STAT 545A
Class meeting #3
Wednesday, September 12, 2012

Dr. Jennifer (Jenny) Bryan

Department of Statistics and Michael Smith Laboratories

Review of last class

take control of where files are read from and written to

'read.table' and friends are main data import functions

'data.frame' is preferred receptacle for data in R

inspect and sanity check R objects early and often

save figures to file, probably PDF, with R code not mouse

'subset' is great function for subsetting a 'data.frame'

walk before run; work small examples; feel free to ignore problematic data and/or make up nice data in order to get started

amplifying on David's comment re: my method of 'taking control of where files are read from and written to' .....

```
setwd("/Users/jenny/teaching/2012-2013/STAT545A/examples/gapminder/")
gDat <- read.delim("data/gapminderData.txt")
canDat <- subset(gDat, country == "Canada")
plot(canDat$year, canDat$pop)
dev.print(pdf, "figs/deleteMeNow.pdf", width = 6, height = 6)
```

vs.

```
whereAmI <- "/Users/jenny/teaching/2012-2013/STAT545A/examples/gapminder/"
gDat <- read.delim(paste0(whereAmI, "data/gapminderData.txt"))
canDat <- subset(gDat, country == "Canada")
plot(canDat$year, canDat$pop)
dev.print(pdf,
          paste0(whereAmI, "figs/deleteMeNow.pdf"), width = 6, height = 6)
```

I haven't been able to construct a great reason to prefer one over the other. The important point is to develop a notion of an analytical project and map that onto the directory(ies) where you read from and write to.

amplifying on David's comment re: my method of 'taking control of where files are read from and written to' .....

Here is some Rstudio documentation on Working Directories and Workspaces and the Projects feature.

The intro underscores my point:
"The default behavior of R for the handling of .RData files and workspaces encourages and facilitates a model of breaking work contexts into distinct working directories. This article describes the various features of RStudio which support this workflow.

IMPORTANT NOTE: In version v0.95 of RStudio a new Projects feature was introduced to make managing multiple working directories more straightforward. The features described below still work however Projects are now the recommended mechanism for dealing with multiple work contexts."

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.

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/notRw/teaching/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. PROBABLY CHANGING TO GITHUB ... WILL DECIDE SOON.

I make heavy use of graphing functions from the lattice package.

Make sure you have it. It is one of the official Recommended packages, so most installations will have it available already.

You will need to load it into your R session before my code will run for you. Do this like so:

```
library(lattice)
```

You may want to make this automatic by adding to your .Rprofile. Here is the official documentation about R Startup.

How *does* R resolve function arguments?

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

> tinyDat <- subset(gDat, subset = country == "Canada")

tinyDat                                  # both give same result
     country year      pop continent lifeExp gdpPercap
241   Canada 1952 14785584  Americas  68.750  11367.16
242   Canada 1957 17010154  Americas  69.960  12489.95
243   Canada 1962 18985849  Americas  71.300  13462.49
244   Canada 1967 20819767  Americas  72.130  16076.59
245   Canada 1972 22284500  Americas  72.880  18970.57
246   Canada 1977 23796400  Americas  74.210  22090.88
247   Canada 1982 25201900  Americas  75.760  22898.79
248   Canada 1987 26549700  Americas  76.860  26626.52
249   Canada 1992 28523502  Americas  77.950  26342.88
250   Canada 1997 30305843  Americas  78.610  28954.93
251   Canada 2002 31902268  Americas  79.770  33328.97
252   Canada 2007 33390141  Americas  80.653  36319.24
```

In the first case above, how does R know what I want it to do with the input ‘country == “Canada”’?

How R resolves function arguments.

By name, if given in ‘name = value’ form.

Otherwise, by *position*.

How I tend to operate: for a function I call often, for arguments I often specify, for the first one or two argument, I may suppress the name, if convenient. Otherwise, I give the name to aid my future self in understanding and re-using the code.

For technical detail, consult the Argument matching section of the R language definition.

```
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, ...)
```

providing data.frame as first argument, determined to be 'x' by position

second argument, if unnamed, will be assumed to be 'subset'

```
tinyDat <- subset(gDat, country == "Canada",
                  select = c("year", "pop"))
```

I only use position to match first 1 or 2 arguments (at most!); after that I give "name = value"

```
> str(gDat)
'data.frame':   3312 obs. of  6 variables:
 $ country  : Factor w/ 187 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      : int  8425333 9240934 10267083 11537966 13079460 14880372 1288181..
 $ continent: Factor w/ 7 levels "","Africa","Americas",..: 4 4 4 4 4 4 4 4 4 ..
 $ lifeExp  : num  28.8 30.3 32 34 36.1 ...
 $ gdpPercap: num  779 821 853 836 740 ...

> peek(gDat)                                  # jb personal function
                  country year      pop continent lifeExp  gdpPercap
1387              Iceland 1994   266075    Europe  79.240 25941.5378
1460              Ireland 2002  3879155    Europe  77.783 34077.0494
2003              Myanmar 1992 40546538      Asia  59.320   347.0000
2090 Netherlands Antilles 1977   170574            72.120 17335.4979
2334          Philippines 1997 75012988      Asia  68.564  2536.5349
2694      Solomon Islands 1972   174500   Oceania  55.506   864.9743
3242            Venezuela 1957  6702668  Americas  57.907  9802.4665
```

We are here: data (seemingly?) successfully imported.

Nothing obviously screwed up.

Let's sanity check and get to know the data.

```
> peek(gDat)                                  # jb personal function
                  country year      pop continent lifeExp  gdpPercap
1387              Iceland 1994   266075    Europe  79.240 25941.5378
1460              Ireland 2002  3879155    Europe  77.783 34077.0494
2003              Myanmar 1992 40546538      Asia  59.320   347.0000
2090 Netherlands Antilles 1977   170574            72.120 17335.4979
2334          Philippines 1997 75012988      Asia  68.564  2536.5349
2694      Solomon Islands 1972   174500   Oceania  55.506   864.9743
3242            Venezuela 1957  6702668  Americas  57.907  9802.4665
```

```
> ## do we have NAs?
> sapply(gDat, function(x) sum(is.na(x)))
  country      year       pop continent   lifeExp gdpPercap
        0         0         0         0         0         0
> ## no NAs ... good!
```

Always check for NAs early. Can be a wildly frustrating and sometimes hard to detect source of trouble in downstream analyses.

(My ‘sapply’ way of checking for NAs will become clear to you very soon.)

exploring the categorical variables:
year (numeric but integer-valued, so sort of categorical)
country
continent

```
> peek(gDat)                                  # jb personal function
                country year      pop continent lifeExp  gdpPercap
1387              Iceland 1994   266075    Europe  79.240 25941.5378
1460              Ireland 2002  3879155    Europe  77.783 34077.0494
2003              Myanmar 1992 40546538      Asia  59.320   347.0000
2090 Netherlands Antilles 1977   170574            72.120 17335.4979
2334          Philippines 1997 75012988      Asia  68.564  2536.5349
2694      Solomon Islands 1972   174500   Oceania  55.506   864.9743
3242            Venezuela 1957  6702668  Americas  57.907  9802.4665
```

```
> ## year
> summary(gDat$year)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   1950    1967    1982    1980    1996    2007


> ## confirming we have 1950, 1951, ..., 2007
> identical(sort(unique(gDat$year)), 1950:2007) # TRUE
[1] TRUE

> length(1950:2007)                             # 58 poss vals for year
[1] 58
```

‘summary’ is often informative

if you know what the possible values should be, check it!

get a sense for how many possible values

```
> table(gDat$year)

