16 How to make a data frame longer

What will this tutorial cover?

In this tutorial you will learn how to make data frames longer. Most of the time you do this when your data frame is not tidy. We'll explain what an un-tidy data frame is and explore a few techniques to making data frames longer.

Who do I have to thank?

For this tutorial I have to thank Hadley Wickham for his article on Tidy Data (Wickham, 2014). Making data frames longer is closely related to tidy data. I used some ideas and examples from his article for this tutorial.

Many data sets are created or cleaned in spreadsheet programs. These programs are optimized for easy data entry and visual review. As a result, people tend to write un-tidy data.

Un-tidy data violates one of these three principles in one way or another (see Wickham, 2014):

- Each variable forms a column
- Each observation forms a row
- Each type of observation unit is a table

One could also say that in an un-tidy dataset, the physical layout is not linked to its semantics (Neo, 2020).

Suppose you receive the following data set from a colleague. Two groups of people "a" and "b" had to indicate for three weeks how often they ran in one week. The columns `w-1` to `w-3` represent the weeks 1 to 3:

```
(running_data <- tribble(
   ~person, ~group,  ~`w-1`, ~`w-2`, ~`w-3`,
    "John",    "a",    4,       NA,       2,
   "Marie",    "a",    2,        7,       3,
    "Jane",    "b",    3,        8,       9,
   "Peter",    "b",    1,        3,       3
```

```
  ))
```

```
# A tibble: 4 x 5
  person group `w-1` `w-2` `w-3`
  <chr>  <chr> <dbl> <dbl> <dbl>
1 John   a         4    NA     2
2 Marie  a         2     7     3
3 Jane   b         3     8     9
4 Peter  b         1     3     3
```

This data frame is un-tidy because not all columns represent variables. According to Wickham (2014), a variable contains "all values that measure the same underlying attribute (such as altitude, temperature, duration) across units" (p. 3). However, in our data frame, the columns `w-1`, `w-2`, and `w-3` represent values of a underlying variable `week` that is not represented as a column. A tidy representation of this data frame would look as follows:

```
  running_data %>%
    pivot_longer(
      cols = `w-1`:`w-3`,
      names_to = "week",
      values_to = "value"
    )
```

```
# A tibble: 12 x 4
   person group week  value
   <chr>  <chr> <chr> <dbl>
 1 John   a     w-1       4
 2 John   a     w-2      NA
 3 John   a     w-3       2
 4 Marie  a     w-1       2
 5 Marie  a     w-2       7
 6 Marie  a     w-3       3
 7 Jane   b     w-1       3
 8 Jane   b     w-2       8
 9 Jane   b     w-3       9
10 Peter  b     w-1       1
11 Peter  b     w-2       3
12 Peter  b     w-3       3
```

You can already see that I used the `pivot_longer` function to create this data frame, which we will get to know in this tutorial. But before we dive deeper, let's get back to the main topic

of this tutorial: How to make a data frame longer. A data frame gets longer when we increase the number of its rows and decrease the number of its columns (see Pivoting). So when we tidy an un-tidy data set, we essentially make the data set longer.

In the following sections, we will go through some common use cases for cleaning up un-tidy data and use `pivot_longer` to make them longer. The use cases are mainly from Wickham (2014):

- Column headers are values of one variable, not variable names
- Multiple variables are stored in columns
- Multiple variables are stored in one column
- Variables are stored in both rows and columns.
- Variables are stored in both rows and columns

We will use some of the data sets from his article, but also others to increase the variability of the examples. Besides the use cases, we will also learn about some important parameters of the `pivot_longer` function.

