CIS 8398
Advanced AI Topics in Business

#Data Wrangling

Yu-Kai Lin

Data wrangling

Data wrangling is the *art* of getting your data into R in a useful form for visualisation and modelling.

Data wrangling is very important: without it you can't work with your own data!

Two key tools:

- **dplyr**: dplyr provides a grammar of data manipulation
- **tidyr**: tidyr provides a set of functions that help you get to tidy data

---

[**Acknowledgements**] The materials in the following slides are based on the source(s) below:

- R for Data Science by Garrett Grolemund and Hadley Wickham

Prerequisites

```r
#install.packages("tidyverse") #install the package if you haven't already
library(tidyverse) # includes dplyr and tidyr
```

```
## ── Attaching core tidyverse packages ──────────────────────── tidyverse 2.0.0 ─
## ✓ dplyr     1.1.2     ✓ readr     2.1.4
## ✓ forcats   1.0.0     ✓ stringr   1.5.0
## ✓ ggplot2   3.4.3     ✓ tibble    3.2.1
## ✓ lubridate 1.9.2     ✓ tidyr     1.3.0
## ✓ purrr     1.0.2
## ── Conflicts ────────────────────────────────────────────── tidyverse_conflicts()
## ✕ dplyr::filter()     masks stats::filter()
## ✕ dplyr::group_rows() masks kableExtra::group_rows()
## ✕ dplyr::lag()        masks stats::lag()
## ℹ Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflic
```

Package dplyr

dplyr provides the following *verbs* for data manipulation.

1. select
2. filter
3. arrange
4. mutate
5. summarise & group_by
6. joining (merging) data frames / tables

We will be using HousePrices.csv to learn `dplyr`. If you haven't already, please download and put the file in your working directory. Use `getwd()` and `setwd(...)` to make sure you have a correct working directory. And read `HousePrices.csv` into R.

```
df = read_csv("data/HousePrices.csv")
```

`select()`: Pick columns by name

```
select(df, c(price, lotsize)) # equivalent to df[, c("price", "lotsize")]
```

```
## # A tibble: 546 × 2
##    price lotsize
##    <dbl>   <dbl>
##  1 42000    5850
##  2 38500    4000
##  3 49500    3060
##  4 60500    6650
##  5 61000    6360
##  6 66000    4160
##  7 66000    3880
##  8 69000    4160
##  9 83800    4800
## 10 88500    5500
## # ℹ 536 more rows
```

Chaining/Pipelining

Pipe operators make the code much more readable. Essentially, you send the data through a set of operations and the operations are connected by a pipe.

Note: You must have a space before and after the pipe

Difference between `|>` and `%>%`

The `|>` pipe is from base R. If you are using R 4.1+, it has this native pipe operator.

The `%>%` pipe is from the `magrittr` package, which is then incorporated in `tidyverse`. You can use this pipe in any version of R as long as you.

```
select(df, c(price, lotsize))

df |> select(c(price, lotsize))  # notice that df is outside select()

df %>% select(c(price, lotsize)) # the code is much more readable
```

The pipe operators will use the output from before the pipe as **the first argument** for the function after the pipe.

If the output from before the pipe is NOT supposed to be **the first argument** for the function after the pipe, you can specify that via a placeholder.

The `|>` pipe uses `_` to represent a placeholder (see the example below).

The `%>%` pipe uses `.` to represent a placeholder (see the example below).

```
df |> lm(price ~ lotsize, data = _)

df %>% lm(price ~ lotsize, data = .)
```

For the purpose of this course, I will be using the base R pipe `|>` in our lectures.

