, 52, 68, 54, 61, 79, 64, 29,  
                 27, 58, 52, 41, 30, 40, 39, 44, 34, 44),
```

```
condition = rep(c("public", "anonymous"), c(10, 10)),  
sex = rep(rep(c("male", "female"), each=5), 2)  
)  
identical(Guyer1, Guyer2)
```

```
[1] TRUE
```

The structure of a data frame:

- a **list** in which each element has the same length

```
str(Guyer1)
```

```
'data.frame': 20 obs. of 3 variables:  
 $ cooperation: num 49 64 37 52 68 54 61 79 64 29 ...  
 $ condition : Factor w/ 2 levels "anonymous","public": 2 2 2 2 2  
 $ sex : Factor w/ 2 levels "female","male": 2 2 2 2 2 1 1
```

## 2.3 From a text file —

### 2.3.1 From a remote site —

If the the values in the file are separated by arbitrary white space use the `read.table` function.

```
Duncan <- read.table(  
  file="https://socialsciences.mcmaster.ca/jfox/Books/Companion/data  
  header=TRUE)  
brief(Duncan) # a 'car' function that prints first 3 and last 2
```

45 x 4 data.frame (40 rows omitted)

	type	income	education	prestige
	[f]	[i]	[i]	[i]
accountant	prof	62	86	82
pilot	prof	72	76	83
architect	prof	75	92	90
. . .				
policeman	bc	34	47	41



```
waiter      bc          8          32          10
```

*# rows along with the type of each variable*

### 2.3.2 Locally —

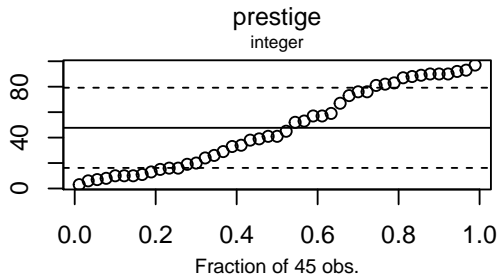
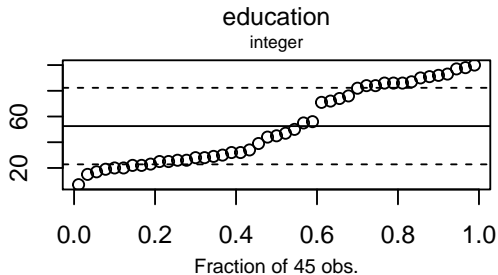
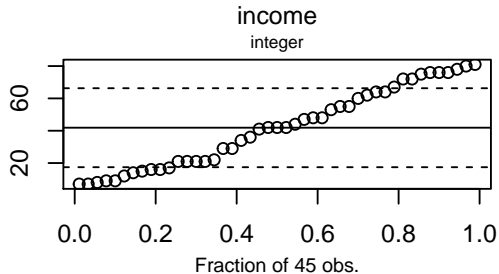
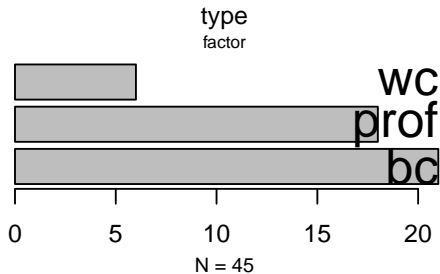
We're going to download this text file from John Fox's website to illustrate what it looks like when reading a local file:

```
download.file(
  "https://socialsciences.mcmaster.ca/jfox/Books/Companion/data/Duncan.txt",
  "Duncan.txt")
Duncan <- read.table("Duncan.txt", header = TRUE)
brief(Duncan)
```

```
45 x 4 data.frame (40 rows omitted)
      type income education prestige
      [f]   [i]      [i]      [i]
accountant prof     62      86      82
pilot      prof     72      76      83
```

architect	prof	75	92	90
. . .				
policeman	bc	34	47	41
waiter	bc	8	32	10

`xqplot(Duncan)`



# use: ?Duncan for more information

## 2.4 Excel files —

### 2.4.1 CSV files —

One way to read a single Excel spreadsheet is to save it from Excel as a comma-separated-values (CSV) file from Excel. You can then read it with the **read.csv** function that works like ‘read.table’ except that ‘header = TRUE’ is the default and fields are separated by commas (,). If a field contains a comma then it must be enclosed in quotes. You don’t need to worry about this. Excel takes care of creating an appropriate file and ‘read.csv’ takes care of reading it.

For more advanced work, look at the help file for **read.table**.

### 2.4.2 Reading worksheets from an Excel file —

The Hadleyverse uses ‘tibbles’: data frames with extra information and an aversion to factors and rownames

I believe that currently the **readxl** package in CRAN may be (**but see some reservations below**) the most effective way to read Excel worksheets directly.

The ‘readxl’ package is part of the ‘tidyverse’ (which is part of what’s often referred to as the ‘Hadleyverse’ after Hadley Wickham who started it). The Hadleyverse adds a lot of functionality to R and redoes much of R’s basic functions. Some people think of it as a new dialect of R, a bit like American English compared with British English. It’s controversial whether one should invest effort learning basic R or whether one should jump into the Hadleyverse from the start. Hadley Wickam’s excellent on-line book, *Advanced R* whose [first edition is on line](#) explores the depths of ‘base’ R, which are complex enough to require an extensive treatment for anyone who aspires to be creative with the language, either in base R or in the Hadleyverse.

```
library("tidyverse") # loads all of the tidyverse packages
```

```
-- Attaching packages -----
```

```
<U+2713> ggplot2 3.2.1      <U+2713> purrr   0.3.3
<U+2713> tibble  2.1.3      <U+2713> dplyr   0.8.3
<U+2713> tidyr   1.0.0      <U+2713> stringr 1.4.0
<U+2713> readr   1.3.1      <U+2713> forcats 0.4.0
```

```
-- Conflicts -----  
x readr::cols() masks spida2::cols()  
x dplyr::filter() masks stats::filter()  
x ggplot2::labs() masks spida2::labs()  
x dplyr::lag() masks stats::lag()  
x purrr::map() masks spida2::map()  
x dplyr::recode() masks car::recode()  
x purrr::some() masks car::some()
```

Note how ‘tidyverse’ masks many functions including functions in ‘base’ packages. This can break code that uses these functions. Having a look at the list of masked function, I notice ‘some’ in ‘car’ that I use so, to set things straight, I can redefine ‘some’ to make sure that we use the version in ‘car’:

```
some <- car::some
```

This stores a local version ‘car::some’ in the global environment which R searches before looking in ‘tidyverse’. This is the list of places where R searches for objects when they are called from the command line:

```
search()
```

```
[1] ".GlobalEnv"          "package:forcats"    "package:stringr"  
[4] "package:dplyr"       "package:purrr"      "package:readr"  
[7] "package:tidyr"       "package:tibble"     "package:ggplot2"  
[10] "package:tidyverse"  "package:spida2"     "package:car"  
[13] "package:carData"    "package:stats"      "package:graphics"  
[16] "package:grDevices"  "package:utils"      "package:datasets"  
[19] "package:methods"    "Autoloads"          "package:base"
```

You can see that packages are searched in the reverse order in which they were loaded.

```
Duncan.tibble <- as_tibble(Duncan)
```

```
print(Duncan.tibble, n=5) # note print() method
```

```
# A tibble: 45 x 4  
  type  income education prestige  
<fct> <int>    <int>    <int>  
1 prof     62      86      82
```

```
2 prof      72      76      83
3 prof      75      92      90
4 prof      55      90      76
5 prof      64      86      90
# ... with 40 more rows
```

```
brief(Duncan.tibble)
```

```
45 x 4 tbl_df (40 rows and 1 columns omitted)
```

```
Warning: Setting row names on a tibble is deprecated.
```

```
   type . . . education prestige
   [f]          [i]      [i]
1  prof          86      82
2  prof          76      83
3  prof          92      90
. . .
44 bc           47      41
45 bc           32      10
```



```
brief(as.data.frame(Duncan.tibble))
```

```
45 x 4 data.frame (40 rows omitted)
```

	type	income	education	prestige
	[f]	[i]	[i]	[i]
1	prof	62	86	82
2	prof	72	76	83
3	prof	75	92	90
. . .				
44	bc	34	47	41
45	bc	8	32	10

- If `file.xlsx` is an Excel file and you want to read the second worksheet that uses 'NA' for missing values, use: `dd <- read_excel('file.xlsx', sheet = 2, na = 'NA')`
- If you want to read an Excel file on a web server (e.g. blackwell), some functions that read files, e.g. `read.csv`, will accept the URL instead of a path to a local file. However, at this time, `read_excel` requires a local path. Thus, you need to download the file before reading it with

`read_excel`. The usual way to download a file within R uses the `download.file` function. However, the default way to download binary Excel files, may corrupt the file. Try using the 'curl' method as illustrated below. This may not be necessary on Macs. Please let us know on Piazza!

- There are two `xlsx` files on blackwell:
  - 'file.xlsx': a small Excel file with clean data except that a numerical value was entered as '\$1,000.00'
  - 'file2.xlsx': same as above except that there is an 'invisible' single blank in the first column of the row after the actual data.

`read_excel` will interpret this as an NA by default and create an entire row of NA values. Also, some entries are blank and some entries have been indicated as NAs by entering 'NA'. These common irregularities in Excel files can create havoc unless you are ready to deal with them.

Run this code line by line:

```
library(readxl)
dir <- 'http://blackwell.math.yorku.ca/MATH4939/R/'
download.file(paste0(dir,'file.xlsx'),'_file.xlsx',
```

```
method = 'curl') # download to _file.xlsx
                # to avoid overwriting
download.file(paste0(dir,'file2.xlsx'),'_file2.xlsx',
              method = 'curl')
dt <- read_excel('_file.xlsx')
dt2 <- read_excel('_file2.xlsx')
dt
```

```
# A tibble: 5 x 3
```

	Name	Age	Purchase
	<chr>	<dbl>	<dbl>
1	Mary Smith	25	1000
2	Chow, Vincent	42	200.
3	Mohammed, Tarik	56	123
4	O'Brien, John	21	2000
5	Jolie, Mary	33	150

Note that '\$1,000.00' was read as a numerical value! 'read.csv' would not do this. See exercises for a way of cleaning up a numeric variable that was entered

with extraneous symbols (\$) and a thousands separator.

```
dt2
```

```
# A tibble: 6 x 3
  Name           Age Purchase
  <chr>          <chr>   <dbl>
1 Mary Smith    25      1000
2 Chow, Vincent NA        200.
3 Mohammed, Tarik 56        NA
4 O'Brien, John <NA>     2000
5 Jolie, Mary   .         150
6 <NA>          <NA>     NA
```

Note that the ‘.’ was interpreted as a character value and turned ‘Age’ into a character variable. With `read.csv` the ‘.’ would have been interpreted as a missing data by default. See [this site](#) for the various possibilities used in different countries. It is easy to use regular expressions (see below) to fix amounts entered in a different format. Also, the symbol used for the radix (the decimal separator) can be specified as the `dec` argument to `read.csv`.

```
dt2$Age <- as.numeric(as.character(dt2$Age))
```

Warning: NAs introduced by coercion

```
dt2
```

```
# A tibble: 6 x 3
```

	Name	Age	Purchase
	<chr>	<dbl>	<dbl>
1	Mary Smith	25	1000
2	Chow, Vincent	NA	200.
3	Mohammed, Tarik	56	NA
4	O'Brien, John	NA	2000
5	Jolie, Mary	NA	150
6	<NA>	NA	NA

To get rid of the blank row

```
dt2 <- subset(dt2, !is.na(Name))
```

```
dt2
```

```
# A tibble: 5 x 3
  Name           Age Purchase
  <chr>         <dbl>   <dbl>
1 Mary Smith     25     1000
2 Chow, Vincent  NA      200.
3 Mohammed, Tarik 56      NA
4 O'Brien, John  NA     2000
5 Jolie, Mary    NA      150
```

The files `dt` and `dt2` are tibbles with categorical variables represented as character vectors.

```
class(dt)
```

```
[1] "tbl_df"      "tbl"        "data.frame"
```

```
sapply(dt, class)
```

```
      Name      Age      Purchase
"character" "numeric" "numeric"
```

You can turn them into data frames with factors for categorical variables as follows:

```
dt <- as.data.frame(as.list(dt))  
class(dt)
```

```
[1] "data.frame"
```

```
sapply(dt, class)
```

```
      Name      Age Purchase  
"factor" "numeric" "numeric"
```

```
dt
```

```
      Name Age Purchase  
1   Mary Smith  25  1000.00  
2  Chow, Vincent  42   200.32  
3 Mohammed, Tarik  56   123.00  
4  O'Brien, John  21  2000.00  
5   Jolie, Mary  33   150.00
```

```
dt2 <- as.data.frame(as.list(dt2))  
class(dt2)
```

```
[1] "data.frame"
```

```
sapply(dt2, class)
```

```
      Name      Age Purchase  
"factor" "numeric" "numeric"
```

```
dt2
```

	Name	Age	Purchase
1	Mary Smith	25	1000.00
2	Chow, Vincent	NA	200.32
3	Mohammed, Tarik	56	NA
4	O'Brien, John	NA	2000.00
5	Jolie, Mary	NA	150.00

Sometimes, `read_excel` will report a warning like: ... expecting numeric: got .... This happens when `read_excel` has decided that a column is numeric based



on its inspection of the top entries but then encounters non-numeric data. The remedy is to read the file as character and to modify the entries that need to be modified. See the [section below](#) on using regular expressions to fix variables **without touching the original data**.

Reread the data this way:

```
dt2 <- read_excel('_file.xlsx', na = 'NA', col_type = rep('text', n))
dt2 <- as.data.frame(as.list(dt2))
```

All variables will now be factors. You need to go through them and modify them as needed. If a variable *x*, say, should be numeric, and **does not need any editing**, you can fix it with:

```
z <- as.numeric(as.character(dd$x)) # note that 'as.character' is
z # have a look
dd$x <- z # if everything is ok
```

If a variable needs editing, for example suppose student numbers that should have 9 digits have been entered in a variety of ways: '123 456 789', or '123-456-789', or '#12346789' or with the wrong number of digits, you could do

this:

```
dd$x.orig <- dd$x # keep the original in case you need to go back
z <- as.character(dd$x)
# Have a look:
z
z <- gsub('[ -]', '', z) # remove all blanks and hyphens.
# Note that the hyphen must be first or last in the brackets,
# otherwise it denotes a range, i.e. '[A-Z]' matches any
# capital letter.
z <- sub('^#', '', z) # remove leading # signs, '^' is an 'anchor' m
z # have another look
table(nchar(z))
# If valid data must have a length of 9:
z9 <- nchar(z) == 9
z <- ifelse(z9, as.numeric(z), NA)
dd$x <- z # Fixed! Invalid input is NA
```

There are a number of packages to write Excel files:

- `openxlsx`
- `writeXLS`

Post your experiences on Piazza.

### 2.4.3 Annoyance in ‘`readxl`’ —

Using `read_excel` to read a file with 9-digit student numbers as text because some were entered incorrectly by students converted numbers to scientific notation: ‘123456789E0’ for no apparent reason because they were read as ‘text’. The transformation back to numeric variables works correctly for 9 digits. One would need to experiment with more digits in the input. It’s annoying that the string is altered in a way that seems unnecessary.

### 2.4.4 A wrapper for ‘`readxl`’ —

From [this thread on stackoverflow](#) here is a function that reads an Excel file and make every variable a character variable to avoid problems with variables that you want to read as characters although the top of the data set contains only values that `read_excel` considers numeric. There is isn’t a single

argument to `read_excel` to request that all variables be read as characters. Instead, you need to know the number of variable to repeat the `col_types` argument as many times as there are columns. That is, the authors did not build in recycling! This function first finds out how many variable there are so it can then call `read_excel` with a correct `col_types` argument.

```
Read_excel<-function(file, sheet, ...)  
{  
  library("readxl")  
  num.columns <- length(readxl::xlsx_col_types(file, sheet = sheet,  
  readxl::read_excel(file, sheet = sheet,  
                      col_types = rep("text", num.col  
})
```

## 2.5 Read sheets from an SPSS file —

SPSS files have long been a problem for R but there is a relatively recent package, 'haven', on CRAN that seems to do an excellent job. It uses R attributes to store SPSS variable labels and correctly transforms SPSS date into

R objects of class 'Date'. Be aware that it is common in SPSS to have user-defined missing values. By default all these values are converted to 'NA' in R but the distinct values are likely to be informative. Use the argument, 'user\_na = TRUE' to recover missing value labels. Like 'read\_excel', 'read\_sav' creates a 'tibble' but the trick that works with Excel files of using 'as.data.frame(as.list(...))' to turn it into a standard data frame does not work here. You might have to some surgery on the variables in some cases.

**Warning:** Some functions, e.g. 'lm' may treat a categorical variable as a numeric variable producing embarrassingly non-sensical results.

```
library(haven)
path <- system.file('examples', 'iris.sav', package = 'haven')
      # get the path to a system file
path
dd <- haven::read_sav(path)
head(dd) # a tibble
class(dd)
fit <- lm(Petal.Width ~ Species, dd)
summary(fit) # Species is numerical
```

```
ds <- as.data.frame(as.list(dd))
head(ds) # Species is still numerical
# You are not expected to understand the next line ... yet!
dd$Species <- factor(names(attr(dd$Species, 'labels'))[dd$Species])
# complicated fix
str(dd$Species) # now it's a factor!
fit <- lm(Petal.Width ~ Species, dd)
# treats 'Species' as a factor with correct levels
summary(fit)
```

## 2.6 Referring to variables in a data frame —

Data frames have two personalities:

- list of variable (each of same length)
- matrix of entries like a spreadsheet so entries can be referred to by row and column

```
str(Duncan)
```

```
'data.frame': 45 obs. of 4 variables:  
 $ type      : Factor w/ 3 levels "bc","prof","wc": 2 2 2 2 2 2 2 2 2  
 $ income    : int  62 72 75 55 64 21 64 80 67 72 ...  
 $ education: int  86 76 92 90 86 84 93 100 87 86 ...  
 $ prestige : int  82 83 90 76 90 87 93 90 52 88 ...
```

Fully qualified name

```
Duncan$prestige      # using the '$' (select) operator
```

```
[1] 82 83 90 76 90 87 93 90 52 88 57 89 97 59 73 38 76 81 45 92  
[21] 39 34 41 16 33 53 67 57 26 29 10 15 19 10 13 24 20 7 3 16  
[41] 6 11 8 41 10
```

3rd row, 4th columns

```
Duncan[3, 4]
```

```
[1] 90
```

All rows, 4th column

```
Duncan[ , 4]
```

```
[1] 82 83 90 76 90 87 93 90 52 88 57 89 97 59 73 38 76 81 45 92  
[21] 39 34 41 16 33 53 67 57 26 29 10 15 19 10 13 24 20 7 3 16  
[41] 6 11 8 41 10
```

Refer to column by name

```
Duncan[ , "prestige"]
```

```
[1] 82 83 90 76 90 87 93 90 52 88 57 89 97 59 73 38 76 81 45 92  
[21] 39 34 41 16 33 53 67 57 26 29 10 15 19 10 13 24 20 7 3 16  
[41] 6 11 8 41 10
```

Using the 'with' function so names are interpreted within the data frame

```
with(Duncan, prestige)
```

```
[1] 82 83 90 76 90 87 93 90 52 88 57 89 97 59 73 38 76 81 45 92  
[21] 39 34 41 16 33 53 67 57 26 29 10 15 19 10 13 24 20 7 3 16
```



```
[41]  6 11  8 41 10
```

```
with(Duncan, mean

```
(prestige))
```


```

```
[1] 47.68889
```

### 3 Subsetting data frames —

`%in%` is very useful to subset rows of a data frame.

The following also illustrates inline `read.csv` and the ‘magrittr’ pipe: `%>%`

```
library(spida2)
library(car)
library(magrittr)  # for pipes
```

```
Attaching package: 'magrittr'
```

```
The following object is masked from 'package:purrr':
```

```
set_names
```

The following object is masked from 'package:tidyr':

```
extract
```

```
df <- read.csv(text =  
'  
name,    age,    sex,    height  
John Smith,    32,    M,    68  
Mary Smith,    36,    F,    67  
Andrew Smith Edwards, 42,    M,    71  
Paul Jones,    31,    M,    65  
"Smith, Mary",    33,    F,    32  
' )  
df
```

	name	age	sex	height
1	John Smith	32	M	68
2	Mary Smith	36	F	67

3	Andrew Smith Edwards	42	M	71
4	Paul Jones	31	M	65
5	Smith, Mary	33	F	32

Using subset with %in%

```
subset(df, name %in% c('John Smith', 'Paul Jones'))
```

	name	age	sex	height
1	John Smith	32	M	68
4	Paul Jones	31	M	65

Implicit drop = FALSE: The resulting factor still has the original levels (sometimes you want this)

```
subset(df, name %in% c('John Smith', 'Paul Jones'))$name
```

```
[1] John Smith Paul Jones
```

```
5 Levels: Andrew Smith Edwards John Smith ... Smith, Mary
```

Use droplevels to get drop = TRUE, and get rid of original levels.