16.1 Column headers are values of one variable, not variable names

We have already seen this problem in action in our running data set. The data set contains columns that represented values and not variable names:

```
running_data %>% colnames
```

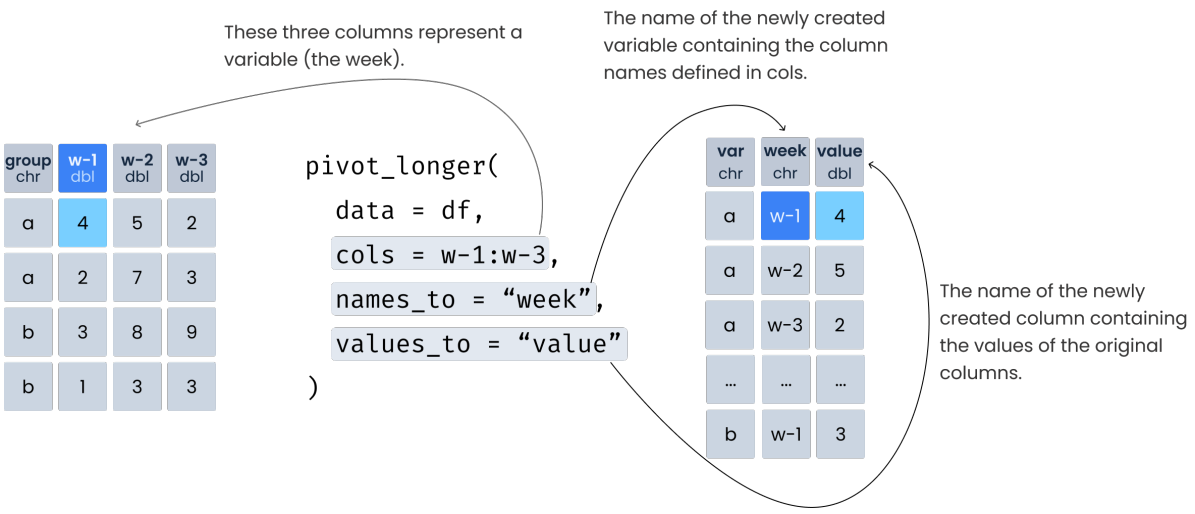
```
[1] "person" "group"  "w-1"    "w-2"    "w-3"
```

To make this data frame longer and tidy, we need to specify arguments for these four parameters in `pivot_longer`:

- `data`: The data frame to make longer
- `columns`: The columns that should be converted to a longer format
- `names_to`: The name of the new column that will contain the names of the columns
- `values_to`: The name of the new column that will contain the values inside the `cols` columns

Let's run the function and see what the results look like:

```
running_data %>%
  pivot_longer(
    cols = `w-1`:`w-3`,
    names_to = "week",
```

