Data Input and Recoding I

Tabular Data Formats

Paul E. Johnson[1,2]

[1]Department of Political Science
University of Kansas

[2]Center for Research Methods and Data Analysis
University of Kansas

2013

Outline

The Usual: Use One Rectangular Data Set

- Following the introduction of SPSS in 1968, social scientists became accustomed to the idea of using a "rectangular data set."

$Var1$	$Var2$	$Var3$	$Var4$	$Var5$
1	5	6	4	31
2	2	3	5	29
3	4	5	5	53
4	2	2	3	22

(1)

- The first column was usually a "respondent identifier"

Variable name; Value Label

Value The values of the variables were usually kept in a numeric format.

Value Label A mapping from numerical values to substantive labels

Codebook A listing of values and labels, e.g.

Var2	
Value	Label
1	Never
2	Sometimes
3	Often
4	Always
8	Don't Know
9	Other Missing

R Terminology

- data frame
  - combine equal-sized columns side-by-side
  - columns can be different data types
  - inside R guts, a data frame is really just an R list, with the "equal length columns" requirement
- variable: a column of information
  - numeric (floating point)
  - integer
  - factor (a categorical indicator)
  - ordered factor (a factor that some procedures treat differently)
  - character

Accessing columns V1 V2 V3 in "dat"

- First, be polite. Ask the data frame what its column names are!

```
colnames(dat)
```

  Suppose the names are V1, V2, V3, V4.
- After that, there are many ways to access a column
  - Use the Dollar Sign: `dat$V1`
  - Use R list notation `dat[["V1"]]`
  - Ask for column number, as if a matrix `dat[, 1]`
  - Ask for columns by name `dat[, c("V1","V2")]`
- These access methods are all equivalent, but some are easier to "fit" into your program than others.
- Retrieving one column creates an R vector, not an $N \times 1$ data frame. Make a mental bookmark for the "drop gotcha" problem, I should have a blog post about it. (But can explain it to you if you need).

The CSV and other text formats

- Suppose the variables are in a file that looks like this:

```
id , age , momage, dadage , iq
1, 14, 33, 36, 117
2, 17, 40, 44, 111
```

- Row 1 is a "header" line
- The separator is the symbol ","
- This is a "free field" format, only the separator and the values matter. Column positioning is ignored.
- We seldom encounter fixed field formats today, but they can be managed.

Little warning about text storage format

- We used to (some still do) call this ASCII data (American Standard Code for Information Interchange)
- In 1990s, encoding formats proliferated, and today it is very unlikely that you actually have ASCII text in a text file.
- ASCII won't recognize slanted quotation marks or many other symbols that are comon today.
- "unicode" is an internationalized character encoding format that is attempting to displace the many formats that have been used.
- We *hope* these character formats "just work" when you use R, but if they don't, we have ways to convert text storage formats. Web search: "locale", "iconv".

Reading raw text files into R

- `read.table()` is the workhorse. I use this instead of type-specific read functions like read.csv.

```
dat <- read.table(file = "whatever.txt", header =
    TRUE, sep = ",")
```

- Key options
  - file="whatever.txt"
  - header=TRUE: specifies first row is variable names. If no header line exists, specify FALSE
  - sep= the separator character
    - space is default, omit sep option
    - "\t" tab
    - "," comma
    - "|" "bar"

Example

- In my current working directory, I have a subfolder called "examples". Look for a file called "practiceData.txt".

```
dat <- read.table("examples/practiceData.txt",
    header = TRUE)
```

- Review the first few lines

```
head(dat)
```

```
  grp ed   inc mar sex
1   1 11 44444   Y Mal
2   2 10 34343   Y Fem
3   1  9 11112   N Mal
4   3 15 23232   N Fem
5   1  7 23111   Y Fem
6   1 12 78787   N Mal
```

Look that over

- > dat # same as print(dat)
- > str(dat) # gives item-by-item information

```
str(dat)
```

```
'data.frame': 19 obs. of  5 variables:
 $ grp: int  1 2 1 3 1 1 2 2 2 1 ...
 $ ed : int  11 10 9 15 7 12 8 6 11 20 ...
 $ inc: int  44444 34343 11112 23232 23111 78787 33233
     22312 32322 76755 ...
 $ mar: Factor w/ 3 levels "N","W","Y": 3 3 1 1 3 1 3 1
     3 1 ...
 $ sex: Factor w/ 2 levels "Fem","Mal": 2 1 2 1 1 2 2 1
     1 1 ...
```

- Confirm presence of column names

```
colnames(dat)
```

```
[1] "grp" "ed"  "inc" "mar" "sex"
```

More Snooping on "dat" object

- R has a method summary.data.frame, which is called here:

```
summary(dat)
```

```
      grp             ed              inc          mar       sex
Min.   :1   Min.   : 6.00   Min.   :11112   N:9   Fem:
     9
1st Qu.:1   1st Qu.: 9.50   1st Qu.:23172   W:1   Mal
    :10
Median :2   Median :12.00   Median :34333   Y:9
Mean   :2   Mean   :12.37   Mean   :38764
3rd Qu.:3   3rd Qu.:14.50   3rd Qu.:44394
Max.   :3   Max.   :20.00   Max.   :78787
```

