Basic care and feeding of data in R

02 September, 2018

Contents

3.2.2.1 Reading, Homeworks, Projects, SemProjects

- Readings:
  - EDA 1-31
- Homeworks
  -
- Data Science Projects:
  - Project 1 given out Sept. 18th, 2018
  - Due Tuesday October 2, 2018
- 451 SemProjects:
  - Meet during Friday Community Hour
- Friday Comm. Hour
  - 451 students SemProjs

3.2.2.2 Textbooks

- Peng: R Programming for Data Science
- Peng: Exploratory Data Analysis with R
- Open Intro Stats, v3
- Wickham: R for Data Science
- Hastie: Intro to Statistical Learning with R

3.2.2.3 Syllabus

3.2.2.4 Buckle your seatbelt

Ignore if you don't need this bit of support.

Now is the time to make sure you are

Day:Date	Foundation	Practicum	Reading	Due
w1a:Tu:8/28/18	ODS Tool Chain	R, Rstudio, Git		
w1b:Th:8/30/18	Setup ODS Tool Chain	Bash, Git, Twitter	PRP4-33	HW1
w2a:Tu:9/4/18	What is Data Science	OIS:Intro2R	PRP35-64	**HW1 Due**
w2b:Th:9/6/18	Data Analytic Style, Git	Teatime:Intro2R, For loops	PRP65-93	HW2
w3a:Tu:9/11/18*	Struct. of Data Analysis, SemProj	ISLR:Intro2R	PRP94-116	**HW2 Due**
w3b:Th:9/13/18*	OIS3 Intro to Data	GapMinder, Dplyr, Magrittr	OI1-1.9,	
w4a:Tu:9/18/18	OIS3, Intro2Data part 2, Data	EDA: PET Degr.	EDA1-31	Proj1
w4b:Th:9/20/18	Hypothesis Testing	GGPlot2 Tutorial	EDA32-58	HW3
w5a:Tu:9/25/18	Distributions	SemProj RepOut1	R4DS1-3	**HW3 Due**
w5b:Th:9/27/18	Wickham DSCI in Tidyverse	SemProj RepOut1	R4DS4-6	**SemProj1,**
w6a:Tu:10/2/18	OIS Found. of Inference	Inference	R4DS7-8	**Proj1 Due**
w6b:Th:10/4/18		Midterm Review	R4DS9-16 Wrangle	
w7a:Tu:10/9/18*	Summ. Stats & Vis.	Data Wrangling		
w7b:Th:10/11/18*	**MIDTERM EXAM**			HW4
w8a:Tu:10/16/18	Numerical Inference	Tidy Check Explore	OIS4	**HW4 Due**
w8b:Th:10/18/18	Algorithms, Models	Pairwise Corr. Plots	OIS5.1-4	Proj 2, HW5
Tu:10/23	**CWRU FALL BREAK**		R4DS17-21 Program	
w9b:Th:10/25/18	Categorical Infer	Predictive Analytics	OIS6.1,2	
w10a:Tu:10/30/18	SemProj	SemProj	OIS7	**SemProj2 HW5 Du**
w10b:Th:11/1/18	Lin. Regr.	Lin. Regr.	OIS8	**Proj.2 due**
w11a:Tu:11/6/18	Inf. for Regression	Curse of Dim.	OIS8	Proj 3
w11b:Th:11/8/18	Model Accuracy	Training Testing	ISLR3	HW6
w12a:Tu:11/13/18	Multiple Regr.	Mul. Regr. & Pred.	ISLR4	**HW6 due**
w12b:Th:11/15/18	Classification		ISLR6	
w13a:Tu:11/20/18	Classification	Clustering	ISLR5	**Proj 3 due**
Th:11/22/18	**THANKSGIVING**			Proj 4
w14a:Tu:11/27/18	Big Data	Hadoop		
w14b:Th:11/29/18	InfoSec	VerisDB		**SemProj3**
w15a:Tu:12/4/18	SemProj ReportOut3			
w15b:Th:12/6/18	SemProj ReportOut3			**Proj4**
	FINAL EXAM	**Monday12/17, 12:00-3:00pm**	Olin 313	**SemProj4 due**

Figure 1: DSCI351-451 Syllabus

- working in an appropriate directory on your computer,
  - probably through the use of an RStudio Project.
- Enter `getwd()` in the Console to see current working directory
  - or, in RStudio, this is displayed in the bar at the top of Console.

You should clean out your workspace.

- In RStudio, click on the "Clear" broom icon from the Environment tab
  - or use Session > Clear Workspace.
- You can also enter `rm(list = ls())` in the Console to accomplish same.

Now restart R.

- This will ensure you don't have any packages loaded
  - from previous calls to `library()`.
- In RStudio, use Session > Restart R.
  - Otherwise, quit R with `q()` and re-launch it.

Why do we do this?

- So that the code you write is complete and re-runnable.
- If you return to a clean slate often,
  - you will root out hidden dependencies where one snippet of code only works
  - because it relies on objects created by code
    - saved elsewhere or, much worse, never saved at all.
- Similary, an aggressive clean slate approach
  - will expose any usage of packages that have not been explicitly loaded.

Finally, open a new R script

- and develop and run your code from there.

In RStudio, use File > New File > R Script.

- Save this script with a name ending in `.r` or `.R`,
- containing no spaces or other funny stuff,
- and that evokes whatever it is we're doing today.
  - Example: `session03_data-aggregation.r`.