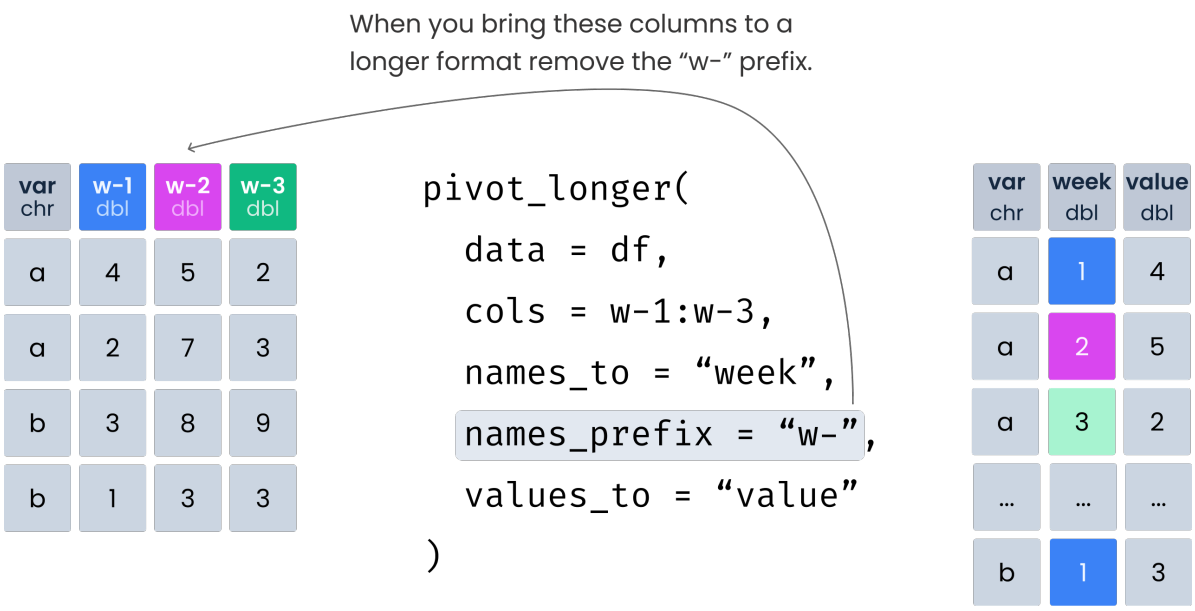



```
    values_to = "value"
  )
```

```
# A tibble: 12 x 4
   person group week  value
   <chr>  <chr> <chr> <dbl>
 1 John   a     w-1       4
 2 John   a     w-2      NA
 3 John   a     w-3       2
 4 Marie  a     w-1       2
 5 Marie  a     w-2       7
 6 Marie  a     w-3       3
 7 Jane   b     w-1       3
 8 Jane   b     w-2       8
 9 Jane   b     w-3       9
10 Peter  b     w-1       1
11 Peter  b     w-2       3
12 Peter  b     w-3       3
```

While our wider data frame had 20 values, our longer data frame has 48 values. We have more than doubled the number of values. The reason for this is that we duplicated the previous column names in our `week` column and also duplicated the values in the `person` and `group` columns.

We can further improve this code by removing the prefixes in the `week` column. `w-1` for example should be displayed as `1`.




```
running_data %>%
  pivot_longer(
    cols = `w-1`:`w-3`,
    names_to = "week",
    values_to = "value",
    names_prefix = "w-"
  )
```

```
# A tibble: 12 x 4
   person group week  value
   <chr>  <chr> <chr> <dbl>
 1 John   a     1         4
 2 John   a     2        NA
 3 John   a     3         2
 4 Marie  a     1         2
 5 Marie  a     2         7
 6 Marie  a     3         3
 7 Jane   b     1         3
 8 Jane   b     2         8
 9 Jane   b     3         9
10 Peter  b     1         1
11 Peter  b     2         3
12 Peter  b     3         3
```

You may also see that the variable `week` is a character and not a double. To convert the data type of the column `names_to` we can use the parameter `names_transform`:

```
running_data %>%
  pivot_longer(
    cols = `w-1`:`w-3`,
    names_to = "week",
    values_to = "value",
    names_prefix = "w-",
    names_transform = as.double
  )
```

```
# A tibble: 12 x 4
   person group  week value
   <chr>  <chr> <dbl> <dbl>
 1 John   a         1     4
 2 John   a         2    NA
 3 John   a         3     2
 4 Marie  a         1     2
 5 Marie  a         2     7
 6 Marie  a         3     3
 7 Jane   b         1     3
 8 Jane   b         2     8
 9 Jane   b         3     9
10 Peter  b         1     1
11 Peter  b         2     3
12 Peter  b         3     3
```

Similarly, you could convert the data type of the `values_to` column with `values_transform` (a factor in this case):

```
running_data %>%
  pivot_longer(
    cols = `w-1`:`w-3`,
    names_to = "week",
    values_to = "value",
    names_prefix = "w-",
    values_transform = as.factor
  )
```

```
# A tibble: 12 x 4
```

```
   person group week  value
   <chr>  <chr> <chr> <fct>
 1 John   a     1     4
 2 John   a     2     <NA>
 3 John   a     3     2
 4 Marie  a     1     2
 5 Marie  a     2     7
 6 Marie  a     3     3
 7 Jane   b     1     3
 8 Jane   b     2     8
 9 Jane   b     3     9
10 Peter  b     1     1
11 Peter  b     2     3
12 Peter  b     3     3
```