- rockchalk package has function `summarize()` that has some conveniences.

```
rockchalk::summarize(dat)
```

More Snooping on "dat" object ...

```
$numerics
             ed      grp        inc
0%      6.000   1.0000      11110
25%     9.500   1.0000      23170
50%    12.000   2.0000      34330
75%    14.500   3.0000      44390
100%   20.000   3.0000      78790
mean   12.370   2.0000      38760
sd      4.072   0.8165      18760
var    16.580   0.6667  351900000
NA's    0.000   0.0000          0
N      19.000  19.0000         19

$factors
                mar                          sex
 N                : 9.0000    Mal              :10.000
 Y                : 9.0000    Fem              : 9.000
 W                : 1.0000    NA's             : 0.000
 NA's             : 0.0000    entropy          : 0.998
 entropy          : 1.2448    normedEntropy    : 0.998
 normedEntropy    : 0.7854    N                :19.000
```

More Snooping on "dat" object ...

```
N                :19.0000
```

- Many other functions can be run "on" the data frame to find out more about it.
  - Ask for "dimensions": rows and columns

```
dim(dat)
```

```
[1] 19  5
```

  - Check for presence of row names

```
rownames(dat)
```

```
 [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "
     10" "11" "12" "13" "14" "15" "16" "17" "18" "
     19"
```

  - `names()` will reveal names even for a non-rectangular collection of stuff (an R list, for example) .

More Snooping on "dat" object ...

```
names(dat)
```

```
[1] "grp" "ed"  "inc" "mar" "sex"
```

- Review any attributes of the data frame object. Again, "names" attribute is the thing we get by explicitly asking for colnames (admittedly confusing).

```
attributes(dat)
```

```
$names
[1] "grp" "ed"  "inc" "mar" "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
```

Exercise: Test your skills on trouble files

- trouble-1.txt and trouble-2.txt are in examples folder.
- ```
  dat <- read.table("examples/trouble-1.txt")
  ```
- Different thing wrong with this

  ```
  dat <- read.table("examples/trouble-2.txt")
  ```
- Note: You will need to open these files in a "flat text" editor to see what's in them–don't use MSWord or Excel. Do use any programming file editor, such as Emacs, Notepad++, RStudio
  ```

Exercise: Test your skills on trouble-1.txt and trouble-2.txt

- There are GUI Spread-Sheetish thing
- Look, don't touch

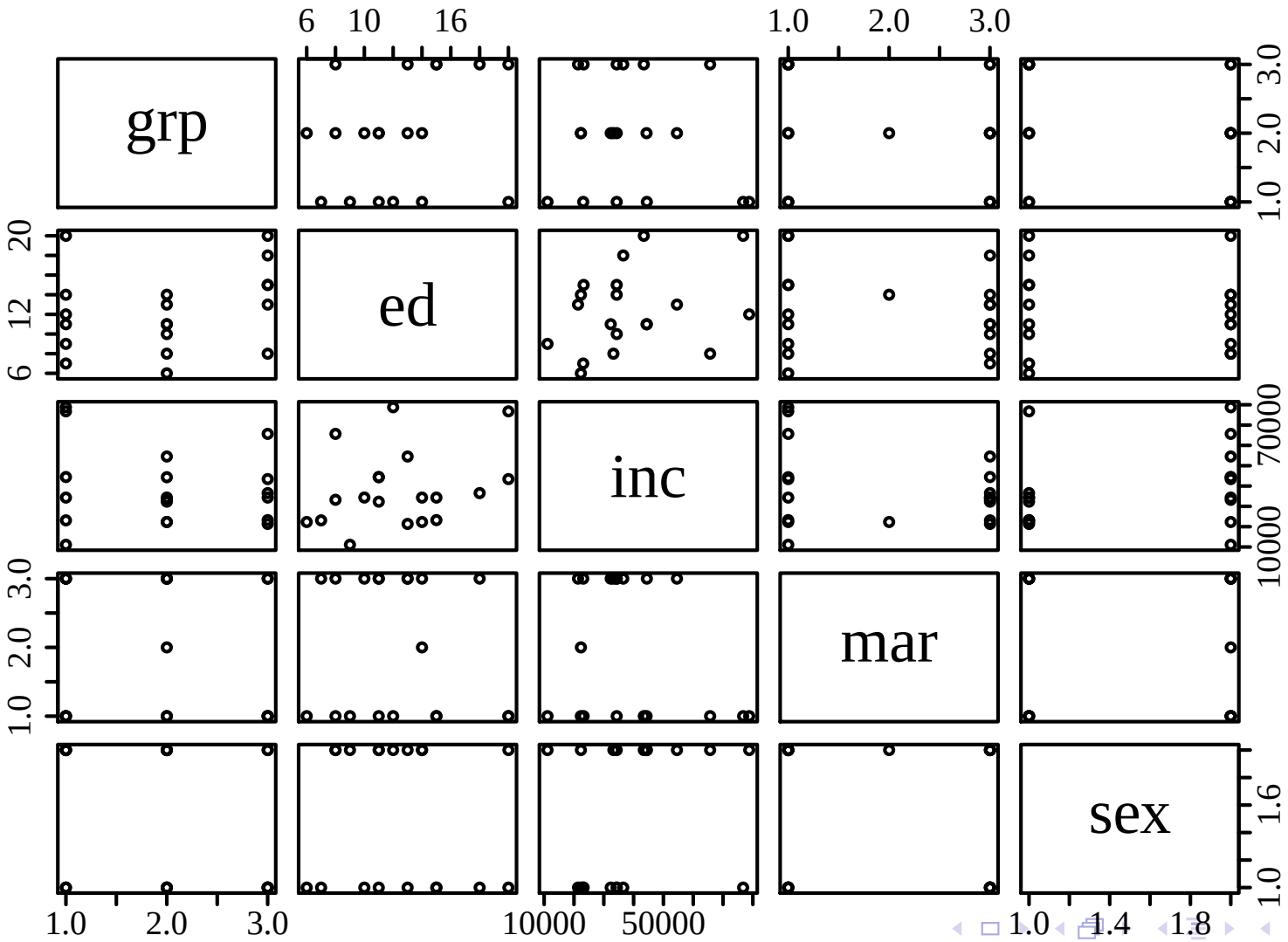
```
View(dat)
```

- Dangerous: allows you to damage data!

