STAT 545A
Class meeting #4
Monday, September 17, 2012

Dr. Jennifer (Jenny) Bryan

Department of Statistics and Michael Smith Laboratories

Where you can find STAT 545A stuff on the web:

#0: The STAT545A subpage on my website:
http://www.stat.ubc.ca/~jenny/teach/STAT545/index.html
This is more of a placeholder / advertisement. Won't be changing much. Won't hold valuable content.

#1: Our collaborative course webspace:
http://www.bryanlab.msl.ubc.ca/stat545a2012/
will host student work, lecture slides, etc.

#2: In a special directory within my Stat website:
http://www.stat.ubc.ca/~jenny/notOcto/STAT545A/ will hold serious business, like well-organized R projects full of code, figures, etc., where I cannot tolerate the annoying interface of the above system.

Review of last class

Basic data checking of categorical variables, both actual factors and an integer-valued variable like year

table() is good, often nice to couple with barchart() or dotplot()

Simple but useful view of simple R objects: character, logical, numeric, or factor

R objects have a mode and a class

Factors are special.

Review of last class, cont'd

Vectors (and matrices and arrays) are at the heart of R. Many computations can and should be "vectorized" (not really explained/demo'd yet).

Most common "data collection" R objects: vector, matrix, array, data.frame, list

attach() is evil. Keep your data safely tucked into a data.frame and pass it to graphing and modelling functions.

Embrace Names.

Focus of next couple of classes

Data checking, cleaning, and exploration of single variables, categorical and quantitative

Data exploration of 2 variables at a time

Care and feeding of R objects

Data aggregation, i.e. doing a repetitive activity on many different subsets of the data. How and why to accomplish in R without loops.

Anatomy of a real world data analysis, so far:

```
/Users/jenny/teaching/STAT545A/examples/gapminder/code:
total used in directory 288 available 278879212
drwxr-xr-x  25 jenny  staff    850 Sep 11 22:24 .
drwxr-xr-x   7 jenny  staff    238 Mar 31  2011 ..
-rw-r--r--@  1 jenny  staff   6148 Sep 11 22:19 .DS_Store
-rw-r--r--   1 jenny  staff   2583 Sep 11 22:19 .Rhistory
-rw-r--r--   1 jenny  staff   4807 Sep 11 13:24 bryan-a01-01-dataPrep.R
-rw-r--r--   1 jenny  staff   6349 Sep 11 13:33 bryan-a01-02-dataMerge.R
-rw-r--r--   1 jenny  staff   5783 Sep 11 14:38 bryan-a01-03-dataExplore.R
-rw-r--r--   1 jenny  staff   3497 Sep 11 22:11 bryan-a01-04-fillContinentData.R
-rw-r--r--   1 jenny  staff   4573 Sep 11 22:24 bryan-a01-05-everyFiveYears.R
```

Last line of bryan-a01-05-everyFiveYears.R writes the “cleaned” data to file:

```
write.table(gDat,
            paste0(whereAmI, "data/gapminderDataFiveYear.txt"),
            quote = FALSE, sep = "\t", row.names = FALSE)
```

From now on, this is what ‘gDat’ holds ... data for years 1952, 1957, ... w/ continent filled in.

Anatomy of a real world data analysis, so far:

```
/Users/jenny/teaching/STAT545A/examples/gapminder/code:
total used in directory 288 available 278879212
drwxr-xr-x  25 jenny  staff    850 Sep 11 22:24 .
drwxr-xr-x   7 jenny  staff    238 Mar 31  2011 ..
-rw-r--r--@  1 jenny  staff   6148 Sep 11 22:19 .DS_Store
-rw-r--r--   1 jenny  staff   2583 Sep 11 22:19 .Rhistory
-rw-r--r--   1 jenny  staff   4807 Sep 11 13:24 bryan-a01-01-dataPrep.R
-rw-r--r--   1 jenny  staff   6349 Sep 11 13:33 bryan-a01-02-dataMerge.R
-rw-r--r--   1 jenny  staff   5783 Sep 11 14:38 bryan-a01-03-dataExplore.R
-rw-r--r--   1 jenny  staff   3497 Sep 11 22:11 bryan-a01-04-fillContinentData.R
-rw-r--r--   1 jenny  staff   4573 Sep 11 22:24 bryan-a01-05-everyFiveYears.R
```

Addressing data deficiencies. Actually cleaning the data and creating a beautiful data file to begin the serious graphing work.

Read at your leisure. Will not discuss in class.

From now on, I will be using the cleaned Gapminder data.

```
> gDat <- read.delim(paste0(whereAmI,"data/gapminderDataFiveYear.txt"))

> str(gDat)
'data.frame':	1704 obs. of  6 variables:
 $ country  : Factor w/ 142 levels "Afghanistan",..: 1 1 1 1 1 1 1 1 1 1 ...
 $ year     : int  1952 1957 1962 1967 1972 1977 1982 1987 1992 1997 ...
 $ pop      : num  8425333 9240934 10267083 11537966 13079460 ...
 $ continent: Factor w/ 5 levels "Africa","Americas",..: 3 3 3 3 3 3 3 3 3 3 ...
 $ lifeExp  : num  28.8 30.3 32 34 36.1 ...
 $ gdpPercap: num  779 821 853 836 740 ...
```

Focusing on the R ways to address collections of data: vectors/arrays, lists, data.frames

... picking up where we left off ....

"indexing"

"pulling out specific bits of your data for inspection, modification, use in a figure, use in a model, etc."

Subscripting vectors

In most contexts, you can subscript in many ways.
The most common/useful:

- a logical vector
- a vector of positive (or negative!) integers
- a vector of character strings

```
> x <- rnorm(6)
> names(x) <- letters[seq_along(x)]
> round(x, 2)
    a     b     c     d     e     f 
 0.51  0.32  0.28  1.40 -0.89 -1.94 
> x[x < 0]
         e          f 
-0.8889749 -1.9428406 
> x[seq(from = 1, to = length(x), by = 2)]
         a          c          e 
 0.5092672  0.2750631 -0.8889749 
> x[-c(2, 5)]
         a          c          d          f 
 0.5092672  0.2750631  1.3958511 -1.9428406 
> x[c('c', 'a', 'f')]
         c          a          f 
 0.2750631  0.5092672 -1.9428406 
```

read the documentation
for seq() and friends, rep()

Subscripting matrices

Requires two indices, e.g. x[i, j]*

But all of the previous options are still open

- a logical vector
- a vector of positive (or negative!) integers
- a vector of character strings

```
> jMat <- outer(as.character(1:4), as.character(1:4),
+               function(x, y) {
+                 paste('x', x, y, sep = "")
+               })

> jMat
     [,1]  [,2]  [,3]  [,4]
[1,] "x11" "x12" "x13" "x14"
[2,] "x21" "x22" "x23" "x24"
[3,] "x31" "x32" "x33" "x34"
[4,] "x41" "x42" "x43" "x44"
```

I'll demo with this.

* technically not true, but that's usually what you want

Functions for getting to know a matrix

```
> jMat
     [,1]  [,2]  [,3]  [,4] 
[1,] "x11" "x12" "x13" "x14"
[2,] "x21" "x22" "x23" "x24"
[3,] "x31" "x32" "x33" "x34"
[4,] "x41" "x42" "x43" "x44"

> str(jMat)
 chr [1:4, 1:4] "x11" "x21" "x31" "x41" "x12" "x22" "x32" ...

> class(jMat)
[1] "matrix"

> mode(jMat)
[1] "character"

> dim(jMat)
[1] 4 4
```

```
> jMat
     [,1]  [,2]  [,3]  [,4] 
[1,] "x11" "x12" "x13" "x14"
[2,] "x21" "x22" "x23" "x24"
[3,] "x31" "x32" "x33" "x34"
[4,] "x41" "x42" "x43" "x44"

> jMat[2, 3]
[1] "x23"

> jMat[7]                              # works! double-edged sword
[1] "x32"

> jMat[2, ]                            # one row
[1] "x21" "x22" "x23" "x24"

> jMat[, 3]                            # one column
[1] "x13" "x23" "x33" "x43"
```

Square brackets to subscript/subset a matrix -- consider ‘drop’

```
> jMat[2, ]
[1] "x21" "x22" "x23" "x24"

> dim(jMat[2, ])
NULL

> is.matrix(jMat[2, ])
[1] FALSE

> is.vector(jMat[2, ])
[1] TRUE
```

```
> jMat[2, , drop = FALSE]
     [,1]  [,2]  [,3]  [,4]
[1,] "x21" "x22" "x23" "x24"

> dim(jMat[2, , drop = FALSE])
[1] 1 4

> is.matrix(jMat[2, , drop = FALSE])
[1] TRUE

> is.vector(jMat[2, , drop = FALSE])
[1] FALSE
```

```
> jMat[, 3]                                  # one col
[1] "x13" "x23" "x33" "x43"
> jMat[, 3, drop = FALSE]                    # same story here
     [,1]
[1,] "x13"
[2,] "x23"
[3,] "x33"
[4,] "x43"
```

Be aware of matrix w/ 1 row or col vs. a vector!

How to query and change row and column names

```
> rownames(jMat)
NULL
> colnames(jMat)
NULL
> rownames(jMat) <- paste0("row", c("One","Two","Three","Four"))
> colnames(jMat) <- c("carrot", "cabbage", "grape", "banana")
> jMat
         carrot cabbage grape banana
rowOne   "x11"  "x12"   "x13" "x14"
rowTwo   "x21"  "x22"   "x23" "x24"
rowThree "x31"  "x32"   "x33" "x34"
rowFour  "x41"  "x42"   "x43" "x44"
> dimnames(jMat)
[[1]]
[1] "rowOne"   "rowTwo"   "rowThree" "rowFour"

[[2]]
[1] "carrot"  "cabbage" "grape"   "banana"

> dimnames(jMat) <- NULL
> dimnames(jMat)
NULL
> dimnames(jMat) <- list(paste0("row", c("One","Two","Three","Four")),
+                        c("carrot", "cabbage", "grape", "banana"))
> jMat
         carrot cabbage grape banana
rowOne   "x11"  "x12"   "x13" "x14"
rowTwo   "x21"  "x22"   "x23" "x24"
rowThree "x31"  "x32"   "x33" "x34"
rowFour  "x41"  "x42"   "x43" "x44"
```

Further proof that you can index using all sorts of different things ....

```
> jMat
          carrot cabbage grape banana
rowOne    "x11"  "x12"   "x13" "x14"
rowTwo    "x21"  "x22"   "x23" "x24"
rowThree "x31"  "x32"   "x33" "x34"
rowFour   "x41"  "x42"   "x43" "x44"

> jMat[c("rowOne", "rowThree"), c("carrot", "banana")]
          carrot banana
rowOne    "x11"  "x14"
rowThree "x31"  "x34"
```

character strings; here re: row or column names

```
> jMat[-c(2, 3), c(TRUE, TRUE, FALSE, FALSE)]
        carrot cabbage
rowOne  "x11"  "x12"
rowFour "x41"  "x42"
```

(negative!) integers, logical vectors

```
> jMat[1, grepl("r[ao]", colnames(jMat))]
carrot  grape
 "x11"  "x13"
```

integers, logical vectors arising from regular expression testing

Also be aware that indexing can be done on the left-hand side of an assignment to replace those values

```
> jMat["rowThree", 2:3] <-
+   c("HEY!", "THIS IS NUTS!")

> jMat
         carrot cabbage grape           banana
rowOne   "x11"  "x12"   "x13"           "x14"
rowTwo   "x21"  "x22"   "x23"           "x24"
rowThree "x31"  "HEY!"  "THIS IS NUTS!" "x34"
rowFour  "x41"  "x42"   "x43"           "x44"
```

now ... data.frames

gDat$year

Use the dollar sign to extract one variable by name (works for lists in general, not just data.frames)

```
> gDat$year
   [1] 1952 1957 1962 1967 1972 1977 1982 1987 1992 1997 2002 2007 1952 1957

   <snip, snip>

[1681] 1952 1957 1962 1967 1972 1977 1982 1987 1992 1997 2002 2007 1952 1957
[1695] 1962 1967 1972 1977 1982 1987 1992 1997 2002 2007

> str(gDat$year)
 int [1:1704] 1952 1957 1962 1967 1972 1977 1982 1987 1992 1997 ...

> mode(gDat$year)
[1] "numeric"

> class(gDat$year)
[1] "integer"

> is.vector(gDat$year)
[1] TRUE

> is.data.frame(gDat$year)
[1] FALSE
```