```
subset(df, name %in% c('John Smith', 'Paul Jones')) %>%  
  droplevels %>%  
  .$name
```

```
[1] John Smith Paul Jones  
Levels: John Smith Paul Jones
```

Using regular expressions and logical subsetting

```
subset(df, grepl('Smith', name)) # Smith anywhere
```

	name	age	sex	height
1	John Smith	32	M	68
2	Mary Smith	36	F	67
3	Andrew Smith Edwards	42	M	71
5	Smith, Mary	33	F	32

```
subset(df, grepl('Smith$', name)) # Smith at end of string
```

```
name age sex height
```

```
1 John Smith 32      M      68
2 Mary Smith 36      F      67
```

### 3.1 match: associative array —

`%in%` is a special case of `match` but it's much more intuitive. Skip this if you prefer.

`match(x, table, nomatch)` # returns the position of each # x matched exactly in table

```
match(c('e', 'b', 'a', 'z', 'ee', 'A'), letters)
```

```
[1] 5 2 1 26 NA NA
```

```
match(c('e', 'b', 'a', 'z', 'ee', 'A'), letters, 0)
```

```
[1] 5 2 1 26 0 0
```

```
match(c('e', 'b', 'a', 'z', 'ee', 'A'), letters, 0) > 0
```

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

```
c('e','b','a','z','ee','A') %in% letters
```

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

can use match to translate

```
LETTERS [match(c('a','s','w', 'else'), letters)]
```

```
[1] "A" "S" "W" NA
```

```
LETTERS [match(c('a','s','w', 'else'), letters, 0)]
```

```
[1] "A" "S" "W"
```

## 3.2 recycling principle —

if a vector is too short, just recycle

```
c(1, 2, 3, 4) + 1
```

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

```
c(1, 2, 3, 4) + c(4, 3)      # no warning if multiple fits
```

```
[1] 5 5 7 7
```

```
c(1, 2, 3, 4) + c(4, 3, 2)  # produces warning otherwise
```

```
Warning in c(1, 2, 3, 4) + c(4, 3, 2): longer object length is  
not a multiple of shorter object length
```

```
[1] 5 5 5 8
```

```
c(1, 2, 3, 4)[T] # T is recycled to length 4. Why?
```

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

```
c(1, 2, 3, 4)[1] # But 1 is not recycled
```

```
[1] 1
```

### 3.3 making vectors: rep and seq —

flexible: note how differently it works if the second argument is a vector:

```
rep(1:4, 5) # recycle vector
```

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

```
rep(1:4, 1:4) # repeat each element
```

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

```
rep(1:4, each = 5) # repeat each element
```

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

like:

```
rep(1:4, c(5,5,5,5))
```

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

seq is similar to : but with more options



```
1:5
```

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

```
seq(1, 5)
```

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

```
seq(1, 5, 0.5)
```

```
[1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
```

```
seq_along(letters) # generates a sequence of
```

```
[1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20  
[21] 21 22 23 24 25 26
```

```
# indices for a vector or list
```

The `seq_along` and the `seq_len` functions are useful in **for loops**. Consider the difference between using `'seq_along(x)'` and `'1:length(x)'` if `'x'` has length 0 which can easily happen inside a function.

```
x <- 1:10
x <- x[FALSE]
x
```

```
integer(0)
```

OR

```
x <- 1:10
x <- x[0]
x
```

```
integer(0)
```

```
1:length(x) # probably not what you want in a for loop
```

```
[1] 1 0
```

```
seq_along(x)
```

```
integer(0)
```

## 4 apply and friends —

apply is the easy one, applied to slices of a matrix or array

apply(m, MARGIN, FUN, ...); MARGIN is a vector with the dimensions to be projected onto

```
a <- array(1:24, c(2,3,4))
```

```
a
```

```
, , 1
```

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

```
, , 2
```

```
      [,1] [,2] [,3]
[1,]    7    9   11
```

```
[2,]    8   10   12
```

```
, , 3
```

```
      [,1] [,2] [,3]  
[1,]   13   15   17  
[2,]   14   16   18
```

```
, , 4
```

```
      [,1] [,2] [,3]  
[1,]   19   21   23  
[2,]   20   22   24
```

```
apply(a, 1, sum)
```

```
[1] 144 156
```

```
apply(a, c(2,3), sum)
```

```
      [,1] [,2] [,3] [,4]  
[1,]     3    15    27    39  
[2,]     7    19    31    43  
[3,]    11    23    35    47
```

```
a[1,1,1] <- NA
```

```
apply(a, c(2,3), sum)
```

```
      [,1] [,2] [,3] [,4]  
[1,]   NA    15    27    39  
[2,]     7    19    31    43  
[3,]    11    23    35    47
```

```
apply(a, c(2,3), sum, na.rm = T) # extra arguments to sum
```

```
      [,1] [,2] [,3] [,4]  
[1,]     2    15    27    39  
[2,]     7    19    31    43
```

```
[3,] 11 23 35 47
```

## 4.1 `lapply`: do the same thing to each element of a list

---

A data frame is an important example of a list.

Suppose you have a data frame with many numeric variables recording temperatures in Celsius and you need to transform them to Fahrenheit

```
df <- read.csv(text=  
'  
city,      day1,  day2,  day3  
Montreal,  20,    25,    30  
Toronto,   23,    26,    19  
New York,  28,    35,    32  
' )  
df
```

```
city day1 day2 day3
```

1	Montreal	20	25	30
2	Toronto	23	26	19
3	New York	28	35	32

```
sapply(df, class) # returns a vector if it can
```

```
      city      day1      day2      day3  
"factor" "integer" "integer" "integer"
```

```
lapply(df, class) # always returns a list
```

```
$city  
[1] "factor"  
  
$day1  
[1] "integer"  
  
$day2  
[1] "integer"
```

```
$day3  
[1] "integer"
```

## 4.2 Simple function —

Simple function for now, later we'll use a generic function and methods

```
to_fahrenheit <- function(x) {  
  if(is.factor(x) || !is.numeric(x) ) x # why 'is.factor'?  
  else 32 + (9/5)*x  
}  
to_fahrenheit
```

```
function(x) {  
  if(is.factor(x) || !is.numeric(x) ) x # why 'is.factor'?  
  else 32 + (9/5)*x  
}
```

```
lapply(df, to_fahrenheit) # but this is a list
```



```
$city
```

```
[1] Montreal Toronto New York
```

```
Levels: Montreal New York Toronto
```

```
$day1
```

```
[1] 68.0 73.4 82.4
```

```
$day2
```

```
[1] 77.0 78.8 95.0
```

```
$day3
```

```
[1] 86.0 66.2 89.6
```

```
as.data.frame(lapply(df, to_fahrenheit))
```

```
  city day1 day2 day3
1 Montreal 68.0 77.0 86.0
2 Toronto 73.4 78.8 66.2
3 New York 82.4 95.0 89.6
```

## 5 Multilevel data —

### 5.1 Extensions of apply functions —

```
library(spida2)
library(lattice)
library(latticeExtra)
```

```
Attaching package: 'latticeExtra'
```

```
The following object is masked from 'package:ggplot2':
```

```
layer
```

Data on math achievement tests in high schools in US 1977 students in 40 schools: 21 Catholic and 19 Public variables: - school id - mathach math achievement - ses socioeconomic status - Sex: Female Male - Minority status: Yes or No - Size of the school - Sector: Catholic or Public - PRACAD: priority given to academics in school - DISCLIM: disciplinary climate of school

```
head(hs)
```

```
   school mathach    ses    Sex Minority Size  Sector PRACAD
1   1317   12.862  0.882 Female      No   455 Catholic  0.95
2   1317    8.961  0.932 Female     Yes   455 Catholic  0.95
3   1317    4.756 -0.158 Female     Yes   455 Catholic  0.95
4   1317   21.405  0.362 Female     Yes   455 Catholic  0.95
5   1317   20.748  1.372 Female      No   455 Catholic  0.95
6   1317   18.362  0.132 Female     Yes   455 Catholic  0.95
```

```
DISCLIM
```

```
1  -1.694
2  -1.694
3  -1.694
4  -1.694
5  -1.694
6  -1.694
```

Note that the first use of 'hs' copies 'hs' from spida2. Changes that you make are only local. If you want to get the original back from spida2, use:

```
data(hs)
```

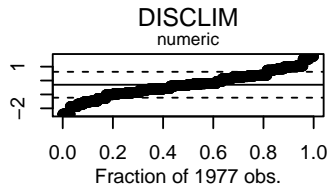
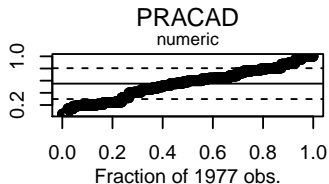
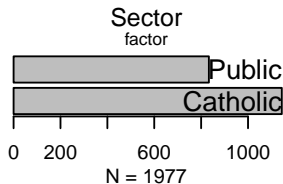
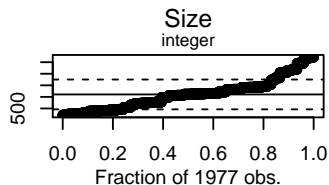
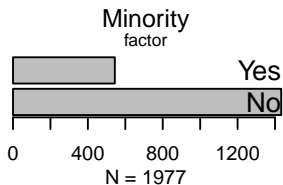
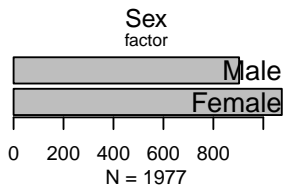
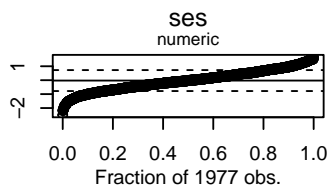
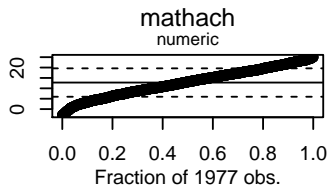
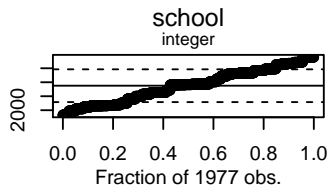
or, if it's necessary to be more specific:

```
data(hs, package = 'spida2')
```

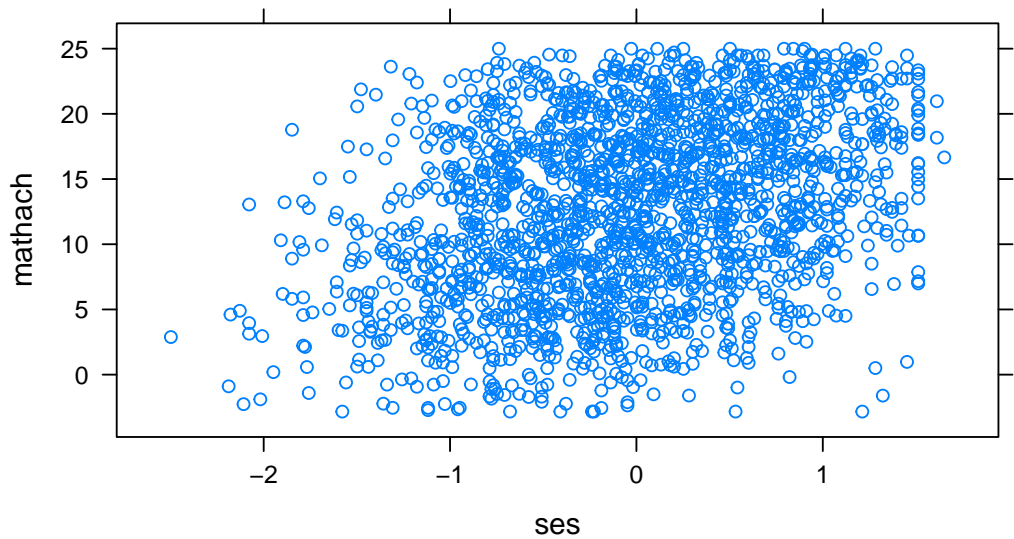
```
dim(hs)
```

```
[1] 1977    9
```

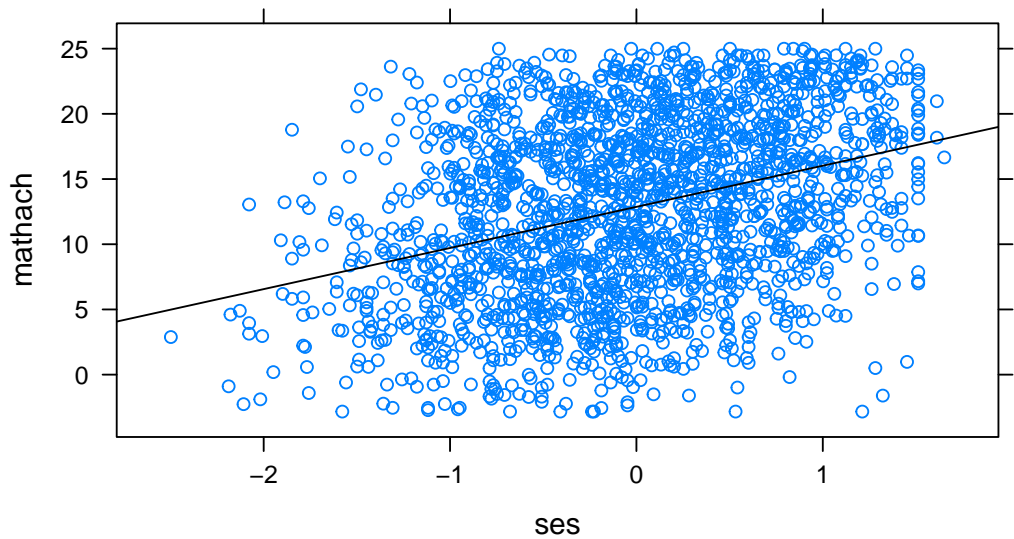
```
xqplot(hs)
```



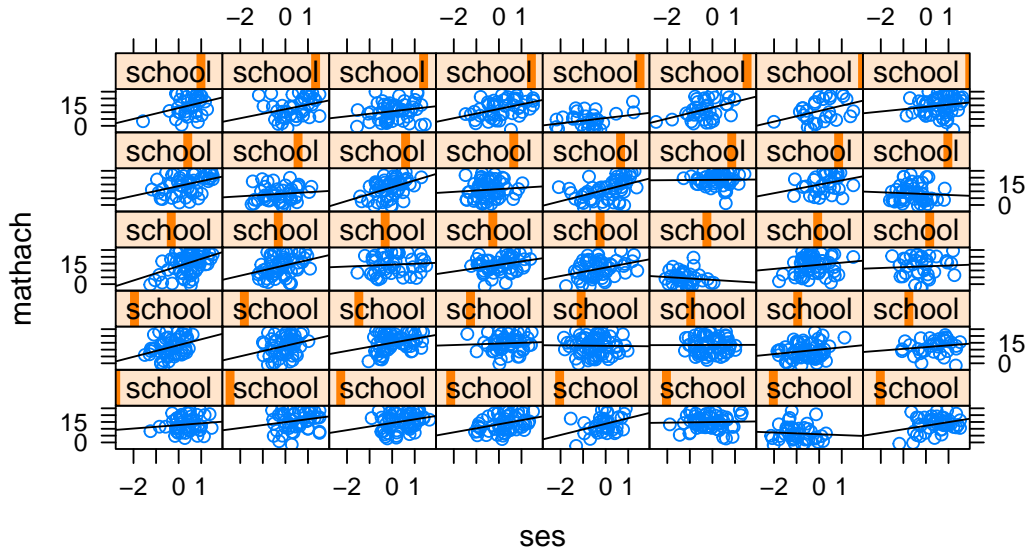
```
xyplot(mathach ~ ses, hs)
```



```
xyplot(mathach ~ ses, hs) + layer(panel.lmline(...))
```

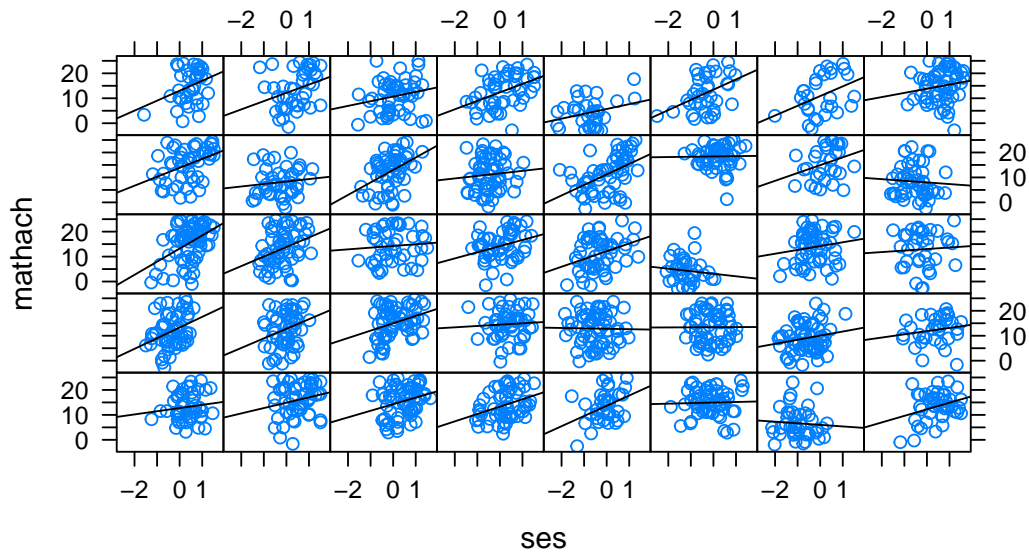


```
xyplot(mathach ~ ses | school, hs) + layer(panel.lmline(...))
```

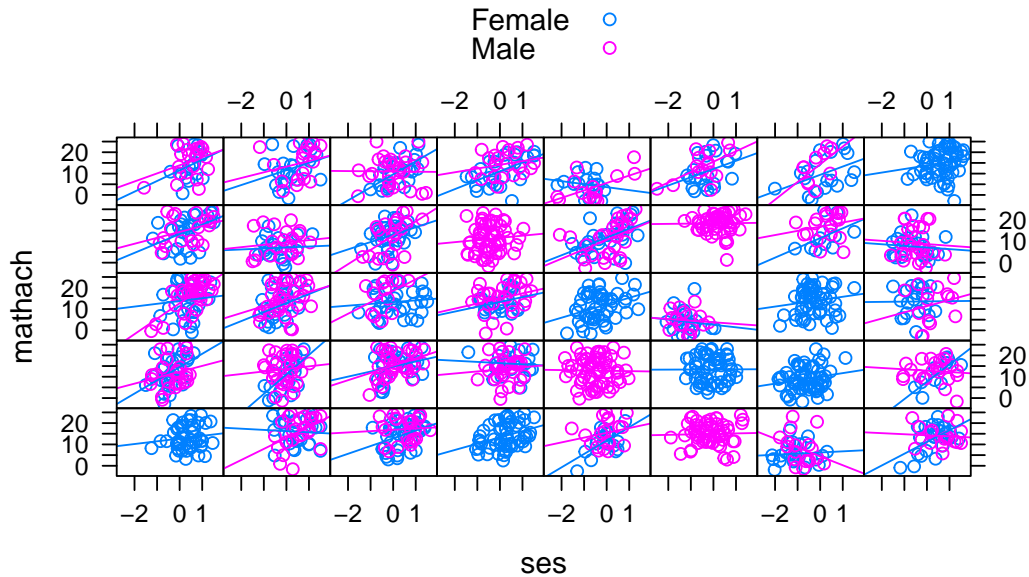


```
xyplot(mathach ~ ses | school, hs, strip = FALSE) +
  layer(panel.lmline(...))
```





```
xyplot(mathach ~ ses | school, hs,  
       groups = Sex, strip = FALSE,  
       auto.key = T) +  
glayer(panel.lmline(...))
```



Note: Two types of variables:

- student-level variables vary from student to student:
  - Synonyms: micro or level 1 variables
- school-level variables vary from school to school but constant within schools
  - Synonyms: macro or level 2 variables, contextual variable
- could have additional levels: School Board, State, etc.

We can use the `tapply` function to get information on individual schools

```
tapply(hs$mathach, hs$school, mean)
```

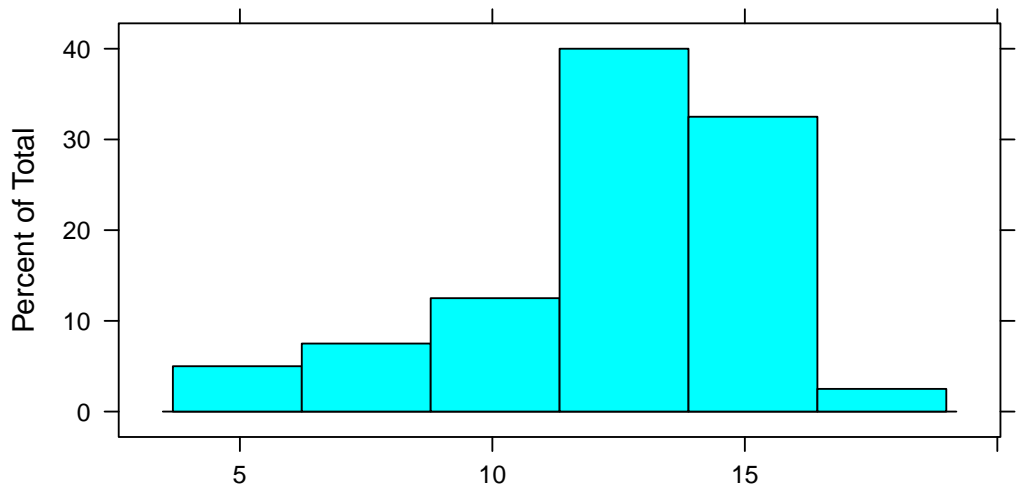
1317	1906	2208	2458	2626	2629
13.177687	15.983170	15.404667	13.985684	13.396605	14.907772
2639	2658	2771	3013	3610	3992
6.615476	13.396156	11.844109	12.610830	15.354953	14.645208
4292	4511	4530	4868	5619	5640
12.864354	13.409034	9.055698	12.310176	15.416242	13.160105
5650	5720	5761	5762	6074	6484
14.273533	14.282302	11.138058	4.324865	13.779089	12.912400
6897	7172	7232	7342	7345	7688

15.097633	8.066818	12.542635	11.166414	11.338554	18.422315
7697	7890	7919	8531	8627	8707
15.721781	8.341098	14.849973	13.528683	10.883717	12.883938
8854	8874	9550	9586		
4.239781	12.055028	11.089138	14.863695		

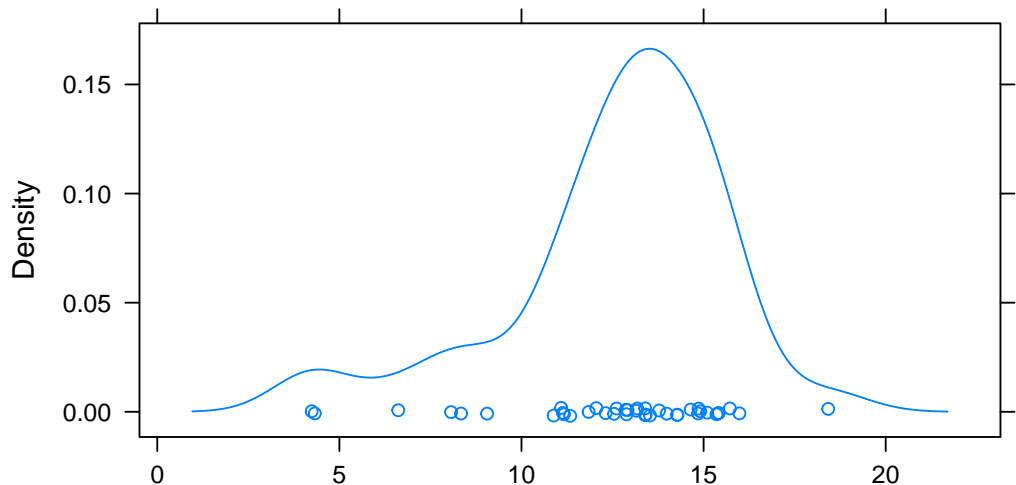
`tapply(Y, id, function, extra arguments):`

- apply 'function' to each chunk of 'Y' created by levels of 'id',
- use 'id' for names

```
library(latticeExtra)
tapply(hs$mathach, hs$school, mean) %>% histogram
```



```
tapply(hs$mathach, hs$school, mean) %>% densityplot
```



But often it's more useful to have the result incorporated back into the data set. We can use `spida2::capply`

```

hs <-
  within(hs, {
    mathach_mean <- capply(mathach, school, mean)
  }
)
head(hs)

```

	school	mathach	ses	Sex	Minority	Size	Sector	PRACAD
1	1317	12.862	0.882	Female	No	455	Catholic	0.95
2	1317	8.961	0.932	Female	Yes	455	Catholic	0.95
3	1317	4.756	-0.158	Female	Yes	455	Catholic	0.95
4	1317	21.405	0.362	Female	Yes	455	Catholic	0.95
5	1317	20.748	1.372	Female	No	455	Catholic	0.95
6	1317	18.362	0.132	Female	Yes	455	Catholic	0.95

	DISCLIM	mathach_mean
1	-1.694	13.17769
2	-1.694	13.17769
3	-1.694	13.17769
4	-1.694	13.17769



```
5 -1.694      13.17769
6 -1.694      13.17769
```

```
some(hs)  # random selection of rows (from car)
```

```
      school mathach      ses      Sex Minority Size      Sector PRACAD
61      1906  22.107  0.592   Male         No   400 Catholic   0.87
165     2458   7.814 -1.058 Female        Yes  545 Catholic   0.89
351     2639   8.978 -1.458   Male        Yes 2713   Public    0.14
421     2771  22.896 -0.048 Female        No   415   Public    0.24
786     4530   6.276  0.042 Female        No   435 Catholic  0.60
1213    6484  21.653 -0.378   Male        No   726   Public    0.19
1273    6897  19.830  1.012 Female        No  1415   Public    0.55
1289    6897  23.584  1.372 Female        Yes  1415   Public    0.55
1492    7345   8.157 -0.798 Female        Yes   978   Public    0.64
1634    7890   3.414 -1.598   Male        No   311   Public    0.21
      DISCLIM mathach_mean
61      -0.939      15.983170
165     -1.484      13.985684
```

351	-0.282	6.615476
421	1.048	11.844109
786	-0.245	9.055698
1213	0.218	12.912400
1273	-0.361	15.097633
1289	-0.361	15.097633
1492	0.336	11.338554
1634	0.845	8.341098

Like `tapply` but return a vector that has the same shape as `Y`

Creative use of functions gives broad possibilities

How variable is `mathach` in each school?

