Data Transformations with dplyr and tidyr

Julian Langer

2/18/2019

It is rare that we obtain a data set and it looks exactly as we want it. Usually, we drop variables, transform variables, merge data etc. While it is possible to perform these tasks using base R, there is now a package called dplyr which simplifies them enormously. Before we actually start working with it, we have to clean our workspace and load some packages.

```
rm(list = ls())    # remove all objects from workspace
library(tibble)    # nicer dataframes
library(dplyr)     # data transformations
library(Ecdat)     # Econ datasets
```

While we are at it, let us also load a dataset that we will use in the following. It contains a cross-section of 601 individuals in the United States, some of their characteristics and how many extramarital affairs they had in the past year.

```
affairs = as_data_frame(Fair)
```

```
## Warning: `as_data_frame()` is deprecated, use `as_tibble()` (but mind the new semantics).
## This warning is displayed once per session.
```

```
glimpse(affairs)
```

```
## Observations: 601
## Variables: 9
## $ sex        <fct> male, female, female, male, male, female, female, m...
## $ age        <dbl> 37, 27, 32, 57, 22, 32, 22, 57, 32, 22, 37, 27, 47,...
## $ ym         <dbl> 10.00, 4.00, 15.00, 15.00, 0.75, 1.50, 0.75, 15.00,...
## $ child      <fct> no, no, yes, yes, no, no, no, yes, yes, no, yes, ye...
## $ religious  <int> 3, 4, 1, 5, 2, 2, 2, 2, 4, 4, 2, 4, 5, 2, 4, 1, 2, ...
## $ education  <dbl> 18, 14, 12, 18, 17, 17, 12, 14, 16, 14, 20, 18, 17,...
## $ occupation <int> 7, 6, 1, 6, 6, 5, 1, 4, 1, 4, 7, 6, 6, 5, 5, 5, 4, ...
## $ rate       <int> 4, 4, 4, 5, 3, 5, 3, 4, 2, 5, 2, 4, 4, 4, 4, 5, 3, ...
## $ nbaffairs  <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...
```

As you can see, the dataframe contains 9 variables.

- **sex**: factor: male or female
- **age**: age in years
- **ym**: number of years married
- **child**: factor: yes or no?
- **religious**: How religious from 1 (anti) to 5 (very)?
- **education**: education in years
- **occupation**: occupation, from 1 to 7, according to hollingshead classification
- **rate**: self rating of marriage, from 1 (very unhappy) to 5 (very happy)
- **nbaffairs**: number of affairs in past year

1 Single table verbs

The dplyr package contains a number of single-table verbs which can be used to handle most of the day-to-day data work. The structure of the commands is always the same: verb(dataframe, operation). The output of the functions is again a dataframe. Let's look at each of the verbs in turn. We start with `filter`.

1.1 `filter` to filter observations

Filter can be used to select a subset of observations based on the truth value of a condition. Let's say that we are only interested in the affairs of men.

```
filter(affairs, sex == "male")
```

```
## # A tibble: 286 x 9
##    sex     age    ym child religious education occupation  rate nbaffairs
##    <fct> <dbl> <dbl> <fct>     <int>     <dbl>      <int> <int>     <dbl>
##  1 male     37 10    no            3        18          7     4         0
##  2 male     57 15    yes           5        18          6     5         0
##  3 male     22  0.75 no            2        17          6     3         0
##  4 male     57 15    yes           2        14          4     4         0
##  5 male     22  1.5  no            4        14          4     5         0
##  6 male     37 15    yes           2        20          7     2         0
##  7 male     27  4    yes           4        18          6     4         0
##  8 male     47 15    yes           5        17          6     4         0
##  9 male     37  4    yes           2        20          6     4         0
## 10 male     42 15    yes           5        20          6     4         0
## # ... with 276 more rows
```

