

# CWRU DSCI353-353M-453: Week02a Tidyverse Review

Profs: R. H. French, L. S. Bruckman, P. Leu, K. Davis, S. Cirlos

TAs: W. Oltjen, K. Hernandez, M. Li, M. Li, D. Colvin

## Contents

2.2.4.1	Tidyverse Cheatsheets, Functions and Reading Your Code . . . . .	1
2.2.4.2	What is a Tidy Data Frame . . . . .	2
2.2.4.2.1	What is data wrangling? Intro, Motivation, Outline, Setup . . . . .	2
2.2.4.2.2	Buckle your seat belt . . . . .	2
2.2.4.2.3	Data frames are awesome . . . . .	3
2.2.4.3	Get the Gapminder data . . . . .	4
2.2.4.4	Non-sequitur: The Equals Operator . . . . .	8
2.2.4.5	Look at the variables inside a data frame . . . . .	9
2.2.4.6	Recap . . . . .	12
2.2.4.7	Tidy Manipulation: Introduction to dplyr package . . . . .	13
2.2.4.7.1	Intro . . . . .	13
2.2.4.7.2	Think before you create excerpts of your data ... . . . .	14
2.2.4.7.3	Use <code>filter()</code> to subset data row-wise. . . . .	15
2.2.4.7.4	Meet the new pipe operator . . . . .	16
2.2.4.7.5	Use <code>select()</code> to subset the data on variables or columns. . . . .	17
2.2.4.7.6	Revel in the convenience . . . . .	18
2.2.4.7.7	Pure, predictable, pipe able . . . . .	18
2.2.4.7.8	Resources . . . . .	19
2.2.4.7.9	Load dplyr and gapminder . . . . .	20
2.2.4.7.10	Create a copy of gapminder . . . . .	20
2.2.4.7.11	Use <code>mutate()</code> to add new variables . . . . .	20
2.2.4.7.12	Use <code>arrange()</code> to row-order data in a principled way . . . . .	22
2.2.4.7.13	Use <code>rename()</code> to rename variables . . . . .	23
2.2.4.7.14	<code>select()</code> can rename and reposition variables . . . . .	24
2.2.4.7.15	<code>group_by()</code> is a mighty weapon . . . . .	24
2.2.4.7.16	Grouped mutate . . . . .	28
2.2.4.7.17	Grand Finale . . . . .	31
2.2.4.7.18	Resources . . . . .	33
2.2.4.7.19	References . . . . .	33
2.2.4.8	Links . . . . .	33

**2.2.4.1 Tidyverse Cheatsheets, Functions and Reading Your Code** Look at the Tidyverse Cheatsheet

- **Tidyverse For Beginners Cheatsheet**
  - In the Git/20s-dsci353-353m-453-prof/3-readings/3-CheatSheets/ folder
- **Data Wrangling with dplyr and tidyr Cheatsheet**

Tidyverse Functions & Conventions

– The pipe operator ``%>%``

- Use `dplyr::filter()` to subset data row-wise.
- Use `dplyr::arrange()` to sort the observations in a data frame
- Use `dplyr::mutate()` to update or create new columns of a data frame
- Use `dplyr::summarize()` to turn many observations into a single data point
- Use `dplyr::arrange()` to change the ordering of the rows of a data frame
- Use `dplyr::select()` to choose variables from a tibble,
  - keeps only variables you mention
- Use `dplyr::rename()` keeps all the variables and renames variables
  - `rename(iris, petal_length = Petal.Length)`
- These can be combined using `dplyr::group_by()`
  - which lets you perform operations "by group".
- The `%in%` matches conditions provided by a vector using the `c()` function
- The `**forcats**` package has tidyverse functions
  - for factors (categorical variables)
- The `**readr**` package has tidyverse functions
  - to read..., melt..., col..., parse... data and objects

Reading Your Code: Whenever you see

- The assignment operator `<-`, think “gets”
- The pipe operator, `%>%`, think “then”

```
library(devtools)
```

## 2.2.4.2 What is a Tidy Data Frame

```
## Loading required package: usethis
```

```
# devtools::install_github("rstudio/EDAWR")
```

### 2.2.4.2.1 What is data wrangling? Intro, Motivation, Outline, Setup

- [Pt. 1 Data Wrangling Introduction](#)
  - Tibbles
  - View
  - Pipe Operator
- [Pt 2 Intro to Data Wrangling with R and the Tidyverse](#)
  - What is a Tidy Dataframe?
  - tidy package for gather and spread
- [dplyr – Pt 3 Intro to the Grammar of Data Manipulation with R](#)
- [\[Working with Two Datasets: Binds, Set Operations, and Joins\]](#)
- [Pt 4 Intro to Data Manipulation\]\(https://youtu.be/AuBgYDCg1Cg?list=WL\)](#)

### 2.2.4.2.2 Buckle your seat belt

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

Now is the time to make sure

- you are working in an appropriate directory on your computer,
- probably through the use of an [RStudio Project](#).

To see where you are

- 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 work space.

- In RStudio, click on the “Clear” broom icon from the Environment tab or
- use *Session > Clear Work space*.
- 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.
- Similarly, 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: `cm004_data-care-feeding.r`.

Another great idea is to do this in an R Markdown document.

#### 2.2.4.2.3 Data frames are awesome

- 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.

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, modeling, and graphing
  - are happy to be passed a data frame via a `data =` argument.
  - This has been true in base R for a long time.
- The set of packages known as the [tidyverse](#)
  - takes this one step further
  - and explicitly prioritizes the processing of data frames.
- This includes popular packages like `dplyr` and `ggplot2`.
- In fact the tidyverse prioritizes
  - a special flavor of data frame, called a “tibble.”

Data frames

- unlike general arrays or, specifically, matrices in R
- can hold variables of different flavors,
  - such as character data (subject ID or name),

- quantitative data (white blood cell count),
  - and categorical information (treated vs. untreated).
- If you use homogeneous structures,
  - like matrices,
  - for data analysis,
  - you are likely to make the terrible mistake
  - of spreading a data set out over multiple, unlinked objects.
- Why? Because you can't put character data,
  - such as subject name,
  - into the numeric matrix that holds white blood cell count.
- This fragmentation is a Bad Idea.

### 2.2.4.3 Get the Gapminder data

- What is Gapminder
  - A project of Hans Rosling
  - [Gapminder Project](#)

[Hans Rosling and Gapminder: 200 years in 4 minutes - BBC News](#)

We will work with some of the data from the [Gapminder project](#).

This is released as an R package,

- so we can install it from CRAN like so:

```
# install.packages("gapminder")
```

Now load the package:

```
library(gapminder)
```

```
## Meet the `gapminder` data frame or "tibble"
```

By loading the `gapminder` package,

- we now have access to a data frame by the same name.

Get an overview of this with `str()`,

- which displays the structure of an object.

```
str(gapminder)
## tibble [1,704 x 6] (S3: tbl_df/tbl/data.frame)
## $ country   : Factor w/ 142 levels "Afghanistan",...: 1 1 1 1 1 1 1 1 1 1 ...
## $ continent: Factor w/ 5 levels "Africa","Americas",...: 3 3 3 3 3 3 3 3 3 3 ...
## $ year      : int [1:1704] 1952 1957 1962 1967 1972 1977 1982 1987 1992 1997 ...
## $ lifeExp   : num [1:1704] 28.8 30.3 32 34 36.1 ...
## $ pop       : int [1:1704] 8425333 9240934 10267083 11537966 13079460 14880372 12881816 13867957 163
## $ gdpPercap: num [1:1704] 779 821 853 836 740 ...
```

`str()` 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.

We could print the `gapminder` object itself to screen.

- However, if you've used R before, you might be reluctant to do this,
- because large data sets just fill up your Console

- and provide very little insight.

This is the first big win for **tibbles**.

- The **tidyverse**
- offers a special case of R’s default data frame: the “tibble”,
  - which is a nod to the actual class of these objects, `tbl_df`.

If you have not already done so,

- install the **tidyverse** meta-package now:

```
# install.packages("tidyverse")
```

Now load it:

```
library(tidyverse)
## -- Attaching packages ----- tidyverse 1.3.2 --
## v ggplot2 3.4.0      v purrr  1.0.1
## v tibble  3.1.8      v dplyr  1.0.10
## v tidyr   1.3.0      v stringr 1.5.0
## v readr   2.1.3      v forcats 0.5.2
## -- Conflicts ----- tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

Now we can boldly print **gapminder** to screen!

- It is a tibble (and also a regular data frame)
- and the **tidyverse** provides a nice print method
  - that shows the most important stuff
  - and doesn’t fill up your Console.

