Statistical Computing with R
Masters in Data Science 503 (S8)
Fourth Batch, SMS, TU, 2025

Shital Bhandary

Associate Professor

Statistics/Bio-statistics, Demography and Public Health Informatics

Patan Academy of Health Sciences, Lalitpur, Nepal

Faculty, Masters in Medical Research, NHRC/Kathmandu University

Faculty, FAIMER Fellowship in Health Professions Education, India/USA

Review Preview (Unit 2, Part 2 & 3)

- Data wrangling
- Data munching
- Tidy data
- “dplyr” package and its use for data manipulation

- Reading database in R
- Big data in R
- Text Mining

Any question on the Session 7, Unit 2?

- Getting data saved as .json file in local computer or website
- We used publicly available json file from nyc.gov website

- Getting data saved as HTML table file
- We used a barebone example followed by getting data on Covid-19 pandemic in Nepal
- We also did some data cleaning, wrangling or munching!

Data wrangling (Course book Chapter 9-16)

R for Data Science, https://r4ds.had.co.nz/index.html

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

- Data wrangling is very important: without it you can't work with your own data! There are three main parts to data wrangling:

    - Import
    - Tidy
    - Transform

Import data in R:

- We have already covered this in the previous classes

- More here: https://r4ds.had.co.nz/data-import.html

- Reach this chapter well as there are some important import functions that are part of this course and may not have discussed so far

- We will discuss about reading “database” in the second part of this class

Tidy data in R

- Tidy data is a consistent way to organize your data in R.

- Getting your/our data into this format requires some upfront work, but that work pays off in the long term.

- Once you/we have tidy data and the tidy tools provided by packages in the **tidyverse**, you will spend much less time **munging/cleaning data** from one representation to another, allowing you to spend more time on the analytic questions at hand.

- Tidy data in tidyverse packages are stored as “tibble”

Let us see what is “tibble” first: https://r4ds.had.co.nz/tibbles.html

- The variant of the data frame used by “tidiverse” is called: **tibble**.
- Tibbles are data frames, but they tweak some older behaviours to make life a little easier.
- R is an old language, and some things that were useful 10 or 20 years ago now get in your way.
- It’s difficult to change base R without breaking existing code, so **most innovation occurs in packages**.
- The **tibble** package provides opinionated data frames that make working in the **tidyverse** a little easier.
- It’s particularly **useful for large datasets** because it only prints the first few rows.

Note:

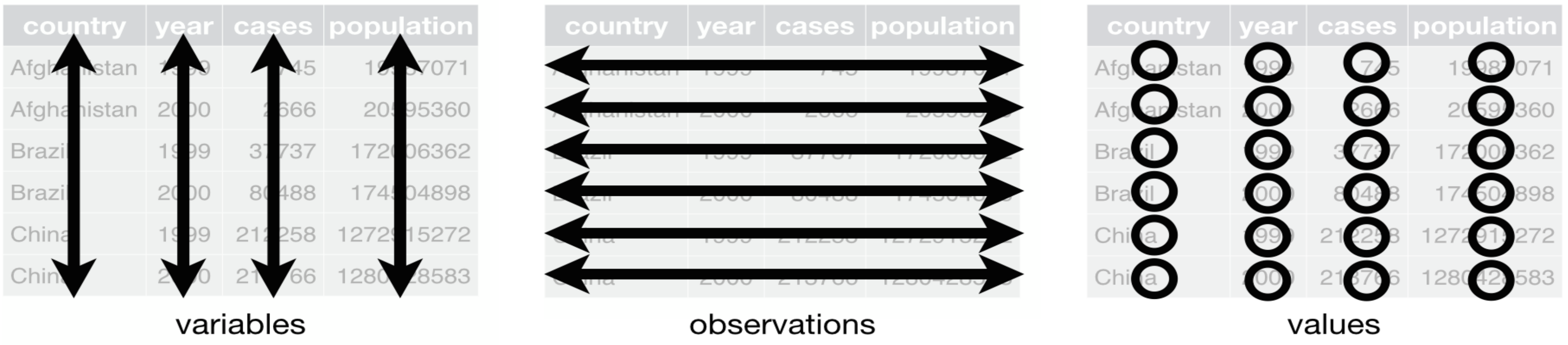
- All the functions of “tidyverse” package works fast with tibble so it will be wise to say that data frame/s should be converted to tibble before using functions of “tidyverse” package

- However, most of the packages of the “tidyverse” super package works well with the data frame too!