https://r4ds.hadley.nz/data-transform.html#sec-the-pipe

> For simple cases, `|>` and `%>%` behave identically. So why do we recommend the base pipe?
>
> Firstly, because it's part of base R, it's always available for you to use, even when you're not using the tidyverse.
>
> Secondly, |> is quite a bit simpler than %>%: in the time between the invention of %>% in 2014 and the inclusion of |> in R 4.1.0 in 2021, we gained a better understanding of the pipe. This allowed the base implementation to jettison infrequently used and less important features.

```r
df |> select(price:driveway) # all columns between price and driveway
```

```
## # A tibble: 546 × 6
##    price lotsize bedrooms bathrooms stories driveway
##    <dbl>   <dbl>    <dbl>     <dbl>   <dbl> <chr>
##  1 42000    5850        3         1       2 yes
##  2 38500    4000        2         1       1 yes
##  3 49500    3060        3         1       1 yes
##  4 60500    6650        3         1       2 yes
##  5 61000    6360        2         1       1 yes
##  6 66000    4160        3         1       1 yes
##  7 66000    3880        3         2       2 yes
##  8 69000    4160        3         1       3 yes
##  9 83800    4800        3         1       1 yes
## 10 88500    5500        3         2       4 yes
## # ℹ 536 more rows
```

```r
# select columns that contain "room" in their column names
df |> select(contains("room"))
```

```
## # A tibble: 546 × 2
##    bedrooms bathrooms
##       <dbl>     <dbl>
##  1        3         1
##  2        2         1
##  3        3         1
##  4        3         1
##  5        2         1
##  6        3         1
##  7        3         2
##  8        3         1
##  9        3         1
## 10        3         2
## # ℹ 536 more rows
```

Hide certain columns

```
df |> select(-c(price, lotsize, gasheat))
```

```
## # A tibble: 546 × 9
##    bedrooms bathrooms stories driveway recreation fullbase aircon garage prefer
##       <dbl>     <dbl>   <dbl> <chr>    <chr>      <chr>    <chr>   <dbl> <chr>
##  1        3         1       2 yes      no         yes      no          1 no
##  2        2         1       1 yes      no         no       no          0 no
##  3        3         1       1 yes      no         no       no          0 no
##  4        3         1       2 yes      yes        no       no          0 no
##  5        2         1       1 yes      no         no       no          0 no
##  6        3         1       1 yes      yes        yes      yes         0 no
##  7        3         2       2 yes      no         yes      no          2 no
##  8        3         1       3 yes      no         no       no          0 no
##  9        3         1       1 yes      yes        yes      no          0 no
## 10        3         2       4 yes      yes        no       yes         1 no
## # ℹ 536 more rows
```

`filter()`: Keep rows that match criteria

```
df |> filter(price < 30000, driveway == "yes")
```

```
## # A tibble: 5 × 12
##   price lotsize bedrooms bathrooms stories driveway recreation fullbase gasheat
##   <dbl>   <dbl>    <dbl>     <dbl>   <dbl> <chr>    <chr>      <chr>    <chr>
## 1 27000    1700        3         1       2 yes      no         no       no
## 2 25000    3620        2         1       1 yes      no         no       no
## 3 25000    3850        3         1       2 yes      no         no       no
## 4 26000    3000        2         1       1 yes      no         yes      no
## 5 27000    3649        2         1       1 yes      no         no       no
## # ℹ 3 more variables: aircon <chr>, garage <dbl>, prefer <chr>
```

Chaining multiple operations

```
df |>
  filter(price < 30000, driveway == "yes") |>
  select(price, driveway, aircon)
```

```
## # A tibble: 5 × 3
##   price driveway aircon
##   <dbl> <chr>    <chr>
## 1 27000 yes      no
## 2 25000 yes      no
## 3 25000 yes      no
## 4 26000 yes      no
## 5 27000 yes      no
```

After the operations, you may want to save the result as a new data frame. You can then use the new data frame for other analyses.

```
new_df <- df |>
  filter(price < 30000, driveway == "yes") |>
  select(price, driveway, aircon)
nrow(new_df)
```

```
## [1] 5
```