```
hs <- within(  
  hs,  
  {  
    mathach_sd <- capply(mathach, school, sd)  
    ses_sd <- capply(ses, school, sd)  
  }  
)
```

)

These variables can be called ‘sample computed contextual’ variables because they would be different for a different sample.

`capply` can also be used for within-school transformations that do not produce contextual variables.

e.g. within-school ranks

```
hs <- within(  
  hs,  
  {  
    mathach_rk <- capply(mathach, school, rank)  
  }  
)  
some(hs)
```

	school	mathach	ses	Sex	Minority	Size	Sector	PRACAD
19	1317	11.027	0.722	Female	Yes	455	Catholic	0.95

95	1906	-1.603	0.282	Male	Yes	400	Catholic	0.87
111	2208	7.847	-0.478	Female	No	1061	Catholic	0.68
145	2208	18.196	-0.468	Male	No	1061	Catholic	0.68
814	4868	19.078	0.902	Female	No	657	Catholic	1.00
905	5619	23.062	0.932	Male	No	1118	Catholic	0.77
1014	5720	19.196	0.432	Female	No	381	Catholic	0.65
1147	5762	-2.832	-1.578	Female	Yes	1826	Public	0.24
1813	8707	15.624	-0.238	Female	No	1133	Public	0.48
1901	9550	5.282	-0.538	Male	No	1532	Public	0.45

	DISCLIM	mathach_mean	ses_sd	mathach_sd	mathach_rk
19	-1.694	13.177687	0.5561583	5.462586	20
95	-0.939	15.983170	0.6135833	6.515435	1
111	-0.864	15.404667	0.5981188	6.122802	10
145	-0.864	15.404667	0.5981188	6.122802	39
814	-0.219	12.310176	0.7100080	5.432838	33
905	-1.286	15.416242	0.5972748	7.280409	57
1014	-0.352	14.282302	0.6641693	5.694073	43
1147	0.364	4.324865	0.5154149	4.993969	1
1813	1.542	12.883938	0.8042577	6.435737	30

1901 0.791 11.089138 0.7847035 7.877998 8

within-school deviations

```
hs <- within(  
  hs,  
  {  
    mathach_dev <- mathach - capply(mathach, school, mean)  
    ses_dev <- ses - capply(ses, school, mean)  
  }  
)  
some(hs)
```

	school	mathach	ses	Sex	Minority	Size	Sector	PRACAD
150	2208	21.890	0.142	Male	No	1061	Catholic	0.68
550	3610	15.896	0.792	Male	Yes	1431	Catholic	0.80
587	3992	9.989	0.072	Female	Yes	1114	Catholic	0.73
918	5640	7.582	-0.518	Female	No	1152	Public	0.41
1015	5720	13.754	-0.838	Female	No	381	Catholic	0.65
1322	7172	5.788	-0.958	Female	Yes	280	Catholic	0.05

1478	7345	24.993	1.282	Male	No	978	Public	0.64
1674	7919	19.043	0.762	Female	No	1451	Public	0.50
1762	8627	3.442	0.002	Male	Yes	2452	Public	0.25
1782	8707	1.850	-1.028	Female	No	1133	Public	0.48

	DISCLIM	mathach_mean	ses_sd	mathach_sd	mathach_rk
150	-0.864	15.404667	0.5981188	6.122802	50
550	-0.621	15.354953	0.6316415	5.894163	33
587	-1.534	14.645208	0.6054177	5.592953	12
918	0.256	13.160105	0.5830261	7.102322	14
1015	-0.352	14.282302	0.6641693	5.694073	27
1322	1.013	8.066818	0.6764417	5.610555	18
1478	0.336	11.338554	0.8257296	7.246025	56
1674	-0.402	14.849973	0.5367005	6.804286	25
1762	0.742	10.883717	0.7077276	6.489265	9
1782	1.542	12.883938	0.8042577	6.435737	2

	ses_dev	mathach_dev
150	-0.2811667	6.4853333
550	0.6718750	0.5410469
587	-0.2933962	-4.6562075

```
918 -0.3414035 -5.5781053
1015 -0.8705660 -0.5283019
1322 -0.6681818 -2.2788182
1478 1.2487500 13.6544464
1674 0.3043243 4.1930270
1762 -0.1028302 -7.4417170
1782 -1.1831250 -11.0339375
```

```
lm(mathach_dev ~ ses_dev, hs)
```

Call:

```
lm(formula = mathach_dev ~ ses_dev, data = hs)
```

Coefficients:

```
(Intercept)      ses_dev
 1.149e-16      2.223e+00
```

```
lm(mathach_dev ~ ses_dev, hs) %>% summary
```

Call:

```
lm(formula = mathach_dev ~ ses_dev, data = hs)
```

Residuals:

	Min	1Q	Median	3Q	Max
	-19.0093	-4.4831	0.2262	4.7600	17.0043

Coefficients:

	Estimate	Std. Error	t value	Pr(> t )
(Intercept)	1.149e-16	1.366e-01	0.00	1
ses_dev	2.223e+00	2.145e-01	10.37	<2e-16 ***

---

Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 6.072 on 1975 degrees of freedom



Multiple R-squared: 0.05159, Adjusted R-squared: 0.05111  
F-statistic: 107.4 on 1 and 1975 DF, p-value: < 2.2e-16

```
lm(mathach ~ ses + factor(school), hs) %>% summary
```

Call:

```
lm(formula = mathach ~ ses + factor(school), data = hs)
```

Residuals:

Min	1Q	Median	3Q	Max
-19.0093	-4.4831	0.2262	4.7600	17.0043

Coefficients:

	Estimate	Std. Error	t value	Pr(> t )	
(Intercept)	12.40995	0.88839	13.969	< 2e-16	***
ses	2.22319	0.21664	10.262	< 2e-16	***
factor(school)1906	2.43579	1.22255	1.992	0.04647	*
factor(school)2208	2.05394	1.18778	1.729	0.08393	.

factor(school)2458	1.06932	1.20174	0.890	0.37368	
factor(school)2626	1.13081	1.33469	0.847	0.39696	
factor(school)2629	2.80384	1.20602	2.325	0.02018	*
factor(school)2639	-3.65037	1.32655	-2.752	0.00598	**
factor(school)2658	0.01146	1.27276	0.009	0.99281	
factor(school)2771	0.18883	1.22047	0.155	0.87706	
factor(school)3013	0.29921	1.22493	0.244	0.80705	
factor(school)3610	2.67795	1.17207	2.285	0.02243	*
factor(school)3992	1.42292	1.22203	1.164	0.24441	
factor(school)4292	1.53522	1.18100	1.300	0.19378	
factor(school)4511	1.23727	1.20074	1.030	0.30294	
factor(school)4530	-2.02725	1.19262	-1.700	0.08932	.
factor(school)4868	-0.90260	1.37476	-0.657	0.51155	
factor(school)5619	2.07182	1.16354	1.781	0.07513	.
factor(school)5640	1.14276	1.20678	0.947	0.34378	
factor(school)5650	1.81369	1.27452	1.423	0.15489	
factor(school)5720	1.79995	1.22390	1.471	0.14154	
factor(school)5761	-0.55380	1.23610	-0.448	0.65419	
factor(school)5762	-5.43072	1.38255	-3.928	8.86e-05	***

factor(school)6074	1.98441	1.21387	1.635	0.10226	
factor(school)6484	0.91406	1.36804	0.668	0.50412	
factor(school)6897	1.91057	1.24550	1.534	0.12520	
factor(school)7172	-3.69881	1.28742	-2.873	0.00411	**
factor(school)7232	0.33303	1.23121	0.270	0.78681	
factor(school)7342	-0.24793	1.20900	-0.205	0.83754	
factor(school)7345	-1.14531	1.20826	-0.948	0.34330	
factor(school)7688	5.59910	1.21712	4.600	4.49e-06	***
factor(school)7697	2.73770	1.39980	1.956	0.05064	.
factor(school)7890	-2.90678	1.24761	-2.330	0.01992	*
factor(school)7919	1.42253	1.34195	1.060	0.28926	
factor(school)8531	0.21146	1.30432	0.162	0.87123	
factor(school)8627	-1.75929	1.22313	-1.438	0.15050	
factor(school)8707	0.12912	1.25258	0.103	0.91791	
factor(school)8854	-6.48777	1.41989	-4.569	5.20e-06	***
factor(school)8874	0.40269	1.36036	0.296	0.76725	
factor(school)9550	-1.43872	1.44385	-0.996	0.31916	
factor(school)9586	1.07281	1.19362	0.899	0.36888	

---

Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 6.133 on 1936 degrees of freedom

Multiple R-squared: 0.2118, Adjusted R-squared: 0.1955

F-statistic: 13 on 40 and 1936 DF, p-value: < 2.2e-16

Other contextual variables:

Proportion of women in each school

```
hs <- within(  
  hs,  
  {  
    female_prop <- capply(Sex == 'Female', school, mean)  
  }  
)  
some(hs)
```

	school	mathach	ses	Sex	Minority	Size	Sector	PRACAD
190	2458	15.496	1.342	Female	No	545	Catholic	0.89

475	3013	16.722	-0.428	Female	No	760	Public	0.56
491	3013	13.009	0.232	Male	No	760	Public	0.56
582	3992	8.693	-0.988	Male	No	1114	Catholic	0.73
917	5640	1.427	-0.768	Female	No	1152	Public	0.41
995	5650	8.526	-0.288	Female	Yes	720	Catholic	0.60
1457	7345	8.788	-1.528	Female	Yes	978	Public	0.64
1542	7688	12.146	-0.068	Male	No	1410	Catholic	0.65
1567	7697	7.015	0.562	Female	No	1734	Public	0.20
1758	8627	18.999	1.012	Female	No	2452	Public	0.25

	DISCLIM	mathach_mean	ses_sd	mathach_sd	mathach_rk
190	-1.484	13.98568	0.6584097	5.848459	36
475	-0.213	12.61083	0.4799328	6.985697	36
491	-0.213	12.61083	0.4799328	6.985697	26
582	-1.534	14.64521	0.6054177	5.592953	7
917	0.256	13.16011	0.5830261	7.102322	4
995	-0.070	14.27353	0.7777414	6.479535	13
1457	0.336	11.33855	0.8257296	7.246025	21
1542	-0.575	18.42231	0.5644347	4.498507	5
1567	0.279	15.72178	0.6133712	6.621170	6

1758	0.742	10.88372	0.7077276	6.489265	49
	ses_dev	mathach_dev	female_prop		
190	1.1142105	1.5103158	1.0000000		
475	-0.3837736	4.1111698	0.3584906		
491	0.2762264	0.3981698	0.3584906		
582	-1.3533962	-5.9522075	0.3962264		
917	-0.5914035	-11.7331053	0.4210526		
995	-0.3104444	-5.7475333	0.7111111		
1457	-1.5612500	-2.5505536	0.5178571		
1542	-0.2538889	-6.2763148	0.0000000		
1567	0.3037500	-8.7067812	0.3437500		
1758	0.9071698	8.1152830	0.4528302		

School-level data set:

Normally, the data set will be at the ‘finest’ level of the data, here students.

If each student had been measured on more than one occasion then the finest level would be the ‘occasion’

But many analyses and graphic displays use the data at a higher level

up(hs, ~ school) # one row per school with level 2 (and higher) va

	school	Size	Sector	PRACAD	DISCLIM	mathach_mean	ses_sd
1317	1317	455	Catholic	0.95	-1.694	13.177687	0.5561583
1906	1906	400	Catholic	0.87	-0.939	15.983170	0.6135833
2208	2208	1061	Catholic	0.68	-0.864	15.404667	0.5981188
2458	2458	545	Catholic	0.89	-1.484	13.985684	0.6584097
2626	2626	2142	Public	0.40	0.142	13.396605	0.5601067
2629	2629	1314	Catholic	0.81	-0.613	14.907772	0.7063209
2639	2639	2713	Public	0.14	-0.282	6.615476	0.6186603
2658	2658	780	Catholic	0.79	-0.961	13.396156	0.6402846
2771	2771	415	Public	0.24	1.048	11.844109	0.5136955
3013	3013	760	Public	0.56	-0.213	12.610830	0.4799328
3610	3610	1431	Catholic	0.80	-0.621	15.354953	0.6316415
3992	3992	1114	Catholic	0.73	-1.534	14.645208	0.6054177
4292	4292	1328	Catholic	0.76	-0.674	12.864354	0.6511382
4511	4511	1068	Catholic	0.52	-1.872	13.409034	0.5813363
4530	4530	435	Catholic	0.60	-0.245	9.055698	0.6210062

4868	4868	657	Catholic	1.00	-0.219	12.310176	0.7100080
5619	5619	1118	Catholic	0.77	-1.286	15.416242	0.5972748
5640	5640	1152	Public	0.41	0.256	13.160105	0.5830261
5650	5650	720	Catholic	0.60	-0.070	14.273533	0.7777414
5720	5720	381	Catholic	0.65	-0.352	14.282302	0.6641693
5761	5761	215	Catholic	0.63	-0.892	11.138058	0.7122389
5762	5762	1826	Public	0.24	0.364	4.324865	0.5154149
6074	6074	2051	Catholic	0.32	-1.018	13.779089	0.6271235
6484	6484	726	Public	0.19	0.218	12.912400	0.6958345
6897	6897	1415	Public	0.55	-0.361	15.097633	0.7445231
7172	7172	280	Catholic	0.05	1.013	8.066818	0.6764417
7232	7232	1154	Public	0.20	0.975	12.542635	0.5743482
7342	7342	1220	Catholic	0.46	0.380	11.166414	0.5648459
7345	7345	978	Public	0.64	0.336	11.338554	0.8257296
7688	7688	1410	Catholic	0.65	-0.575	18.422315	0.5644347
7697	7697	1734	Public	0.20	0.279	15.721781	0.6133712
7890	7890	311	Public	0.21	0.845	8.341098	0.5932263
7919	7919	1451	Public	0.50	-0.402	14.849973	0.5367005
8531	8531	2190	Public	0.58	0.132	13.528683	0.6829747



8627	8627	2452	Public	0.25	0.742	10.883717	0.7077276
8707	8707	1133	Public	0.48	1.542	12.883938	0.8042577
8854	8854	745	Public	0.18	-0.228	4.239781	0.8036439
8874	8874	2650	Public	0.20	1.742	12.055028	0.7137251
9550	9550	1532	Public	0.45	0.791	11.089138	0.7847035
9586	9586	262	Catholic	1.00	-2.416	14.863695	0.5949914

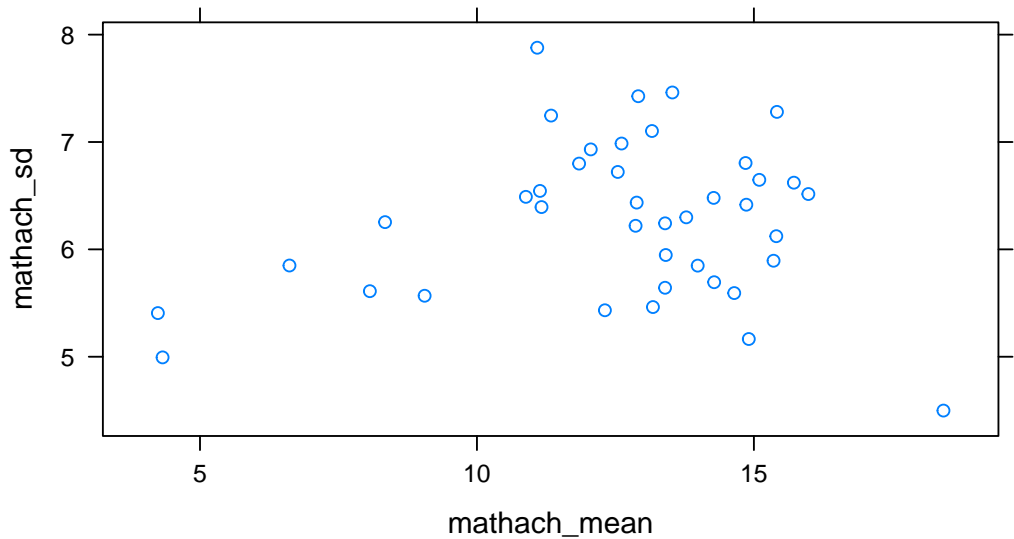
mathach\_sd female\_prop

1317	5.462586	1.0000000
1906	6.515435	0.5094340
2208	6.122802	0.5833333
2458	5.848459	1.0000000
2626	6.242649	0.4736842
2629	5.165071	0.0000000
2639	5.849492	0.5714286
2658	5.642341	0.6000000
2771	6.798981	0.5090909
3013	6.985697	0.3584906
3610	5.894163	0.4531250
3992	5.592953	0.3962264

4292	6.219492	0.0000000
4511	5.947499	1.0000000
4530	5.567967	1.0000000
4868	5.432838	0.3235294
5619	7.280409	0.4545455
5640	7.102322	0.4210526
5650	6.479535	0.7111111
5720	5.694073	0.4528302
5761	6.544368	1.0000000
5762	4.993969	0.5675676
6074	6.298483	1.0000000
6484	7.426520	0.5714286
6897	6.647635	0.5918367
7172	5.610555	0.5000000
7232	6.721008	0.5769231
7342	6.393930	0.0000000
7345	7.246025	0.5178571
7688	4.498507	0.0000000
7697	6.621170	0.3437500

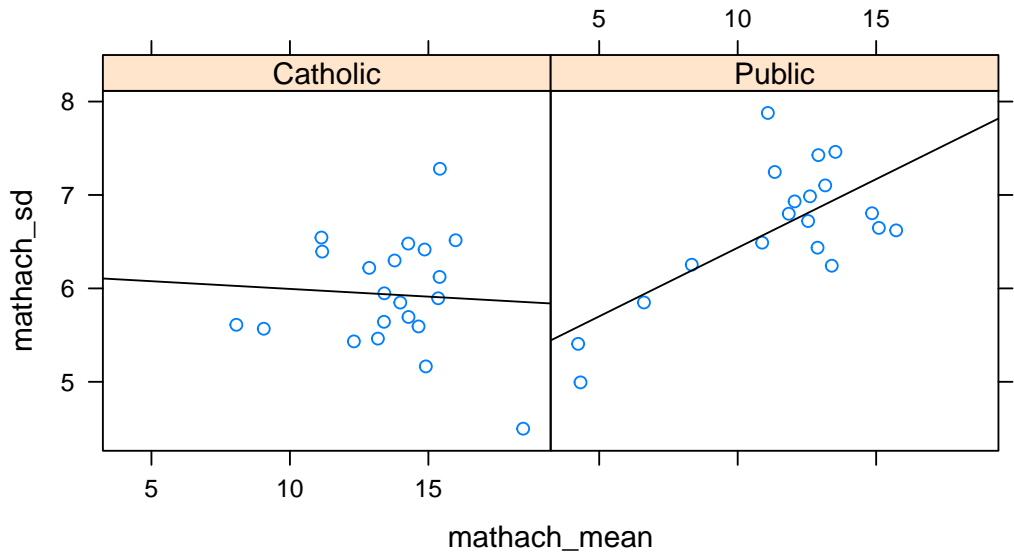
7890	6.253697	0.4705882
7919	6.804286	0.4324324
8531	7.461228	0.5609756
8627	6.489265	0.4528302
8707	6.435737	0.5416667
8854	5.406482	0.5312500
8874	6.931169	0.5833333
9550	7.877998	0.6551724
9586	6.416000	1.0000000