- There are two main differences in the usage of a data frame vs a tibble: printing, and subsetting. https://posit.co/blog/tibble-1-0-0/

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.



Example: Which one is "tidy"? Why?

```
#Creating tibble:

table1 <- tibble(
country = c("Afghanistan", "Afghanistan", "Brazil", "Brazil", "China", "China"),
year = c(1999, 2000, 1999, 2000, 1999, 2000),
cases = c(745,2666,37737,80488,212258,213766),
population = c(19987071,20595360,172006362, 174504898, 1272915272,1280428583)
)

# Data frame to tibble: as_tibble(data_frame)
# Tibble to data frame: as.data.frame(tibble_data)
```

- table1
- A tibble: 6 x 4
- year country cases population
- <dbl> <chr> <dbl> <dbl>
- 1 1999 Afghanistan 745 19987071
- 2 2000 Afghanistan" 2666 20595360
- 3 1999 Brazil 37737 172006362
- 4 2000 Brazil 80488 174504898
- 5 1999 China 212258 1272915272
- 6 2000 China 213766 1280428583

- **dbl = duble instead of number in "tibble"!**

Example: Which one is “tidy”? Why?

- table2
- #> # A tibble: 12 × 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

- table3
- #> # A tibble: 6 × 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

Example: Which one is “tidy”? Why?

Spread across two tibbles

table4a # cases

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

table4b # population

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

Why ensure that your data is tidy?
There are two main advantages:

- There's a general advantage to picking one consistent way of storing data. If you have a consistent data structure, it's easier to learn the tools that work with it because they have an underlying uniformity.

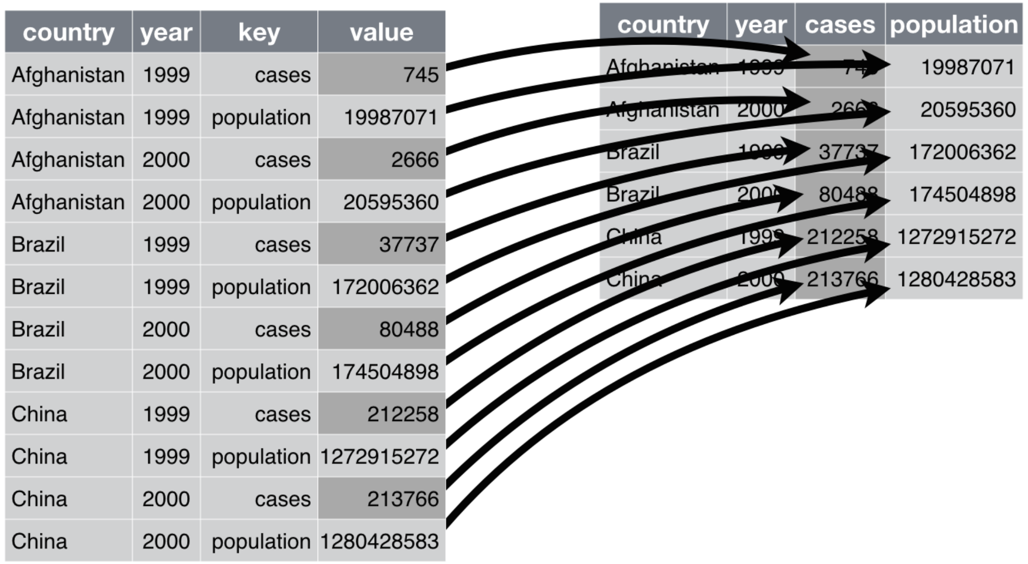
- There's a specific advantage to placing variables in columns because it allows R's vectorized nature to shine.

- dplyr, ggplot2, and all the other packages in the **tidyverse** are designed to work with tidy data.