`arrange()`: Reorder rows

Use `desc()` for a descending order

```
df |>
  select(price, aircon, stories) |>
  arrange(price)
```

```
## # A tibble: 546 × 3
##    price aircon stories
##    <dbl> <chr>    <dbl>
##  1 25000 no           1
##  2 25000 no           1
##  3 25000 no           2
##  4 25245 no           1
##  5 26000 no           1
##  6 26500 no           1
##  7 27000 no           2
##  8 27000 no           1
##  9 28000 no           2
## 10 30000 no           2
## # ℹ 536 more rows
```

```
df |>
  select(price, aircon, stories) |>
  arrange(desc(price))
```

```
## # A tibble: 546 × 3
##     price aircon stories
##     <dbl> <chr>    <dbl>
##  1 190000 yes          3
##  2 175000 yes          4
##  3 175000 no           2
##  4 174500 yes          2
##  5 163000 yes          2
##  6 155000 yes          1
##  7 145000 yes          4
##  8 145000 no           2
##  9 141000 yes          2
## 10 140000 yes          4
## # ℹ 536 more rows
```

`mutate()`: Add new variables

Create new variables that are functions of existing variables

```
df |>
  mutate(rooms = bedrooms+bathrooms) |>
  filter(rooms > 7) |>
  select(bedrooms, bathrooms, rooms)
```

```
## # A tibble: 4 × 3
##   bedrooms bathrooms rooms
##      <dbl>     <dbl> <dbl>
## 1        5         3     8
## 2        4         4     8
## 3        6         2     8
## 4        5         3     8
```

`summarise()`: Reduce variables to values

- `group_by()` creates the groups that will be operated on
- `summarise()` uses the provided aggregation function to summarise each group

```
df |>
  group_by(aircon) |>
  summarise(avg_price = mean(price))
```

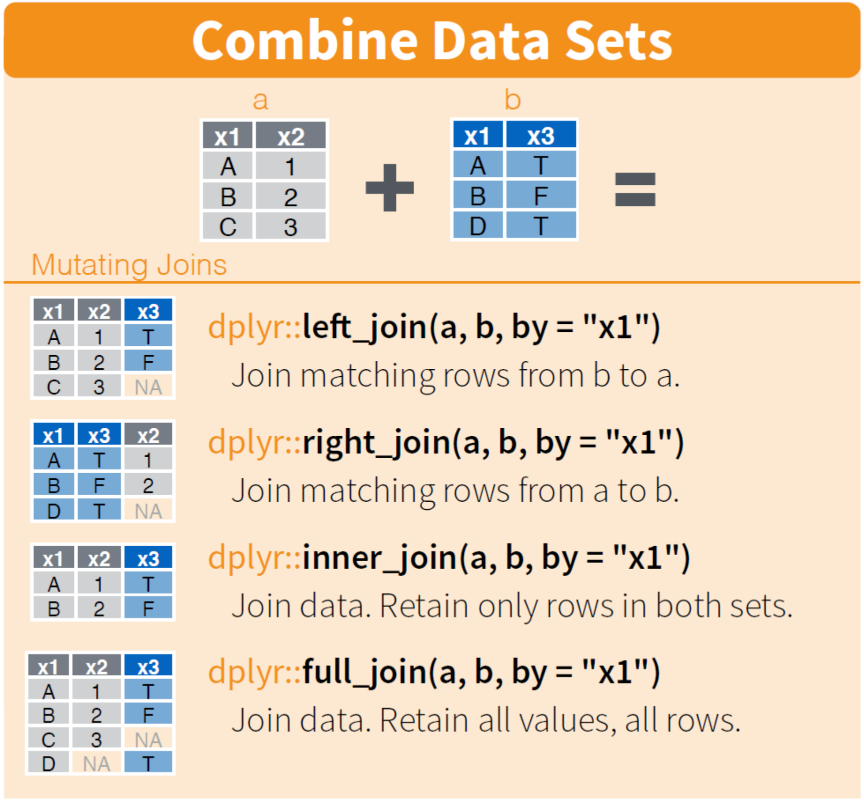
```
## # A tibble: 2 × 2
##   aircon avg_price
##   <chr>      <dbl>
## 1 no        59885.
## 2 yes       85881.
```