How to extract certain rows and/or variables (columns) from a data.frame*

```
> cDat <- subset(gDat, subset = country == "Canada",
+                select = c(country, continent, lifeExp))

> str(cDat)
'data.frame':  12 obs. of  3 variables:
 $ country  : Factor w/ 142 levels "Afghanistan",..: 21 21 21 21 21 21 21 21
2..
 $ continent: Factor w/ 5 levels "Africa","Americas",..: 2 2 2 2 2 2 2 2 2
2 ...
 $ lifeExp  : num  68.8 70 71.3 72.1 72.9 ...

> mode(cDat)
[1] "list"

> class(cDat)
[1] "data.frame"

> is.vector(cDat)
[1] FALSE

> is.data.frame(cDat)
[1] TRUE
```

* subset() also works for vectors and matrices, but it is especially important for data.frames

```
> cDat <- subset(gDat, subset = country == "Canada",
+                  select = c(country, continent, lifeExp))
```

```
subset                    package:base                    R Documentation

Subsetting Vectors, Matrices and Data Frames

Description:

     Return subsets of vectors, matrices or data frames which meet
     conditions.

Usage:

     <snip, snip>

     ## S3 method for class 'data.frame'
     subset(x, subset, select, drop = FALSE, ...)

Arguments:

       x: object to be subsetted.

  subset: logical expression indicating elements or rows to keep:
          missing values are taken as false.

  select: expression, indicating columns to select from a data frame.
```

```
> cDat <- subset(gDat, subset = country == "Canada",
+                select = c(country, continent, lifeExp))
```

```
subset                    package:base                    R Documentation

Subsetting Vectors, Matrices and Data Frames

Description:

     Return subsets of vectors, matrices or data frames which meet
     conditions.

Usage:

     <snip, snip>

     ## S3 method for class 'data.frame'
     subset(x, subset, select, drop = FALSE, ...)

Arguments:

       x: object to be subsetted.

  subset: logical expression indicating elements or rows to keep:
          missing values are taken as false.

  select: expression, indicating columns to select from a data frame.
```

```
subset                  package:base                  R Documentation

Subsetting Vectors, Matrices and Data Frames

<snip, snip>

Details:

     This is a generic function, with methods supplied for matrices,
     data frames and vectors (including lists).  Packages and users can
     add further methods.

     For ordinary vectors, the result is simply 'x[subset &
     !is.na(subset)]'.

     For data frames, the 'subset' argument works on the rows.  Note
     that 'subset' will be evaluated in the data frame, so columns can
     be referred to (by name) as variables in the expression (see the
     examples).

     The 'select' argument exists only for the methods for data frames
     and matrices.  It works by first replacing column names in the
     selection expression with the corresponding column numbers in the
     data frame and then using the resulting integer vector to index
     the columns.  This allows the use of the standard indexing
     conventions so that for example ranges of columns can be specified
     easily, or single columns can be dropped (see the examples).
```

I really do encourage you to see the examples and notice how I use subset().

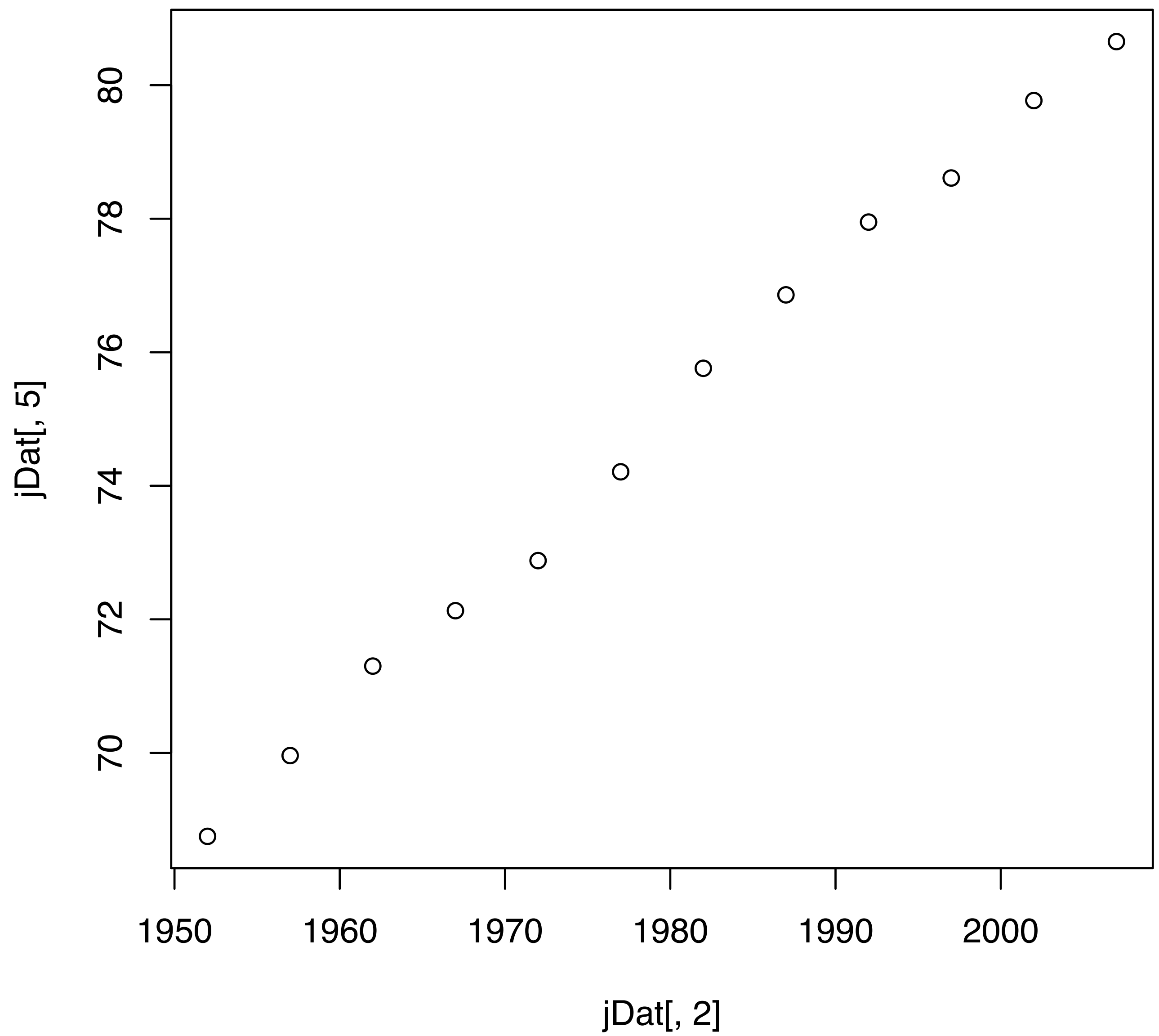
Just FYI, you can index a data.frame like you would a matrix:

```
> gDat[20:25, c("country", "lifeExp")]
   country lifeExp
20 Albania  72.000
21 Albania  71.581
22 Albania  72.950
23 Albania  75.651
24 Albania  76.423
25 Algeria  43.077
```

... but using subset() is often easier and leads to more readable, robust code

Bad practices in data manipulation/access .....

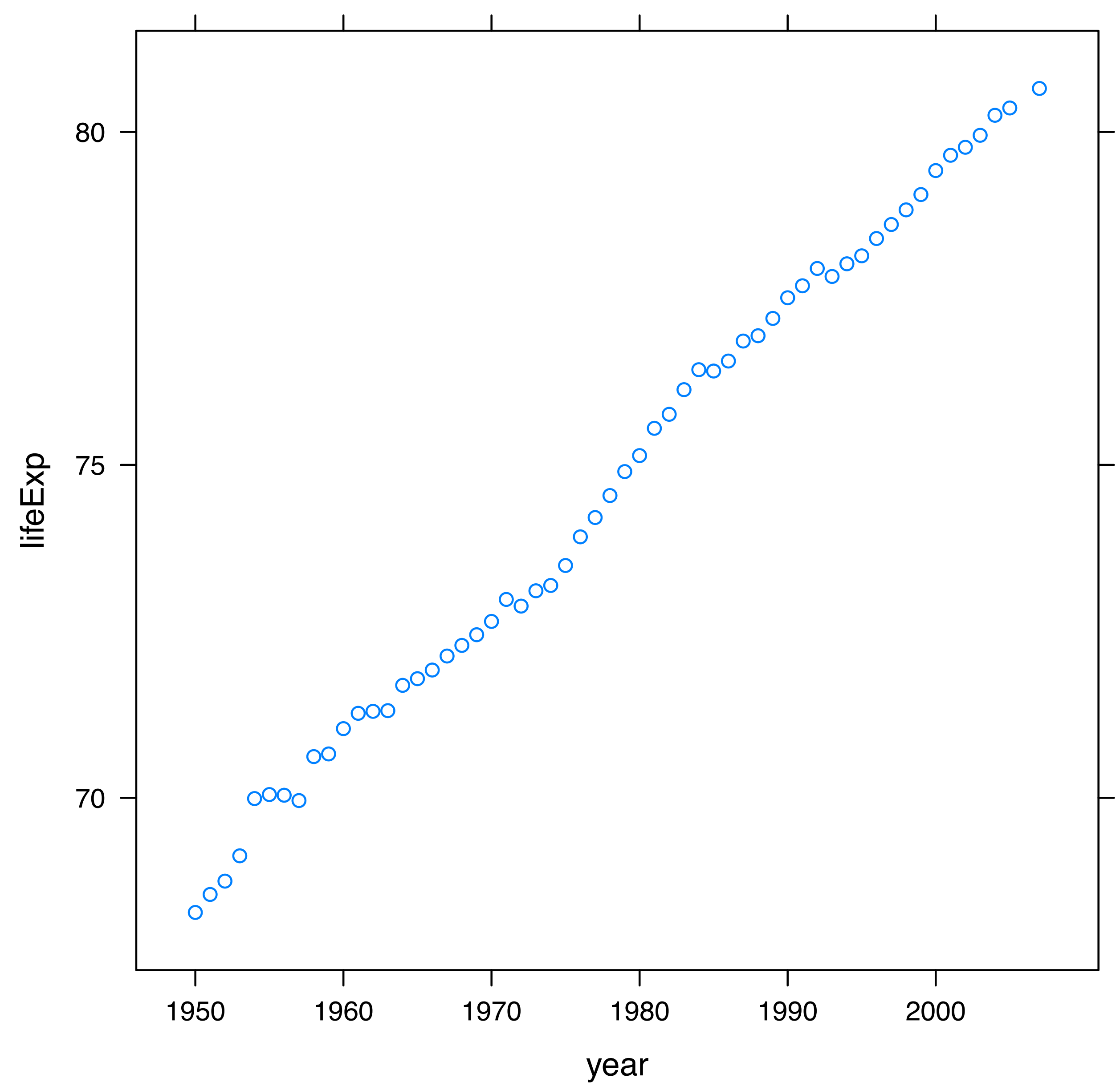
```
jDat <- subset(gDat, country == "Canada")

## don't refer to variables by number
## leads to poor figure labels and confusion

plot(jDat[,5] ~ jDat[, 2])                # BAD
```



bad = will ultimately lead to more mistakes, wasting your time

Bad practices in data manipulation/access .....