1950 1951 1952 1953 1954 1955 1956 1957 1958 1959 1960 1961 1962 1963 1964 1965
  39   24  143   24   24   24   24  144   25   25   26   26  151   26   26   27
1966 1967 1968 1969 1970 1971 1972 1973 1974 1975 1976 1977 1978 1979 1980 1981
  27  156   27   27   27   27  168   32   27   27   27  171   27   27   27   27
1982 1983 1984 1985 1986 1987 1988 1989 1990 1991 1992 1993 1994 1995 1996 1997
 171   27   27   27   27  171   27   27   32   33  183   33   33   33   33  184
1998 1999 2000 2001 2002 2003 2004 2005 2006 2007
  33   33   33   33  187   33   32   30   18  183
```

‘table’ is the main function for tabulation

expects categorical data; here it’s operating on numeric data but OK since year is integer-valued

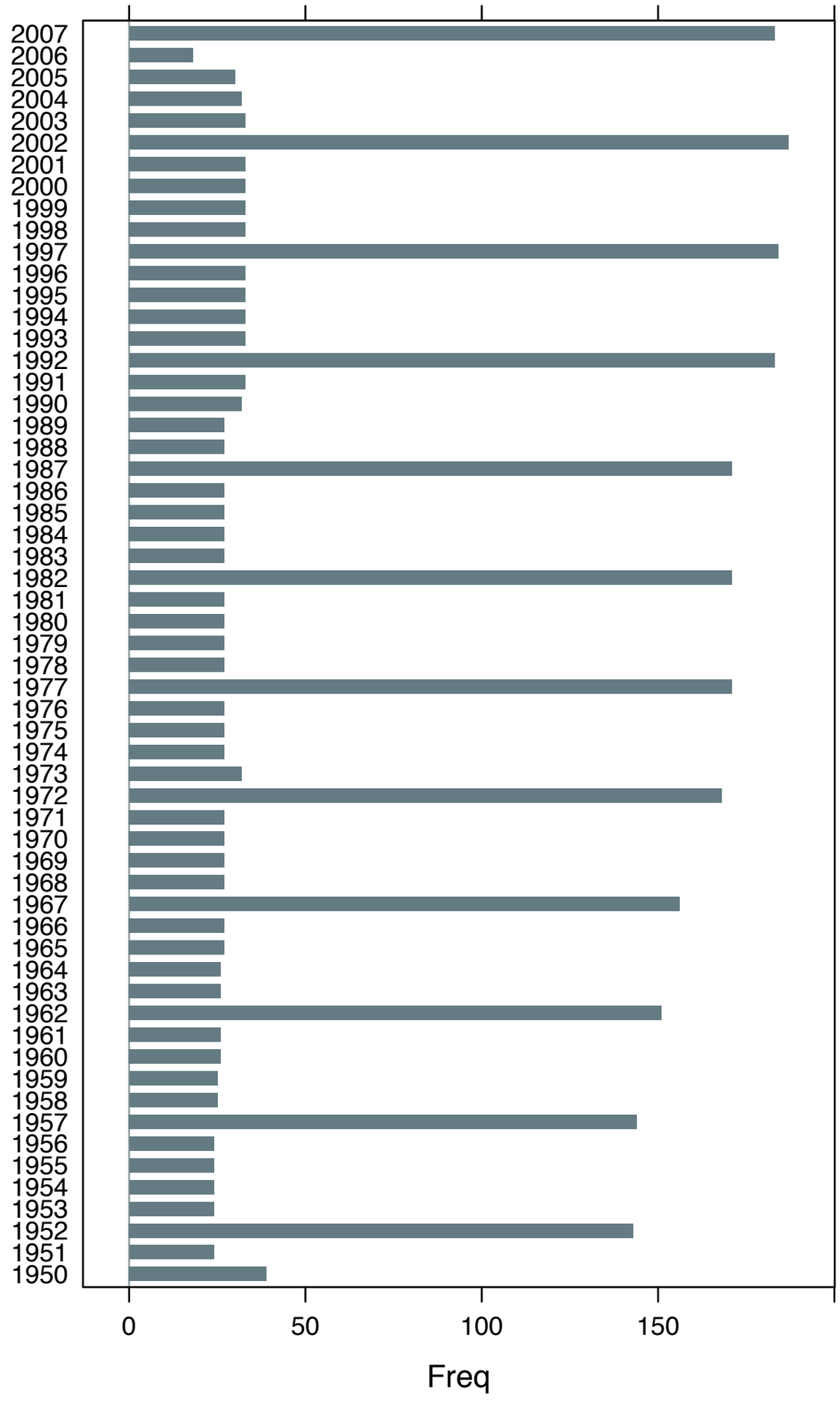
BUT who wants to look at that table? QUICK tell me, does every year appear with the same frequency? OK, no. Is there a pattern? Who knows???

