Tidy Data and Relational Data

Tidy Data

The 3 properties of tidy dataset are:

1. Each variable has its own column.
2. Each observation has its own row.
3. Each value has its own cell.

These 3 properties are visually represented in the following depiction:

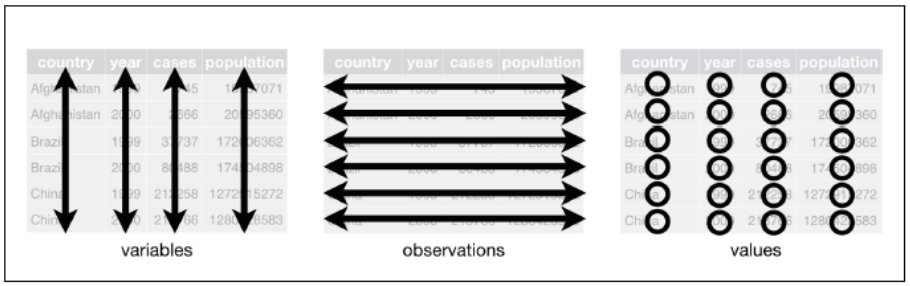


Figure 9-1. The following three rules make a dataset tidy: variables are in columns, observations are in rows, and values are in cells

You might wonder: Why spend time tidying data? One advantage is that having a consistent format of data will make your workflow more efficient. If your workflow leapfrogs the tidy-data stage to the analysis stage, then you'll need to a more expansive knowledge of functions and packages that will allow to accomplish your objectives. On the other, when data are tidy, you'll need a smaller set of tools to undertake analysis. And anyways this advantage ignores that fact that you'll typically have to engage in some tidy procedures just to start your analysis.

Okay. Let's tidy an untidy dataset by working with gapminder data that track global socio-economic indicators. Specifically, we'll import longitudinal data from Github on population, life expectancy and GDP per capita by country beginning in 1799.

```
# Import population data from github; call it population_df
pop_df <- read_csv("https://bit.ly/3qyBCaA")

# Import income data from github; call it income_df
income_df <- read_csv("https://bit.ly/3drOzPP")

# Import life expectancy data from github; call it le_df
le_df <- read_csv("https://bit.ly/3y2JYda")
```

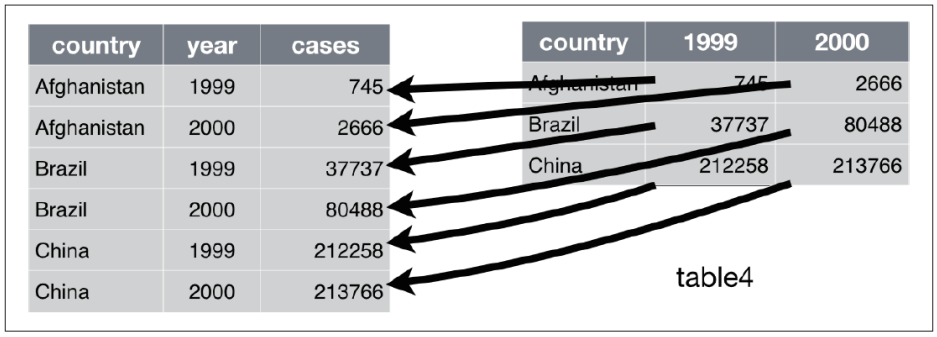
Using `glimpse()` on these data frames, you'll see they have the same structure: The first column is a vector of countries; the rest of the columns are years, every year giving a country's population, life expectancy and GDP per capita.

These data sets aren't tidy: All of them violate Property 1, which requires each variable have its own column.[1]

[1] In statistical work using longitudinal data, the problem with these data sets is that they are originally in *wide* format (each

These are longitudinal data (i.e., data tracking entities (countries) over time (years)), and when dealing with longitudinal data you'll want 2 columns: One for the entities and one for the time variable. To that end, let's transform the `pop_df` data frame into a 3-column tidy dataset: The first column will list countries, the second listing years, and the third column will list the population of each country for each year.

To see where we're headed, take a look at this figure, the dataset depicted untidy in the same manner as these 3 gapminder datasets.