```
fix(dat)
```

There's even a plot method for data frames

```
plot(dat)
```



Want to save that data frame as text?

- Creates a text file "newPractice.txt" in the current working directory.

```
write.table(dat, file = "newPractice.txt", row.names
    = FALSE, sep = "|")
```

- This creates the file in the subdirectory "examples"

```
write.table(dat, file = "examples/newPractice.txt",
    row.names = FALSE, sep = "|")
```

- `row.names = FALSE` is important. Otherwise, R this DOES NOT create a rectangular data structure (try it and see).
- Sometimes its helpful to specify an unusual separator, but don't forget to use same separator when re-opening the table
- Note that factor lables are written out as character strings

```
dat <- read.table("examples/newPractice.txt", sep = "
    |", header = TRUE)
```

Outline

Most Common Problem: "File not found"

Suppose you try to load a file and this bad thing happens?

```
> dat <- read.table(file="nonexistant.txt", header=T)
```

```
Error in file(file, "rt") : cannot open the connection
In addition: Warning message:
In file(file, "rt") :
  cannot open file 'nonexistant.txt': No such file or
      directory
```

- First, check that the file is in the current working directory `list.files()`
- Did you misspell something?

What to do if you don't see the file . . .

You have at least 3 options. I strongly recommend the first as a part of your file organization scheme.

1. Move the file into the current working directory
   Ask "where am I are now?":

   ```
   getwd()
   ```

   And copy the file in there (or into a subfolder in there)!
   If you ever say "I don't know what my working directory is" it means you aren't doing your work properly. Consider changing your work habits: Open an R file in an editor that knows about R and helps R start in that location (i.e, do NOT start R from an icon on the desktop that is disconnected from the folder in which you intend to work).

2. Change the current working directory

   ```
   > setwd("a-valid-path-specifier-here")
   ```

What to do if you don't see the file . . . ...

```
> setwd("/home/pauljohn/Wherever")
```

R can talk to Windows directories, use forward slashes

```
> setwd("c:/Users/pauljohn/Wherever")}
```

3. Revise the file option to specify a full path.

```
> ffn <- paste("c:/Users/pauljohn/XYZ/practiceData.txt"
    )}
> read.table(file= ffn, header=T)}
```

- Method 1 is best: keeps everything together.
- I only use Method 3 when there is one data frame being used among several separate R projects.

What if You Have Compressed Text Files?

- Some programs generate huge text files and disk space is gobbled up!
- We should compress files with programs like gzip or bzip2
- These are preferred to the proprietary "zip" compression format.
- Free/Open Source programs available for all platforms, "7-zip", etc.
- R can read in a compressed file like so:

```
dat2 <- read.table(file=gzfile("examples/
    practiceData.txt.gz"), header=T)
identical(dat, dat2)
```

```
[1] TRUE
```

- For bz2 files, use `bzfile` instead of `gzfile`
- R can also create compressed files, so that saving text output may not cause the disk to fill up.

Need to Download a File?

- Possible to read from Web files "on the fly" like so

```
dat <- read.table(url("http://pj.freefaculty.org/
    guides/Rcourse/data-1/examples/practiceData.txt"),
     sep = ",", header = TRUE)
```

- Disadvantages
  - nothing "saved"
  - requires always-on Internet
  - difficult to debug
- Instead, I suggest

```
download.file("http://pj.freefaculty.org/guides/
    Rcourse/data-1/examples/practiceData.txt",
    destfile = "practiceData.txt")
```

  Then use `read.table()` to import that.
- Abstract the file name definitions, look more "professional"

Need to Download a File? ...

```
fn1 <- "practiceData.txt"
addr <- "http://pj.freefaculty.org/guides/Rcourse/
    data-1/examples"
download.file(paste0(addr, fn1), destfile = fn1)
dat <- read.table(fn1, header = TRUE, sep = ",")
```

Outline

Easy to Add and Remove Variables

- To remove a variable, simply set it to NULL. Any of these will work:

```
dat$ed <- NULL
dat[[``ed'']] <- NULL
dat[ , c(``ed'')] <- NULL
```

- Add a variable. Name a column using any of the usual methods. For example,

```
dat$noise <- rnorm(nrow(dat), m = 444, sd = 234)
dat[["moreNoise"]] <- rnorm(nrow(dat), m = 0, sd = 1)
```

- Copy ed to a new variable name

```
dat$newed <- dat$ed
```

- Remove the original ed

```
dat[["ed"]] <- NULL
```

Easy to Add and Remove Variables ...

- I'll undo this damage to dat later.

