

# Data Visualization in R with **ggplot2**

Introduction to R and RStudio

**Cédric Scherer**

Physalia Courses | March 2-6 2020

Photo by Richard Strozyński

**HELLO**  
my name is

Cédric

[cedricscherer.netlify.com](http://cedricscherer.netlify.com)



@CedScherer



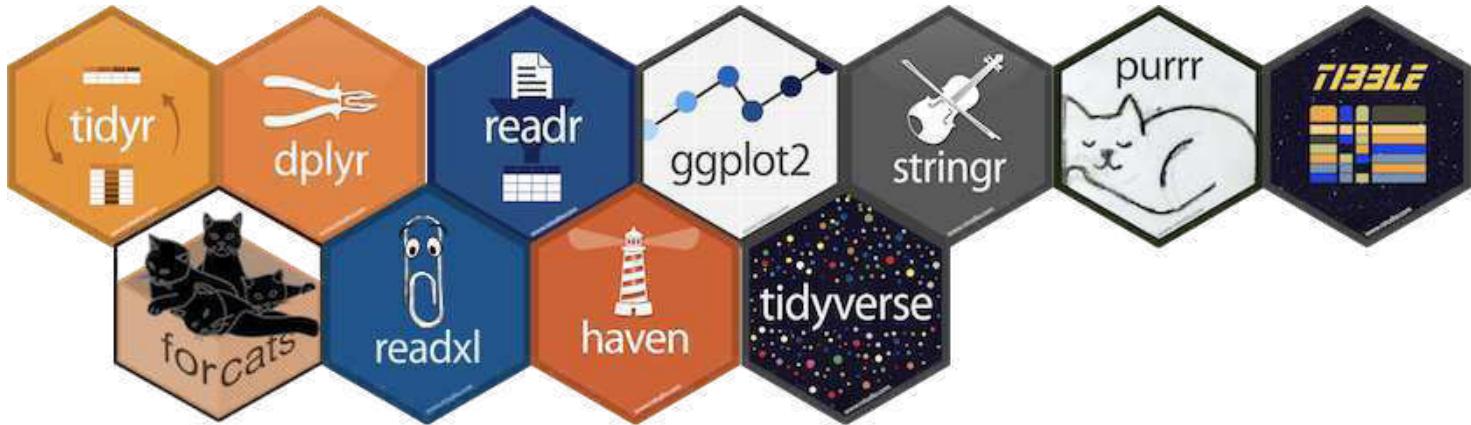
@Z3tt

# About Me

- B.Sc. in *Life Sciences* (2008-2011 in Potsdam)
- M.Sc. in *Ecology, Evolution & Nature Conservation* (2011-2014 in Potsdam)
- Ph.D. in *Ecology* (2015-2019 in Berlin)
  - movement ecology of animals
  - dynamics of populations and communities
  - ecology of wildlife diseases
- Data Visualization Designer (Practitioner/Engineer/Expert/...)
  - own and collaborative research data (*for a living*)
  - freelancing client projects (*for a living*)
  - data challenges and personal projects (*for fun & training*)

# About Me

For the preparation and visualization of all these datasets,  
I use mostly R in combination with the **tidyverse** and **ggplot2**.

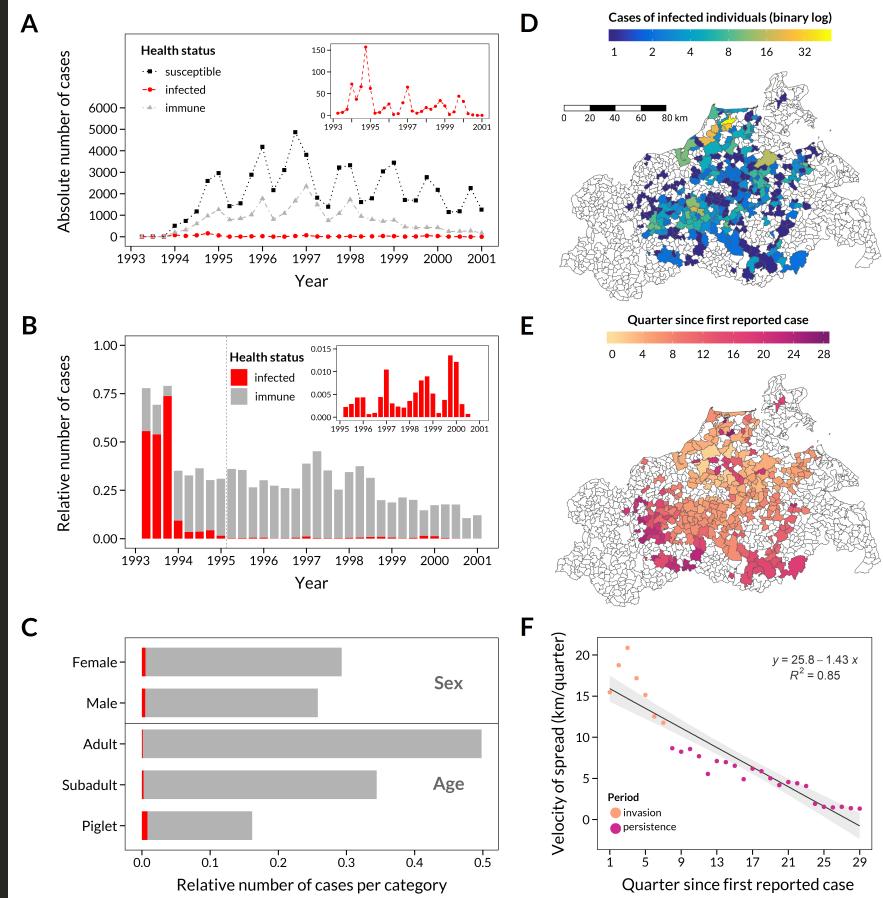


Source: [dcook.org/files/rstudio/#3](http://dcook.org/files/rstudio/#3)

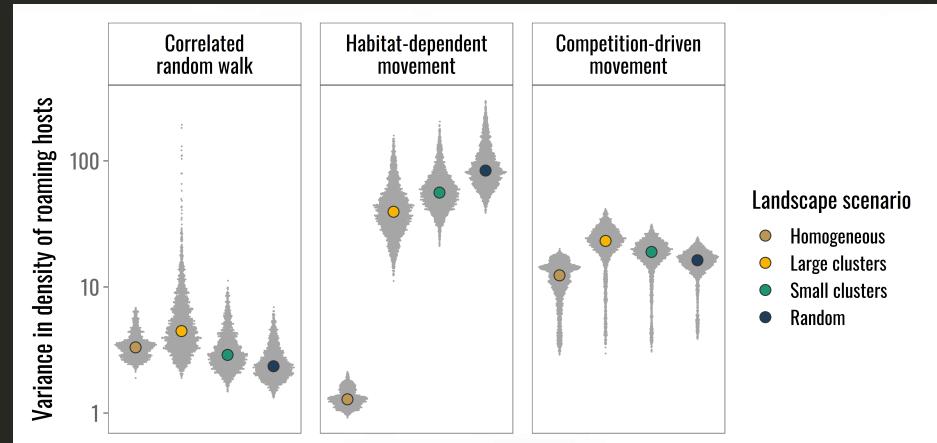
**HELLO**  
my name is

*Your Turn!*

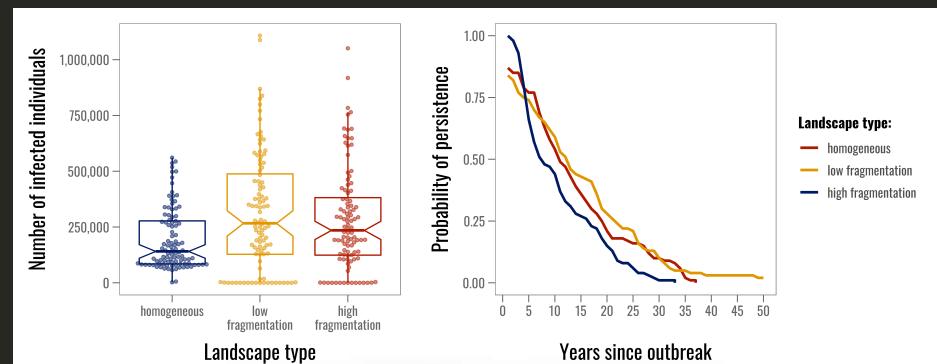
# Data Visualizations for Scientific Publications & Talks



Scherer et al. 2019 *Journal of Animal Ecology*

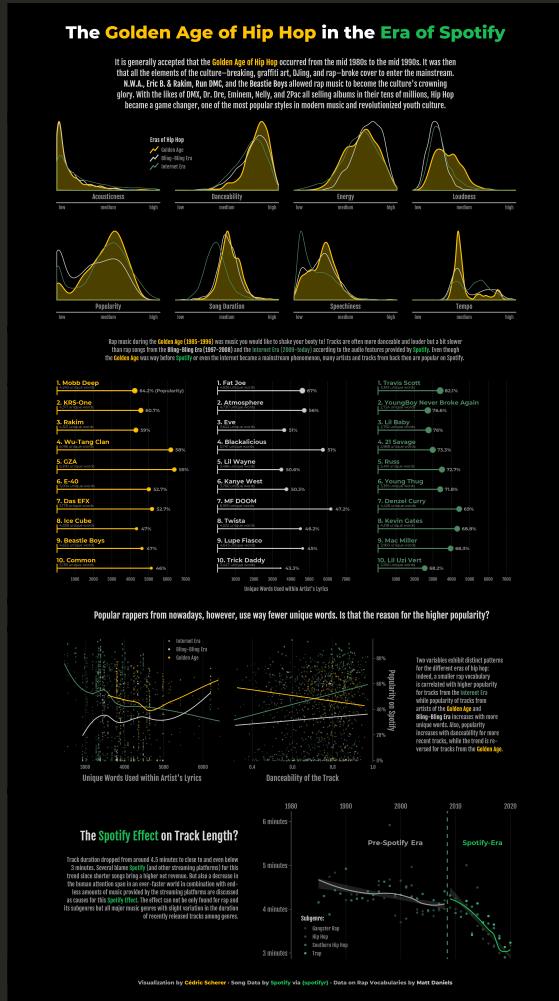


Scherer et al. 2020 *Oikos*

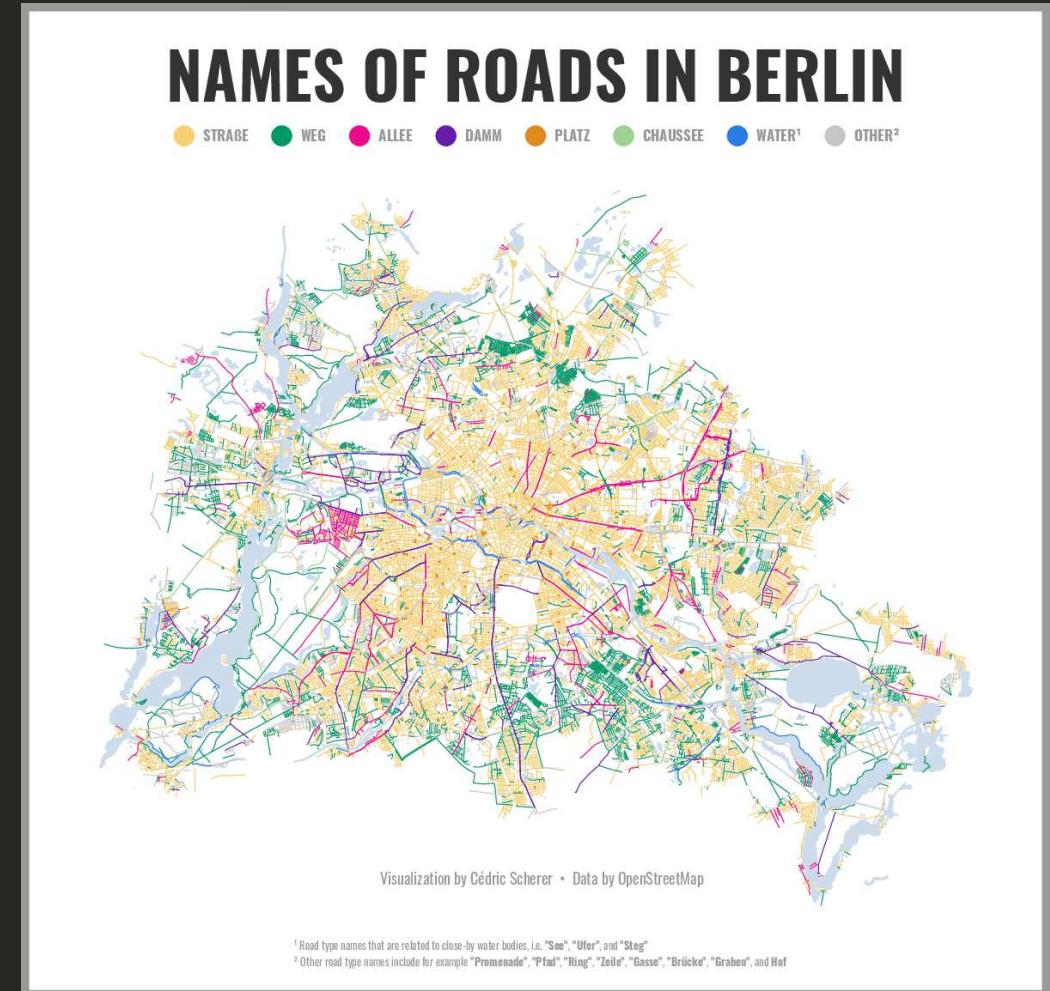


Sciaiani et al. 2019 *Methods in Ecology & Evolution*

# Data Visualizations as Challenge Contributions

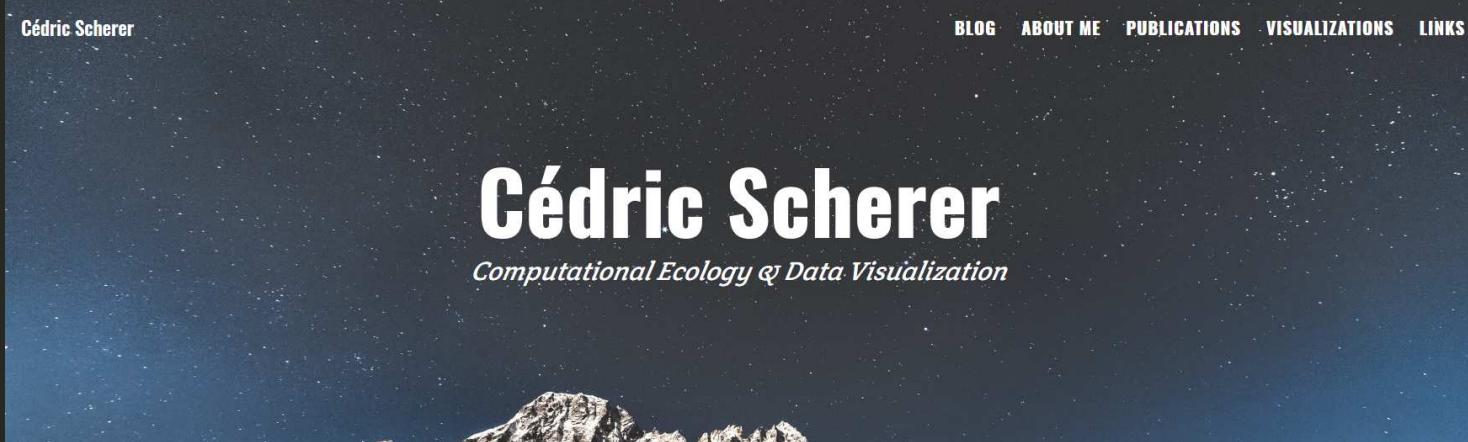


Contribution to #TidyTuesday



Contribution to #30DaymapChallenge

# My Blog



Cédric Scherer

BLOG ABOUT ME PUBLICATIONS VISUALIZATIONS LINKS

# Cédric Scherer

*Computational Ecology & Data Visualization*

---

## Comparing the Extent of the Australian Bushfires 2019/20

The massive bushfires in Australia are in the news worldwide. The incredible extent of burnt land and plume of smoke is hard to imagine so I have compared the areas to countries in Europe and worldwide.