You can have multiple summary/aggregate statistics:

```
df |>
  group_by(aircon) |>
  summarise(n_house = n(), # n() gives the number of rows in each group
            avg_price = mean(price))
```

```
## # A tibble: 2 × 3
##   aircon n_house avg_price
##   <chr>    <int>     <dbl>
## 1 no         373    59885.
## 2 yes        173    85881.
```

Joining two data frames (or tables)



Note: `left_join(a, b, by="x1")` is equivalent to `a |> left_join(b, by="x1")`

Demo the join operatioins

Let's try these join operations on two small data frames.

```r
(df1 <- tibble(id = c(1, 2), name = c("Alice", "Bob")))
```

```
## # A tibble: 2 × 2
##      id name 
##   <dbl> <chr>
## 1     1 Alice
## 2     2 Bob  
```

```r
(df2 <- tibble(id = c(1, 3), state = c("FL", "NY")))
```

```
## # A tibble: 2 × 2
##      id state
##   <dbl> <chr>
## 1     1 FL   
## 2     3 NY   
```

`left_join(x, y)` includes all observations in x, regardless of whether they match or not. This is the most commonly used join because it ensures that you do not lose observations from your primary table.

```
df1
```

```
## # A tibble: 2 × 2
##      id name
##   <dbl> <chr>
## 1     1 Alice
## 2     2 Bob
```

```
df2
```

```
## # A tibble: 2 × 2
##      id state
##   <dbl> <chr>
## 1     1 FL
## 2     3 NY
```

```
# same as left_join(df1, df2)
df1 |> left_join(df2)
```

```
## # A tibble: 2 × 3
##      id name  state
##   <dbl> <chr> <chr>
## 1     1 Alice FL
## 2     2 Bob   <NA>
```

`NA` will be used when a value is missing.

`inner_join(x, y)` only includes observations that match in both x and y.

```r
df1
```

```
## # A tibble: 2 × 2
##      id name 
##   <dbl> <chr>
## 1     1 Alice
## 2     2 Bob  
```

```r
df2
```

```
## # A tibble: 2 × 2
##      id state
##   <dbl> <chr>
## 1     1 FL   
## 2     3 NY   
```

```r
# same as inner_join(df1, df2)
df1 |> inner_join(df2)
```

```
## # A tibble: 1 × 3
##      id name  state
##   <dbl> <chr> <chr>
## 1     1 Alice FL   
```

`full_join(x, y)` includes all observations from x and y.

```r
df1
```

```
## # A tibble: 2 × 2
##      id name 
##   <dbl> <chr>
## 1     1 Alice
## 2     2 Bob  
```

```r
df2
```

```
## # A tibble: 2 × 2
##      id state
##   <dbl> <chr>
## 1     1 FL   
## 2     3 NY   
```

```r
# same as full_join(df1, df2)
df1 |> full_join(df2)
```

```
## # A tibble: 3 × 3
##      id name  state
##   <dbl> <chr> <chr>
## 1     1 Alice FL   
## 2     2 Bob   <NA> 
## 3     3 <NA>  NY   
```

Your turn

1. Show `price`, `aircon`, `gasheat`, `garage` for houses that have no `garage`

2. Create a new variable `price_per_bedroom` which is price divided by the number of bedrooms. Show only `price`, `bedrooms`, and `price_per_bedroom` columns and arrange the rows in the descending order of `price_per_bedroom`

3. Create a new variable `has_4_or_more_bedrooms` which is `TRUE` if the house has 4 or more bedrooms and `FALSE` otherwise. Use this variable and `summarise()` to find how many houses have 4 or more bedrooms and how many don't

Package tidyr

Next, we will learn a way to organize data in R--an organization called tidy data.