The function you'll need to tidy these data is `gather()` from **tidyverse**. What `gather()` does is to collapse multiple columns (our year columns) into 2 columns: a *key* column and a *value* column. The key is the new variable consisting of the original column names. In our case, it'll be a column, which we'll call `year`, consisting of all column names except `country`. The value consists of all values from the cells of the collapsed columns. For us, the value is the population data, the life expectancy data and the income data.

Just like the other **tidyverse** function we've covered, `gather()` takes a data frame as its first argument. Its second and third arguments are the column name for the key (`key =`) and the column name for value (`value =`). The last argument of `gather()` lists the columns you want to collapse.

Every column not listed in the last argument will not be collapsed, and so you'll usually need to remove some columns from your data frame using `select()`. Otherwise, after running your `gather()` command, you'll likely end up with an undesired data frame. We'll include `select()` in our piped command to remove columns representing future projections (the years after 2010).

Let's tidy the `pop_df` data frame first.

```
# Transform longitudinal data into tidy data
pop_gather_df <- pop_df %>%
  select(country:`2010`) %>%
  gather(key = year, value = pop, `1799`:`2010`)
```

```
# Alternative approach with pivot_longer()
pop_pivot <- pop_df %>%
  select(country:`2010`) %>%
  pivot_longer(!country, names_to = "year", values_to = "pop")
```

Notice something about how the year columns are indexed. Instead of entering the first and last years within quotes, that is, `"1799":"2099"`, you need to enclose the years with backticks, `` `1799`:`2099` ``. This is necessary because, despite the years being names, they're *non-syntactic*, meaning they violate the rules for named objects. As you know, the first symbol of a name must be a letter and not a number. R, however, automatically imports columns with the names from the original data file, irrespective of whether R's naming rules. So you need different notation when calling non-syntactic names.

Though `pop_gather_df` now has a tidy structure, there's a serious obstacle standing in the way of analyzing these data. If you viewed the data frame, you probably noticed the `M`'s and the `k`'s after each country's population to abbreviate numbers in the millions and thousands. R, in turn, interpreted the values as

country gets one row) and should be in *long* format (each row is one year per country).

characters, which is ultimately why both the `year` and `pop` vectors are character vectors in `pop_gather_df`. If you're going to analyze these data, you'll need to convert the `pop` vector to a double vector. This requires 2 separate tasks:

1. Remove the `M`'s and the `k`'s from the entries in `pop`
2. Convert the remaining number to its appropriate form by either multiplying it by 1000000 or 1000.

To that end, we'll add a new vector to `pop_gather_df`, which we'll call `pop_corrected`, with `mutate()`. Now, to be frank, these aren't simple tasks and accomplishing them will require some challenging coding with new functions. But this is such a common problem when collecting data it merits careful attention. Here's the code:

```
# Remove M and k and convert to numbers
pop_gather_df <- pop_gather_df %>%
  mutate(pop_corrected = case_when(
    str_detect(pop_gather_df$pop, "M") ~ parse_number(pop_gather_df$pop)*1e6,
    str_detect(pop_gather_df$pop, "k") ~ parse_number(pop_gather_df$pop)*1e3,
    TRUE ~ parse_number(pop_gather_df$pop)))
```

The command should be familiar until the start of the new column `pop_corrected`'s definition. The definition starts with the `case_when()` function, which is a generalization of the `if_else()`.[2] Like `if_else()`, and as its name suggests, `case_when()` deals with cases, and specifically a group of mutually-exclusive and exhaustive cases. Also like `if_else()`, each case of `case_when()` begins with a logical condition, that is, a statement that either evaluates to `TRUE` or `FALSE`. Each case inside `case_when()` has a left- and right-hand side, the sides separated by `~`. The logical condition takes the left position and, if true, returns the value(s) on the right-hand side of `~`. If false, the next logical condition is checked, continuing until the logical condition is satisfied.

Okay. What cases do you to consider? For this problem, there are 3:

1. There are cells that end with an `M`
2. There are cells that end with an `k`
3. And then there are cells that neither end in `M` nor `k`.

We need to identify each of these 3 cases with a logical condition. To that end, use the `str_detect()` function. This function checks whether a particular string (sequence of letters/numbers) is contained in a vector. If the string is present, then `str_detect()` returns `TRUE`. If not, it evaluates to `FALSE`. So, the first two cases have the following logical conditions: `str_detect(pop_gather_df$pop, 'M')` and `str_detect(pop_gather_df$pop, 'k')`.