```
## see? it's still a regular data frame, but also a tibble
class(gapminder)
## [1] "tbl_df"      "tbl"        "data.frame"
gapminder
## # A tibble: 1,704 x 6
##   country    continent  year lifeExp    pop gdpPercap
##   <fct>      <fct>    <int>  <dbl>    <int>  <dbl>
## 1 Afghanistan Asia      1952   28.8  8425333    779.
## 2 Afghanistan Asia      1957   30.3  9240934    821.
## 3 Afghanistan Asia      1962   32.0 10267083    853.
## 4 Afghanistan Asia      1967   34.0 11537966    836.
## 5 Afghanistan Asia      1972   36.1 13079460    740.
## 6 Afghanistan Asia      1977   38.4 14880372    786.
## 7 Afghanistan Asia      1982   39.9 12881816    978.
## 8 Afghanistan Asia      1987   40.8 13867957    852.
## 9 Afghanistan Asia      1992   41.7 16317921    649.
## 10 Afghanistan Asia      1997   41.8 22227415    635.
## # ... with 1,694 more rows
```

If you are dealing with plain vanilla data frames,

- you can rein in data frame printing explicitly
  - with `head()` and `tail()`.
- Or turn it into a tibble with `as_tibble()`!

```
head(gapminder)
## # A tibble: 6 x 6
```

```
##   country      continent  year lifeExp      pop gdpPercap
##   <fct>        <fct>    <int>  <dbl>    <int>    <dbl>
## 1 Afghanistan Asia      1952   28.8  8425333    779.
## 2 Afghanistan Asia      1957   30.3  9240934    821.
## 3 Afghanistan Asia      1962   32.0 10267083    853.
## 4 Afghanistan Asia      1967   34.0 11537966    836.
## 5 Afghanistan Asia      1972   36.1 13079460    740.
## 6 Afghanistan Asia      1977   38.4 14880372    786.
tail(gapminder)
## # A tibble: 6 x 6
##   country continent  year lifeExp      pop gdpPercap
##   <fct>        <fct>    <int>  <dbl>    <int>    <dbl>
## 1 Zimbabwe Africa    1982   60.4  7636524    789.
## 2 Zimbabwe Africa    1987   62.4  9216418    706.
## 3 Zimbabwe Africa    1992   60.4 10704340    693.
## 4 Zimbabwe Africa    1997   46.8 11404948    792.
## 5 Zimbabwe Africa    2002   40.0 11926563    672.
## 6 Zimbabwe Africa    2007   43.5 12311143    470.
as_tibble(iris)
## # A tibble: 150 x 5
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##   <dbl>        <dbl>        <dbl>        <dbl> <fct>
## 1         5.1         3.5         1.4         0.2 setosa
## 2         4.9         3         1.4         0.2 setosa
## 3         4.7         3.2         1.3         0.2 setosa
## 4         4.6         3.1         1.5         0.2 setosa
## 5         5         3.6         1.4         0.2 setosa
## 6         5.4         3.9         1.7         0.4 setosa
## 7         4.6         3.4         1.4         0.3 setosa
## 8         5         3.4         1.5         0.2 setosa
## 9         4.4         2.9         1.4         0.2 setosa
## 10        4.9         3.1         1.5         0.1 setosa
## # ... with 140 more rows
```

More ways to query basic info on a data frame:

```
names(gapminder)
## [1] "country" "continent" "year" "lifeExp" "pop" "gdpPercap"
ncol(gapminder)
## [1] 6
length(gapminder)
## [1] 6
dim(gapminder)
## [1] 1704 6
nrow(gapminder)
## [1] 1704
```

A statistical overview can be obtained with `summary()`

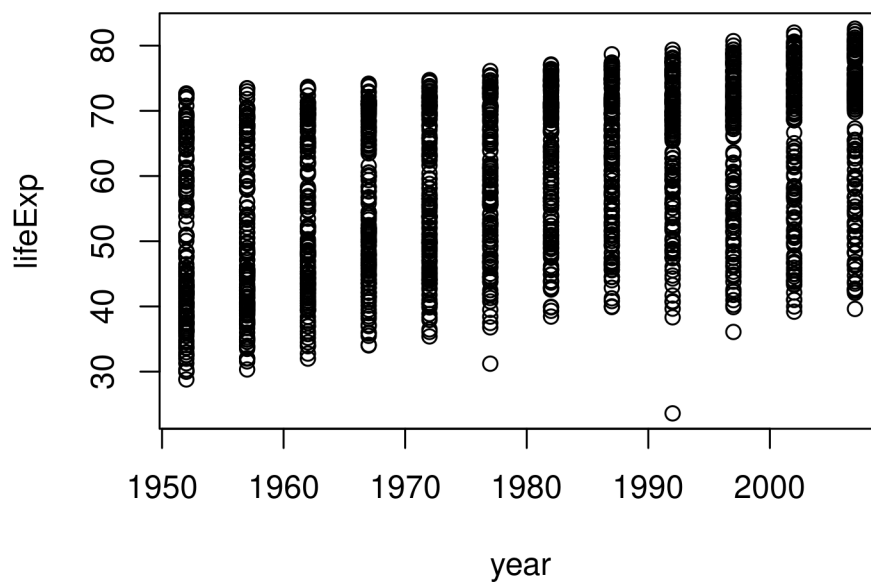
```
summary(gapminder)
##   country      continent      year      lifeExp
## Afghanistan: 12 Africa :624 Min. :1952 Min. :23.60
## Albania : 12 Americas:300 1st Qu.:1966 1st Qu.:48.20
## Algeria : 12 Asia :396 Median :1980 Median :60.71
## Angola : 12 Europe :360 Mean :1980 Mean :59.47
```

```
## Argentina : 12 Oceania : 24 3rd Qu.:1993 3rd Qu.:70.85
## Australia : 12 Max. :2007 Max. :82.60
## (Other) :1632
## pop gdpPercap
## Min. :6.001e+04 Min. : 241.2
## 1st Qu.:2.794e+06 1st Qu.: 1202.1
## Median :7.024e+06 Median : 3531.8
## Mean :2.960e+07 Mean : 7215.3
## 3rd Qu.:1.959e+07 3rd Qu.: 9325.5
## Max. :1.319e+09 Max. :113523.1
##
```

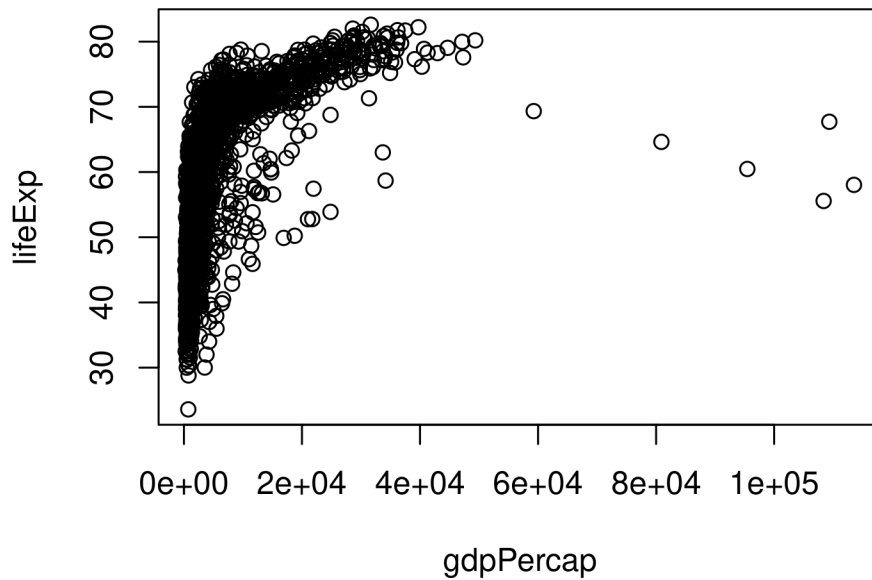
Although we haven't begun our formal coverage of visualization yet,

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

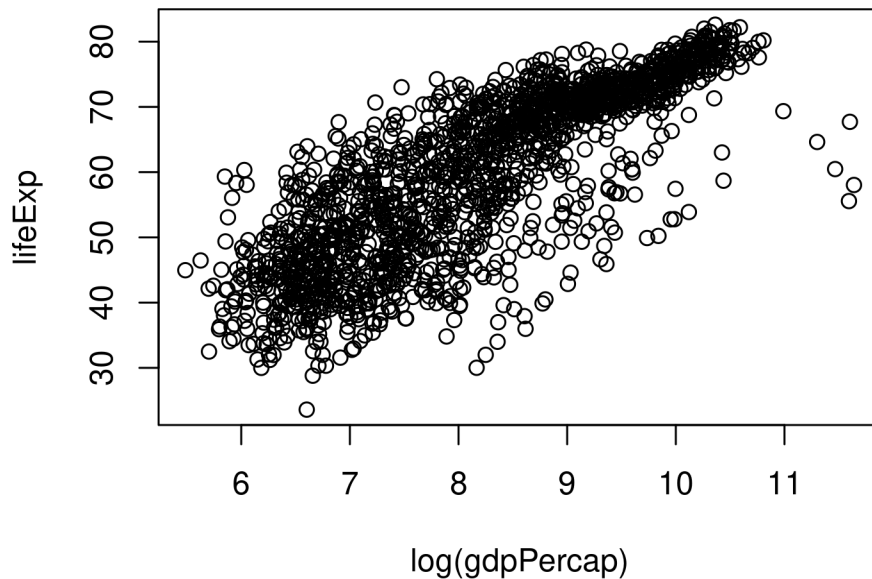
```
plot(lifeExp ~ year, gapminder)
```



```
plot(lifeExp ~ gdpPercap, gapminder)
```



```
plot(lifeExp ~ log(gdpPercap), gapminder)
```



#### 2.2.4.4 Non-sequitur: The Equals Operator

- Sidebar on equals:
  - A single equal sign `=` is most commonly used
    - \* to specify values of arguments when calling functions in R,
    - \* e.g. `group = continent`.
  - It can be used for assignment
    - \* but we advise against that,
    - \* in favor of `<-`.
  - A double equal sign `==` is a binary comparison operator,
    - \* akin to less than `<` or greater than `>`,
    - \* returning the logical value `TRUE` in the case of equality
    - \* and `FALSE` otherwise.
  - Although you may not yet understand exactly why,
    - \* `subset = country == "Colombia"` restricts operation – scatter plotting,



\* in above examples – to observations where the country is Colombia.

Let's go back to the result of `str()` to talk about what a data frame is.

