Introduction to Applied Data Science

Lecture 5: Cleaning and Transforming Data

Bas Machielsen
Utrecht University
2024-04-22

Lecture 5: Transforming and Cleaning Data

Lecture 5: Transforming and Cleaning

- Overview of this class:
    - Introduction to Data Science
    - Introduction to R & Programming
    - Getting Data: API's and Databases
    - Getting Data: Web Scraping
    - **This lecture**: Transforming and Cleaning Data
    - Spatial & Network Data
    - Text as Data and Mining
    - Data Science Project

Transforming and Cleaning Data

- Most of the time when you're pursuing a data science research project, you will have to deal with *raw data*
- As a rule, data is untidy, but it also scattered around many places and it takes considerable effort to structure the data
- Remember that our end goal is *tidy data*, that is, data in which each observation corresponds to a *row* and each variable corresponds to a *column*
- We want this because this is the data format that is usually suitable for statistical analysis and visualization
- This lecture will acquaint you with (some of the) arsenal required to take pieces of raw data and tell R to assemble it into a tidy data format

Recap: Tidy Data

- There are three interrelated rules which make a dataset tidy:
    - Each variable must have its own column.
    - Each observation must have its own row.
    - Each value must have its own cell.

```r
palmerpenguins::penguins |> head(8)
```

```
## # A tibble: 8 × 8
##   species island    bill_length_mm bill_depth_mm flipper_length_mm body_mass_g sex
##   <fct>   <fct>              <dbl>         <dbl>             <int>       <int> <fct>
## 1 Adelie  Torgersen           39.1          18.7               181        3750 male
## 2 Adelie  Torgersen           39.5          17.4               186        3800 femal
## 3 Adelie  Torgersen           40.3          18                 195        3250 femal
## 4 Adelie  Torgersen           NA            NA                  NA          NA <NA>
## 5 Adelie  Torgersen           36.7          19.3               193        3450 femal
## 6 Adelie  Torgersen           39.3          20.6               190        3650 male
## 7 Adelie  Torgersen           38.9          17.8               181        3625 femal
## 8 Adelie  Torgersen           39.2          19.6               195        4675 male
```

Data Frames

- The most **fundamental, familiar and intuitive** format of data we have seen so far is the `data.frame`
- We generally store all of our data in `data.frame`s
- For example, usually, you import something like a `.csv` file as a `data.frame`
- A tidy `data.frame` is composed of **rows** (in tidy data, observations) and **columns** (variables)
- But it is rare that you get the data in exactly the **right form** you need
- One of the most common tasks you will face is to create new variables or summarize the data

Recap: Basic Data Transformation Functions

Data Transformation

- We have already worked with the `tidyverse` set of packages.
- The `tidyverse` library in R provides ways to transform your data using these basic 'verbs'.
  - Pick observations by their values (`filter()`).
  - Reorder the rows (`arrange()`).
  - Pick variables by their names (`select()`).
  - Create new variables with functions of existing variables (`mutate()`).
  - Collapse many values down to a single summary (`summarise()`).
- These can all be used in conjunction with `group_by()` which changes the scope of each function from operating on the entire dataset to operating on it group-by-group.

The Pipe Operator

- One of the things specific to R is a *pipe*
- I have used, and referred to it, on multiple occasions already
  - You might come across either `%>%` or `▷`, which are essentially the same
- They are used to express **a sequence of multiple operations**
- The point of the pipe is to help you write code in a way that is **easier to read and understand**
- You can take the pipe to mean "take the output from the previous line and use it as (one of the) inputs on the next line"

Example Pipe

- The pipe operator |> is a powerful tool in the R programming language that simplifies and enhances the readability of code, especially in data analysis workflows.

- It takes the output from one function and uses it as the first argument of the next function in the chain.