Posted by Cédric Thursday, January 9, 2020

---

## Best TidyTuesday 2019

Here are my favorite visualizations of the #TidyTuesday challenge in 2019 (from those I've seen and which I remember). I present my personal top 3 in terms of design and storytelling.

Posted by Cédric Monday, December 30, 2019

---

## Merry (White?) Christmas!

At the end of the year, I explore the history of snow cover and white Christmas in Berlin. I wish you a merry Christmas and wonderful holidays 2019!

Posted by Cédric Tuesday, December 24, 2019

---

### ABOUT ME



Computational Ecologist, DataViz  
Enthusiast and Proud Dad



---

### FEATURED TAGS



# My Blog

Cédric Scherer

BLOG ABOUT ME PUBLICATIONS VISUALIZATIONS LINKS

R ggplots tidyverse DataViz tutorial

## A ggplot2 Tutorial for Beautiful Plotting in R

Posted by Cédric on Monday, August 5, 2019

Last update: 2019-11-01

### Introductory Words

I don't care, just show me the content!

Begin of 2016, I had to prepare my PhD introductory talk and I started using `(ggplot2)` to visualize my data since I never liked the syntax and style of base plots in R. Because I was short on time, I plotted these figures by try'n'error and with the help of lots of googling. The resource I came always back to was a blog entry called [Beautiful plotting in R: A ggplot2 cheatsheet](#) by Zev Ross, posted on 4. August 2014, updated last in January 2016. After giving the talk which contained some quite beautiful plots thanks to the blog post, I decided to go through this tutorial step-by-step. I learned so much from it and directly started modifying the codes and over the time I added some additional code snippets, chart types and resources.

Since the blog entry by Zev Ross was not updated for some years, I hosted the updated version on my GitHub. Now it finds its proper place on this homepage! [Plus I added some updates, for example the fantastic `(patchwork)` and `(ggforce)` packages. And pie charts because everyone looooves pie charts!]

Some exemplary plots included in this tutorial.

Cédric Scherer

BLOG ABOUT ME PUBLICATIONS VISUALIZATIONS LINKS

R ggplots tidyverse TidyTuesday DataViz tutorial ggplot Evolution animations

## The Evolution of a ggplot (Ep. 1)

Posted by Cédric on Friday, May 17, 2019

### The Evolution of a ggplot

Data: UNESCO Institute for Statistics  
Visualization by Cédric Scherer

- Aim of this Tutorial
- Data Preparation
- The Default Boxplot
- Sort Your Data!
- Let Your Plot Shine - Get Rid of the Default Settings
- The Choice of the Chart Type
- More Geoms, More Fun, More Info!
- Add Text Boxes to Let The Plot Speak for Itself
- Bonus: Add a Tile Map as Legend
- The Final Evolved Visualization
- Complete Code for Final Plot
- Post Scriptum: Mean versus Median

#### Aim of this Tutorial

In this series of blog posts, I aim to show you how to turn a default ggplot into a plot that visualizes information in an appealing and easily understandable way. The goal of each blog post is to provide a step-by-step tutorial explaining how my visualization have evolved from a typical basic ggplot. All plots are going to be created with 100% `(ggplot2)` and 0% Inkscape.

In the first episode, I transform a basic boxplot into a colorful and self-explanatory combination of a jittered dot strip plot and a lollipop plot. I am going to use data provided by the UNESCO on global student to teacher ratios that was selected as data #TidyTuesday for the challenge 19 of 2019.

# My GitHub

Search or jump to...

Pull requests Issues Marketplace Explore

Customize your pins

Overview Repositories 37 Projects 0 Packages 0 Stars 187 Followers 128 Following 68

 Set status

**Cédric Scherer**  
Z3tt

[Edit profile](#)

I am a computational ecologist by training and a data visualization designer by heart using mainly R for daily-business-tasks.

 Leibniz Institute for Zoo and Wildlife R...  
 Berlin  
 cedricphilippsscherer@gmail.com  
 cedricscherer.netlify.com

 PRO

**1,014 contributions in the last year**

Contribution settings ▾

2020 2019 2018 2017 2016

Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec Jan

Mon Wed Fri

Learn how we count contributions. Less More

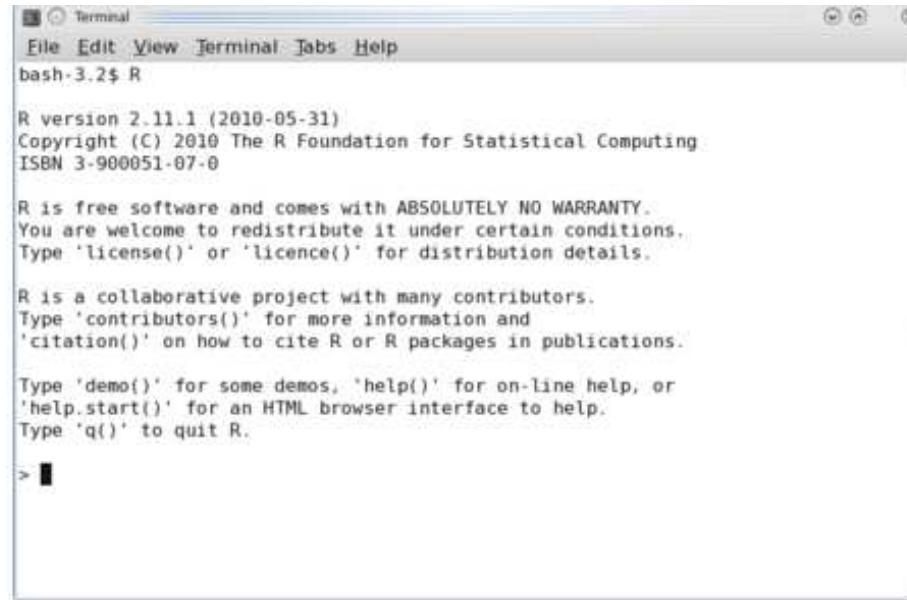
github.com/Z3tt

# Introduction to R

# What is R ?

A programming language and free software environment for statistical computing and graphics.

R is very popular among, scientists, statisticians, and data miners  
for developing statistical software and analyse and visualize data



The screenshot shows a terminal window titled "Terminal". The window title bar includes "File Edit View Terminal Tabs Help". Below the title bar, the text "bash-3.2\$ R" is visible. The main content area displays the R startup message:

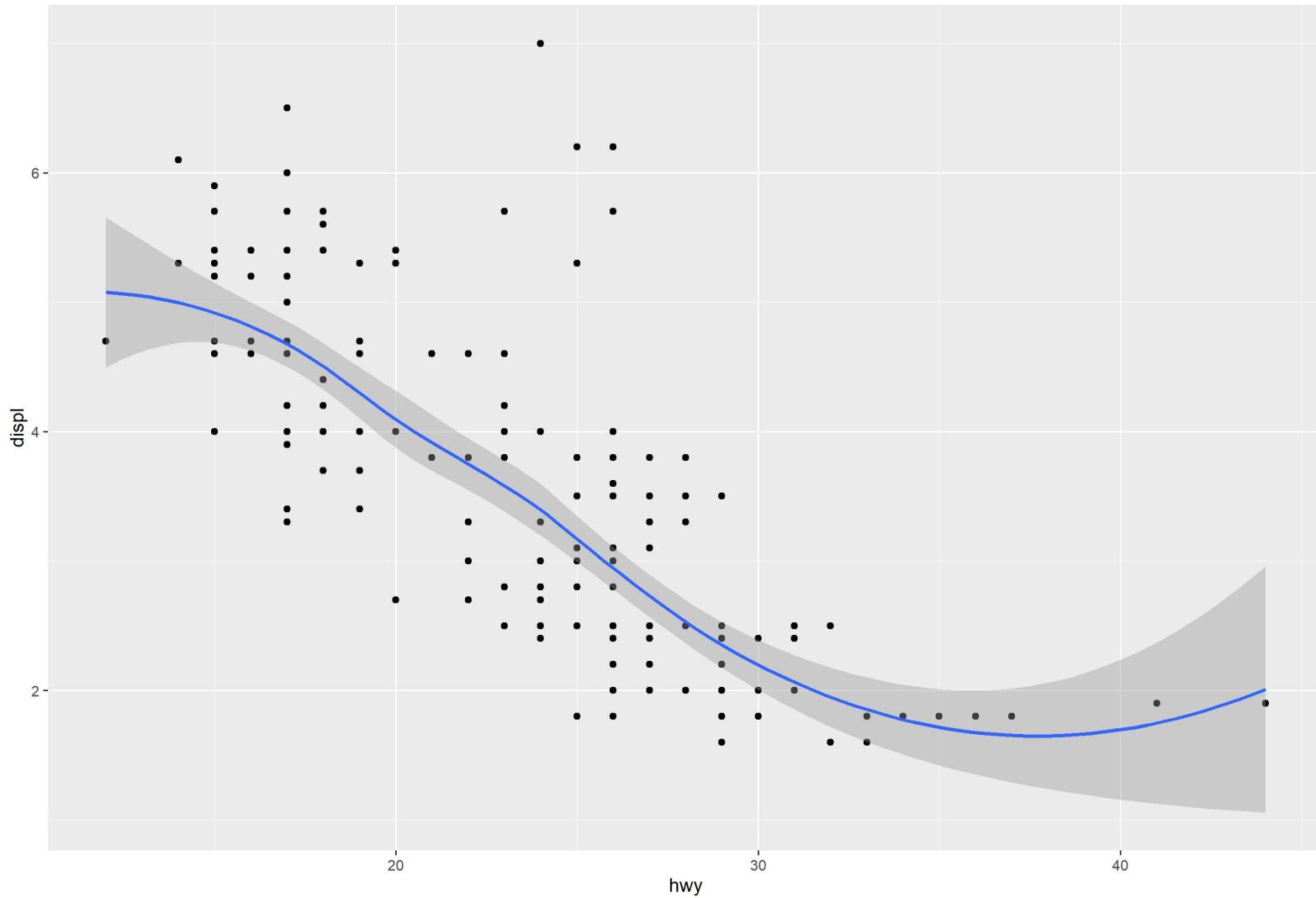
```
R version 2.11.1 (2010-05-31)
Copyright (C) 2010 The R Foundation for Statistical Computing
ISBN 3-900051-07-0

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> █
```



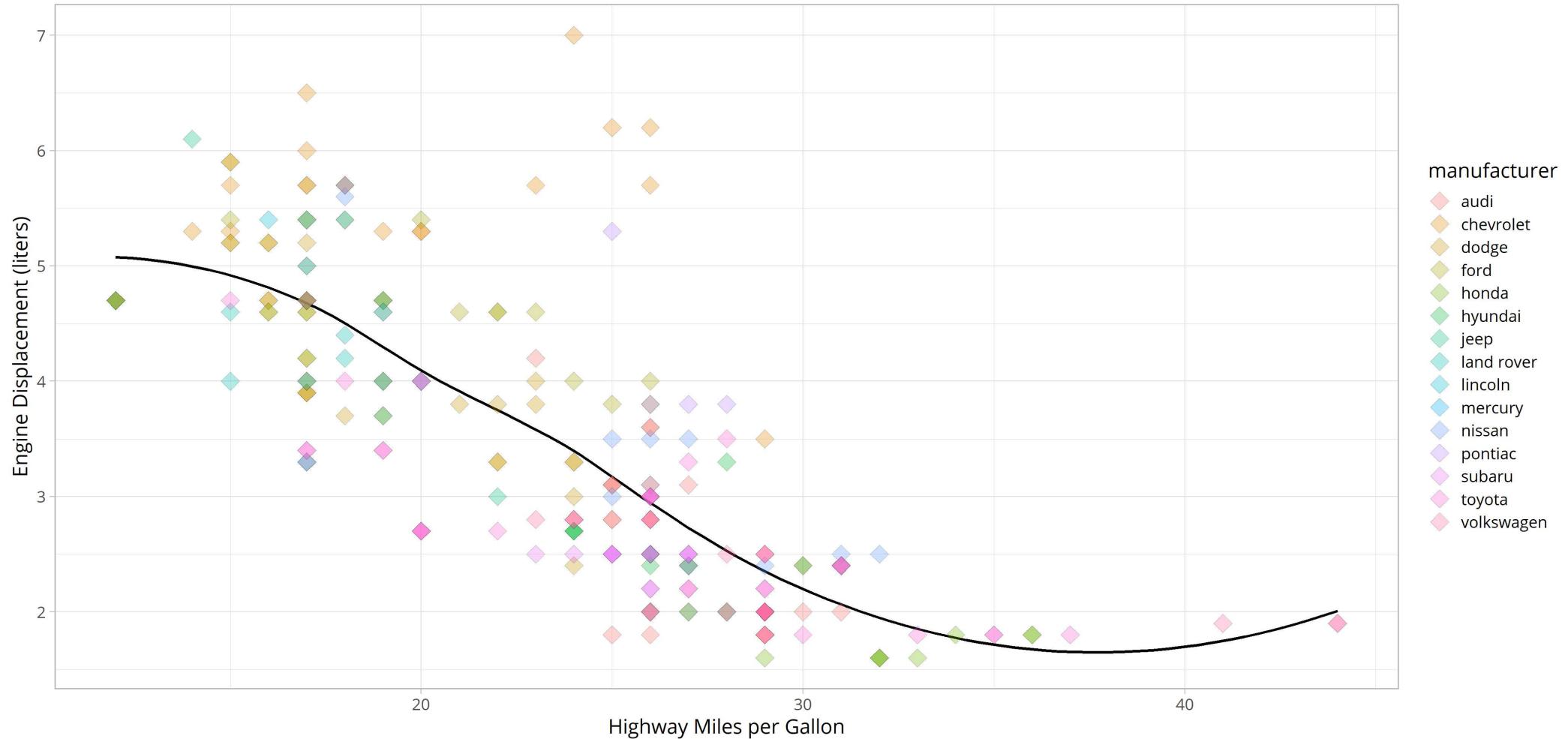
# The Code

```
library(ggplot2)

ggplot(mpg, aes(hwy, displ)) +
  geom_point() +
  stat_smooth()
```

## Weight and displacement by manufacturers of 38 popular models of cars

Simple linear regression of engine displacement (in liters) versus highway miles per gallon (MPG) for popular models of cars from 1999 to 2008.