As you can see, this gives us dataframe which just contains the observations for men. So, far we have just applied the function `filter` but have not saved the result anywhere. We need to assign it to a new dataframe or reassign it to the existing one.

```
affairs_men = filter(affairs, sex == "male")
head(affairs_men)
```

```
## # A tibble: 6 x 9
##   sex     age    ym child religious education occupation  rate nbaffairs
##   <fct> <dbl> <dbl> <fct>     <int>     <dbl>      <int> <int>     <dbl>
## 1 male     37 10    no            3        18          7     4         0
## 2 male     57 15    yes           5        18          6     5         0
## 3 male     22  0.75 no            2        17          6     3         0
## 4 male     57 15    yes           2        14          4     4         0
## 5 male     22  1.5  no            4        14          4     5         0
## 6 male     37 15    yes           2        20          7     2         0
```

We can of course add further conditions. Let's say we are only interested in men without children.

```
affairs_childless_men = filter(affairs, sex == "male", child == "no")
head(affairs_childless_men)
```

```
## # A tibble: 6 x 9
##   sex     age     ym child religious education occupation  rate nbaffairs
##   <fct> <dbl>  <dbl> <fct>     <int>     <dbl>      <int> <int>     <dbl>
## 1 male     37 10     no            3        18          7     4         0
## 2 male     22  0.75  no            2        17          6     3         0
## 3 male     22  1.5   no            4        14          4     5         0
## 4 male     27  0.417 no            4        17          6     4         0
## 5 male     22  4     no            3        16          5     5         0
## 6 male     22  4     no            1        18          5     5         0
```

You can add as many conditions as you want, these are then linked by a logical 'and'. You can also create more complicated statements with the following logical and relational operators:

- `==` equal
- `!=` not equal
- `!` not
- `&` and
- `|` or
- `>` greater
- `<` smaller
- `>=` greater or equal
- `<=` smaller or equal

So, let's say we want to have the very religious, childless men. This can be accomplished as follows:

```
affairs_childless_zealots = filter(affairs, sex == "male", child == "no", religious == 4 | religious ==
```

So, there you go. One last thing. Sometimes you have missing values in your dataframe, which are denoted as `NA`s (not available). I will create a small dataframe for us in the following.

```
na_df = tibble(
  x = c(1:10),
  y = c(1:2, NA, NA, 3:8)
)
na_df
```

```
## # A tibble: 10 x 2
##         x     y
##     <int> <int>
##  1      1     1
##  2      2     2
##  3      3    NA
##  4      4    NA
##  5      5     3
##  6      6     4
##  7      7     5
##  8      8     6
##  9      9     7
## 10     10     8
```

To check for `NA`, we can use the function `is.na` which returns either `TRUE` or `FALSE`. We can exploit this, if we want to filter `NA`s out.

```
filter(na_df, is.na(y))   # dataframe consisting of observations with NAs
```

```
## # A tibble: 2 x 2
##       x     y
##   <int> <int>
## 1     3    NA
## 2     4    NA
```

```
filter(na_df, !is.na(y))  # dataframe consisting of observations with non-missing values
```

```
## # A tibble: 8 x 2
##       x     y
##   <int> <int>
## 1     1     1
## 2     2     2
## 3     5     3
## 4     6     4
## 5     7     5
## 6     8     6
## 7     9     7
## 8    10     8
```

1.2 arrange to arrange a dataframe according to variables

Next in line, we have the `arrange` command. It can be used to change the order of a dataframe according to one or several variables. For example, assume we want to sort the dataframe by sex and age.

```
affairs = arrange(affairs, sex, age)
```

To change the direction of ordering, use the `desc` function.

```
affairs = arrange(affairs, sex, desc(age)) # sorted from old to young
```

And that is everything there is to know about the `arrange` function.