- It enables a more natural, left-to-right style of coding, similar to how we read and interpret information.
- Example (more will follow):

```r
# Example without pipe operator
result <- sqrt(mean(c(1, 4, 9, 16)))
```

```r
# Example with pipe operator
result <- c(1, 4, 9, 16) |>
            mean() |>
            sqrt()
```

Zooming In On Data

Rows (`filter` and `arrange`)

- The most important verbs that operate on **rows** of a dataset are `filter()`, which changes which rows are present without changing their order, and `arrange()`, which changes the order of the rows without changing which are present.

```r
library(palmerpenguins)
penguins |>
  filter(bill_length_mm < 40) |>
  head(3)
```

```
## # A tibble: 3 × 8
##   species island    bill_length_mm bill_depth_mm flipper_length_mm body_mass_g sex
##   <fct>   <fct>              <dbl>         <dbl>             <int>       <int> <fct>
## 1 Adelie  Torgersen           39.1          18.7               181        3750 male
## 2 Adelie  Torgersen           39.5          17.4               186        3800 femal
## 3 Adelie  Torgersen           36.7          19.3               193        3450 femal
```

Rows (`filter` and `arrange`)

- The most important verbs that operate on **rows** of a dataset are `filter()`, which changes which rows are present without changing their order, and `arrange()`, which changes the order of the rows without changing which are present.

```r
palmerpenguins::penguins |>
  arrange(desc(bill_length_mm))
```

```
## # A tibble: 344 × 8
##    species   island bill_length_mm bill_depth_mm flipper_length_mm body_mass_g sex
##    <fct>     <fct>           <dbl>         <dbl>             <int>       <int> <fct>
##  1 Gentoo    Biscoe           59.6          17                 230        6050 male
##  2 Chinstrap Dream            58            17.8               181        3700 femal
##  3 Gentoo    Biscoe           55.9          17                 228        5600 male
##  4 Chinstrap Dream            55.8          19.8               207        4000 male
##  5 Gentoo    Biscoe           55.1          16                 230        5850 male
##  6 Gentoo    Biscoe           54.3          15.7               231        5650 male
##  7 Chinstrap Dream            54.2          20.8               201        4300 male
##  8 Chinstrap Dream            53.5          19.9               205        4500 male
##  9 Gentoo    Biscoe           53.4          15.8               219        5500 male
## 10 Chinstrap Dream            52.8          20                 205        4550 male
## # ℹ 334 more rows
```

Common Mistakes

- When you're starting out with R, the easiest mistake to make is to use `=` instead of `==` when testing for equality. `filter()` will let you know when this happens:

```
penguins |>
  filter(year = 2007)
```

```
## Error in `filter()`:
## ! We detected a named input.
## ℹ This usually means that you've used `=` instead of `==`.
## ℹ Did you mean `year == 2007`?
```

- Another mistake is you writing "or" statements like you would in English:

```
penguins |>
  filter(species == "Adelie" | "Chinstrap")
```

```
## Error in `filter()`:
## ℹ In argument: `species == "Adelie" | "Chinstrap"`.
## Caused by error in `species == "Adelie" | "Chinstrap"`:
## ! operations are possible only for numeric, logical or complex types
```

Columns (`mutate` and `select`)

- The job of `mutate()` is to add **new columns** that are calculated from the existing columns.
- And `select()` allows you to rapidly **zoom in** on a useful subset using operations based on the names of the variables

```
penguins |>
  mutate(body_mass_kg = body_mass_g/1000) |>
  select(species, island, body_mass_kg)
```

```
## # A tibble: 344 × 3
##    species island    body_mass_kg
##    <fct>   <fct>            <dbl>
##  1 Adelie  Torgersen         3.75
##  2 Adelie  Torgersen         3.8
##  3 Adelie  Torgersen         3.25
##  4 Adelie  Torgersen        NA
##  5 Adelie  Torgersen         3.45
##  6 Adelie  Torgersen         3.65
##  7 Adelie  Torgersen         3.62
##  8 Adelie  Torgersen         4.68
##  9 Adelie  Torgersen         3.48
## 10 Adelie  Torgersen         4.25
```