```
colnames(dat)
```

```
[1] "grp"       "inc"       "mar"       "sex"       "
    noise"      "moreNoise" "newed"
```

```
rockchalk::summarize(dat)
```

```
$numerics
          grp       inc moreNoise  newed   noise
0%     1.0000     11110   -1.6620  6.000    18.6
25%    1.0000     23170   -0.4029  9.500   352.1
50%    2.0000     34330    0.4912 12.000   418.4
75%    3.0000     44390    1.1340 14.500   588.6
100%   3.0000     78790    2.1970 20.000   869.3
mean   2.0000     38760    0.3464 12.370   459.2
sd     0.8165     18760    1.1950  4.072   200.1
var    0.6667 351900000    1.4290 16.580 40040.0
NA's   0.0000         0    0.0000  0.000     0.0
N     19.0000        19   19.0000 19.000    19.0
```

Easy to Add and Remove Variables ...

```
$factors
                mar                              sex
 N                : 9.0000    Mal                 :10.000
 Y                : 9.0000    Fem                 : 9.000
 W                : 1.0000    NA's                : 0.000
 NA's             : 0.0000    entropy             : 0.998
 entropy          : 1.2448    normedEntropy       : 0.998
 normedEntropy    : 0.7854    N                   :19.000
 N                :19.0000
```

What If the Data Frame has the Wrong Column Names?

- Recall, colnames() displays the names.
- R philosophy: provide similarly named assignment function
  - Re-name all of them in one swipe:

```
colnames(dat) <- c("colone", "coltwo", "three", "
    andSoForth")
```

  - Just rename one at a time, for example, change the second column name to "columntwo"

```
colnames(dat)[2] <- ``columntwo''
```

- Because this is error prone, I tend to be more verbose about when I really want to get it right.
  - First, Catch that vector of names and look it over

```
mycolnames <- colnames(dat)
mycolnames
```

What If the Data Frame has the Wrong Column Names?

...

```
[1] "grp"        "inc"        "mar"        "sex"
          "noise"      "moreNoise" "newed"
```

- Edit the vector of names

```
origname <- mycolnames[2] ##need a copy
mycolnames[2] <- "WhateverPJSays"
colnames(dat) <- mycolnames
colnames(dat)
```

```
[1] "grp"             "WhateverPJSays" "mar"
                "sex"                "noise"              "
    moreNoise"       "newed"
```

- Better put it back the way it was (or else the rest of the program won't work). And I'd better restore the ed variable while I'm at it.

What If the Data Frame has the Wrong Column Names?

...

```
colnames(dat)[2] <- origname
dat$ed <- dat$newed
colnames(dat)
```

```
[1] "grp"       "inc"       "mar"       "sex"
          "noise"     "moreNoise" "newed"     "ed"
```

Outline

Cleaning Up Typographical Errors in Data

- Resist the temptation to edit the data file directly with Excel or such (non-traceable changes are dangerous)
- Use any preferred method to scan data and detect troubles.
- Write code to recode for typographical errors.
- Example: Change the value of dat$ed from 47 to 17.
  - Recode: take the column "ed" (as dat$ed) and then, in that vector, find the index of values that are equal to 47, and change them to 17.

```
dat$ed[dat$ed==47] <- 17
```

  - That is two equal signs together
  - Equivalent alternative coding

```
dat[dat$ed==47, "ed"] <- 17
```

Cleaning Up Typographical Errors in Data ...

- We can grab particular row and column values by their numerical position if we want

```
dat[7, 2] <- 17
```

- Use %in%: Its a Multiple Matcher!

```
dat$ed[dat$ed %in% c(190, 191, 192, 200) ] <- 99
```

Outline

Maybe There Are Missing Value Indicators?

- Out-of-range scores like “99” or “999” may mean “unavailable” or “don’t know” or some other “missing value”
- It may be necessary to manually mark those scores as missing
- R uses NA as the value for missings
- NA is a “symbol”, can be assigned as if it were a numerical value

```
dat$ed[dat$ed==99] <- NA
```

- Call any `dat$noise` value bigger than the mean a missing

```
dat$noise[dat$noise > mean(dat$noise)] <- NA
```

- Or use %in% to collect mutiple discrete values

```
dat$ed[dat$ed %in% c(110, 190, 191, 192, 200) ] <- NA
```

Anticipate Missing Codes when Importing Data

- Suppose some SAS user gives you a file with some periods where you wish you had NA
- R won't understand that:

```
dat2 <- read.table("examples/newp.sas.txt",header=T,
    sep="|")
str(dat2$noise)
```

```
Factor w/ 10 levels ".","-186.784638260593",..: 1 7 1
   9 10 NA 1 NA NA 6 ...
```

- R treats period "." as a letter, so the whole column is treated as a character variable (which, by default, is converted to a factor)
- Don't manually edit the file
- Do revise your R code: Specify the missing strings and it will be OK!

Anticipate Missing Codes when Importing Data ...