Before considering the third and final case, let's examine the right-hand sides of these two conditions. As already noted, the expression on the right-hand side of a `case_when()` argument is evaluated if the logical condition is true. In the present example, if the condition is true, we need the `M` or `k` after the value removed. To do this, we'll use the `parse_number()` function.

The `parse_number()` function deals with a common data problem; namely, that numbers are often surrounded by non-numerical signs like currencies (e.g., $500) and percentages (e.g., 0.7%). The `parse_number()` function drops non-numeric characters before and after numbers. You can then enter `parse_number(pop_gather_df$pop)` to remove the unwanted `M`'s and `k`'s.

But this alone won't solve the problem. To understand why, see what happens when you remove the `M` and `k` from the first and third entries in the `pop` vector These entries are `3.28M` and `400k`.

```
# Remove M from 1st entry in pop
parse_number("3.28M")
```

[2] The `case_when()` function generalizes `if_else()` by allowing more than 2 mutually exclusive cases. While it's true you can embed `if_else()` functions within `if_else()` functions to add more than 2 cases, this is clumsy and laborious compared to `case_when()`.

```
# Remove k from 3rd entry in pop
parse_number("400k")
```

Now, 3.28 million people is obviously more than 400,000 people, but 3.28 "people" are less than 400 people. To rectify the situation, you need to multiply `parse_number("3.28M")` by 1 million (i.e., `1e6`) and `parse_number("400k")` by 1000 (i.e., `1e4`) so that the order of the population numbers is preserved. That's why you see `*1e6` and `*1e4` after the `parse_number()` functions.

There's still the final case to consider: Those entries followed neither by `M` nor `k`. These entries do exist in `pop_gather_df`; the population of the Holy See in 1799 is 905, for example. But even if you're pretty certain all entries are followed by either `M` or `k`, and thus the first 2 cases exhaust all possible cases, it's good practice to include a final case, mutually exclusive of the previous ones, that would capture any instances that didn't satisfy the other logical conditions. In this example, any entry in `pop` that failed to satisfy either the first or second case, will automatically satisfy this final case and its value will be unchanged in the new vector. The final case can then be expressed as: `TRUE ~ parse_number(pop_gather_df$pop)`.[3]

After this longish explanation of what's happening inside `case_when()`, read the code again and try putting it into words. Here's the entire code block again:

```
# Remove M and k and convert to numbers
pop_gather_df <- pop_gather_df %>%
  mutate(pop_corrected = case_when(
    str_detect(pop_gather_df$pop, "M") ~ parse_number(pop_gather_df$pop)*1e6,
    str_detect(pop_gather_df$pop, "k") ~ parse_number(pop_gather_df$pop)*1e3,
    TRUE ~ parse_number(pop_gather_df$pop)))
```

Okay. So we've tidied `pop_gather_df`. Now, use `glimpse()` to verify that `le_df` and `income_df` are both untidy, too. While you're glimpsing the data, also check the data types of the vectors. If the year vectors of either life expectancies or incomes are character vectors, indicated by `<chr>` before its values, this means you'll have to create a new, corrected vector like we just went over.[4]

Since these data sets are untidy, transform them into tidy data frames. As before, restrict the years to be in the range 1799-2010 by using `select()`.

```
# Transform longitudinal data into tidy data
le_gather_df <- le_df %>%
  select(country:`2010`) %>%
  gather(key = year, value = le, `1799`:`2010`)

# Transform longitudinal data into tidy data
income_gather_df <- income_df %>%
  select(country:`2010`) %>%
  gather(key = year, value = income, `1799`:`2010`)

income_gather_df <- income_gather_df %>%
  mutate(income_corrected = case_when(
    str_detect(income_gather_df$income, "k") ~ parse_number(income_gather_df$income)*1e3,
    TRUE ~ parse_number(income_gather_df$income)))
```

[3] We used `parse_number(pop_gather_df$pop)`, instead of simply `pop_gather_df$pop`, because the new vector, `pop_number`, needs to be a double and the `parse_number()` function returns a double.