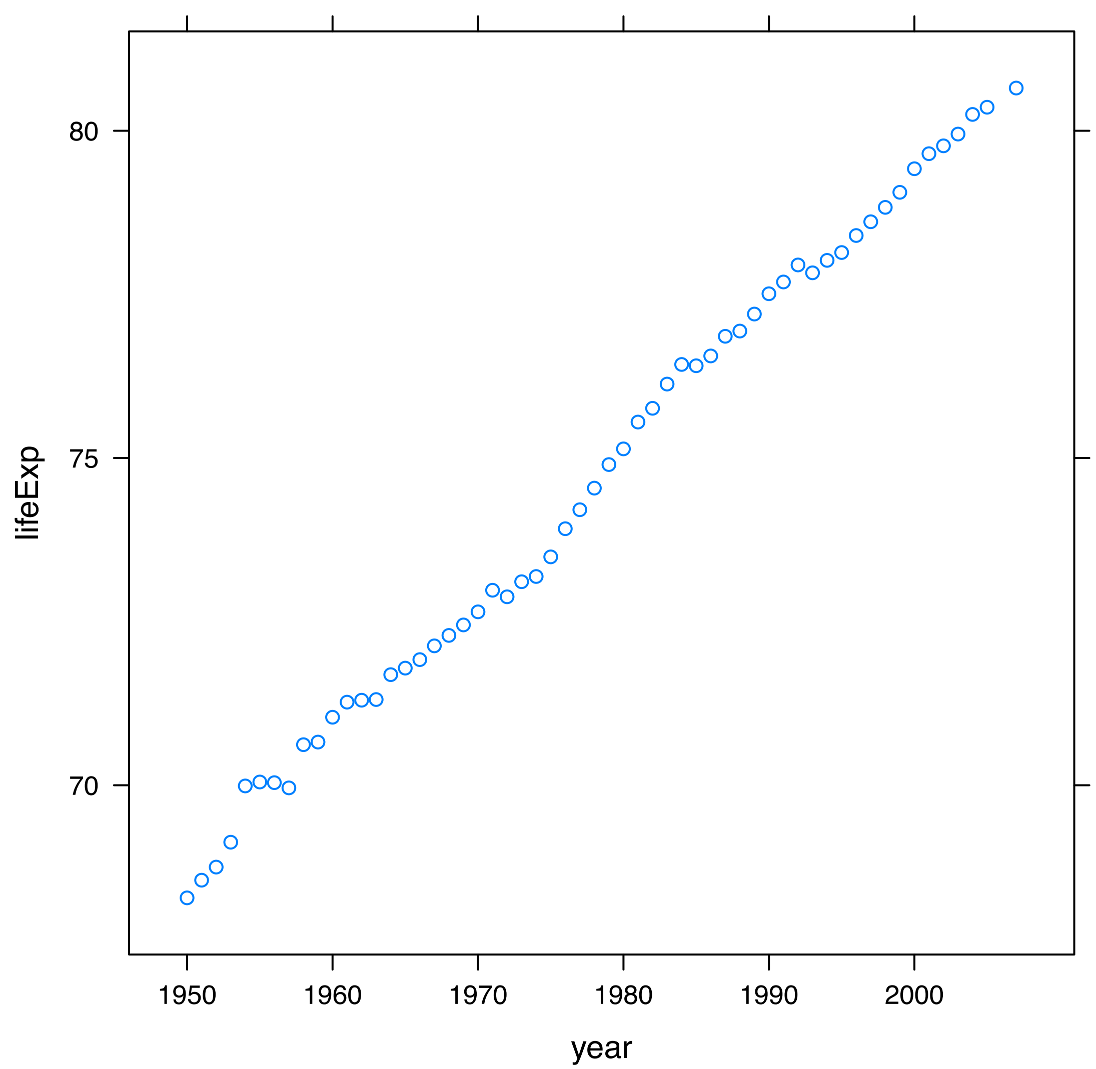
```
## use data.frames!
## don't create new, stand-alone copies of certain variables

jDat <- subset(gDat, country == "Canada")

year <- jDat[, 2]                          # BAD
lifeExp <- jDat[, 5]

xyplot(lifeExp ~ year)

rm(year,lifeExp)
```



bad = will ultimately lead to more mistakes, wasting your time

Bad practices in data manipulation/access .....

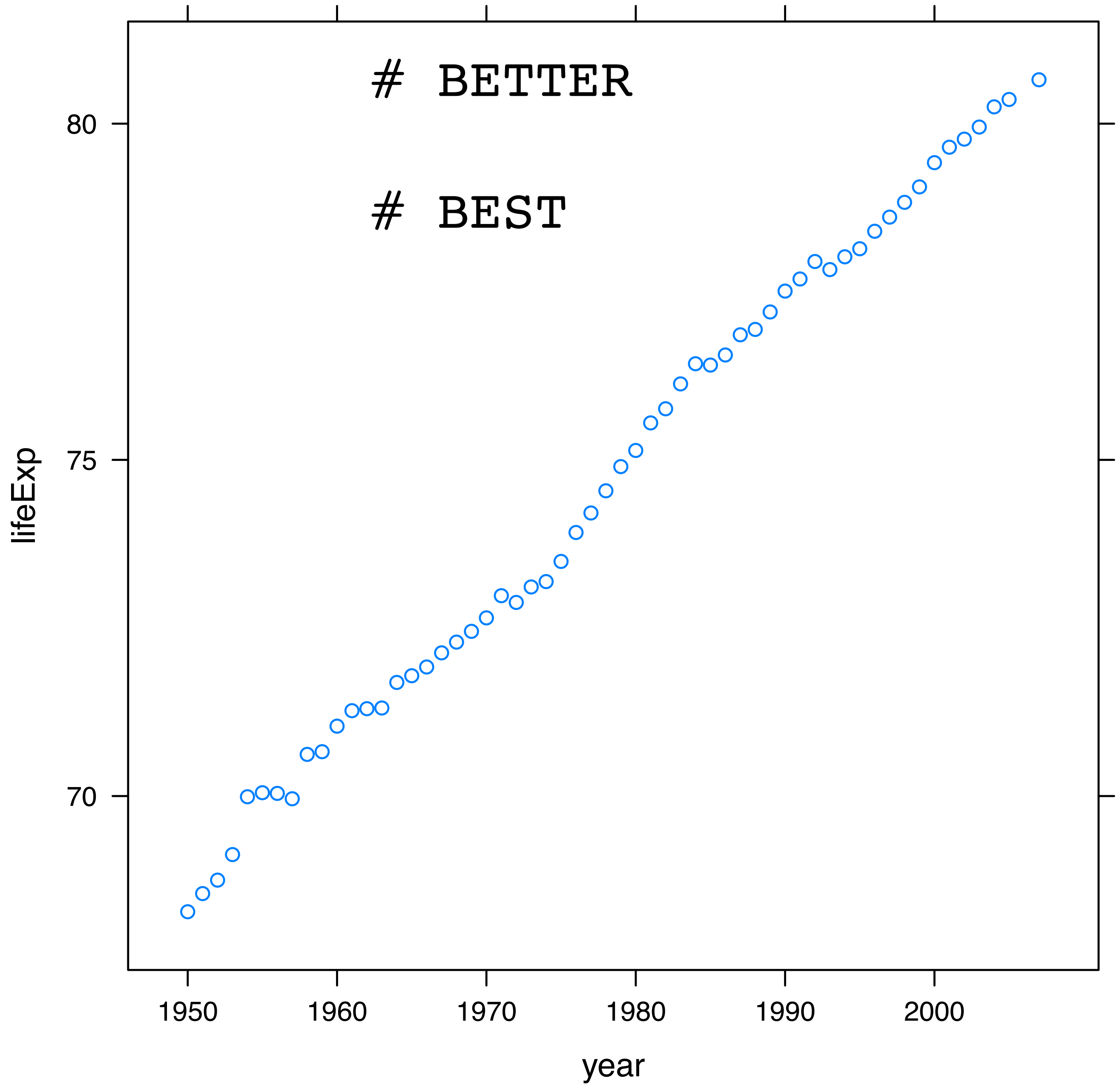
```
## use data.frames!
## don't attach data.frame

attach(gDat)

xyplot(lifeExp ~ year)                        # BAD

detach(gDat)
```



bad = will ultimately lead to more mistakes, wasting your time

Best practices in data manipulation/access .....

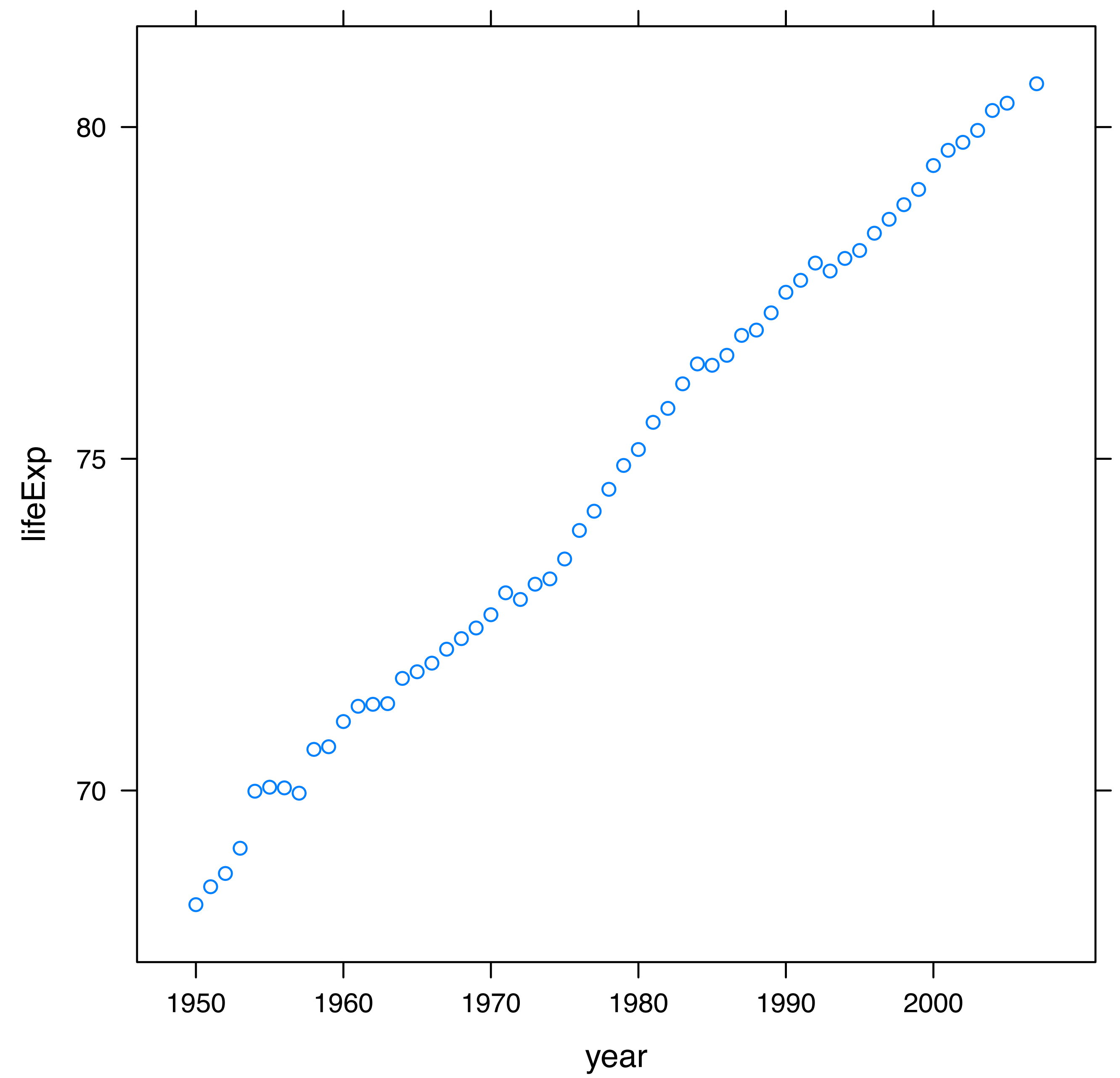
```
> ## give variables within data.frames short, informative names
> ## will make it easy to access variables by name

> xyplot(jDat$lifeExp ~ jDat$year)          # GOOD

> with(jDat,
+      xyplot(lifeExp ~ year))              # BETTER

> xyplot(lifeExp ~ year, jDat)              # BEST
```



good = more robust, self-documenting

Best practices in data manipulation/access .....

```
> ## if you don't need the data subset long-term,
> ## don't create a persistent copy! use the subset argument available
> ## in many functions to subset on the fly

> xyplot(lifeExp ~ year, gDat,
+         subset = country == "Canada")    # BESTEST
```



good = more robust, self-documenting, minimalistic

```
with(jDat,
     xyplot(lifeExp ~ year))*
```

with() is a handy function

can make it more pleasant to rigorously use data.frames and reference-by-name -- cuts down on the repetitive typing

good for weaning yourself off of ‘attach’-ing R objects

*Note: example is slightly silly, since xyplot() has a ‘data =’ argument, but you get the point.

```
> jDf <- data.frame(jMat, stringsAsFactors = FALSE)
> jDf
   X1  X2  X3  X4
1 x11 x12 x13 x14
2 x21 x22 x23 x24
3 x31 x32 x33 x34
4 x41 x42 x43 x44
```

preferred (?)