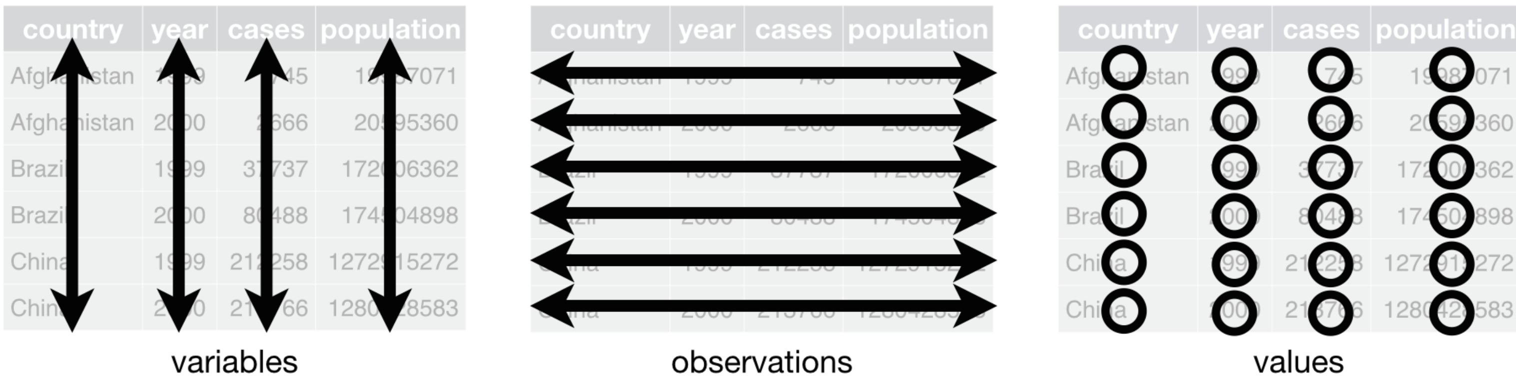
Getting data into this format requires some upfront work, but it pays off in the long run

Marie Kondo was right: **tidying sparks joy!**

Tidy data principles

We say that a data set is *tidy* if it follows the three principles:

- Each `variable` must have its own column.
- Each `observation` must have its own row.
- Each `value` must have its own cell.



While these seem so obvious, most data that you will encounter will be untidy.

Important: Issues such as NA/NULL values, outliers, and unclear column names do not violate any of the tidy data principles.

Example data 1

In `table1` each row is a (country, year) with variables `cases` and `population`.

```
table1
```

```
## # 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
```

Example data 2

In `table2` each row is country, year, variable ("cases", "population") combination, and there is a `count` variable with the numeric value of the combination.

```
table2
```

```
## # A tibble: 12 × 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
##  7 Brazil       2000 cases           80488
##  8 Brazil       2000 population  174504898
##  9 China        1999 cases          212258
## 10 China        1999 population 1272915272
## 11 China        2000 cases          213766
## 12 China        2000 population 1280428583
```

Example data 3

In `table3`, each row is a (country, year) combination with the column rate having the rate of cases to population as a character string in the format `"cases/population"`.

```
table3
```

```
## # A tibble: 6 × 3
##   country      year rate
##   <chr>       <dbl> <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
```

Example data 4

Table 4 is split into two tables, one table for each variable: `table4a` is the table for cases, while `table4b` is the table for population. Within each table, each row is a country, each column is a year, and the cells are the value of the variable for the table.

```
table4a #Numbers can't be column names. Use backticks `...` to force this name
```

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

```
table4b
```

```
## # A tibble: 3 × 3
##   country         `1999`     `2000`
##   <chr>            <dbl>      <dbl>
## 1 Afghanistan   19987071   20595360
## 2 Brazil       172006362  174504898
## 3 China       1272915272 1280428583
```

Questions

Suppose we want to compute the `rate` (case per 10 thousand population for each country per year) for `table1`, `table2`, and `table4a` + `table4b`

Which representation is easiest to work with? Which is hardest? Why?

How to tidy data?

For most real world analyses, you almost alway need to do some tidying on your data.

- The first step is always to figure out what the variables and observations are. Sometimes this is easy; other times you'll need to consult with the people who originally generated the data.

- The second step is to resolve one of two common problems:

    1. One variable might be spread across multiple columns.
    2. One observation might be scattered across multiple rows.

To fix these problems, you'll need the two most important functions in tidyr: `pivot_longer()` and `pivot_wider()`.