3.2.2.5 Get the Gapminder data

What is Gapminder

- A project of Hans Rosling
- Gapminder Project

We will work with some of the data from the Gapminder project.

- Here is an excerpt prepared for your use.
- Please save this file locally,
  - for example, in the data directory associated with your RStudio Project:
- http://www.stat.ubc.ca/~jenny/notOcto/STAT545A/examples/gapminder/data/gapminderDataFiveYear.txt

You should now have a plain text file

- called `gapminderDataFiveYear.txt` on your computer,
- in your working directory. Do this to confirm:

```
list.files()
```

If you **don't** see `gapminderDataFiveYear.txt` there,

* DEAL WITH THAT BEFORE YOU MOVE ON.

3.2.2.5.1 Create a data.frame via import

In real life you will usually bring data into R from an outside file.

* This rarely goes smoothly for "wild caught" datasets,
  - which have little gremlins lurking in them
  - that complicate import and require cleaning.
* Since this is not our focus today,
  - we will work with a "domesticated" dataset JB uses a lot in teaching,
  - an extract from the Gapminder data Hans Rosling has popularized.

  Assumption: The file `gapminderDataFiveYear.txt` is saved on your computer and available for reading in R's current working directory.

Bring the data into R.

* Note that RStudio's tab completion facilities can help you with filenames,
* as well as function and object names. Try it out!

```
gDat <- read.delim("./data/gapminderDataFiveYear.txt")
```

One can also read data directly from a URL,

* though this is more of a party trick than a great general strategy.

```
## data import from URL
gdURL <- "http://www.stat.ubc.ca/~jenny/notOcto/STAT545A/examples/gapminder/data/gapminderDataFiveYear.
# gdURL <- "http://tiny.cc/gapminder"
gDat <- read.delim(file = gdURL)
```

3.2.2.5.2 The R function `read.table()`

* is the main workhorse for importing rectangular spreadsheet-y data into an R data.frame.
* Use it. Read the documentation.
* There you will learn about handy wrappers around `read.table()`,
  - such as `read.delim()`
  - where many arguments have been preset to anticipate some common file formats.
* Competent use of the many arguments of `read.table()`
  - can eliminate a very great deal of agony and post-import fussing around.

Whenever you have rectangular, spreadsheet-y data,

* your default data receptacle in R is a data.frame.
* Do not depart from this without good reason.

3.2.2.5.3 Data.frames are awesome because...

* data.frames package related variables neatly together,
  - keeping them in sync vis-a-vis row order
  - applying any filtering of observations uniformly
* most functions for inference, modelling, and graphing
  - are happy to be passed a data.frame via a `data =` argument
  - as the place to find the variables you're working on;
  - the latest and greatest packages
    * actually **require** that your data be in a data.frame
* data.frames – unlike general arrays or, specifically, matrices in R –

    - can hold variables of different flavors (heuristic term defined later),
    - such as character data (subject ID or name),
        * quantitative data (white blood cell count),
        * and categorical information (treated vs. untreated)

3.2.2.5.4 Get an overview of the object we just created with `str()`

- which displays the structure of an object.
- It will provide a sensible description of almost anything
    - and, worst case, nothing bad can actually happen.
- When in doubt, just `str()`
    - some of the recently created objects to get some ideas about what to do next.

```
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 could print the whole thing to screen

- (not so useful with datasets of any size)
- but it's nicer to look at the first bit or the last bit or a random snippet
    - (I've written a function `peek()` to look at some random rows).

```
head(gDat)
##       country year      pop continent lifeExp gdpPercap
## 1 Afghanistan 1952  8425333      Asia  28.801  779.4453
## 2 Afghanistan 1957  9240934      Asia  30.332  820.8530
## 3 Afghanistan 1962 10267083      Asia  31.997  853.1007
## 4 Afghanistan 1967 11537966      Asia  34.020  836.1971
## 5 Afghanistan 1972 13079460      Asia  36.088  739.9811
## 6 Afghanistan 1977 14880372      Asia  38.438  786.1134
tail(gDat)
##       country year      pop continent lifeExp gdpPercap
## 1699 Zimbabwe 1982  7636524    Africa  60.363  788.8550
## 1700 Zimbabwe 1987  9216418    Africa  62.351  706.1573
## 1701 Zimbabwe 1992 10704340    Africa  60.377  693.4208
## 1702 Zimbabwe 1997 11404948    Africa  46.809  792.4500
## 1703 Zimbabwe 2002 11926563    Africa  39.989  672.0386
## 1704 Zimbabwe 2007 12311143    Africa  43.487  469.7093
#peek(gDat) # you won't have this function!
```

JB note to self: possible sidebar constructing `peek()` here

More ways to query basic info on a data.frame.

- Note: with some of the commands below we're benefitting from the fact that
    - even though data.frames are technically NOT matrices,
    - it's usually fine to think of them that way
- and many functions have reasonable methods for both types of input.