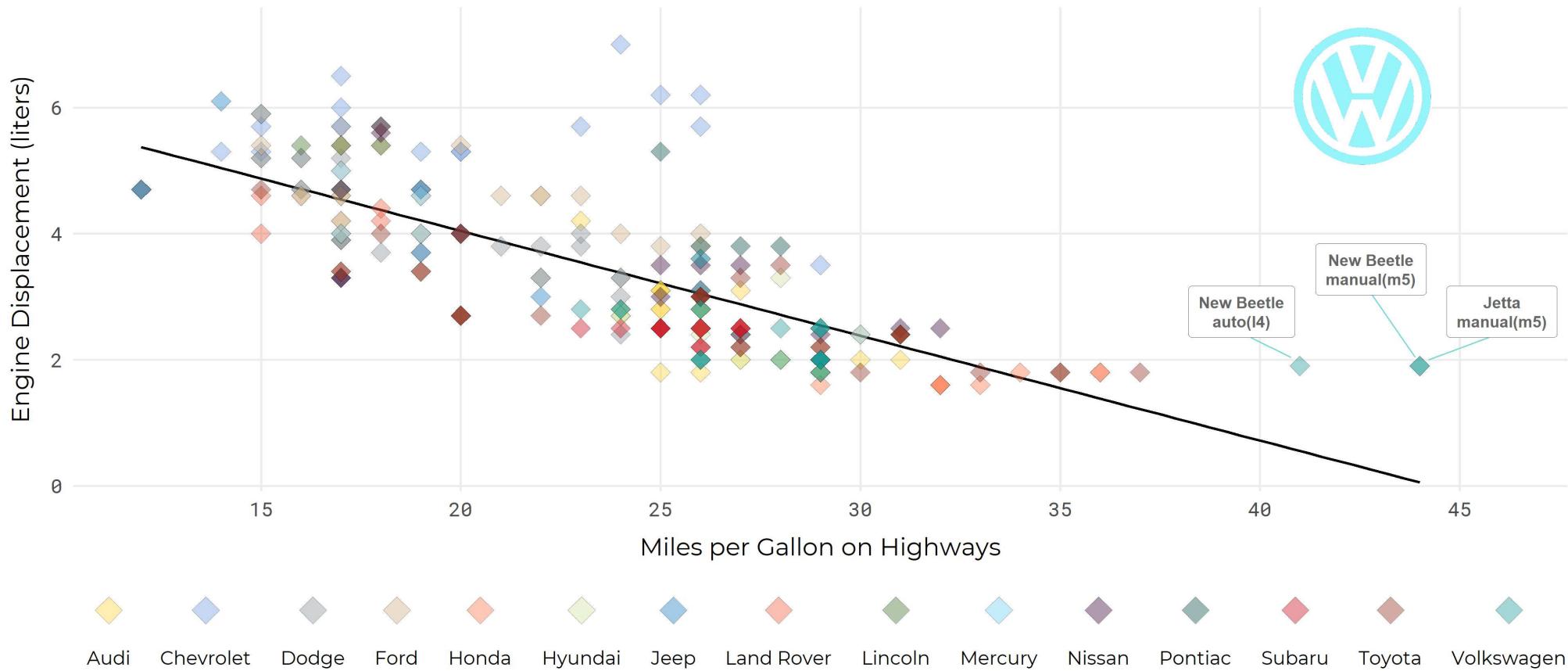
Data: EPA ([www.fueleconomy.gov](http://www.fueleconomy.gov))

# The Code

```
ggplot(mpg, aes(hwy, displ)) +  
  stat_smooth(  
    method = "lm",  
    color = "black",  
    se = F  
  ) +  
  geom_point(  
    aes(fill = manufacturer),  
    shape = 23,  
    size = 5,  
    alpha = .3,  
    color = "black",  
    stroke = .3  
  ) +  
  labs(  
    x = "Highway Miles per Gallon",  
    y = "Engine Displacement (liters)",  
    title = "Weight and displacement by manufacturers of 38 popular models of cars",  
    subtitle = "Simple linear regression of engine displacement (in liters) versus hig  
    caption = "\nData: EPA (www.fueleconomy.gov)"  
  ) +  
  theme_light()
```

# "New Beetle" and "Jetta" by Volkswagen have the worst car efficiency

Linear regression of engine displacement versus highway miles per gallon (MPG) for 38 popular models of cars from 1999 to 2008.



# What is R?

R was developed by Ross Ihaka and Robert Gentleman as an open source implementation of the S programming language. R was conceived in 1992 and released in 1995.

Pros:

- free and open source
- platform-independent
- dedicated packages that provide extra functionality
- highly compatible with many other programming languages
- huge online community
- often experienced as *easy to code* (from a non-programmer perspective)

Cons:

- data handling (physical memory storage + speed)
- security issues (relevant for web applications)
- often experienced as *strange to code* (from a programmer perspective)

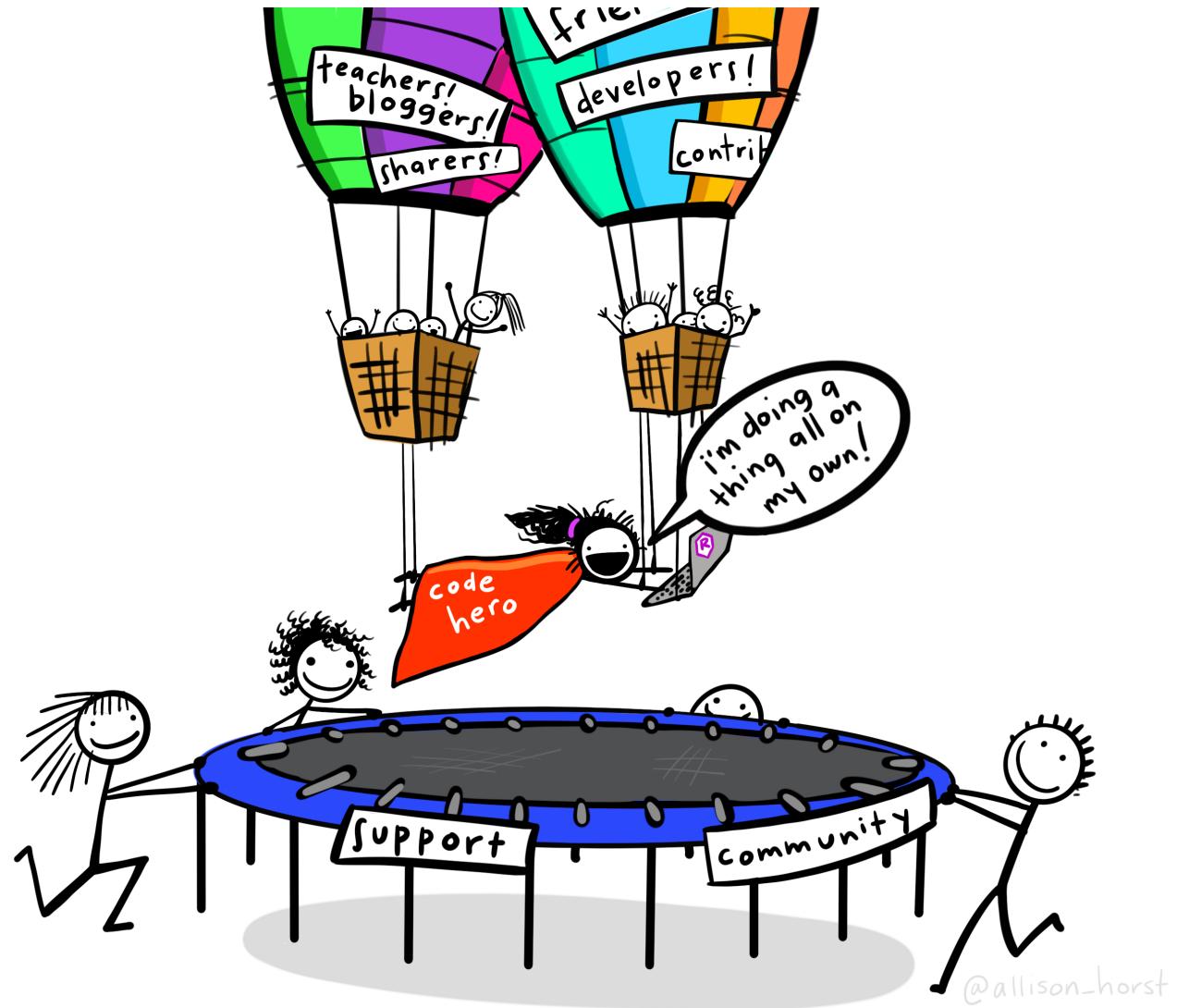


Illustration by Allison Horst ([github.com/allisonhorst/stats-illustrations](https://github.com/allisonhorst/stats-illustrations))

@allison\_horst

# Getting Help with R

- R Foundation Help Page: [r-project.org/help.html](http://r-project.org/help.html)
- RStudio Community: [community.rstudio.com](http://community.rstudio.com)
- R For Data Science Online Learning Community: [rfordatasci.com](http://rfordatasci.com)
- Stack Overflow: [stackoverflow.com](http://stackoverflow.com)
- R User Groups (local): [jumpingrivers.github.io/meetingsR/r-user-groups.html](http://jumpingrivers.github.io/meetingsR/r-user-groups.html)
- R Ladies (global & local): [rladies.org](http://rladies.org)
- Twitter: [#rstats](#), [#tidyverse](#), [#ggplot2](#)

# #rstats on Twitter

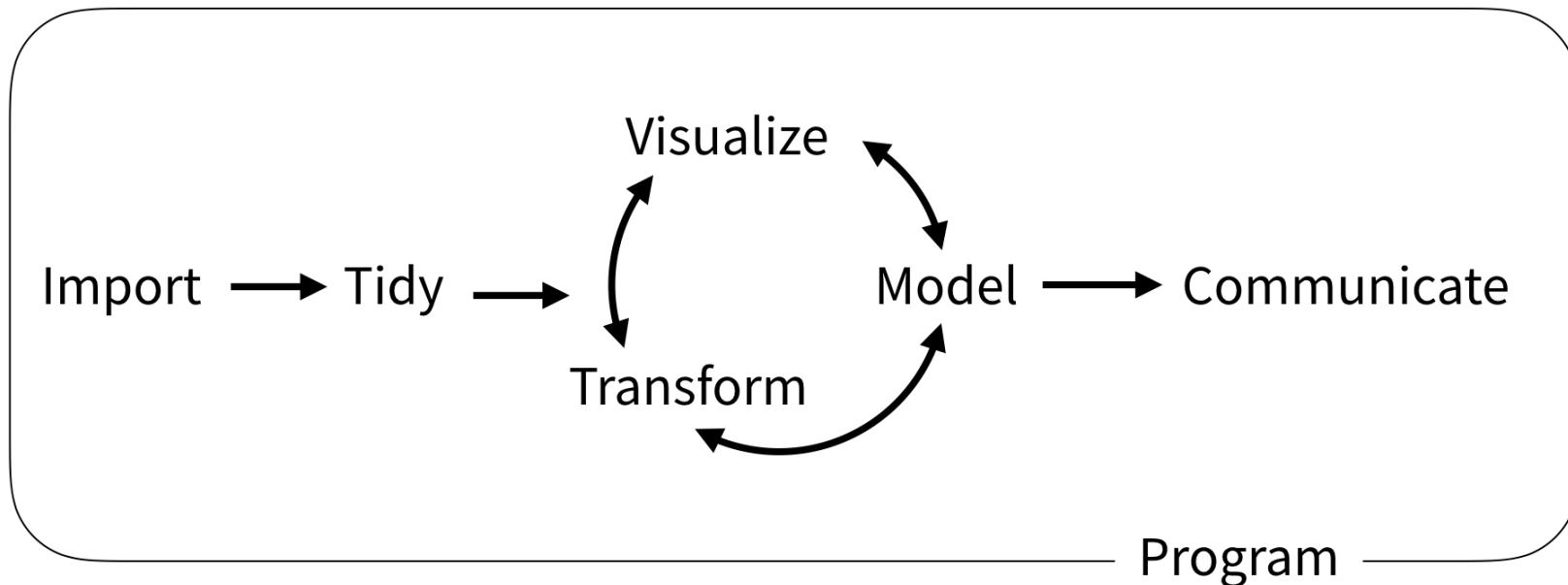
- #rstats, #tidyverse, #ggplot2
- R For Data Science Online Learning Community:  
@R4DScommunity
- R Weekly: @rweekly\_org
- ROpenSci: @rOpenSci
- TidyTuesday challenge: @thomas\_mock and @tidypod
- R For The Rest Of Us: @rfortherest
- R Function A Day: @Rfunction\_a\_day
- Mara Averick: @dataandme



Illustration by Allison Horst ([github.com/allisonhorst/stats-illustrations](https://github.com/allisonhorst/stats-illustrations))

# What is the **tidyverse**?

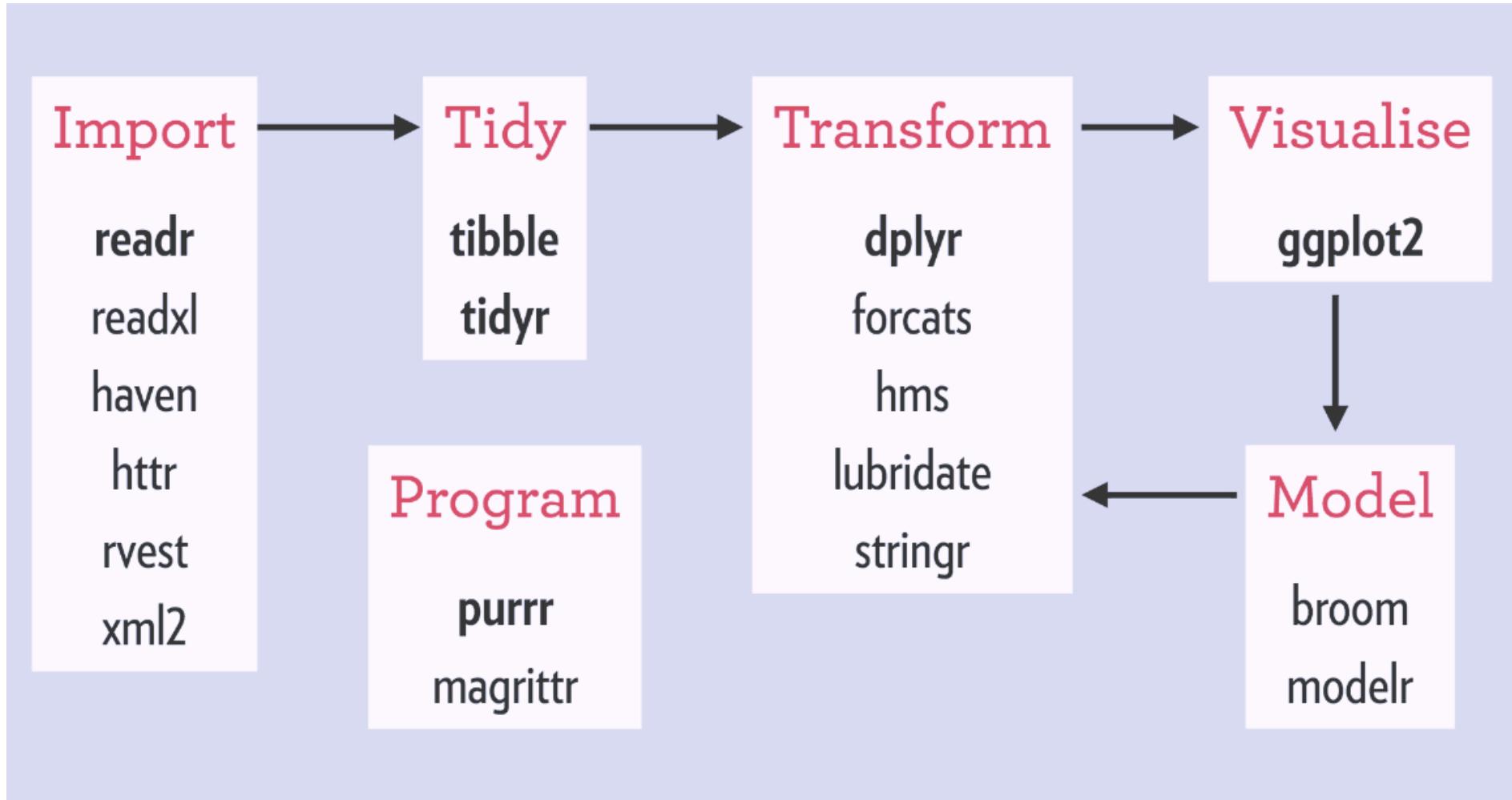
The **tidyverse** provides exemplary support for data wrangling and is the main reason for the recent popularity of **R**, especially in data-driven environments.