```
str(gapminder)
## tibble [1,704 x 6] (S3: tbl_df/tbl/data.frame)
## $ country : Factor w/ 142 levels "Afghanistan",...: 1 1 1 1 1 1 1 1 1 1 ...
## $ continent: Factor w/ 5 levels "Africa","Americas",...: 3 3 3 3 3 3 3 3 3 3 ...
## $ year      : int [1:1704] 1952 1957 1962 1967 1972 1977 1982 1987 1992 1997 ...
## $ lifeExp   : num [1:1704] 28.8 30.3 32 34 36.1 ...
## $ pop       : int [1:1704] 8425333 9240934 10267083 11537966 13079460 14880372 12881816 13867957 163
## $ gdpPercap: num [1:1704] 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 a 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,
- e.g. numeric, character, and categorical data can be stored together.
- This comes up a lot!

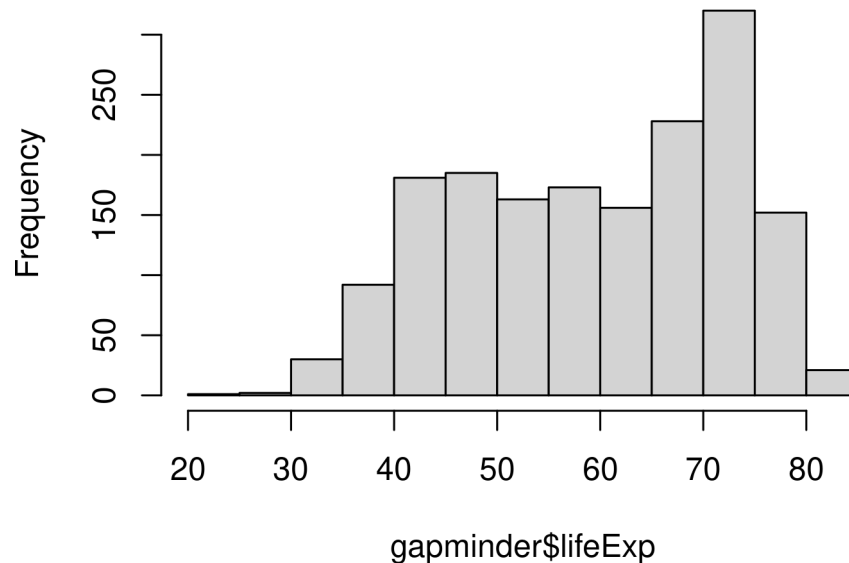
#### 2.2.4.5 Look at the variables inside a data frame

- To specify a single variable from a data frame,
  - use the dollar sign `$`.

Let's explore the numeric variable for life expectancy.

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

## Histogram of gapminder\$lifeExp



The year variable is an integer variable,

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

```
summary(gapminder$year)
##      Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    1952   1966   1980   1980   1993   2007
table(gapminder$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(gapminder$continent)
## [1] "factor"
summary(gapminder$continent)
##   Africa Americas   Asia  Europe Oceania
##    624     300    396    360     24
levels(gapminder$continent)
## [1] "Africa" "Americas" "Asia"    "Europe" "Oceania"
nlevels(gapminder$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(gapminder$continent)`
  - if you are skeptical.

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

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 your *factors*

- The pros far outweigh the cons.

Specifically in modeling and figure-making,

- factors are anticipated and accommodated
- by the functions and packages you will want to exploit.

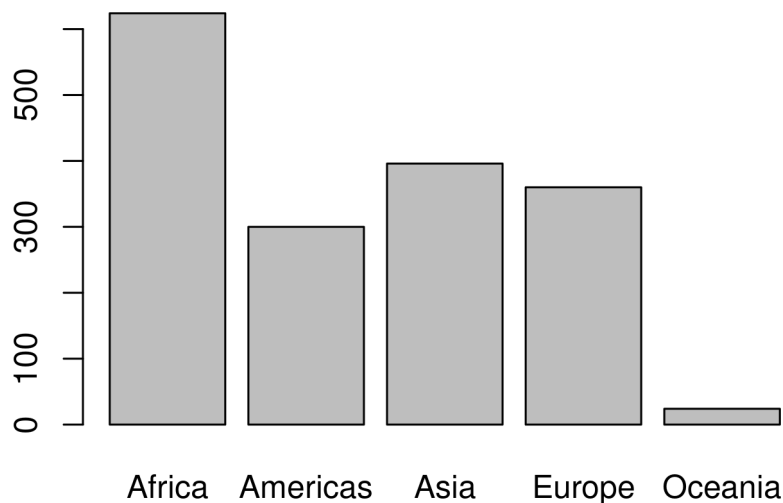
Here we count how many observations are associated with each continent

- 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 data set.

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



In the figures below, we see how factors

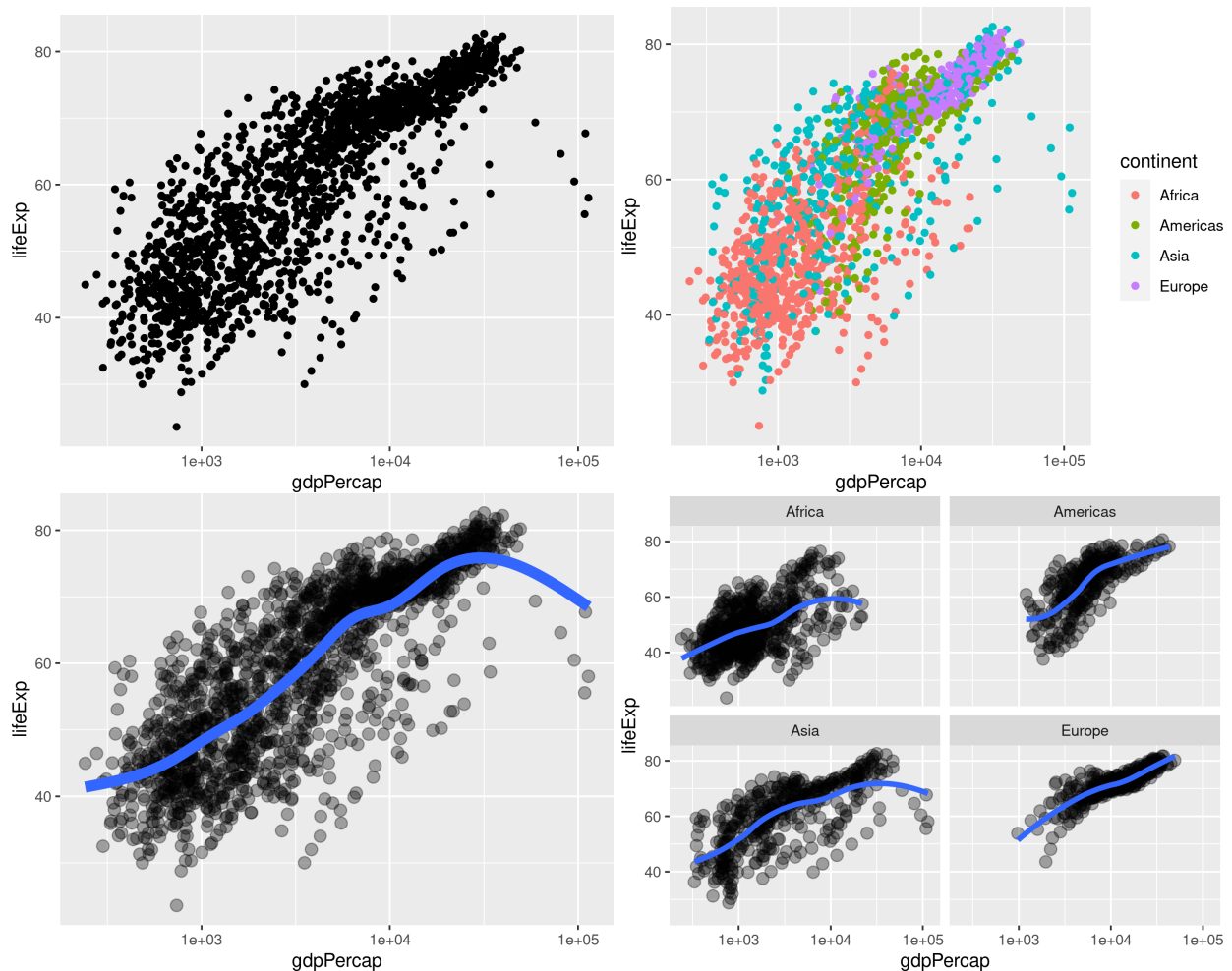
- can be put to work in figures.

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.\*