```
barchart(table(gDat$year))
```

Figures are as useful for data checking as they are for downstream tasks

Easy to see that most countries only have data every five years, i.e. 1952, 1957, ...

‘barchart’ is a useful -- but often over-used way -- to present tabulated data



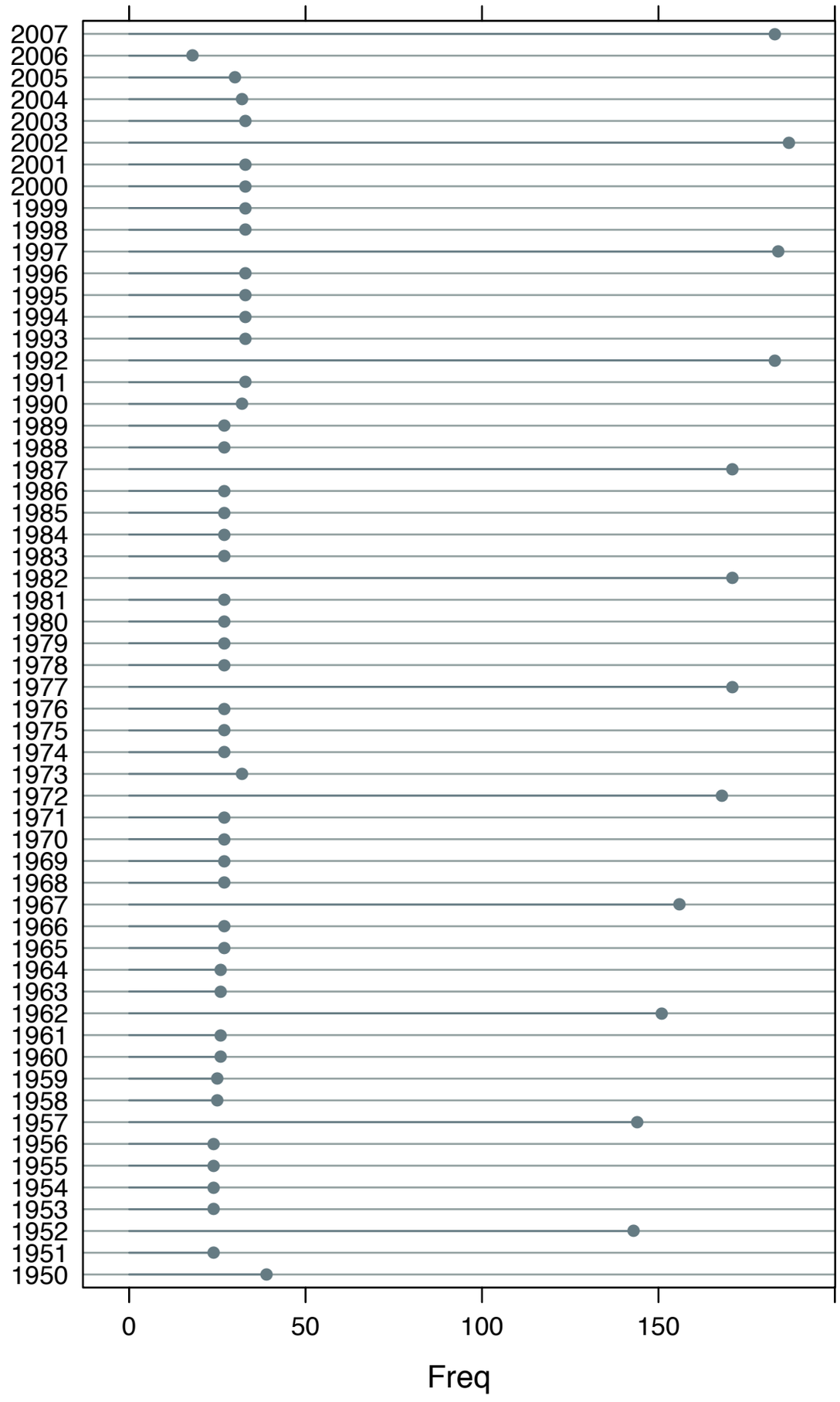
```
dotplot(table(gDat$year),
        origin = 0,
        type = c("p", "h"))
```

‘dotplot’ is often a better choice than ‘barchart’

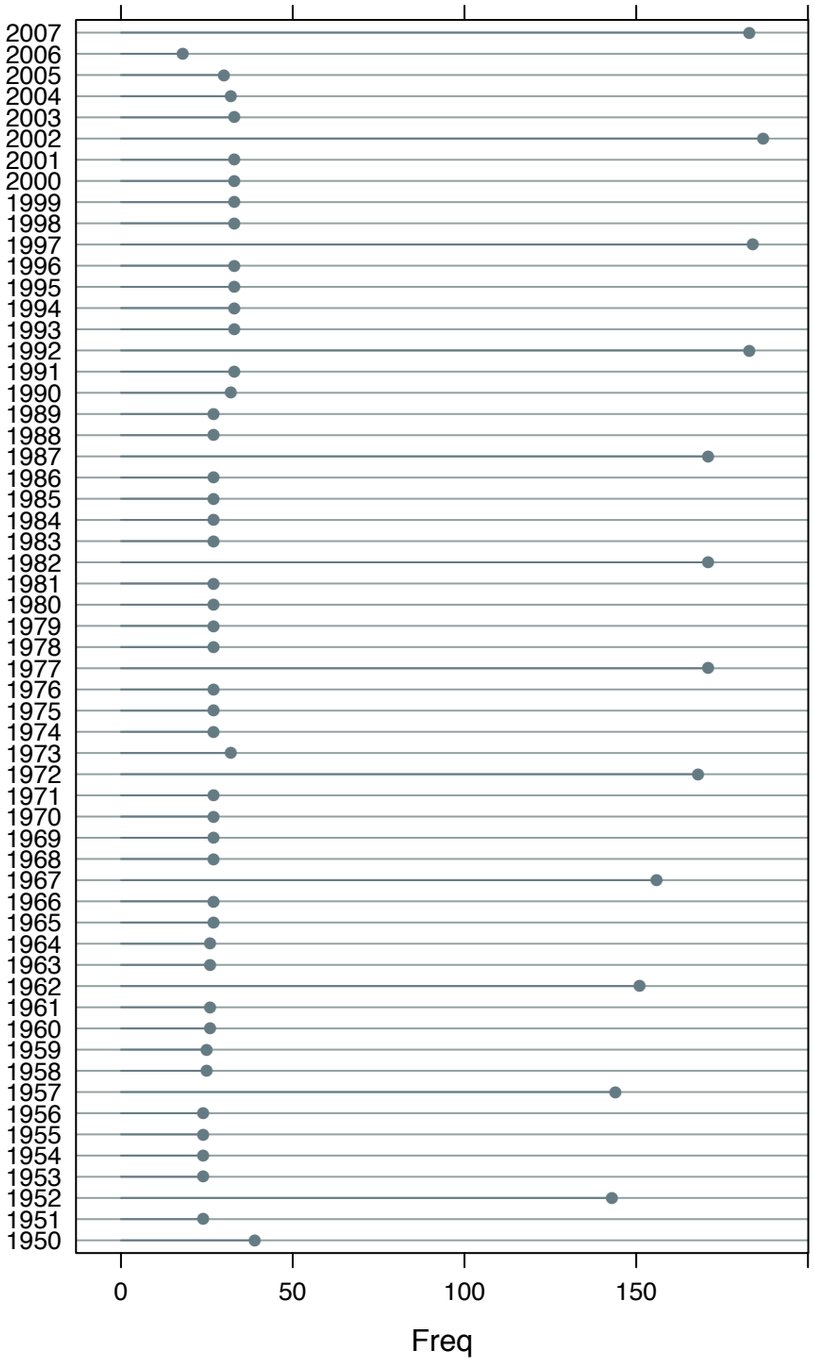
Why better?

higher data : ink ratio

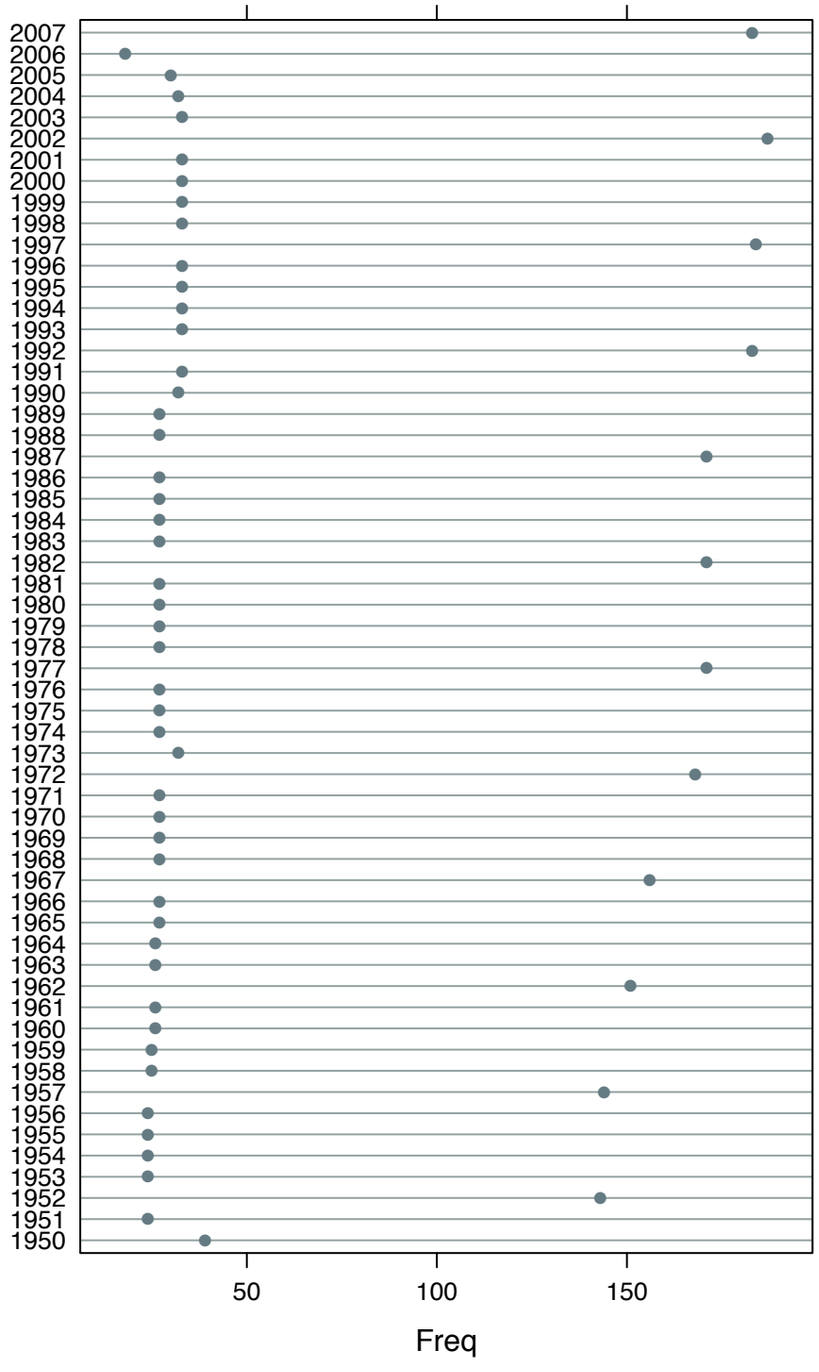
better handling of origin, esp. when origin isn’t well defined or when meaning of number is relative to others, not some absolute scale



```
dotplot(table(gDat$year),
        origin = 0,
        type = c("p", "h"))
```




```
dotplot(table(gDat$year))
```



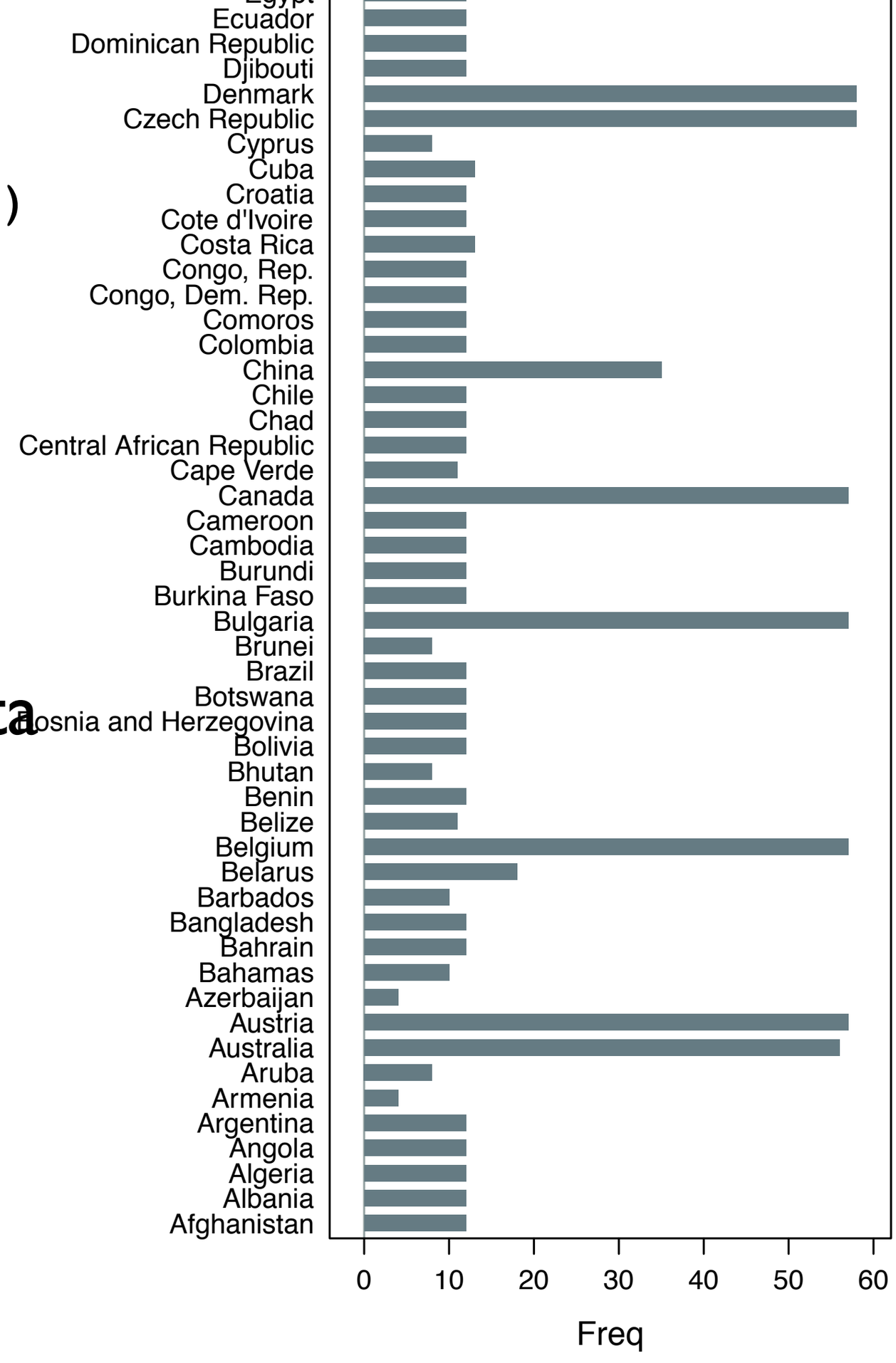
```
> ## country
> str(gDat$country)                          # 187 countries
 Factor w/ 187 levels "Afghanistan",..: 1 1 1 1 1 1 1 1 1 1 ...