```
names(gDat)# variable or column names
## [1] "country"   "year"      "pop"       "continent" "lifeExp"   "gdpPercap"
```

```r
ncol(gDat)
## [1] 6
length(gDat)
## [1] 6
head(rownames(gDat)) # boring, in this case
## [1] "1" "2" "3" "4" "5" "6"
dim(gDat)
## [1] 1704    6
nrow(gDat)
## [1] 1704
#dimnames(gDat) # ill-advised here ... too many rows
```

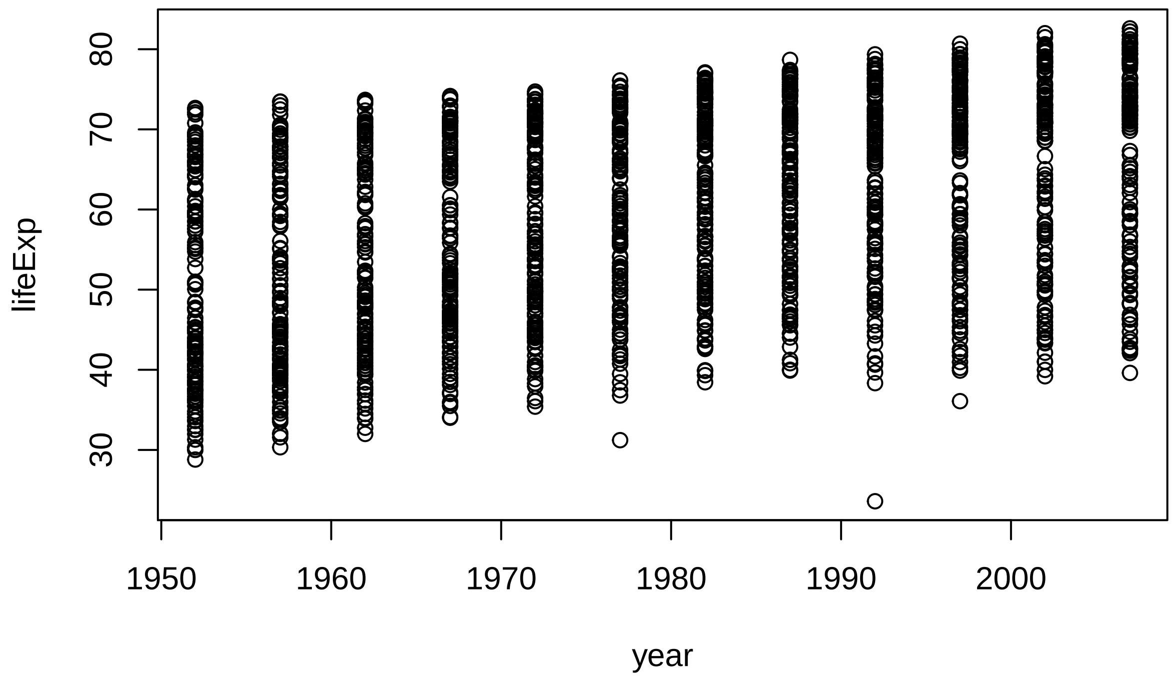
3.2.2.5.5 A statistical overview can be obtained with summary()