```
## we exploit the fact that ggplot2 was installed and loaded via the tidyverse
p <- ggplot(filter(gapminder, 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) +
  geom_smooth(lwd = 1.5, se = FALSE)
## `geom_smooth()` using method = 'loess' and formula = 'y ~ x'
```



#### 2.2.4.6 Recap

- Use data frames!!!
- Use the **tidyverse**!!! This will provide a special type of data frame called a “tibble” that has nice default printing behavior, among other benefits.
- When in doubt, `str()` something or print something.
- Always understand the basic extent of your data frames: number of rows and columns.
- Understand what flavor the variables are.
- Use factors!!! But with intention and care.

- Do basic statistical and visual sanity checking of each variable.
- Refer to variables by name, e.g., `gapminder$lifeExp`, not by column number. Your code will be more robust and readable.

CLASS STOPPED HERE

## 2.2.4.7 Tidy Manipulation: Introduction to dplyr package

### 2.2.4.7.1 Intro

- `dplyr` is a package for data manipulation,
  - developed by Hadley Wickham and Romain Francois.
  - It is built to be fast, highly expressive, and open-minded
    - \* about how your data is stored.
  - It is installed as part of the the `tidyverse` meta-package
    - \* and, as a core package, it is among those loaded via `library(tidyverse)`.

`dplyr`'s roots are in an earlier package called `plyr`,

- which implements the “split-apply-combine” strategy for data analysis (PDF).
- Where `plyr` covers a diverse set of inputs and outputs
  - (e.g., arrays, data frames, lists),
- `dplyr` has a laser-like focus on data frames
  - or, in the `tidyverse`, “tibbles”.
- `dplyr` is a package-level treatment of the `ddply()` function
  - from `plyr`,
- because “data frame in, data frame out”
- proved to be so incredibly important.

Have no idea what I'm talking about? Not sure if you care?

- If you use these base R functions:
  - `subset()`, `apply()`, `[sl]apply()`, `tapply()`, `aggregate()`,
  - `split()`, `do.call()`, `with()`, `within()`,
  - then you should keep reading.
- Also, if you use `for()` loops a lot,
  - you might enjoy learning other ways
  - to iterate over rows or groups of rows
  - or variables in a data frame.

Load `dplyr` and `gapminder`

I choose to load the `tidyverse`,

- which will load `dplyr`,
  - among other packages we use incidentally below.

Also load `gapminder`.

```
# library(gapminder)
# library(tidyverse)
```

Say hello to the Gapminder tibble

- The `gapminder` data frame is a special kind of data frame: a tibble.

```
gapminder
## # A tibble: 1,704 x 6
##   country    continent year lifeExp      pop gdpPercap
##   <fct>      <fct>    <int>  <dbl>   <int>    <dbl>
```

```
## 1 Afghanistan Asia      1952    28.8  8425333    779.
## 2 Afghanistan Asia      1957    30.3  9240934    821.
## 3 Afghanistan Asia      1962    32.0 10267083    853.
## 4 Afghanistan Asia      1967    34.0 11537966    836.
## 5 Afghanistan Asia      1972    36.1 13079460    740.
## 6 Afghanistan Asia      1977    38.4 14880372    786.
## 7 Afghanistan Asia      1982    39.9 12881816    978.
## 8 Afghanistan Asia      1987    40.8 13867957    852.
## 9 Afghanistan Asia      1992    41.7 16317921    649.
## 10 Afghanistan Asia     1997    41.8 22227415    635.
## # ... with 1,694 more rows
```

It's tibble-ness is why we get nice compact printing.

- For a reminder of the problems with base data frame printing,
  - go type `iris` in the R Console
- or, better yet, print a data frame to screen
  - that has lots of columns.

Note how `gapminder`'s `class()` includes `tbl_df`;

- the “tibble” terminology is a nod to this.

```
class(gapminder)
## [1] "tbl_df"      "tbl"        "data.frame"
```

There will be some functions, like `print()`,

- that know about tibbles and do something special.
- There will others that do not, like `summary()`.
- In which case the regular data frame treatment will happen,
  - because every tibble is also a regular data frame.

To turn any data frame into a tibble use `as_tibble()`:

```
as_tibble(iris)
## # A tibble: 150 x 5
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##         <dbl>         <dbl>         <dbl>         <dbl> <fct>
## 1         5.1         3.5         1.4         0.2 setosa
## 2         4.9         3         1.4         0.2 setosa
## 3         4.7         3.2         1.3         0.2 setosa
## 4         4.6         3.1         1.5         0.2 setosa
## 5         5         3.6         1.4         0.2 setosa
## 6         5.4         3.9         1.7         0.4 setosa
## 7         4.6         3.4         1.4         0.3 setosa
## 8         5         3.4         1.5         0.2 setosa
## 9         4.4         2.9         1.4         0.2 setosa
## 10        4.9         3.1         1.5         0.1 setosa
## # ... with 140 more rows
```

#### 2.2.4.7.2 Think before you create excerpts of your data ...

- If you feel the urge to store a little snippet of your data:

```
(canada <- gapminder[241:252, ])
## # A tibble: 12 x 6
##   country continent  year lifeExp      pop gdpPercap
```

```
##      <fct>    <fct>      <int>    <dbl>      <int>      <dbl>
##  1 Canada  Americas    1952      68.8 14785584    11367.
##  2 Canada  Americas    1957      70.0 17010154    12490.
##  3 Canada  Americas    1962      71.3 18985849    13462.
##  4 Canada  Americas    1967      72.1 20819767    16077.
##  5 Canada  Americas    1972      72.9 22284500    18971.
##  6 Canada  Americas    1977      74.2 23796400    22091.
##  7 Canada  Americas    1982      75.8 25201900    22899.
##  8 Canada  Americas    1987      76.9 26549700    26627.
##  9 Canada  Americas    1992      78.0 28523502    26343.
## 10 Canada  Americas    1997      78.6 30305843    28955.
## 11 Canada  Americas    2002      79.8 31902268    33329.
## 12 Canada  Americas    2007      80.7 33390141    36319.
```

Stop and ask yourself ...

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

If YES, **use proper data aggregation techniques** or faceting in `ggplot2` – **don't subset the data**. Or, more realistic, only subset the data as a temporary measure while you develop your elegant code for computing on or visualizing these data subsets.

If NO, then maybe you really do need to store a copy of a subset of the data. But seriously consider whether you can achieve your goals by simply using the `subset` = argument of, e.g., the `lm()` function, to limit computation to your excerpt of choice. Lots of functions offer a `subset` = argument!

Copies and excerpts of your data

- clutter your work space,
  - invite mistakes,
  - and sow general confusion.
- Avoid whenever possible.

Reality can also lie somewhere in between.

- You will find the workflows presented below
  - can help you accomplish your goals
- with minimal creation of temporary, intermediate objects.

### 2.2.4.7.3 Use `filter()` to subset data row-wise.

- `filter()` takes logical expressions
  - and returns the rows for which all are TRUE.
- Added `head()` to suppress superfluous outputs

```
filter(gapminder, lifeExp < 29) %>% head()
## # A tibble: 2 x 6
##   country    continent  year lifeExp    pop gdpPercap
##   <fct>      <fct>      <int>   <dbl>   <int>   <dbl>
## 1 Afghanistan Asia      1952    28.8 8425333    779.
## 2 Rwanda    Africa     1992    23.6 7290203    737.
filter(gapminder, country == "Rwanda", year > 1979) %>% head()
## # A tibble: 6 x 6
##   country    continent  year lifeExp    pop gdpPercap
##   <fct>      <fct>      <int>   <dbl>   <int>   <dbl>
```

```
## 1 Rwanda Africa 1982 46.2 5507565 882.
## 2 Rwanda Africa 1987 44.0 6349365 848.
## 3 Rwanda Africa 1992 23.6 7290203 737.
## 4 Rwanda Africa 1997 36.1 7212583 590.
## 5 Rwanda Africa 2002 43.4 7852401 786.
## 6 Rwanda Africa 2007 46.2 8860588 863.
filter(gapminder, country %in% c("Rwanda", "Afghanistan")) %>% head()
## # A tibble: 6 x 6
##   country continent year lifeExp pop gdpPercap
##   <fct>      <fct>   <int>   <dbl>   <int>   <dbl>
## 1 Afghanistan Asia    1952    28.8  8425333    779.
## 2 Afghanistan Asia    1957    30.3  9240934    821.
## 3 Afghanistan Asia    1962    32.0 10267083    853.
## 4 Afghanistan Asia    1967    34.0 11537966    836.
## 5 Afghanistan Asia    1972    36.1 13079460    740.
## 6 Afghanistan Asia    1977    38.4 14880372    786.
```

Compare with some base R code to accomplish the same things

```
gapminder[gapminder$lifeExp < 29,] %>% head() ## repeat `gapminder`, [i, j] indexing is distracting
subset(gapminder, country == "Rwanda") %>% head() ## almost same as filter; quite nice actually
```

Under no circumstances

- should you subset your data
  - the way I did at first:

```
excerpt <- gapminder[241:252, ]
```

Why is this a terrible idea?

- It is not self-documenting.
  - What is so special about rows 241 through 252?
- It is fragile.
  - This line of code will produce different results
  - if someone changes the row order of `gapminder`,
  - e.g. sorts the data earlier in the script.