```
dat2 <- read.table("examples/newp.sas.txt", header =
    TRUE, sep = "|", na.strings = c("NA","."))
str(dat2$noise)
```

```
num [1:19] NA 321 NA 346 363 ...
```

Outline

Numbers are Easy

- First, make sure a variable really is a number. Should have no attributes:

```
attributes(dat$ed)
```

```
NULL
```

```
is.numeric(dat$ed)
```

```
[1] TRUE
```

```
is.factor(dat$ed)
```

```
[1] FALSE
```

Numerical Recoding: As Easy as Math

- Want the log?

```
dat$edlog <- log(1 + dat$ed)
dat$edsqrt <- sqrt(dat$ed)
dat$edexp <- exp(dat$ed)
head(dat)
```

	grp	inc	mar	sex	noise	moreNoise	newed	ed	edlog	edsqrt	edexp
1	1	44444.142	Y	Mal	NA	0.2987237	11	11	2.484907	3.316625	59874
2	2	34343.466	Y	Fem	NA	0.7796219	10	10	2.397895	3.162278	22026
3	1	11112.084	N	Mal	418.4230	1.4557851	9	9	2.302585	3.000000	8103
4	3	23232.372	N	Fem	337.8817	−0.6443284	15	15	2.772589	3.872983	3269017
5	1	23111.633	Y	Fem	NA	−1.5531374	7	7	2.079442	2.645751	1096
6	1	78787.791	N	Mal	18.5983	−1.5977095	12	12	2.564949	3.464102	162754

- NB 1: I don't usually obliterate old variables. Create new instead.

Numerical Recoding: As Easy as Math ...

- NB 2: Suggested naming scheme keeps original variable name and appends new letters. This keeps similar variables alphabetically grouped. (Do NOT use dat$loged. DO use dat$edlog).

Outline

What is a Factor?

- A factor is a structured thing ("look-up table"), with numbers and labels.
  - R's internal numerical score, always 1, 2, 3, 4, ...
  - A list of labels of "levels" for each number
  - The idea behind factors is that statistical routines should be *smart* enough to give you the correct answer, depending on whether your data is numeric or categorical.

Internal Value	Label
1	Catholic
2	Protestant
3	Muslem
4	Hindu
5	Jewish

Little Factor Wrinkles

- Unlike SPSS, where users can assign any numerical scores they want for values, R always uses consecutive 1,2,3, ...
- Those internal scores are what you get when you use `as.numeric()` on a factor.
- So, if you "import" an SPSS dataset and allow R to convert those variables to factors, the SPSS coded values 1, 3, 5, 7, 9 will be lost forever, R will internally re-number that 1, 2, 3, 4, 5. You can NEVER recover the SPSS numeric scores.
- R gets the labels right. From R's point of view, the separate labels are the only important information. The numbering is not important. (Only silly people base any work on the integers associated with factor levels.)

Convert Numeric to Factor

This arises in 2 contexts, which we treat separately.

1. A numeric variable coded 1, 2, 3 should become a factor variable with discrete labels like c("Catholic" "Protestant" "Muslem") or c("Midwest","South", "East").
2. A numeric variable has to be grouped into ranges ("low", "medium", and "high")

Convert "grp" into R factor

- Recall dat's variable grp

```
dat$grp
```

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

- We want to faithfully reproduce that, without re-grouping or losing values.

The `factor()` function

- The `factor` function converts existing values into characters and enters them as factor levels, alphabetically
- Try that without entering any detailed arguments

```
dat$grpfac1 <- factor(dat$grp)
str(dat$grpfac1)
```

```
 Factor w/ 3 levels "1","2","3": 1 2 1 3 1 1 2 2 2 1
    ...
```

```
with(dat, table(grpfac1, grp))
```

```
        grp
grpfac1 1 2 3
      1 6 0 0
      2 0 7 0
      3 0 0 6
```

- That's treating 1 as a character, "1", etc.

Assign More Meaningful Labels for the Levels

- Let's be very concrete about this:

```
dat$grpfac1 <- factor(dat$grp, labels = c("Number1",
    "Number2", "Number3"))
str(dat$grpfac1)
```

```
Factor w/ 3 levels "Number1","Number2",..: 1 2 1 3 1 1
    2 2 2 1 ...
```

```
with(dat, table(grp, grpfac1))
```

```
   grpfac1
grp Number1 Number2 Number3
  1       6       0       0
  2       0       7       0
  3       0       0       6
```

The factor function's levels argument RE-ORDERS the input!

- Common mistake, to mis-understand difference between `levels()` function and levels argument in `factor()` function.
- In `factor`, the levels argument indicates which existing scores are to be included, *and in which order*.

```
dat$grpfaco <- factor(dat$grp, levels = c("2","1","3"
    ), labels = c("Number2", "Number1", "Number3"))
str(dat$grpfaco)
```

```
Factor w/ 3 levels "Number2","Number1",..: 2 1 2 3 2 2
    1 1 1 2 ...
```

- Note that the labels were re-arranged accordingly.

