# Basic Data Manipulation and Cleaning

## 450X

**Stanford University**

Department of Political Science

Toby Nowacki

# Plan

1. Introduction: Working with data
2. What is tidy data?
3. Inspecting data
4. Data types, structures, and selecting data
5. Working with `data.frame`
    1. New variables
    2. Naming variables
    3. Apply family
6. Reshaping data (in base R)
7. Merging data

# Introduction

How do we use data sets?

- summary statistics
  - (conditional) averages, standard deviations, trends, scatterplots...
- statistical analysis
  - regressions, machine learning, ...
- individual datapoints
  - looking up individual 'cases'

**Key point #1**

**Data can be stored in many different ways.**
**It is almost always stored as a matrix.**

**What is the *best* way to organise data?**

# Why we need tidy data

**Challenge #1**

**Which of the following is best for:**

**(a) calculating the mean GDP?**
**(b) fitting a regression specification:** $GDP_{it} = \alpha + \beta_1 population_{it}$

Dataset 1:

| Country | Year | Variable | Value |
|---|---|---|---|
| United Kingdom | 2005 | GDP | 22,000 |
| United Kingdom | 2005 | Population | 40,000,000 |
| United Kingdom | 2006 | GDP | 23,000 |
| United Kingdom | 2006 | Population | 42,000,000 |
| Ireland, Rep. | 2005 | GDP | 18,000 |
| Ireland, Rep. | 2005 | Population | 3,000,000 |
| Ireland, Rep. | 2006 | GDP | 19,000 |
| Ireland, Rep. | 2006 | Population | 3,400,000 |

# Why we need tidy data

**Challenge #1**

**Which of the following is best for:**

**(a) calculating the mean GDP?**
**(b) fitting a regression specification:** $GDP_{it} = \alpha + \beta_1 population_{it}$

Dataset 2:

| Country | Variable | Value |
|---|---|---|
| United Kingdom | Year | 2005 |
| United Kingdom | GDP | 22,000 |
| United Kingdom | Population | 40,000,000 |
| United Kingdom | Year | 2006 |
| United Kingdom | GDP | 23,000 |
| United Kingdom | Population | 42,000,000 |
| Ireland, Rep. | Year | 2005 |
| Ireland, Rep. | GDP | 18,000 |

# Why we need tidy data

**Challenge #1**

**Which of the following is best for:**

**(a) calculating the mean GDP?**
**(b) fitting a regression specification:** $GDP_{it} = \alpha + \beta_1 population_{it}$

Dataset 3:

| Country | Year | GDP | Population |
|---|---|---|---|
| United Kingdom | 2005 | 22,000 | 40,000,000 |
| United Kingdom | 2006 | 23,000 | 42,000,000 |
| Ireland, Rep. | 2005 | 18,000 | 3,000,000 |
| Ireland, Rep. | 2006 | 19,000 | 3,400,000 |

# Why we need tidy data

**Key Point #2**

**A dataset is tidy iff:**

1. **Each variable has its own column;**
2. **Each observation forms its own row;**
3. **Each type of observational unit forms a table (matrix).**

Of the three previous examples, only dataset 3 is ***tidy***.

We can easily perform analyses with tidy data:

```
mean(df$GDP) #unconditional mean
mean(df$GDP[df$Year == 2005, ]) # mean in Year 2005
mod1 ← lm(GDP ~ Population, df) # simple regression
```

The objective of this class is to give you tools to inspect, transform, and manipulate your data.

# Inspecting data

# Inspecting data

- `?function` yields extensive documentation on how to use a specific function: arguments, default values, examples.

```
?head
```

```
# head                  package:utils                  R Documentat

# Return the First or Last Part of an Object

# Description:

#      Returns the first or last parts of a vector, matrix, table, da
#      frame or function.  Since ‘head()’ and ‘tail()’ are generic
#      functions, they may also have been extended to other classes.

# Usage:

#      head(x, ...)
#      ## Default S3 method:
```

# Inspecting data

- `head(x)` prints the first n (by default 5) rows of your table.

```
data(Seatbelts)
df ← as.data.frame(Seatbelts)
head(df, n = 6)

#   DriversKilled drivers front rear   kms PetrolPrice
# 1           107    1687   867  269  9059   0.1029718
# 2            97    1508   825  265  7685   0.1023630
# 3           102    1507   806  319  9963   0.1020625
# 4            87    1385   814  407 10955   0.1008733
# 5           119    1632   991  454 11823   0.1010197
# 6           106    1511   945  427 12391   0.1005812
#   VanKilled law
# 1        12   0
# 2         6   0
# 3        12   0
# 4         8   0
```