```
up(hs, ~ school) %>% xyplot(mathach_sd ~ mathach_mean, .)
```



```
up(hs, ~ school) %>% xyplot(mathach_sd ~ mathach_mean | Sector, .)
```





## Aggregating some variables that vary within schools

```
up(hs, ~school, ~Sex )
```

	school	Size	Sector	PRACAD	DISCLIM	mathach_mean	ses_sd
1317	1317	455	Catholic	0.95	-1.694	13.177687	0.5561583
1906	1906	400	Catholic	0.87	-0.939	15.983170	0.6135833
2208	2208	1061	Catholic	0.68	-0.864	15.404667	0.5981188
2458	2458	545	Catholic	0.89	-1.484	13.985684	0.6584097
2626	2626	2142	Public	0.40	0.142	13.396605	0.5601067
2629	2629	1314	Catholic	0.81	-0.613	14.907772	0.7063209
2639	2639	2713	Public	0.14	-0.282	6.615476	0.6186603
2658	2658	780	Catholic	0.79	-0.961	13.396156	0.6402846
2771	2771	415	Public	0.24	1.048	11.844109	0.5136955
3013	3013	760	Public	0.56	-0.213	12.610830	0.4799328
3610	3610	1431	Catholic	0.80	-0.621	15.354953	0.6316415
3992	3992	1114	Catholic	0.73	-1.534	14.645208	0.6054177
4292	4292	1328	Catholic	0.76	-0.674	12.864354	0.6511382
4511	4511	1068	Catholic	0.52	-1.872	13.409034	0.5813363

4530	4530	435	Catholic	0.60	-0.245	9.055698	0.6210062
4868	4868	657	Catholic	1.00	-0.219	12.310176	0.7100080
5619	5619	1118	Catholic	0.77	-1.286	15.416242	0.5972748
5640	5640	1152	Public	0.41	0.256	13.160105	0.5830261
5650	5650	720	Catholic	0.60	-0.070	14.273533	0.7777414
5720	5720	381	Catholic	0.65	-0.352	14.282302	0.6641693
5761	5761	215	Catholic	0.63	-0.892	11.138058	0.7122389
5762	5762	1826	Public	0.24	0.364	4.324865	0.5154149
6074	6074	2051	Catholic	0.32	-1.018	13.779089	0.6271235
6484	6484	726	Public	0.19	0.218	12.912400	0.6958345
6897	6897	1415	Public	0.55	-0.361	15.097633	0.7445231
7172	7172	280	Catholic	0.05	1.013	8.066818	0.6764417
7232	7232	1154	Public	0.20	0.975	12.542635	0.5743482
7342	7342	1220	Catholic	0.46	0.380	11.166414	0.5648459
7345	7345	978	Public	0.64	0.336	11.338554	0.8257296
7688	7688	1410	Catholic	0.65	-0.575	18.422315	0.5644347
7697	7697	1734	Public	0.20	0.279	15.721781	0.6133712
7890	7890	311	Public	0.21	0.845	8.341098	0.5932263
7919	7919	1451	Public	0.50	-0.402	14.849973	0.5367005



8531	8531	2190	Public	0.58	0.132	13.528683	0.6829747
8627	8627	2452	Public	0.25	0.742	10.883717	0.7077276
8707	8707	1133	Public	0.48	1.542	12.883938	0.8042577
8854	8854	745	Public	0.18	-0.228	4.239781	0.8036439
8874	8874	2650	Public	0.20	1.742	12.055028	0.7137251
9550	9550	1532	Public	0.45	0.791	11.089138	0.7847035
9586	9586	262	Catholic	1.00	-2.416	14.863695	0.5949914

	mathach_sd	female_prop	Sex_Female	Sex_Male
--	------------	-------------	------------	----------

1317	5.462586	1.0000000	1.0000000	0.0000000
1906	6.515435	0.5094340	0.5094340	0.4905660
2208	6.122802	0.5833333	0.5833333	0.4166667
2458	5.848459	1.0000000	1.0000000	0.0000000
2626	6.242649	0.4736842	0.4736842	0.5263158
2629	5.165071	0.0000000	0.0000000	1.0000000
2639	5.849492	0.5714286	0.5714286	0.4285714
2658	5.642341	0.6000000	0.6000000	0.4000000
2771	6.798981	0.5090909	0.5090909	0.4909091
3013	6.985697	0.3584906	0.3584906	0.6415094
3610	5.894163	0.4531250	0.4531250	0.5468750

3992	5.592953	0.3962264	0.3962264	0.6037736
4292	6.219492	0.0000000	0.0000000	1.0000000
4511	5.947499	1.0000000	1.0000000	0.0000000
4530	5.567967	1.0000000	1.0000000	0.0000000
4868	5.432838	0.3235294	0.3235294	0.6764706
5619	7.280409	0.4545455	0.4545455	0.5454545
5640	7.102322	0.4210526	0.4210526	0.5789474
5650	6.479535	0.7111111	0.7111111	0.2888889
5720	5.694073	0.4528302	0.4528302	0.5471698
5761	6.544368	1.0000000	1.0000000	0.0000000
5762	4.993969	0.5675676	0.5675676	0.4324324
6074	6.298483	1.0000000	1.0000000	0.0000000
6484	7.426520	0.5714286	0.5714286	0.4285714
6897	6.647635	0.5918367	0.5918367	0.4081633
7172	5.610555	0.5000000	0.5000000	0.5000000
7232	6.721008	0.5769231	0.5769231	0.4230769
7342	6.393930	0.0000000	0.0000000	1.0000000
7345	7.246025	0.5178571	0.5178571	0.4821429
7688	4.498507	0.0000000	0.0000000	1.0000000

7697	6.621170	0.3437500	0.3437500	0.6562500
7890	6.253697	0.4705882	0.4705882	0.5294118
7919	6.804286	0.4324324	0.4324324	0.5675676
8531	7.461228	0.5609756	0.5609756	0.4390244
8627	6.489265	0.4528302	0.4528302	0.5471698
8707	6.435737	0.5416667	0.5416667	0.4583333
8854	5.406482	0.5312500	0.5312500	0.4687500
8874	6.931169	0.5833333	0.5833333	0.4166667
9550	7.877998	0.6551724	0.6551724	0.3448276
9586	6.416000	1.0000000	1.0000000	0.0000000

So far, recap:

hs: level 1 data set, 'long' data set up(hs, ~ school): level 2 data set, 'short' data set

What if you want to add new level 2 data to the level 1 data set

```
states <- read.csv(text=  
'  
school,state
```

1317,New York  
1906,New York  
2208,New York  
2458,New York  
2626,New York  
2629,New York  
2639,New York  
2658,New York  
2771,New York  
3013,New York  
3610,Oregon  
3992,Oregon  
4292,Oregon  
4511,Oregon  
4530,Oregon  
4868,Oregon  
5619,Oregon  
5640,Oregon

5650,Oregon  
5720,West Virginia  
5761,West Virginia  
5762,West Virginia  
6074,West Virginia  
6484,West Virginia  
6897,West Virginia  
7172,West Virginia  
7232,West Virginia  
7342,West Virginia  
7345,West Virginia  
7688,South Dakota  
7697,South Dakota  
7890,South Dakota  
7919,South Dakota  
8531,South Dakota  
8627,South Dakota  
8707,South Dakota

```
8854,Vermont
```

```
8874,Vermont
```

```
9550,Vermont
```

```
9586,Vermont
```

```
' )
```

```
states # note that this is fictional
```

	school	state
1	1317	New York
2	1906	New York
3	2208	New York
4	2458	New York
5	2626	New York
6	2629	New York
7	2639	New York
8	2658	New York
9	2771	New York
10	3013	New York

11	3610	Oregon
12	3992	Oregon
13	4292	Oregon
14	4511	Oregon
15	4530	Oregon
16	4868	Oregon
17	5619	Oregon
18	5640	Oregon
19	5650	Oregon
20	5720	West Virginia
21	5761	West Virginia
22	5762	West Virginia
23	6074	West Virginia
24	6484	West Virginia
25	6897	West Virginia
26	7172	West Virginia
27	7232	West Virginia
28	7342	West Virginia
29	7345	West Virginia

30	7688	South Dakota
31	7697	South Dakota
32	7890	South Dakota
33	7919	South Dakota
34	8531	South Dakota
35	8627	South Dakota
36	8707	South Dakota
37	8854	Vermont
38	8874	Vermont
39	9550	Vermont
40	9586	Vermont

Merging states into hs

```
dm <- merge(hs, states, by = 'school', all.x = T) # left outer join  
dim(dm)
```

```
[1] 1977  17
```



some(dm)

	school	mathach	ses	Sex	Minority	Size	Sector	PRACAD
144	2208	23.529	0.932	Female	No	1061	Catholic	0.68
170	2458	17.205	0.262	Female	Yes	545	Catholic	0.89
592	3992	14.913	0.192	Male	Yes	1114	Catholic	0.73
605	3992	8.531	0.912	Male	Yes	1114	Catholic	0.73
972	5650	4.682	0.482	Female	Yes	720	Catholic	0.60
1290	6897	11.309	-1.188	Female	Yes	1415	Public	0.55
1556	7688	19.567	-1.278	Male	No	1410	Catholic	0.65
1566	7697	13.179	-0.718	Male	No	1734	Public	0.20
1615	7890	9.680	-1.518	Female	No	311	Public	0.21
1872	8874	7.779	-0.568	Female	No	2650	Public	0.20
	DISCLIM	mathach_mean	ses_sd	mathach_sd	mathach_rk			
144	-0.864	15.404667	0.5981188	6.122802	57.0			
170	-1.484	13.985684	0.6584097	5.848459	39.0			
592	-1.534	14.645208	0.6054177	5.592953	25.0			
605	-1.534	14.645208	0.6054177	5.592953	6.0			

972	-0.070	14.273533	0.7777414	6.479535	5.0
1290	-0.361	15.097633	0.7445231	6.647635	16.0
1556	-0.575	18.422315	0.5644347	4.498507	29.0
1566	0.279	15.721781	0.6133712	6.621170	12.5
1615	0.845	8.341098	0.5932263	6.253697	34.0
1872	1.742	12.055028	0.7137251	6.931169	11.0

	ses_dev	mathach_dev	female_prop	state
144	0.50883333	8.1243333	0.5833333	New York
170	0.03421053	3.2193158	1.0000000	New York
592	-0.17339623	0.2677925	0.3962264	Oregon
605	0.54660377	-6.1142075	0.3962264	Oregon
972	0.45955556	-9.5915333	0.7111111	Oregon
1290	-1.53755102	-3.7886327	0.5918367	West Virginia
1556	-1.46388889	1.1446852	0.0000000	South Dakota
1566	-0.97625000	-2.5427812	0.3437500	South Dakota
1615	-0.99529412	1.3389020	0.4705882	South Dakota
1872	-0.22722222	-4.2760278	0.5833333	Vermont

## 5.2 Merge examples —

```
grades <- read.table(header = TRUE, text =  
'  
student course    gp  
John    Calculus    3.5  
Mary    Algebra      3.9  
Paul    Calculus      3.2  
John    Statistics    3.9  
John    Algebra      3.9  
Mary    Statistics    4.0  
' )  
grades
```

	student	course	gp
1	John	Calculus	3.5
2	Mary	Algebra	3.9
3	Paul	Calculus	3.2
4	John	Statistics	3.9

```
5   John   Algebra 3.9
6   Mary  Statistics 4.0
```

```
courses <- read.table(header = TRUE, text =
'
course  credits
Calculus    6
Algebra     3
Statistics  3
')
courses
```

```
      course credits
1  Calculus      6
2   Algebra      3
3 Statistics      3
```

```
email <- read.table(header = T, text =
'
student email
```

```
John    john123
Paul    paul456
Walter  wally6
')
email
```

```
      student  email
1      John  john123
2      Paul  paul456
3      Walter wally6
```

### 5.2.1 Calculate GPA —

1. need weights

```
grades <- merge(grades, courses, by = 'course', all = T)
grades
```

```
      course student  gp credits
1      Algebra   Mary 3.9       3
```

2	Algebra	John	3.9	3
3	Calculus	John	3.5	6
4	Calculus	Paul	3.2	6
5	Statistics	John	3.9	3
6	Statistics	Mary	4.0	3

```
grades$gp_tot <- with(grades, capply(gp * credits, student, sum))
grades$credit_tot <- with(grades, capply(credits, student, sum))
```

2. weighted average

```
grades$gpa <- with(grades, gp_tot / credit_tot)
```

```
up(grades, ~ student)
```

	student	gp_tot	credit_tot	gpa
John	John	44.4	12	3.70
Mary	Mary	23.7	6	3.95
Paul	Paul	19.2	6	3.20

```
grade_report <- up(grades, ~ student)
grade_report
```

	student	gp_tot	credit_tot	gpa
	John	44.4	12	3.70
	Mary	23.7	6	3.95
	Paul	19.2	6	3.20

### 3. merge with email

```
merge(grade_report, email, by = 'student')
```

	student	gp_tot	credit_tot	gpa	email
1	John	44.4	12	3.7	john123
2	Paul	19.2	6	3.2	paul456

*# inner join, only students in BOTH files*

```
merge(grade_report, email, by = 'student', all = T)
```

	student	gp_tot	credit_tot	gpa	email
--	---------	--------	------------	-----	-------

```
1   John   44.4           12 3.70 john123
2   Mary   23.7           6 3.95   <NA>
3   Paul   19.2           6 3.20 paul456
4   Walter  NA             NA  NA    wally6
```

```
# outer join, students in EITHER files
```

```
merge(grade_report, email, by = 'student', all.x = T)
```

```
  student gp_tot credit_tot  gpa  email
1   John   44.4           12 3.70 john123
2   Mary   23.7           6 3.95   <NA>
3   Paul   19.2           6 3.20 paul456
```

```
# all rows in first file
```

Other way of using capply on data frames but not efficient with very large files

```
grades$gpa2 <- capply(grades, grades$student, with, sum(gp*credits)/
up(grades, ~ student)
```

```
  student gp_tot credit_tot  gpa gpa2
```



John	John	44.4	12	3.70	3.70
Mary	Mary	23.7	6	3.95	3.95
Paul	Paul	19.2	6	3.20	3.20

Create transcripts

Add course average to student file

```
grades$course_average <-
  with(grades, capply(gp, course, mean)) # no weights! why?
```

List of transcripts

```
split(grades, grades$student)
```

\$John

	course	student	gp	credits	gp_tot	credit_tot	gpa	gpa2
2	Algebra	John	3.9	3	44.4	12	3.7	3.7
3	Calculus	John	3.5	6	44.4	12	3.7	3.7
5	Statistics	John	3.9	3	44.4	12	3.7	3.7
	course_average							

2	3.90
3	3.35
5	3.95

\$Mary

	course	student	gp	credits	gp_tot	credit_tot	gpa	gpa2
1	Algebra	Mary	3.9	3	23.7	6	3.95	3.95
6	Statistics	Mary	4.0	3	23.7	6	3.95	3.95

course\_average

1	3.90
6	3.95

\$Paul

	course	student	gp	credits	gp_tot	credit_tot	gpa	gpa2
4	Calculus	Paul	3.2	6	19.2	6	3.2	3.2

course\_average

4	3.35
---	------

## 6 The many ways of referring to variables —

A confusing aspect of R is that there are many ways to refer to an object

- name: if an object is in the current environment

```
grades
```

```
      course student  gp credits gp_tot credit_tot  gpa gpa2
1   Algebra   Mary 3.9      3   23.7         6 3.95 3.95
2   Algebra   John 3.9      3   44.4        12 3.70 3.70
3  Calculus   John 3.5      6   44.4        12 3.70 3.70
4  Calculus   Paul 3.2      6   19.2         6 3.20 3.20
5 Statistics   John 3.9      3   44.4        12 3.70 3.70
6 Statistics   Mary 4.0      3   23.7         6 3.95 3.95
```

```
course_average
```

```
1      3.90
2      3.90
3      3.35
4      3.35
```

```
5          3.95
6          3.95
```

- selecting from a data frame (FQN: fully qualified name)

```
grades$student
```

```
[1] Mary John John Paul John Mary
Levels: John Mary Paul
```

```
grades[['student']]
```

```
[1] Mary John John Paul John Mary
Levels: John Mary Paul
```

```
grades['student'] # but this is the data frame with the variable
```

```
  student
1   Mary
2   John
3   John
```

4 Paul  
5 John  
6 Mary

- by name using with or within

```
with(grades, student)
```

```
[1] Mary John John Paul John Mary  
Levels: John Mary Paul
```

```
grades <- within(grades, {  
  gpa3 <- capply(gp * credits, student, sum)/capply(credits, student,  
  })  
grades
```

	course	student	gp	credits	gp_tot	credit_tot	gpa	gpa2
1	Algebra	Mary	3.9	3	23.7	6	3.95	3.95
2	Algebra	John	3.9	3	44.4	12	3.70	3.70
3	Calculus	John	3.5	6	44.4	12	3.70	3.70
4	Calculus	Paul	3.2	6	19.2	6	3.20	3.20

5	Statistics	John	3.9	3	44.4	12	3.70	3.70
6	Statistics	Mary	4.0	3	23.7	6	3.95	3.95
	course_average		gpa3					
1			3.90				3.95	
2			3.90				3.70	
3			3.35				3.70	
4			3.35				3.20	
5			3.95				3.70	
6			3.95				3.95	

- as argument to a function

```
sum(grades$credits)
```

```
[1] 24
```

- formula

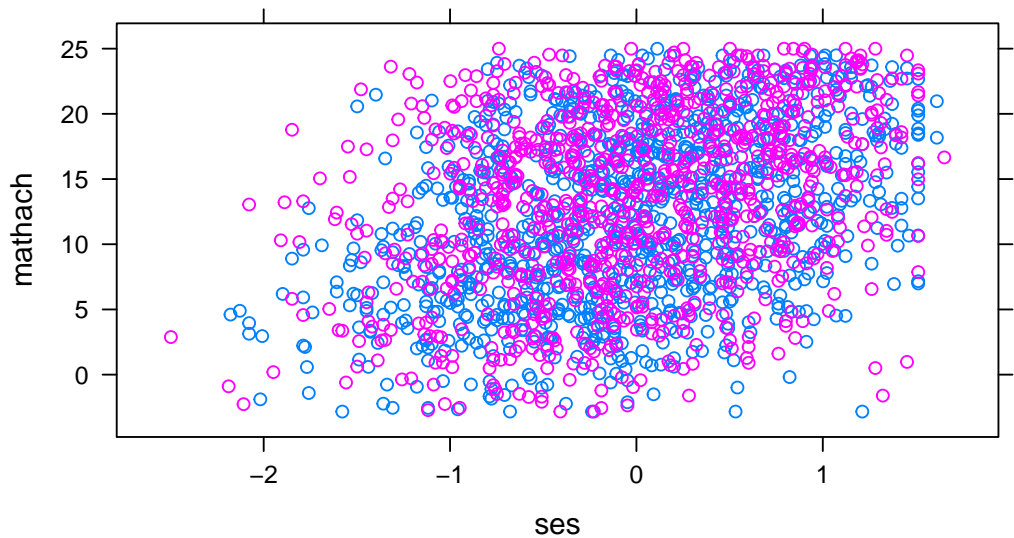
```
tab(grades, ~ student)
```

```
student
```

John	Mary	Paul	Total
3	2	1	6

- name in a data frame that is another argument

```
xyplot(mathach ~ ses, hs, group = Sex)
```



Note: in this example mathach and ses are referenced by a formula interpreted in `hs` but 'Sex' is interpreted by name interpreted with `hs`



- by name in a character string

```
merge(grades, courses, by = 'course') # can't just use: by = cours
```

	course	student	gp	credits.x	gp_tot	credit_tot	gpa	gpa2
1	Algebra	Mary	3.9	3	23.7	6	3.95	3.95
2	Algebra	John	3.9	3	44.4	12	3.70	3.70
3	Calculus	John	3.5	6	44.4	12	3.70	3.70
4	Calculus	Paul	3.2	6	19.2	6	3.20	3.20
5	Statistics	John	3.9	3	44.4	12	3.70	3.70
6	Statistics	Mary	4.0	3	23.7	6	3.95	3.95
	course_average	gpa3	credits.y					
1		3.90	3.95	3				
2		3.90	3.70	3				
3		3.35	3.70	6				
4		3.35	3.20	6				
5		3.95	3.70	3				
6		3.95	3.95	3				

Why so many ways that seem – and are – completely inconsistent

Because R is an evolving language that tries to be backward compatible.

In the early days (of S) data frames didn't even exist. To run a regression you needed the X matrix and the y vector and use 'lsfit(X,y)'.

Formulas, environments, etc. were all added gradually. When a new idea, formulas for example, is added to R, many people writing packages think it's cool and start using it. So a lot of packages written since 2000 make heavy use of formulas to refer to variables. But the old original functions often don't.

Some additions really catch on, e.g. pipes: %>%, which are just a few years old.

## 7 OOP: Object-oriented programming —-

- 'generic function': a function that selects another function to perform a task. The selection is based on the 'class' of the object that is the first argument of the generic function.
- 'method': a function called by a generic function depending on the class of the object.

## Example:

```
to_fahrenheit
```

```
function(x) {  
  if(is.factor(x) || !is.numeric(x) ) x # why 'is.factor'?  
  else 32 + (9/5)*x  
}  
<bytecode: 0x0000000018f98058>
```

Omesh (a student in 2018) asked ‘why can’t we write it to use it on a data frame?’ But we would also like to use it on variables because sometimes it won’t be every numeric variable that’s a temperature in C.

First: note that the objects we work on have ‘classes’

```
class(2.3)
```

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

```
class(1:2)
```

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

```
class(factor('a'))
```

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

```
class('ab')
```

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

```
class(df)
```

```
[1] "data.frame"
```

Note that, in contrast with 'is.numeric', classes distinguish between numeric and factor.

Generic function:

```
to_fahrenheit <- function(x,...) {  
  UseMethod('to_fahrenheit')  
}
```

'to\_fahrenheit' will look at the class of X and use one of the following methods.

Methods:

```
to_fahrenheit.numeric <- function(x,...) {  
  32 + (9/5)*x  
}
```

```
to_fahrenheit.default <- function(x,...) {  
  x # for any other class  
}
```

But what about data frames??