Extracting one variable from a data.frame:

data.frame style vs list style vs matrix style

If you extract a variable like so ...	... you will get this <object> str(<object>):	... you will get this (plain English):
jDf$X3 jDf[['X3']] jDf[, 'X3']	[1] "x13" "x23" "x33" "x43" chr [1:4] "x13" "x23" "x33" "x43"	the variable, as a vector here, a character vector of length 4
jDf['X3'] jDf[, 'X3', drop = FALSE] subset(jDf, select = X3)	X3 1 x13 2 x23 3 x33 4 x43 'data.frame': 4 obs. of 1 variable: $ X3: chr "x13" "x23" "x33" "x43"	data.frame with only this variable in it

Difference between $ and [[access of a single variable from a data.frame

```
jDf$X3

jDf[['X3']]
```

Both achieve the same thing here: extracting the component named X3 from the data.frame (i.e. list) jDf

Both methods -- $ and [[-- can extract *only one component*

```
> ## this will not work
> jDf[[c('X1','X2')]]
Error in .subset2(x, i, exact = exact) : subscript out of bounds
```

Main difference: If the name of the component you want is stored as on R object, you must use [[.

```
> (luckyVar <- sample(names(jDf), 1))
[1] "X2"
> jDf[[luckyVar]]
[1] "x12" "x22" "x32" "x42"
```

Extracting > 1 variables from a data.frame

```
> jDf <- data.frame(jMat, stringsAsFactors = FALSE)
> jDf
   X1  X2  X3  X4
1 x11 x12 x13 x14
2 x21 x22 x23 x24
3 x31 x32 x33 x34
4 x41 x42 x43 x44
```

Valid commands	Description	Comments
jDf[, c('X2','X3')]	matrix style subscripting	why do this?
jDf[c('X2','X3')]	vector style subscripting	good
subset(jDf, select = c(X2,X3))	using the subset() function	very good
colsToKeep <- grep('[2|3]$', names(jDf), value = TRUE)) # followed by either of these: subset(jDf, select = colsToKeep) jDf[colsToKeep]	example of programmatically generating the names, in this case, of the variables to keep similar idea also works if indexing by a vector of numbers of by a logical vector	very useful when the columns to keep can or must be derived

```
   X2  X3
1 x12 x13
2 x22 x23
3 x32 x33
4 x42 x43
```

```
'data.frame':    4 obs. of  2 variables:
 $ X2: chr  "x12" "x22" "x32" "x42"
 $ X3: chr  "x13" "x23" "x33" "x43"
```

```
> ## randomly permute the variables and ...
> ## still get X2 & X3
> jDfScrambled <- jDf[sample(length(jDf))]
> jDfScrambled
   X1  X4  X2  X3
1 x11 x14 x12 x13
2 x21 x24 x22 x23
3 x31 x34 x32 x33
4 x41 x44 x42 x43
> jDfScrambled[match(colsToKeep,
                     names(jDfScrambled))]

   X2  X3
1 x12 x13
2 x22 x23
3 x32 x33
4 x42 x43
```

match(), %in%, grep(), grepl(), which() are all useful for subscripting

```
> jDf[names(jDf) %in% c('X1','X4')]
   X1  X4
1 x11 x14
2 x21 x24
3 x31 x34
4 x41 x44

> jDfPlus <- data.frame(jDf,
+                       Y1 = rnorm(nrow(jDf)),
+                       Y2 = rexp(nrow(jDf)))

> jDfPlus
   X1  X2  X3  X4          Y1         Y2
1 x11 x12 x13 x14 -0.43312107 0.31115650
2 x21 x22 x23 x24  0.80392540 0.07896557
3 x31 x32 x33 x34  0.02549102 0.89839139
4 x41 x42 x43 x44 -1.29231415 0.62356150

> jDfPlus[grepl('^Y', names(jDfPlus))]
           Y1         Y2
1 -0.43312107 0.31115650
2  0.80392540 0.07896557
3  0.02549102 0.89839139
4 -1.29231415 0.62356150
```

Helpful function round-up

Inspecting

str()   summary()
mode()   class()
methods()   head()   tail()
peek()*

Creating

c()   factor()   matrix()   array()
list()   data.frame()   read.table()

Sizing

length(), nrow(),
ncol(), dim()

Testing & converting

The is / as family: is.numeric()
is.character() is.vector()
as.matrix()   as.numeric()   etc etc

unlist()   unclass()

*JB function

Naming & inspecting names

names() dimnames()
row.names() rownames()
colnames()

Fussing with factors

factor()
levels() nlevels()
droplevels()
reorder() relevel()
as.character()
recode()**

**from the car add-on package

Subsetting a data.frame

Ask yourself ...

Do I want to create sub-data.frames for each level of some factor (or unique combination of several factors) ... in order to compute or graph something?

If YES, use data aggregation techniques or conditioning in lattice plots -- don't subset the data.frame.

If NO, then maybe you really do need to subset the data.frame. See previous section, esp. subset().

Sources for further study of topics covered:

Chapter 8 ("Data Aggregation") of Spector (2008). This whole book is extremely valuable. Author's webpage (lots of great material here). Google books search.

Anatomy of a real world data analysis, so far:

```
/Users/jenny/teaching/STAT545A/examples/gapminder/code:
total used in directory 296 available 275170116
drwxr-xr-x  25 jenny  staff    850 Sep 17 13:44 .
drwxr-xr-x   8 jenny  staff    272 Sep 14 12:05 ..
-rw-r--r--@  1 jenny  staff   6148 Sep 11 22:42 .DS_Store
-rw-r--r--   1 jenny  staff   2833 Sep 16 22:49 .Rhistory
-rw-r--r--   1 jenny  staff   4807 Sep 11 13:24 bryan-a01-01-dataPrep.R
-rw-r--r--   1 jenny  staff   6349 Sep 11 13:33 bryan-a01-02-dataMerge.R
-rw-r--r--   1 jenny  staff   5783 Sep 11 14:38 bryan-a01-03-dataExplore.R
-rw-r--r--   1 jenny  staff   3497 Sep 11 22:11 bryan-a01-04-fillContinentData.R
-rw-r--r--   1 jenny  staff   4573 Sep 11 22:24 bryan-a01-05-everyFiveYears.R

 <snip, snip>

 rw-r--r--   1 jenny  staff   6438 Sep 17 13:44 bryan-a01-40-dataAggregation.R
```

The code of my demos of data aggregation using the Gapminder data can be found in the file bryan-a01-40-dataAggregation.R

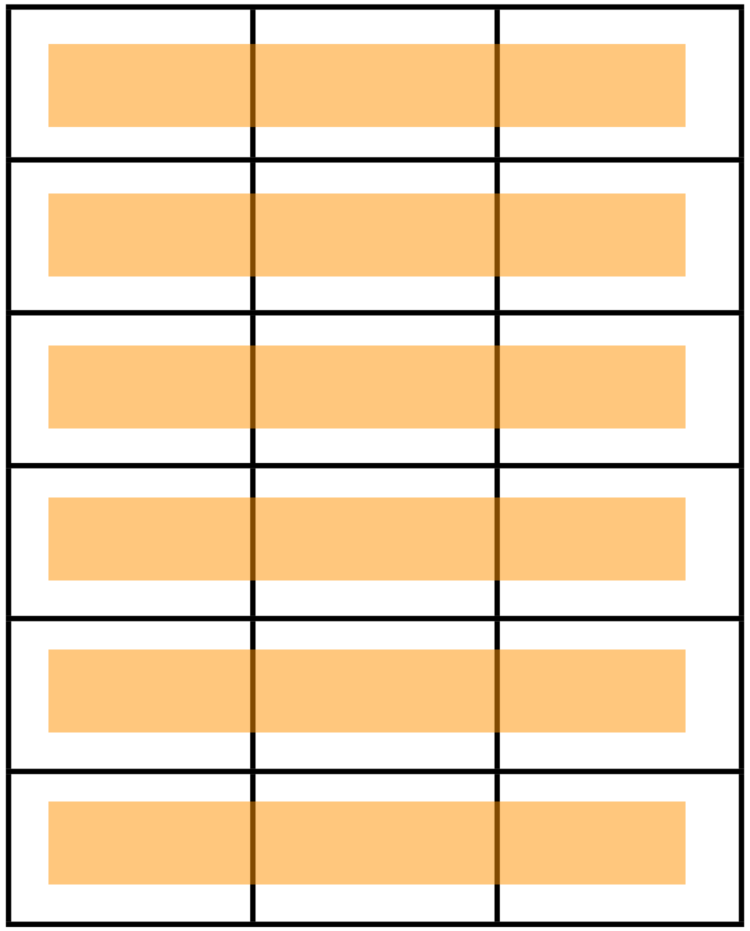
For those situations ... when you need to do <sthg> for various 'chunks' of your dataset

Best method depends on the nature of these chunks

chunks are ...	relevant functions
rows, columns, etc. of matrices / arrays	apply
components of a list(remember data.frames are lists!)	sapply, lapply
groups induced by levels of ≥ 1 factor(s)	aggregate tapply by split (+ [sl]apply)

chunks are ...	relevant functions
rows, columns, etc. of matrices / arrays	apply
components of a list(remember data.frames are lists!)	sapply, lapply
groups induced by levels of ≥ 1 factor(s)	aggregate tapply by split (+ [sl]apply)

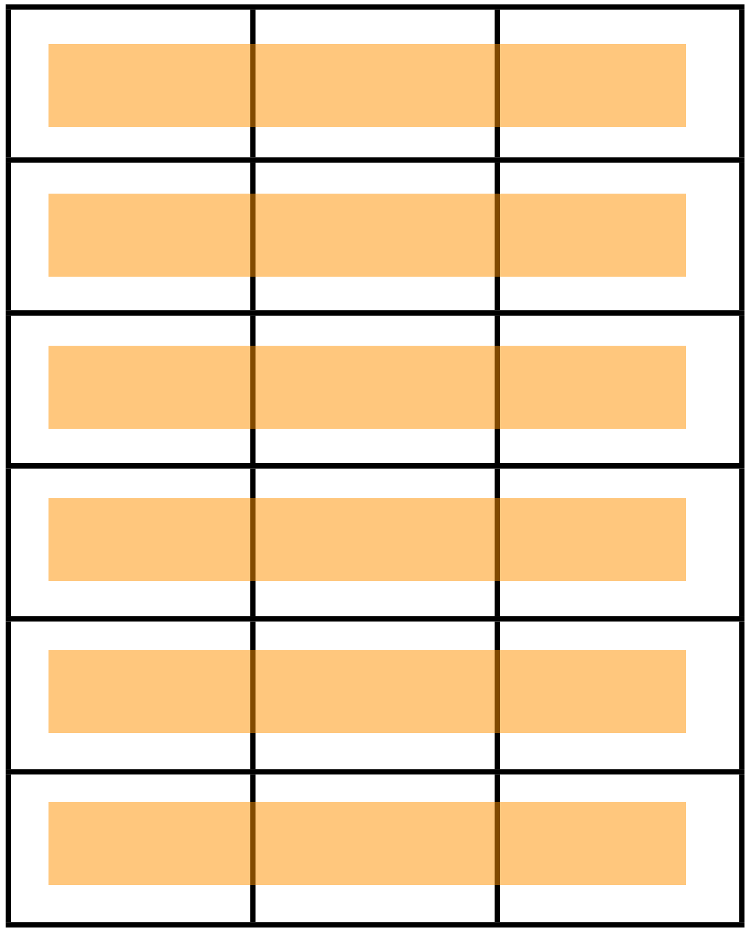
Let chunk = row or column of matrix



```
## grab the Gapminder data to use in examples
whereAmI <- "/Users/jenny/teaching/STAT545A/examples/gapminder/"

## data import from local file
gDat <- read.delim(paste0(whereAmI,
                          "data/gapminderDataFiveYear.txt"))

## reach out and touch the data
str(gDat)
## 'data.frame': 1704 obs. of  6 variables:
##  $ country  : Factor w/ 142 levels "Afghanistan",..: 1 1 1 1 1 1 1 1 1 1 ...
##  $ year     : int  1952 1957 1962 1967 1972 1977 1982 1987 1992 1997 ...
##  $ pop      : num  8425333 9240934 10267083 11537966 13079460 ...
##  $ continent: Factor w/ 5 levels "Africa","Americas",..: 3 3 3 3 3 3  ...
##  $ lifeExp  : num  28.8 30.3 32 34 36.1 ...
##  $ gdpPercap: num  779 821 853 836 740 ...

## creating a toy matrix, so I can demo apply
(jCountries <- sort(c('Canada', 'United States', 'Mexico')))
tinyDat <- subset(gDat, country %in% jCountries)
str(tinyDat)                   # 'data.frame': 36 obs. of 6 variables:
(nY <- length(unique(tinyDat$year)))    # 12 years

jLifeExp <- matrix(tinyDat$lifeExp, nrow = nY)
colnames(jLifeExp) <- jCountries
rownames(jLifeExp) <- tinyDat$year[1:nY]
jLifeExp

apply(jLifeExp, 1, mean)

apply(jLifeExp, 2, median)

jCountries[apply(jLifeExp, 1, which.max)]
```