# Inspecting data

- `str(x)` gives you a compact overview over all subelements of x.

```
str(df)

# 'data.frame':   192 obs. of  8 variables:
#  $ DriversKilled: num  107 97 102 87 119 106 110 106 107 134 ...
#  $ drivers      : num  1687 1508 1507 1385 1632 ...
#  $ front        : num  867 825 806 814 991 ...
#  $ rear         : num  269 265 319 407 454 427 522 536 405 437 ...
#  $ kms          : num  9059 7685 9963 10955 11823 ...
#  $ PetrolPrice  : num  0.103 0.102 0.102 0.101 0.101 ...
#  $ VanKilled    : num  12 6 12 8 10 13 11 6 10 16 ...
#  $ law          : num  0 0 0 0 0 0 0 0 0 0 ...
```

# Inspecting data

- `class(x)` returns the class / type of the object in question.

```
class(df)
# [1] "data.frame"

class(df$drivers)
# [1] "numeric"
```

- `dim(x)` returns the dimensions of the dataframe (rows / columns).

```
dim(df)
# [1] 192   8
```

# Data types, structures and selecting data

# Data types (most common)

- `numeric`: `2.5`, `4.944`, ...
- `integer`: `1L`, `2L`, ...
- `character`: `"Banana"`, ...
- `logical`: `TRUE`, `FALSE`
- possible to convert between them using simple functions, e.g. `as.numeric(x)`.

# Data structures

- `vector` is a series of elements of the same data type (e.g., numeric, logical...)

```
ex_vec ← c(1L, 2L, 3L)
ex_vec
# [1] 1 2 3
class(ex_vec)
# [1] "integer"
ex_vec ← as.character(ex_vec)
ex_vec
# [1] "1" "2" "3"
class(ex_vec)
# [1] "character"
```

- select items within vector using `x[i]` or `x[i:j]`.

# Data structures

- `list` is a container for other data structures.
- items within lists can be different data types.

```
ex_list ← list("ABC", c(2, 4, 6), c("DEF", "GHI"))
ex_list
# [[1]]
# [1] "ABC"

# [[2]]
# [1] 2 4 6

# [[3]]
# [1] "DEF" "GHI"
```

- select items within list using `x[[i]]` or `x[i:j]`.
- you can also select objects *within* list items: `x[[i]][k]`

# Data structures

- `matrix` is a two-dimensional vector object.
- all elements within the matrix have to be of the same data type.

```
ex_mat ← matrix(1:9, nrow = 3, ncol = 3)
ex_mat

#      [,1] [,2] [,3]
# [1,]    1    4    7
# [2,]    2    5    8
# [3,]    3    6    9
```

- select items within matrix using `x[rows, columns]`.
- selection can be individual rows/columns, or vector (i.e., multiple ones).
- if all columns, or all rows, write `x[rows, ]` or `x[columns, ]`.

# Data structures

- `data.frame` is similar to a matrix, but different columns can have different data types.
- columns (= variables) have names!

```
ex_df ← data.frame(col1 = 1:3,
                   col2 = c("A", "B", "C"),
                   col3 = c(T, F, F))
ex_df
#   col1 col2  col3
# 1    1    A  TRUE
# 2    2    B FALSE
# 3    3    C FALSE
```

# Data structures

- `data.frame` is similar to a matrix, but different columns can have different data types.
- subsetting / selection works the same as with matrices
- in addition, also possible to select columns by name:

```
ex_df$col1
# [1] 1 2 3

ex_df[, c("col1", "col2")]
#   col1 col2
# 1    1    A
# 2    2    B
# 3    3    C
```

# Data structures

- this overview only covers the basics. For more depth and additional functions (e.g., naming columns, lists, etc.), see:

Data Types and Structures

- becomes important when creating own datasets or running more complex calculations
- most functions to convert one data type into another are intuitive. Plenty of online resources, too.
- packages and extensions to base R will introduce additional data types and structures (e.g., tibbles; panel data, document-term matrices)
- special data types (e.g., dates) covered separately!

# More tricks with `data.frame`

# Creating and modifying variables

- variables in dataframes are selected using the `$` operator, e.g., `df$var1`.
- the same works for creating new variables within a `data.frame`:

```
df$newvar ← c("a", "b", "c")
```

- recall all variable names in a dataframe using `names()`:

```
names(ex_df)
# [1] "col1" "col2" "col3"
```

- specific packages to make this easier (covered in later classes)

# The apply family

**Challenge #2**

**How do we transform each row of a dataset separately?**

- most straightforward way: for-loop

```
new_list ← list()
for(i in 1:nrow(df)){
  new_list[[i]] ← transf_fun(df[i, ])
}
new_df ← do.call(rbind, new_list)
```

- cumbersome coding
- requires new object in advance (list)

# The apply family

**Challenge #2**

**How do we transform each row of a dataset separately?**

- instead: `apply` family.
- takes each element of object passed to apply and transforms it separately
- `lapply, sapply, apply` differ in syntax and output but underlying logic is the same

```
out ← apply(df, 1, function(x) transf_fun(x))
```

- usually more shorthand than for-loop and sometimes more performant

# Reshaping and merging data

# Reshaping data

- difference between long and wide format

**Wide Format**

| | 1 | 2 | 3 |
|---|---|---|---|
| a | 0.1 | 0.2 | 0.3 |
| b | 0.2 | 0.4 | 0.6 |