Grouping & Summarizing

- You can use `group_by()` to divide your dataset into groups meaningful for your analysis:

```
penguins |>
  group_by(species) |>
  head(5)
```

```
## # A tibble: 5 × 8
## # Groups:   species [1]
##   species island    bill_length_mm bill_depth_mm flipper_length_mm body_mass_g sex
##   <fct>   <fct>              <dbl>         <dbl>             <int>       <int> <fct>
## 1 Adelie  Torgersen           39.1          18.7               181        3750 male
## 2 Adelie  Torgersen           39.5          17.4               186        3800 femal
## 3 Adelie  Torgersen           40.3          18                 195        3250 femal
## 4 Adelie  Torgersen           NA            NA                  NA          NA <NA>
## 5 Adelie  Torgersen           36.7          19.3               193        3450 femal
```

- By itself, `group_by()` does nothing. However, `group_by()` is often used together with `summarize` or `mutate`, but this means subsequent operations will now work "by species".

Grouping & Summarizing

- The most important grouped operation is a **summary**, which, if being used to calculate a single summary statistic, reduces the data frame to have a single row for each group.

```
penguins ▷
  group_by(species) ▷
  summarize(name = mean(body_mass_g, na.rm=T))
```

```
## # A tibble: 3 × 2
##   species    name
##   <fct>     <dbl>
## 1 Adelie    3701.
## 2 Chinstrap 3733.
## 3 Gentoo    5076.
```

Grouping & Mutate

- The function `group_by()` can also be used together with mutate to create a group characteristic:

```
penguins ▷
  group_by(species) ▷
  mutate(bill_length_depth = bill_length_mm * bill_depth_mm) ▷
  select(species, island, body_mass_g, bill_length_depth)
```

```
## # A tibble: 344 × 4
## # Groups:   species [3]
##    species island    body_mass_g bill_length_depth
##    <fct>   <fct>           <int>             <dbl>
##  1 Adelie  Torgersen        3750              731.
##  2 Adelie  Torgersen        3800              687.
##  3 Adelie  Torgersen        3250              725.
##  4 Adelie  Torgersen          NA               NA
##  5 Adelie  Torgersen        3450              708.
##  6 Adelie  Torgersen        3650              810.
##  7 Adelie  Torgersen        3625              692.
##  8 Adelie  Torgersen        4675              768.
##  9 Adelie  Torgersen        3475              617.
## 10 Adelie  Torgersen        4250              848.
## # ℹ 334 more rows
```

Merging Data

Merging Datasets

- It’s rare that a data analysis involves only a single data frame.
- Typically you have many data frames, and you must **join** them together to answer the questions that you’re interested in.
- Joins add new variables to one data frame from matching observations in another.
- With the exception of one join, called `anti_join()`, which filters observations from one data frame based on whether or not they match an observation in another. We will see that in lecture 6.
- This week, we'll be focusing almost exclusively on joins of the first kind.

Keys

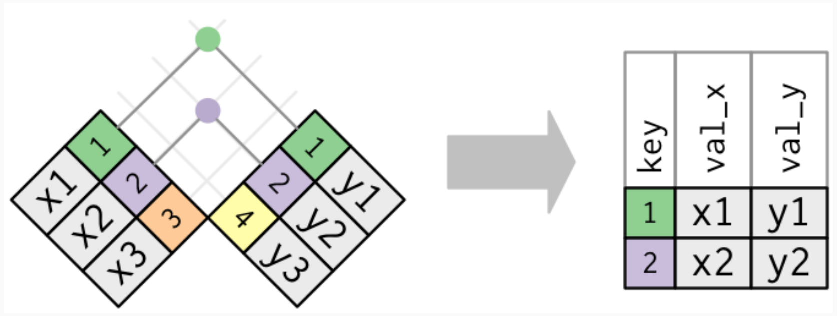
- To understand **joins**, you need to first understand how two tables can be connected through a pair of **keys**, within each table.
- The variables used to connect each pair of tables are called **keys**. Two datasets you might want to merge look as follows:

```
x ← tribble(
  ~key, ~val_x,
     1, "x1",
     2, "x2",
     3, "x3"
)
y ← tribble(
  ~key, ~val_y,
     1, "y1",
     2, "y2",
     4, "y3"
)
```