*Source: rstudio-education.github.io/tidyverse-cookbook*

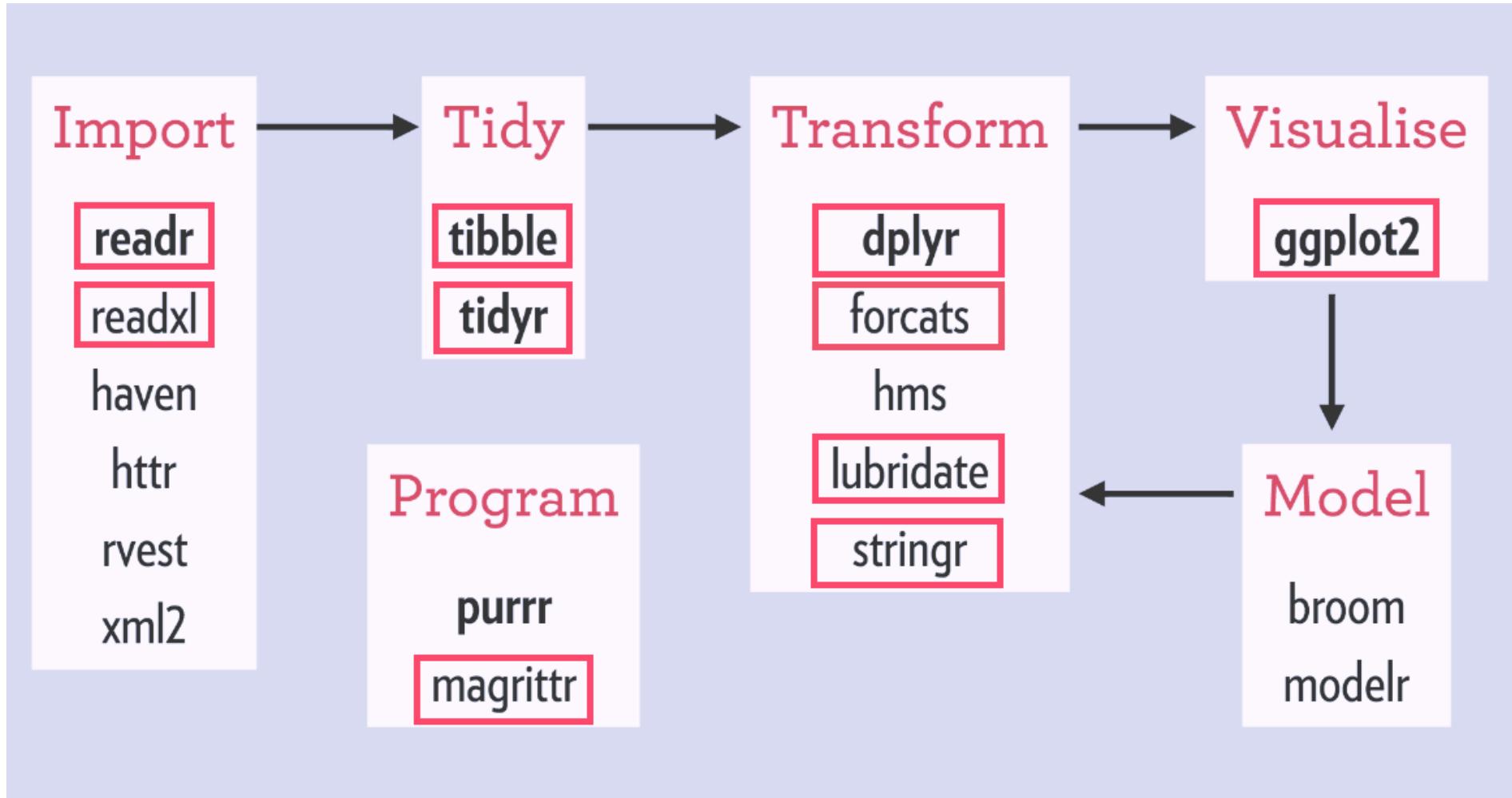
*Modified from Hadley Wickham's "R for Data Science" (R4DS)*

# The Typical Data Science Project Flow



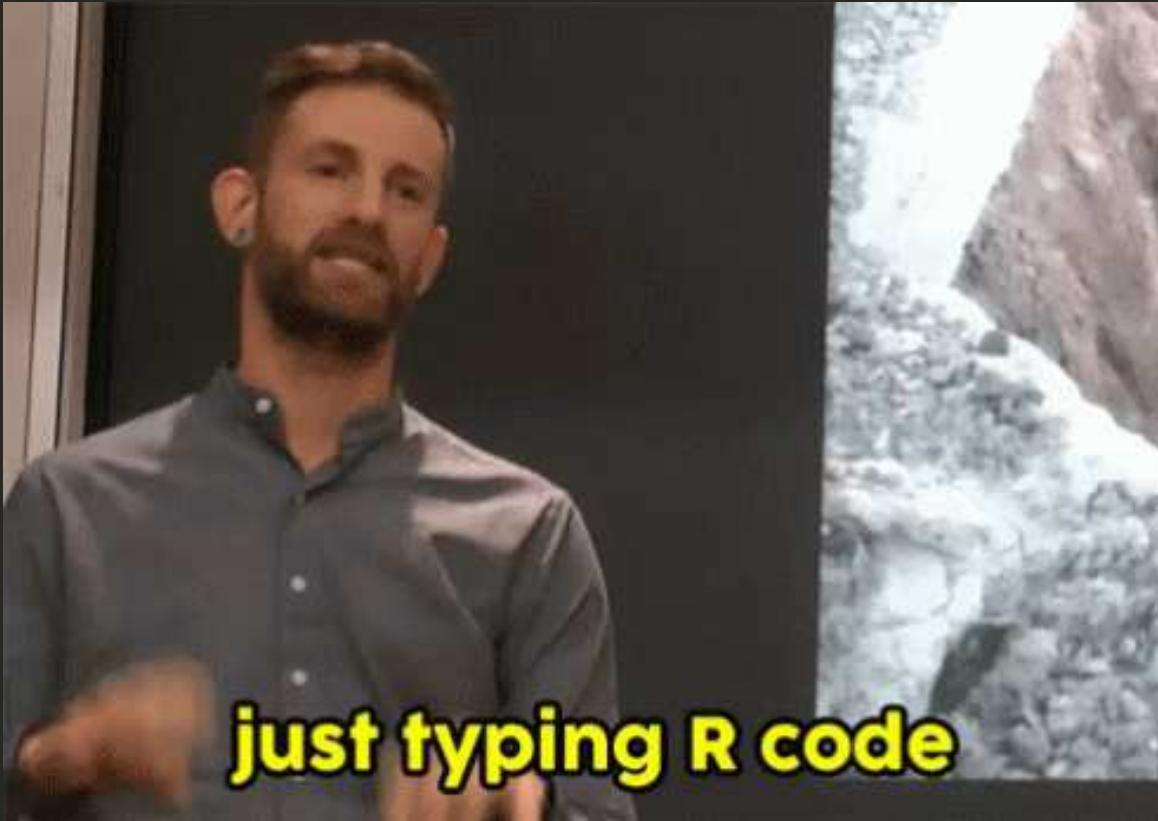
Source: [www.rviews.rstudio.com/post/2017-06-09-What-is-the-tidyverse\\_files/tidyverse1.png](http://www.rviews.rstudio.com/post/2017-06-09-What-is-the-tidyverse_files/tidyverse1.png)

# The Typical Data Science Project Flow



Source: [www.rviews.rstudio.com/post/2017-06-09-What-is-the-tidyverse\\_files/tidyverse1.png](http://www.rviews.rstudio.com/post/2017-06-09-What-is-the-tidyverse_files/tidyverse1.png)

# Who is Hadley Wickham?



# Who is Hadley Wickham?

Statistician from New Zealand, living in Houston, TX

Adjunct Professor of Statistics at the University of Auckland,  
Stanford University, and Rice University

Chief Scientist at RStudio

Inventor/Developer of {ggplot2} and the tidyverse

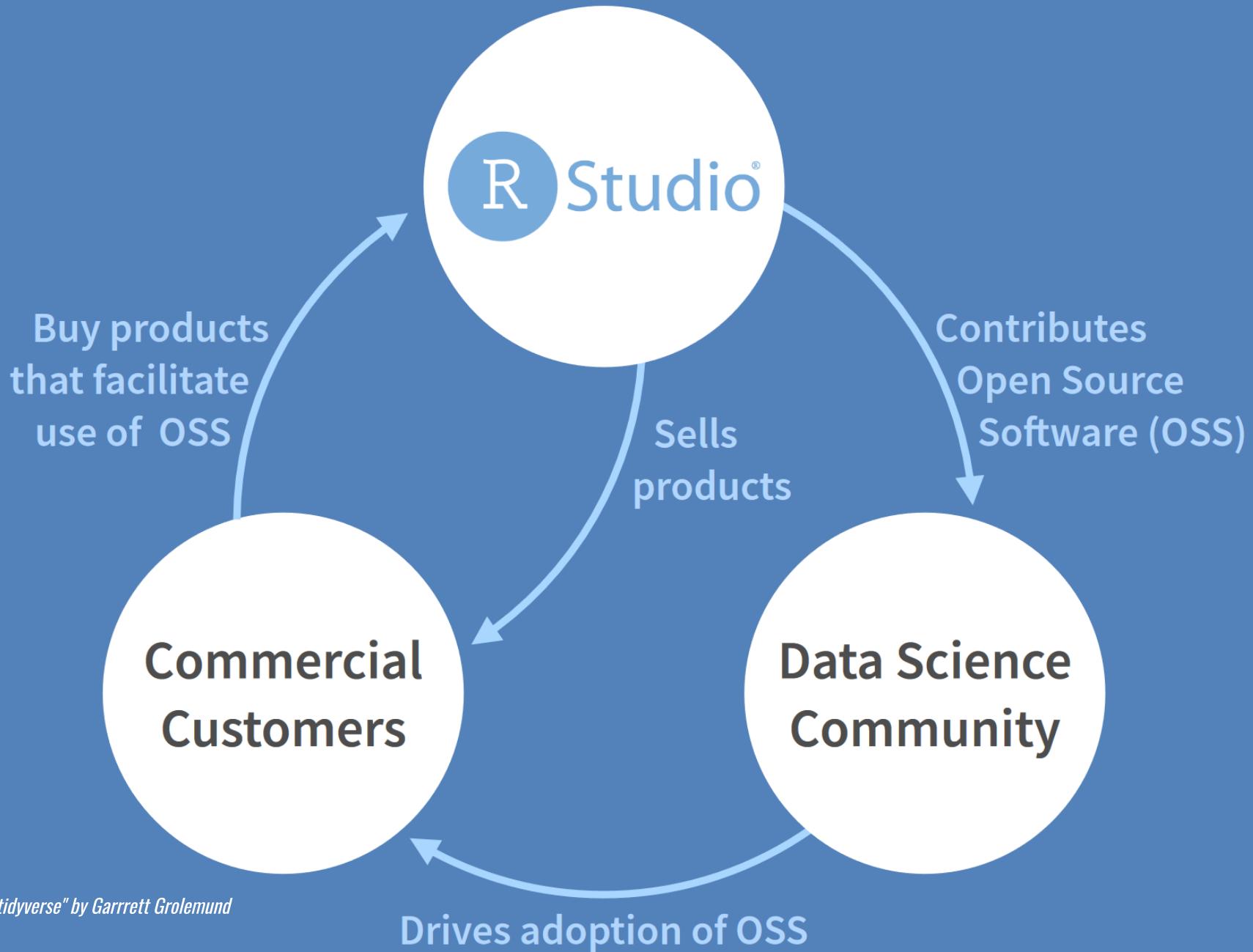
# RStudio IDE

# R and RStudio



*Modified from moderndive.netlify.com/1-getting-started.html*

- RStudio is an open-source IDE (integrated development environment) for R
- most popular R IDE since several years (*what is tinnR?!*)
- many features + extensions to facilitate workflows (*projects, notebooks, toc, ...*)
- availability of R projects and Rmarkdown (*later more*)