```
to_fahrenheit.data.frame <- function(x,...) {  
  as.data.frame(lapply(x, to_fahrenheit))  
}
```

Let's try this out

```
to_fahrenheit(0)
```

```
[1] 32
```

```
to_fahrenheit(37)
```

```
[1] 98.6
```

```
to_fahrenheit(-273.15)
```

```
[1] -459.67
```

```
to_fahrenheit(100)
```

```
[1] 212
```

```
to_fahrenheit(factor(0))
```

```
[1] 0
```

```
Levels: 0
```

```
to_fahrenheit(c('absolute zero', 'boiling point'))
```

```
[1] "absolute zero" "boiling point"
```

```
df
```

```
      city day1 day2 day3
1 Montreal   20   25   30
2 Toronto   23   26   19
3 New York   28   35   32
```

```
class(df)
```

```
[1] "data.frame"
```

```
to_fahrenheit.data.frame(df)
```

```
      city day1 day2 day3
1 Montreal 68.0 77.0 86.0
2 Toronto 73.4 78.8 66.2
3 New York 82.4 95.0 89.6
```

```
to_fahrenheit(df)
```

```
      city day1 day2 day3
1 Montreal 68.0 77.0 86.0
2  Toronto 73.4 78.8 66.2
3  New York 82.4 95.0 89.6
```

I can use 'to\_fahrenheit' to do 'anything'!!

```
methods(to_fahrenheit)
```

```
[1] to_fahrenheit.data.frame to_fahrenheit.default
[3] to_fahrenheit.numeric
see '?methods' for accessing help and source code
```

The above illustrate creating a 'generic function' and 'methods' for existing classes: here 'numeric', 'integer', 'default' and 'data.frame'

If an object has a class attribute, e.g. data.frames, then the value of the attribute is its class.



If it doesn't have a class attribute then it has an implicit class depending on its type and structure. For example, matrices has the class "matrix" whether their content is numeric or character. However, for a vector, class is 'integer' for an integer, 'numeric' for a double, 'character' for a character.

Is there an underlying logic to it all?

## **7.1 Creating a new class —**

I can define new methods for other classes

### **7.1.1 Creating a class and methods for existing generics —**

Many functions are 'generic', e.g. print, summary

So when you create a new statistical methods you can write a function that creates a new class

then you can write methods for your new class

e.g. lm

```
fit <- lm(day1 ~ day2, df)
class(fit)
```

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

```
print(fit)
```

Call:

```
lm(formula = day1 ~ day2, data = df)
```

Coefficients:

(Intercept)	day2
3.5055	0.7033

```
summary(fit)
```

Call:

```
lm(formula = day1 ~ day2, data = df)
```

Residuals:

1	2	3
-1.0879	1.2088	-0.1209

Coefficients:

	Estimate	Std. Error	t value	Pr(> t )
(Intercept)	3.5055	6.0753	0.577	0.667
day2	0.7033	0.2094	3.359	0.184

Residual standard error: 1.631 on 1 degrees of freedom

Multiple R-squared: 0.9186, Adjusted R-squared: 0.8372

F-statistic: 11.28 on 1 and 1 DF, p-value: 0.1842

what function is actually used?

```
methods(class = 'lm')
```

[1] add1	alias	anova
[4] Anova	avPlot	Boot

[7]	bootCase	boxCox	brief
[10]	case.names	ceresPlot	coerce
[13]	confidenceEllipse	confint	Confint
[16]	cooks.distance	crPlot	deltaMethod
[19]	deviance	dfbeta	dfbetaPlots
[22]	dfbetas	dfbetasPlots	drop1
[25]	dummy.coef	durbinWatsonTest	effects
[28]	extractAIC	family	formula
[31]	fortify	getData	getFix
[34]	hatvalues	hccm	infIndexPlot
[37]	influence	influencePlot	initialize
[40]	inverseResponsePlot	kappa	labels
[43]	leveneTest	leveragePlot	linearHypothesis
[46]	logLik	mcPlot	mmp
[49]	model.frame	model.matrix	ncvTest
[52]	nextBoot	nobs	outlierTest
[55]	plot	powerTransform	predict
[58]	Predict	print	proj
[61]	qqnorm	qqPlot	qr

```
[64] residualPlot      residualPlots      residuals
[67] rstandard          rstudent          S
[70] show                sigmaHat          simulate
[73] slotsFromS3         spreadLevelPlot   summary
[76] variable.names      vcov
```

see '?methods' for accessing help and source code

great way to get ideas about what to do with an object!

```
getS3method('print', 'lm') # most methods are 'S3', a few are 'S4'
```

```
function (x, digits = max(3L, getOption("digits") - 3L), ...)
{
  cat("\nCall:\n", paste(deparse(x$call), sep = "\n", collapse =
    "\n\n", sep = ""))
  if (length(coef(x))) {
    cat("Coefficients:\n")
    print.default(format(coef(x), digits = digits), print.gap
      quote = FALSE)
  }
}
```

```
else cat("No coefficients\n")
cat("\n")
invisible(x)
}
<bytecode: 0x00000000354b70b8>
<environment: namespace:stats>
```

```
getS3method('summary', 'lm')
```

```
function (object, correlation = FALSE, symbolic.cor = FALSE,
  ...)
{
  z <- object
  p <- z$rank
  rdf <- z$df.residual
  if (p == 0) {
    r <- z$residuals
    n <- length(r)
    w <- z$weights
```

```
if (is.null(w)) {
  rss <- sum(r^2)
}
else {
  rss <- sum(w * r^2)
  r <- sqrt(w) * r
}
resvar <- rss/rdf
ans <- z[c("call", "terms", if (!is.null(z$weights)) "weights"),
class(ans) <- "summary.lm"
ans$aliases <- is.na(coef(object))
ans$residuals <- r
ans$df <- c(0L, n, length(ans$aliases))
ans$coefficients <- matrix(NA_real_, 0L, 4L, dimnames = list(
  c("Estimate", "Std. Error", "t value", "Pr(>|t|)")))
ans$sigma <- sqrt(resvar)
ans$r.squared <- ans$adj.r.squared <- 0
ans$cov.unscaled <- matrix(NA_real_, 0L, 0L)
if (correlation)
```

```
        ans$correlation <- ans$cov.unscaled
    return(ans)
}
if (is.null(z$terms))
    stop("invalid 'lm' object: no 'terms' component")
if (!inherits(object, "lm"))
    warning("calling summary.lm(<fake-lm-object>) ...")
Qr <- qr.lm(object)
n <- NROW(Qr$qr)
if (is.na(z$df.residual) || n - p != z$df.residual)
    warning("residual degrees of freedom in object suggest thi
r <- z$residuals
f <- z$fitted.values
w <- z$weights
if (is.null(w)) {
    mss <- if (attr(z$terms, "intercept"))
        sum((f - mean(f))^2)
    else sum(f^2)
    rss <- sum(r^2)
```



```

}
else {
  mss <- if (attr(z$terms, "intercept")) {
    m <- sum(w * f/sum(w))
    sum(w * (f - m)^2)
  }
  else sum(w * f^2)
  rss <- sum(w * r^2)
  r <- sqrt(w) * r
}
resvar <- rss/rdf
if (is.finite(resvar) && resvar < (mean(f)^2 + var(c(f))) *
  1e-30)
  warning("essentially perfect fit: summary may be unreliabl
p1 <- 1L:p
R <- chol2inv(Qr$qr[p1, p1, drop = FALSE])
se <- sqrt(diag(R) * resvar)
est <- z$coefficients[Qr$pivot[p1]]
tval <- est/se

```

```

ans <- z[c("call", "terms", if (!is.null(z$weights)) "weights"
ans$residuals <- r
ans$coefficients <- cbind(Estimate = est, `Std. Error` = se,
  `t value` = tval, `Pr(>|t|)` = 2 * pt(abs(tval), rdf,
    lower.tail = FALSE))
ans$aliased <- is.na(z$coefficients)
ans$sigma <- sqrt(resvar)
ans$df <- c(p, rdf, NCOL(Qr$qr))
if (p != attr(z$terms, "intercept")) {
  df.int <- if (attr(z$terms, "intercept"))
    1L
  else 0L
  ans$r.squared <- mss/(mss + rss)
  ans$adj.r.squared <- 1 - (1 - ans$r.squared) * ((n -
    df.int)/rdf)
  ans$fstatistic <- c(value = (mss/(p - df.int))/resvar,
    numdf = p - df.int, dendf = rdf)
}
else ans$r.squared <- ans$adj.r.squared <- 0

```

```
ans$cov.unscaled <- R
dimnames(ans$cov.unscaled) <- dimnames(ans$coefficients)[c(1,
  1)]
if (correlation) {
  ans$correlation <- (R * resvar)/outer(se, se)
  dimnames(ans$correlation) <- dimnames(ans$cov.unscaled)
  ans$symbolic.cor <- symbolic.cor
}
if (!is.null(z$na.action))
  ans$na.action <- z$na.action
class(ans) <- "summary.lm"
ans
}
<bytecode: 0x00000000355a2cc8>
<environment: namespace:stats>
```

Suppose you create a new kind of object, e.g. 'wald'

```
w <- wald(fit)
class(w)
```

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

Usually created by a 'constructor' function of the same name Note the last thing the function does:

```
wald
```

```
function (fit, Llist = "", clevel = 0.95, pred = NULL, data = NULL,
  debug = FALSE, maxrows = 25, full = FALSE, fixed = FALSE,
  invert = FALSE, method = "svd", df = NULL, pars = NULL, ...)
{
  if (full)
    return(wald(fit, getX(fit)))
  if (!is.null(pred))
    return(wald(fit, getX(fit, pred)))
  dataf <- function(x, ...) {
    x <- cbind(x)
    rn <- rownames(x)
    if (length(unique(rn)) < length(rn))
```

```
        rownames(x) <- NULL
    data.frame(x, ...)
}
as.dataf <- function(x, ...) {
  x <- cbind(x)
  rn <- rownames(x)
  if (length(unique(rn)) < length(rn))
    rownames(x) <- NULL
  as.data.frame(x, ...)
}
unique.rownames <- function(x) {
  ret <- c(tapply(1:length(x), x, function(xx) {
    if (length(xx) == 1) "" else 1:length(xx)
  }))[tapply(1:length(x), x)]
  ret <- paste(x, ret, sep = "")
  ret
}
if (inherits(fit, "stanfit")) {
  fix <- if (is.null(pars))
```

```
        getFix(fit)
    else getFix(fit, pars = pars, ...)
    if (!is.matrix(Llist))
        stop(paste("Sorry: wald needs Llist to be a n x",
                    length(fix$fixed), "matrix for this stanfit object
}
else {
    fix <- getFix(fit)
}
beta <- fix$fixed
vc <- fix$vcov
dfs <- if (is.null(df))
    fix$df
else df + 0 * fix$df
if (is.character(Llist))
    Llist <- structure(list(Llist), names = Llist)
if (!is.list(Llist))
    Llist <- list(Llist)
ret <- list()
```

```
for (ii in 1:length(Llist)) {
  ret[[ii]] <- list()
  Larg <- Llist[[ii]]
  L <- NULL
  if (is.character(Larg)) {
    L <- Lmat(fit, Larg, fixed = fixed, invert = invert)
  }
  else {
    if (is.numeric(Larg)) {
      if (is.null(dim(Larg))) {
        if (debug)
          disp(dim(Larg))
        if ((length(Larg) < length(beta)) && (all(Larg >
          0) || all(Larg < 0))) {
          L <- diag(length(beta))[Larg, ]
          dimnames(L) <- list(names(beta)[Larg], names(b
        )
      }
      else L <- rbind(Larg)
    }
  }
}
```

```
        else L <- Larg
    }
}
if (debug) {
    disp(Larg)
    disp(L)
}
Ldata <- attr(L, "data")
Lna <- L[, is.na(beta), drop = FALSE]
narows <- apply(Lna, 1, function(x) sum(abs(x))) > 0
L <- L[, !is.na(beta), drop = FALSE]
attr(L, "data") <- Ldata
beta <- beta[!is.na(beta)]
if (method == "qr") {
    qqr <- qr(t(na.omit(L)))
    L.rank <- qqr$rank
    if (debug)
        disp(t(qr.Q(qqr)))
    L.full <- t(qr.Q(qqr))[1:L.rank, , drop = FALSE]
```



```
}  
else if (method == "svd") {  
  if (debug)  
    disp(L)  
  sv <- svd(na.omit(L), nu = 0)  
  if (debug)  
    disp(sv)  
  tol.fac <- max(dim(L)) * max(sv$d)  
  if (debug)  
    disp(tol.fac)  
  if (tol.fac > 1e+06)  
    warning("Poorly conditioned L matrix, calculated n  
  tol <- tol.fac * .Machine$double.eps  
  if (debug)  
    disp(tol)  
  L.rank <- sum(sv$d > tol)  
  if (debug)  
    disp(L.rank)  
  if (debug)
```

```
        disp(t(sv$v))
    L.full <- t(sv$v)[seq_len(L.rank), , drop = FALSE]
}
else stop("method not implemented: choose 'svd' or 'qr'")
if (debug && method == "qr") {
    disp(qqr)
    disp(dim(L.full))
    disp(dim(vc))
    disp(vc)
}
if (debug)
    disp(L.full)
if (debug)
    disp(vc)
vv <- L.full %*% vc %*% t(L.full)
eta.hat <- L.full %*% beta
Fstat <- (t(eta.hat) %*% qr.solve(vv, eta.hat, tol = 1e-10
included.effects <- apply(L, 2, function(x) sum(abs(x),
    na.rm = TRUE)) != 0
```

```
denDF <- min(dfs[included.effects])
numDF <- L.rank
ret[[ii]]$anova <- list(numDF = numDF, denDF = denDF,
  `F-value` = Fstat, `p-value` = pf(Fstat, numDF, denDF,
    lower.tail = FALSE))
etahat <- L %*% beta
etahat[narows] <- NA
if (nrow(L) <= maxrows) {
  etavar <- L %*% vc %*% t(L)
  etasd <- sqrt(diag(etavar))
}
else {
  etavar <- NULL
  etasd <- sqrt(apply(L * (L %*% vc), 1, sum))
}
denDF <- apply(L, 1, function(x, dfs) min(dfs[x != 0]),
  dfs = dfs)
aod <- cbind(Estimate = c(etahat), Std.Error = etasd,
  DF = denDF, `t-value` = c(etahat/etasd), `p-value` = 2
```

```

        pt(abs(etahat/etasd), denDF, lower.tail = FALSE))
colnames(aod)[ncol(aod)] <- "p-value"
if (debug)
  disp(aod)
if (!is.null(clevel)) {
  hw <- qt(1 - (1 - clevel)/2, aod[, "DF"]) * aod[,
    "Std.Error"]
  aod <- cbind(aod, LL = aod[, "Estimate"] - hw, UL = aod[,
    "Estimate"] + hw)
  if (debug)
    disp(colnames(aod))
  labs <- paste(c("Lower", "Upper"), format(clevel))
  colnames(aod)[ncol(aod) + c(-1, 0)] <- labs
}
if (debug)
  disp(rownames(aod))
aod <- as.dataf(aod)
rownames(aod) <- rownames(as.dataf(L))
labs(aod) <- names(dimnames(L))[1]

```

```
    ret[[ii]]$estimate <- aod
    ret[[ii]]$coef <- c(etahat)
    ret[[ii]]$vcov <- etavar
    ret[[ii]]$L <- L
    ret[[ii]]$se <- etasd
    ret[[ii]]$L.full <- L.full
    ret[[ii]]$L.rank <- L.rank
    if (debug)
      disp(attr(Larg, "data"))
    data.attr <- attr(Larg, "data")
    if (is.null(data.attr) && !(is.null(data)))
      data.attr <- data
    ret[[ii]]$data <- data.attr
  }
  names(ret) <- names(Llist)
  attr(ret, "class") <- "wald"
  ret
}
```

<bytecode: 0x0000000032542d50>

```
<environment: namespace:spida2>
```

What can we do with a 'wald' object?

```
methods(class='wald')
```

```
[1] as.data.frame cell          coef          print  
[5] rpfmt  
see '?methods' for accessing help and source code
```

```
coef(w)
```

```
[1] 3.5054945 0.7032967
```

```
as.data.frame(w)
```

	coef	se	U2	L2
(Intercept)	3.5054945	6.0753033	15.656101	-8.6451121
day2	0.7032967	0.2093688	1.122034	0.2845591

```
w # prints
```

```
numDF denDF F.value p.value
```

```

      2      1 321.5723  0.0394
      Estimate Std.Error DF t-value p-value Lower 0.95
(Intercept) 3.505495 6.075303  1  0.577007 0.66683 -73.688553
day2        0.703297 0.209369  1  3.359129 0.18420  -1.956986
      Upper 0.95
(Intercept) 80.699542
day2        3.363579

```

```
rpfmt(w)
```

```

      Estimate
(Intercept) "3.505 (0.66683)"
day2        "0.703 (0.18420)"

```

```
cell(w) # ???
```

```

      coefficient coefficient
[1,] -117.848592  4.83502624
[2,] -117.607744  4.86752202
[3,] -116.888917  4.88358349

```

[4,]	-115.694948	4.88314729
[5,]	-114.030549	4.86621513
[6,]	-111.902290	4.83285383
[7,]	-109.318568	4.78319505
[8,]	-106.289582	4.71743478
[9,]	-102.827284	4.63583254
[10,]	-98.945340	4.53871037
[11,]	-94.659069	4.42645158
[12,]	-89.985387	4.29949920
[13,]	-84.942740	4.15835426
[14,]	-79.551027	4.00357377
[15,]	-73.831529	3.83576861
[16,]	-67.806817	3.65560100
[17,]	-61.500667	3.46378200
[18,]	-54.937968	3.26106863
[19,]	-48.144619	3.04826090
[20,]	-41.147431	2.82619867
[21,]	-33.974018	2.59575831
[22,]	-26.652691	2.35784927



[23,]	-19.212343	2.11341047
[24,]	-11.682338	1.86340659
[25,]	-4.092394	1.60882429
[26,]	3.527535	1.35066828
[27,]	11.147378	1.08995740
[28,]	18.737062	0.82772054
[29,]	26.266633	0.56499264
[30,]	33.706377	0.30281056
[31,]	41.026932	0.04220902
[32,]	48.199407	-0.21578351
[33,]	55.195495	-0.47014885
[34,]	61.987586	-0.71988314
[35,]	68.548876	-0.96400079
[36,]	74.853469	-1.20153838
[37,]	80.876484	-1.43155845
[38,]	86.594151	-1.65315322
[39,]	91.983905	-1.86544817
[40,]	97.024475	-2.06760545
[41,]	101.695968	-2.25882724

[42,]	105.979949	-2.43835889
[43,]	109.859510	-2.60549185
[44,]	113.319340	-2.75956654
[45,]	116.345785	-2.89997490
[46,]	118.926901	-3.02616278
[47,]	121.052501	-3.13763219
[48,]	122.714197	-3.23394321
[49,]	123.905431	-3.31471574
[50,]	124.621501	-3.37963101
[51,]	124.859581	-3.42843284
[52,]	124.618733	-3.46092861
[53,]	123.899906	-3.47699009
[54,]	122.705937	-3.47655388
[55,]	121.041538	-3.45962172
[56,]	118.913279	-3.42626042
[57,]	116.329557	-3.37660164
[58,]	113.300571	-3.31084137
[59,]	109.838273	-3.22923913
[60,]	105.956329	-3.13211697

[61,]	101.670058	-3.01985818
[62,]	96.996376	-2.89290580
[63,]	91.953729	-2.75176085
[64,]	86.562016	-2.59698037
[65,]	80.842518	-2.42917520
[66,]	74.817806	-2.24900760
[67,]	68.511656	-2.05718860
[68,]	61.948957	-1.85447522
[69,]	55.155608	-1.64166749
[70,]	48.158420	-1.41960526
[71,]	40.985007	-1.18916490
[72,]	33.663680	-0.95125586
[73,]	26.223332	-0.70681706
[74,]	18.693327	-0.45681318
[75,]	11.103383	-0.20223088
[76,]	3.483454	0.05592512
[77,]	-4.136389	0.31663601
[78,]	-11.726073	0.57887286
[79,]	-19.255644	0.84160077

[80,]	-26.695388	1.10378284
[81,]	-34.015943	1.36438439
[82,]	-41.188418	1.62237692
[83,]	-48.184506	1.87674226
[84,]	-54.976597	2.12647655
[85,]	-61.537887	2.37059420
[86,]	-67.842480	2.60813178
[87,]	-73.865495	2.83815186
[88,]	-79.583162	3.05974663
[89,]	-84.972916	3.27204157
[90,]	-90.013486	3.47419885
[91,]	-94.684979	3.66542065
[92,]	-98.968960	3.84495230
[93,]	-102.848521	4.01208526
[94,]	-106.308351	4.16615995
[95,]	-109.334796	4.30656830
[96,]	-111.915912	4.43275619
[97,]	-114.041512	4.54422560
[98,]	-115.703208	4.64053662

```
[99,] -116.894442  4.72130915
[100,] -117.610512  4.78622442
[101,] -117.848592  4.83502624
```

```
attr(,"parms")
```

```
attr(,"parms")$center
```

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

```
center 3.505495 0.7032967
```

```
attr(,"parms")$shape
```

```
      Coefficients
```

```
Coefficients (Intercept)          day2
```

```
(Intercept)  36.909310 -1.25661152
```

```
day2         -1.256612  0.04383529
```

```
attr(,"parms")$radius
```

```
[1] 19.97498
```

```
attr(,"class")
```

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

if you want to see the method:

```
spida2:::coef.wald # if you know where it is
```

```
function (obj, se = FALSE)
{
  if (length(obj) == 1) {
    ret <- ret <- obj[[1]]$coef
    if (is.logical(se) && (se == TRUE)) {
      ret <- cbind(coef = ret, se = obj[[1]]$se)
    }
    else if (se > 0) {
      ret <- cbind(coef = ret, coefp = ret + se * obj[[1]]$s
        coefm = ret - se * obj[[1]]$se)
      attr(ret, "factor") <- se
    }
  }
  else ret <- sapply(obj, coef.wald)
  ret
}
```

```
}  
<bytecode: 0x0000000031816098>  
<environment: namespace:spida2>
```

```
getAnywhere(coef.wald)
```

```
A single object matching 'coef.wald' was found  
It was found in the following places  
  registered S3 method for coef from namespace spida2  
  namespace:spida2  
with value
```

```
function (obj, se = FALSE)  
{  
  if (length(obj) == 1) {  
    ret <- ret <- obj[[1]]$coef  
    if (is.logical(se) && (se == TRUE)) {  
      ret <- cbind(coef = ret, se = obj[[1]]$se)  
    }  
  }  
}
```

```
    else if (se > 0) {
      ret <- cbind(coef = ret, coefp = ret + se * obj[[1]]$s
                  coefm = ret - se * obj[[1]]$se)
      attr(ret, "factor") <- se
    }
  }
  else ret <- sapply(obj, coef.wald)
  ret
}
```

```
<bytecode: 0x0000000031816098>
```

```
<environment: namespace:spida2>
```

```
getS3method('coef', 'wald')
```

```
function (obj, se = FALSE)
{
  if (length(obj) == 1) {
    ret <- ret <- obj[[1]]$coef
    if (is.logical(se) && (se == TRUE)) {
```



```
      ret <- cbind(coef = ret, se = obj[[1]]$se)
    }
    else if (se > 0) {
      ret <- cbind(coef = ret, coefp = ret + se * obj[[1]]$s
                  coefm = ret - se * obj[[1]]$se)
      attr(ret, "factor") <- se
    }
  }
  else ret <- sapply(obj, coef.wald)
  ret
}
<bytecode: 0x0000000031816098>
<environment: namespace:spida2>
```

## 8 Data wrangling —-

### 8.1 Regular Expressions to replace strings within string variables —-

Expertise with regular expressions is one of the most valuable skills you can learn for data manipulation.

Here's a site you can use to experiment with regular expressions. Add it to your R editing bookmarks.

Contribute questions, links and comments to Piazza.

Here's a useful summary prepared by Jeff Lee in the Winter 2016 class. Most descriptions of regular expressions make them look extremely complicated. You can get along with a few basic ideas that are very flexible and that have sufficed for 99.9% of my problems.

### 8.1.1 Basic Regular Expressions —

Using regular expressions is a way to alter, search, count, adjust texts or strings of characters.

There are 3 main groups of R functions that use regular expressions that we will look at.

First look at the function `grep`

```
x <- c("Hello", "He", "Hel", "hello", "hel")  
grep("hel", x)
```

```
[1] 4 5
```

As you can see, `grep` returns the index of all elements of `x` that contain “hel”. It does not return the index of “Hello” because `grep` is case sensitive.

We say the *pattern* “hel” *matches* substrings in the target.

To ignore the case, we can use:

```
grep("hel", x, ignore.case = T)
```

```
[1] 1 3 4 5
```

A similar effect is achieved by using square brackets: `[]`, which signify 'match any one character in the list'.