- `pivot_longer()` makes wide tables **narrower and longer**;
- `pivot_wider()` makes long tables **shorter and wider**.

Pivot Longer

A common problem is a dataset where some of the column names are not names of variables, but values of a variable.

Take `table4a`: the column names `1999` and `2000` represent values of the `year` variable, and each row represents two observations, not one.

```
table4a
```

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

To tidy a dataset like this, we need to **pivot_longer** those columns into a new pair of variables. To describe that operation we need three parameters:

- A vector of column names that contain values, not variables. In this example, those are the columns `1999` and `2000`.
- The name of the variable whose values form the column name. We call that the `names_to`, and here it is `year`.
- The name of the variable whose values are spread over the cells. We call that `values_to`, and here it's the number of `cases`.

```
table4a |>
  pivot_longer(c(`1999`, `2000`), names_to = "year", values_to = "cases")
```

```
## # A tibble: 6 × 3
##   country     year   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
```

We can use `pivot_longer()` to tidy `table4b` in a similar fashion. The only difference is that the variable is stored in the cell values:

```
table4b |>
  pivot_longer(c(`1999`, `2000`), names_to = "year", values_to = "population")
```

```
## # A tibble: 6 × 3
##   country     year  population
##   <chr>       <chr>      <dbl>
## 1 Afghanistan 1999    19987071
## 2 Afghanistan 2000    20595360
## 3 Brazil      1999   172006362
## 4 Brazil      2000   174504898
## 5 China       1999  1272915272
## 6 China       2000  1280428583
```

To combine the tidied versions of `table4a` and `table4b` into a single tibble, we can use `dplyr::left_join()`

```
tidy4a <- table4a |>
  pivot_longer(c(`1999`, `2000`), names_to = "year", values_to = "cases")

tidy4b <- table4b |>
  pivot_longer(c(`1999`, `2000`), names_to = "year", values_to = "population")

left_join(tidy4a, tidy4b)
```

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

Can you now easily generate the `rate` variable?

Pivot Wider

`Pivot wider` is the opposite of `pivot longer`. You use it when an observation is scattered across multiple rows. For example, take `table2`: an observation is a country in a year, but each observation is spread across two rows.

```
table2
```

```
## # A tibble: 12 × 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
##  7 Brazil       2000 cases           80488
##  8 Brazil       2000 population  174504898
##  9 China        1999 cases          212258
## 10 China        1999 population 1272915272
## 11 China        2000 cases          213766
## 12 China        2000 population 1280428583
```

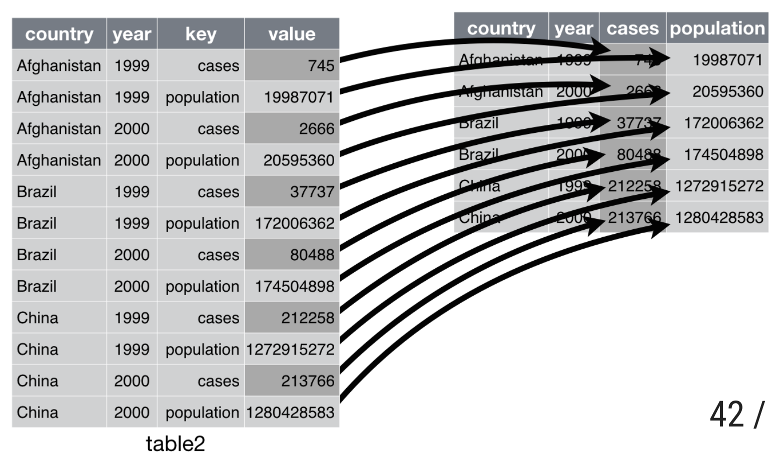
To tidy this up, we first analyze the representation in similar way to `pivot_longer()`. This time, however, we only need two parameters:

- The column that contains variable names, the **names_from** column. Here, it's `type`.
- The column that contains values from multiple variables, the **values_from** column. Here it's `count`.