[4] You might be wondering about R's decision procedure when importing alphanumeric data like the population data with its numbers followed by `M`'s and `k`'s. Does R categorize such vectors as doubles or character vectors or some other data type? The general rule is that, given a vector containing multiple data types, the most complex data type always wins. A character vector is the most complex because each of its entries is a string. This is why R interpreted the years in the `pop_gather_df` as character vectors. Numeric vectors (doubles and integer vectors) are second to character vectors in complexity. Finally, logical vectors are the least complex because their entries can only assume 3 values: `TRUE`, `FALSE`, and `NA`.

Relational Data

What often gets pushed into the background when speaking "big data" is that the data typically aren't found in one dataset. This is almost certainly the case in the early stages of analysis, when variables are located on multiple datasets. One aspect of data wrangling concerns the transformation of data belonging to a particular data frame, whether that be tidying data, subsetting data and so on. Now, we'll consider another aspect: How to merge different but related data sets into a single data frame.

Relational data concerns *pairs* of data sets. So what matters is that each dataset is pairwise related to at least 1 other dataset when merging data.

When data are spread across multiple related tables it's often necessary to *join* them in order to work with them. There are four major ways join data frames `x` and `y`:

- Inner Joins: Keep data that appear in both `x` and `y`
- Left Joins: Keep data that appear in `x`
- Right Joins: Keep data that appear in `y`
- Full Joins: Keep data that appear in either `x` or `y`.

Left joins are the most common, because they add data from a smaller table `y` into a larger table `x` without removing anything from `x`. In what follows, though, we'll concentrate on full joins.

In the previous section you transformed 3 untidy datasets into tidy ones. Merging these datasets, `pop_gather_df`, `income_gather_df` and `le_gather_df`, will prove to be a straightforward exercise in relational data because all 3 data frames uniquely identify any particular observation with the same key. A *key* is a variable or set of variables that uniquely identify observations. Keys are what permit datasets to be related to one another. For our 3 data frames, the variables `country` and `year` together identify any specific observation in their respective data frames.

It's important to recognize that neither `country` nor `year` by itself could serve as a key. Suppose, for example, you only chose `country` as the key to merge these data frames. Each country in these data frames has 212 observations; either a population count, a life expectancy estimate, or income per capita in every year from 1799 until 2010. That is, $(2010 - 1799) + 1 = 212$ observations per country. You can verify this using the `count()` function, which counts unique values for one or more variables. When followed by the `filter` command, `filter(n > 1)`, a data frame counting how many duplicate rows per country is returned:

```
# Count number of unique observations by country
income_gather_df %>% count(country) %>% filter(n > 1)
```

The `country` vector clearly can't uniquely identify observations since, as verified, each country has 212 observations.

Adding `year` inside `count()` verifies that `country` and `year` together identify uniquely each row of the data frame. That is, when looking down the `country` and `year` columns, although some rows have the same value for `country` or for `year`, no two observations have the same values for both variables.

```
# Count number of unique observations by country and by year
income_gather_df %>% count(country, year) %>% filter(n > 1)
```

So we know our key to merge these data frames.

```
# Full join pop_gather_df and income_gather_df using country and year as keys
gapminder_df <- pop_gather_df %>%
  full_join(income_gather_df, by = c("country", "year"))
```

Then join `le_gather_df`.

```
# Full join gapminder_df and income_gather_df using same as keys
gapminder_df <- gapminder_df %>%
  full_join(le_gather_df, by = c("country", "year"))
```

We've merged 3 datasets into 1. Let's join a dataset consisting of country and continent information and then put it all together in a plot.

```
# Import dataset with country names
pais <- read_csv("https://bit.ly/3mrRfk4")

# Join pais df to gapminder_df
paises <- pais %>%
  full_join(gapminder_df, by = "country")
```

Now for the plot:

```
# Plot income v le in 2009 with population and continent
paises %>% filter(year == 2009 & continent != "NA") %>%
  ggplot(mapping = aes(x = income_corrected, y = le, size = pop_corrected)) +
  geom_point(mapping = aes(color = continent)) +
  labs(x = "Income per Capita", y = "Life Expectancy")
```

```
## Warning: Removed 7 rows containing missing values (geom_point).
```