```r
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
##
```

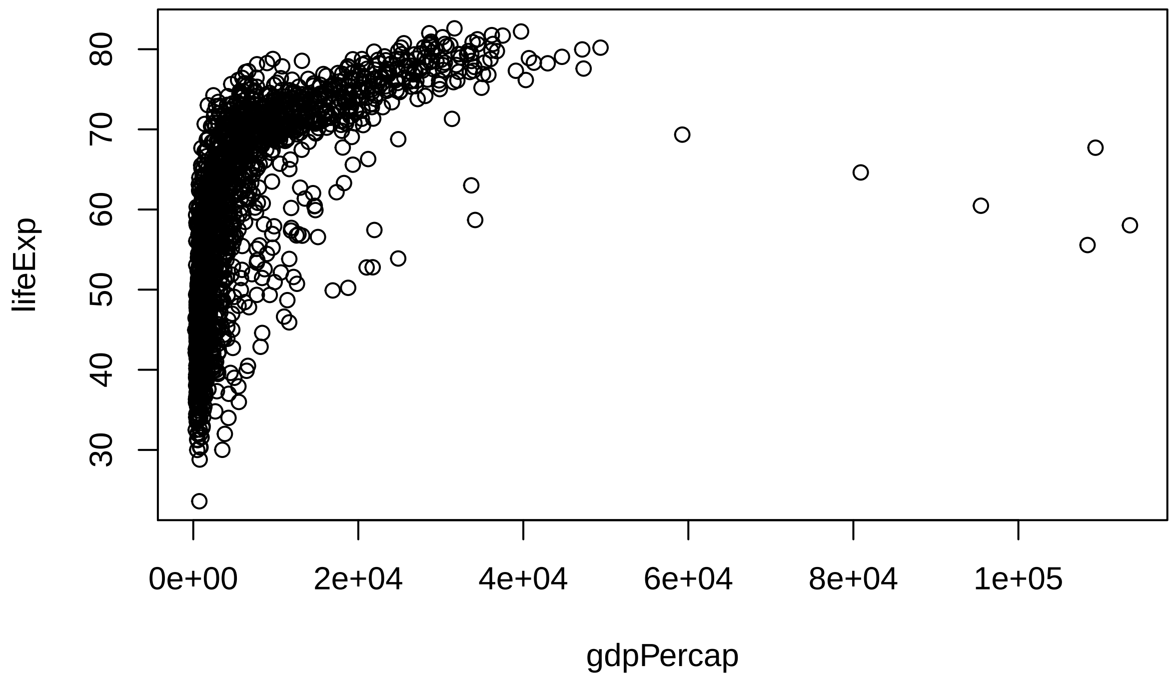
Although we haven't begun our formal coverage of visualization yet,

- it's so important for smell-testing dataset
  - that we will make a few figures anyway.
- Here we use only base R graphics, which are very basic.

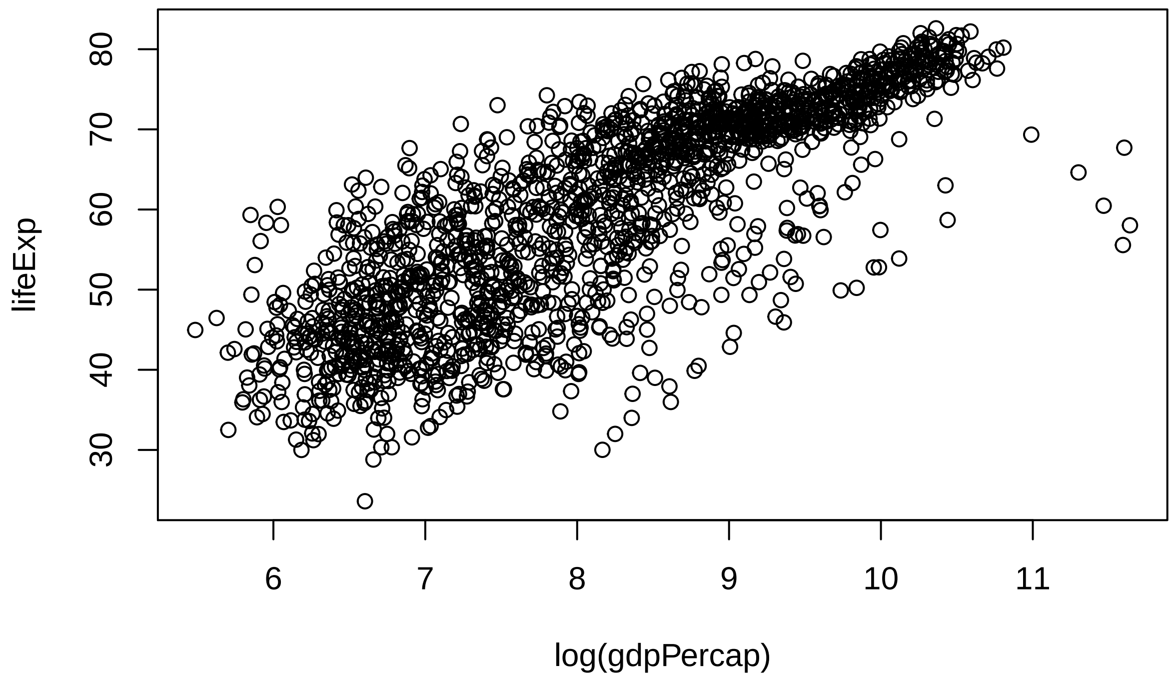
```r
plot(lifeExp ~ year, gDat)
```




```r
plot(lifeExp ~ gdpPercap, gDat)
```




```r
plot(lifeExp ~ log(gdpPercap), gDat)
```



Let's go back to the result of `str()` to talk about data.frames and vectors in R

```
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 ...
```

A data.frame is a special case of a *list*,

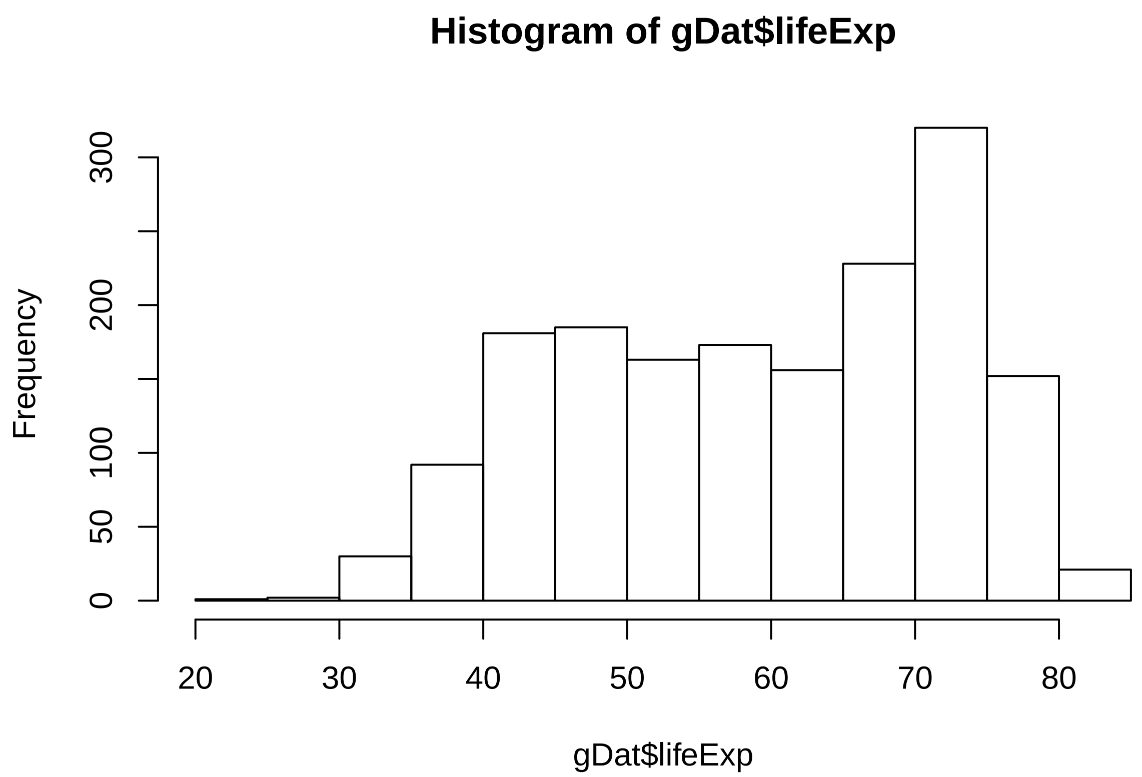
- which is used in R to hold just about anything.
- data.frames are the special case where
  - the length of each list component is the same.
- data.frames are superior to matrices in R
  - because they can hold vectors of different flavors
  - (heuristic term explained below),
- e.g. numeric, character, and categorical data can be stored together.
  - This comes up alot.

3.2.2.6 Look at the variables inside a data.frame

To specify a single variable from a data.frame,

- use the dollar sign `$`.
- Let's explore the numeric variable for life expectancy.