```
filter(gapminder, country == "Canada") %>% head()
```

This call explains itself and is fairly robust.

#### 2.2.4.7.4 Meet the new pipe operator

- Before we go any further,
  - we should exploit the new pipe operator
  - that the tidyverse imports
    - \* from the `magrittr` package by Stefan Bache.

This is going to change your data analytic life.

- You no longer need to enact multi-operation commands
  - by nesting them inside each other,
  - like so many Russian nesting dolls.
- This new syntax leads to code
  - that is much easier to write and to read.

Here's what it looks like: `%>%`.



- The RStudio keyboard shortcut:
  - Ctrl + Shift + M (Windows), Cmd + Shift + M (Mac).

Let's demo then I'll explain:

```
gapminder %>% head()
## # A tibble: 6 x 6
##   country    continent  year lifeExp      pop gdpPercap
##   <fct>      <fct>    <int>  <dbl>    <int>    <dbl>
## 1 Afghanistan Asia      1952   28.8  8425333    779.
## 2 Afghanistan Asia      1957   30.3  9240934    821.
## 3 Afghanistan Asia      1962   32.0 10267083    853.
## 4 Afghanistan Asia      1967   34.0 11537966    836.
## 5 Afghanistan Asia      1972   36.1 13079460    740.
## 6 Afghanistan Asia      1977   38.4 14880372    786.
```

This is equivalent to `head(gapminder)`.

- The pipe operator takes the thing on the left-hand-side
  - and **pipes** it into the function call
  - on the right-hand-side
- literally, drops it in as the first argument.

Never fear, you can still specify other arguments to this function!

To see the first 3 rows of Gapminder,

- we could say `head(gapminder, 3)` or this:

```
gapminder %>% head(3)
## # A tibble: 3 x 6
##   country    continent  year lifeExp      pop gdpPercap
##   <fct>      <fct>    <int>  <dbl>    <int>    <dbl>
## 1 Afghanistan Asia      1952   28.8  8425333    779.
## 2 Afghanistan Asia      1957   30.3  9240934    821.
## 3 Afghanistan Asia      1962   32.0 10267083    853.
```

I've advised you to think

- “gets” whenever you see the assignment operator, `<-`.

Similarly, you should think

- “then” whenever you see the pipe operator, `%>%`.

You are probably not impressed yet, but the magic will soon happen.

#### 2.2.4.7.5 Use `select()` to subset the data on variables or columns. Back to dplyr ...

Use `select()` to subset the data on variables or columns. Here's a conventional call:

```
select(gapminder, year, lifeExp) %>% head()
## # A tibble: 6 x 2
##   year lifeExp
##   <int>  <dbl>
## 1  1952   28.8
## 2  1957   30.3
## 3  1962   32.0
## 4  1967   34.0
## 5  1972   36.1
## 6  1977   38.4
```

And here's the same operation,

- but written with the pipe operator
- and piped through `head()`:

```
gapminder %>%
  select(year, lifeExp) %>%
  head(4)
## # A tibble: 4 x 2
##   year lifeExp
##   <int>   <dbl>
## 1  1952    28.8
## 2  1957    30.3
## 3  1962    32.0
## 4  1967    34.0
```

Think: “Take `gapminder`,

- then select the variables `year` and `lifeExp`,
- then show the first 4 rows.”

#### 2.2.4.7.6 Revel in the convenience

- Here's the data for Cambodia,
  - but only certain variables:

```
gapminder %>%
  filter(country == "Cambodia") %>%
  select(year, lifeExp) %>% head()
## # A tibble: 6 x 2
##   year lifeExp
##   <int>   <dbl>
## 1  1952    39.4
## 2  1957    41.4
## 3  1962    43.4
## 4  1967    45.4
## 5  1972    40.3
## 6  1977    31.2
```

and what a typical base R call would look like:

```
gapminder[gapminder$country == "Cambodia", c("year", "lifeExp")] %>% head()
## # A tibble: 6 x 2
##   year lifeExp
##   <int>   <dbl>
## 1  1952    39.4
## 2  1957    41.4
## 3  1962    43.4
## 4  1967    45.4
## 5  1972    40.3
## 6  1977    31.2
```

#### 2.2.4.7.7 Pure, predictable, pipe able

- We've barely scratched the surface of `dplyr`
  - but I want to point out key principles you may start to appreciate.
  - If you're new to R or “programming with data”,

- \* feel free skip this section
- \* and [move on](#).

`dplyr`'s verbs, such as `filter()` and `select()`,

- are what's called [pure functions](#).
- To quote from Wickham's [Advanced R Programming book](#):

The functions that are the easiest to understand and reason about are pure functions: functions that always map the same input to the same output and have no other impact on the work space.

In other words, pure functions have no side effects: they don't affect the state of the world in any way apart from the value they return.

In fact, these verbs are a special case of pure functions: they take the same flavor of object as input and output.

Namely, a data frame or one of the other data receptacles `dplyr` supports.

And finally,

- the data is **always**
  - the very first argument of the verb functions.

This set of deliberate design choices,

- together with the new pipe operator,
- produces a highly effective,
  - low friction [domain-specific language](#)
  - for data analysis.

Go to the next block, [dplyr functions for a single dataset](#), for more `dplyr`!

#### 2.2.4.7.8 Resources

- `dplyr` official stuff
  - package home [on CRAN](#)
    - \* note there are several vignettes, with the [introduction](#) being the most relevant right now
    - \* the [one on window functions](#) will also be interesting to you now
  - development home [on GitHub](#)

[RStudio Data Wrangling cheatsheet](#), covering `dplyr` and `tidyr`. Remember you can get to these via `Help > Cheat sheets`.

[Excellent slides](#) on pipelines and `dplyr` by TJ Mahr, talk given to the Madison R Users Group.

Blog post [Hands-on dplyr tutorial for faster data manipulation in R](#) by Data School, that includes a link to an R Markdown document and links to videos

`dplyr` functions for a single data set

- In the introduction to `dplyr`, we used two very important verbs and an operator:
  - `filter()` for subsetting data with row logic
  - `select()` for subsetting data variable- or column-wise
  - the pipe operator `%>%`,
    - \* which feeds the LHS as the first argument
    - \* to the expression on the RHS

We also discussed `dplyr`'s role inside the tidyverse and tibbles:

- `dplyr` is a core package in the [tidyverse](#) meta-package.
- Since we often make incidental usage of the others,

- we will load dplyr and the others via `library(tidyverse)`.
- The tidyverse embraces a special flavor of data frame,
  - called a tibble.
- The `gapminder` data set is stored as a tibble.

#### 2.2.4.7.9 Load dplyr and gapminder

- I choose to load the tidyverse, which will load dplyr, among other packages we use incidentally below. Also load gapminder.

```
# library(gapminder)
# library(tidyverse)
```

#### 2.2.4.7.10 Create a copy of gapminder

- We're going to make changes to the `gapminder` tibble.

To eliminate any fear

- that you're damaging the data that comes with the package,
- we create an explicit copy of `gapminder` for our experiments.

```
(my_gap <- gapminder)
## # A tibble: 1,704 x 6
##   country    continent  year lifeExp      pop gdpPercap
##   <fct>      <fct>    <int>  <dbl>    <int>    <dbl>
## 1 Afghanistan Asia      1952   28.8  8425333    779.
## 2 Afghanistan Asia      1957   30.3  9240934    821.
## 3 Afghanistan Asia      1962   32.0 10267083    853.
## 4 Afghanistan Asia      1967   34.0 11537966    836.
## 5 Afghanistan Asia      1972   36.1 13079460    740.
## 6 Afghanistan Asia      1977   38.4 14880372    786.
## 7 Afghanistan Asia      1982   39.9 12881816    978.
## 8 Afghanistan Asia      1987   40.8 13867957    852.
## 9 Afghanistan Asia      1992   41.7 16317921    649.
## 10 Afghanistan Asia      1997   41.8 22227415    635.
## # ... with 1,694 more rows
```

Pay close attention to when we evaluate statements

- but let the output just print to screen:

```
## let output print to screen, but do not store
my_gap %>% filter(country == "Canada") %>% head()
```

... versus when we assign the output to an object,

- possibly overwriting an existing object.