**Long Format**

| | | |
|---|---|---|
| a | 1 | 0.1 |
| b | 1 | 0.2 |
| a | 2 | 0.2 |
| b | 2 | 0.4 |
| a | 3 | 0.3 |
| b | 4 | 0.6 |

# Reshaping data

- `reshape` is key for our purposes
- Example from `jozef.io`:

```
gdi_long_full ← reshape(data = gdi          # data.frame in wide form
                        , direction = "long" # still going from wide t
                        , varying = 2:23     # columns that will be st
                        , idvar = "country"  # what identifies the row
                        , v.names = "GDI"    # how will the column wit
                        , timevar = "year"   # how will the time colum
                        , times = 1995:2016  # what are the values for
                        )
```

- for further documentation, see help files
- again, we will cover packages that make this a lot easier!

# Merging data

**Challenge #3**

**How do we merge two datasets together by a joint variable?**

# Merging data

**Challenge #3**

**How do we merge two datasets together by a joint variable?**

- Two datasets, `dfA` and `dfB`, to be merged on variable `key`.
- `for` loop?

```
dfA$newvar1 ← NA
dfA$newvar2 ← NA

for(i in 1:nrow(dfA)){
  row_key ← dfA$key[i]
  matching_row ← which(dfB$key == i)
  dfA$newvar1[i] ← dfB$newvar1[matching_row]
  dfA$newvar2[i] ← dfB$newvar2[matching_row]
}
```

# Merging data

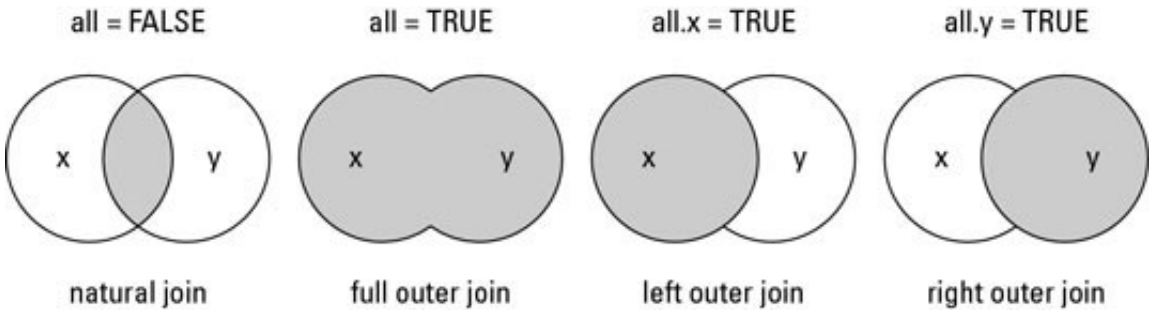
- `rbind` and `cbind` 'glue' two datasets together -- identical dimensions required!
- `merge` combines two datasets by a unique key
- You can also merge on multiple variables at the same time.

```
dfA ← data.frame(V1 = c(1, 2, 3),
                 V2 = c("A", "B", "C"))
dfB ← data.frame(V2 = c("B", "C", "A"),
                 V3 = c("Banana", "Canteloupe", "Apple"))

dfbind ← cbind(dfA, dfB)
dfM ← merge(dfA, dfB, by = c("V2"))
```

# Merging data

```
dfbind
#  V1 V2 V2         V3
#1  1  A  B     Banana
#2  2  B  C Canteloupe
#3  3  C  A      Apple

dfM
#   V2 V1         V3
# 1  A  1      Apple
# 2  B  2     Banana
# 3  C  3 Canteloupe
```

# Types of merges




```
merge(dfA, dfB, all.x = TRUE, all.y = FALSE)
```

- Cartesian join: if DF1 has $N$ rows and DF2 has $M$ rows, then the Cartesian merge produces all $N \times M$ combinations:

```
merge(dfA, dfB, by = NULL)
```

# Conclusion

- Introduction to loading and inspecting data
- Introduction to data structures and types
    - **structures** refer to the way data are organised (vector, table, ...)
    - **types** refer to the class of data that is stored (numeric, ...)
    - There's a parallel world inside R for advanced users (object-oriented programming, see here)
- Introduction to tidy data, and how to get there
- **Roadmap**:
    - How packages like `tidyverse` can make our lives a lot easier...
    - How to make visually appealing plots in `ggplot2`
    - Introduction to functional programming
    - Work beyond R: How to set up your work environment and store your

# Resources

### For `tidyverse` and other applied stuff

- R for Data Science by Garrett Grolemund and Hadley Wickham.
- Exploratory Data Analysis with R by Roger D. Peng.
- ggplot2: Elegant Graphics for Data Analysis by Hadley Wickham.

### For a deeper dive into programming

- Advanced R by Hadley Wickham.

35 / 35