Here is another example where column headers represent values (from the tidyr package):

```
relig_income
```

```
# A tibble: 18 x 11
   religion      `<$10k` $10-2~1 $20-3~2 $30-4~3 $40-5~4 $50-7~5 $75-1~6 $100-~7
   <chr>           <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>
 1 Agnostic           27      34      60      81      76     137     122     109
 2 Atheist            12      27      37      52      35      70      73      59
 3 Buddhist           27      21      30      34      33      58      62      39
 4 Catholic          418     617     732     670     638    1116     949     792
 5 Don't know/r~      15      14      15      11      10      35      21      17
 6 Evangelical ~     575     869    1064     982     881    1486     949     723
 7 Hindu               1       9       7       9      11      34      47      48
 8 Historically~     228     244     236     238     197     223     131      81
 9 Jehovah's Wi~      20      27      24      24      21      30      15      11
10 Jewish             19      19      25      25      30      95      69      87
11 Mainline Prot     289     495     619     655     651    1107     939     753
12 Mormon             29      40      48      51      56     112      85      49
13 Muslim              6       7       9      10       9      23      16       8
14 Orthodox           13      17      23      32      32      47      38      42
15 Other Christ~       9       7      11      13      13      14      18      14
16 Other Faiths       20      33      40      46      49      63      46      40
17 Other World ~       5       2       3       4       2       7       3       4
18 Unaffiliated      217     299     374     365     341     528     407     321
# ... with 2 more variables: `>150k` <dbl>, `Don't know/refused` <dbl>, and
#   abbreviated variable names 1: `$10-20k`, 2: `$20-30k`, 3: `$30-40k`,
```

```
#   4: `$40-50k`, 5: `$50-75k`, 6: `$75-100k`, 7: `$100-150k`
```

The columns `<$10k` to `Don't know/Refused` are values of an underlying variable income. The values under these columns indicate the frequency with which individuals reported having a certain income. Let us tidy this data frame by making it longer:

```
relig_income %>%
  pivot_longer(
    cols = `<$10k`:`Don't know/refused`,
    names_to = "income",
    values_to = "freq"
  )
```

```
# A tibble: 180 x 3
   religion income              freq
   <chr>    <chr>              <dbl>
 1 Agnostic <$10k                 27
 2 Agnostic $10-20k               34
 3 Agnostic $20-30k               60
 4 Agnostic $30-40k               81
 5 Agnostic $40-50k               76
 6 Agnostic $50-75k              137
 7 Agnostic $75-100k             122
 8 Agnostic $100-150k            109
 9 Agnostic >150k                 84
10 Agnostic Don't know/refused    96
# ... with 170 more rows
```

Again, our tidy data frame has more values ($180 * 3 = 540$) than our un-tidy data frame ($18 * 11 = 198$). Going back to our original statement that spreadsheet software is made for easy data entry, we can clearly see that it is easier to work with 198 values than 540.