1.3 select to select/drop variables

Our next verb can be used to select variables from a dataframe. Before we start, let me quickly get rid of the dataframes that we do not need anymore.

```
rm(affairs_childless_men, affairs_childless_zealots, affairs_men, na_df)
```

So, let's remind ourselves what is in our dataset.

```
head(affairs)
```

```
## # A tibble: 6 x 9
##   sex      age    ym child religious education occupation  rate nbaffairs
##   <fct>  <dbl> <dbl> <fct>     <int>     <dbl>      <int> <int>     <dbl>
## 1 female    57    15 yes           4        16          6     4         0
## 2 female    57    15 yes           2        18          5     2         0
## 3 female    57    15 yes           3        18          5     2         0
## 4 female    57    15 no            4        20          6     5         0
## 5 female    57    15 yes           1        18          5     4         2
## 6 female    52    15 yes           5        12          1     3         0
```

We now create a new dataset consisting only of the variables `sex`, `age`, `nbaffairs`.

```
head(select(affairs, sex, age, nbaffairs))
```

```
## # A tibble: 6 x 3
##   sex      age nbaffairs
##   <fct>  <dbl>     <dbl>
## 1 female    57         0
## 2 female    57         0
## 3 female    57         0
## 4 female    57         0
## 5 female    57         2
## 6 female    52         0
```

We could also have given the variables new names while selecting them.

```r
head(select(affairs, sex, age, number_of_affairs = nbaffairs))
```

```
## # A tibble: 6 x 3
##   sex      age number_of_affairs
##   <fct>  <dbl>             <dbl>
## 1 female    57                 0
## 2 female    57                 0
## 3 female    57                 0
## 4 female    57                 0
## 5 female    57                 2
## 6 female    52                 0
```

Note however that there is a better function for renaming variables, `rename`, since it does not drop all the variables that are not explicitly mentioned.

```r
head(rename(affairs, number_of_affairs = nbaffairs))
```

```
## # A tibble: 6 x 9
##   sex     age    ym child religious education occupation  rate
##   <fct> <dbl> <dbl> <fct>     <int>     <dbl>      <int> <int>
## 1 fema~    57    15 yes           4        16          6     4
## 2 fema~    57    15 yes           2        18          5     2
## 3 fema~    57    15 yes           3        18          5     2
## 4 fema~    57    15 no            4        20          6     5
## 5 fema~    57    15 yes           1        18          5     4
## 6 fema~    52    15 yes           5        12          1     3
## # ... with 1 more variable: number_of_affairs <dbl>
```

If you want to keep most variables but just get rid of some, you can use the minus sign. The following command will keep all variables except rate:

```r
head(select(affairs, - rate))
```

```
## # A tibble: 6 x 8
##   sex      age    ym child religious education occupation nbaffairs
##   <fct>  <dbl> <dbl> <fct>      <int>     <dbl>      <int>     <dbl>
## 1 female    57    15 yes           4        16          6         0
## 2 female    57    15 yes           2        18          5         0
## 3 female    57    15 yes           3        18          5         0
## 4 female    57    15 no            4        20          6         0
## 5 female    57    15 yes           1        18          5         2
## 6 female    52    15 yes           5        12          1         0
```

Vertical slicing is also possible. To get all variables from `education` to `nbaffairs` you can write. The minus operator works here as well.

```r
head(select(affairs, education:nbaffairs))
```

```
## # A tibble: 6 x 4
##   education occupation  rate nbaffairs
##       <dbl>      <int> <int>     <dbl>
## 1        16          6     4         0
## 2        18          5     2         0
## 3        18          5     2         0
## 4        20          6     5         0
## 5        18          5     4         2
## 6        12          1     3         0
```

```r
head(select(affairs, -(education:nbaffairs)))
```

```
## # A tibble: 6 x 5
##   sex      age    ym child religious
##   <fct>  <dbl> <dbl> <fct>     <int>
## 1 female    57    15 yes           4
## 2 female    57    15 yes           2
## 3 female    57    15 yes           3
## 4 female    57    15 no            4
## 5 female    57    15 yes           1
## 6 female    52    15 yes           5
```