```
grep("[Hh]e1", x)
```

```
[1] 1 3 4 5
```

Suppose you want to know how many of these elements of `x` contain "hel" or "Hel"

```
length(grep("[hH]e1", x))
```

```
[1] 4
```

If you want to see the actual strings matched instead of their indices, use

```
grep("[Hh]e1", x, value = TRUE)
```

```
[1] "Hello" "Hel" "hello" "hel1"
```

or, with `spida2`:

```
library(spida2)
grepv("[Hh]el", x)
```

```
[1] "Hello" "Hel" "hello" "hel1"
```

Finally if you want a logical index vector:

```
grepl("[Hh]el", x)
```

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

### 8.1.2 Taking a closer look at gsub —

gsub and sub are great ways to modify substrings in a reproducible way. For example, you can use them to modify variable names in a way that will work when you receive an updated version of a data set. In most data sets, you will have variables names that are acronyms or short forms and you may want to

replace those variable names with something that people will understand.

The difference between `sub` and `gsub` is that `sub` will replace only the first match in each string, `gsub` (g stands for global) will replace all matches. Compare:

```
sub("1", "WWW", x)
```

```
[1] "HeWWW1o" "He"      "HeWWW"    "heWWW1o" "heWWW1"
```

```
gsub("1", "WWW", x)
```

```
[1] "HeWWWWWo" "He"      "HeWWW"    "heWWWWWo" "heWWW1"
```

The most difficult part about regular expressions is the syntax. These are helpful websites with information on syntax.

- [Quick-Start: Regex Cheat Sheet](#)
- [Regular Expressions in R by Albert Y. Kim](#)
- [RegExr](#) to interactively try out regular expressions

There's a thorough treatment at

[Microsoft's Regular Expression Language – Quick Reference](#)

Also you can get help in R:

```
?regex
```

```
?gsub
```

There are many special characters that let you do almost anything you want with regular expressions. Here are the most important ones:

- Special characters: All characters match themselves except the special characters. `$ ^ { [ ( | ) * + ? \ .` Also `} ]` are special characters when they close a matching brace and `-` is a special character when it appears within square brackets.
- Special matching characters:
  - `.`: a period matches any single character
  - `[abc]`: matches any single character in the list
  - `[A-Z]`: matches a single character in the range A to Z. If you want to include a hyphen as matching character, it must come first, e.g. `[-a-z]`.
  - `[A-Za-z0-9]`: matches any single alphanumeric character
  - `[^a-z]`: matches any single character that is NOT a lower case letter.

The caret  $\wedge$  at the beginning of the bracketed list negates the rest of the list. A caret anywhere else is just a caret.

- ( and ) can be used to form sub groups. ( are not ) matched. To match a parenthesis you need to 'escape' it: " $\backslash(a\backslash)$ " in a string in R.
- | means "or":  $(a|b)c$  is the same as  $[ab]c$

- Anchors:

- $\wedge$  matches the beginning and  $\$$  matches the end of a string. Thus " $\wedge$ and" matches only strings that start with "and", while "and $\$$ " matches only strings that end with "and". To only get exact matches, i.e. strings that are exactly equal to "and", use both " $\wedge$ and $\$$ ", e.g. " $\wedge$ match this exactly $\$$ ".

- Quantifiers: how many repeats of the previous match:

- \* matches the previous match 0 or more times
  - + matches the previous match one or more times
  - ? matches the previous match zero or one time
  - {n} matches the previous match exactly n times
  - {n,m} matches the previous match n to m times
  - {n,} matches the previous match at least n times
  - {,m} matches the previous match at most m times
- Quantifiers are



'greedy' in the sense that they will match as much of the string as they can. Adding ? to a quantifier makes it 'lazy'. It will match as few occurrences as possible.

### 8.1.3 Common Regular Expressions —

".": the '.' means any single character and "" means zero or more of the previous match. So this is the 'universal' match. It matches anything?

```
some_names <- c('Mary Jones', 'Bush, George H. W.', 'George W. Bush')
sub(".*", "OhOh", some_names)
```

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

"^ John (.\*?) Brown \$"

A powerful tool for substitution is the 'backreference' \\N where N is a single digit from 1 to 9. In a replacement string \\N refers to the Nth parenthesized expression in the pattern. For example:

```
x <- c('Wong, Rodney', 'Smith, John', 'Robert Jones')
sub("^( [^, ]+ ), +([ ]*)$", "\\2 \\1", x)
```

```
[1] "Rodney Wong" "John Smith" "Robert Jones"
```

Parsing Using parentheses to match substrings to change their order in the replacement string.

```
sub(  
  "^([^,]*), ?(.*)$",  
  "\\2 \\1",  
  some_names)
```

```
[1] "Mary Jones"          "George H. W. Bush" "George W. Bush"  
[4] "Truman Capote"
```

#### 8.1.4 Quiz question —

- What is the purpose of ' ? ' in the regular expression above?
- What would happen if we used ' \* ' instead?

Important application: Changing the form of variable names in preparation for restructuring from wide to long format

prog

"SC/MATH HON 4 ACT ..."

```
var_names <- c('id', 'Gender', 'Age', 'T1_data', 't2_date', 'T3_date',  
var_names
```

```
[1] "id"          "Gender"      "Age"         "T1_data"     "t2_date"  
[6] "T3_date"     "T1_pulse"   "T2_pulse"   "T3_pulse"
```

fix 'data':

```
modified_names <- sub('data$', 'date', var_names)  
modified_names
```

```
[1] "id"          "Gender"      "Age"         "T1_date"     "t2_date"  
[6] "T3_date"     "T1_pulse"   "T2_pulse"   "T3_pulse"
```

reorder variable name and time:

```
modified_names <- sub("^(?=[tT])([0-9])_(.*)$", "\\3_\\2", modified_names)  
modified_names # note that names that don't match the pattern are l
```

```
[1] "id"          "Gender"      "Age"         "date__1"     "date__2"  
[6] "date__3"     "pulse__1"   "pulse__2"   "pulse__3"
```

In order to match a special character it needs to be escaped with a backslash ‘\’ before the character.

```
s <- ("HEL$LO")
```

```
s
```

```
[1] "HEL$LO"
```

```
gsub("$", replacement = ".", s) # $ matches the end of the string
```

```
[1] "HEL$LO."
```

```
gsub("\\$", replacement = ".", s) # \\$ matches the actual $
```

```
[1] "HEL.LO"
```

As you can see, using two back slashes will actually replace \$ with a period. In a string in R you need to use two backslashes to produce one backslash, i.e. you need to escape the escape.

```
y <- c("hello123", "hello213", "he12222lo", "llo he123" )
```

```
gsub(".*2", "--", y)
```

```
[1] "--3" "--13" "--1o" "--3"
```

```
gsub(".*2", "", y) # Note that "" will delete the match
```

```
[1] "3" "13" "1o" "3"
```

This will remove everything up to and including a 2 in each string. As you can see in `hel2222lo`, it removes the last 2.

```
gsub("^hel2", "4939", y)
```

```
[1] "hello123" "hello213" "49392221o" "1lo he123"
```

The `^` will replace everything that starts with `hel2`. In this case only the 3rd word started with `hel2` so it replaces it with `4939`.

```
gsub("213$", "4939", y)
```

```
[1] "hello123" "hello4939" "hel22221o" "1lo he123"
```

The `$` will replace everything that ends with `213`. In this case only the 2nd word ended with `213` so it replaces it with `4939`.

```
gsub("\\bhe", "4939", y)
```

```
[1] "493911o123" "493911o213" "4939122221o" "11o 4939123"
```

The double backslash b will replace everything that starts with 'he' on words instead of strings. In this case, every word had a 'he' in this case.

```
gsub("hel*1", "4939", y)
```

```
[1] "hello123" "hello213" "hel22221o" "11o 493923"
```

The \* will replace anything that matches at least 0 times. In this case, the last word matches hel and 1 matches 0 times.

The special character | allows alternative choices. It matches either what comes before the | or what comes after it.

```
gsub("hel|213", "4939", y)
```

```
[1] "49391o123" "49391o4939" "493922221o" "11o he123"
```

The | is an 'or' feature. This pattern will replace anything with a hel or 213. If

it can match either `hel` and `213` it will replace both.

Note that you can use and mix quantifiers and operators together. Perhaps the most common combination is `.*` which matches anything

### 8.1.5 Taking a look at `regexr` —

```
y <- c("hello123", "hello213", "hel2222lo", "llo he123", "zork")
regexr("he(.*)", y)
```

```
[1] 1 1 1 5 -1
attr("match.length")
[1] 8 8 9 5 -1
attr("index.type")
[1] "chars"
attr("useBytes")
[1] TRUE
```

`regexr` returns the position of the first character matched.



`attr("match.length")` is the number of characters matched in each string, -1 if no match.

```
regexpr("hel(.*)", y)
```

```
[1] 1 1 1 -1 -1
attr(,"match.length")
[1] 8 8 9 -1 -1
attr(,"index.type")
[1] "chars"
attr(,"useBytes")
[1] TRUE
```

As you can see, the 4th word does not have hel in it.

```
gregexpr("he(.*)", y)
```

```
[[1]]
[1] 1
attr(,"match.length")
[1] 8
```

```
attr(,"index.type")
```

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

```
attr(,"useBytes")
```

```
[1] TRUE
```

```
[[2]]
```

```
[1] 1
```

```
attr(,"match.length")
```

```
[1] 8
```

```
attr(,"index.type")
```

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

```
attr(,"useBytes")
```

```
[1] TRUE
```

```
[[3]]
```

```
[1] 1
```

```
attr(,"match.length")
```

```
[1] 9
```

```
attr(,"index.type")
```

```
[1] "chars"  
attr(,"useBytes")  
[1] TRUE
```

```
[[4]]  
[1] 5  
attr(,"match.length")  
[1] 5  
attr(,"index.type")  
[1] "chars"  
attr(,"useBytes")  
[1] TRUE
```

```
[[5]]  
[1] -1  
attr(,"match.length")  
[1] -1  
attr(,"index.type")  
[1] "chars"
```

```
attr(,"useBytes")  
[1] TRUE
```

`gregexpr` will return a list of all the matches.

## 9 Reshaping Data —-

I have to reshape data almost every time I see a client. In fact some clients come to see me just to have their data reshaped. I need to keep it fast and simple.

Most serious data errors I encounter come from mishaps in attempting to reshape data by hand, for example, by cutting and pasting portions of worksheets in Excel.

I encounter two major reasons for reshaping data:

1. Longitudinal data and hierarchical data (where each subject may be seen and measured more than once) needs to be in different shapes (long or wide) for different methods of analysis. Traditional multivariate methods expect wide data and newer mixed model approaches require long data.

2. Categorical data needs to be in different forms; long (one row per observation), aggregated, or tabular for different analyses (logistic regression, binomial regression or log-linear modeling).

The shape in which you get the data must not determine your method of analysis. You need to be able to go back and forth easily among data shapes to use the analyses you wish to apply.

*A longitudinal example:* This is a simple example from a pretend medical study in which each subject is seen on a varying number of visits. This is the data set in **long** form.

## 9.1 Long form —

```
dlong <- read.table(strip.white = T, header = TRUE, text =  
"  
sid  name  visit date      sex      sysbp temp  
1    Sam   1    2019-01-21  male    124    36.5  
1    Sam   2    2019-03-15  male    129    36.8
```

```
2   Joan   1   2019-02-10   female   115   37.1
3   Kate   1   2018-06-16   female   132   37.3
3   Kate   2   2018-09-03   female   139   36.7
3   Kate   3   2019-04-20   female   138   36.9
")
```

dlong

```
sid name visit date sex sysbp temp
1 1 Sam 1 2019-01-21 male 124 36.5
2 1 Sam 2 2019-03-15 male 129 36.8
3 2 Joan 1 2019-02-10 female 115 37.1
4 3 Kate 1 2018-06-16 female 132 37.3
5 3 Kate 2 2018-09-03 female 139 36.7
6 3 Kate 3 2019-04-20 female 138 36.9
```

We can identify four types of variables:

1. a **subject id** variable that uniquely identifies each subject. Names are not usually adequate for this purpose since two subjects could share the same

- name. A good example in a university setting is the student number.
2. a **time-index** variable consisting of small integers that, for each subject, identifies the *visit* or *occasion*.
  3. **Value** variables that are measurements or characteristics of subjects or of visits. They fall into two classes:
    - a. **Time-varying** (or **visit-level**) variables that can vary from visit to visit. In this example, these are: *date*, *sysbp* and *temp*.
    - b. **Time-invariant** (or **\_subject-level\_**) variables that remain the same within each subject from visit to visit. In this example these are: *name* and *sex*. Sometimes, a variable may appear to be time-invariant in the observed data but would be time-varying if one had observed more data.

Note:

1. The **subject id** by **time index** combinations should be unique although it is possible to have deeper indexing. For example, if each visit has two phases: *am* and *pm*, then there could be a deeper indexing variable, *phase* with values *am* and *pm*. Then the combinations of the **subject id** by **time index** by **phase index** would need to be unique.

2. It is not necessary to have all possible combinations in the data.
3. The groups of rows belonging to the same subject are often called **clusters**.

## 9.2 Wide form —

Here's the same data in **wide** form with one row per subject. Sorry the input is too wide for the screen.



```

dwide <- read.table(strip.white = T, header = TRUE, text =
"
sid name sex date_1 date_2 date_3 sysbp_1 sysbp_2 sys
1 Sam male 2019-01-21 2019-03-15 NA 124 129 NA
2 Joan female 2019-02-10 NA NA 113 NA NA
3 Kate female 2018-06-16 2018-09-03 2018-04-20 132 NA 138
")
dwide

```

	sid	name	sex	date_1	date_2	date_3	sysbp_1
1	1	Sam	male	2019-01-21	2019-03-15	<NA>	124
2	2	Joan	female	2019-02-10	<NA>	<NA>	113
3	3	Kate	female	2018-06-16	2018-09-03	2018-04-20	132

	sysbp_2	sysbp_3	temp_1	temp_2	temp_3
1	129	NA	36.5	36.8	NA
2	NA	NA	37.1	NA	NA
3	NA	138	37.3	36.7	36.9

### 9.3 Relational data base form

SQL

In an RDB, this data would be represented by two *relations* (data frames) which can be merged as needed for analyses.

One relation contains time invariant variables and the second contain time-varying variables plus the subject id variable (called a **key**) needed to link the time-varying variables with the time-invariant variables.

Instead of re-entering from scratch, we'll start using the tools in 'spida2'

```
library(spida2)
```

The time-invariant variable relation contains the following.

```
dti <- up(dlong, ~sid)
```

```
dti
```

	id	name	sex
1	1	Sara	male
2	2	Joan	female
3	3	Kate	female

The time-varying relation is:

```
dtv <- subset(dlong, select = !(names(dlong) %in% names(dti)[-1]))
dtv
```

	sid	visit	date	sysbp	temp
1	1	1	2019-01-21	124	36.5
2	1	2	2019-03-15	129	36.8
3	2	1	2019-02-10	115	37.1
4	3	1	2018-06-16	132	37.3
5	3	2	2018-09-03	139	36.7
6	3	3	2019-04-20	138	36.9

Note that the 'select' argument of the 'subset' function selects variables.

You can get the long file back with:

```
merge(dti, dtv, all = T)
```

	sid	name	sex	visit	date	sysbp	temp
1	1	Sam	male	1	2019-01-21	124	36.5

2	1	Sam	male	2	2019-03-15	129	36.8
3	2	Joan	female	1	2019-02-10	115	37.1
4	3	Kate	female	1	2018-06-16	132	37.3
5	3	Kate	female	2	2018-09-03	139	36.7
6	3	Kate	female	3	2019-04-20	138	36.9

I encourage researchers who use Excel for data entry to keep their data in multiple spreadsheets, one for each data level as in a relational data base. This reduces errors in data entry and updating. The principle is that **if you need to correct the value of a variable you should only have to do it in one place.**

Keeping separate spreadsheets for different data levels makes this possible. For example, if you need to correct the spelling of a name, you only need to make the correction in one place. Currently, I find that the best way to read Excel spreadsheets is with the 'read\_excel' function in the 'readxl' package.

## 9.4 From Wide to Long —

The `tolong` function in the ‘spida2’ package relies on the form of the variable names to transform the wide data frame to a long form. The function looks for a *separator* between the name of the value variable and the *time index*. The default is ‘\_’ which can be changed with the ‘sep’ argument. The default name created for the *time index* variable is ‘time’.

dwide

	sid	name	sex	date_1	date_2	date_3	sysbp_1
1	1	Sam	male	2019-01-21	2019-03-15	<NA>	124
2	2	Joan	female	2019-02-10	<NA>	<NA>	113
3	3	Kate	female	2018-06-16	2018-09-03	2018-04-20	132
	sysbp_2	sysbp_3	temp_1	temp_2	temp_3		
1	129	NA	36.5	36.8	NA		
2	NA	NA	37.1	NA	NA		
3	NA	138	37.3	36.7	36.9		

```
tolong(dwide)
```

	name	sex	sid	time	date	sysbp	temp	id
1.1	Sam	male	1	1	2019-01-21	124	36.5	1
2.1	Joan	female	2	1	2019-02-10	113	37.1	2
3.1	Kate	female	3	1	2018-06-16	132	37.3	3
1.2	Sam	male	1	2	2019-03-15	129	36.8	1
2.2	Joan	female	2	2	<NA>	NA	NA	2
3.2	Kate	female	3	2	2018-09-03	NA	36.7	3
1.3	Sam	male	1	3	<NA>	NA	NA	1
2.3	Joan	female	2	3	<NA>	NA	NA	2
3.3	Kate	female	3	3	2018-04-20	138	36.9	3

It's best to specify a name for the *time index*. Otherwise, if a variable named 'time' already exists it will get clobbered by 'tolong'.

Also, the new 'id' variable generated by 'tolong' refers to the row numbers of the input data frame. If a variable named 'id' already exists and has unique values, 'tolong' will use that variable. You can specify a variable name as the id variable

```
dtolong <- tolong(dwide, timevar = 'visit', idvar = 'sid')
dtolong
```

	name	sex	sid	visit	date	sysbp	temp
1.1	Sam	male	1	1	2019-01-21	124	36.5
2.1	Joan	female	2	1	2019-02-10	113	37.1
3.1	Kate	female	3	1	2018-06-16	132	37.3
1.2	Sam	male	1	2	2019-03-15	129	36.8
2.2	Joan	female	2	2	<NA>	NA	NA
3.2	Kate	female	3	2	2018-09-03	NA	36.7
1.3	Sam	male	1	3	<NA>	NA	NA
2.3	Joan	female	2	3	<NA>	NA	NA
3.3	Kate	female	3	3	2018-04-20	138	36.9

It's often useful to reorder longitudinal data, e.g. for plotting:

```
sortdf(dtolong, ~ sid/visit)
```

	name	sex	sid	visit	date	sysbp	temp
1.1	Sam	male	1	1	2019-01-21	124	36.5

1.2	Sam	male	1	2	2019-03-15	129	36.8
1.3	Sam	male	1	3	<NA>	NA	NA
2.1	Joan	female	2	1	2019-02-10	113	37.1
2.2	Joan	female	2	2	<NA>	NA	NA
2.3	Joan	female	2	3	<NA>	NA	NA
3.1	Kate	female	3	1	2018-06-16	132	37.3
3.2	Kate	female	3	2	2018-09-03	NA	36.7
3.3	Kate	female	3	3	2018-04-20	138	36.9

When the variables are not conveniently named we can often use regular expressions to transform the names into a form that works with ‘`tolong`’. See the additional material on regular expressions in the extra notes.

## 9.5 From Long to Wide —

This is a bit trickier because there are no clues from the form of the variable names that some are subscripted. We need to specify the **id** variable and the **time index** variable.

Standard reshape functions also expect you to indicate which variables are



time-varying so that only those variables get indexed in the wide form. With a large dataset this can be an enormous amount of work, which the 'towide' function gets the computer to do for you. The function identifies which variables are time-varying and which are not and only the time-varying variables get expanded by indexing.