```
with(dat, table(grp, grpfaco))
```

The factor function's levels argument RE-ORDERS the input! ...

```
   grpfaco
grp  Number2  Number1  Number3
  1        0        6        0
  2        7        0        0
  3        0        0        6
```

- Now inside grpfaco, the internal numbering of the scores is changed. The labes are correct:

```
head(dat[   , c("grp", "grpfaco", "grpfac1")])
```

```
  grp  grpfaco  grpfac1
1   1 Number1  Number1
2   2 Number2  Number2
3   1 Number1  Number1
4   3 Number3  Number3
5   1 Number1  Number1
6   1 Number1  Number1
```

The factor function's levels argument RE-ORDERS the input! ...

- But the internal numeric scores have changed

```
  rbind(grp = dat$grp, grpfaco = as.numeric(dat$grpfaco
      ))[ ,1:6]
```

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

- The ordering can be important. Statistical procedures will generally us the first one as the baseline and provide estimates of the other levels as "contrasts". The way procedures handle factors is controlled by the session `options()`.

Collapse a numeric range into a Factor

- Sometimes researchers want to convert temperature scores from numeric to c("cold","warm","hot") or such.
- R provides a function called `cut()` that is intended for that purpose.
- The user must supply `breaks` so that the scores are subdivided.
- Labels for the levels of the new factor will be supplied automatically, but many users will not like them.

Let's convert noise into a new factor

- Recall dat$moreNoise

```
quantile(dat$moreNoise)
```

```
        0%        25%        50%        75%       100%
-1.6620502 -0.4028670  0.4911883  1.1338291  2.1968335
```

- Let's create 5 groupings

```
dat$mn1 <- cut(dat$moreNoise, breaks = c(-10, -1,
    -0.3, 0.7, 1, 10))
table(dat$mn1)
```

```
(-10,-1] (-1,-0.3] (-0.3,0.7]   (0.7,1]    (1,10]
       3         3         6         2         5
```

```
levels(dat$mn1)
```

```
[1] "(-10,-1]"   "(-1,-0.3]"  "(-0.3,0.7]" "(0.7,1]"
      "(1,10]"
```

Let's convert noise into a new factor ...

- Because the labels are so ugly, many people will change them either
  - at the time of creation

```
dat$mn1 <- cut(dat$moreNoise, breaks = c(-10, 1,
    -0.3, 0.7, 1, 10),
labels = c("never", "seldom", "some", "freq", "
    often"))
```

  - after creation

```
levels$mn1 <- c("never", "seldom", "some", "freq"
    , "often")
```

Brief Exercise: Run these commands (chunk exercise10 in R file)

```
x <- c("Y","N","Y","Y","F","N")
is.factor(x)
is.character(x)
xf1 <- factor(x)
xf1
levels(xf1)
x[1] <- "P"
xf1[1] <- "P"
xf1[1] <- "F"
xf1levs <- levels(xf1)
xf1[1] <-  xf1levs[2]
xf2 <- factor(x, levels = c("Y","N","anything"), labels =
     c("HECK","YES","irrelevant"))
table(x, xf2, exclude = NULL)
xf2[1] <- "Y"
levels(xf2)
xf2[1] <- "HECK"
xf2
```

Create a character variable and convert it to a factor

- Begin with a character vector

```
x <- c("Y","N","Y","Y","F","N")
is.factor(x)
```

```
[1] FALSE
```

```
is.character(x)
```

```
[1] TRUE
```

- Turn that into a factor, using defaults

```
xf1 <- factor(x)
xf1
```

```
[1] Y N Y Y F N
Levels: F N Y
```

```
levels(xf1)
```

```
[1] "F" "N" "Y"
```

Notice What R No Longer Allows

- x will still let us write anything we want

```
x[1] <- "P"
```

- But xf1 will refuse any assignment that is not a valid level.
- Why? xf1 is not an ordinary character vector anymore. It has levels that must be used for values.

```
levels(xf1)
```

```
[1] "F" "N" "Y"
```

- Try to set xf1 to a value that is not in levels(xf1)

```
xf1[1] <- "P"
```

```
Warning message:
In `[<-.factor`(`*tmp*`, 1, value = "P") :
  invalid factor level, NAs generated
```

Assigning Values To Factors

- Assign new value either with properly quoted, legal label, as in

```
xf1[1] <- "F"
```

- If the level were longer, or had spaces or other details that might cause danger of typographical errors, it is better to review the levels and then take the one you want. Examples:

```
xf1levs <- levels(xf1)
xf1[1] <-  xf1levs[2]
```

- Or in one step, put the value of xf1[1] to level 2

```
xf1[1] <- levels(xf1[1])[2]
```

- That is better because it avoids danger of typographical errors in long labels

The "drop unused levels" controversy

- "levels" Orders the new vactor using the pre-existing variable.
- "labels" Supplies new labels of this new variable

```
xf2 <- factor(x, levels=c("Y","N","anything"), labels
    =c("HECK","YES","irrelevant"))
table(x, xf2, exclude = NULL)
```

```
      xf2
x      HECK YES irrelevant <NA>
  F       0   0          0    1
  N       0   2          0    0
  P       0   0          0    1
  Y       2   0          0    0
  <NA>    0   0          0    0
```

- The levels argument includes an "unobserved" level.
- If we run the factor through factor(), it will "drop unused levels".

The "drop unused levels" controversy ...