Finally, another example. The data set `billboard` contains the top billboard rankings of the year 2000:

```
billboard
```

```
# A tibble: 317 x 79
   artist track date.ent~1   wk1   wk2   wk3   wk4   wk5   wk6   wk7   wk8   wk9
   <chr>  <chr> <date>     <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
 1 2 Pac  Baby~ 2000-02-26    87    82    72    77    87    94    99    NA    NA
```

```
 2 2Ge+h~ The ~ 2000-09-02    91    87    92    NA    NA    NA    NA    NA    NA
 3 3 Doo~ Kryp~ 2000-04-08    81    70    68    67    66    57    54    53    51
 4 3 Doo~ Loser 2000-10-21    76    76    72    69    67    65    55    59    62
 5 504 B~ Wobb~ 2000-04-15    57    34    25    17    17    31    36    49    53
 6 98^0   Give~ 2000-08-19    51    39    34    26    26    19     2     2     3
 7 A*Tee~ Danc~ 2000-07-08    97    97    96    95   100    NA    NA    NA    NA
 8 Aaliy~ I Do~ 2000-01-29    84    62    51    41    38    35    35    38    38
 9 Aaliy~ Try ~ 2000-03-18    59    53    38    28    21    18    16    14    12
10 Adams~ Open~ 2000-08-26    76    76    74    69    68    67    61    58    57
# ... with 307 more rows, 67 more variables: wk10 <dbl>, wk11 <dbl>,
#   wk12 <dbl>, wk13 <dbl>, wk14 <dbl>, wk15 <dbl>, wk16 <dbl>, wk17 <dbl>,
#   wk18 <dbl>, wk19 <dbl>, wk20 <dbl>, wk21 <dbl>, wk22 <dbl>, wk23 <dbl>,
#   wk24 <dbl>, wk25 <dbl>, wk26 <dbl>, wk27 <dbl>, wk28 <dbl>, wk29 <dbl>,
#   wk30 <dbl>, wk31 <dbl>, wk32 <dbl>, wk33 <dbl>, wk34 <dbl>, wk35 <dbl>,
#   wk36 <dbl>, wk37 <dbl>, wk38 <dbl>, wk39 <dbl>, wk40 <dbl>, wk41 <dbl>,
#   wk42 <dbl>, wk43 <dbl>, wk44 <dbl>, wk45 <dbl>, wk46 <dbl>, wk47 <dbl>, ...
```

Looking at the columns, we find a whopping 76 column names that are values of a variable week and not variables (`wk1` to `wk76`):

```
billboard %>%
  colnames
```

```
 [1] "artist"       "track"        "date.entered" "wk1"          "wk2"
 [6] "wk3"          "wk4"          "wk5"          "wk6"          "wk7"
[11] "wk8"          "wk9"          "wk10"         "wk11"         "wk12"
[16] "wk13"         "wk14"         "wk15"         "wk16"         "wk17"
[21] "wk18"         "wk19"         "wk20"         "wk21"         "wk22"
[26] "wk23"         "wk24"         "wk25"         "wk26"         "wk27"
[31] "wk28"         "wk29"         "wk30"         "wk31"         "wk32"
[36] "wk33"         "wk34"         "wk35"         "wk36"         "wk37"
[41] "wk38"         "wk39"         "wk40"         "wk41"         "wk42"
[46] "wk43"         "wk44"         "wk45"         "wk46"         "wk47"
[51] "wk48"         "wk49"         "wk50"         "wk51"         "wk52"
[56] "wk53"         "wk54"         "wk55"         "wk56"         "wk57"
[61] "wk58"         "wk59"         "wk60"         "wk61"         "wk62"
[66] "wk63"         "wk64"         "wk65"         "wk66"         "wk67"
[71] "wk68"         "wk69"         "wk70"         "wk71"         "wk72"
[76] "wk73"         "wk74"         "wk75"         "wk76"
```