Complete code, for reference

```
> jLifeExp
     Canada Mexico United States
1952 68.750 50.789        68.440
1957 69.960 55.190        69.490
1962 71.300 58.299        70.210
1967 72.130 60.110        70.760
1972 72.880 62.361        71.340
1977 74.210 65.032        73.380
1982 75.760 67.405        74.650
1987 76.860 69.498        75.020
1992 77.950 71.455        76.090
1997 78.610 73.670        76.810
2002 79.770 74.902        77.310
2007 80.653 76.195        78.242
```



```
> apply(jLifeExp, 1, mean)
    1952     1957     1962     1967     1972     1977     1982     1987
62.65967 64.88000 66.60300 67.66667 68.86033 70.87400 72.60500 73.79267
    1992     1997     2002     2007
75.16500 76.36333 77.32733 78.36333

> apply(jLifeExp, 2, median)
       Canada        Mexico United States
      74.9850       66.2185       74.0150

> jCountries[apply(jLifeExp, 1, which.max)]
 [1] "Canada" "Canada" "Canada" "Canada" "Canada" "Canada" "Canada" "Canada
 [9] "Canada" "Canada" "Canada" "Canada"
```

```
apply(jLifeExp, 1, mean)
```

"Take this matrix and for every row, compute the mean."

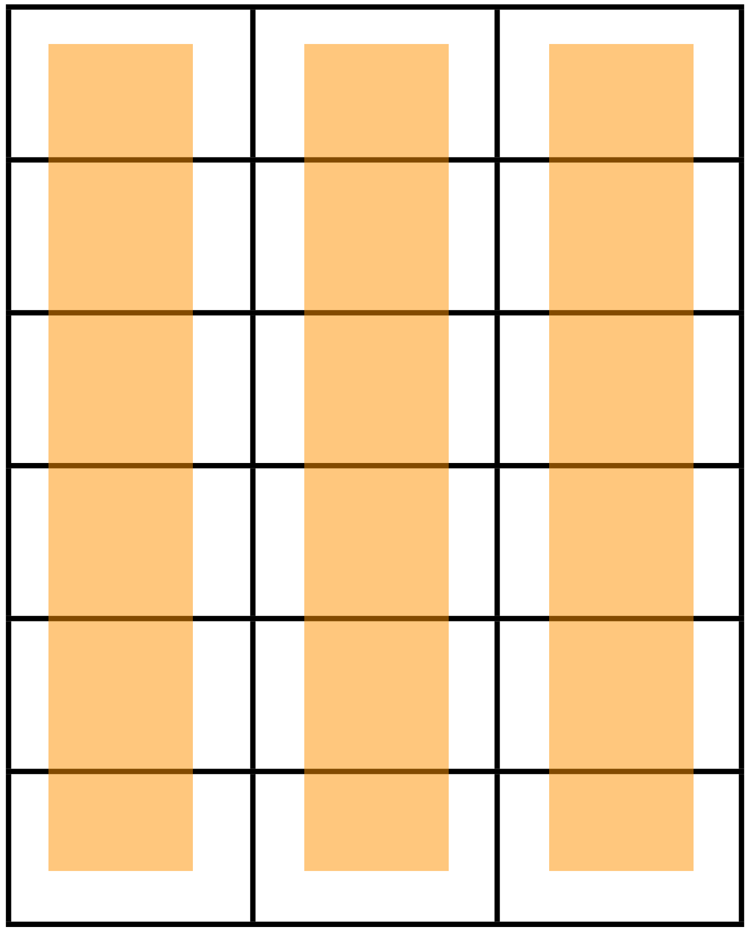
```
apply(jLifeExp, 2, median)
```

"Take this matrix and for every column, compute the median."

Note: apply() works perfectly well on arrays of dimension 3 and higher. Read the docs and proceed with care.

chunks are ...	relevant functions
rows, columns, etc. of matrices / arrays	apply
components of a list(remember data.frames are lists!)	sapply, lapply
groups induced by levels of ≥ 1 factor(s)	aggregate tapply by split (+ [sl]apply)

Let chunk = variable in a data.frame



```
> sapply(gDat, summary)
$country
             Afghanistan                            Albania                            Algeria
                      12                                 12                                 12
 <snip, snip>
                 (Other)
                     516

$year
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   1952    1966    1980    1980    1993    2007

$pop
     Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
6.001e+04 2.794e+06 7.024e+06 2.960e+07 1.959e+07 1.319e+09

$continent
  Africa Americas     Asia   Europe  Oceania
     624      300      396      360       24

$lifeExp
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  23.60   48.20   60.71   59.47   70.85   82.60

$gdpPercap
    Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
   241.2   1202.0   3532.0   7215.0   9325.0 113500.0
```

```
sapply(gDat, summary)
```

To each component of this list, apply summary().
Recall that a data.frame is a list.

Of course, in the case of summary(), there's an even better way to do this. See next page.
But I wanted to demo with a function that did something sensible for all variables.

```
> summary(gDat)
        country          year           pop               continent  
 Afghanistan:  12   Min.   :1952   Min.   :6.001e+04   Africa  :624  
 Albania    :  12   1st Qu.:1966   1st Qu.:2.794e+06   Americas:300  
 Algeria    :  12   Median :1980   Median :7.024e+06   Asia    :396  
 Angola     :  12   Mean   :1980   Mean   :2.960e+07   Europe  :360  
 Argentina  :  12   3rd Qu.:1993   3rd Qu.:1.959e+07   Oceania : 24  
 Australia  :  12   Max.   :2007   Max.   :1.319e+09                 
 (Other)    :1632                                                    
    lifeExp        gdpPercap       
 Min.   :23.60   Min.   :   241.2  
 1st Qu.:48.20   1st Qu.:  1202.1  
 Median :60.71   Median :  3531.8  
 Mean   :59.47   Mean   :  7215.3  
 3rd Qu.:70.85   3rd Qu.:  9325.5  
 Max.   :82.60   Max.   :113523.1  
```

For the record, this is the best way to get a summary() of each variable in a data.frame.

```
> sapply(gDat, is.numeric)
  country      year       pop continent   lifeExp gdpPercap
    FALSE      TRUE      TRUE     FALSE      TRUE      TRUE
```

```
> sapply(gDat, is.numeric)
  country      year       pop continent   lifeExp gdpPercap
    FALSE      TRUE      TRUE     FALSE      TRUE      TRUE

> gDatNum <- subset(gDat, select = sapply(gDat, is.numeric))

> str(gDatNum)
'data.frame':  1704 obs. of  4 variables:
 $ year     : int  1952 1957 1962 1967 1972 1977 1982 1987 1992 1997 ...
 $ pop      : num  8425333 9240934 10267083 11537966 13079460 ...
 $ lifeExp  : num  28.8 30.3 32 34 36.1 ...
 $ gdpPercap: num  779 821 853 836 740 ...
```

Effect: new data.frame containing only the numeric Gapminder variables.

```
> str(gDatNum)
'data.frame':	1704 obs. of  4 variables:
 $ year     : int  1952 1957 1962 1967 1972 1977 1982 1987 1992 1997 ...
 $ pop      : num  8425333 9240934 10267083 11537966 13079460 ...
 $ lifeExp  : num  28.8 30.3 32 34 36.1 ...
 $ gdpPercap: num  779 821 853 836 740 ...

> sapply(gDatNum, median)
      year          pop       lifeExp    gdpPercap
   1979.5000 7023595.5000      60.7125    3531.8470

> lapply(gDatNum, median)
$year
[1] 1979.5

$pop
[1] 7023596

$lifeExp
[1] 60.7125

$gdpPercap
[1] 3531.847
```

```
> sapply(gDatNum, median)
      year          pop     lifeExp   gdpPercap
 1979.5000 7023595.5000     60.7125   3531.8470

> lapply(gDatNum, median)
$year
[1] 1979.5

$pop
[1] 7023596

$lifeExp
[1] 60.7125

$gdpPercap
[1] 3531.847
```

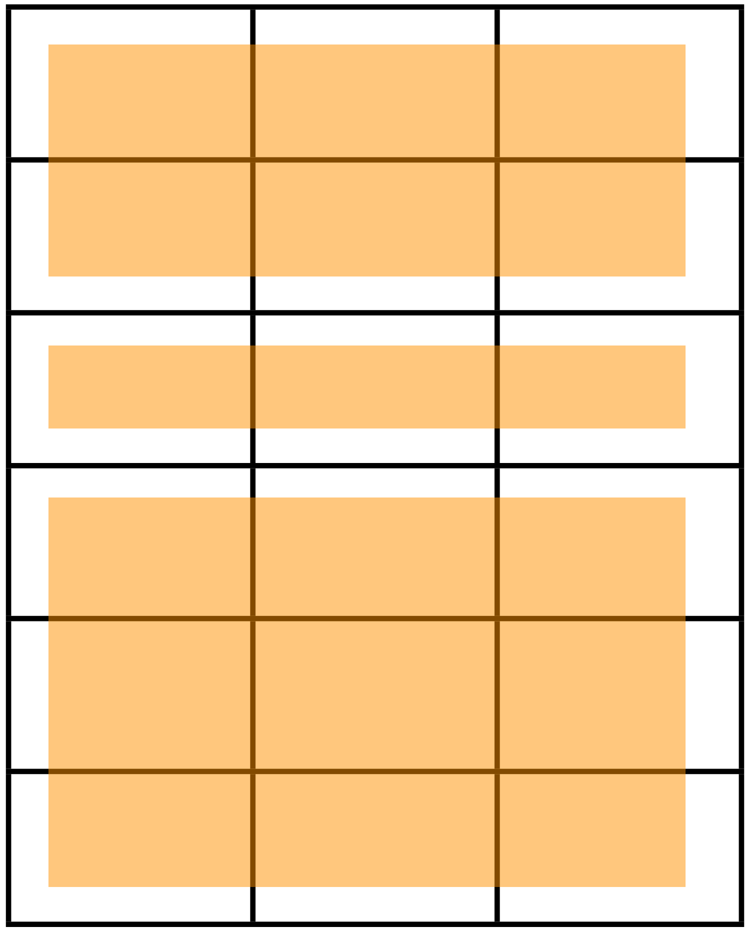
sapply() and lapply() both operate on lists, component-wise.
sapply() tries hard to tidy up the return value, e.g. re-package for your convenience. lapply() does not; it always returns a list.

```
> sapply(gDatNum, range)
     year        pop lifeExp   gdpPercap
[1,] 1952      60011  23.599    241.1659
[2,] 2007 1318683096  82.603 113523.1329

> lapply(gDatNum, range)
$year
[1] 1952 2007

$pop
[1]      60011 1318683096

$lifeExp
[1] 23.599 82.603

$gdpPercap
[1]    241.1659 113523.1329
```

Another demo of difference in return value.

chunks are ...	relevant functions
rows, columns, etc. of matrices / arrays	apply
components of a list (remember data.frames are lists!)	sapply, lapply
groups induced by levels of ≥ 1 factor(s)	tapply aggregate by split (+ [sl]apply)

Let chunk = ragged groups of elements of a vector or rows of a data.frame



Divide this vector into chunks ...