Tidy data: Pivoting – Longer to wider
(To do standard statistical analysis)

- table2
- #> # A tibble: 12 × 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

```
table2 %>%
  pivot_wider(names_from = type, values_from = count)
```



country	year	key	value
Afghanistan	1999	cases	745
Afghanistan	1999	population	19987071
Afghanistan	2000	cases	2666
Afghanistan	2000	population	20595360
Brazil	1999	cases	37737
Brazil	1999	population	172006362
Brazil	2000	cases	80488
Brazil	2000	population	174504898
China	1999	cases	212258
China	1999	population	1272915272
China	2000	cases	213766
China	2000	population	1280428583

country	year	cases	population
Afghanistan	1999	745	19987071
Afghanistan	2000	2666	20595360
Brazil	1999	37737	172006362
Brazil	2000	80488	174504898
China	1999	212258	1272915272
China	2000	213766	1280428583

table2

Figure 12.3: Pivoting table2 into a wider, tidy form.

Tidy data: Pivoting – Wider to Longer
(To do ANOVA/Variance components analysis)

- table4a
- #> # A tibble: 3 × 3
- #>   country   `1999` `2000`
- #> * <chr>   <int> <int>
- #> 1 Afghanistan   745   2666
- #> 2 Brazil   37737 80488
- #> 3 China   212258 213766

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

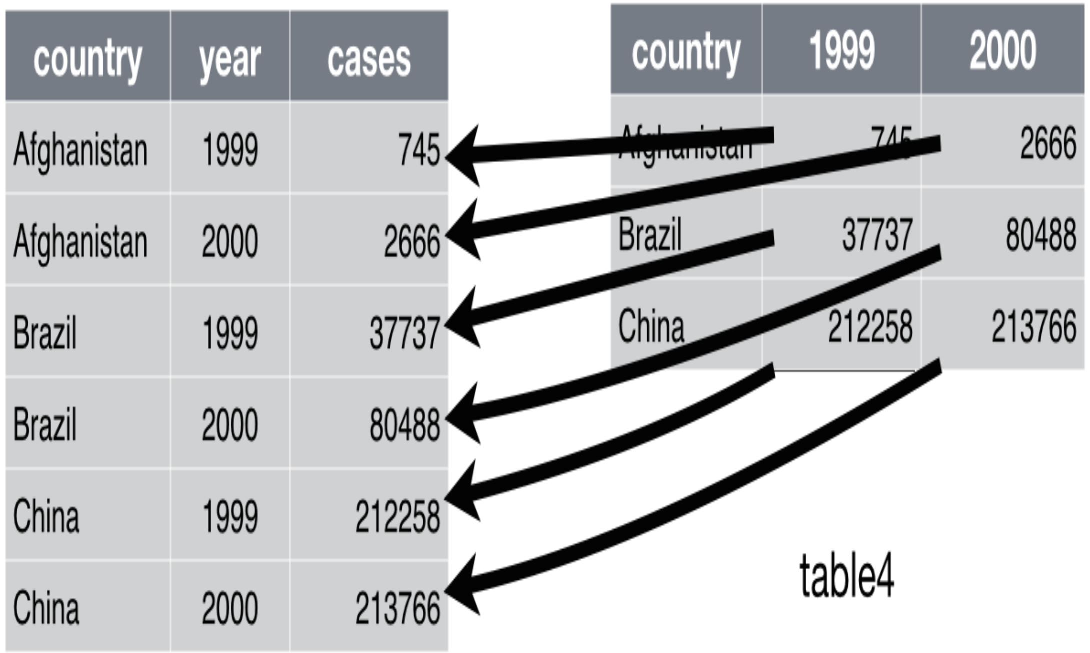


Figure 12.2: Pivoting table4 into a longer, tidy form.