```
towide(dlong, idvar = 'sid', timevar = 'visit')
```

	sid	date_1	sysbp_1	temp_1	date_2	sysbp_2	temp_2
1	1	2019-01-21	124	36.5	2019-03-15	129	36.8
2	2	2019-02-10	115	37.1	<NA>	NA	NA
3	3	2018-06-16	132	37.3	2018-09-03	139	36.7
		date_3	sysbp_3	temp_3	name	sex	
1		<NA>	NA	NA	Sam	male	
2		<NA>	NA	NA	Joan	female	
3		2019-04-20	138	36.9	Kate	female	

## 9.6 More examples —

Many sources of global data let you retrieve data from various countries by variable. After concatenating the raw data for the various variables, you get something that looks like this:

```
dd <- read.table(header=T,text="
country    variable    1990 1991 1992 1993
Canada     population  20   21   24   26
Mexico     population  50   52   53   54
Canada     income      10   12   12   11
Mexico     income      30   31   33   34
")
dd
```

	country	variable	X1990	X1991	X1992	X1993
1	Canada	population	20	21	24	26
2	Mexico	population	50	52	53	54
3	Canada	income	10	12	12	11
4	Mexico	income	30	31	33	34

Note how ‘read.table’ prepended an ‘X’ to the years since a valid variable names can’t start with a number.

We need to get the variable names in the right form for ‘tolong’. The ‘easy’ way is to use regular expressions.

```
names(dd) <- sub('^X', 'value__', names(dd))
dd
```

	country	variable	value__1990	value__1991	value__1992
1	Canada	population	20	21	24
2	Mexico	population	50	52	53
3	Canada	income	10	12	12
4	Mexico	income	30	31	33
		value__1993			
1		26			
2		54			
3		11			
4		34			

The regular expression ‘^X’ matches a capital X at the beginning of a string.

Wherever it is found, it gets replaced by 'year\_\_'. I'm in the habit of using a repeated underscore, '\_\_', as a separator to avoid conflicts with other underscores in variable names.

Now we're ready for the first step:

```
d1 <- tolong(dd, sep = '__', timevar = 'year')
d1
```

	country	variable	year	value	id
1.1990	Canada	population	1990	20	1
2.1990	Mexico	population	1990	50	2
3.1990	Canada	income	1990	10	3
4.1990	Mexico	income	1990	30	4
1.1991	Canada	population	1991	21	1
2.1991	Mexico	population	1991	52	2
3.1991	Canada	income	1991	12	3
4.1991	Mexico	income	1991	31	4
1.1992	Canada	population	1992	24	1
2.1992	Mexico	population	1992	53	2

3.1992	Canada	income	1992	12	3
4.1992	Mexico	income	1992	33	4
1.1993	Canada	population	1993	26	1
2.1993	Mexico	population	1993	54	2
3.1993	Canada	income	1993	11	3
4.1993	Mexico	income	1993	34	4

Now, our ‘id’ or **key** uses the combination of two variables: *country* and *year* because we want one row for each of those combinations.

Also, our ‘timevar’ is ‘variable’:

```
d2 <- towide(d1, idvar = c('country','year'), timevar = 'variable')
d2
```

	country	year	value_population	id_population	value_income
1	Canada	1990	20	1	10
2	Canada	1991	21	1	12
3	Canada	1992	24	1	12
4	Canada	1993	26	1	11
5	Mexico	1990	50	2	30

6	Mexico	1991	52	2	31
7	Mexico	1992	53	2	33
8	Mexico	1993	54	2	34

id\_income

1	3
2	3
3	3
4	3
5	4
6	4
7	4
8	4

We don't need the 'id\_..' variables and we rename the value variables:

```
d2 <- d2[, -grep('^id_', names(d2))]
```

```
d2
```

	country	year	value_population	value_income
1	Canada	1990	20	10

2	Canada	1991	21	12
3	Canada	1992	24	12
4	Canada	1993	26	11
5	Mexico	1990	50	30
6	Mexico	1991	52	31
7	Mexico	1992	53	33
8	Mexico	1993	54	34

```
names(d2) <- sub('^value_', '', names(d2))  
d2
```

	country	year	population	income
1	Canada	1990	20	10
2	Canada	1991	21	12
3	Canada	1992	24	12
4	Canada	1993	26	11
5	Mexico	1990	50	30
6	Mexico	1991	52	31
7	Mexico	1992	53	33
8	Mexico	1993	54	34

... and you are ready to do some analyses.

## 9.7 Variables and years in long form —

Another common format for global health has both variables and time in long form.

```
dd <- read.table(header=TRUE, text = "  
country    year    variable    value    country.code    rownum  
Canada     2001    atemp       20      CAN             1  
Canada     2002    atemp       23      CAN             2  
US         2001    atemp       23      USA             3  
US         2002    atemp       23      USA             4  
Canada     2001    wind        120     CAN             5  
Canada     2002    wind        123     CAN             6  
US         2001    wind        123     USA             7  
US         2002    wind        123     USA             8  
Canada     2001    rain        220     CAN             9  
Canada     2002    rain        223     CAN            10
```



```
US      2001    rain      223    USA      11
US      2002    rain      223    USA      12
```

```
)
```

```
(dw <- towide(
  dd,
  idvar = c('country','year'),
  timevar = 'variable'))
```

```
country year value_atemp rownum_atemp value_wind rownum_wind
1 Canada 2001          20           1         120           5
2 Canada 2002          23           2         123           6
3      US 2001          23           3         123           7
4      US 2002          23           4         123           8
value_rain rownum_rain country.code
1         220           9         CAN
2         223          10         CAN
3         223          11         USA
4         223          12         USA
```

```
#  
# to keep only the variable name as a name  
#  
names(dw) <- sub('^value_', '', names(dw))  
dw
```

	country	year	atemp	rownum_atemp	wind	rownum_wind	rain
1	Canada	2001	20	1	120	5	220
2	Canada	2002	23	2	123	6	223
3	US	2001	23	3	123	7	223
4	US	2002	23	4	123	8	223

	rownum_rain	country.code
1	9	CAN
2	10	CAN
3	11	USA
4	12	USA

```
#  
# to get rid of other time varying variable
```

```
#  
dw <- dw[, - grep('_', names(dw))]  
dw
```

	country	year	atemp	wind	rain	country.code
1	Canada	2001	20	120	220	CAN
2	Canada	2002	23	123	223	CAN
3	US	2001	23	123	223	USA
4	US	2002	23	123	223	USA

## 9.8 Working with long data frames —

One advantage of working with long (as opposed to wide) data is the ease with which you can do calculations using the **clusters** much more easily if you have the right tool.

Using the original long data frame:

```
dlong
```

	sid	name	visit	date	sex	sysbp	temp
1	1	Sam	1	2019-01-21	male	124	36.5
2	1	Sam	2	2019-03-15	male	129	36.8
3	2	Joan	1	2019-02-10	female	115	37.1
4	3	Kate	1	2018-06-16	female	132	37.3
5	3	Kate	2	2018-09-03	female	139	36.7
6	3	Kate	3	2019-04-20	female	138	36.9

we would like to have the variables in different columns and the years in different rows.

We create a long data frame with respect to year and then a wide one with respect to variable, suppose we want to create new variables for the mean 'sysbp' and 'temp' for each subject.

The **capply** function does this. It applies a function to the values of a variable in each cluster and returns a result that has the right form to be added as a variable to the data frame.

```
dlong2 <- within(  
  dlong, {
```

```
temp_m <- capply(temp, sid, mean)
}
)
dlong2
```

	sid	name	visit	date	sex	sysbp	temp	temp_m
1	1	Sam	1	2019-01-21	male	124	36.5	36.65000
2	1	Sam	2	2019-03-15	male	129	36.8	36.65000
3	2	Joan	1	2019-02-10	female	115	37.1	37.10000
4	3	Kate	1	2018-06-16	female	132	37.3	36.96667
5	3	Kate	2	2018-09-03	female	139	36.7	36.96667
6	3	Kate	3	2019-04-20	female	138	36.9	36.96667

**capply** applies the function *mean* to each *cluster* of values of *temp* defined by a common value of *sid* and returns a result that has the right shape to be added to the data frame. Note that, in contrast with *SAS*, the order of rows in the data frame doesn't matter. That is, clusters don't have to be in contiguous rows. Also, in contrast with *tapply*, the function does not have to return a single value.

```

dlong2 <- within(
  dlong2,
  {
    sysbp_m <- capply(sysbp, sid, mean)
    sysbp_rank <- capply(sysbp, sid, rank)
    temp_rank <- capply(temp, sid, rank)
    temp_sd <- capply(temp, sid, sd)
  }
)
dlong2

```

	sid	name	visit	date	sex	sysbp	temp	temp_m	temp_sd
1	1	Sam	1	2019-01-21	male	124	36.5	36.65000	0.212132
2	1	Sam	2	2019-03-15	male	129	36.8	36.65000	0.212132
3	2	Joan	1	2019-02-10	female	115	37.1	37.10000	NA
4	3	Kate	1	2018-06-16	female	132	37.3	36.96667	0.305505
5	3	Kate	2	2018-09-03	female	139	36.7	36.96667	0.305505
6	3	Kate	3	2019-04-20	female	138	36.9	36.96667	0.305505

	temp_rank	sysbp_rank	sysbp_m
1	1	1	126.5000
2	2	2	126.5000
3	1	1	115.0000
4	3	1	136.3333
5	1	3	136.3333
6	2	2	136.3333

The variable 'temp\_sd' is a measure of the variability in their temperature. This way be a variable of interest. Once it has been computed in the long file, it is available for analysis in models at the subject level, with:

```
up(dlong2, ~sid)
```

	sid	name	sex	temp_m	temp_sd	sysbp_m
1	1	Sam	male	36.65000	0.212132	126.5000
2	2	Joan	female	37.10000	NA	115.0000
3	3	Kate	female	36.96667	0.305505	136.3333

The long file often provides a much easier way to create new subject-level

variables than working with the original data in wide form.

## 9.9 Reshaping categorical data —

Purely categorical data (in which all variables are treated as categorical) can be represented in many ways.

1. Frequency table well suited for log-linear analysis
2. Subject-level long data frame with one observation per subject for logistic regression
3. Aggregated data frame with a frequency variable for Poisson models
4. Data frame with frequencies wide on one variable for binomial or multinomial analyses

We use the ‘Titanic’ table in base R. It’s an array with class ‘table’ so functions that have methods for the class ‘table’ will use those methods. The table cells contain the frequencies for each outcome.



## 9.9.1 Tabular data —

Titanic

```
, , Age = Child, Survived = No
```

Class	Sex	
	Male	Female
1st	0	0
2nd	0	0
3rd	35	17
Crew	0	0

```
, , Age = Adult, Survived = No
```

Class	Sex	
	Male	Female
1st	118	4
2nd	154	13

3rd	387	89
Crew	670	3

, , Age = Child, Survived = Yes

Sex		
Class	Male	Female
1st	5	1
2nd	11	13
3rd	13	14
Crew	0	0

, , Age = Adult, Survived = Yes

Sex		
Class	Male	Female
1st	57	140
2nd	14	80
3rd	75	76

Crew 192 20

A different view: a flattened table:

```
ftable(Titanic)
```

			Survived	
			No	Yes
Class	Sex	Age		
1st	Male	Child	0	5
		Adult	118	57
	Female	Child	0	1
		Adult	4	140
2nd	Male	Child	0	11
		Adult	154	14
	Female	Child	0	13
		Adult	13	80
3rd	Male	Child	35	13
		Adult	387	75
	Female	Child	17	14
		Adult	89	76

Crew	Male	Child	0	0
		Adult	670	192
	Female	Child	0	0
		Adult	3	20

```
dimnames(Titanic)
```

```
$Class
```

```
[1] "1st" "2nd" "3rd" "Crew"
```

```
$Sex
```

```
[1] "Male" "Female"
```

```
$Age
```

```
[1] "Child" "Adult"
```

```
$Survived
```

```
[1] "No" "Yes"
```

Permuting the dimensions of the array:

```
fctable(aperm(Titanic, c('Class', 'Sex', 'Survived', 'Age')))
```

		Age		
Class	Sex	Survived	Child	Adult
1st	Male	No	0	118
		Yes	5	57
	Female	No	0	4
		Yes	1	140
2nd	Male	No	0	154
		Yes	11	14
	Female	No	0	13
		Yes	13	80
3rd	Male	No	35	387
		Yes	13	75
	Female	No	17	89
		Yes	14	76
Crew	Male	No	0	670
		Yes	0	192

Female	No	0	3
	Yes	0	20

```
dim(Titanic) # 4-dimensional array
```

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

The 'tab' function in 'spida2' operates on tables to show marginal distributions.

```
tab(Titanic, ~ Sex)
```

Sex

Male	Female	Total
1731	470	2201

```
tab(Titanic, ~ Sex + Age) # frequencies
```

Sex	Age		Total
	Child	Adult	
Male	64	1667	1731
Female	45	425	470

```
Total    109  2092  2201
```

```
tab(Titanic, ~ Sex + Age, pct = 1) # row percentages
```

```
      Age
Sex    Child    Adult    Total
Male   3.697285  96.302715 100.000000
Female 9.574468  90.425532 100.000000
All    4.952294  95.047706 100.000000
```

```
tab(Titanic, ~ Sex + Age, pct = 2) # column percentages
```

```
      Age
Sex    Child    Adult    All
Male   58.71560  79.68451  78.64607
Female 41.28440  20.31549  21.35393
Total 100.00000 100.00000 100.00000
```

To suppress margins, use the variants 'tab\_\_' and 'tab\_\_\_'

```
tab_(Titanic, ~ Sex)
```

```
Sex
  Male Female
1731   470
```

```
tab_(Titanic, ~ Sex + Age) # frequencies
```

```
      Age
Sex    Child Adult
Male    64  1667
Female  45   425
```

```
tab_(Titanic, ~ Sex + Age, pct = 1) # row percentages
```

```
      Age
Sex    Child  Adult
Male  3.697285 96.302715
Female 9.574468 90.425532
All    4.952294 95.047706
```

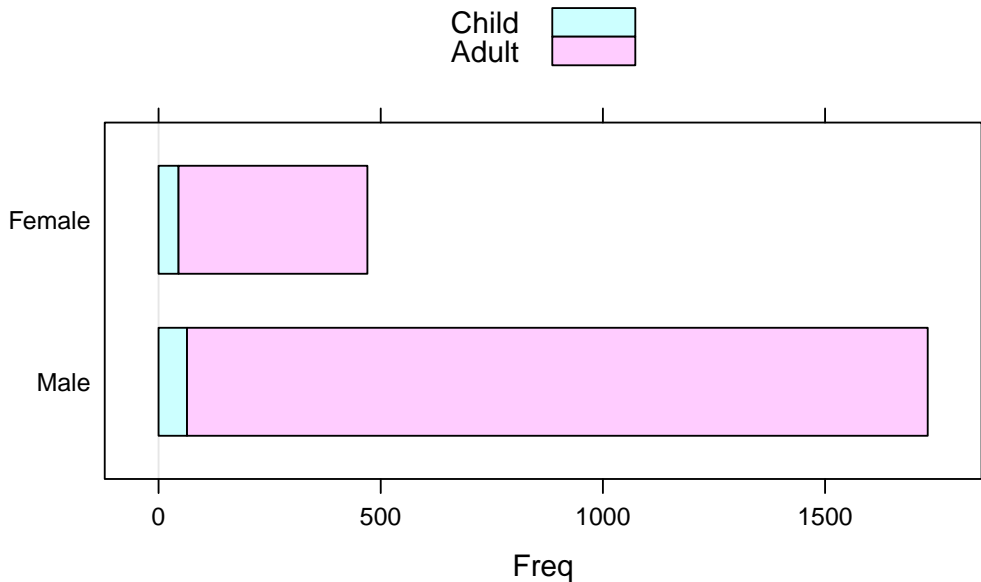


```
tab__(Titanic, ~ Sex + Age, pct = 1) # row percentages
```

	Age	
Sex	Child	Adult
Male	3.697285	96.302715
Female	9.574468	90.425532

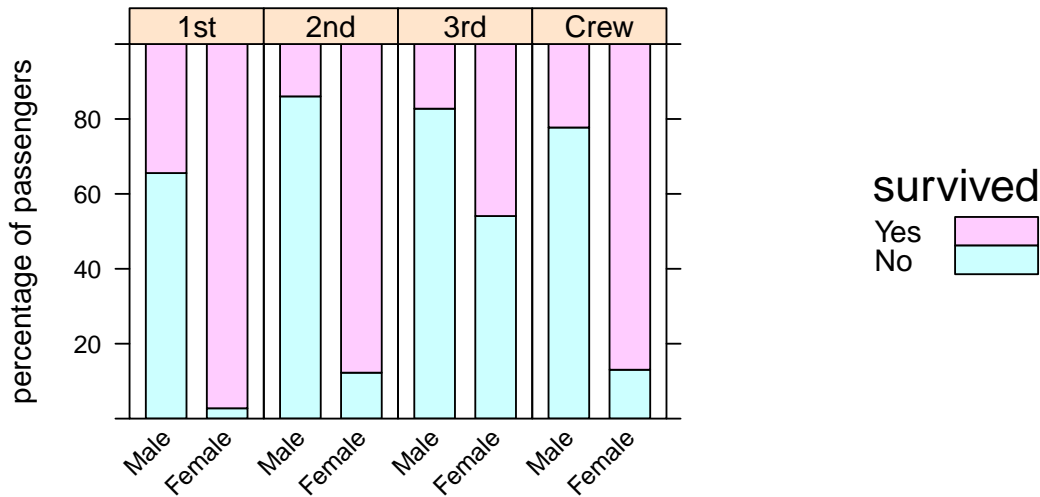
The output lends itself well to barcharts

```
tab_(Titanic, ~ Sex + Age) %>% barchart(auto.key=T)
```



```
tab__(Titanic, ~ Sex + Class + Survived, pct = c(1,2)) %>%  
  barchart(ylab = 'percentage of passengers',
```

```
horizontal = FALSE,  
ylim = c(0,100), layout = c(5,1),  
scales = list(x=list(rot=45)),  
auto.key=list(space='right',title='survived', reverse.row
```

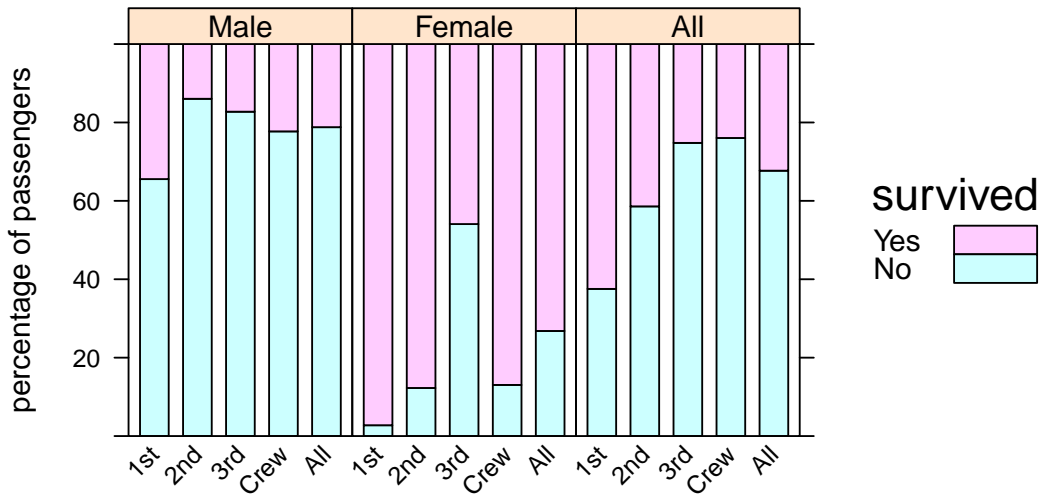


```

tab_(Titanic, ~ Class + Sex + Survived, pct = c(1,2)) %>%
  barchart(ylab = 'percentage of passengers',

```

```
horizontal = FALSE,  
ylim = c(0,100), layout = c(3,1),  
scales = list(x=list(rot=45)),  
auto.key=list(space='right',title='survived', reverse.row
```



## 9.9.2 Making marginal tables —

```
tab(Titanic, ~ Sex + Survived)
```

Sex	Survived		Total
	No	Yes	
Male	1364	367	1731
Female	126	344	470
Total	1490	711	2201

```
tab(Titanic, ~ Sex + Survived, pct = 1)
```

Sex	Survived		Total
	No	Yes	
Male	78.79838	21.20162	100.00000
Female	26.80851	73.19149	100.00000
All	67.69650	32.30350	100.00000

```
tab(Titanic, ~ Sex + Survived, pct = 1) %>%  
  round(1)
```

Sex	Survived		Total
	No	Yes	
Male	78.8	21.2	100.0
Female	26.8	73.2	100.0
All	67.7	32.3	100.0

```
tab(Titanic, ~ Sex + Survived + Age, pct = c(1,3)) %>%
  round(1)
```

```
, , Age = Child
```

Sex	Survived		Total
	No	Yes	
Male	54.7	45.3	100.0
Female	37.8	62.2	100.0
All	47.7	52.3	100.0

```
, , Age = Adult
```



	Survived		
Sex	No	Yes	Total
Male	79.7	20.3	100.0
Female	25.6	74.4	100.0
All	68.7	31.3	100.0

, , Age = All

	Survived		
Sex	No	Yes	Total
Male	78.8	21.2	100.0
Female	26.8	73.2	100.0
All	67.7	32.3	100.0