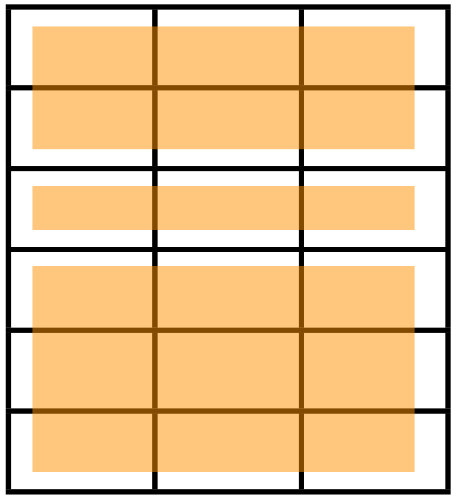
based on this factor and ...

apply this function to each chunk

```
## introducing tapply
with(gDat,
     tapply(lifeExp, continent, max))
 Africa Americas     Asia   Europe  Oceania
 76.442   80.653   82.603   81.757   81.235
```

The function to evaluate can be built-in, like max() above, custom but defined in advance, or custom and defined 'on the fly', as I've done below and later in this class.

```
> ## how many countries for each continent?
> with(gDat,
+      tapply(country, continent, function(x) {
+        length(unique(x))
+      }))
  Africa Americas     Asia   Europe  Oceania
      52       25       33       30        2
```



```
> ## tapply result often needs clean-up

> (rangeLifeExp <- with(gDat,
+                        tapply(lifeExp, continent, range)))
$Africa
[1] 23.599 76.442

$Americas
[1] 37.579 80.653

$Asia
[1] 28.801 82.603

$Europe
[1] 43.585 81.757

$Oceania
[1] 69.120 81.235

> str(rangeLifeExp)
List of 5
 $ Africa  : num [1:2] 23.6 76.4
 $ Americas: num [1:2] 37.6 80.7
 $ Asia    : num [1:2] 28.8 82.6
 $ Europe  : num [1:2] 43.6 81.8
 $ Oceania : num [1:2] 69.1 81.2
 - attr(*, "dim")= int 5
 - attr(*, "dimnames")=List of 1
  ..$ : chr [1:5] "Africa" "Americas" "Asia" "Europe" ...
```

Output of tapply() often benefits from some simplification and/or clean-up.

```
> (rangeLifeExp <- with(gDat,
+                        tapply(lifeExp, continent, range)))
$Africa
[1] 23.599 76.442

$Americas
[1] 37.579 80.653

$Asia
[1] 28.801 82.603

$Europe
[1] 43.585 81.757

$Oceania
[1] 69.120 81.235

<snip, snip>
```

Here, you'd like to stack up the vectors row- or column-wise. rbind() and cbind() are the functions that do that. But the naive implementation, demonstrated here, is inherently flawed. In the long run, you will be killed by the requirement to enumerate the vectors you wish to rbind().

```
> ## rbind does what we want, but does not scale well
> rbind(rangeLifeExp[[1]], rangeLifeExp[[2]],
+       rangeLifeExp[[3]], rangeLifeExp[[4]], rangeLifeExp[[5]])
       [,1]   [,2]
[1,] 23.599 76.442
[2,] 37.579 80.653
[3,] 28.801 82.603
[4,] 43.585 81.757
[5,] 69.120 81.235
```

```
> (rangeLifeExp <- with(gDat,
+                       tapply(lifeExp, continent, range)))
$Africa
[1] 23.599 76.442

$Americas
[1] 37.579 80.653

$Asia
[1] 28.801 82.603

$Europe
[1] 43.585 81.757

$Oceania
[1] 69.120 81.235

<snip, snip>
```

This is the right way -- the general way -- to do this. do.call() is an extremely handy function for making matrices or data.frames out of lists with valid components. This is a recurring task after data aggregation. Note we also get informative row names for free, i.e. they propagate from the factor level labels. This happens alot, reinforcing the rewards of good names, factor level labels, etc.

```
> ## do.call scales (and gets nice row names)
> do.call(rbind, rangeLifeExp)
           [,1]   [,2]
Africa   23.599 76.442
Americas 37.579 80.653
Asia     28.801 82.603
Europe   43.585 81.757
Oceania  69.120 81.235
```

```
> (rangeLifeExp <- with(gDat,
+                        tapply(lifeExp, continent, range)))
$Africa
[1] 23.599 76.442

$Americas
[1] 37.579 80.653

$Asia
[1] 28.801 82.603

$Europe
[1] 43.585 81.757

$Oceania
[1] 69.120 81.235

<snip, snip>
```

To sum up ...

By analogy: sapply has lapply
tapply() has <not much>
i.e. you have to do your own clean-up :-(

But the do.call() trick will work wonders.

```
> ## do.call scales (and gets nice row names)
> do.call(rbind, rangeLifeExp)
           [,1]   [,2]
Africa   23.599 76.442
Americas 37.579 80.653
Asia     28.801 82.603
Europe   43.585 81.757
Oceania  69.120 81.235
```

```
> (rangeLifeExp <- with(gDat,
+                        tapply(lifeExp, continent, range)))
$Africa
[1] 23.599 76.442

$Americas
[1] 37.579 80.653

$Asia
[1] 28.801 82.603

$Europe
[1] 43.585 81.757

$Oceania
[1] 69.120 81.235

<snip, snip>
```

BUT ...

tapply() only works on single variables; in this example, lifeExp

what to do when you need to work with multiple variables at once?

consider by()

```
> ## do.call scales (and gets nice row names)
> do.call(rbind, rangeLifeExp)
           [,1]   [,2]
Africa   23.599 76.442
Americas 37.579 80.653
Asia     28.801 82.603
Europe   43.585 81.757
Oceania  69.120 81.235
```

Divide this [data.frame](#) into chunks ...

based on this factor and ...

apply this function to each chunk

```
> (yearMin <- min(gDat$year))
[1] 1952
> coefEst <- by(gDat, gDat$country, function(cty) {
+   coef(lm(lifeExp ~ I(year - yearMin), cty))
+ })
> coefEst
gDat$country: Afghanistan
      (Intercept) I(year - yearMin)
       29.9072949         0.2753287
-------------------------------------
gDat$country: Albania
      (Intercept) I(year - yearMin)
       59.2291282         0.3346832
-------------------------------------
<snip, snip>
-----------------------------------------------------------
gDat$country: Zimbabwe
      (Intercept) I(year - yearMin)
      55.22124359       -0.09302098
```

Suddenly, fitting a regression model to each of 142 countries doesn't look so bad.

Clean-up tasks you will do over and over again: convert to data.frame (often with the do.call trick), exert control over factor conversion & levels, give variables decent names.

```
> coefEst
gDat$country: Afghanistan
      (Intercept) I(year - yearMin)
       29.9072949         0.2753287
------------------------------------
<snip, snip>
------------------------------------
gDat$country: Zimbabwe
      (Intercept) I(year - yearMin)
      55.22124359       -0.09302098

> ## clean up
> coefEst <- data.frame(do.call(rbind,coefEst))
> coefEst <-
+   data.frame(country = factor(rownames(coefEst),
+                levels = levels(gDat$country)),
+              coefEst)
> names(coefEst) <- c('country','intercept','slope')
> rownames(coefEst) <- NULL
> peek(coefEst)
        country intercept       slope
10      Belgium  67.89192  0.20908462
67        Japan  65.12205  0.35290420
88      Myanmar  41.41155  0.43309510
91  Netherlands  71.88962  0.13668671
93    Nicaragua  43.04513  0.55651958
121       Sudan  37.87419  0.38277483
141      Zambia  47.65803 -0.06042517
```

Bringing the continent info back .....

match() is invaluable for "table look-up" tasks.

```
> peek(coefEst)
             country intercept     slope
11             Benin  39.58851 0.3342329
15            Brazil  51.51204 0.3900895
56  Hong Kong, China  63.42864 0.3659706
78            Malawi  36.91037 0.2342259
94             Niger  35.15067 0.3421091
114        Singapore  61.84588 0.3408860
124      Switzerland  69.45372 0.2222315

> ## bring in continent

> coefEstVersion1 <- coefEst

> coefEstVersion1$continent <-
+   gDat$continent[match(coefEst$country, gDat$country)]

> peek(coefEstVersion1)
        country intercept     slope continent
9    Bangladesh  36.13549 0.4981308      Asia
36     Djibouti  36.27715 0.3674035    Africa
45       France  67.79013 0.2385014    Europe
48      Germany  67.56813 0.2136832    Europe
80         Mali  33.05123 0.3768098    Africa
85   Montenegro  62.24163 0.2930014    Europe
140 Yemen, Rep.  30.13028 0.6054594      Asia
```

return the numeric index of the first match of this in that

```
> coefEstVersion1$continent <-
+   gDat$continent[match(coefEst$country, gDat$country)]
```

then grab the corresponding elements of this other variable

```
> peek(coefEstVersion1)
          country intercept     slope continent
7         Austria  66.44846 0.2419923    Europe
71    Korea, Rep.  49.72750 0.5554000      Asia
93      Nicaragua  43.04513 0.5565196  Americas
118  South Africa  49.34128 0.1691594    Africa
131       Tunisia  44.55531 0.5878434    Africa
135 United States  68.41385 0.1841692  Americas
140   Yemen, Rep.  30.13028 0.6054594      Asia
```

```
> peek(coefEst)
              country intercept     slope
11              Benin  39.58851 0.3342329
15             Brazil  51.51204 0.3900895
56   Hong Kong, China  63.42864 0.3659706
78             Malawi  36.91037 0.2342259
94              Niger  35.15067 0.3421091
114         Singapore  61.84588 0.3408860
124       Switzerland  69.45372 0.2222315

> ## bring in continent
> ## method 2: using merge

> ## create a table with variables for country and continent
> justCountryContinent <- subset(gDat, select = c(country, continent))
> dups <- duplicated(justCountryContinent)
> justCountryContinent <- subset(justCountryContinent, !dups)
> str(justCountryContinent)
'data.frame':	142 obs. of  2 variables:
 $ country  : Factor w/ 142 levels "Afghanistan",..: 1 2 3 4 5 6 7 8 9 10 ...
 $ continent: Factor w/ 5 levels "Africa","Americas",..: 3 4 1 1 2 5 4 3 3 4 ...

> coefEstVersion2 <- merge(coefEst, justCountryContinent)

> peek(coefEstVersion2)
         country continent intercept     slope
46         Gabon    Africa  38.93535 0.4467329
81    Mauritania    Africa  40.02560 0.4464175
82     Mauritius    Africa  55.37077 0.3484538
110 Saudi Arabia      Asia  40.81412 0.6496231
116     Slovenia    Europe  66.08635 0.2005238
132       Turkey    Europe  46.02232 0.4972399
140  Yemen, Rep.      Asia  30.13028 0.6054594
```

merge() is invaluable for ... merging. More general than match.

Data aggregation

tapply is very, very useful when you need to compute on 1 variable for groups defined by 1 or more factors

aggregate is just a wrapper for tapply; personally I don't find it that useful

by is very, very useful ... sort of like tapply for data.frames (under the hood, it is just a wrapper for tapply)

split, and mapply come up in more complicated settings

What is the payoff for all of this hard work doing data aggregation?