Tidy data: Separate

- table3
- #> # A tibble: 6 × 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

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

country	year	rate
Afghanistan	1999	**745** / 19987071
Afghanistan	2000	**2666** / 20595360
Brazil	1999	**37737** / 172006362
Brazil	2000	**80488** / 174504898
China	1999	**212258** / 1272915272
China	2000	**213766** / 1280428583

table3

country	year	cases	population
Afghanistan	1999	**745**	19987071
Afghanistan	2000	**2666**	20595360
Brazil	1999	**37737**	172006362
Brazil	2000	**80488**	174504898
China	1999	**212258**	1272915272
China	2000	**213766**	1280428583

Figure 12.4: Separating table3 makes it tidy

Tidy data: 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 %>%
  unite(new, century, year)
```

OR

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

country	year	rate
Afghanistan	1999	745 / 19987071
Afghanistan	2000	2666 / 20595360
Brazil	1999	37737 / 172006362
Brazil	2000	80488 / 174504898
China	1999	212258 / 1272915272
China	2000	213766 / 1280428583

country	century	year	rate
Afghanistan	19	99	745 / 19987071
Afghanistan	20	0	2666 / 20595360
Brazil	19	99	37737 / 172006362
Brazil	20	0	80488 / 174504898
China	19	99	212258 / 1272915272
China	20	0	213766 / 1280428583

table6

Figure 12.5: Uniting table5 makes it tidy

Missing values

- Changing the representation of a dataset brings up an important subtlety of missing values.

- Surprisingly, a value can be missing in one of two possible ways:

- **Explicitly**, i.e. flagged with NA (Missingness!)
- **Implicitly**, i.e. simply not present in the data (Nothingness!)

Missing values: Example

```
#Create a tibble with missing values:

stocks <- tibble(
        year   = c(2015, 2015, 2015, 2015, 2016, 2016, 2016),
        qtr    = c(   1,    2,    3,    4,    2,    3,    4),
        return = c(1.88, 0.59, 0.35,   NA, 0.92, 0.17, 2.66)
)
```

Missing values: Example

There are two missing values in this dataset:

- The return for the fourth quarter of 2015 is **explicitly missing**, because the cell where its value should be instead contains NA.

- The return for the first quarter of 2016 is **implicitly missing**, because it simply does not appear in the dataset.

Missing values: Example

- stocks %>%
-   pivot_wider(names_from = year, values_from = return)
- #> # A tibble: 4 × 3
- #>   qtr   `2015`   `2016`
- #>   <dbl>   <dbl>   <dbl>
- #> 1   1   1.88   NA
- #> 2   2   0.59   0.92
- #> 3   3   0.35   0.17
- #> 4   4   NA   2.66

Missing values: What will happen now?

- stocks %>%

```
        pivot_wider(names_from = year, values_from = return) %>%
        pivot_longer(
        cols = c(`2015`, `2016`),
        names_to = "year",
        values_to = "return",
        values_drop_na = TRUE
)
```

Missing values: We can use "complete" command!

- stocks %>%
- complete(year, qtr)
- #> # A tibble: 8 × 3
- #>   year   qtr return
- #>   <dbl> <dbl> <dbl>
- #> 1 2015   1   1.88
- #> 2 2015   2   0.59
- #> 3 2015   3   0.35
- #> 4 2015   4 NA
- #> 5 2016   1 NA
- #> 6 2016   2   0.92
- #> # … with 2 more rows

Missing values: Another example (tibble by row or tribble!)

- treatment <- tribble(

```
~ person,           ~ treatment, ~response,
"Derrick Whitmore", 1,           7,
NA,                 2,           10,
NA,                 3,           9,
"Katherine Burke",  1,           4
)
```

- treatment

Missing values: fill() for another example

- treatment %>%
- fill(person)   **# “tidyr” package is required here!**
- #> # A tibble: 4 × 3
- #>   person   treatment   response
- #>   <chr>   <dbl>   <dbl>
- #> 1 Derrick Whitmore   1   7
- #> 2 Derrick Whitmore   2   10
- #> 3 Derrick Whitmore   3   9
- #> 4 Katherine Burke   1   4

Transform/manipulate data with “dplyr”

- To learn five key “dplyr” package functions that allow you to solve the vast majority of your data manipulation challenges:

  - 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.

Data manipulation with “dplyr”

- These six functions provide the verbs for a language of data manipulation.

- All verbs work similarly:

  - The first argument is a data frame.
  - The subsequent arguments describe what to do with the data frame, using the variable names (without quotes).
  - The result is a new data frame.

- Together these properties make it easy to chain together multiple simple steps to achieve a complex result.

Key reading for the next class:

- https://r4ds.had.co.nz/transform.html

Question/Queries?

- We will practice these six “dplyr” functions in next class!

Thank you!

@shitalbhandary