```
## store the output as an R object
my_precious <- my_gap %>% filter(country == "Canada")
```

#### 2.2.4.7.11 Use mutate() to add new variables

- Imagine we wanted to recover each country's GDP.

After all, the Gapminder data

- has a variable for population
  - and GDP per capita.

- Let's multiply them together.

`mutate()` is a function that

- defines and inserts new variables into a tibble.
- You can refer to existing variables by name.

```
my_gap %>%
  mutate(gdp = pop * gdpPercap) %>% head()
## # A tibble: 6 x 7
##   country      continent year lifeExp      pop gdpPercap      gdp
##   <fct>        <fct>    <int>   <dbl>    <int>    <dbl>    <dbl>
## 1 Afghanistan Asia      1952   28.8  8425333    779.  6567086330.
## 2 Afghanistan Asia      1957   30.3  9240934    821.  7585448670.
## 3 Afghanistan Asia      1962   32.0 10267083    853.  8758855797.
## 4 Afghanistan Asia      1967   34.0 11537966    836.  9648014150.
## 5 Afghanistan Asia      1972   36.1 13079460    740.  9678553274.
## 6 Afghanistan Asia      1977   38.4 14880372    786. 11697659231.
```

Hmmmm ... those GDP numbers are almost uselessly large and abstract.

Consider the advice of Randall Munroe of xkcd:

One thing that bothers me is large numbers presented without context...

'If I added a zero to this number, would the sentence containing it mean something different to me?'

If the answer is 'no,' maybe the number has no business being in the sentence in the first place."

Maybe it would be more meaningful to consumers of my tables and figures to stick with GDP per capita.

But what if I reported GDP per capita, *relative to some benchmark country*.

Since Canada is my adopted home, I'll go with that.

I need to create a new variable

- that is `gdpPercap` divided by Canadian `gdpPercap`,
  - taking care that I always divide two numbers that pertain to the same year.

How I achieve:

- Filter down to the rows for Canada.
- Create a new temporary variable in `my_gap`:
  - Extract the `gdpPercap` variable from the Canadian data.
  - Replicate it once per country in the data set, so it has the right length.
- Divide raw `gdpPercap` by this Canadian figure.
- Discard the temporary variable of replicated Canadian `gdpPercap`.

```
ctib <- my_gap %>%
  filter(country == "Canada")
## this is a semi-dangerous way to add this variable
## I'd prefer to join on year, but we haven't covered joins yet
my_gap <- my_gap %>%
  mutate(
    tmp = rep(ctib$gdpPercap, nlevels(country)),
    gdpPercapRel = gdpPercap / tmp,
    tmp = NULL
  )
```

Note that, `mutate()` builds new variables sequentially

- so you can reference earlier ones (like `tmp`)
  - when defining later ones (like `gdpPercapRel`).
- Also, you can get rid of a variable
  - by setting it to `NULL`.

How could we sanity check that this worked?

- The Canadian values for `gdpPercapRel` better all be 1!

```
my_gap %>%
  filter(country == "Canada") %>%
  select(country, year, gdpPercapRel)
## # A tibble: 12 x 3
##   country year gdpPercapRel
##   <fct>   <int>         <dbl>
## 1 Canada  1952             1
## 2 Canada  1957             1
## 3 Canada  1962             1
## 4 Canada  1967             1
## 5 Canada  1972             1
## 6 Canada  1977             1
## 7 Canada  1982             1
## 8 Canada  1987             1
## 9 Canada  1992             1
## 10 Canada 1997             1
## 11 Canada 2002             1
## 12 Canada 2007             1
```

I perceive Canada to be a “high GDP” country,

- so I predict that the distribution of `gdpPercapRel` is located below 1,
  - possibly even well below.
- Check your intuition!

```
summary(my_gap$gdpPercapRel)
##      Min.   1st Qu.   Median     Mean   3rd Qu.    Max.
## 0.007236 0.061648 0.171521 0.326659 0.446564 9.534690
```

The relative GDP per capita numbers are, in general, well below 1.

We see that most of the countries covered by this data set

- have substantially lower GDP per capita, relative to Canada,
- across the entire time period.

Remember: Trust No One. Including (especially?) yourself.

- Always try to find a way to check that you’ve done what meant to.
- Prepare to be horrified.

#### 2.2.4.7.12 Use `arrange()` to row-order data in a principled way

- `arrange()` reorders the rows in a data frame.
  - Imagine you wanted this data ordered by year then country,
    - \* as opposed to by country then year.

```
my_gap %>%
  arrange(year, country) %>% head()
```

```
## # A tibble: 6 x 7
##   country      continent year lifeExp      pop gdpPercap gdpPercapRel
##   <fct>        <fct>    <int>  <dbl>    <int>    <dbl>        <dbl>
## 1 Afghanistan Asia      1952   28.8  8425333    779.        0.0686
## 2 Albania      Europe    1952   55.2  1282697   1601.        0.141
## 3 Algeria      Africa    1952   43.1  9279525   2449.        0.215
## 4 Angola       Africa    1952   30.0  4232095   3521.        0.310
## 5 Argentina    Americas  1952   62.5  17876956  5911.        0.520
## 6 Australia    Oceania   1952   69.1  8691212  10040.        0.883
```

Or maybe you want just the data from 2007,

- sorted on life expectancy?

```
my_gap %>%
  filter(year == 2007) %>%
  arrange(lifeExp) %>% head()
## # A tibble: 6 x 7
##   country      continent year lifeExp      pop gdpPercap gdpPercapRel
##   <fct>        <fct>    <int>  <dbl>    <int>    <dbl>        <dbl>
## 1 Swaziland    Africa    2007   39.6  1133066   4513.        0.124
## 2 Mozambique   Africa    2007   42.1  19951656    824.        0.0227
## 3 Zambia       Africa    2007   42.4  11746035   1271.        0.0350
## 4 Sierra Leone Africa    2007   42.6   6144562    863.        0.0237
## 5 Lesotho      Africa    2007   42.6   2012649   1569.        0.0432
## 6 Angola       Africa    2007   42.7  12420476   4797.        0.132
```

Oh, you'd like to sort on life expectancy

- in descending order? Then use `desc()`.

```
my_gap %>%
  filter(year == 2007) %>%
  arrange(desc(lifeExp)) %>% head()
## # A tibble: 6 x 7
##   country      continent year lifeExp      pop gdpPercap gdpPercapRel
##   <fct>        <fct>    <int>  <dbl>    <int>    <dbl>        <dbl>
## 1 Japan        Asia      2007   82.6  127467972  31656.        0.872
## 2 Hong Kong, China Asia      2007   82.2   6980412   39725.        1.09
## 3 Iceland      Europe    2007   81.8    301931   36181.        0.996
## 4 Switzerland Europe    2007   81.7   7554661   37506.        1.03
## 5 Australia    Oceania   2007   81.2  20434176  34435.        0.948
## 6 Spain        Europe    2007   80.9  40448191  28821.        0.794
```

I advise that your analyses

- NEVER rely on rows or variables being in a specific order.
- But it's still true that human beings write the code
  - and the interactive development process can be much nicer
  - if you reorder the rows of your data as you go along.
- Also, once you are preparing tables for human eyeballs,
  - it is imperative that you step up
  - and take control of row order.

#### 2.2.4.7.13 Use `rename()` to rename variables

- When I first cleaned this Gapminder excerpt,

- I was a `camelCase` person,
- but now I'm all about `snake_case`.

So I am vexed by the variable names I chose

- when I cleaned this data years ago.
- Let's rename some variables!

```
my_gap %>%
  rename(life_exp = lifeExp,
         gdp_percap = gdpPercap,
         gdp_percap_rel = gdpPercapRel) %>% head()
## # A tibble: 6 x 7
##   country    continent  year life_exp      pop gdp_percap gdp_percap_rel
##   <fct>      <fct>    <int>   <dbl>    <int>    <dbl>      <dbl>
## 1 Afghanistan Asia      1952    28.8  8425333    779.      0.0686
## 2 Afghanistan Asia      1957    30.3  9240934    821.      0.0657
## 3 Afghanistan Asia      1962    32.0 10267083    853.      0.0634
## 4 Afghanistan Asia      1967    34.0 11537966    836.      0.0520
## 5 Afghanistan Asia      1972    36.1 13079460    740.      0.0390
## 6 Afghanistan Asia      1977    38.4 14880372    786.      0.0356
```

I did NOT assign the post-rename object back to `my_gap`

- because that would make the chunks in this practicum
  - harder to copy/paste and run out of order.
- In real life, I would probably assign this back to `my_gap`,
  - in a data preparation script,
  - and proceed with the new variable names.

#### 2.2.4.7.14 `select()` can rename and reposition variables

- You've seen simple use of `select()`.

There are two tricks you might enjoy:

1. `select()` can rename the variables you request to keep.
2. `select()` can be used with `everything()` to hoist a variable up to the front of the tibble.