> table(gDat$country)

             Afghanistan                  Albania                  Algeria
                      12                       12                       12
                  Angola                Argentina                  Armenia
                      12                       12                        4
                       8                       12                       12
 <snip snip .... many other lines like this scrolled by .... yawn >
           United States                  Uruguay                Uzbekistan
                      57                       12                        4
                 Vanuatu                Venezuela                  Vietnam
                       7                       12                       12
 West Bank and Gaza                 Yemen, Rep.                   Zambia
                      12                       12                       12
                Zimbabwe
                      12
```

as before, with raw data and with our tabulation by year, huge tables of numbers are hard to digest ... make a figure!

```
barchart(table(gDat$country))
```



what we learn / confirm:

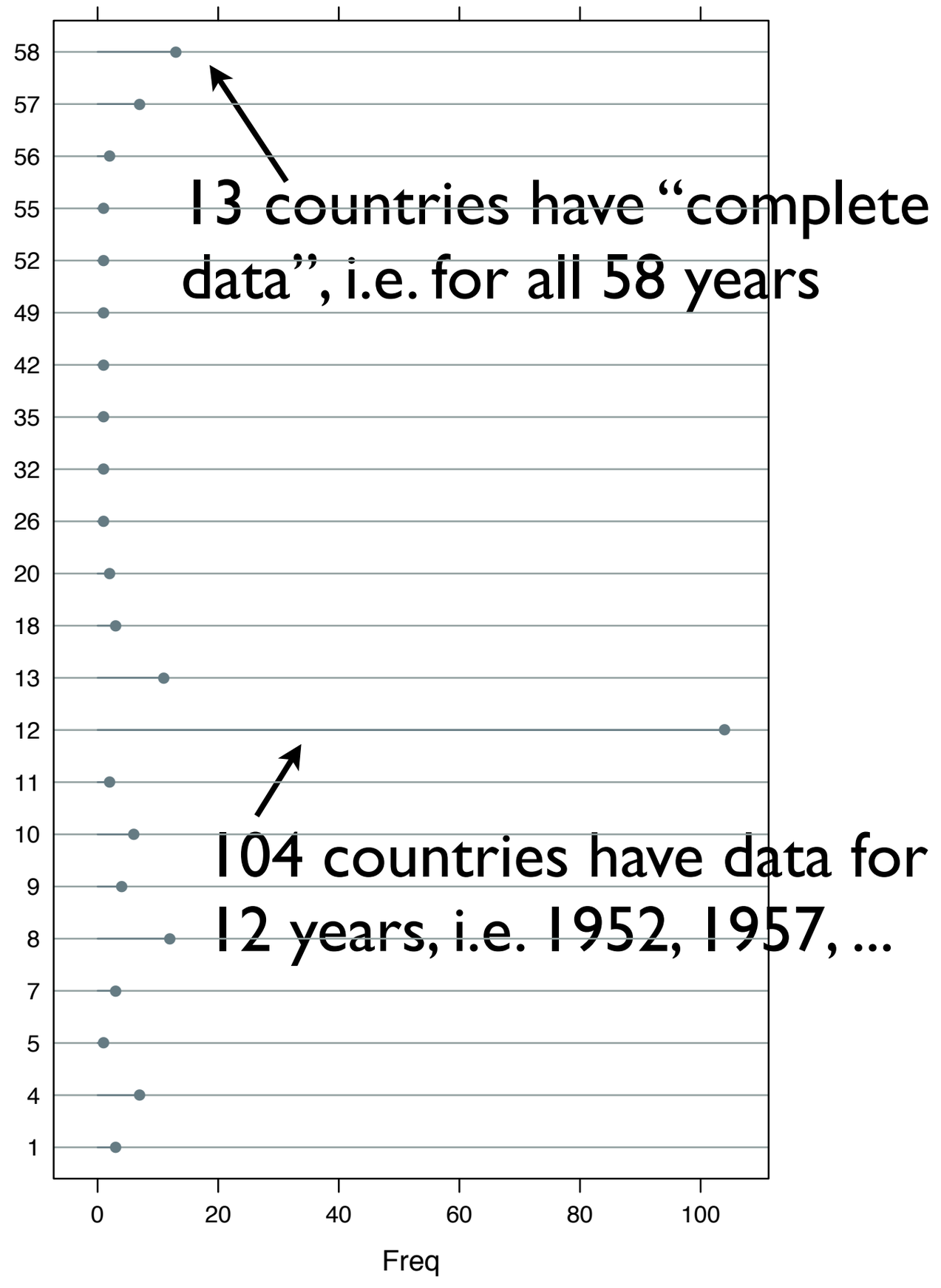
most countries only have data for 12 years (i.e. 1952, 1957, ...)

a few countries, like Canada and Belgium, have data for all 58 years

```
> as.data.frame(table(table(gDat$country)))
   nObs nCountries
1     1          3
...
9    12        104
...
21   57          7
22   58         13

> dotplot(table(table(gDat$country)),
        type = c("p","h"),
        origin = 0)
```

obscure but often useful:
tabulate your original table!



```
> ## continent
> str(gDat$continent)                          # 7 values for continent,
 Factor w/ 7 levels "","Africa","Americas",..: 4 4 4 4 4 4 4 4 4 4 ...

> table(gDat$continent)

      Africa Americas     Asia   Europe      FSU  Oceania
 301     613      343      557     1302      122       74
```

Ouch -- 301 observations don't have continent info. Which countries are affected? Maybe I can ignore them?

```
> noContinent <- droplevels(subset(gDat, continent == ""))

> nlevels(noContinent$country)              # 26 levels
[1] 26

> levels(noContinent$country)
 [1] "Armenia"                "Aruba"                  "Australia"
 [4] "Bahamas"                "Barbados"               "Belize"
 [7] "Canada"                 "French Guiana"          "French Polynesia"
[10] "Georgia"                "Grenada"                "Guadeloupe"
[13] "Haiti"                  "Hong Kong, China"       "Maldives"
[16] "Martinique"             "Micronesia, Fed. Sts."  "Netherlands Antilles"
[19] "New Caledonia"          "Papua New Guinea"       "Reunion"
[22] "Samoa"                  "Sao Tome and Principe"  "Tonga"
[25] "Uzbekistan"             "Vanuatu"
```

No -- we will need to fix this.
(Use of 'droplevels', 'nlevels', 'levels' will be explained shortly.)

We've identified two major issues:

[1] Most countries only have data for twelve years: 1952, 1957, ..., 2007.

[2] 26 countries don't have the continent specified.

How I chose to handle:
Focus only those twelve years and populate the missing continent data myself.

This is an example of "data cleaning".

A mundane but critical step in any real world data analysis.

See the file bryan-a01-04-fillContinentData.R for gory details of filling in the continent data.

Fiddly but well-documented, easy to repeat, extend.

Broad message: try to fix data deficiencies with a script, instead of artisanal Excel work.

Why? Because the heart-breaking truth is that you will need to redo this when the underlying data source rolls to next version, a colleague uses the same instrument to collect new data, a collaborator sends data collected in the same weird manner, etc.

Main R code for filtering on year:

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

str(gDat)
## 'data.frame': 3312 obs. of  6 variables:
.....

gDat <- subset(gDat, subset = year %% 5 == 2)

str(gDat)                  # 'data.frame':    2012 obs. of  6 variables:
```

See the file bryan-a01-05-everyFiveYears.R for gory details.