- Note that some values of `key` aren't present in both datasets

Inner and Full Join

- The simplest type of join is the **inner join**. An inner join matches pairs of observations whenever their keys are equal:

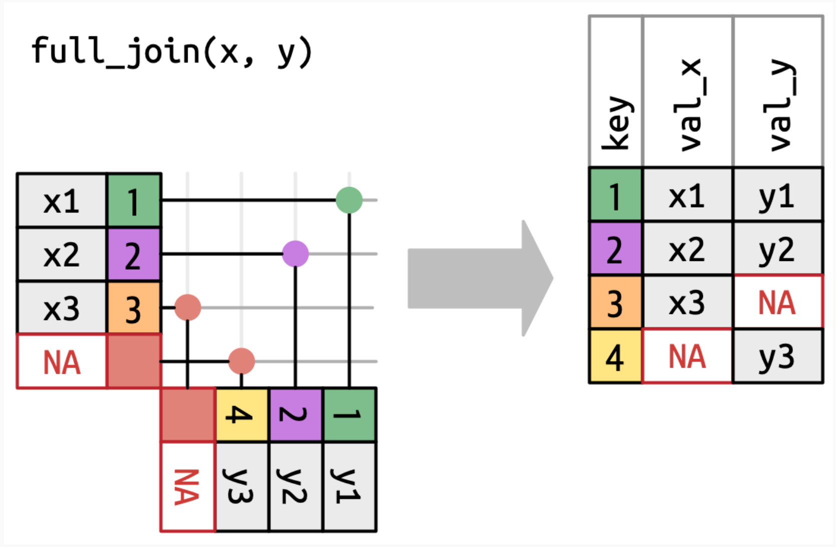


```
inner_join(x, y, by = "key")
```

```
## # A tibble: 2 × 3
##     key val_x val_y
##   <dbl> <chr> <chr>
## 1     1 x1    y1
## 2     2 x2    y2
```

Inner and Full Join

- Does the `inner_join()` function represent a filtering join or a mutating join?
- In addition, there is `full_join()`:



Inner and Full Join

- As you can see, a full join keeps all observations in `x` and `y`.

```r
full_join(x, y, by = "key")
```

```
## # A tibble: 4 × 3
##     key val_x val_y
##   <dbl> <chr> <chr>
## 1     1 x1    y1
## 2     2 x2    y2
## 3     3 x3    <NA>
## 4     4 <NA>  y3
```

Left and Right Joins

- However, the most common situation is that you have one particular `data.frame` to which you want to **merge information** from another `data.frame`
- To this end, the functions `left_join()` and `right_join()` can be used. They are **symmetrical functions**, as you'll see shortly
- `left_join()` proceeds on the basis of the "left" `data.frame`, the `data.frame` that is specified as the *first* argument to the `left_join()` function
- It then merges the observations from the right `data.frame` (the second argument of the function) to the left `data.frame` in so far as these have a match in the left `data.frame`:

```
left_join(x,y)
```

```
## # A tibble: 3 × 3
##     key val_x val_y
##   <dbl> <chr> <chr>
## 1     1 x1    y1
## 2     2 x2    y2
## 3     3 x3    <NA>
```