```
head(gDat$lifeExp)
## [1] 28.801 30.332 31.997 34.020 36.088 38.438
summary(gDat$lifeExp)
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
##   23.60   48.20   60.71   59.47   70.85   82.60
hist(gDat$lifeExp)
```



The year variable is a numeric integer variable,

- but since there are so few unique values
- it also functions a bit like a categorical variable.

```
summary(gDat$year)
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
##    1952    1966    1980    1980    1993    2007
table(gDat$year)
## 
## 1952 1957 1962 1967 1972 1977 1982 1987 1992 1997 2002 2007 
##  142  142  142  142  142  142  142  142  142  142  142  142
```

The variables for country and continent

- hold truly categorical information,
- which is stored as a *factor* in R.

```
class(gDat$continent)
## [1] "factor"
summary(gDat$continent)
```

```
##   Africa Americas     Asia   Europe  Oceania
##      624      300      396      360       24
levels(gDat$continent)
## [1] "Africa"   "Americas" "Asia"     "Europe"   "Oceania"
nlevels(gDat$continent)
## [1] 5
```

The **levels** of the factor `continent`

- are "Africa", "Americas", etc. and
  - this is what's usually presented to your eyeballs by R.
- In general, the levels are friendly human-readable character strings,
  - like "male/female" and "control/treated".
- But never ever ever forget that, under the hood,
  - R is really storing integer codes 1, 2, 3, etc.
- Look at the result from `str(gDat$continent)` if you are skeptical.