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
```

Shows how I worked to take unruly data Gapminder actually provides for download and created the input file gapminderData.txt.

Read at your leisure. Will not discuss in class.

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
```

What we're doing today for categorical variables and later for quantitative.

Diagnostic data exploration. What needs to be fixed? What should I be aware of?

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 ...
```

We lost lots of observations (~1700 vs 3100) and countries (142 vs 187), in the name of tidying up.

exploring the quantitative variables:
population
life expectancy
GDP per capita

... will happen next time ...

Basic -- but not necessarily well-known -- R facts that are really useful

(vs. highly technical material only a developer could love)

Sources I relied heavily upon:

Chapters 1 ("Data in R"), 5 ("Factors"), 6 ("Subscripting"), and 8 ("Data Aggregation") of Spector (2008). This whole book is extremely valuable. Author's webpage (lots of great material here). Google books search.

The R language Definition

Personal experience, painful at times

Mode of R objects

- Most, though not all, R objects have one of these "modes" (there are many others):
  - numeric
  - character
  - logical
- An object can only have one mode.
- Numeric includes integer and double floating point, but the user can often gloss over that distinction. When it's time to worry about that, you'll know.

Mode of R objects

- Helpful functions:
  - mode()
  - is.numeric(), is.character(), is.logical()
  - as.numeric(), as.character(), as.logical()

Class of R objects

- An object can also have a class. Here things are more complicated.....
- The rationale for R classes is as with other object-oriented languages. Method dispatch example: Generic functions, like print() and summary(), use the class to determine what exactly they should do with an object.
- Objects can have more than one class (inheritance and all that good stuff) and can have no class (in which case, the mode is usually the class).
- Typical user, especially a newbie, does not need to worry too much about classes.

Class of R objects

- Helpful functions:
  - class()
  - unclass() -- use with care
  - methods() -- examples to run:
    - To see all methods available for objects of class “lm”, the result of fitting a linear model, try
      `methods(class = "lm")`
    - To see all the class-specific methods there are for the all-purpose function str() try
      `methods(generic.function = "str")`

```
> mode(gDat)
[1] "list"

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


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

> class(gDat$country)
[1] "factor"

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

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


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

> class(gDat$lifeExp)
[1] "numeric"
```

mode and class of some of the Gapminder objects

Reach out and touch -- but do not print to screen - your data

```
str()
summary()
head()
tail()
peek() -- not built-in
mode()
class()
```

Reminder of other functions that help you to get and stay acquainted with your R objects. Use them early, use them often.

Simple view of simple R objects that will get you pretty far

Simple view	Technically correct R view		
	mode	class	typeof
character	character	character	character
logical	logical	logical	logical
numeric	numeric	integer or numeric	integer or double
factor	numeric	factor	integer

Simple view of simple R objects that will get you pretty far

Simple view	Technically correct R view		
	mode	class	typeof
character	character	character	character
logical	logical	logical	logical
numeric	numeric	integer or numeric	integer or double
factor	numeric	factor	integer

Factors

See Chapter 5 of Spector (2008).

- Valuable way to store categorical data BUT ...
  - Jenny's Law: A factor variable will be the source of at least one major headache in each data analysis, costing me ~~hours~~ several minutes of valuable time.
- Why needed
  - In modelling: proper use of factors will make it much easier to specify models, construct contrasts, etc.
  - In visualization: lattice is smart about conditioning on factors or conveying factor levels through color, line type, etc.

Factors

See Chapter 5 of Spector (2008).

- Basic trickiness: Factors are stored as integers, with an associated set of labels (usually character strings). The character info is more visible/interpretable, but *don't ever forget* factors are really numeric.
- Factors are "high-maintenance" variables, but I still advise you to Embrace Factors and Their Labels/Levels.
  - Make the labels informative yet concise.
  - Make a deliberate choice of the first or reference level, when relevant.
  - Choose the overall order in a principled way, when relevant. Be prepared to change the order or drop levels at various points in an analysis.

```
> 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 ...

> peek(gDat)
      country year      pop continent lifeExp  gdpPercap
64  Australia 1967 11872264   Oceania  71.100 14526.1246
152 Bosnia and Herzegovina 1987  4338977    Europe  71.140  4314.1148
998  Mongolia 1957   882134      Asia  45.248   912.6626
1115 Nicaragua 2002  5146848  Americas  70.836  2474.5488
1126     Niger 1997  9666252    Africa  51.313   580.3052
1168  Pakistan 1967 60641899      Asia  49.800   942.4083
1213 Philippines 1952 22438691      Asia  47.752  1272.8810
```

Factors in Gapminder dataset are ‘country’ and ‘continent’.

```
> levels(gDat$continent)
[1] "Africa"   "Americas" "Asia"     "Europe"   "Oceania"

> nlevels(gDat$continent)
[1] 5

> table(gDat$continent)

  Africa Americas     Asia   Europe  Oceania
     624      300      396      360       24

> summary(gDat$continent)
  Africa Americas     Asia   Europe  Oceania
     624      300      396      360       24
```

Getting to know a factor ....

Factors

See Chapter 5 of Spector (2008).

- read.table() and data.frame() are the two main functions you will use to create data.frames. By default, they will both convert character variables to factors.
- If you have a good reason, how to prevent this?
  - For an R session: options(stringsAsFactors = FALSE). Put in .Rprofile to make it truly global.
  - Universally within a call to read.table(): include stringsAsFactors = FALSE in the call.
  - For specific variables within a call to read.table(): use the arguments 'as.is' (my top choice) or 'colClasses' (my second choice).

Factors

See Chapter 5 of Spector (2008).

- How to prevent conversion of character to factor when forming data.frames (cont'd)?
  - Universally within a call to data.frame(): include stringsAsFactors = FALSE in the call.
  - For specific variables within a call to data.frame(): protect the variable with I().

Factor booby traps

- Take great care when replacing or adding data to a factor, e.g. catenating two factors with c() or adding observations to a data.frame. Basic approach: factor --> character, add/combine the data, character --> factor.
- Take great care when changing the labels of the levels or when changing the order of the levels. It's easy to mangle the mapping of old labels to new labels or to change only the levels but not the labels, etc etc.
- Beware of subscripting with a factor -- you're probably thinking of the variable as character and trying to subscript by name, but R will use the underlying numeric vector and will subscript by position.

Example of silent but deadly failure when hoping to add a factor level

```
> (jCountry <- factor(c("USA", "Canada")))
[1] USA    Canada
Levels: Canada USA

> ## oops I forgot Mexico!
> (jCountry <- c(jCountry, "Mexico"))
[1] "2"      "1"      "Mexico"
> ## does NOT work
```

```
> (jCountry <- factor(c("USA", "Canada")))
[1] USA    Canada
Levels: Canada USA

> (jCountry <- factor(c(as.character(jCountry), "Mexico")))
[1] USA    Canada Mexico
Levels: Canada Mexico USA
> ## works :-)
```

illustrates this non-obvious workflow:
factor --> character --> (add data) --> factor

Handy Tip

Why do I sometimes surround an R expression with parentheses?

To create and inspect an object at once

```
> ## create, then print to screen
> x <- 4 + 3
> x
[1] 7

> ## surround the expression with () to
> ## create and print to screen at once!
> (x <- 4 + 3)
[1] 7
```

```
> (foo <- sample(gDat$country, size = 5))
[1] Australia Niger     Burundi   Cuba      Cambodia
142 Levels: Afghanistan Albania Algeria Angola Argentina Australia ... Zimbabwe

> (foo2 <- factor(foo))
[1] Australia Niger     Burundi   Cuba      Cambodia
Levels: Australia Burundi Cambodia Cuba Niger

> (foo3 <- foo[ , drop = TRUE])
[1] Australia Niger     Burundi   Cuba      Cambodia
Levels: Australia Burundi Cambodia Cuba Niger

> (foo4 <- droplevels(foo))
[1] Australia Niger     Burundi   Cuba      Cambodia
Levels: Australia Burundi Cambodia Cuba Niger
```

After you eliminate some data, sometimes you wish to rationalize the factor levels, i.e. reduce to those that actually occur. New-ish function ‘droplevels’ is probably best way to go.

Factors

See Chapter 5 of Spector (2008).

- Helpful functions
  - factor()
  - levels(), nlevels()
  - droplevels()
  - reorder(), relevel()*
  - as.character()

*sadly, not nearly as great as they sound

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

Vectors, matrices, arrays

- Single values or scalars are minor players in real problems people tackle with R.
- In fact, in R they are simply vectors of length 1.
  - So vectors are essentially the most basic R object.
- All elements of a vector must be of same mode.
  - R will silently convert if necessary, so be aware this can happen. Often inadvertent or inept combining of data of different modes leads to unexpected conversion.

remember this?