Even though this data frame is much wider than our previous examples, we can use the same function and parameters to make it longer:

```
billboard %>%
  pivot_longer(
    cols = contains("wk"),
    names_to = "week",
    values_to = "value",
    names_prefix = "^wk",
    names_transform = as.double
  )
```

```
# A tibble: 24,092 x 5
   artist track                   date.entered  week value
   <chr>  <chr>                   <date>       <dbl> <dbl>
 1 2 Pac  Baby Don't Cry (Keep... 2000-02-26       1    87
 2 2 Pac  Baby Don't Cry (Keep... 2000-02-26       2    82
 3 2 Pac  Baby Don't Cry (Keep... 2000-02-26       3    72
 4 2 Pac  Baby Don't Cry (Keep... 2000-02-26       4    77
 5 2 Pac  Baby Don't Cry (Keep... 2000-02-26       5    87
 6 2 Pac  Baby Don't Cry (Keep... 2000-02-26       6    94
 7 2 Pac  Baby Don't Cry (Keep... 2000-02-26       7    99
 8 2 Pac  Baby Don't Cry (Keep... 2000-02-26       8    NA
 9 2 Pac  Baby Don't Cry (Keep... 2000-02-26       9    NA
10 2 Pac  Baby Don't Cry (Keep... 2000-02-26      10    NA
# ... with 24,082 more rows
```

16.2 Multiple variables are stored in columns

The previous use cases were quite clear, since we could assume that the columns used for `cols` represent all values of a single variable. However, this is not always the case. Let's take the data set `anscombe`:

```
anscombe
```

```
   x1 x2 x3 x4    y1   y2    y3    y4
1  10 10 10  8  8.04 9.14  7.46  6.58
2   8  8  8  8  6.95 8.14  6.77  5.76
3  13 13 13  8  7.58 8.74 12.74  7.71
4   9  9  9  8  8.81 8.77  7.11  8.84
5  11 11 11  8  8.33 9.26  7.81  8.47
6  14 14 14  8  9.96 8.10  8.84  7.04
7   6  6  6  8  7.24 6.13  6.08  5.25
```

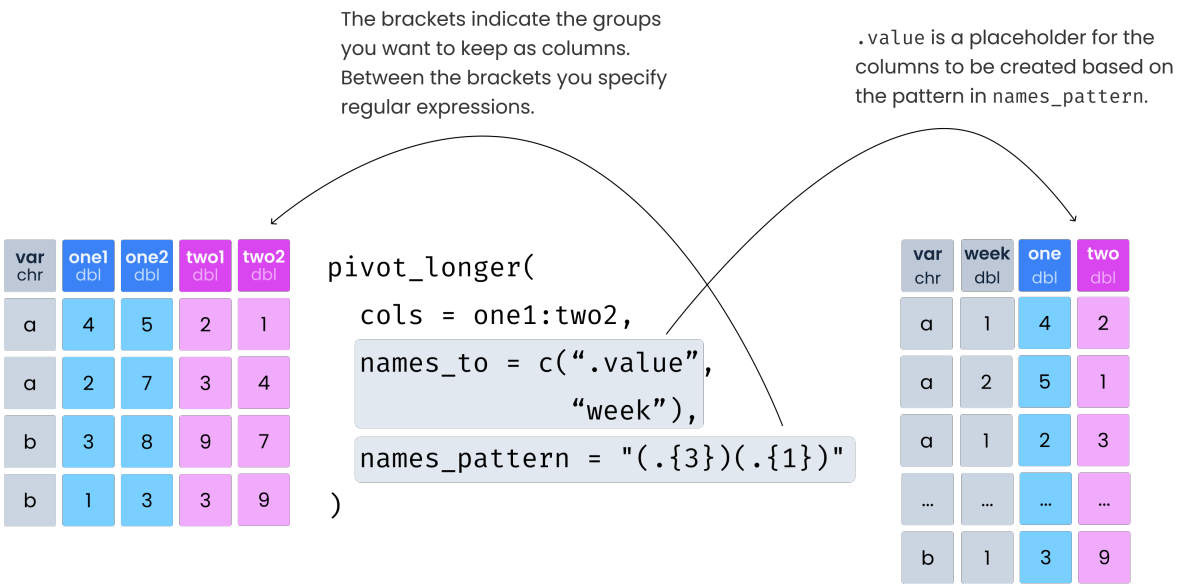
```
8   4  4  4 19  4.26 3.10  5.39 12.50
9  12 12 12  8 10.84 9.13  8.15  5.56
10  7  7  7  8  4.82 7.26  6.42  7.91
11  5  5  5  8  5.68 4.74  5.73  6.89
```