```
my_gap %>%
  filter(country == "Burundi", year > 1996) %>%
  select(yr = year, lifeExp, gdpPercap) %>%
  select(gdpPercap, everything()) %>% head()
## # A tibble: 3 x 3
##   gdpPercap  yr lifeExp
##   <dbl> <int>   <dbl>
## 1    463.  1997    45.3
## 2    446.  2002    47.4
## 3    430.  2007    49.6
```

`everything()` is one of several helpers for variable selection.

- Read its help to see the rest.

#### 2.2.4.7.15 `group_by()` is a mighty weapon

- I have found ~~friends and family~~ collaborators
  - love to ask seemingly innocuous questions like,
    - \* “which country experienced the sharpest 5-year drop in life expectancy?”.



- In fact, that is a totally natural question to ask.
  - \* But if you are using a language that doesn't know about data,
  - \* it's an incredibly annoying question to answer.

`dplyr` offers powerful tools to solve this class of problem.

- `group_by()` adds extra structure to your data set – grouping information – which lays the groundwork for computations within the groups.
- `summarize()` takes a data set with  $n$  observations, computes requested summaries, and returns a data set with 1 observation.
- Window functions take a data set with  $n$  observations and return a data set with  $n$  observations.
- `mutate()` and `summarize()` will honor groups.
- You can also do very general computations on your groups with `do()`, though elsewhere in this course, I advocate for other approaches that I find more intuitive, using the `purrr` package.

Combined with the verbs you already know,

- these new tools allow you
  - to solve an extremely diverse set of problems
  - with relative ease.

Counting things up

- Let's start with simple counting.

How many observations do we have per continent?

```
my_gap %>%
  group_by(continent) %>%
  summarize(n = n()) %>% head()
## # A tibble: 5 x 2
##   continent      n
##   <fct>      <int>
## 1 Africa      624
## 2 Americas    300
## 3 Asia        396
## 4 Europe      360
## 5 Oceania     24
```

Let us pause here to think about the tidyverse.

You could get these same frequencies using `table()` from base R.

```
table(gapminder$continent)
##
##   Africa Americas      Asia  Europe  Oceania
##    624      300      396     360      24
str(table(gapminder$continent))
## 'table' int [1:5(1d)] 624 300 396 360 24
## - attr(*, "dimnames")=List of 1
##   ..$ : chr [1:5] "Africa" "Americas" "Asia" "Europe" ...
```

But the object of class `table` that is returned

- makes downstream computation a bit fiddlier than you'd like.

For example, it's too bad the continent levels

- come back only as *names*
  - and not as a proper factor,
  - with the original set of levels.

This is an example of how the tidyverse

- smooths transitions where you want
- the output of step i
  - to become the input of step  $i + 1$ .

The `tally()` function is a convenience function

- that knows to count rows.
- It honors groups.

```
my_gap %>%
  group_by(continent) %>%
  tally() %>% head()
## # A tibble: 5 x 2
##   continent     n
##   <fct>       <int>
## 1 Africa       624
## 2 Americas     300
## 3 Asia         396
## 4 Europe       360
## 5 Oceania      24
```

The `count()` function is an even more convenient function

- that does both grouping and counting.

```
my_gap %>%
  count(continent)
## # A tibble: 5 x 2
##   continent     n
##   <fct>       <int>
## 1 Africa       624
## 2 Americas     300
## 3 Asia         396
## 4 Europe       360
## 5 Oceania      24
```

What if we wanted to add the number of unique countries for each continent?

You can compute multiple summaries inside `summarize()`.

- Use the `n_distinct()` function
  - to count the number of distinct countries
  - within each continent.

```
my_gap %>%
  group_by(continent) %>%
  summarize(n = n(),
            n_countries = n_distinct(country)) %>% head()
## # A tibble: 5 x 3
##   continent     n n_countries
##   <fct>       <int>       <int>
## 1 Africa       624         52
## 2 Americas     300         25
## 3 Asia         396         33
## 4 Europe       360         30
## 5 Oceania      24          2
```

General summarization

- The functions you'll apply within `summarize()`
  - include classical statistical summaries,
    - \* like `mean()`, `median()`, `var()`, `sd()`, `mad()`,
    - \* `IQR()`, `min()`, and `max()`.
  - Remember they are functions that take  $n$  inputs
    - \* and distill them down into 1 output.

Although this may be statistically ill-advised,

- let's compute the average life expectancy by continent.

```
my_gap %>%
  group_by(continent) %>%
  summarize(avg_lifeExp = mean(lifeExp)) %>% head()
## # A tibble: 5 x 2
##   continent avg_lifeExp
##   <fct>      <dbl>
## 1 Africa      48.9
## 2 Americas    64.7
## 3 Asia        60.1
## 4 Europe      71.9
## 5 Oceania     74.3
```

`summarize_at()` applies the same summary function(s)

- to multiple variables.
- Let's compute average and median life expectancy and GDP per capita
  - by continent by year ...
  - but only for 1952 and 2007.

```
my_gap %>%
  filter(year %in% c(1952, 2007)) %>%
  group_by(continent, year) %>%
  summarize_at(vars(lifeExp, gdpPercap), funs(mean, median)) %>% head()
## # A tibble: 6 x 6
## # Groups:   continent [3]
##   continent year lifeExp_mean gdpPercap_mean lifeExp_median gdpPercap_median
##   <fct>    <int>      <dbl>         <dbl>          <dbl>          <dbl>
## 1 Africa   1952      39.1          1253.           38.8           987.
## 2 Africa   2007      54.8          3089.           52.9          1452.
## 3 Americas 1952      53.3          4079.           54.7          3048.
## 4 Americas 2007      73.6          11003.          72.9          8948.
## 5 Asia     1952      46.3           5195.           44.9          1207.
## 6 Asia     2007      70.7          12473.           72.4          4471.
```

Let's focus just on Asia.

- What are the minimum and maximum life expectancies
- seen by year?

```
my_gap %>%
  filter(continent == "Asia") %>%
  group_by(year) %>%
  summarize(min_lifeExp = min(lifeExp),
            max_lifeExp = max(lifeExp)) %>% head()
## # A tibble: 6 x 3
##   year min_lifeExp max_lifeExp
##   <int>      <dbl>      <dbl>
```

```
## 1 1952      28.8      65.4
## 2 1957      30.3      67.8
## 3 1962      32.0      69.4
## 4 1967      34.0      71.4
## 5 1972      36.1      73.4
## 6 1977      31.2      75.4
```

Of course it would be much more interesting to see

- *which* country contributed these extreme observations.
  - Is the minimum (maximum) always coming from the same country?
- We tackle that with window functions shortly.

#### 2.2.4.7.16 Grouped mutate

- Sometimes you don't want to collapse the  $n$  rows for each group into one row.
  - You want to keep your groups,
  - but compute within them.

Computing with group-wise summaries

- Let's make a new variable that is
  - the years of life expectancy gained (lost) relative to 1952,
    - \* for each individual country.
  - We group by country
    - \* and use `mutate()` to make a new variable.
  - The `first()` function extracts the first value from a vector.
  - Notice that `first()` is
    - \* operating on the vector of life expectancies
    - \* *within each country group*.

```
my_gap %>%
  group_by(country) %>%
  select(country, year, lifeExp) %>%
  mutate(lifeExp_gain = lifeExp - first(lifeExp)) %>%
  filter(year < 1963) %>% head()
## # A tibble: 6 x 4
## # Groups:   country [2]
##   country      year lifeExp lifeExp_gain
##   <fct>      <int>   <dbl>      <dbl>
## 1 Afghanistan  1952    28.8         0
## 2 Afghanistan  1957    30.3        1.53
## 3 Afghanistan  1962    32.0        3.20
## 4 Albania      1952    55.2         0
## 5 Albania      1957    59.3        4.05
## 6 Albania      1962    64.8        9.59
```

Within country,

- we take the difference between life expectancy in year  $i$ 
  - and life expectancy in 1952.
- Therefore we always see zeroes for 1952 and,
  - for most countries,
  - a sequence of positive and increasing numbers.

Window functions

- Window functions
  - take  $n$  inputs
    - \* and give back  $n$  outputs.
  - Furthermore, the output depends on all the values.
  - So `rank()` is a window function
    - \* but `log()` is not.

Here we use window functions

- based on ranks and offsets.

Let's revisit the worst and best life expectancies in Asia over time,

- but retaining info about *which* country
- contributes these extreme values.