```
> (jCountry <- factor(c("USA", "Canada")))
[1] USA    Canada
Levels: Canada USA

> ## oops I forgot Mexico!
> (jCountry <- c(jCountry, "Mexico"))
[1] "2"      "1"      "Mexico"
> ## does NOT work
```

```
> (z <- 1:10)
 [1]  1  2  3  4  5  6  7  8  9 10

> mode(z)
[1] "numeric"

> is.vector(z)
[1] TRUE
```

```
> (y <- c("red", "blue", "green"))
[1] "red"   "blue"  "green"

> mode(y)
[1] "character"

> is.vector(y)
[1] TRUE
```

c() for concatenate is the most basic way to make a vector

```
> (zz <- sample(100, 4))
[1] 67  7 20 17

> mode(zz)
[1] "numeric"

> is.vector(zz)
[1] TRUE
```

```
> (x <- c("cabbage", pi, TRUE))
[1] "cabbage"          "3.14159265358979" "TRUE"

> mode(x)
[1] "character"

> is.vector(x)
[1] TRUE
```

Vectors, matrices, arrays

- Arrays are multidimensional extensions of vectors. Most common are two-dimensional arrays, i.e. matrices.
- "Vectorized" computations are common and encouraged in R.
  - If two vectors have different lengths, R will recycle the shorter one, often silently. Awesome when that's what you want, awful if you don't.
  - Internally, R stores matrices -- all multidimentioanl arrays, in fact -- as vectors, "stacked" by column. When a matrix is used in a vector context, R silently uses the underlying vector representation. Awesome when that's what you want, awful if you don't.

```
> (x <- matrix(c("cabbage", pi, TRUE, 4.3), nrow = 2))
     [,1]              [,2]
[1,] "cabbage"         "TRUE"
[2,] "3.14159265358979" "4.3"

> mode(x)
[1] "character"

> class(x)
[1] "matrix"

> dim(x)
[1] 2 2

> nrow(x)
[1] 2

> ncol(x)
[1] 2

> x[2, 1]
[1] "3.14159265358979"