Once we've made this data frame longer, we can see what it's all about. I can tell you this much: x represents values on the x-axis and y represents values on the y-axis. In other words, the column names represent two variables, `x` and `y`. Applying our `pivot_longer` logic to this example would not work because we would not be able to capture these two columns:

```
anscombe %>%
  pivot_longer(
    cols = x1:y4,
    names_to = "axis",
    values_to = "value"
  )
```

```
# A tibble: 88 x 2
   axis  value
   <chr> <dbl>
 1 x1    10
 2 x2    10
 3 x3    10
 4 x4     8
 5 y1     8.04
 6 y2     9.14
 7 y3     7.46
 8 y4     6.58
 9 x1     8
10 x2     8
# ... with 78 more rows
```

To create two new variable columns in `names_to`, we need to use `.value` and the `names_pattern` parameter:



The most obscure element here is `.value`. To understand what `.value` does, let's first discuss `names_pattern`. `names_pattern` takes a regular expression. In the world of regular expressions, anything enclosed between parentheses is called a **group**. Groups allow us to capture parts of a string that belong together. In this example, the first group `(.{3})` contains the first three letters of a string. The second group `(.{1})` contains the fourth letter of this string. You can see that the length of the vector in `names_to` is equal to the number of groups defined in `names_pattern`. In other words, the elements in this vector represent the groups. If we look at the first group, we find two different elements: `one` and `two`. `.value` is a placeholder for these two elements. For each element, a new column is created with the text captured in the regular expression.

Now that we've seen how it works, let's tidy our data:

```
(anscombe_tidy <- anscombe %>%
  pivot_longer(
    cols = x1:y4,
    names_to = c(".value", "number"),
    names_pattern = "([xy])(\\d+)",
    values_to = "value"
  ))
```

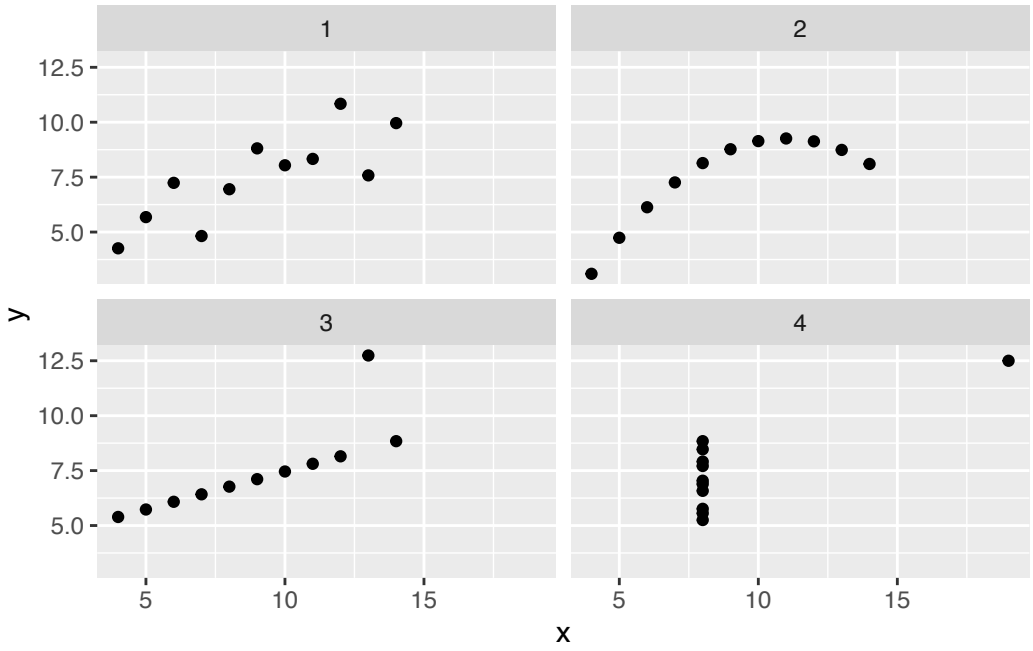
```
# A tibble: 44 x 3
   number     x     y
   <chr>  <dbl> <dbl>
 1 1         10  8.04
 2 2         10  9.14
```

```
 3 3         10  7.46
 4 4          8  6.58
 5 1          8  6.95
 6 2          8  8.14
 7 3          8  6.77
 8 4          8  5.76
 9 1         13  7.58
10 2         13  8.74
# ... with 34 more rows
```