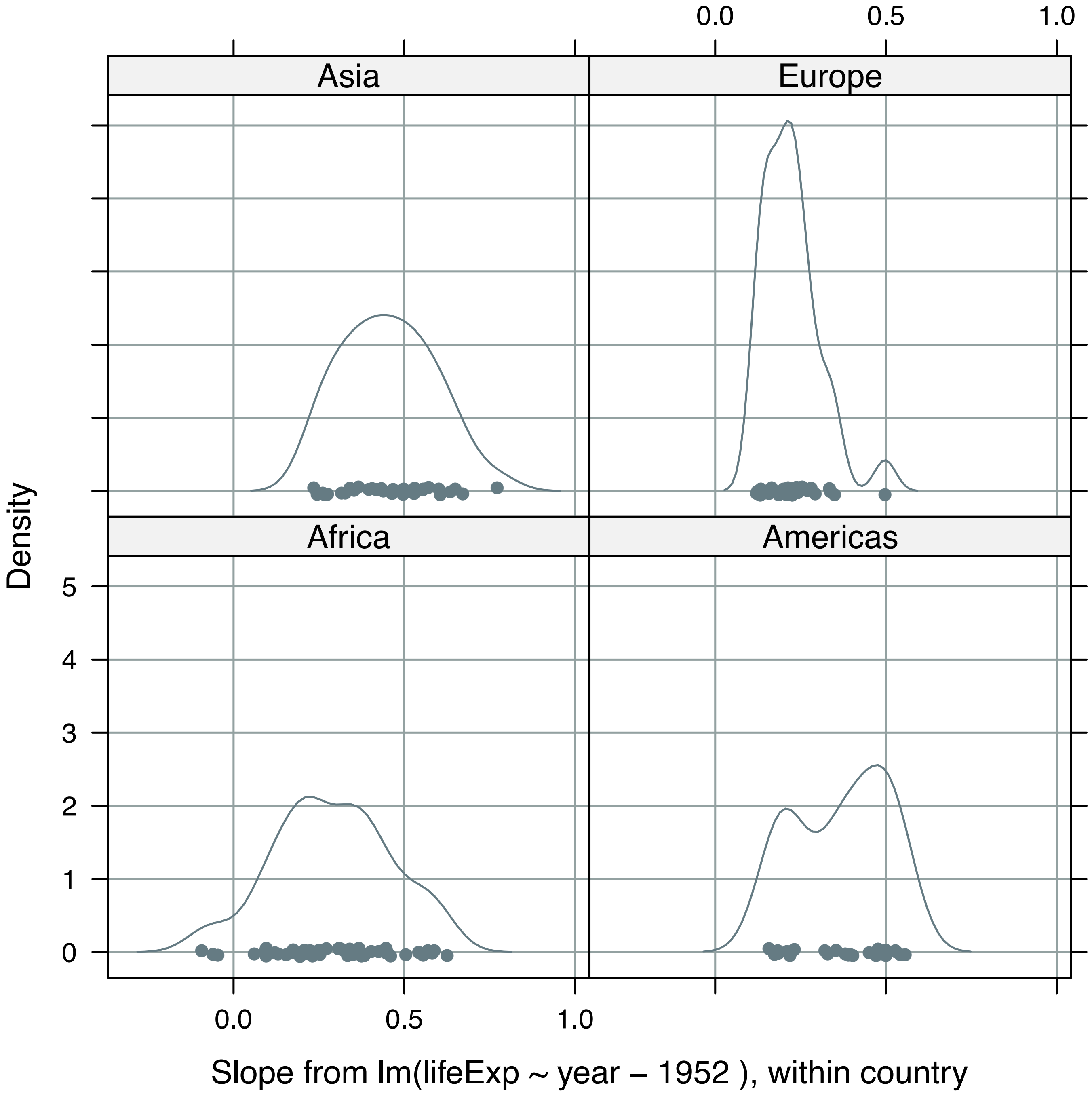
You can make data summaries and figures that other people -- people less skilled at data manipulation -- can’t or won’t make. They don’t know how and/or don’t have enought time.

Visualize: how swiftly is life expectancy increasing over time.

```
densityplot(~ slope | continent, coefEst,
            type = c('p','g'),
            xlab = paste("Slope from lm(lifeExp ~ year -",
              yearMin, "), within country"))
```

```
> bestWorst <- by(coefEst, coefEst$continent, function(z)
{
+   z[c(which.min(z$slope), which.max(z$slope)),]
+ })
> ## drop Oceania ... only 2 countries
> bestWorst <- subset(bestWorst, continent != "Oceania")
> bestWorst                                # lovely!
    country intercept       slope continent status
1  Zimbabwe  55.22124 -0.09302098    Africa  worst
2     Libya  42.10194  0.62553566    Africa   best
3  Paraguay  62.48183  0.15735455  Americas  worst
4 Nicaragua  43.04513  0.55651958  Americas   best
5      Iraq  50.11346  0.23521049      Asia  worst
6      Oman  37.20774  0.77217902      Asia   best
7   Denmark  71.03359  0.12133007    Europe  worst
8    Turkey  46.02232  0.49723986    Europe   best
```

Get the best and worst

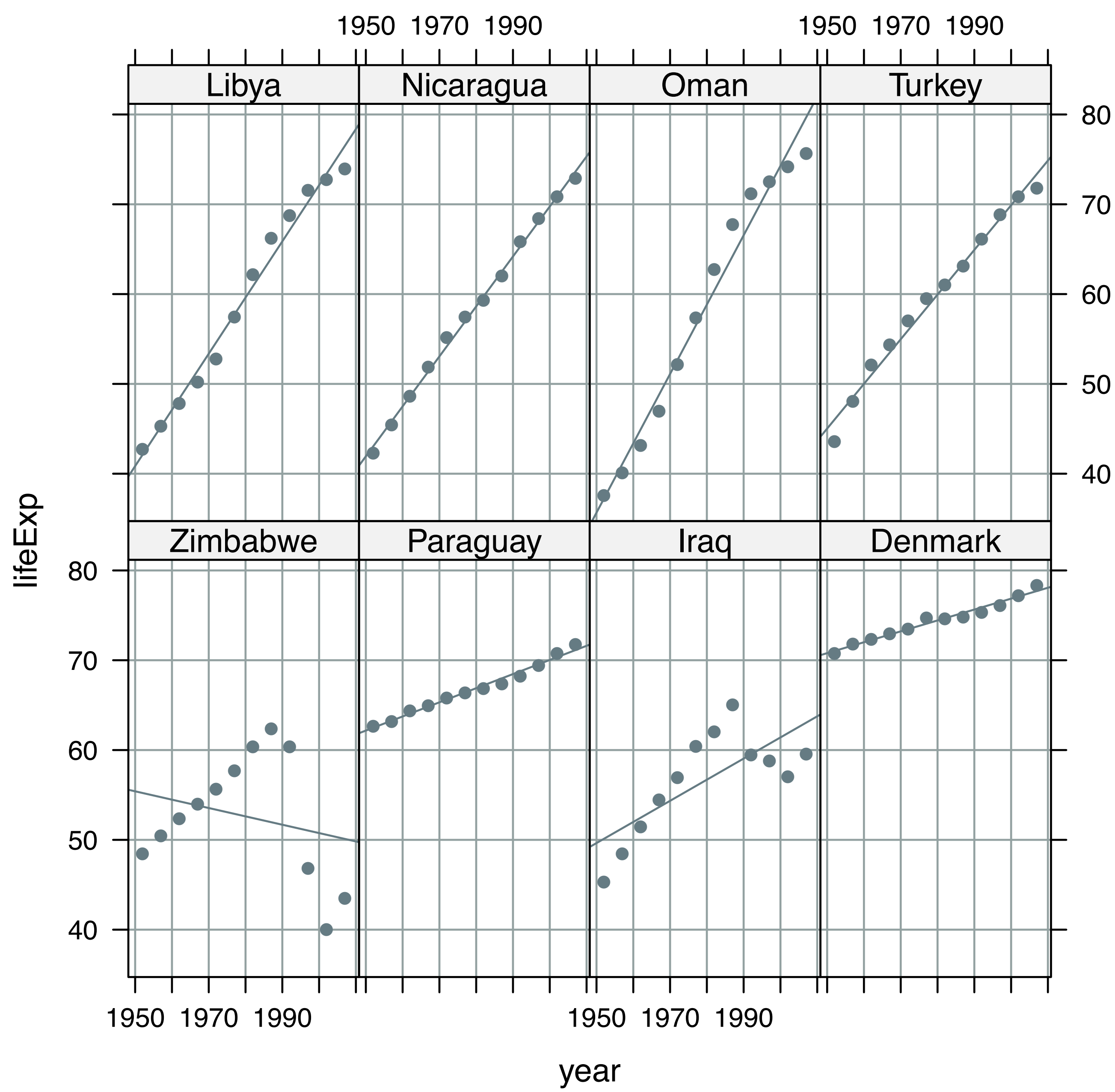



```
> bestWorst                                               # lovely!
      country intercept       slope continent status
1    Zimbabwe  55.22124 -0.09302098    Africa  worst
2       Libya  42.10194  0.62553566    Africa   best
3    Paraguay  62.48183  0.15735455  Americas  worst
4   Nicaragua  43.04513  0.55651958  Americas   best
5        Iraq  50.11346  0.23521049      Asia  worst
6        Oman  37.20774  0.77217902      Asia   best
7     Denmark  71.03359  0.12133007    Europe  worst
8      Turkey  46.02232  0.49723986    Europe   best
```

Europe has least variability in slope (except for Turkey), also lowest mean/median/mode, Asia and Americas have highest mean/median/ mode, Africa has most spread, intriguing ‘first world vs developing nations’ angle via bimodality for the Americas?

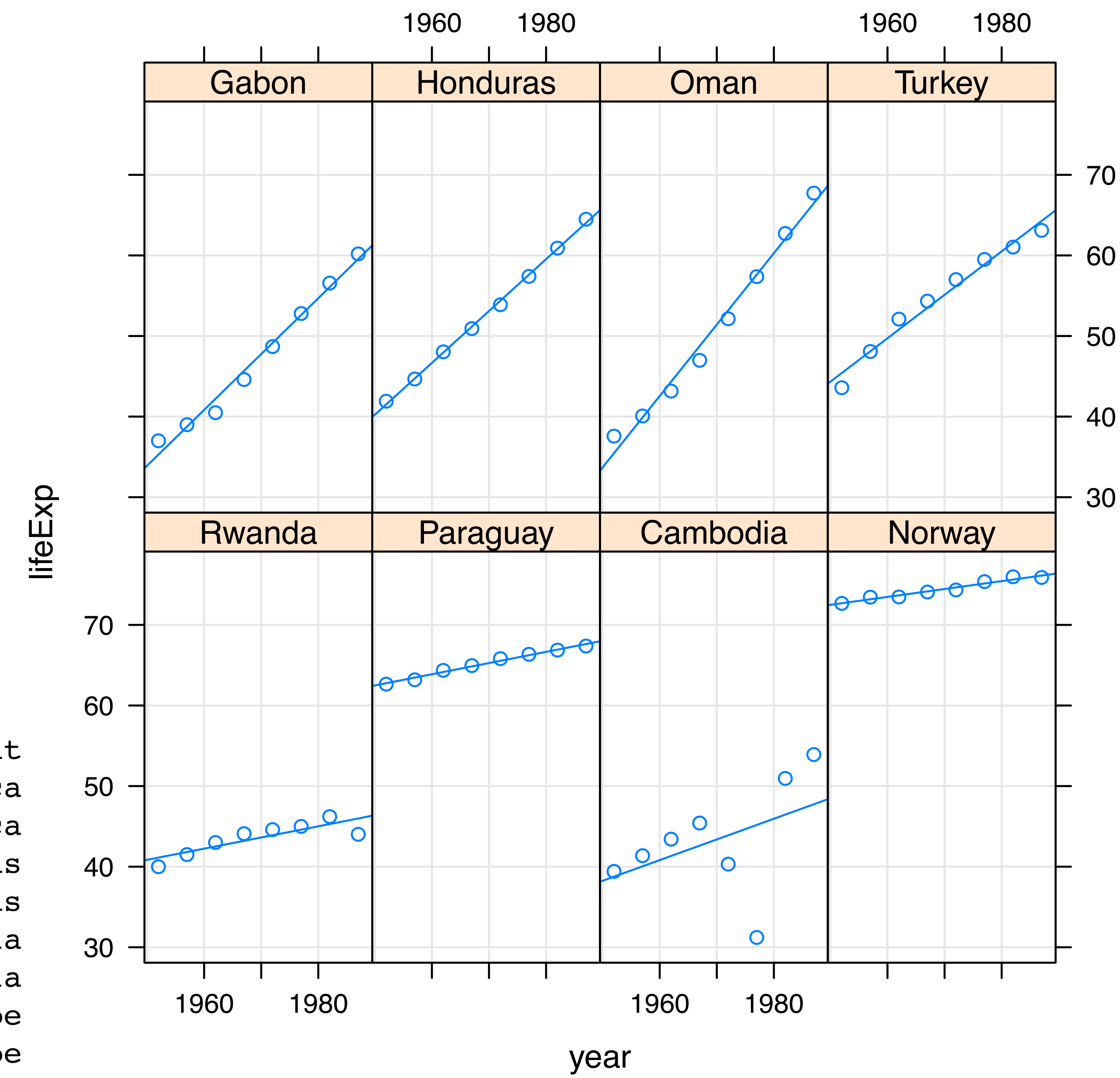
```
## revisiting "raw" data for these interesting examples
zDat <- droplevels(subset(gDat, subset = country %in%
                          bestWorst$country))

## taking charge of the order of levels for country
zDat$country <-
  factor(zDat$country,
         levels = with(bestWorst,
           as.character(country)[c(which(status == 'worst'),
                                   which(status == 'best'))]))

xyplot(lifeExp ~ year | country, zDat,
       layout = c(4,2), type = c('p','g','r'))
```

```
> bestWorst                                       # lovely!
    country intercept       slope continent status
1  Zimbabwe  55.22124 -0.09302098    Africa  worst
2     Libya  42.10194  0.62553566    Africa   best
3  Paraguay  62.48183  0.15735455  Americas  worst
4 Nicaragua  43.04513  0.55651958  Americas   best
5      Iraq  50.11346  0.23521049      Asia  worst
6      Oman  37.20774  0.77217902      Asia   best
7   Denmark  71.03359  0.12133007    Europe  worst
8    Turkey  46.02232  0.49723986    Europe   best
```