Once we've figured that out, we can use `pivot_wider()` as below:

```
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
```



Separating and uniting

So far you’ve learned how to tidy `table2` and `table4`, but not `table3`. `table3` has a different problem: we have one column (`rate`) that contains two variables (`cases` and `population`).

To fix this problem, we'll need the `separate()` function. We'll also learn about the complement of `separate()`: `unite()`, which you use if a single variable is spread across multiple columns.

Separate

`separate()` pulls apart one column into multiple columns, by splitting wherever a separator character appears. Take `table3`:

```
table3
```

```
## # A tibble: 6 × 3
##   country      year rate
##   <chr>       <dbl> <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
```

The `rate` column contains both `cases` and `population` variables, and we need to split it into two variables. `separate()` takes the name of the column to separate, and the names of the columns to separate into:

```
table3 |>
  separate(rate, into = c("cases", "population"), sep = "/")
```

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

Look carefully at the column types: you'll notice that `cases` and `population` are character columns. This is the default behaviour in `separate()`: it leaves the type of the column as is. Here, however, it's not very useful as those really are numbers. We can ask `separate()` to try and convert to better types using `convert = TRUE`:

```
table3 |>
  separate(rate, into = c("cases", "population"), sep = "/", convert = TRUE)
```

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

IMPORTANT: The year column does not violate any tidy data principles! Here I simply want to demonstrate that it is possible to separate a value into to two values using a position rather than a pattern.

You can also pass a vector of integers to `sep`. `separate()` will interpret the integers as positions to split at. Positive values start at 1 on the far-left of the strings; negative value start at -1 on the far-right of the strings.

```
table5 <- table3 |>
  separate(year, into = c("century", "year"), sep = 2)
table5
```

```
## # A tibble: 6 × 4
##   country     century year  rate
##   <chr>       <chr>   <chr> <chr>
## 1 Afghanistan 19      99    745/19987071
## 2 Afghanistan 20      00    2666/20595360
## 3 Brazil      19      99    37737/172006362
## 4 Brazil      20      00    80488/174504898
## 5 China       19      99    212258/1272915272
## 6 China       20      00    213766/1280428583
```

Unite

`unite()` is the inverse of `separate()`: it combines multiple columns into a single column. You'll need it much less frequently than `separate()`, but it's still a useful tool to have in your back pocket.

```
table5
```

```
## # A tibble: 6 × 3
##   country     century year
##   <chr>       <chr>   <chr>
## 1 Afghanistan 19      99
## 2 Afghanistan 20      00
## 3 Brazil      19      99
## 4 Brazil      20      00
## 5 China       19      99
## 6 China       20      00
```

```
table5 |>
  unite(year, century, year, sep="")
```

```
## # A tibble: 6 × 2
##   country     year
##   <chr>       <chr>
## 1 Afghanistan 1999
## 2 Afghanistan 2000
## 3 Brazil      1999
## 4 Brazil      2000
## 5 China       1999
## 6 China       2000
```

FAQs about tidy data principles

Tidy data principles:

- Each `variable` must have its own column.
- Each `observation` must have its own row.
- Each `value` must have its own cell.

1. **If a data set has missing values or outliers, does it violate the tidy data principles?** ANS: No, because none of the principles concerns about missing values or outliers
2. **If the column names are poorly named, does it violate the tidy data principles?** ANS: No, because none of the principles concerns about column names
3. **Does "January 1, 2000" violate the third violate the tidy data principle? In other words, should we split it into "January", "1", and "2000"?** ANS: No, because "January 1, 2000" is perfectly reasonable to be deeded as one value, so are "2000-01-01 10:50:59", "2000-01-01" and "2000-01"
4. **Does "Miami, FL" violate the third violate the tidy data principle? In other words, should we split it into "Miami" and "FL"?** ANS: No (typically), because "Miami, FL" is perfectly reasonable to be deeded as one value. FYI: How Many States Have a City Named Miami?

Final words

1. Preparing for group project
    - Read project instruction
    - Project group formation (3 to 5 students in a team)
    - Look into project topics/APIs before the class and rank your preferences
2. Assignment 1 due next week