Right Join

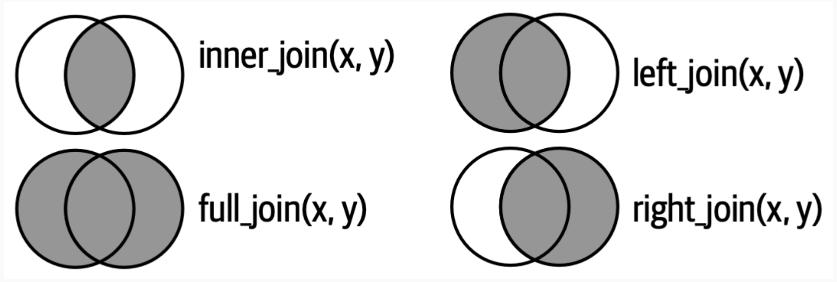
- The function `right_join()` works in exactly the same way:

```
right_join(x, y)
```

```
## # A tibble: 3 × 3
##     key val_x val_y
##   <dbl> <chr> <chr>
## 1     1 x1    y1
## 2     2 x2    y2
## 3     4 <NA>  y3
```

Types of Joins

- The following figure depicts the kinds of `joins` you can do precisely:



Specifying Keys

- It also occurs often that you want to join `data.frame`s on the basis of more than one key
    - For example, if you have two datasets of firms (key 1) accounting information in year t (key 2)
    - Then, the key consists of a firm name and a year in the first dataset, and a firm name and a year in the second dataset
- Here is a second example. Suppose you want to merge a dataset of a country's electricity output in a particular year to a dataset of a country's GDP per capita in that same year:

Specifying Keys

```r
library(wbstats)
electricity <- wb_data("4.1.1_TOTAL.ELECTRICITY.OUTPUT") |>
  select(country, date, contains('4.1.1'))

electricity |> head(5)
```

```
## # A tibble: 5 × 3
##   country  date `4.1.1_TOTAL.ELECTRICITY.OUTPUT`
##   <chr>   <dbl>                            <dbl>
## 1 Aruba    1990                              338
## 2 Aruba    1991                              339
## 3 Aruba    1992                              341
## 4 Aruba    1993                              531
## 5 Aruba    1994                              564
```

Specifying Keys

- As you can see, an observation in this `data.frame` is uniquely identified by a combination of `country` and `date`:

```r
gdp ← wb_data("NY.GDP.PCAP.KD") ▷
  select(country, date, contains("NY."))
gdp
```

```
## # A tibble: 13,888 × 3
##    country  date NY.GDP.PCAP.KD
##    <chr>   <dbl>          <dbl>
##  1 Aruba    1960             NA
##  2 Aruba    1961             NA
##  3 Aruba    1962             NA
##  4 Aruba    1963             NA
##  5 Aruba    1964             NA
##  6 Aruba    1965             NA
##  7 Aruba    1966             NA
##  8 Aruba    1967             NA
##  9 Aruba    1968             NA
## 10 Aruba    1969             NA
## # ℹ 13,878 more rows
```

Merging With More Keys

- All of the `_join()` functions can also **accommodate merging** on the basis of more than one key
- If the keys in the datasets have the same name, you do not have to specify any other arguments in the function

```
left_join(gdp, electricity) ▷
  drop_na()
```

```
## # A tibble: 4,806 × 4
##    country  date NY.GDP.PCAP.KD `4.1.1_TOTAL.ELECTRICITY.OUTPUT`
##    <chr>   <dbl>          <dbl>                            <dbl>
##  1 Aruba    1990         25411.                             338
##  2 Aruba    1991         26565.                             339
##  3 Aruba    1992         27194.                             341
##  4 Aruba    1993         28307.                             531
##  5 Aruba    1994         29666.                             564
##  6 Aruba    1995         29498.                             616
##  7 Aruba    1996         28958.                             642
##  8 Aruba    1997         30074.                             675
##  9 Aruba    1998         29766.                             730
## 10 Aruba    1999         29263.                             738.
## # ℹ 4,796 more rows
```