```
xf2 <- factor(xf2)
table(x, xf2, exclude = NULL)
```

```
      xf2
x      HECK YES <NA>
  F       0   0    1
  N       0   2    0
  P       0   0    1
  Y       2   0    0
  <NA>    0   0    0
```

Values listed for xf2 only include "HECK" and "YES", the levels that are obseved in xf2.

- This fails

```
xf2[1] <- "anything"
```

```
Warning message:
In `[<-.factor`(`*tmp*`, 1, value = "anything") :
invalid factor level, NAs generated
```

The "drop unused levels" controversy ...

- Better check the valid levels

```
levels(xf2)
```

```
[1] "HECK" "YES"
```

- But this is OK

```
xf2[1] <- "HECK"
```

- See?

```
xf2
```

```
[1] HECK YES  HECK HECK <NA> YES
Levels: HECK YES
```

Add New Values: Requires a "fiddle" with Levels

- Copy xf2 to xf3, then append a new level "Denver"

```
xf3 <- xf2
levels(xf3) <- c(levels(xf3), "Denver")
xf3[5] <- "Denver"
data.frame(x, xf1, xf2, xf3)
```

```
  x xf1   xf2      xf3
1 P    N HECK    HECK
2 N    N  YES     YES
3 Y    Y HECK    HECK
4 Y    Y HECK    HECK
5 F    F <NA> Denver
6 N    N  YES     YES
```

Practice this: Choose Your Names. Choose Your Order

- `factor()` levels and labels must match.

```
dat$grpfac2 <- factor(dat$grp, levels = c(2,1,3),
    labels = c("Western","Midwest","Eastern"))
str(dat$grpfac2)
```

```
Factor w/ 3 levels "Western","Midwest",..: 2 1 2 3 2 2
    1 1 1 2 ...
```

```
with(dat, table(grpfac2, grp))
```

```
          grp
grpfac2    1 2 3
  Western  0 7 0
  Midwest  6 0 0
  Eastern  0 0 6
```

Now, suppose you want to re-order the levels

- Everybody has made this mistake at least once (Maybe twice): It is VERY tempting to do something like

```
levels(dat$grpfac2) <- c("Midwest", "Eastern", "
    Western")
```

- Caution. Disaster occurred!

```
dat$grpfac3 <- dat$grpfac2
levels(dat$grpfac3) <- c("Midwest","Western","Eastern
    ")
str(dat$grpfac3)
```

```
Factor w/ 3 levels "Midwest","Western",..: 2 1 2 3 2 2
    1 1 1 2 ...
```

```
with(dat, table(grpfac3, grpfac2))
```

Now, suppose you want to re-order the levels ...

```
          grpfac2
grpfac3    Western Midwest Eastern
  Midwest        7       0       0
  Western        0       6       0
  Eastern        0       0       6
```

- Big point: "`levels()`" does not "reorganize" the information. It just changes the labels of the current order.

Why would you want to use `levels()`?

- I use levels *ALMOST EXCLUSIVELY* to review existing variables (not to set new levels)

```
levels(dat$grpfac3)
```

  is perfectly safe

- Putting an argument on the right hand side can be tricky. Do that in order to
  - To rename (respell) same level in the same order
  - Perhaps you want shorter strings. This is automatic

```
shortLabels <- abbreviate( levels(dat$grpfac3),
minlength= 1)
levels(dat$grpfac3) <- shortLabels
```

  - Same to do it manually, but perhaps more error prone because we might type W, M and E "out of order"

```
levels(dat$grpfac3) <- c("M","W","E")
```

You Can Reorganize Factor Variables, However ...

- At create time, use the levels argument.

```
newFactor <- levels(oldFactor, levels=c("C", "B", "A"
    ), labels=c("car", "bus", "auto"))
```

- Suppose the current levels of grpfac2 are

```
levels(dat$grpfac2)
```

```
[1] "Western" "Midwest" "Eastern"
```

- create grpfac4 by re-ordering

```
dat$grpfac4 <- with(dat, factor(grpfac2, levels=c("
    Eastern","Western","Midwest")))
with(dat, table(grpfac4, grpfac2))
```

```
          grpfac2
grpfac4    Western Midwest Eastern
  Eastern        0       0       6
  Western        7       0       0
  Midwest        0       6       0
```

- Effect: Changes the way results are reported (plots, regression)

relevel function is a convenience function

- For unordered factors,"relevel()" can be used to properly re-sort a variable so that one value "comes first"

```
dat$grpfac5 <- with(dat, relevel(grpfac2, ref="
    Eastern"))
with(dat, table(grpfac5, grpfac2))
```

```
          grpfac2
grpfac5    Western Midwest Eastern
  Eastern        0       0       6
  Western        7       0       0
  Midwest        0       6       0
```

- Has very limited effect of moving one value to the front of the levels
- Effect: Changes regression tables

grpfac2 has "Western" as the Reference Category

```
coef(summary(lm(newed ~ grpfac2, data = dat)))[ ,1:2]
```

```
                Estimate Std. Error
(Intercept)    10.428571   1.449695
grpfac2Midwest  1.738095   2.133893
grpfac2Eastern  4.404762   2.133893
```

grpfac4 has "Eastern" as the Reference Category

```
coef(summary(lm(newed ~ grpfac4, data = dat)))[,1:2]
```

```
                Estimate Std. Error
(Intercept)    14.833333   1.565850
grpfac4Western -4.404762   2.133893
grpfac4Midwest -2.666667   2.214446
```

The Problem of "combining" levels

- Suppose you have a factor variable with 3 levels

```
x <- c("A","B","C","B","C")
```

- However, "C" is a redundant scoring. It is really same as B.
- We want to put "C" cases into "B". The "obvious approach" fails.