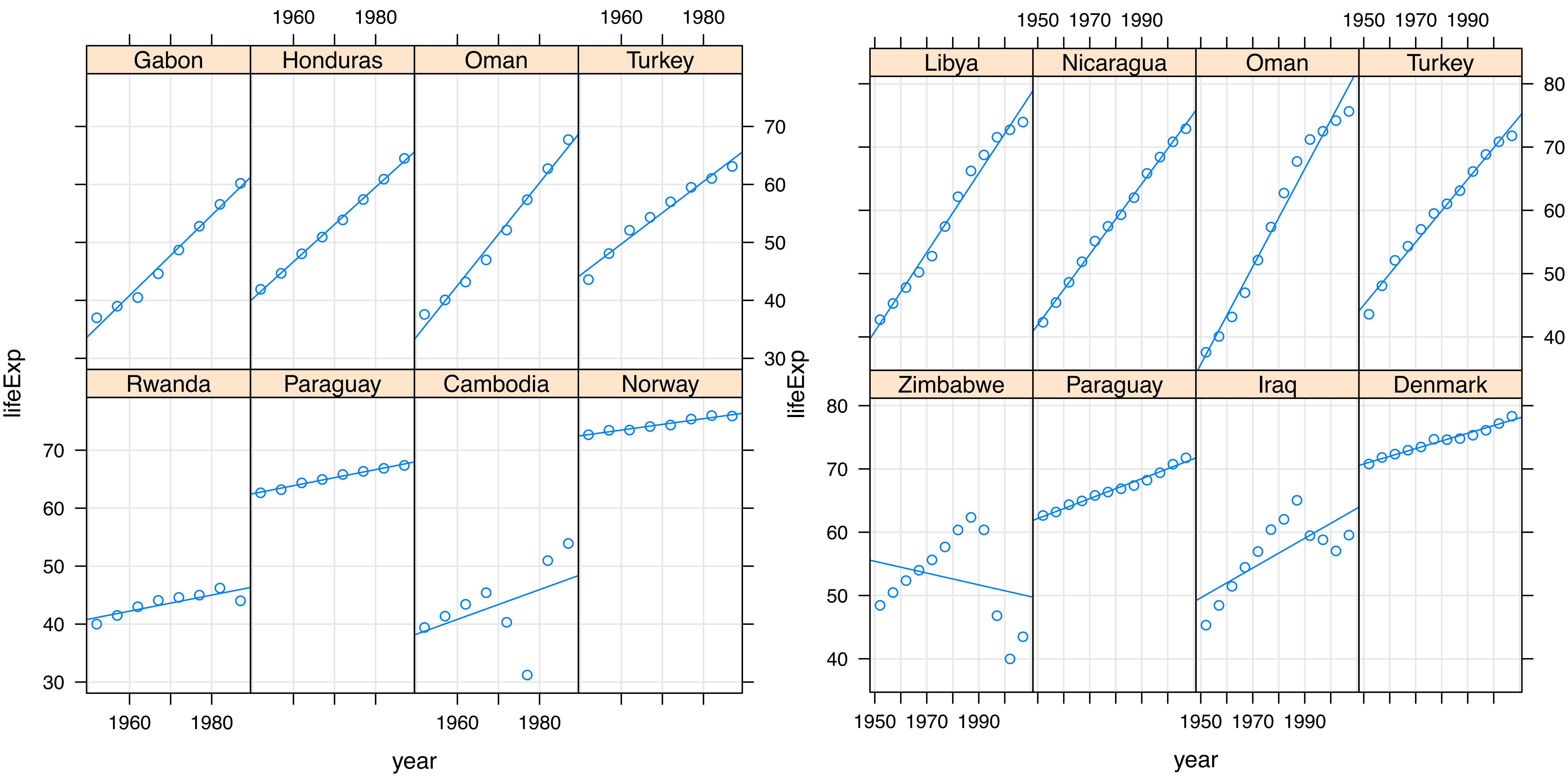
Confirms intuition about biggest slope <--> lowest life exp in 1952 ... sort of, low slopes come about more through sudden marked declines in life expectancy than gradual trends, obvious effect of the Iraq war and the Zimbabwe land redistribution fiasco

from a previous analysis w/ slightly different data cleaning




```
> bestWorst
                  country  intercept     slope continent
Africa.105         Rwanda  -229.5024 0.1386429    Africa
Africa.44           Gabon -1317.7717 0.6931643    Africa
Americas.97      Paraguay  -208.3430 0.1388881  Americas
Americas.52      Honduras -1214.8432 0.6436262  Americas
Asia.20          Cambodia  -462.0226 0.2565500      Asia
Asia.94              Oman -1693.8775 0.8859357      Asia
Europe.93          Norway  -118.5604 0.0979762    Europe
Europe.128         Turkey -1006.5381 0.5389095    Europe
```

Confirms intuition about biggest slope <--> lowest life exp in 1950 ... sort of, Rwanda seems like an especially desperate country, Cambodia in the 1970s: data quality problem? more likely, US/Vietnam War + famine + Pol Pot

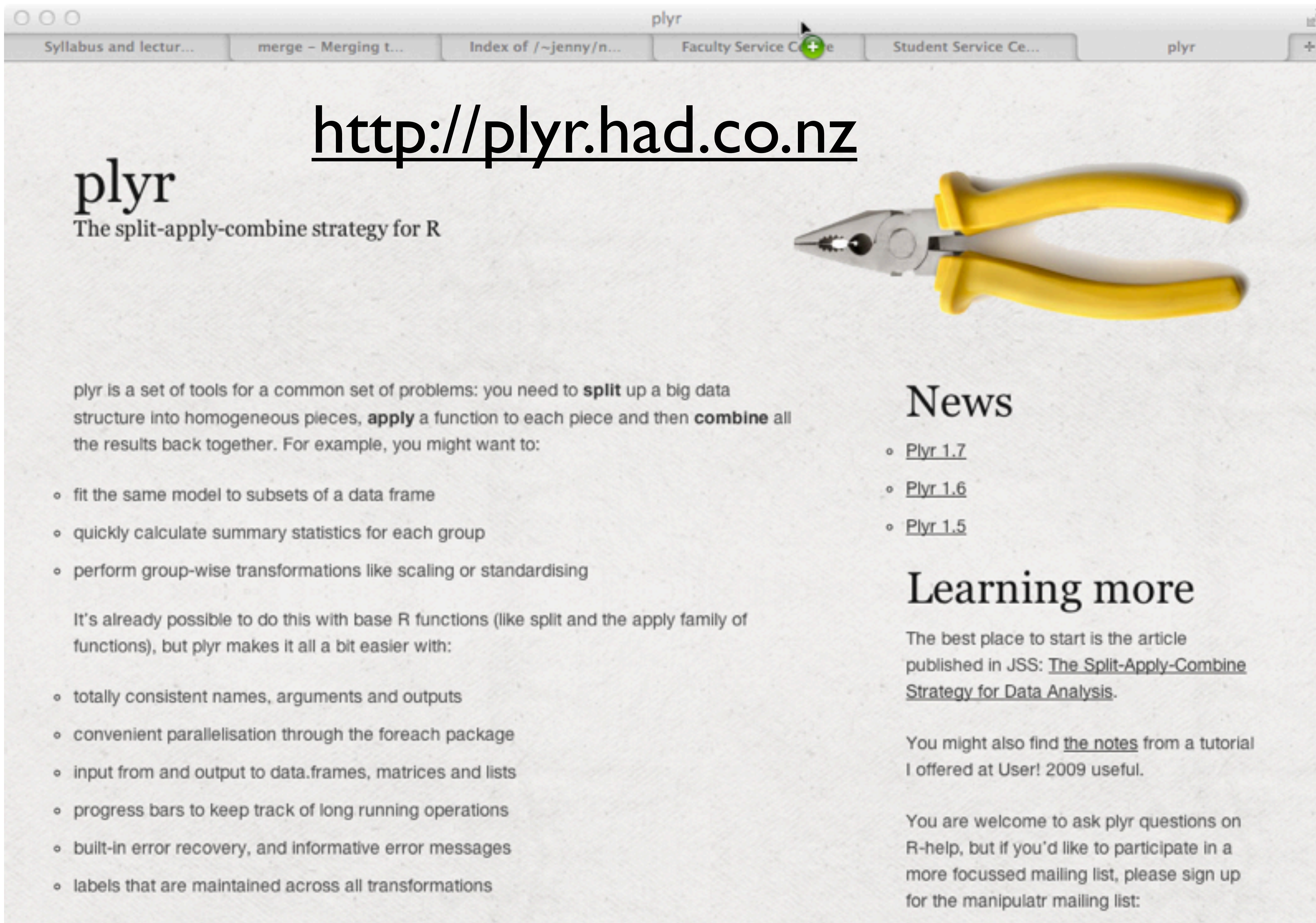


Gabon
Honduras
Oman
Turkey
Rwanda
Paraguay
Cambodia
Norway
lifeExp
year
Libya
Nicaragua
Oman
Turkey
Zimbabwe
Paraguay
Iraq
Denmark
lifeExp
year

Data aggregation conclusions

- [slt]apply and by give you tremendous power; learning curve? Yes, but worth it.
- Above functions, coupled with lattice graphics, make it easy to enact and visualize lots of analyses that would be too much trouble if forced to script them from scratch.
- Most for loops are rendered unnecessary, once you harness the power of data aggregation functions; more about for loops later.
- See code to reveal the least elegant aspect of this case study: recurring need to modify levels of factors; two main issues: eliminating unused levels, reordering factor levels for the side effect in lattice re: panel order.

The plyr package may be worth adopting for data aggregation. JB intends to make the switch! Still good to know about the base R functions, though.....



Next time!

exploring the numeric variables:
population
life expectancy
GDP per capita

... will happen next time ...