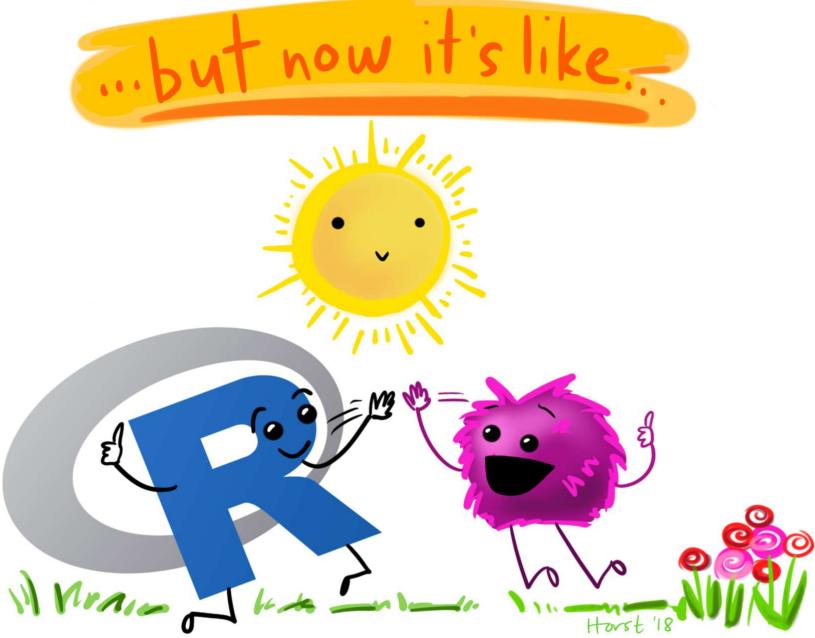
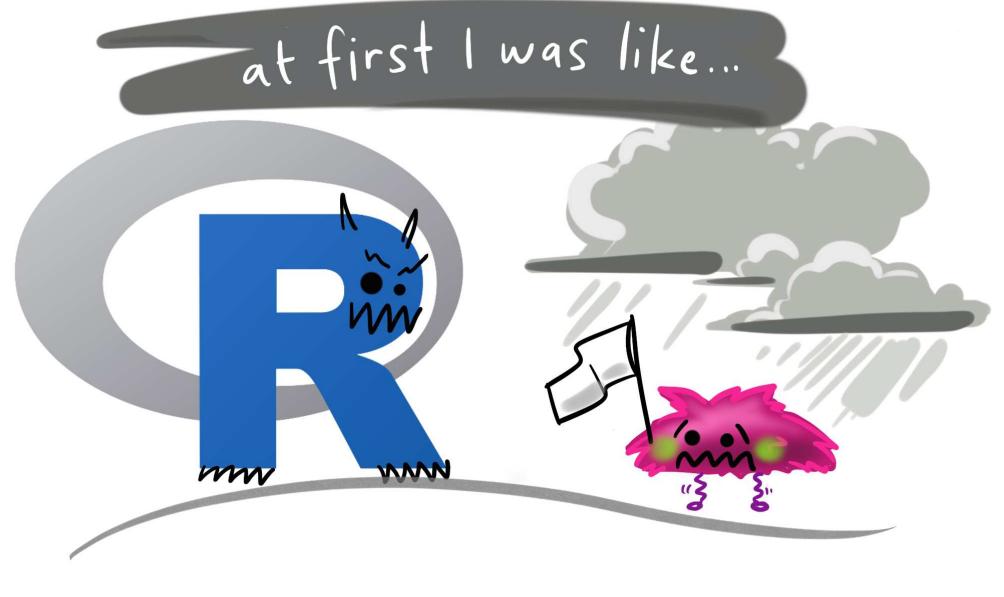


Illustration by Allison Horst ([github.com/allisonhorst/stats-illustrations](https://github.com/allisonhorst/stats-illustrations))

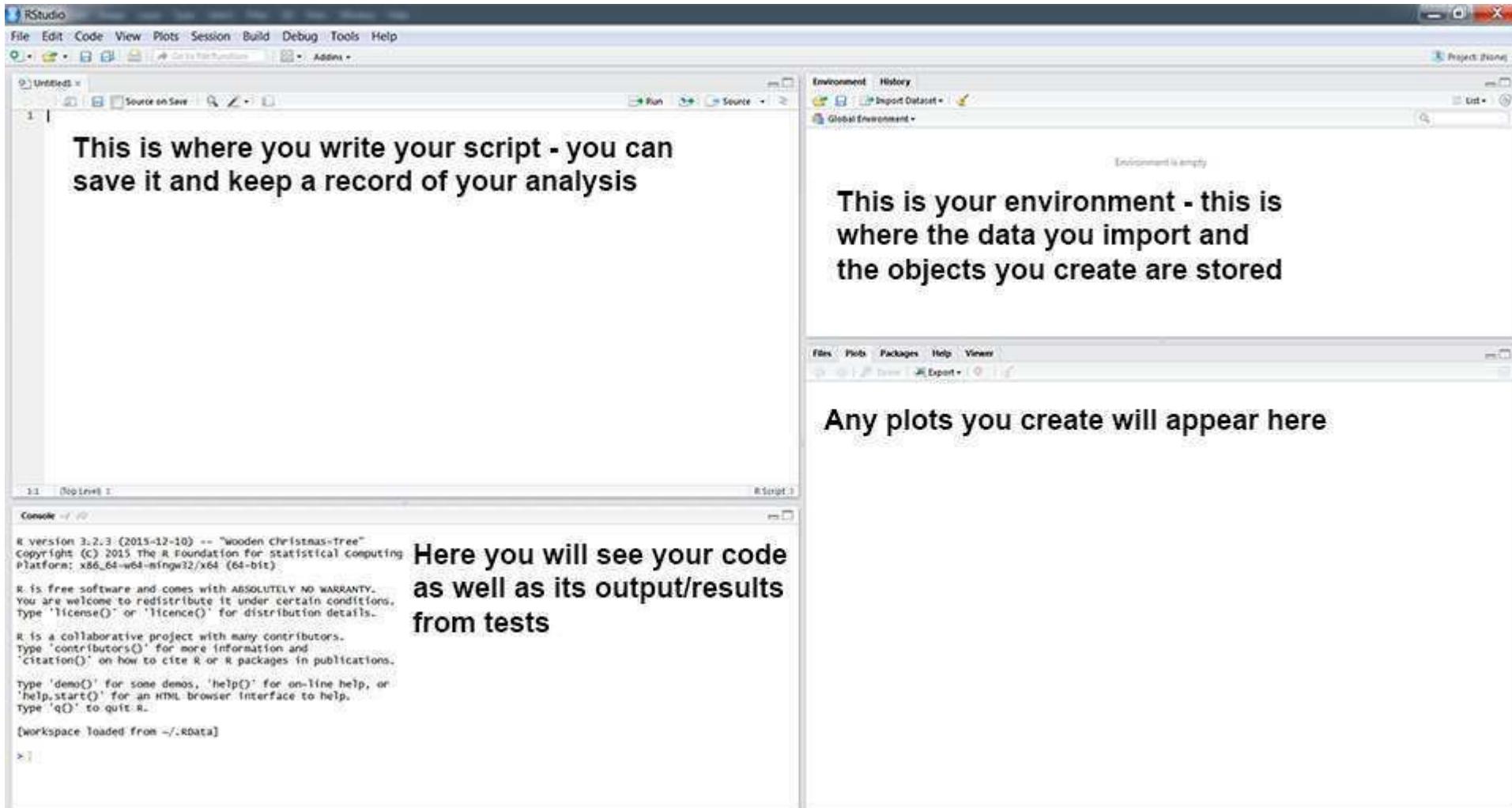
# Installation of R and RStudio

- Download and install R via  
<https://cloud.r-project.org/>
- Download and install RStudio Desktop via  
<https://www.rstudio.com/products/rstudio/download/>

# Your Turn!

- Open RStudio and get familiar with its environment.
- Go to *Help > Cheatsheets > RStudio IDE Cheat Sheet* and study the document.

# RStudio IDE

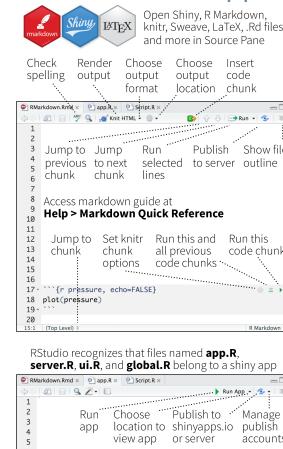


Source: [ourcodingclub.github.io/2016/11/13/intro-to-r.html](http://ourcodingclub.github.io/2016/11/13/intro-to-r.html)

# RStudio IDE

RStudio IDE :: CHEAT SHEET

## Documents and Apps



## Debug Mode

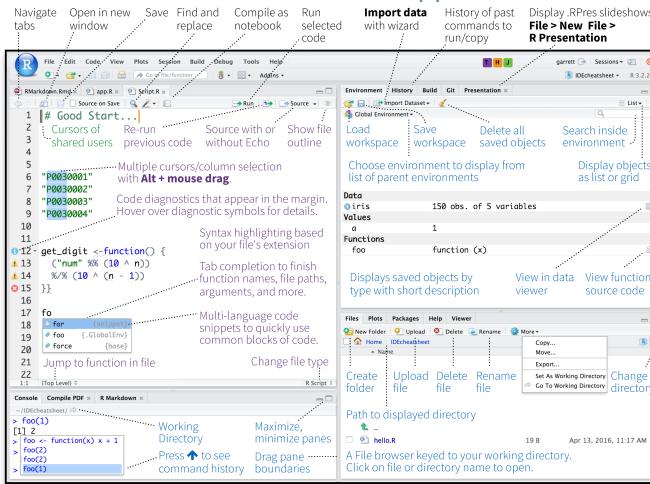
Open with **debug()**, **browser()**, or a breakpoint. RStudio will open the debugger mode when it encounters a breakpoint while executing code.

The screenshot shows the PyCharm IDE with a debugger window open. A tooltip points to the line of code 'x = 10 \* x'. The status bar at the bottom right shows 'Step into' and 'cocotb'. The code editor shows a function definition:

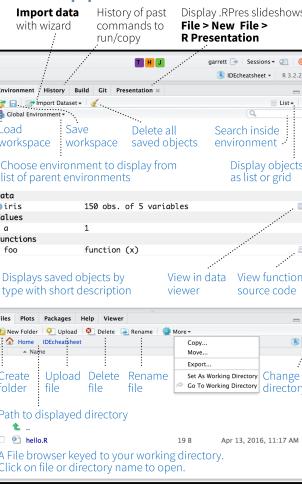
```
def test():
    x = 10
    y = 20
    z = 30
    # calculate the sum of x, y and z
    result = x + y + z
    # check if the calculated sum is correct
    assert result == 60, "The result should be 60"
    print("The result is correct")
```



## Write Code



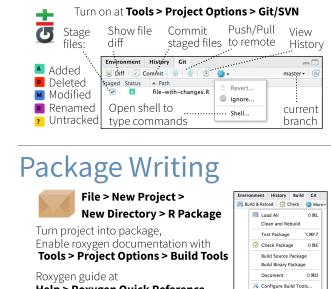
R Support



## Pro Features



Version Control with Git or SVN



## Package Writing



# RStudio IDE

## Write Code

Navigate tabs Open in new window Save Find and replace Compile as notebook Run selected code

The screenshot shows the RStudio IDE interface. The top menu bar includes File, Edit, Code, View, Plots, Session, Build, Debug, Tools, and Help. The main area displays R code in a script editor:

```
1 # Good Start...
2 Cursors of shared users Re-run previous code Source with or without Echo Show file outline
3 "P0030001"
4 "P0030002"
5 "P0030003"
6 "P0030004"
7
8 get_digit <- function() {
9   ("num" %% (10 ^ n))
10  %% (10 ^ (n - 1))
11 }
12
13 for (foo in .GlobalEnv) {
14   force {base}
15 }
16
17 foo
18
19
20
21 Jump to function in file
22
```

Annotations explain various features:

- Cursors of shared users
- Re-run previous code
- Show file outline
- Multiple cursors/column selection with **Alt + mouse drag**.
- Code diagnostics that appear in the margin. Hover over diagnostic symbols for details.
- Syntax highlighting based on your file's extension
- Tab completion to finish function names, file paths, arguments, and more.
- Multi-language code snippets to quickly use common blocks of code.
- Change file type
- Jump to function in file
- Working Directory
- Maximize, minimize panes
- Press **↑** to see command history
- Drag pane boundaries

The bottom panel contains a Console tab showing:

```
> foo(1)
[1] 2
> foo <- function(x) x + 1
> foo(2)
[1] 3
> foo(2)
[1] 3
> foo(1)
```

Annotations for the console include:

- Working Directory
- Maximize, minimize panes
- Press **↑** to see command history
- Drag pane boundaries

## R Support

Import data with wizard History of past commands to run/copy Display .RPres slideshows  
**File > New File > R Presentation**

The screenshot shows the RStudio IDE interface with the Environment and Files panes open.

**Environment Pane:**

- Import Dataset
- Global Environment
- Load workspace
- Save workspace
- Delete all saved objects
- Search inside environment

**Data Pane:**

- Choose environment to display from list of parent environments
- Display objects as list or grid

**Files Pane:**

- 150 obs. of 5 variables
- Values
- Functions
- View in data viewer
- View function source code

**Files Tab:**

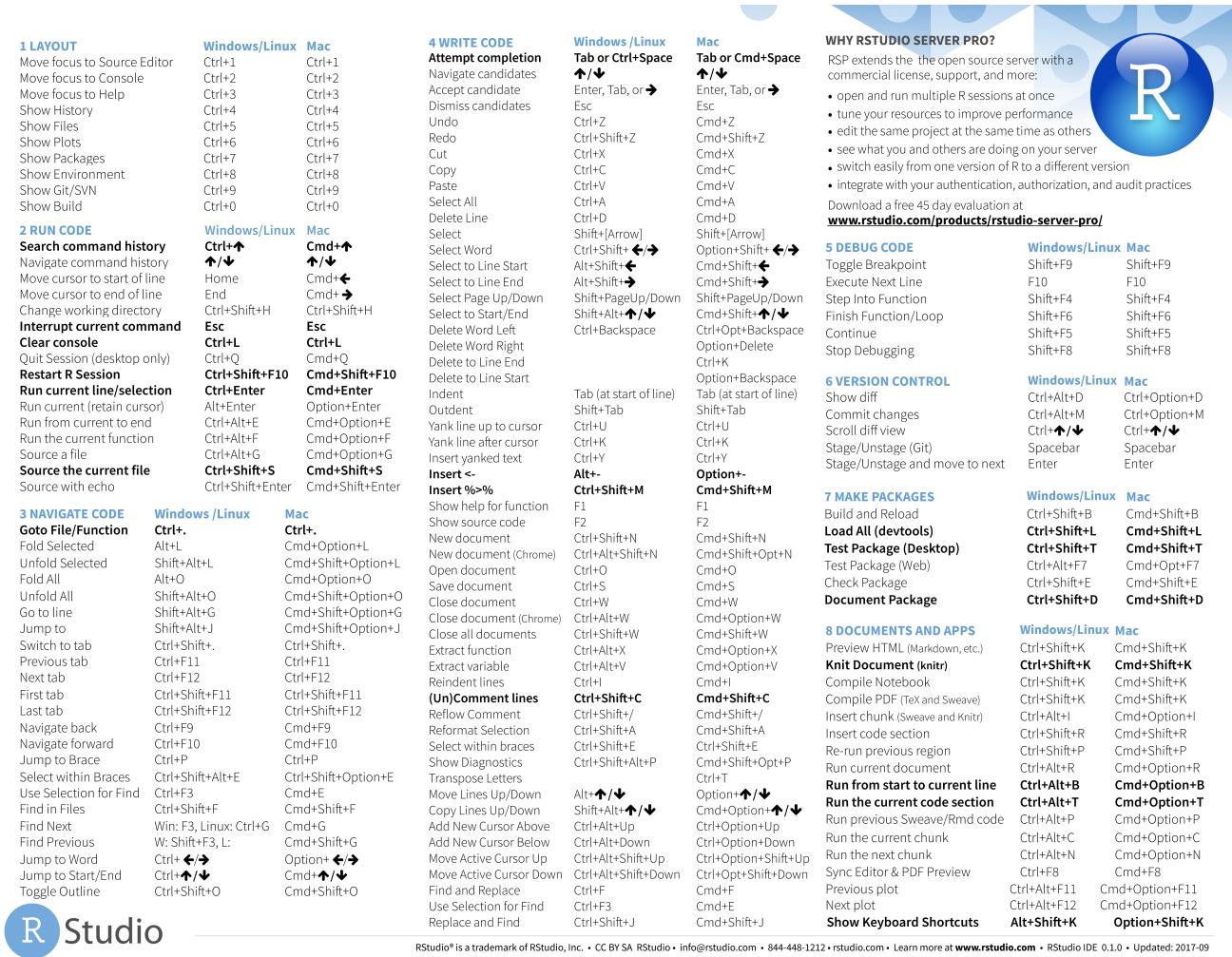
- New Folder
- Upload
- Delete
- Rename
- More
- Copy...
- Move...
- Export...
- Set As Working Directory
- Change directory