> x[3]
[1] "TRUE"
```

```
> ## recycling happens
> (y <- 1:3)
[1] 1 2 3

> (z <- 3:7)
[1] 3 4 5 6 7

> y + z
[1] 4 6 8 7 9
Warning message:
In y + z : longer object length is not a multiple of
shorter object length
```

Vectors, matrices, arrays

- Helpful functions
  - c(), matrix(), array()
  - is.vector(), as.matrix(), etc etc
  - length(), nrow(), ncol(), dim()
  - names(), dimnames(), row.names(), rownames(), colnames()

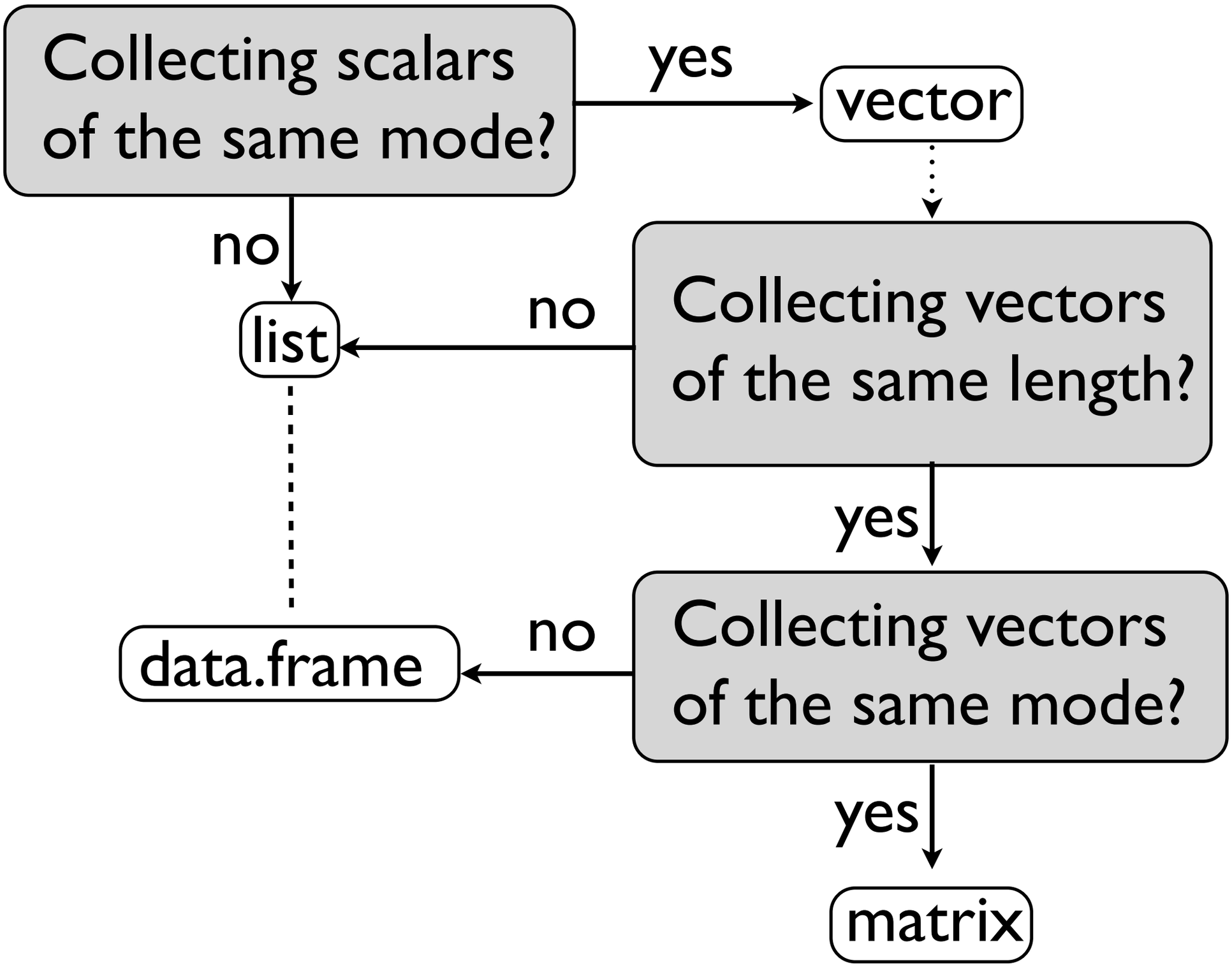
One last data type: list

- Think of R lists as a generalization of R vectors. A list has elements BUT they don't have to have the same type or length.
- For the most part, let's learn about lists on a need-to-know basis. Handy facts and functions:
  - A data.frame is a very special list in which the elements -- usually factors or numeric or character vectors -- have the same length.
  - list(), is.list(), unlist(), length(), names()

Names

- Elements of vectors and, indeed, of more complicated objects like data.frames and matrices, can have names.
- Names are used for display (printing to screen, plots, etc.) and can be used for access and even assignment.
- It took me a while to learn this but trust me: Embrace Names. Specific recommendations:
  - Set-up names carefully. Make them informative yet concise.
  - Use names heavily for access & subsetting. Makes code much more robust and self-documenting. Eventually this will force you to increase your skills with regular expressions, so be prepared.

“Simple view” of data collections



data.frame

- data.frame should be your default receptacle for rectangular, spreadsheet-y data
- Allows holistic management of, for example, subject ID, a quantitative response, and categorical covariates
- a data.frame is accepted by many functions for modeling and graphing via a 'data' argument, allowing you to refer to the constituent variables by variable name and causing various good things to happen automagically (e.g. axis labels)
- data.frame is a very special *list* (in the technical R sense) that also quacks like a matrix ... offers the best of both worlds

Many data analyses revolve around the idea of a dataset, a collection of related values which can be treated as a single unit. For example, you might collect information about different companies; for each company you would have a name, an industry type, the number of employees, type of health care plans offered, etc. For each of the companies you study you would have values for each of these variables. If we store the data in a matrix, with rows representing observations and columns representing variables, it would be easy to access the data, but since the modes of the variables in a dataset will often not be the same, a matrix would force, say, numeric variables to be stored as character variables. To allow the ease of indexing that a matrix would provide while accommodating different modes, R provides the data frame. A data frame is a list with the restriction that each element of the list (the variables) must be of the same length as every other element of the list. Thus, the mode of a data frame is list, and its class is data.frame. While there is some overhead for storing data in a data frame as opposed to a matrix, data frames are the preferred method for working with "observations and variables"-style datasets

from Chapter 1 of
Spector (2008).

To attach() or not attach()? NOT!

- R looks for objects on its search path. You can inspect it with search(). I don't want to go into more detail now.
- attach() puts a 'database' into the search path, where 'database' is typically a data.frame
- Effect: you can refer to 'pop' instead of 'gDat$pop'
- Unfortunately attach() can be seen in otherwise well-written books and documentation. I will charitably assume it's for reasons of space and presentation (?).
- Use of attach() is a really bad idea outside of these highly artificial, static settings.

To attach() or not attach()? NOT!

- This thread on stackoverflow hits the main points re: the bad consequences of using attach().
- Even the clever people at Google don't allow it in their R code.
- Helpful habits and functions for living an attach()-free life
  - Short names for data.frames
  - with(), transform()
  - Use of the data argument in many functions
  - Maybe get better at typing? Seriously.

a few examples of the goodness that comes from ....

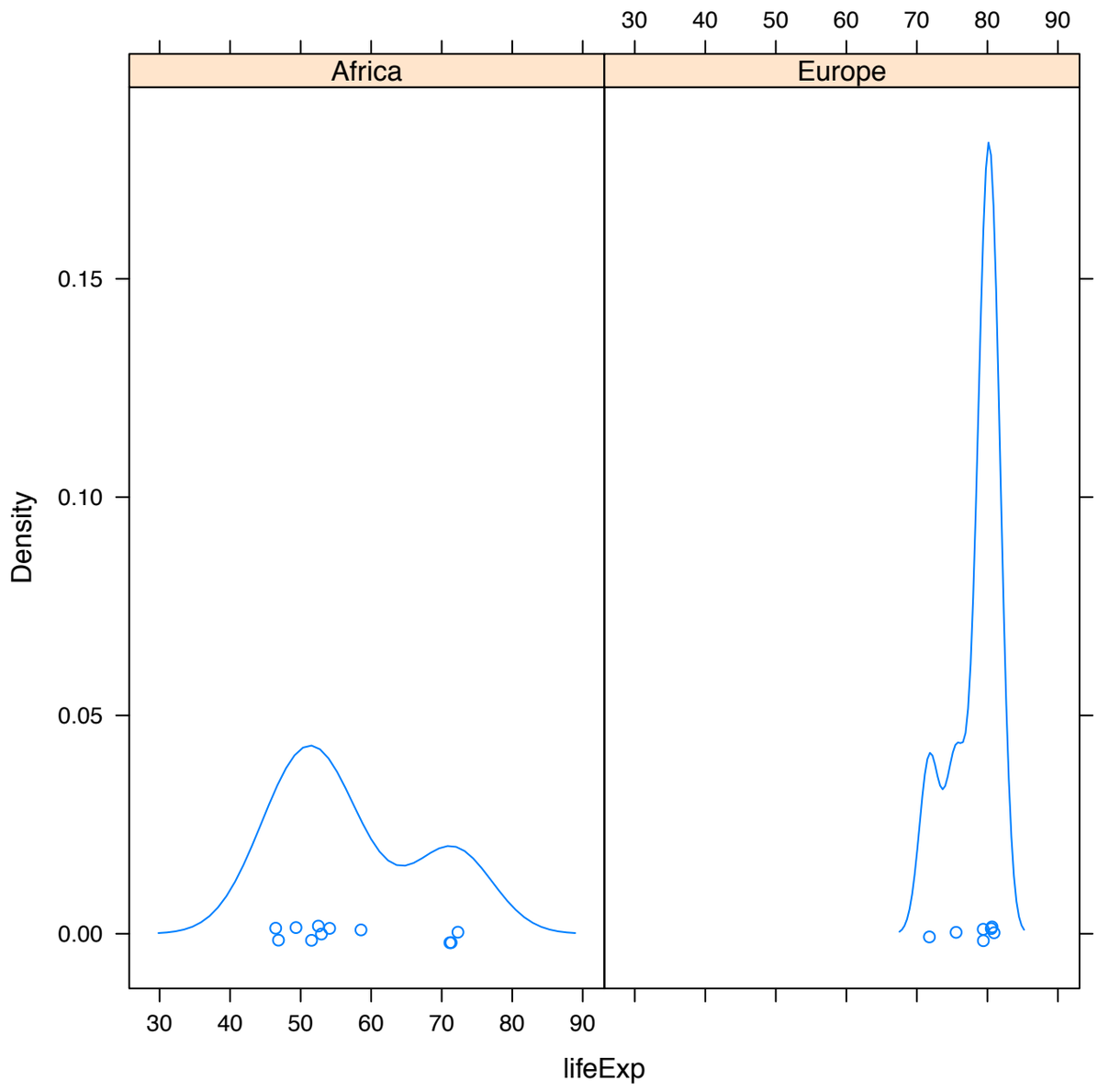
data.frames passed as the local ‘database’ for high-level functions

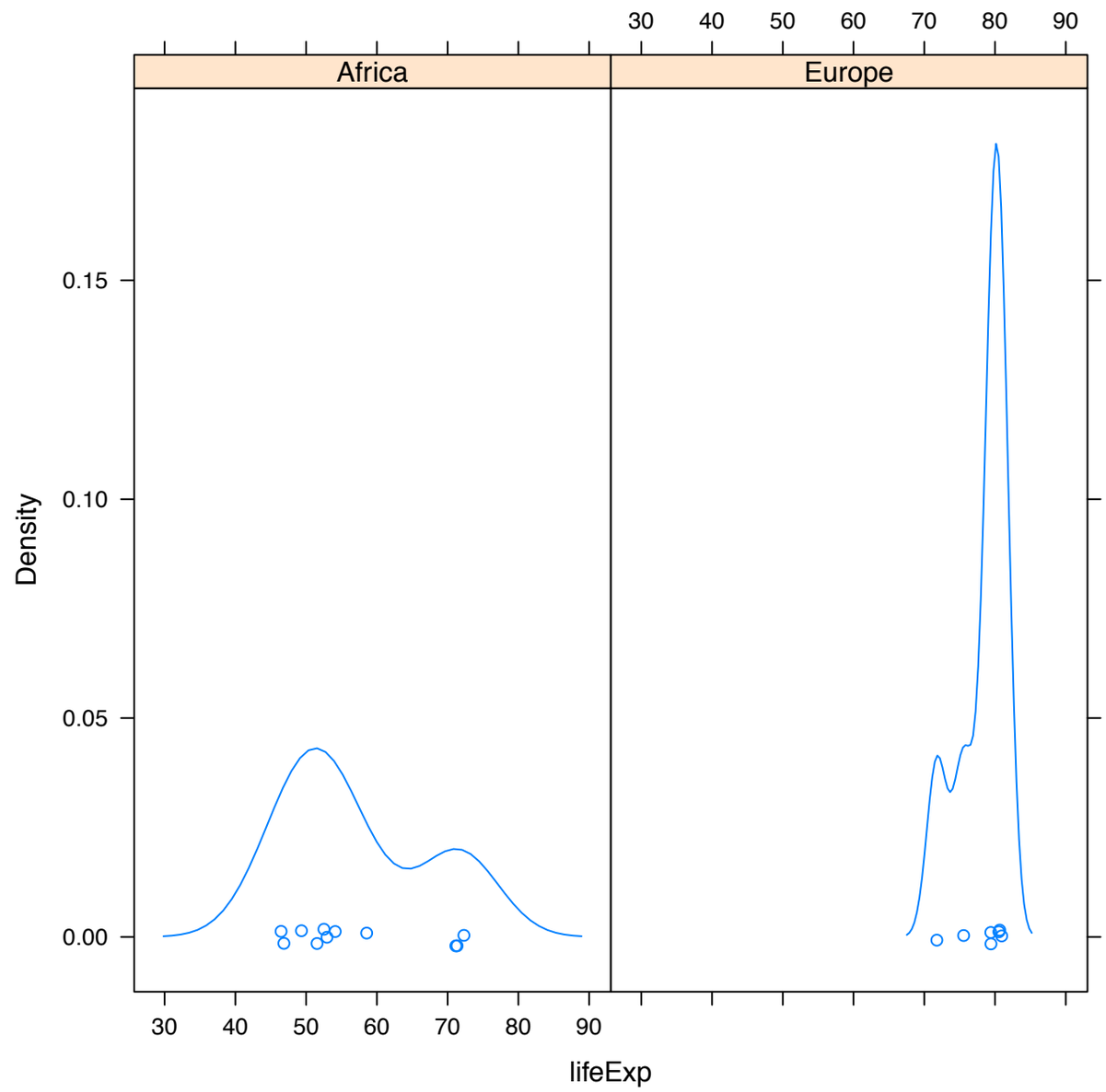
conditional plotting on a factor w/ lattice

the model formula syntax (more on that later)