Note that select also works nicely to reorder the columns of a dataframe. For example, if we want to have the `nbaffairs` in first place, we can write.

```r
head(select(affairs, nbaffairs, everything()))
```

```
## # A tibble: 6 x 9
##   nbaffairs sex      age    ym child religious education occupation  rate
##       <dbl> <fct>  <dbl> <dbl> <fct>     <int>     <dbl>      <int> <int>
## 1         0 female    57    15 yes           4        16          6     4
## 2         0 female    57    15 yes           2        18          5     2
## 3         0 female    57    15 yes           3        18          5     2
## 4         0 female    57    15 no            4        20          6     5
## 5         2 female    57    15 yes           1        18          5     4
## 6         0 female    52    15 yes           5        12          1     3
```

The function `everthing` is a function to capture all other variables except the ones explicitly mentioned. There are also some other helper functions to select all variables that start, end, or contain with a certain string.

- starts_with("string")
- ends_with("string")
- contains("string")
- matches("reg_expression")

Here is one example.

```r
head(select(affairs, starts_with("r")))
```

```
## # A tibble: 6 x 2
##   religious  rate
##       <int> <int>
## 1         4     4
## 2         2     2
## 3         3     2
## 4         4     5
## 5         1     4
## 6         5     3
```

That's it. Now onwards to the `mutate` verb!

1.4 mutate to create new variables

With `mutate` we can create new variables. For example, we might want to create a variable to capture the age at which a person got married. In our dataset we have two variables available to do just this: `age` (unfortunately, only measured in discrete steps, so our measure is somewhat imprecise) and `ym`. So, let's do it.

```
affairs = mutate(affairs, age_married = age - ym)
```

You can use several functions to create new variables:

- arithmetic operators: `+, -, *, /, ^`
- aggregate functions: you can use aggregate functions such as `sum` and `mean` which we will talk about later
- `log`: to create variables in logs, you can use `log`, `log2`, and `log10`
- offsets: you can use `lag` and `lead` to refer to leading or lagged values, provided that your data are grouped (more about that later)
- logical comparisons: you can create Boolean variables using logical comparisons such as `<, <=, >=, !=, ==`.

1.5 `summarise` to create summary tables

The last single-table verb, we'll get to know is summarise. It conjunction with aggregate functions such as `mean` it can be used to collapse our dataframe.

```
summarise(affairs, age = mean(age, na.rm = TRUE))
```

```
## # A tibble: 1 x 1
##     age
##   <dbl>
## 1  32.5
```

As you can see, this collapses our dataframe to just one number, the average age of the persons in the sample. It gets more interesting if we structure our dataframe before applying `summarise`. Say we want to have the average age by sex. (By the way, why did we use the `na.rm` argument here?)

```
affairs_grouped = group_by(affairs, sex)
summarise(affairs_grouped, age = mean(age, na.rm = TRUE))
```

```
## # A tibble: 2 x 2
##   sex      age
##   <fct>  <dbl>
## 1 female  30.8
## 2 male    34.3
```

Aha! Now, we have the mean ages for both men and women! Let's go further by cross-tabulating sex and whether or not the person has children.

```
affairs_grouped = group_by(affairs, sex, child)
summarise(affairs_grouped, age = mean(age, na.rm = TRUE))
```

```
## # A tibble: 4 x 3
## # Groups:   sex [?]
##   sex    child   age
##   <fct>  <fct> <dbl>
## 1 female no     24.5
## 2 female yes    33.7
## 3 male   no     28.7
## 4 male   yes    36.3
```

As you can see the results of the `summarise` verb depend on how we grouped our data beforehand. You can ungroup a dataframe by using the `ungroup` function:

```
affairs_ungrouped = ungroup(affairs_grouped)
summarise(affairs_ungrouped, age = mean(age, na.rm = TRUE)) # returns only one value again since data a
```

```
## # A tibble: 1 x 1
##     age
##   <dbl>
## 1  32.5
```