```
tab(Titanic, ~ Sex + Survived + Age, pct = c(1,3)) %>%
  round(1) %>%
  ftable
```

	Age			All
Sex	Survived	Child	Adult	

Male	No	54.7	79.7	78.8
	Yes	45.3	20.3	21.2
	Total	100.0	100.0	100.0
Female	No	37.8	25.6	26.8
	Yes	62.2	74.4	73.2
	Total	100.0	100.0	100.0
All	No	47.7	68.7	67.7
	Yes	52.3	31.3	32.3
	Total	100.0	100.0	100.0

```

tab(Titanic, ~ Sex + Age + Survived, pct = c(1,2)) %>%
  round(1) %>%
  ftable

```

		Survived	No	Yes	Total
Sex	Age				
Male	Child	54.7	45.3	100.0	
	Adult	79.7	20.3	100.0	
	All	78.8	21.2	100.0	
Female	Child	37.8	62.2	100.0	

	Adult	25.6	74.4	100.0
	All	26.8	73.2	100.0
All	Child	47.7	52.3	100.0
	Adult	68.7	31.3	100.0
	All	67.7	32.3	100.0

## 9.10 Frequency data frame —

From table to frequency data frame:

```
Titanic.df <- as.data.frame(Titanic)
brief(Titanic.df)
```

```
32 x 5 data.frame (27 rows omitted)
  Class Sex Age Survived Freq
  [f]   [f] [f]      [f] [n]
1  1st Male Child      No    0
2  2nd Male Child      No    0
3  3rd Male Child      No   35
. . .
```

31	3rd	Female	Adult	Yes	76
32	Crew	Female	Adult	Yes	20

## 9.11 Individual data frame —

One row per subject (i.e. passenger)

```
indices <- rep(1:nrow(Titanic.df), Titanic.df$Freq)
Titanic.ind <- Titanic.df[indices,]
Titanic.ind$Freq <- NULL
brief(Titanic.ind)
```

2201 x 4 data.frame (2196 rows omitted)

	Class	Sex	Age	Survived
	[f]	[f]	[f]	[f]
3	3rd	Male	Child	No
3.1	3rd	Male	Child	No
3.2	3rd	Male	Child	No
. . .				
32.18	Crew	Female	Adult	Yes

32.19 Crew Female Adult Yes

```
dd <- droplevels(subset(Titanic.ind, Class != 'Crew'))  
(fit <- glm(Survived ~ Class * Sex * Age, Titanic.ind,  
            subset = Class != 'Crew', family = binomial))
```

Call: glm(formula = Survived ~ Class \* Sex \* Age, family = binomial,  
data = Titanic.ind, subset = Class != "Crew")

Coefficients:

(Intercept)	Class2nd
1.657e+01	-1.466e-07
Class3rd	SexFemale
-1.756e+01	2.175e-07
AgeAdult	Class2nd:SexFemale
-1.729e+01	-1.996e-07
Class3rd:SexFemale	Class2nd:AgeAdult
7.962e-01	-1.670e+00

```
Class3rd:AgeAdult          SexFemale:AgeAdult
      1.664e+01                4.283e+00
Class2nd:SexFemale:AgeAdult Class3rd:SexFemale:AgeAdult
      -6.801e-02                -3.596e+00
```

```
Degrees of Freedom: 1315 Total (i.e. Null); 1304 Residual
Null Deviance:      1747
Residual Deviance: 1165    AIC: 1189
```

```
(fit <- glm(Survived ~ Class * Sex * Age,
            dd, family = binomial))
```

```
Call: glm(formula = Survived ~ Class * Sex * Age, family = binomi
data = dd)
```

```
Coefficients:
```

```
(Intercept)          Class2nd
      1.657e+01        -1.466e-07
```

Class3rd	SexFemale
-1.756e+01	2.175e-07
AgeAdult	Class2nd:SexFemale
-1.729e+01	-1.996e-07
Class3rd:SexFemale	Class2nd:AgeAdult
7.962e-01	-1.670e+00
Class3rd:AgeAdult	SexFemale:AgeAdult
1.664e+01	4.283e+00
Class2nd:SexFemale:AgeAdult	Class3rd:SexFemale:AgeAdult
-6.801e-02	-3.596e+00

Degrees of Freedom: 1315 Total (i.e. Null); 1304 Residual  
 Null Deviance: 1747  
 Residual Deviance: 1165 AIC: 1189

```
fit2 <- glm(Survived ~ (Class + Sex + Age)^2,
            dd, family = binomial)
summary(fit2)
```

Call:

```
glm(formula = Survived ~ (Class + Sex + Age)^2, family = binomial,  
     data = dd)
```

Deviance Residuals:

Min	1Q	Median	3Q	Max
-2.6771	-0.5952	-0.5952	0.3152	2.2293

Coefficients:

	Estimate	Std. Error	z value	Pr(> z )	
(Intercept)	16.26450	920.38635	0.018	0.986	
Class2nd	-0.82145	1005.81944	-0.001	0.999	
Class3rd	-17.25489	920.38641	-0.019	0.985	
SexFemale	3.59619	0.74781	4.809	1.52e-06	***
AgeAdult	-16.99213	920.38637	-0.018	0.985	
Class2nd:SexFemale	-0.06801	0.67120	-0.101	0.919	
Class3rd:SexFemale	-2.79995	0.56875	-4.923	8.52e-07	***
Class2nd:AgeAdult	-0.84881	1005.81949	-0.001	0.999	



```
Class3rd:AgeAdult    16.34159  920.38643  0.018  0.986
SexFemale:AgeAdult   0.68679    0.52541  1.307  0.191
```

```
---
```

```
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

(Dispersion parameter for binomial family taken to be 1)

```
Null deviance: 1746.8  on 1315  degrees of freedom
Residual deviance: 1165.4  on 1306  degrees of freedom
AIC: 1185.4
```

Number of Fisher Scoring iterations: 15

```
summary(fit)  # Why NAs? What next?
```

Call:

```
glm(formula = Survived ~ Class * Sex * Age, family = binomial,
     data = dd)
```

## Deviance Residuals:

Min	1Q	Median	3Q	Max
-2.6771	-0.5952	-0.5952	0.3152	2.2293

## Coefficients:

	Estimate	Std. Error	z value
(Intercept)	1.657e+01	1.073e+03	0.015
Class2nd	-1.466e-07	1.294e+03	0.000
Class3rd	-1.756e+01	1.073e+03	-0.016
SexFemale	2.175e-07	2.629e+03	0.000
AgeAdult	-1.729e+01	1.073e+03	-0.016
Class2nd:SexFemale	-1.996e-07	2.806e+03	0.000
Class3rd:SexFemale	7.962e-01	2.629e+03	0.000
Class2nd:AgeAdult	-1.670e+00	1.294e+03	-0.001
Class3rd:AgeAdult	1.664e+01	1.073e+03	0.016
SexFemale:AgeAdult	4.283e+00	2.629e+03	0.002
Class2nd:SexFemale:AgeAdult	-6.801e-02	2.806e+03	0.000
Class3rd:SexFemale:AgeAdult	-3.596e+00	2.629e+03	-0.001

	Pr(> z )
(Intercept)	0.988
Class2nd	1.000
Class3rd	0.987
SexFemale	1.000
AgeAdult	0.987
Class2nd:SexFemale	1.000
Class3rd:SexFemale	1.000
Class2nd:AgeAdult	0.999
Class3rd:AgeAdult	0.988
SexFemale:AgeAdult	0.999
Class2nd:SexFemale:AgeAdult	1.000
Class3rd:SexFemale:AgeAdult	0.999

(Dispersion parameter for binomial family taken to be 1)

Null deviance: 1746.8 on 1315 degrees of freedom  
Residual deviance: 1165.4 on 1304 degrees of freedom  
AIC: 1189.4

Number of Fisher Scoring iterations: 15

Anova(fit)

Analysis of Deviance Table (Type II tests)

Response: Survived

	LR	Chisq	Df	Pr(>Chisq)
Class	114.88	2	< 2.2e-16	***
Sex	318.53	1	< 2.2e-16	***
Age	20.34	1	6.486e-06	***
Class:Sex	64.07	2	1.220e-14	***
Class:Age	37.26	2	8.101e-09	***
Sex:Age	1.69	1	0.1942	
Class:Sex:Age	0.00	2	1.0000	

---

Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1

```
anova(fit, fit2)
```

Analysis of Deviance Table

Model 1: Survived ~ Class \* Sex \* Age

Model 2: Survived ~ (Class + Sex + Age)^2

	Resid. Df	Resid. Dev	Df	Deviance
1	1304	1165.4		
2	1306	1165.4	-2	-1.506e-06

## 9.12 Frequency data frame with response variable on rows —

```
brief(Titanic.df)
```

32 x 5 data.frame (27 rows omitted)

Class	Sex	Age	Survived	Freq
[f]	[f]	[f]	[f]	[n]

1	1st	Male	Child	No	0
2	2nd	Male	Child	No	0
3	3rd	Male	Child	No	35
.	.	.	.	.	.
31	3rd	Female	Adult	Yes	76
32	Crew	Female	Adult	Yes	20

```
Titanic.wide <-
  towide(Titanic.df,
        idvar = c('Class','Sex','Age'),
        timevar = 'Survived')
fitbin <- glm(
  cbind(Freq_No,Freq_Yes) ~ Class * Sex * Age,
  Titanic.wide, subset = Class != "Crew",
  family = binomial)
Anova(fitbin)
```

Analysis of Deviance Table (Type II tests)

Response: cbind(Freq\_No, Freq\_Yes)

	LR	Chisq	Df	Pr(>Chisq)	
Class	114.88	2	< 2.2e-16	***	
Sex	318.53	1	< 2.2e-16	***	
Age	20.34	1	6.486e-06	***	
Class:Sex	64.07	2	1.220e-14	***	
Class:Age	37.26	2	8.101e-09	***	
Sex:Age	1.69	1	0.1942		
Class:Sex:Age	0.00	2	1.0000		

---

Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1

```
summary(fitbin)
```

Call:

```
glm(formula = cbind(Freq_No, Freq_Yes) ~ Class * Sex * Age, family  
     data = Titanic.wide, subset = Class != "Crew")
```

Deviance Residuals:

[1] 0 0 0 0 0 0 0 0 0 0 0 0 0

Coefficients:

	Estimate	Std. Error	z value
(Intercept)	-2.554e+01	9.518e+04	0
Class2nd	-6.665e-01	1.307e+05	0
Class3rd	2.653e+01	9.518e+04	0
SexFemale	9.704e-01	1.619e+05	0
AgeAdult	2.626e+01	9.518e+04	0
Class2nd:SexFemale	-1.121e+00	2.053e+05	0
Class3rd:SexFemale	-1.767e+00	1.619e+05	0
Class2nd:AgeAdult	2.337e+00	1.307e+05	0
Class3rd:AgeAdult	-2.561e+01	9.518e+04	0
SexFemale:AgeAdult	-5.253e+00	1.619e+05	0
Class2nd:SexFemale:AgeAdult	1.189e+00	2.053e+05	0
Class3rd:SexFemale:AgeAdult	4.567e+00	1.619e+05	0
	Pr(> z )		
(Intercept)	1		
Class2nd	1		



Class3rd	1
SexFemale	1
AgeAdult	1
Class2nd:SexFemale	1
Class3rd:SexFemale	1
Class2nd:AgeAdult	1
Class3rd:AgeAdult	1
SexFemale:AgeAdult	1
Class2nd:SexFemale:AgeAdult	1
Class3rd:SexFemale:AgeAdult	1

(Dispersion parameter for binomial family taken to be 1)

Null deviance: 5.8140e+02 on 11 degrees of freedom  
Residual deviance: 3.0911e-10 on 0 degrees of freedom  
AIC: 60.924

Number of Fisher Scoring iterations: 23

Compare with:

Anova(fit)

Analysis of Deviance Table (Type II tests)

Response: Survived

	LR	Chisq	Df	Pr(>Chisq)	
Class	114.88	2	< 2.2e-16	***	
Sex	318.53	1	< 2.2e-16	***	
Age	20.34	1	6.486e-06	***	
Class:Sex	64.07	2	1.220e-14	***	
Class:Age	37.26	2	8.101e-09	***	
Sex:Age	1.69	1	0.1942		
Class:Sex:Age	0.00	2	1.0000		

---

Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1

## summary(fit)

Call:

```
glm(formula = Survived ~ Class * Sex * Age, family = binomial,  
     data = dd)
```

Deviance Residuals:

Min	1Q	Median	3Q	Max
-2.6771	-0.5952	-0.5952	0.3152	2.2293

Coefficients:

	Estimate	Std. Error	z value
(Intercept)	1.657e+01	1.073e+03	0.015
Class2nd	-1.466e-07	1.294e+03	0.000
Class3rd	-1.756e+01	1.073e+03	-0.016
SexFemale	2.175e-07	2.629e+03	0.000
AgeAdult	-1.729e+01	1.073e+03	-0.016

Class2nd:SexFemale	-1.996e-07	2.806e+03	0.000
Class3rd:SexFemale	7.962e-01	2.629e+03	0.000
Class2nd:AgeAdult	-1.670e+00	1.294e+03	-0.001
Class3rd:AgeAdult	1.664e+01	1.073e+03	0.016
SexFemale:AgeAdult	4.283e+00	2.629e+03	0.002
Class2nd:SexFemale:AgeAdult	-6.801e-02	2.806e+03	0.000
Class3rd:SexFemale:AgeAdult	-3.596e+00	2.629e+03	-0.001

Pr(>|z|)

(Intercept)	0.988
Class2nd	1.000
Class3rd	0.987
SexFemale	1.000
AgeAdult	0.987
Class2nd:SexFemale	1.000
Class3rd:SexFemale	1.000
Class2nd:AgeAdult	0.999
Class3rd:AgeAdult	0.988
SexFemale:AgeAdult	0.999
Class2nd:SexFemale:AgeAdult	1.000

Class3rd:SexFemale:AgeAdult 0.999

(Dispersion parameter for binomial family taken to be 1)

Null deviance: 1746.8 on 1315 degrees of freedom  
Residual deviance: 1165.4 on 1304 degrees of freedom  
AIC: 1189.4

Number of Fisher Scoring iterations: 15

## 10 Using R Script with Markdown —

Here's a posting that describes quite well the difference between an R Markdown script (with extension .Rmd) and a .R script with Markdown. The main advantages of the latter are expressed well:

- 1) you don't need to transform your original .R script manually into a .Rmd script and
- 2) the same script can be run interactively in R and be used to generate a

clean report.

One problem is that Ctrl-Shift-K produces diagnostics that refer to line numbers in the .Rmd file, whose numbering can be very different from that of the .R file. When this happens you can ‘knit’ the .R file in a way that keeps the intermediate .Rmd file by using the command:

This will leave the intermediate files in your directory so you can interpret error messages.

## 11 Attributes —

The attributes of an object work like Post-it notes on the object. When functions use the object, they can consult the attributes to decide how to use it.

For example, a matrix is stored as a long vector recording the contents of the matrix column by column. The object itself has no information about the dimension of the matrix. The contents of a 3 by 4 matrix could just as easily be a 2 by 6 matrix or a 1 by 12 matrix or, indeed, just a vector of length 12. Functions that use the object as a matrix know what to do with the 12 numbers

because of the ‘dim’ attribute.

```
m <- matrix(1:12, 3, 4)
colnames(m) <- letters[1:4]
rownames(m) <- LETTERS[1:3]
attributes(m)
```

```
$dim
```

```
[1] 3 4
```

```
$dimnames
```

```
$dimnames[[1]]
```

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

```
$dimnames[[2]]
```

```
[1] "a" "b" "c" "d"
```

Many attributes are set by the function creating the object. For example the dim attribute is set by the ‘matrix’ function:

```
m <- matrix(1:12, 3, 4)
attributes(m)
```

```
$dim
[1] 3 4
```

Many attributes can also be set by **replacement** functions and they can be read by the cognate regular function of the corresponding name. For example, you can read and change the shape of a matrix with the 'dim' functions.

```
dim(m)
```

```
[1] 3 4
```

```
m
```

	[,1]	[,2]	[,3]	[,4]
[1,]	1	4	7	10
[2,]	2	5	8	11
[3,]	3	6	9	12



```
dim(m) <- c(2,6)
```

```
m
```

	[,1]	[,2]	[,3]	[,4]	[,5]	[,6]
[1,]	1	3	5	7	9	11
[2,]	2	4	6	8	10	12

### 11.0.1 Exercises —

1. What happens if you try to set a dimension that doesn't correspond to the size of the matrix?
2. What happens to column and row names if you change the dimension of a matrix?

Other familiar functions that read attributes of a matrix are 'nrow', 'ncol', 'row', 'dimnames'. A very important attribute used for OOP is the 'class' attribute.

See also the 'attr' and its replacement to create and read new attributes.

Here are the attributes of the 'Guyer1' data frame.

```
attributes(Guyer1)
```

```
$names
```

```
[1] "cooperation" "condition"    "sex"
```

```
$class
```

```
[1] "data.frame"
```

```
$row.names
```

```
[1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
```

```
dim(Guyer1)
```

```
[1] 20 3
```

## 12 Traps and Pitfalls —

Contribute traps and pitfalls on Piazza

Some of these observations may change as R develops. It would be a good idea to add the version of R in which each behaviour was observed.

## **12.1 Factors —**

Many of the tricky silent traps are encountered in the use of factors.

### **12.1.1 Transformation of factors to characters or codes —**

In its raw form, a factor is a vector of integers that provides indices into a vector of ‘levels’ for the factor. The levels are attached as an attribute to the factor.

A factor vector can be coerced to its character form or to its numerical indices:

Most functions operating on factors use either the factor’s character form or its numerical form. In most cases, the form used is the only sensible one and there are no surprises. Sometimes the result is not what the user expected and mysterious bugs or outright errors can be produced.

### 12.1.2 Factors transformed to character —

The following functions use the character form of the factor:

### 12.1.3 Factors transformed to numeric —

The following functions use the numeric form. In the first case (indexing) that might seem to be the only sensible interpretation. However, since it is possible to index by name in R, a user could intend to use the character values of a factor to index names but end up with an entirely different result.

In the second case, ('rbind'), the use of numeric values seems contrary to expectation considering the behaviour of 'matrix' above.

Then using 'rbind' with a factor and a character, the coercion of the factor to character occurs **after** extracting the numeric codes.

### 12.1.4 Factors operations that return a factor —

Some operators on factors return a factor:

### 12.1.5 Other special factor pitfalls —

Special pitfalls can occur when attempting to transform a factor whose levels are character representations of numbers into a numeric object:

Note in passing that the levels have been ordered numerically instead of lexicographically, as would have been the case if the argument to ‘factor’ had been `c(‘1’,‘10’,‘2’)`. Thus the ‘factor’ function is ‘numeric-smart’. `facn` almost seems numeric but it is not:

either `as.character` nor `as.numeric` returns the original numeric vector:

To get the original numeric vector, one must compose both:

or, one can define a function:

### 12.1.6 ‘drop’ doesn’t work with subset —

doesn’t drop levels in ‘id’ (as it should?). Instead, use:

## 12.2 diag can be tricky —

If you use `diag` in a function to get the main diagonal of a matrix (not necessarily square) you might get a bug if you happen to have a  $1 \times 1$  matrix represented by a scalar (vector of length 1) because:

```
m <- matrix(1:12,3)
```

```
m
```

```
      [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
```

```
diag(m)
```

```
[1] 1 5 9
```

```
m <- 3.2
```

```
diag(m) # Why?
```

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

```
[1,]    1    0    0
[2,]    0    1    0
[3,]    0    0    1
```

If you want to use `diag` in a way that won't give you an identity matrix when the argument happens to be a scalar, the safe way is:

```
diag(as.matrix(m)) # gives you what you want in any case
```

```
[1] 3.2
```

Here's another example where 'diag' can fail.

Many algorithms using eigenvalue or singular value decompositions (with 'eigen' or 'svd') form a diagonal matrix with the vector of eigen/singular values using the 'diag' function, e.g.

This will fail if the rank of  $X$  is equal to 1 since, in that case, `diag(d.inv)` will be an identity matrix of dimension 'floor(d.inv)', while what is needed is a 1 x 1 matrix with a single element 'd.inv'. One solution is to use:

Another is to use the fact that *matrix* premultiplication by a diagonal matrix is

the same as *scalar* premultiplication by the vector of diagonal elements. This is so because multiplying the vector by the matrix causes the vector to be recycled to the length of the matrix and pairwise scalar multiplication takes place column by column for the matrix.

Note that extra parentheses are needed because these multiplications are not associative.

## **12.3 Reading and Writing Data Files —**

### **12.3.1 NA as a valid value (the Namibia problem) —**

Many commands that read data files, e.g. `read.csv` and `read.xls` in the package `gdata`, will, by default, treat the string ‘NA’ as a missing value whether it occurs in a character or a numeric variable. In numeric variables, blanks are also turned into missing values. If ‘NA’ occurs as a valid value, for example the two-character ISO country code for Namibia, then you may use the argument ‘`na.strings = NULL`’ to ensure that ‘NA’ is not turned into a missing value. However, NA’s used to indicate missing numeric values will now be interpreted as valid character values and numeric variables with NA’s will be read as factors.



## 12.4 Prediction —

### 12.4.1 Prediction with nlme —

To get

to produce pp of length equal to 'nrow(dd)', you can use the following combination of 'na.action's:

### 12.4.2 Exercises

1. What is the difference between the result of: `fac <- factor(letters)`  
`levels(fac) <- rev(letters)` and `fac <- factor(letters, levels = rev(letters))`

## 13 Useful Techniques and Tricks —

Contribute “how to’s” and useful tricks on Piazza.

## 13.1 Changing all variables to characters in a data frame —-

When data frames are being manipulated only as data sets, not for immediate statistical analyses, it is often convenient to have all variables as characters to avoid problems due to the inconsistent behaviour of factors. A very easy way to do this, if `dd` is a data frame:

```
dd[] <- lapply(dd, as.character)
```

Any side effects?

- Some variable attributes may be lost with `as.character`.

## References —-

Fox, John, and Sanford Weisberg. 2019. *An R and S-Plus Companion to Applied Regression*. 3rd ed. Sage Publications.