```
str(gDat$continent)
##  Factor w/ 5 levels "Africa","Americas",..: 3 3 3 3 3 3 3 3 3 3 ...
```

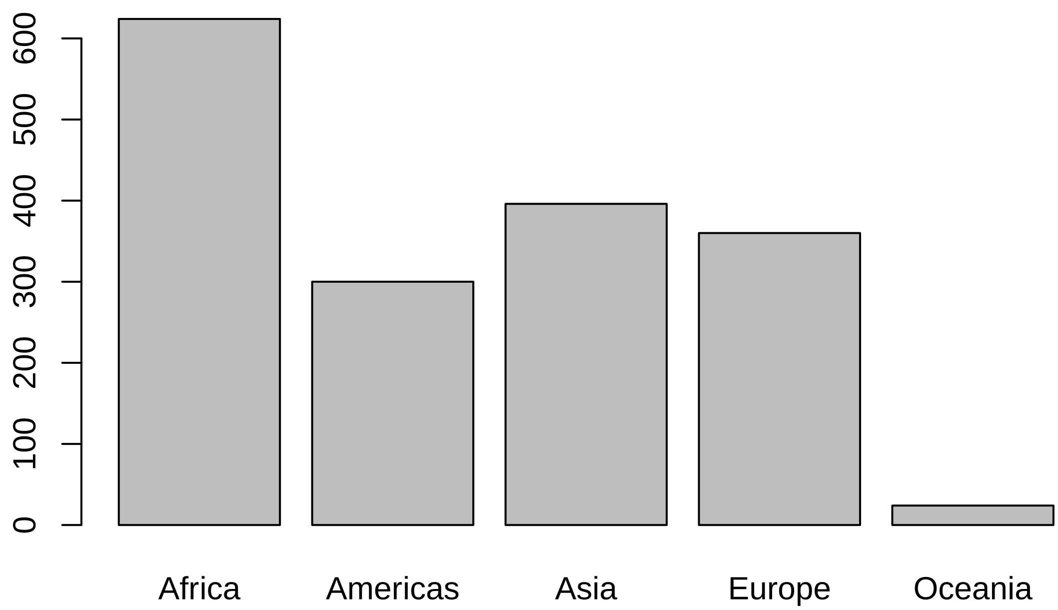
3.2.2.6.1 This Janus-like nature of factors

- means they are rich with booby traps for the unsuspecting
  - but they are a necessary evil.
- I recommend you resolve to learn how to properly care and feed for factors.
  - The pros far outweigh the cons.
- Specifically in modelling and figure-making,
  - factors are anticipated and accomodated
  - by the functions and packages you will want to exploit.

Here we count how many observations

- are associated with each continent
  - and, as usual, try to portray that info visually.
- This makes it much easier to quickly see that African countries
  - are well represented in this dataset.

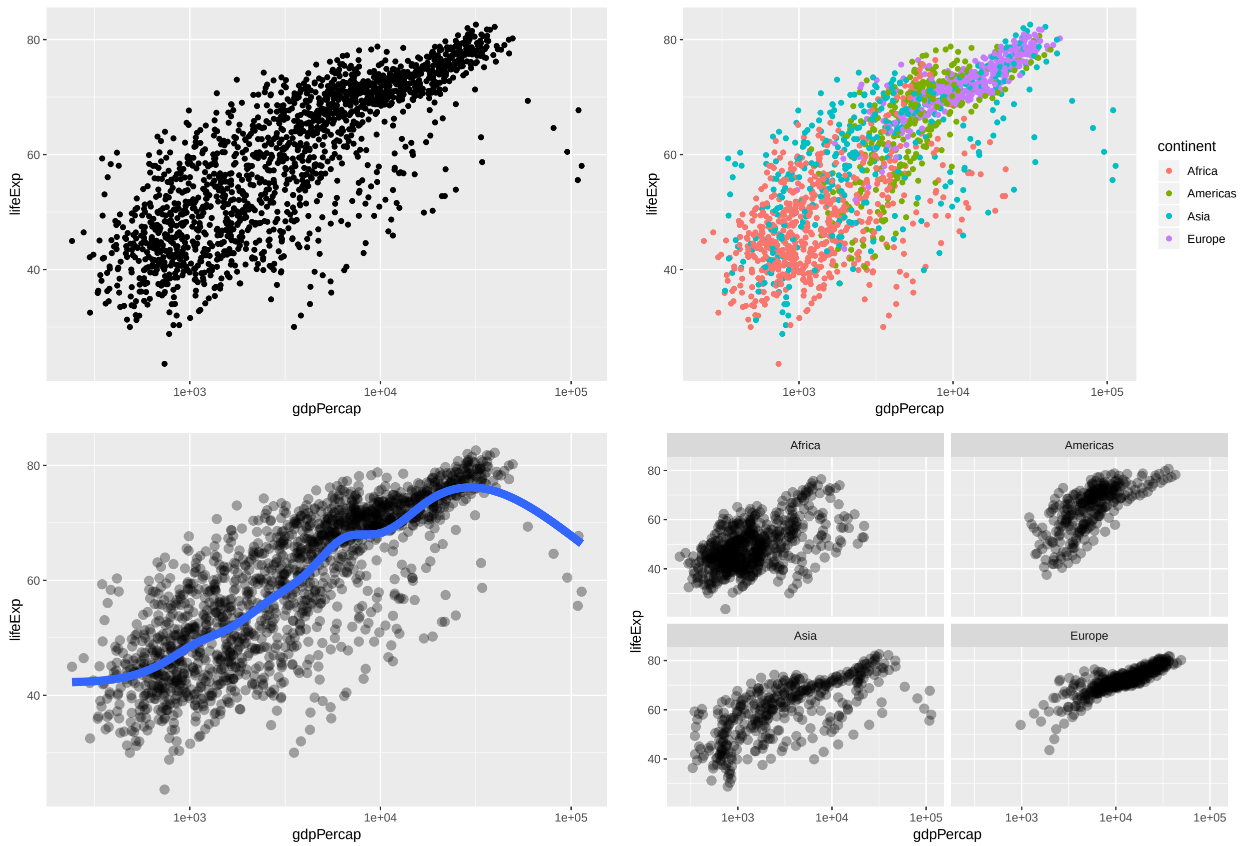
```
table(gDat$continent)
##
##   Africa Americas     Asia   Europe  Oceania
##      624      300      396      360       24
barplot(table(gDat$continent))
```



In the figures below, we see how factors can be put to work in figures.

- The `continent` factor is easily
    - mapped into "facets" or colors and a legend by the `ggplot2` package.
- Making figures with `ggplot2` is covered elsewhere
    - so feel free to just sit back and enjoy these plots
    - or blindly copy/paste.

```
## install ggplot2 if you don't have it!
## install.packages(ggplot2)
library(ggplot2)
p <- ggplot(subset(gDat, continent != "Oceania"),
            aes(x = gdpPercap, y = lifeExp)) # just initializes
p <- p + scale_x_log10() # log the x axis the right way
p + geom_point() # scatterplot
p + geom_point(aes(color = continent)) # map continent to color
p + geom_point(alpha = (1/3), size = 3) + geom_smooth(lwd = 3, se = FALSE)
## `geom_smooth()` using method = 'gam' and formula 'y ~ s(x, bs = "cs")'
p + geom_point(alpha = (1/3), size = 3) + facet_wrap(~ continent)
```



3.2.2.7 `subset()` is a nice way to isolate bits of data.frames (and other things)

Logical little pieces of data.frames are useful for sanity checking,

- prototyping visualizations or computations for later scale-up, etc.

Many functions are happy to restrict their operations

- to a subset of observations via a formal `subset =` argument.

There is a stand-alone function, also confusingly called `subset()`,

- that can isolate pieces of an object for inspection or assignment.
- Although `subset()` can work on objects other than data.frames,
  - we focus on that usage here.

The `subset()` function has a `subset =` argument

- (sorry, not my fault it's so confusing)
- or specifying which observations to keep.

This expression will be evaluated within the specified data.frame,

- which is non-standard but convenient.