Of course, `mean` is not the only aggregate function that we can use in combination with `summarise`. Here are some more:

- `sum(x)`: create group-wise sum for variable `x`
- `median(x)`: create group-wise median for variable `x`
- `sd(x)`: group-wise standard deviation for variable `x`
- `min(x)`: minimum by group for variable `x`
- `max(x)`: see above
- `quantile(x, v)`: vth quantile for variable `x` by group
- `first(x)`: first x-value in group by group
- `last(x)`: last x-value in group by group
- `n()`: number of values in group by group
- `n_distinct(x)`: number of distinct values in group by group

2 Using functions in combination with logical subsetting

The Boolean values TRUE and FALSE are treated as 1 and 0, respectively. This makes them useful in combination with our aggregate functions. Say we want to have the total number of very religious people. We can use `summarize` for this.

```
summarise(affairs, no_very_religious = sum(religious >= 4))
```

```
## # A tibble: 1 x 1
##   no_very_religious
##               <int>
## 1               260
```

How does this work? Well the expression `religious >= 4` produces a vector of the same length as `religious` with TRUE and FALSE values. If we sum over these values, we add a 1 for every observation for which the statement is true.

We can also use subsetting in a different way. Let's say we want to have the mean age of all those very religious people. We can then write:

```
summarise(affairs, mean_age = mean(age[religious >= 4]))
```

```
## # A tibble: 1 x 1
##   mean_age
##      <dbl>
## 1     34.5
```

OK, let's try to understand this. The expression `age[religious >= 4]` selects only those observations from the age vector for which `religious >= 4` is true. Then, the mean of these observations is calculated. You might get a feeling now how powerful the interplay of logical subsetting and the functions can be.

3 The pipe operator

Now that we understand `dplyr` a little better, let's make our code more beautiful and readible at the same time. I now introduce the pipe operator. Before, let's quickly get rid of our mess.

```
rm(list = c("affairs_grouped", "affairs_ungrouped"))
```

We now have our original dataframe again. Let's say I want to select the variables `age`, `ym`, and `nbaffairs`, create the variable which gives us age at marriage and sort the data by the number of affairs and our new variables. The wordy way looks like this:

```
affairs = select(affairs, age, ym, nbaffairs)
affairs = mutate(affairs, age_married = age - ym)
affairs = arrange(affairs, nbaffairs, age_married)
head(affairs)
```

```
## # A tibble: 6 x 4
##     age    ym nbaffairs age_married
##   <dbl> <dbl>     <dbl>       <dbl>
## 1  17.5 10            0         7.5
## 2  22    7            0        15
## 3  22    7            0        15
## 4  22    7            0        15
## 5  17.5  1.5          0        16
## 6  17.5  0.75         0        16.8
```

Now, I don't know about you but I have the feeling that I wrote the word `affairs` way too many times. This is because we a) have to reassign the new dataframe to the old dataframe so many times and b) always have to make sure that R knows on which dataframe we are using our verbs. Here, using the pipe operator comes in handy. Let's first restore our original dataset.

```
affairs = as_data_frame(Fair) # restore dataframe
```

We can alternatively write the code as:

```
affairs = affairs %>%
  select(age, ym, nbaffairs) %>%
  mutate(age_married = age - ym) %>%
  arrange(nbaffairs, age_married)
head(affairs)
```

Now, this look much nicer, right? We can skip the tedious reassignments and also don't have to tell R on which dataframe we want to operate. You can read this as follows:

1. Take the affairs dataframe and apply the `select` verb on it. The result is a new dataframe with the three variables.
2. Take this dataframe of three variables and apply the `mutate` verb to it. The result is a new dataframe which also includes the variable age_married.
3. Take the new dataframe and apply the `arrange` verb to it. The result is a new dataframe.
4. Assign this new dataframe to our variable `affairs`.