Specifying Keys

- However, sometimes, variables have **different names** in **different datasets**.
- In this situation, you can use the `by` argument inside the `_join()` functions. This also works when variables do have the same name
- The syntax is `by=c('key1_in_df1' = 'key1_in_df2', 'key2_in_df1' = 'key2_in_df2')`:

```r
right_join(gdp, electricity,
           by = c('country' = 'country', 'date' = 'date'))
```

```
## # A tibble: 5,886 × 4
##    country  date NY.GDP.PCAP.KD `4.1.1_TOTAL.ELECTRICITY.OUTPUT`
##    <chr>   <dbl>          <dbl>                            <dbl>
##  1 Aruba    1990         25411.                              338
##  2 Aruba    1991         26565.                              339
##  3 Aruba    1992         27194.                              341
##  4 Aruba    1993         28307.                              531
##  5 Aruba    1994         29666.                              564
##  6 Aruba    1995         29498.                              616
##  7 Aruba    1996         28958.                              642
##  8 Aruba    1997         30074.                              675
##  9 Aruba    1998         29766.                              730
## 10 Aruba    1999         29263.                              738.
## # ℹ 5,876 more rows
```

Joining With Inexact Keys

- It also frequently happens that you have two `data.frame`s with keys that *should* be the same, but aren't.
- For example, you might have one `data.frame` with an observation "Netherlands", and another with an observation "The Netherlands"
- Since these observations do not match exactly, the `_join()` family of functions cannot handle this well
- In this case, we might need the `fuzzyjoin` library, and join two `data.frame`s on the basis of *string distance*: a **string distance** is some kind of measure encapsulating how far away to strings are by looking at **commonalities** between two strings.

Joining With Inexact Keys

- The relevant family of functions from the `fuzzyjoin` package is `stringdist_*_join`, for example, `stringdist_left_join`. It is best to illustrate this with an example. Let me scrape two tables from Wikipedia:

```
library(fuzzyjoin); library(rvest); library(janitor)
table_gdp_ppp ← read_html('https://en.wikipedia.org/wiki/List_of_countries_by_GDP
  html_element('table.wikitable') ▷
  html_table() ▷
  row_to_names(1) ▷
  clean_names()

table_gdp_ppp ▷ head(5)
```

```
## # A tibble: 5 × 8
##   country_or_territory un_region forecast    year       estimate    year_2     estimat
##   <chr>                <chr>     <chr>       <chr>      <chr>       <chr>      <chr>
## 1 World                —         185,677,122 2024       164,155,327 2022       127,800
## 2 China                Asia      35,291,015  [n 1]2024 30,327,320  [n 2]2022 23,009,
## 3 United States        Americas  28,781,083  2024       25,462,700  2022       19,846,
## 4 India                Asia      14,594,460  2024       11,874,583  2022       8,443,3
## 5 Japan                Asia      6,720,962   2024       5,702,287   2022       5,224,8
```

Joining With Inexact Keys

```r
table_electricity <- read_html("https://en.wikipedia.org/wiki/List_of_countries_by
  html_table() |>
  pluck(1) |>
  row_to_names(row_number = 1) |>
  clean_names()

table_electricity |> head(5)
```

```
## # A tibble: 5 × 11
##   location      total_t_wh coal   gas   hydro nuclear wind  solar oil   bio   geo
##   <chr>         <chr>      <chr>  <chr> <chr> <chr>   <chr> <chr> <chr> <chr> <chr>
## 1 World         28,003     10,095 6,399 4,246 2,750   1,848 1,047 868   657   93
## 2 China         8,534      5,329  287   1,300 408     656   327   62    166   0
## 3 United States 4,154      898    1,579 246   780     378   164   35    54    18
## 4 India         1,715      1,274  60    160   44      68    68    3     37    0
## 5 Russia        1,110      169    485   215   222     4     2     12    0.8   0.4
```

Joining With Inexact Keys

- As you can see, both `table_gdp_ppp` and `table_electricity` contain country names
    - In `country_or_territory` and `location` respectively
- However, country names might be spelled differently, e.g. Netherlands vs. The Netherlands
    - They might still contain commonalities: "The Netherlands" and "Netherlands" have everthing in common except "The"
    - It turns out you can calculate *string distances* that quantify the differences between strings
    - Then, we can use these distances to match observations given strings in two datasets