```
subset(gDat, subset = country == "Uruguay")
##      country year     pop continent lifeExp gdpPercap
## 1621 Uruguay 1952 2252965  Americas  66.071  5716.767
## 1622 Uruguay 1957 2424959  Americas  67.044  6150.773
## 1623 Uruguay 1962 2598466  Americas  68.253  5603.358
## 1624 Uruguay 1967 2748579  Americas  68.468  5444.620
```

```
## 1625 Uruguay 1972 2829526  Americas  68.673  5703.409
## 1626 Uruguay 1977 2873520  Americas  69.481  6504.340
## 1627 Uruguay 1982 2953997  Americas  70.805  6920.223
## 1628 Uruguay 1987 3045153  Americas  71.918  7452.399
## 1629 Uruguay 1992 3149262  Americas  72.752  8137.005
## 1630 Uruguay 1997 3262838  Americas  74.223  9230.241
## 1631 Uruguay 2002 3363085  Americas  75.307  7727.002
## 1632 Uruguay 2007 3447496  Americas  76.384 10611.463
```

Contrast the above command

- with this one accomplishing the same thing:

```
gDat[1621:1632, ]
##      country year     pop continent lifeExp gdpPercap
## 1621 Uruguay 1952 2252965  Americas  66.071  5716.767
## 1622 Uruguay 1957 2424959  Americas  67.044  6150.773
## 1623 Uruguay 1962 2598466  Americas  68.253  5603.358
## 1624 Uruguay 1967 2748579  Americas  68.468  5444.620
## 1625 Uruguay 1972 2829526  Americas  68.673  5703.409
## 1626 Uruguay 1977 2873520  Americas  69.481  6504.340
## 1627 Uruguay 1982 2953997  Americas  70.805  6920.223
## 1628 Uruguay 1987 3045153  Americas  71.918  7452.399
## 1629 Uruguay 1992 3149262  Americas  72.752  8137.005
## 1630 Uruguay 1997 3262838  Americas  74.223  9230.241
## 1631 Uruguay 2002 3363085  Americas  75.307  7727.002
## 1632 Uruguay 2007 3447496  Americas  76.384 10611.463
```

Yes, these both return the same result.

But the second command is horrible for these reasons:

- It contains Magic Numbers.
  - The reason for keeping rows 1621 to 1632 will be non-obvious
  - to someone else and that includes **you** in a couple of weeks.
- It is fragile.
  - If the rows of `gDat` are reordered
  - or if some observations are eliminated,
  - these rows may no longer correspond to the Uruguay data.

In contrast, the first command, using `subset()`,

- is self-documenting;
  - one does not need to be an R expert
  - to take a pretty good guess at what's happening.
- It's also more robust.
  - It will still produce the correct result
  - even if `gDat` has undergone some reasonable set of transformations.

The `subset()` function can also be used

- to select certain variables via the `select` argument.
- It also offers unusual flexibility,
  - so you can, for example,
  - provide the names of variables you wish to keep
  - without surrounding by quotes.
- I suppose this is mostly a good thing,
  - but even the documentation stresses that the `subset()` function

    - is intended for interactive use
    - (which I interpret more broadly to mean data analysis, as opposed to programming).

You can use `subset =` and `select =` together

- to simultaneously filter rows and columns or variables.

```
subset(gDat, subset = country == "Mexico",
       select = c(country, year, lifeExp))
##     country year lifeExp
## 985  Mexico 1952  50.789
## 986  Mexico 1957  55.190
## 987  Mexico 1962  58.299
## 988  Mexico 1967  60.110
## 989  Mexico 1972  62.361
## 990  Mexico 1977  65.032
## 991  Mexico 1982  67.405
## 992  Mexico 1987  69.498
## 993  Mexico 1992  71.455
## 994  Mexico 1997  73.670
## 995  Mexico 2002  74.902
## 996  Mexico 2007  76.195
```

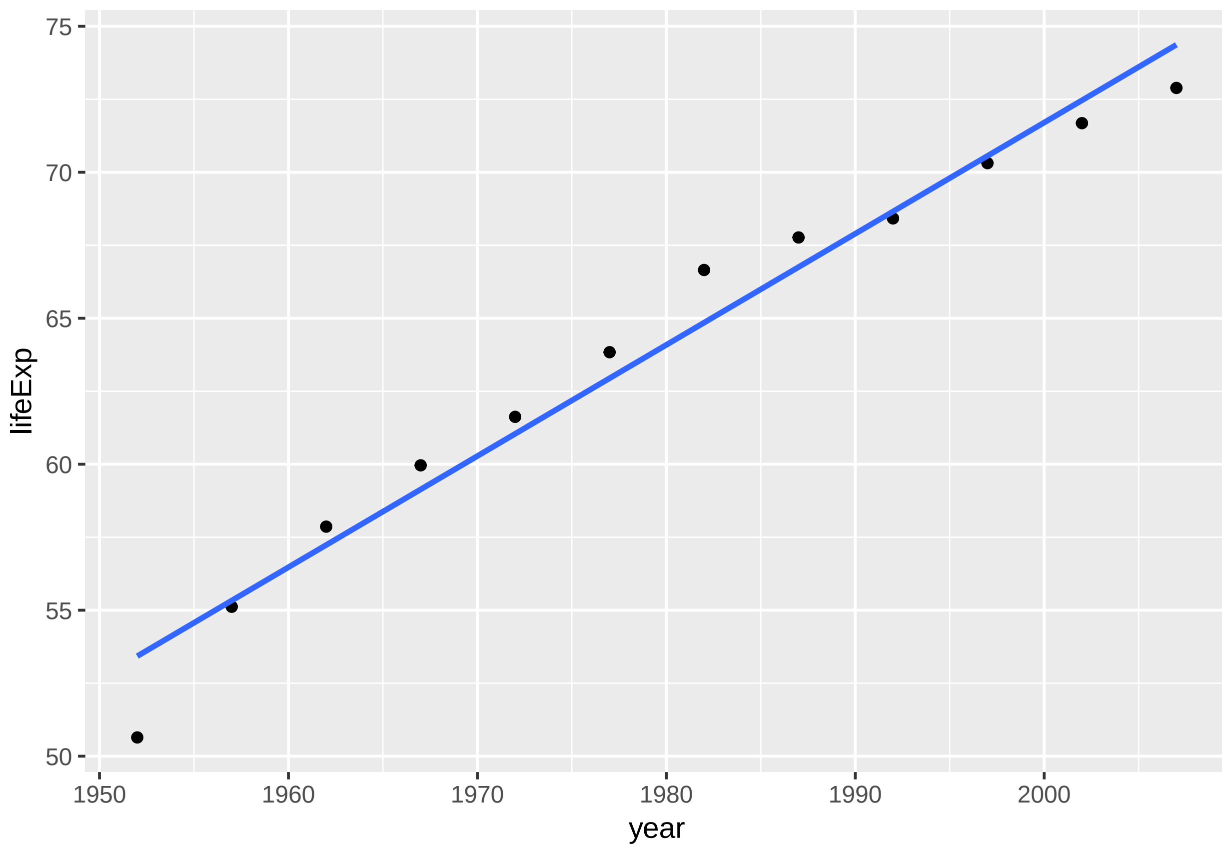
3.2.2.8 Many of the functions for inference, modelling, and graphics

- that permit you to specify a data.frame via`data =`
- also offer a `subset =` argument
    - that limits the computation to certain observations.

Here's an example of subsetting the data

- to make a plot just for Colombia
- and a similar call to `lm`
    - for fitting a linear model to just the data from Colombia.