Neat, eh? Piping works so nicely, because the input and output of `dplyr` verbs is always a dataframe. You can use the pipe operator also with other packages and we will use it when it is reasonable to do so.

4 Multiple table verbs

`Dplyr` also features verbs which you can apply to two tables. These are usually functions to join or merge two datasets.

4.1 Mutating joins

Mutating joins allow you to combine variables from multiple tables. To see how they work, let's create two small tibbles.

```r
first_df = tibble(
  country = c('Afghanistan', 'Belgium', 'China', 'Denmark'),
  population = c(33369945, 11371928, 1382323332, 5690750)
)

# gdp in millions
second_df = tibble(
  country = c('Afghanistan', 'Belgium', 'Denmark', 'Germany'),
  gdp = c(35146, 422809, 211916, 3232545)
)

head(first_df)
```

```
## # A tibble: 4 x 2
##   country     population
##   <chr>            <dbl>
## 1 Afghanistan   33369945
## 2 Belgium       11371928
## 3 China       1382323332
## 4 Denmark        5690750
```

```r
head(second_df)
```

```
## # A tibble: 4 x 2
##   country         gdp
##   <chr>         <dbl>
## 1 Afghanistan   35146
## 2 Belgium      422809
## 3 Denmark      211916
## 4 Germany     3232545
```

4.1.1 `left_join`

The `left_join` function matches rows from the second dataframe to the first dataframe. All rows from the first dataframe are kept.

```r
left_join(first_df, second_df, by = "country")
```

```
## # A tibble: 4 x 3
##   country     population    gdp
##   <chr>            <dbl>  <dbl>
## 1 Afghanistan   33369945  35146
## 2 Belgium       11371928 422809
## 3 China       1382323332     NA
## 4 Denmark        5690750 211916
```

4.1.2 `right_join`

The `right_join` function does exactly the opposite. It matches rows from the second dataframe to the first dataframe. All rows from the second dataframe are kept.

```r
right_join(first_df, second_df, by = "country")
```

```
## # A tibble: 4 x 3
##   country     population     gdp
##   <chr>            <dbl>   <dbl>
## 1 Afghanistan   33369945   35146
## 2 Belgium       11371928  422809
## 3 Denmark        5690750  211916
## 4 Germany             NA 3232545
```

4.1.3 `inner_join`

The `inner_join` function joins the data from both dataframes but only keeps those observations which exist in both dataframes.

```r
inner_join(first_df, second_df, by = "country")
```

```
## # A tibble: 3 x 3
##   country     population    gdp
##   <chr>            <dbl>  <dbl>
## 1 Afghanistan   33369945  35146
## 2 Belgium       11371928 422809
## 3 Denmark        5690750 211916
```

4.1.4 `full_join`

The `full_join` function also joins the data from both dataframes but keeps all observations.

```r
full_join(first_df, second_df, by = "country")
```

```
## # A tibble: 5 x 3
##   country     population     gdp
##   <chr>            <dbl>   <dbl>
## 1 Afghanistan   33369945   35146
## 2 Belgium       11371928  422809
## 3 China       1382323332      NA
## 4 Denmark        5690750  211916
## 5 Germany             NA 3232545
```

4.2 Filtering joins

Filtering joins are helpful if you want to filter the observations in one dataset based on whether they exist in the other dataset.

4.2.1 `semi_join`

The `semi_join` function keeps all observations in the first dataframe which also exist in the second.

```r
semi_join(first_df, second_df, by = "country")
```

```
## # A tibble: 3 x 2
##   country     population
##   <chr>            <dbl>
## 1 Afghanistan   33369945
## 2 Belgium       11371928
## 3 Denmark        5690750
```

4.2.2 anti_join

The anti_join function drops all observations in the first dataframe which also exist in the second.

```
anti_join(first_df, second_df, by = "country")
```

```
## # A tibble: 1 x 2
##   country population
##   <chr>        <dbl>
## 1 China   1382323332
```

4.3 The by argument

The by argument controls by what variables two dataframes are matched. If you do not specify it, dplyr uses all variables that exist in both tables, a so-called natural join. In our examples we could also have left the by argument unspecified since both dataframes only share the variable country.