Annotations for the files pane include:

- Path to displayed directory
- Change directory
- Working Directory
- Maximize, minimize panes
- Press **↑** to see command history
- Drag pane boundaries

Source: [rstudio.com/resources/cheatsheets/](http://rstudio.com/resources/cheatsheets/)

# RStudio IDE



RStudio® is a trademark of RStudio, Inc. • CC BY SA RStudio • info@rstudio.com • 844-448-1212 • rstudio.com • Learn more at [www.rstudio.com](http://www.rstudio.com) • RStudio IDE 0.1.0 • Updated: 2017-09

# Hello World !

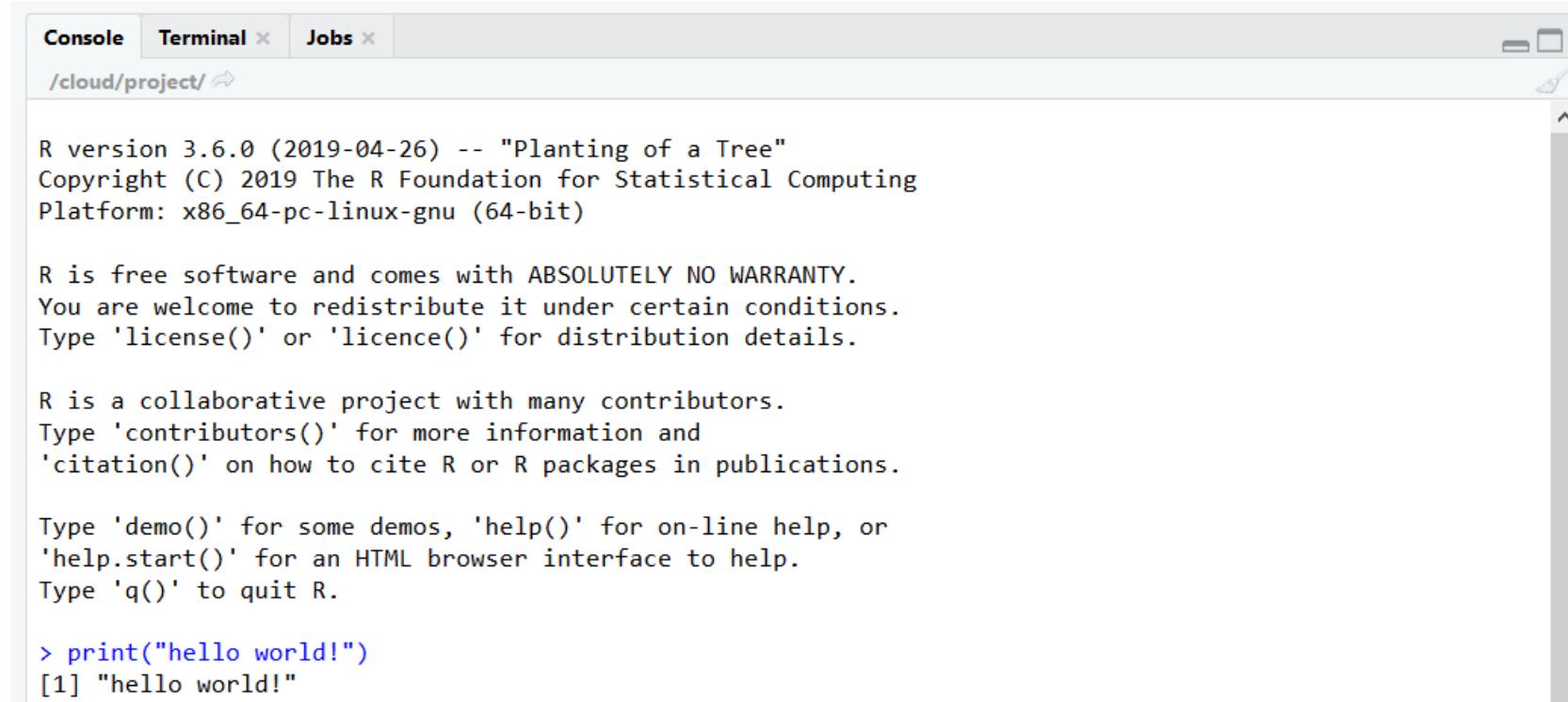
# Your Turn!

- Open a script by clicking *File > New File > R script*
- Type the classic `print("hello world!")` and hit the *Run* button.  
(alternatively: **Ctrl/Cmd + Enter**)
- What happens if you just run "hello world!"?

# The Console

R code you are running appears here - the line starts with > (by default).

If there is anything R returns, it will also be printed here after your call - here starting with [1] .



The screenshot shows a software interface with a top navigation bar containing 'Console', 'Terminal', and 'Jobs' tabs. The 'Console' tab is active. Below the tabs is a URL bar with the text '/cloud/project/'. The main area displays the R startup message, followed by a command '> print("hello world!")' and its output '[1] "hello world!"'. The interface has a light gray background and includes standard window controls (minimize, maximize, close) in the top right corner.

```
R version 3.6.0 (2019-04-26) -- "Planting of a Tree"
Copyright (C) 2019 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> print("hello world!")
[1] "hello world!"
```

Btw, you can also type code directly in the console and hit Enter to evaluate it.

# Values

1

"hello world!"

"2020-03-02 09:00:00"

# Your Turn!

- Type `2 + 3` and run it.
- Play around with other calculus operators such as `-`, `*`, `/`, `^`, or `sqrt`.
- What is the difference between the first 5 and the `sqrt` calculation?

# R is a calculator!

```
2 + 2
## [1] 4

(59 + 73 + 2) * 5
## [1] 670

1 / 200 * 30
## [1] 0.15

sin(pi / 2)
## [1] 1

10^12 * sqrt(4312)
## [1] 6.566582e+13

log(exp(5))
## [1] 5
```

# Your Turn!

- Run `x <- 1` and afterwards `x`
  - Next, run `x + 2`.
  - Now run `x <- 5` and again `x + 2`.
- 
- Why is there no output when you run the first command?
  - What is the arrow symbol `<-` doing?
  - What is the value of `x` in the end?
  - What if you type `y <- x` and what if `y <- x <- 2`
  - And what happens if you type `y <- 2 <- x`?

# Assignments

**Values** - 1, "Florida", "2010-01-25"

**Objects** - x <- 22/7

**Values** - 1, "Florida", "2010-01-25"

**Objects** - x <- 22/7

A name  
without quotes

< followed by -  
(it looks like an arrow)

A value,  
object, or  
function result

# Assignments

New objects are created with `<-` and the process is called assignment:

```
x <- 24  
x  
## [1] 24
```

# Assignments

All **R** statements where you create objects have the same form:

```
object_name <- value
```

(When reading that code above say “*object x gets value 24*” in your head.)

# Assignments

All **R** statements where you create objects **have the same form**:

```
object_name <- value
```

You can also create them by using the equal sign:

```
object_name = value
```

This practice is controversially discussed with many people arguing it can cause confusion.

However, the **=** way is much more common in other programming languages thus some prefer it.

```
y = "hello world"  
y  
## [1] "hello world"
```

# Your Turn!

Which of these are numbers?

1

"1"

"one"

one

# The Environment

All assigned objects belong to the *global environment* called `R_GlobalEnv`\*.

You can find an overview of everything you have defined in the *environment pane*.

The screenshot shows the RStudio interface with the 'Environment' tab selected. The pane displays a table of values under the heading 'Values'. There are two entries: 'x' with the value '24' and 'y' with the value '"hello world!"'. The interface includes standard RStudio toolbars and a search bar at the top.

Values	
x	24
y	"hello world!"

\* There are also other types of environments but we do not bother with that.

# Object Names

# Object Names

## Object names

- must start with a letter
- can only contain letters, numbers, `_` and `.`
- should be descriptive
- ideally follow a convention for multiple words
- ideally follow a convention for different classes/types

`i_use_snake_case`

`otherPeopleUseCamelCase`

`some.people.use.periods`

`And_aFew.People_RENOUNCEconvention`

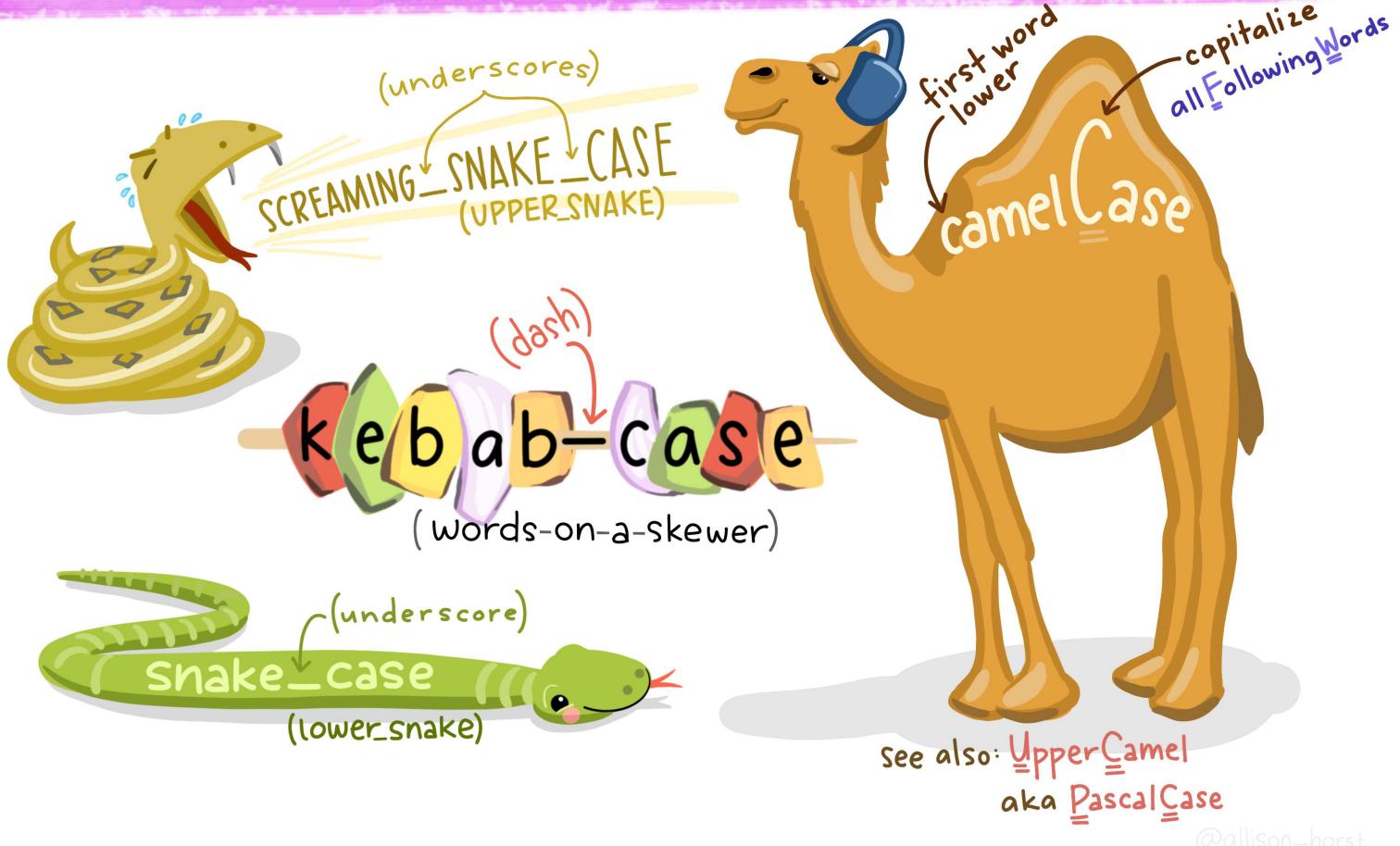
```
df_us_population <- a
```

```
DfUsPopulation <- b
```

```
df.us.population <- c
```

```
dfUS_population <- d
```

# in that case...



# Functions

# Functions

R has a large collection of built-in functions that are called like this:

```
function_name(arg1 = val1, arg2 = val2, ...)
```

# Functions

R has a large collection of built-in functions that are called like this:

```
function_name(arg1 = val1, arg2 = val2, ...)
```

We already have seen some functions in the exercise before: `+`, `-`, and `sqrt` all are functions!

# Functions

```
`+`  
## function (e1, e2) .Primitive("+")  
  
`-`  
## function (e1, e2) .Primitive("-")  
  
sqrt  
## function (x) .Primitive("sqrt")  
  
mean  
## function (x, ...)  
## UseMethod("mean")  
## <bytecode: 0x0000000009bc2610>  
## <environment: namespace:base>
```

# Functions

```
citation
## function (package = "base", lib.loc = NULL, auto = NULL)
## {
##   if (!is.null(auto) && !is.logical(auto) && !any(is.na(match(c("Package",
##     "Version", "Title"), names(meta <- as.list(auto)))))) &&
##     !all(is.na(match(c("Authors@R", "Author"), names(meta))))) {
##     auto_was_meta <- TRUE
##     package <- meta$Package
##   }
##   else {
##     auto_was_meta <- FALSE
##     dir <- system.file(package = package, lib.loc = lib.loc)
##     if (dir == "") {
##       stop(packageNotFoundError(package, lib.loc, sys.call()))
##     }
##     meta <- packageDescription(pkg = package, lib.loc = dirname(dir))
##     citfile <- file.path(dir, "CITATION")
##     test <- file_test("-f", citfile)
##     if (!test) {
##       citfile <- file.path(dir, "inst", "CITATION")
##       test <- file_test("-f", citfile)
##     }
##     if (is.null(auto))
```

# **Functions** - round(x, digits = 3)

A name  
without  
quotes

followed by  
() to run the  
function

Arguments:  
values, objects, or  
function results

# Your Turn!

Which of these will work supposing `one <- 1`?