```
jYear <- 2007
tinyDat <-
  subset(gDat, year == jYear & continent %in% c('Africa', 'Europe'))

densityplot(~ lifeExp | continent, tinyDat)
```



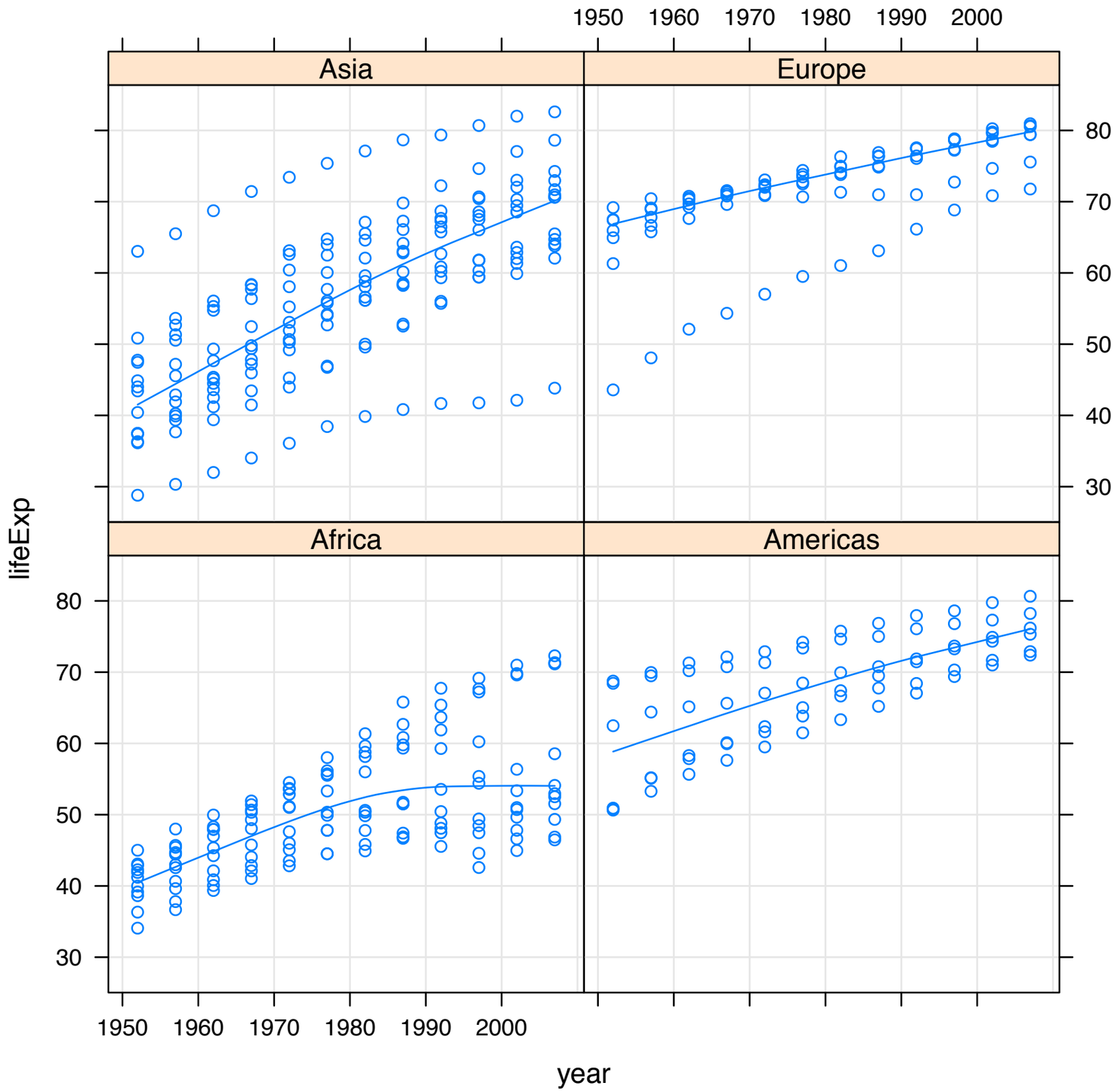


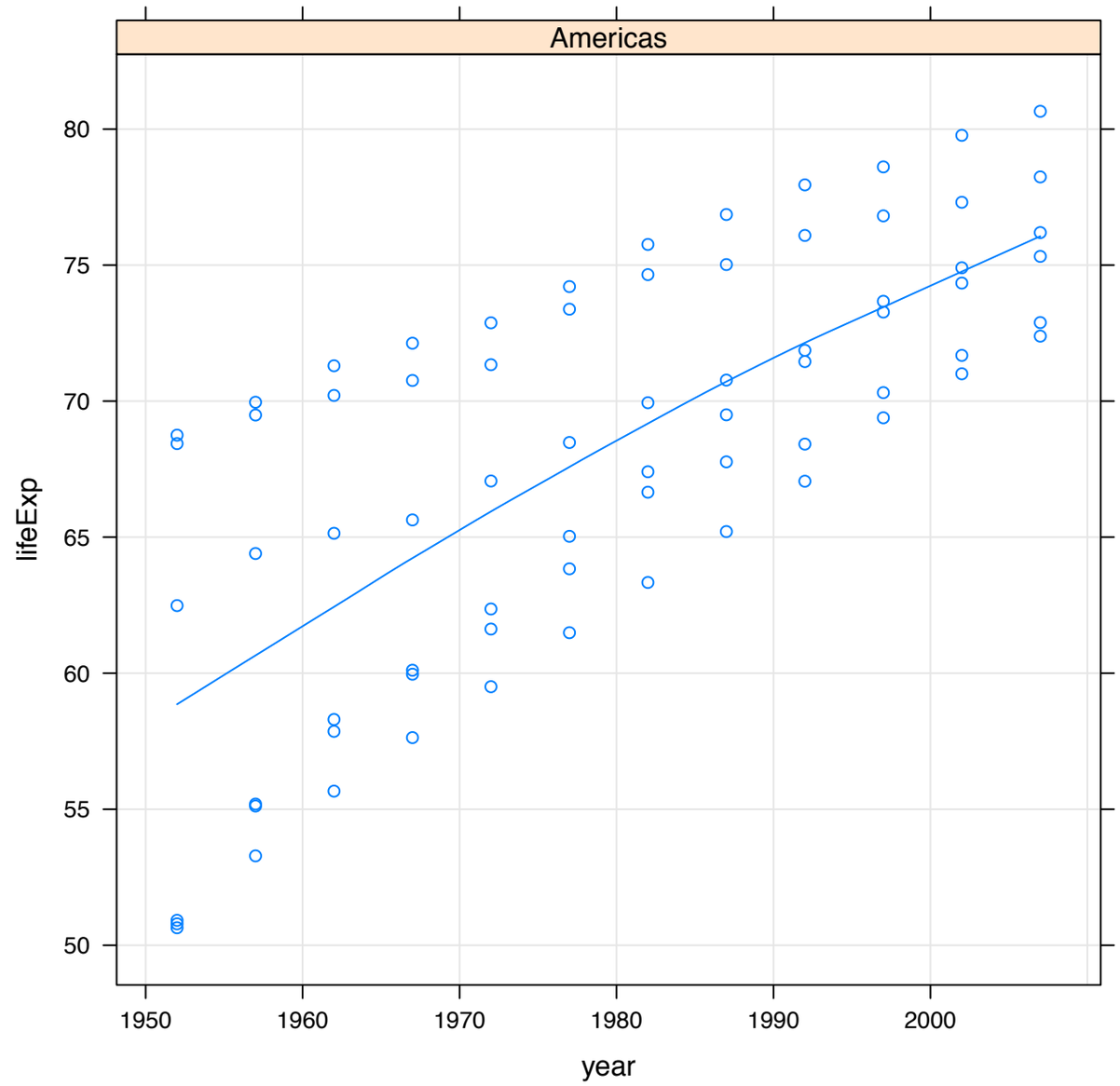

```
> t.test(lifeExp ~ continent, tinyDat)

    Welch Two Sample t-test

data:  lifeExp by continent
t = -6.5267, df = 13.291, p-value = 1.727e-05
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -28.35922 -14.27766
sample estimates:
mean in group Africa mean in group Europe
            57.01227             78.33071
```

```
xyplot(lifeExp ~ year | continent, gDat,
       type = c('p','smooth','g'))
```






```
> jFit <- lm(lifeExp ~ I(year - 1950), gDat,
+            subset = continent == 'Americas')
> summary(jFit)

<snip, snip>

Coefficients:
               Estimate Std. Error t value Pr(>|t|)
(Intercept)    59.03624    1.20834  48.857  < 2e-16 ***
I(year - 1950)  0.30944    0.03535   8.753 7.52e-13 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 5.178 on 70 degrees of freedom
Multiple R-squared: 0.5225,Adjusted R-squared: 0.5157
F-statistic: 76.61 on 1 and 70 DF,  p-value: 7.524e-13
```

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

... to be continued ....