You can also pass character vector to specify by which variable the two dataframes are supposed to be matched. This is what we did in the preceding examples.

Finally, what to do if the variable by which you want to match has different names in the two dataframes. Use a named character vector!

```
first_df = tibble(
  country = c('Afghanistan', 'Belgium', 'China', 'Denmark'),
  population = c(33369945, 11371928, 1382323332, 5690750)
)

# gdp in millions
second_df = tibble(
  country_name = c('Afghanistan', 'Belgium', 'Denmark', 'Germany'),
  gdp = c(35146, 422809, 211916, 3232545)
)

full_join(first_df, second_df, by = c('country' = 'country_name'))
```

```
## # A tibble: 5 x 3
##   country     population     gdp
##   <chr>            <dbl>   <dbl>
## 1 Afghanistan   33369945   35146
## 2 Belgium       11371928  422809
## 3 China       1382323332      NA
## 4 Denmark        5690750  211916
## 5 Germany             NA 3232545
```

As you can see, this matched variable country in our first dataframe with country_name in our second dataframe. The variable name for the resulting dataframe is country.

5 Reshaping data with tidyr

Another thing that happens very often is that the datasets we obtain are not in the right format. What do I mean by ‘right’ here? You already talked with Lachlan about this but some basic principles are:

1. Every row is an observation.

2. Every column is a variable.
3. Each cell contains the value of a variable for a specific observation.

Datasets which conform to these three principles are called `tidy` in the `R` world. There is now package called `tidyr` which helps you in converting untidy datasets to tidy ones.

```
library(tidyr)
```

In the following we will look at two cases which I will call dirty wide and dirty long datasets.

5.1 From dirty wide to tidy long

Have a look at the following small dataset, containing population data.

```
dirty_wide = as_data_frame(table4a)
head(dirty_wide)
```

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

Are these tidy? No! First of all, we have column names which are actually variable values: 1999 and 2000. Secondly, it is not clear what data the cells contain. To reshape the data from dirty long to tidy long, we use the `gather` command.

```
gather(dirty_wide, `1999`, `2000`, key = 'year', value = 'population')
```

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

How did this work? The first argument is the name of the dataset. The following arguments are the column names which are actually the values of a variable. We want these values to be stored in a variable called `year` so we write: `key = 'year'`. 'Key' in this context is just another word for variable name. Finally, we specify the variable name `population` for the values in our cells by writing `value = 'population'`. The result is a tidy long dataframe.

5.2 From dirty long to tidy wide

We can also have a dirty long dataframe. Look at the following tibble.

```
dirty_long = as_data_frame(table2)
head(dirty_long)
```

```
## # A tibble: 6 x 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
```

Again this dataframe is not tidy. The variable `tidy` contains variable names and the `count` variable the corresponding values. We want to reshape these data and create two variables `cases` and `population` with the cells containing the corresponding values. This can be accomplished using the `spread` command.

```
spread(dirty_long, key = type, value = count)
```

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

The first argument is the name of the dataframe again. Secondly, we specify the column, `type` in this case, which contains the variable names or keys. Finally, we specify the variable, `count`, which contains the values of the variables. The resulting dataframe is tidy again.

Sources

You can find the cheat sheet for `dplyr` and `tidyr` here: https://www.rstudio.com/wp-content/uploads/2015/02/data-wrangling-cheatsheet.pdf. If you want to know more about the principles underlying tidy data, have a look at the following article by Hadley Wickham: https://www.jstatsoft.org/article/view/v059i10. There is also another packages called `reshape2` which can do everything that `tidyr` can and more: https://cran.r-project.org/web/packages/reshape2/index.html.