```
f <- factor(x, levels = c("A","B","C"), labels = c("A
    ","B","B"))
```

```
Warning message:
In `levels<-`(`*tmp*`, value = if (nl == nL)
    as.character(labels) else paste0(labels, :
    duplicated levels will not be allowed in factors
    anymore
```

- Its necessary to "create" a new level, then recode to use it (seems tedious).

The Problem of "combining" levels ...

```
levels(x) <- c(levels(x), "BorC")
x[ x %in% c("B", "C")] <- "BorC"
x <- factor(x)
table(x)
```

```
x
   A BorC
   1    4
```

- The use of factor(x) causes the old, unused levels "B" and "C" to be removed.

```
levels(x)
```

```
[1] "A"    "BorC"
```

```
levels(x) <- c("A","B")
table(x)
```

The Problem of "combining" levels ...

```
x
A B
1 4
```

- Package rockchalk has a function called `combineLevels()` that is intended to automate this. Example usage

```
x <- factor(c("A","B","C","B","C","A"))
xrc <- rockchalk::combineLevels(x, c("B","C"), "BorC"
    )
```

```
The original levels A B C
have been replaced by A BorC
```

```
table(xrc, x)
```

```
      x
xrc    A B C
  A    2 0 0
  BorC 0 2 2
```

R has its Own Data Storage Formats

- R's `save()` and `load()`
- Correct suffixes: "RData" and "rda". NOT "Rdata" (as I often do mistakenly)
- Objects saved in this way are compressed
- Are compatible across platforms: Can email from Mac user to Linux user and R can load "as if" it were created there.

Try this magic trick

- Suppose "dat" is a data frame

```
save(dat, file="practiceData.RData")
```

- Remove the dat object from memory

```
rm(dat)
```

- Get it back

```
load("practiceData.RData")
str(dat)
```

Lately I prefer RDS format (`saveRDS()`)

- Shortcoming of `load()`: the collection of saved objects is restored into memory and existing objects with same names are *obliterated!*
- See ?saveRDS, which describes a way to save a single R object, along with the `readRDS()` that can restore an object and re-name it in the process.
- File name suffix standard is "rds".

```
saveRDS(dat, "practiceData.rds")
dat99 <- readRDS("practiceData.rds")
str(dat99)
```

Lately I prefer RDS format (saveRDS()) ...

```
'data.frame': 19 obs. of  18 variables:
 $ grp       : int  1 2 1 3 1 1 2 2 2 1 ...
 $ inc       : int  44444 34343 11112 23232 23111 78787
     33233 22312 32322 76755 ...
 $ mar       : Factor w/ 3 levels "N","W","Y": 3 3 1 1 3
     1 3 1 3 1 ...
 $ sex       : Factor w/ 2 levels "Fem","Mal": 2 1 2 1 1
     2 2 1 1 1 ...
 $ noise     : num  NA NA 418 338 NA ...
 $ moreNoise: num  0.299 0.78 1.456 -0.644 -1.553 ...
 $ newed     : int  11 10 9 15 7 12 8 6 11 20 ...
 $ ed        : int  11 10 9 15 7 12 8 6 11 20 ...
 $ edlog     : num  2.48 2.4 2.3 2.77 2.08 ...
 $ edsqrt    : num  3.32 3.16 3 3.87 2.65 ...
 $ edexp     : num  59874 22026 8103 3269017 1097 ...
 $ grpfac1   : Factor w/ 3 levels "Number1","Number2",..
     : 1 2 1 3 1 1 2 2 2 1 ...
 $ grpfaco   : Factor w/ 3 levels "Number2","Number1",..
     : 2 1 2 3 2 2 1 1 1 2 ...
 $ mn1       : Factor w/ 5 levels "(-10,-1]","(-1,-0.3]"
     ,..: 3 4 5 2 1 1 5 2 3 3 ...
```

Lately I prefer RDS format (saveRDS()) ...

```
$ grpfac2   : Factor w/ 3 levels "Western","Midwest",..
    : 2 1 2 3 2 2 1 1 1 2 ...
$ grpfac3   : Factor w/ 3 levels "Midwest","Western",..
    : 2 1 2 3 2 2 1 1 1 2 ...
$ grpfac4   : Factor w/ 3 levels "Eastern","Western",..
    : 3 2 3 1 3 3 2 2 2 3 ...
$ grpfac5   : Factor w/ 3 levels "Eastern","Western",..
    : 3 2 3 1 3 3 2 2 2 3 ...
```

```
identical(dat, dat99)
```

```
[1] TRUE
```