```
p <- ggplot(subset(gDat, country == "Colombia"), aes(x = year, y = lifeExp))
p + geom_point() + geom_smooth(lwd = 1, se = FALSE, method = "lm")
```




```r
(minYear <- min(gDat$year))
## [1] 1952
myFit <- lm(lifeExp ~ I(year - minYear), data = gDat, subset = country == "Colombia")
summary(myFit)
##
## Call:
## lm(formula = lifeExp ~ I(year - minYear), data = gDat, subset = country ==
##     "Colombia")
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -2.7841 -0.3816  0.1840  0.8413  1.8034
##
## Coefficients:
##                   Estimate Std. Error t value Pr(>|t|)
## (Intercept)       53.42712    0.71223   75.01 4.33e-15 ***
## I(year - minYear)  0.38075    0.02194   17.36 8.54e-09 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.312 on 10 degrees of freedom
## Multiple R-squared:  0.9679, Adjusted R-squared:  0.9647
## F-statistic: 301.3 on 1 and 10 DF,  p-value: 8.537e-09
```

3.2.2.9 Review of data.frames and the best ways to exploit them

Use data.frames!!!

The most modern, slick way to work with data.frame

- is with `dplyr`. We'll focus on this in the R for Data Science book.:
- Introduction to dplyr
- `dplyr` functions for a single dataset

Work within your data.frames

- by passing them to the `data =` argument of functions that offer that.

If you need to restrict operations,

- use the `subset =` argument.

Do computations or make figures *in situ*

- don't create little copies and excerpts of your data.
- This will leave a cleaner workspace and cleaner code.

This workstyle leaves behind code

- that is also fairly self-documenting, e.g.,

```
lm(lifeExp ~ year, gDat, subset = country == "Colombia")
plot(lifeExp ~ year, gDat, subset = country == "Colombia")
```

The availability and handling of `data =` and `subset =` arguments

- is broad enough– though sadly not universal
- that sometimes you can even copy and paste these argument specifications,
  - for example, from an exploratory plotting command
  - into a model-fitting command.
- Consistent use of this convention
  - also makes you faster at writing and reading such code.

Two important practices

- give variables short informative names (`lifeExp` versus "X5")
- refer to variables by name, not by column number

This will produce code that is self-documenting and more robust.

- Variable names often propagate to downstream outputs
- like figures and numerical tables
  - and therefore good names have a positive multiplier effect
- throughout an analysis.

If a function doesn't have a `data =` argument

- where you can provide a data.frame,
- you can fake it with `with()`.
  - `with()` helps you avoid the creation of
    * temporary, confusing little partial copies of your data.
- Use it – possibly in combination with `subset()` –
  - to do specific computations
    * without creating all the intermediate temporary objects
    * you have no lasting interest in.
- `with()` is also useful if you are tempted to use `attach()`
  - in order to save some typing.
- **Never ever use `attach()`. It is evil.**

    - If you've never heard of it, consider yourself lucky.

Example: How would you compute the correlation of

- life expectancy and GDP per capita for the country of Colombia?
- The `cor()` function sadly does not offer
    - the usual `data =` and `subset =` arguments.
- Here's a nice way to combine `with()` and `subset()`
    - to accomplish without unnecessary object creation
    - and with fairly readable code.

```
with(subset(gDat, subset = country == "Colombia"),
     cor(lifeExp, gdpPercap))
## [1] 0.9514699
```

3.2.2.10 Links

Jenny Bryan Stat 545