Joining With Inexact Keys

- Let's `left_join` `table_gdp_ppp` to `table_electricity` on the basis of fuzzy string matching:

```r
matched_df ← stringdist_left_join(table_gdp_ppp, table_electricity,
                     by = c('country_or_territory'='location'),
                     max_dist = 0.5)

matched_df ▷ head(5)
```

```
## # A tibble: 5 × 19
##   country_or_territory un_region forecast year  estimate year_2 estimate_2 year_3 lo
##   <chr>                <chr>     <chr>    <chr> <chr>    <chr>  <chr>      <chr>  <c
## 1 World                —         185,677… 2024  164,155… 2022   127,800,0… 2017   Wo
## 2 China                Asia      35,291,… [n 1… 30,327,… [n 2]… 23,009,780 [n 1]… Ch
## 3 United States        Americas  28,781,… 2024  25,462,… 2022   19,846,720 2020   Un
## 4 India                Asia      14,594,… 2024  11,874,… 2022   8,443,360  2020   In
## 5 Japan                Asia      6,720,9… 2024  5,702,2… 2022   5,224,850  2019   Ja
## # ℹ 5 more variables: wind <chr>, solar <chr>, oil <chr>, bio <chr>, geo <chr>
```

Pivoting Data Sets

Pivoting

- A dataset can also be in the wrong shape, or contain all the information you need, but not according to the principles of tidy data
  - For example, take a look at this dataset about tuberculosis cases:

```r
table4a
```

```
## # A tibble: 3 × 3
##   country     `1999` `2000`
##   <chr>        <dbl>  <dbl>
## 1 Afghanistan    745   2666
## 2 Brazil       37737  80488
## 3 China       212258 213766
```

Pivoting

- This data is not tidy, as two *variables* (year, and cases) are jointly stored in several columns instead of as one variable, one column
  - We want a dataset that contains the columns `country`, `year`, and `tb_cases`
- The first step is always to figure out what the **variables and observations** should be
- The second step is to resolve one of two common problems:
  - One variable might be spread across multiple columns
  - One observation might be scattered across multiple rows.
- **Pivoting** is the way in which you **reshape** datasets from such a format to a tidy format and the other way around
  - There exists two kind of pivots: from **wide** to **long** and from **long** to **wide**
  - Which do we have to use now?

Pivot Longer

- `pivot_longer` takes three arguments:
  - the columns whose names are values, not variables. In this example, those are the columns with the years in it
  - the name of the variable to which we want to move the columns' **names**
  - the name of the variable to which we want to move the columns' **value**

```
table4a ▷
  pivot_longer(cols = where(is.double),
                           names_to = 'year',
                           values_to = 'tc_cases')
```

```
## # A tibble: 6 × 3
##   country     year  tc_cases
##   <chr>       <chr>    <dbl>
## 1 Afghanistan 1999       745
## 2 Afghanistan 2000      2666
## 3 Brazil      1999     37737
## 4 Brazil      2000     80488
## 5 China       1999    212258
## 6 China       2000    213766
```

Pivot Wider

- `pivot_wider()` is the opposite of `pivot_longer().` You use it when an observation is scattered across multiple rows:

```
table2 ▷ head(6)
```

```
## # A tibble: 6 × 4
##   country      year type            count
##   <chr>       <dbl> <chr>           <dbl>
## 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
```

Parameters of Pivot Wider

- This time, however, we only need two parameters:
  - The column to take variable names from. Here, it's `type`
  - The column to take values from. Here it's `count`

```
table2 ▷
  pivot_wider(names_from = type,
              values_from = count)
```

```
## # A tibble: 6 × 4
##   country      year  cases population
##   <chr>       <dbl>  <dbl>      <dbl>
## 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
```

Recapitulation

Recapitulation

- In this lecture, we went over some **important features/operations** used to zoom in on, and clean data
- We also learned to use some of these functions in the context of **grouped data**
- We extensively focused on **merging data**:
  - We learned the standard merging logic of `_join()`
  - We learned several alternatives in case you do not have key variables
- We learned how to **reshape data** into a tidy format