```
my_gap %>%
  filter(continent == "Asia") %>%
  select(year, country, lifeExp) %>%
  group_by(year) %>%
  filter(min_rank(desc(lifeExp)) < 2 | min_rank(lifeExp) < 2) %>%
  arrange(year) %>%
  print(n = Inf) %>% head()
## # A tibble: 24 x 3
## # Groups:   year [12]
##   year country    lifeExp
##   <int> <fct>      <dbl>
## 1  1952 Afghanistan  28.8
## 2  1952 Israel      65.4
## 3  1957 Afghanistan  30.3
## 4  1957 Israel      67.8
## 5  1962 Afghanistan  32.0
## 6  1962 Israel      69.4
## 7  1967 Afghanistan  34.0
## 8  1967 Japan       71.4
## 9  1972 Afghanistan  36.1
##10  1972 Japan       73.4
##11  1977 Cambodia    31.2
##12  1977 Japan       75.4
##13  1982 Afghanistan  39.9
##14  1982 Japan       77.1
##15  1987 Afghanistan  40.8
##16  1987 Japan       78.7
##17  1992 Afghanistan  41.7
##18  1992 Japan       79.4
##19  1997 Afghanistan  41.8
##20  1997 Japan       80.7
##21  2002 Afghanistan  42.1
##22  2002 Japan       82
##23  2007 Afghanistan  43.8
##24  2007 Japan       82.6
## # A tibble: 6 x 3
## # Groups:   year [3]
##   year country    lifeExp
##   <int> <fct>      <dbl>
## 1  1952 Afghanistan  28.8
```

```
## 2 1952 Israel      65.4
## 3 1957 Afghanistan 30.3
## 4 1957 Israel      67.8
## 5 1962 Afghanistan 32.0
## 6 1962 Israel      69.4
```

We see that (min = Afghanistan, max = Japan) is the most frequent result,

- but Cambodia and Israel pop up at least once each
  - as the min or max, respectively.
- That table should make you impatient for our upcoming work
  - on tidying and reshaping data!

Wouldn't it be nice to have one row per year?

- How did that actually work?
- First, I store and view a partial
  - that leaves off the `filter()` statement.
- All of these operations should be familiar.

```
asia <- my_gap %>%
  filter(continent == "Asia") %>%
  select(year, country, lifeExp) %>%
  group_by(year)
asia %>% head()
## # A tibble: 6 x 3
## # Groups:   year [6]
##   year country    lifeExp
##   <int> <fct>      <dbl>
## 1  1952 Afghanistan  28.8
## 2  1957 Afghanistan  30.3
## 3  1962 Afghanistan  32.0
## 4  1967 Afghanistan  34.0
## 5  1972 Afghanistan  36.1
## 6  1977 Afghanistan  38.4
```

Now we apply a window function – `min_rank()`.

- Since `asia` is grouped by year,
  - `min_rank()` operates within mini-data sets,
  - each for a specific year.
- Applied to the variable `lifeExp`, `min_rank()`
  - returns the rank of each country's observed life expectancy.
- FYI, the `min` part just specifies how ties are broken.
- Here is an explicit peek at these within-year life expectancy ranks,
  - in both the (default) ascending and descending order.

For concreteness, I use `mutate()`

- to actually create these variables,
  - even though I dropped this in the solution above.
- Let's look at a bit of that.

```
asia %>%
  mutate(le_rank = min_rank(lifeExp),
         le_desc_rank = min_rank(desc(lifeExp))) %>%
  filter(country %in% c("Afghanistan", "Japan", "Thailand"), year > 1995) %>% head()
## # A tibble: 6 x 5
```

```
## # Groups:   year [3]
##   year country    lifeExp le_rank le_desc_rank
##   <int> <fct>      <dbl>   <int>      <int>
## 1  1997 Afghanistan  41.8     1         33
## 2  2002 Afghanistan  42.1     1         33
## 3  2007 Afghanistan  43.8     1         33
## 4  1997 Japan        80.7    33         1
## 5  2002 Japan        82       33         1
## 6  2007 Japan        82.6    33         1
```

Afghanistan tends to present 1's in the `le_rank` variable,

- Japan tends to present 1's in the `le_desc_rank` variable
- and other countries,
  - like Thailand,
  - present less extreme ranks.

You can understand the original `filter()` statement now:

```
# filter(min_rank(desc(asia$lifeExp)) < 2 | min_rank(asia$lifeExp) < 2)
```

These two sets of ranks are formed on-the-fly, within year group,

- and `filter()` retains rows with rank less than 2,
  - which means ... the row with rank = 1.
- Since we do for ascending and descending ranks,
  - we get both the min and the max.
- If we had wanted just the min OR the max,
  - an alternative approach using `top_n()`
  - would have worked.

```
my_gap %>%
  filter(continent == "Asia") %>%
  select(year, country, lifeExp) %>%
  arrange(year) %>%
  group_by(year) %>%
  #top_n(1, wt = lifeExp)          ## gets the min
  top_n(1, wt = desc(lifeExp)) ## gets the max
## # A tibble: 12 x 3
## # Groups:   year [12]
##   year country    lifeExp
##   <int> <fct>      <dbl>
## 1  1952 Afghanistan  28.8
## 2  1957 Afghanistan  30.3
## 3  1962 Afghanistan  32.0
## 4  1967 Afghanistan  34.0
## 5  1972 Afghanistan  36.1
## 6  1977 Cambodia    31.2
## 7  1982 Afghanistan  39.9
## 8  1987 Afghanistan  40.8
## 9  1992 Afghanistan  41.7
## 10 1997 Afghanistan  41.8
## 11 2002 Afghanistan  42.1
## 12 2007 Afghanistan  43.8
```

#### 2.2.4.7.17 Grand Finale

- So let's answer that "simple" question:
  - which country experienced the sharpest 5-year drop in life expectancy?
  - Recall that this excerpt of the Gapminder data
    - \* only has data every five years, e.g. for 1952, 1957, etc.
  - So this really means looking at life expectancy changes
    - \* between adjacent time points.
  - At this point, that's just too easy,
    - \* so let's do it by continent while we're at it.

```
my_gap %>%
  select(country, year, continent, lifeExp) %>%
  group_by(continent, country) %>%
  ## within country, take (lifeExp in year i) - (lifeExp in year i - 1)
  ## positive means lifeExp went up, negative means it went down
  mutate(le_delta = lifeExp - lag(lifeExp)) %>%
  ## within country, retain the worst lifeExp change = smallest or most negative
  summarize(worst_le_delta = min(le_delta, na.rm = TRUE)) %>%
  ## within continent, retain the row with the lowest worst_le_delta
  top_n(-1, wt = worst_le_delta) %>%
  arrange(worst_le_delta)
## `summarise()` has grouped output by 'continent'. You can override using the
## `.groups` argument.
## # A tibble: 5 x 3
## # Groups:   continent [5]
##   continent country      worst_le_delta
##   <fct>      <fct>          <dbl>
## 1 Africa    Rwanda             -20.4
## 2 Asia      Cambodia            -9.10
## 3 Americas  El Salvador         -1.51
## 4 Europe    Montenegro          -1.46
## 5 Oceania   Australia             0.170
```

Ponder that for a while.

The subject matter and the code.

Mostly you're seeing what genocide looks like

- in dry statistics
- on average life expectancy.

Break the code into pieces,

- starting at the top,
- and inspect the intermediate results.

That's certainly how I was able to *write* such a thing.

These commands do not leap fully formed

- out of anyone's forehead
- they are built up gradually,
  - with lots of errors and refinements along the way.
- I'm not even sure it's a great idea
  - to do so much manipulation in one fell swoop.

Is the statement above really hard for you to read?

- If yes, then by all means



- break it into pieces
- and make some intermediate objects.
- Your code should be easy
  - to write and read
  - when you’re done.

In later practicums, we’ll explore more of dplyr,

- such as operations based on two data sets.

#### 2.2.4.7.18 Resources

- dplyr official stuff
  - package home [on CRAN](#)
    - \* note there are several vignettes, with the [introduction](#) being the most relevant right now
    - \* the [one on window functions](#) will also be interesting to you now
  - development home [on GitHub](#)
  - [tutorial HW delivered](#) (note this links to a DropBox folder) at useR! 2014 conference

[RStudio Data Wrangling cheatsheet](#), covering dplyr and tidyr. Remember you can get to these via *Help > Cheatsheets*.

[Data transformation](#) chapter of [R for Data Science](#)

[Excellent slides](#) on pipelines and dplyr by TJ Mahr, talk given to the Madison R Users Group.

Blog post [Hands-on dplyr tutorial for faster data manipulation in R](#) by Data School, that includes a link to an R Markdown document and links to videos

#### 2.2.4.7.19 References

- [Data import](#) chapter of [R for Data Science](#) by Hadley Wickham and Garrett Golemund.

Nine simple ways to make it easier to (re)use your data by Ethan P White, Elita Baldridge, Zachary T. Brym, Kenneth J. Locey, Daniel J. McGlinn, Sarah R. Supp.

- First appeared here: PeerJ PrePrints 1:e7v2 <http://dx.doi.org/10.7287/peerj.preprints.7v2>
- Published here: Ideas in Ecology and Evolution 6(2): 1?10, 2013. [doi:10.4033/iee.2013.6b.6.f](https://doi.org/10.4033/iee.2013.6b.6.f) <http://library.queensu.ca/ojs/index.php/IEE/article/view/4608>
- Section 4 “Use Standard Data Formats” is especially good reading.

Tidy data by Hadley Wickham.

- In the Journal of Statistical Software Vol 59 (2014), Issue 10, 10.18637/jss.v059.i10: <http://www.jstatsoft.org/article/view/v059i10>
- PDF also available here: <http://vita.had.co.nz/papers/tidy-data.pdf>

#### 2.2.4.8 Links

- Jenny Bryan, RStudio software engineer
- Stats Prof at U British Columbia
  - <https://twitter.com/JennyBryan>