`log(1)`    `log("1")`    `log("one")`    `log(one)`

# Functions

```
one <- 1

log(1)
## [1] 0

log("1")
## Error in log("1"): non-numeric argument to mathematical function

log("one")
## Error in log("one"): non-numeric argument to mathematical function

log(one)
## [1] 0
```

# Functions

```
citation()
##
## To cite R in publications use:
##
##   R Core Team (2019). R: A language and environment for statistical
##   computing. R Foundation for Statistical Computing, Vienna, Austria.
##   URL https://www.R-project.org/.
##
## A BibTeX entry for LaTeX users is
##
## @Manual{,
##   title = {R: A Language and Environment for Statistical Computing},
##   author = {{R Core Team}},
##   organization = {R Foundation for Statistical Computing},
##   address = {Vienna, Austria},
##   year = {2019},
##   url = {https://www.R-project.org/},
## }
##
## We have invested a lot of time and effort in creating R, please cite it
## when using it for data analysis. See also 'citation("pkgname")' for
## citing R packages.
```

# Functions

```
sqrt(9)  
## [1] 3  
mean(5, 1, 3)  
## [1] 5
```



*via NBA on GIPHY*

# Functions

?mean()

mean [base]

R Documentation

## Arithmetic Mean

### Description

Generic function for the (trimmed) arithmetic mean.

### Usage

```
mean(x, ...)

## Default S3 method:
mean(x, trim = 0, na.rm = FALSE, ...)
```

### Arguments

**x** An R object. Currently there are methods for numeric/logical vectors and date, date-time and time interval objects. Complex vectors are allowed for `trim = 0`, only.  
**trim** the fraction (0 to 0.5) of observations to be trimmed from each end of `x` before the mean is computed. Values of `trim` outside that range are taken as the nearest endpoint.  
**na.rm** a logical value indicating whether NA values should be stripped before the computation proceeds.  
**...** further arguments passed to or from other methods.

### Value

If `trim` is zero (the default), the arithmetic mean of the values in `x` is computed, as a numeric or complex vector of length one. If `x` is not logical (coerced to numeric), numeric (including integer) or complex, `NA_real_` is returned, with a warning. If `trim` is non-zero, a symmetrically trimmed mean is computed with a fraction of `trim` observations deleted from each end before the mean is computed.

### References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

### See Also

[weighted.mean](#), [mean.POSIXct](#), [colMeans](#) for row and column means.

### Examples

```
x <- c(0:10, 50)
xm <- mean(x)
c(xm, mean(x, trim = 0.10))
```

# Data Types

# Data Types: Classes

Atomic classes in R:

```
class(1)
## [1] "numeric"

class("a")
## [1] "character"

class(TRUE) ## T is short for 'TRUE', F is short for 'FALSE',
## [1] "logical"

class(1L)
## [1] "integer"

class(1 + 0i)
## [1] "complex"
```

# Data Types: Vectors

As in the case of the `mean` function we need a *numerical/logical vector*:

```
v <- c(5, 1)
class(v)
## [1] "numeric"

v2 <- c(TRUE, FALSE)
class(v2)
## [1] "logical"

v3 <- c(1L, 2L)
class(v3)
## [1] "integer"
```

# Data Types: Vectors

As in the case of the `mean` function we need a *numerical/logical vector*:

```
v <- c(5, 1)
class(v)
## [1] "numeric"

v2 <- c(TRUE, FALSE)
class(v2)
## [1] "logical"

v3 <- c(1L, 2L)
class(v3)
## [1] "integer"
```

... and there are also *character vectors*:

```
v4 <- c("Physalia", "DataViz")
class(v4)
## [1] "character"
```

# Data Types: Vectors

You create *atomic vectors* by using `c(val1, val2, ...)`.

Atomic vectors only contain one data type and all values will be *coerced implicitly* to the same data type:

```
(v5 <- c("ggplot", 2020, TRUE))
## [1] "ggplot" "2020"   "TRUE"
class(v5)
## [1] "character"

(v6 <- c(1L, 0.2, 3 + 0i))
## [1] 1.0+0i 0.2+0i 3.0+0i
class(v6)
## [1] "complex"
```

# Data Types: Vectors

You create *atomic vectors* by using `c(val1, val2, ...)`.

One can *explicitly coerce* atomic vectors to a particular data type:

```
(v5 <- c("ggplot", 2020, TRUE))
## [1] "ggplot" "2020"   "TRUE"
class(v5)
## [1] "character"

(v6 <- c(1L, 0.2, 3 + 0i))
## [1] 1.0+0i 0.2+0i 3.0+0i
class(v6)
## [1] "complex"

as.numeric(v5)
## [1] NA 2020 NA

as.character(v6)
## [1] "1+0i"   "0.2+0i" "3+0i"

as.logical(v6)
## [1] TRUE TRUE TRUE
```

# Your Turn!

- Fix our calculation of the mean of 5, 1, and 3.
- Guess what the following vectors look like without running them first:
  - `a <- c(1.7, "Physalia")` and `as.numeric(a)`
  - `b <- c(TRUE, 2020, 1.375)` and `as.complex(b)`
  - `c <- c(TRUE, "TRUE", "T", FALSE, "False", "f")` and `as.logical(c)`
  - `as.integer(c(45L, 0.237, 4.9))`
- What does `as.factor(c("red", "blue))` do?

# Function Arguments

# Function Arguments

```
mean(x = c(5, 1, 3))
## [1] 3

## or:
my_vector <- c(5, 1, 3)
mean(x = my_vector)
## [1] 3
```

# Function Arguments

```
mean(x = c(5, 1, 3))
## [1] 3

max(x = 5)
## [1] 5
max(x = c(5, 1, 3))
## [1] 5

quantile(x = c(5, 1, 3), probs = 0.25)
## 25%
##      2
```

# Implicit Matching of Arguments

```
mean(x = c(5, 1, 3))  
## [1] 3
```

```
max(x = 5)  
## [1] 5  
max(x = c(5, 1, 3))  
## [1] 5
```

```
quantile(x = c(5, 1, 3), probs = 0.25)  
## 25%  
## 2
```

```
mean(c(5, 1, 3))  
## [1] 3
```

```
max(5)  
## [1] 5  
max(c(5, 1, 3))  
## [1] 5
```

```
quantile(c(5, 1, 3), 0.25)  
## 25%  
## 2
```

# Implicit Matching of Arguments

```
mean(x = c(5, 1, 3))  
## [1] 3
```

```
max(x = 5)  
## [1] 5  
max(x = c(5, 1, 3))  
## [1] 5
```

```
quantile(x = c(5, 1, 3), probs = 0.25)  
## 25%  
## 2
```

```
mean(c(5, 1, 3))  
## [1] 3
```

```
max(5)  
## [1] 5  
max(c(5, 1, 3))  
## [1] 5
```

```
quantile(c(5, 1, 3), 0.25)  
## 25%  
## 2
```

**BUT:**

```
quantile(0.25, c(5, 1, 3))  
## Error in quantile.default(0.25, c(5, 1, 3)): 'probs' outside [0,1]  
quantile(0.25, c(1, 0.345))  
## 100% 34.5%  
## 0.25 0.25
```

# Coercion

# Implicit and Explicit Coercion

```
(a <- c(1.7, "Physalia"))
## [1] "1.7"      "Physalia"

(b <- c(TRUE, 2020, 1.375))
## [1] 1.000 2020.000 1.375

(c <- c(TRUE, "TRUE", "T", FALSE, "False", "f"))
## [1] "TRUE"   "TRUE"   "T"      "FALSE"  "False"  "f"
```

# Implicit and Explicit Coercion

```
(a <- c(1.7, "Physalia"))
## [1] "1.7"      "Physalia"

(b <- c(TRUE, 2020, 1.375))
## [1] 1.000 2020.000 1.375

(c <- c(TRUE, "TRUE", "T", FALSE, "False", "f"))
## [1] "TRUE"    "TRUE"    "T"       "FALSE"   "False"   "f"

as.numeric(a)
## [1] 1.7 NA

as.complex(b)
## [1] 1.000+0i 2020.000+0i     1.375+0i

as.logical(c)
## [1] TRUE  TRUE  TRUE FALSE FALSE    NA

as.integer(c(45L, 0.237, 4.9))
## [1] 45   0    4
```

# Implicit Coercion

The coercion rule goes:

**logical** → **integer** → **numeric** → **complex** → **character**

```
class(c(TRUE, 1L, 1.2, 1 + 0i, "a"))
## [1] "character"

class(c(TRUE, 1L, 1.2, 1 + 0i))
## [1] "complex"

class(c(TRUE, 1L, 1.2))
## [1] "numeric"

class(c(TRUE, 1L))
## [1] "integer"

class(c(TRUE))
## [1] "logical"
```

# Factors

# Factors

Conceptually, factors are variables which take only a predefined number of values (i.e. categorical variables).

Factors are important for *ordered categorical data*, a thing we will need often when working with **ggplot2**:

```
v4
## [1] "Physalia" "DataViz"
as.factor(v4)
## [1] Physalia DataViz
## Levels: DataViz Physalia
factor(v4)
## [1] Physalia DataViz
## Levels: DataViz Physalia

factor(v4, levels = c("Physalia", "DataViz"))
## [1] Physalia DataViz
## Levels: Physalia DataViz

factor(v4, levels = c("Physalia", "DataViz"),
       labels = c("Physalia Courses Berlin", "Data Visualization"))
## [1] Physalia Courses Berlin Data Visualization
## Levels: Physalia Courses Berlin Data Visualization
```

# Factors

Conceptually, factors are variables which take only a predefined number of values (i.e. categorical variables).

Factors are important for *ordered categorical data*, a thing we will need often when working with **ggplot2**:

```
f <- factor(v4, levels = c("Physalia", "DataViz"),
             labels = c("Physalia Courses Berlin", "Data Visualization"))

class(f)
## [1] "factor"
levels(f)
## [1] "Physalia Courses Berlin" "Data Visualization"

as.character(f)
## [1] "Physalia Courses Berlin" "Data Visualization"
as.numeric(f)
## [1] 1 2
```

# Factors

Conceptually, factors are variables which take only a predefined number of values (i.e. categorical variables).

Factors are important for *ordered categorical data*, a thing we will need often when working with **ggplot2**:

```
factor(c(0, 0, 1, 0, 1), levels = c(1, 0), labels = c("no", "yes"))
## [1] yes yes no   yes no
## Levels: no yes

factor(c(0, 0, 1, 0, 1, 2), levels = c(1, 0), labels = c("no", "yes"))
## [1] yes yes no   yes no   <NA>
## Levels: no yes
```

# Missing Data

# Missing Values

Missing values are denoted by **NA** (*not available*),  
undefined mathematical operations by **NaN** (*not a number*) in **R**.

```
(my_experiment <- c(0.61, 0.43, NA, 0.96))
## [1] 0.61 0.43  NA 0.96
is.na(my_experiment)
## [1] FALSE FALSE  TRUE FALSE
!is.na(my_experiment)
## [1]  TRUE  TRUE FALSE  TRUE

0 / 0
## [1] NaN
is.nan(0 / 0)
## [1] TRUE
is.na(0 / 0)
## [1] TRUE

is.nan(my_experiment)
## [1] FALSE FALSE FALSE FALSE
```

# Missing Values

Since `NA` values are unknown, most comparisons and calculations will return `NA` as well if there is any missing data:

```
max(my_experiment)
## [1] NA
mean(my_experiment)
## [1] NA

max(my_experiment, na.rm = TRUE)
## [1] 0.96
mean(my_experiment, na.rm = TRUE)
## [1] 0.6666667

NA == 1
## [1] NA
NA == NA
## [1] NA
NA == NaN
## [1] NA
```

# Tabular Data

# Data Frames

You can store several vectors (with different data types) as tabular data in so-called *data frames*:

```
(my_df <- data.frame(v, v2, v3, v4))
##   v     v2 v3      v4
## 1 5  TRUE 1 Physalia
## 2 1 FALSE 2 DataViz

## explicitly declaring column names
(my_df <- data.frame(numeric = v, logical = v2, integer = v3, character = v4))
## numeric logical integer character
## 1         5    TRUE          1 Physalia
## 2         1   FALSE          2 DataViz
```

# The New Data Frame: Meet the Tibble!

Tibbles are a modern reimagining of the `data.frame`, keeping what time has proven to be effective, and throwing out what is not.

```
(my_tbl <- tibble::tibble(numeric = v, logical = v2, integer = v3, character = v4))  
## # A tibble: 2 x 4  
##   numeric logical integer character  
##     <dbl> <lgl>    <int> <chr>  
## 1      5 TRUE        1 Physalia  
## 2      1 FALSE       2 DataViz
```

*The name comes from the way you originally created these objects: `tbl_df()`, which was most easily pronounced as “tibble diff” or “tibble d. f.”*

# The New Data Frame: Meet the Tibble!

Note:

Similarly as the functions we have used before, `tibble()` is a function.

While `mean()` and `+` are *base functions*, `tibble()` belongs to an *add-on package* which is named *tibble* as well.

To use use functions from installed packages, you either call them as I did before or by loading the package into your session via another function called `library()`:

```
## install the add-on package from the CRAN server
install.packages("tibble")

## without loading the package but using "namespace"
my_tbl <- tibble::tibble(numeric = v, logical = v2, integer = v3, character = v4)

## loading the package
library(tibble)
## all functions contained in the library are available now
my_tbl <- tibble(numeric = v, logical = v2, integer = v3, character = v4)
```

# Data Frame vs Tibble

Tibbles are a modern reimagining of the `data.frame`, keeping what time has proven to be effective, and throwing out what is not.

Main differences in the usage of a data frame vs a tibble:

- tibbles have a refined print method
  - show only the first 10 rows, and all the columns that fit on screen
  - in addition to its name, each column reports its type
  - also, the type of the data and its dimensions are shown

# Data Frame vs Tibble

```
tbl_flights
```

```
## # A tibble: 100 x 13
##   year month   day sched_dep_time
##   <int> <int> <int>          <int>
## 1 2013     1     1            515
## 2 2013     1     1            529
## 3 2013     1     1            540
## 4 2013     1     1            545
## 5 2013     1     1            600
## 6 2013     1     1            558
## 7 2013     1     1            600
## 8 2013     1     1            600
## 9 2013     1     1            600
## 10 2013    1     1            600
## # ... with 90 more rows, and 9 more
## # variables: dep_delay <dbl>,
## # sched_arr_time <int>,
## # arr_delay <dbl>, carrier <chr>,
## # flight <int>, tailnum <chr>,
## # origin <chr>, dest <chr>,
## # time_hour <dttm>
```

```
as.data.frame(tbl_flights)
```

```
##   year month   day sched_dep_time
##   <int> <int> <int>          <int>
## 1 2013     1     1            515
## 2 2013     1     1            529
## 3 2013     1     1            540
## 4 2013     1     1            545
## 5 2013     1     1            600
## 6 2013     1     1            558
## 7 2013     1     1            600
## 8 2013     1     1            600
## 9 2013     1     1            600
## 10 2013    1     1            600
## 11 2013    1     1            600
## 12 2013    1     1            600
## 13 2013    1     1            600
## 14 2013    1     1            600
## 15 2013    1     1            600
## 16 2013    1     1            559
## 17 2013    1     1            600
## 18 2013    1     1            600
## 19 2013    1     1            600
## 20 2013    1     1            600
```

# Data Frame vs Tibble

Tibbles are a modern reimagining of the `data.frame`, keeping what time has proven to be effective, and throwing out what is not.

Main differences in the usage of a data frame vs a tibble:

- tibbles have a refined print method
  - show only the first 10 rows, and all the columns that fit on screen
  - in addition to its name, each column reports its type
  - also, the type of the data and its dimensions are shown
- tibbles are strict about subsetting

# Data Frame vs Tibble

Tibbles are strict about subsetting:

```
my_df$integer  
## [1] 1 2
```

```
my_tbl$integer  
## [1] 1 2
```

```
my_df$int  
## [1] 1 2
```

```
my_tbl$int  
## NULL
```

# Data Import

# Data Import

Usually, you create data frames by loading data files such as **.txt**, **.csv**, **.xlsx**, **.rds**, ...

```
## data.frames (base R)
my_data <- read.delim("./data/cool_data.txt", sep = "\t")
my_data <- read.csv("./data/cool_data.csv")

## tibbles (tidyverse)
my_data <- readr::read_delim("./data/cool_data.txt", delim = "\t")
my_data <- readr::read_csv("./data/cool_data.csv")
my_data <- readr::read_csv("https://some_webpage.com/online_data.csv")
my_data <- readr::read_csv2("./data/actually_not_a_csv_but_okay.csv") ## ';' as sep
my_data <- readxl::read_xls("./data/oh_no_excel_data.xls", sheet = 1)
my_data <- readxl::read_xlsx("./data/oh_no_excel_data.xlsx", sheet = 1)

## one of R's data formats (base R)
my_data <- readRDS("./data/one_of_Rs_data_formats.Rds")
```

# Your Turn!

- Download or create a data set with the software of your choice and import it into R.
- Inspect the imported data using `summary()`, `str()`, `tibble::glimpse()`, and `View()`.
- Also explore other functions such as `dim()`, `length()`, `unique()`, and `range()`.
- Save your data set as .Rds (use `saveRDS()` and `?saveRDS()` to get help).

# Data Summaries

# Data Summaries

```
summary(tbl_flights[, c(7, 8, 13)])
##      arr_delay          carrier           time_hour
##  Min.   :-33.00    Length:100   Min.   :2013-01-01 05:00:00
##  1st Qu.:-10.00    Class :character 1st Qu.:2013-01-01 06:00:00
##  Median :- 2.50    Mode  :character Median :2013-01-01 06:00:00
##  Mean   : 1.03
##  3rd Qu.: 10.25
##  Max.   : 49.00
##  
```

# Data Summaries

```
str(tbl_flights)
## Classes 'tbl_df', 'tbl' and 'data.frame': 100 obs. of 13 variables:
## $ year           : int 2013 2013 2013 2013 2013 2013 2013 2013 2013 2013 ...
## $ month          : int 1 1 1 1 1 1 1 1 1 ...
## $ day            : int 1 1 1 1 1 1 1 1 1 ...
## $ sched_dep_time: int 515 529 540 545 600 558 600 600 600 600 ...
## $ dep_delay       : num 2 4 2 -1 -6 -4 -5 -3 -3 -2 ...
## $ sched_arr_time: int 819 830 850 1022 837 728 854 723 846 745 ...
## $ arr_delay       : num 11 20 33 -18 -25 12 19 -14 -8 8 ...
## $ carrier         : chr "UA" "UA" "AA" "B6" ...
## $ flight          : int 1545 1714 1141 725 461 1696 507 5708 79 301 ...
## $ tailnum         : chr "N14228" "N24211" "N619AA" "N804JB" ...
## $ origin          : chr "EWR" "LGA" "JFK" "JFK" ...
## $ dest            : chr "IAH" "IAH" "MIA" "BQN" ...
## $ time_hour       : POSIXct, format: "2013-01-01 05:00:00" "2013-01-01 05:00:00" ...
```



# Data Summaries

```
str(tbl_flights)
## Classes 'tbl_df', 'tbl' and 'data.frame'
## # $ year          : int 2013 2013 2013 ...
## # $ month         : int 1 1 1 1 1 1 1 ...
## # $ day           : int 1 1 1 1 1 1 1 ...
## # $ sched_dep_time: int 515 529 540 54...
## # $ dep_delay     : num 2 4 2 -1 -6 -4 ...
## # $ sched_arr_time: int 819 830 850 10...
## # $ arr_delay     : num 11 20 33 -18 -...
## # $ carrier       : chr "UA" "UA" "AA" ...
## # $ flight        : int 1545 1714 1141 ...
## # $ tailnum       : chr "N14228" "N242...
## # $ origin        : chr "EWR" "LGA" "J...
## # $ dest          : chr "IAH" "IAH" "M...
## # $ time_hour     : POSIXct, format: .
```



```
tibble::glimpse(tbl_flights)
## # Observations: 100
## # Variables: 13
## # $ year          <int> 2013, 2013, ...
## # $ month         <int> 1, 1, 1, 1, ...
## # $ day           <int> 1, 1, 1, 1, ...
## # $ sched_dep_time <int> 515, 529, 54...
## # $ dep_delay     <dbl> 2, 4, 2, -1, ...
## # $ sched_arr_time <int> 819, 830, 85...
## # $ arr_delay     <dbl> 11, 20, 33, ...
## # $ carrier       <chr> "UA", "UA", ...
## # $ flight        <int> 1545, 1714, ...
## # $ tailnum       <chr> "N14228", "N...
## # $ origin        <chr> "EWR", "LGA"...
## # $ dest          <chr> "IAH", "IAH"...
## # $ time_hour     <dttm> 2013-01-01 ...
```

# Data Summaries

```
dim(tbl_flights)
## [1] 100 13
length(tbl_flights)
## [1] 13

length(tbl_flights$origin)
## [1] 100
unique(tbl_flights$origin)
## [1] "EWR" "LGA" "JFK"

range(tbl_flights$dep_delay, na.rm = T)
## [1] -9 47
quantile(tbl_flights$dep_delay, na.rm = T)
##    0%   25%   50%   75% 100%
##   -9    -4    -2     0    47
var(tbl_flights$dep_delay, na.rm = T)
## [1] 63.41121
sd(tbl_flights$dep_delay, na.rm = T)
## [1] 7.963116
```

# Generate Data in R

# Generate Data Sets in R

You can combine vectors to a dataframe or, better, a tibble:

```
name <- c("Brian", "Jason", "Tyler", "Sam")
age <- seq(23, 26, by = 1) ## or: 23:26 or c(23, 24, 25, 26)
sex <- rep("male", length(name))

my_tbl <- tibble::tibble(name, age, sex)
```

# Generate Data Sets in R

From scratch, you would create these columns within the `tibble()` call:

```
my_tbl <- tibble::tibble(name, age, sex)

(my_tbl <- tibble::tibble(
  name = c("Brian", "Jason", "Tyler", "Sam"),
  age = seq(23, 26, by = 1), ## or: 23:26 or c(23, 24, 25, 26)
  sex = rep("male", length(name)))
)
## # A tibble: 4 x 3
##   name    age   sex
##   <chr> <dbl> <chr>
## 1 Brian     23 male
## 2 Jason     24 male
## 3 Tyler     25 male
## 4 Sam       26 male
```

# Generate Data Sets in R

You can also create tibbles using an easier to read row-by-row layout - a **tribble**:

```
my_tbl <- tibble::tibble(name, age, sex)

(my_tbl <- tibble::tribble(
  ~name,      ~age,    ~sex,
  "Brian",   23,      "male",
  "Jason",   24,      "male",
  "Tyler",   25,      "male",
  "Sam",     26,      "male"
))
## # A tibble: 4 x 3
##   name    age   sex
##   <chr> <dbl> <chr>
## 1 Brian     23 male
## 2 Jason     24 male
## 3 Tyler     25 male
## 4 Sam       26 male
```

# The tidyverse



*Illustration by Allison Horst ([github.com/allisonhorst/stats-illustrations](https://github.com/allisonhorst/stats-illustrations))*

# Why *tidyverse*?

Tidy data is a standard way of mapping the meaning of a dataset to its structure.

- Each variable forms a column.
- Each observation forms a row.
- Each type of observational unit forms a table.

Messy data is any other arrangement of the data.

# Tidy Data

- Each variable forms a column.
- Each observation forms a row.
- Each type of observational unit forms a table.

country	year	cases	population
Afghanistan	1999	745	19987071
Afghanistan	2000	2666	20595360
Brazil	1999	31737	17206362
Brazil	2000	80488	174504898
China	1999	212258	1272915272
China	2000	213766	1280425583

variables

country	year	cases	population
Afghanistan	1999	745	19987071
Afghanistan	2000	2666	20595360
Brazil	1999	31737	17206362
Brazil	2000	80488	174504898
China	1999	212258	1272915272
China	2000	213766	1280425583

observations

country	year	cases	population
Afghanistan	1999	745	19987071
Afghanistan	2000	2666	20595360
Brazil	1999	31737	17206362
Brazil	2000	80488	174504898
China	1999	212258	1272915272
China	2000	213766	1280425583

values

Source: Hadley Wickham's "R for Data Science" (R4DS)

# Messy Data

Real datasets often violate the three precepts of tidy data in almost every way imaginable:

- Column headers are values, not variable names.
- Multiple variables are stored in one column.
- Variables are stored in both rows and columns.
- Multiple types of observational units are stored in the same table.
- A single observational unit is stored in multiple tables.

# Why Should I Care About *Tidy* Data?

- makes it easy for an analyst or a computer to extract needed variables
- ensures that values of different variables from the same observation are always paired
- a good ordering makes it easier to scan the raw values

# Your Turn!

```
#> # A tibble: 12 x 4
#>   country      year type        count
#>   <chr>       <int> <chr>      <int>
#> 1 Afghanistan 1999 cases      745
#> 2 Afghanistan 1999 population 19987071
#> 3 Afghanistan 2000 cases     2666
#> 4 Afghanistan 2000 population 20595360
#> 5 Brazil       1999 cases     37737
#> 6 Brazil       1999 population 172006362
#> # ... with 6 more rows
```

# Your Turn!

```
#> # A tibble: 6 x 3
#>   country      year    rate
#>   * <chr>      <int> <chr>
#> 1 Afghanistan  1999  745/19987071
#> 2 Afghanistan  2000  2666/20595360
#> 3 Brazil       1999  37737/172006362
#> 4 Brazil       2000  80488/174504898
#> 5 China        1999  212258/1272915272
#> 6 China        2000  213766/1280428583
```

# Your Turn!

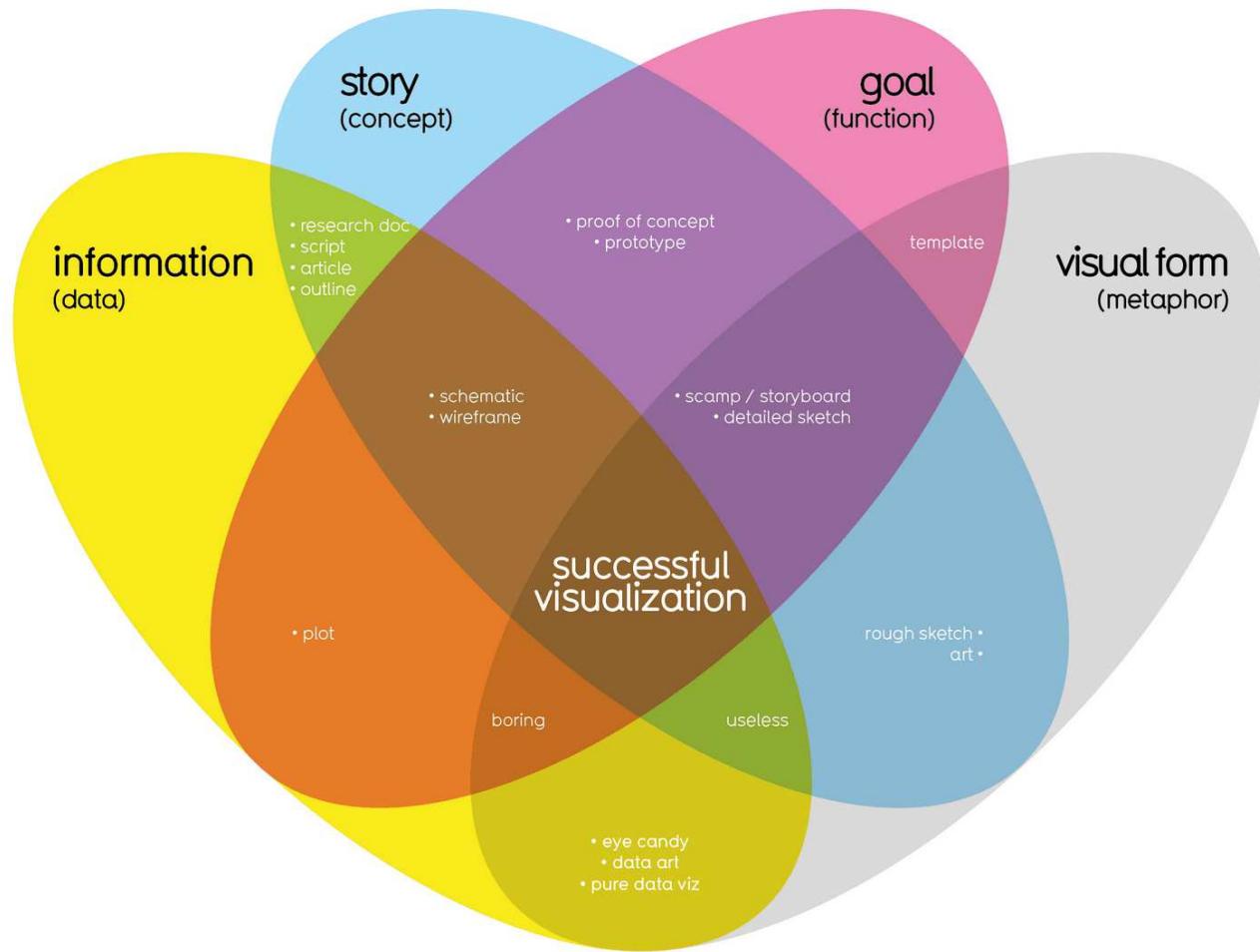
```
#> # A tibble: 6 x 4
#>   country      year  cases population
#>   <chr>        <int> <int>       <int>
#> 1 Afghanistan  1999     745  19987071
#> 2 Afghanistan  2000    2666  20595360
#> 3 Brazil        1999  37737  172006362
#> 4 Brazil        2000  80488  174504898
#> 5 China         1999 212258 1272915272
#> 6 China         2000 213766 1280428583
```

# Your Turn!

```
#> # A tibble: 3 x 3
#>   no_cases
#>   country     `1999` `2000`
#> * <chr>       <int>  <int>
#> 1 Afghanistan    745    2666
#> 2 Brazil        37737   80488
#> 3 China         212258  213766

#> # A tibble: 3 x 3
#>   population
#>   country     `1999`     `2000`
#> * <chr>       <int>      <int>
#> 1 Afghanistan 19987071  20595360
#> 2 Brazil       172006362 174504898
#> 3 China        1272915272 1280428583
```

# What Makes a Good Visualization?



Visualization by David McCandless (*Information is Beautiful*)