The regular expression needs some explanation. Again, the regex has two groups: `"([xy])(\\d+)"`. The first group captures either the letter `x` or `y`: `([xy])`. The second group captures one or more numbers: `(\\d+)`.

Now that we have tidy data, we can visualize the idea behind the Anscombe dataset:

```
anscombe_tidy %>%
  ggplot(aes(x = x, y = y)) +
  geom_point() +
  facet_wrap(vars(number))
```



The data depicts four sets of data that have identical descriptive statistics (e.g., mean, standard deviation) but appear to be visually different from each other (see Anscombe Quartet).

16.3 Multiple variables are stored in one column

Let's look at another use case and example. The `who` dataset comes from the World Health Organization. It records confirmed tuberculosis cases broken down by country, year, and demographic group. The demographic groups are sex and age. Cases are broken down by four types: rel = relapse, sn = negative lung smear, sp = positive lung smear, ep = extrapulmonary.

```
who
```

```
# A tibble: 7,240 x 60
   country     iso2  iso3   year new_s~1 new_s~2 new_s~3 new_s~4 new_s~5 new_s~6
   <chr>       <chr> <chr> <int>   <int>   <int>   <int>   <int>   <int>   <int>
 1 Afghanistan AF    AFG    1980      NA      NA      NA      NA      NA      NA
 2 Afghanistan AF    AFG    1981      NA      NA      NA      NA      NA      NA
 3 Afghanistan AF    AFG    1982      NA      NA      NA      NA      NA      NA
 4 Afghanistan AF    AFG    1983      NA      NA      NA      NA      NA      NA
 5 Afghanistan AF    AFG    1984      NA      NA      NA      NA      NA      NA
 6 Afghanistan AF    AFG    1985      NA      NA      NA      NA      NA      NA
 7 Afghanistan AF    AFG    1986      NA      NA      NA      NA      NA      NA
 8 Afghanistan AF    AFG    1987      NA      NA      NA      NA      NA      NA
 9 Afghanistan AF    AFG    1988      NA      NA      NA      NA      NA      NA
10 Afghanistan AF    AFG    1989      NA      NA      NA      NA      NA      NA
# ... with 7,230 more rows, 50 more variables: new_sp_m65 <int>,
#   new_sp_f014 <int>, new_sp_f1524 <int>, new_sp_f2534 <int>,
#   new_sp_f3544 <int>, new_sp_f4554 <int>, new_sp_f5564 <int>,
#   new_sp_f65 <int>, new_sn_m014 <int>, new_sn_m1524 <int>,
#   new_sn_m2534 <int>, new_sn_m3544 <int>, new_sn_m4554 <int>,
#   new_sn_m5564 <int>, new_sn_m65 <int>, new_sn_f014 <int>,
#   new_sn_f1524 <int>, new_sn_f2534 <int>, new_sn_f3544 <int>, ...
```

```
who %>% colnames
```

```
 [1] "country"      "iso2"         "iso3"         "year"         "new_sp_m014"
 [6] "new_sp_m1524" "new_sp_m2534" "new_sp_m3544" "new_sp_m4554" "new_sp_m5564"
[11] "new_sp_m65"   "new_sp_f014"  "new_sp_f1524" "new_sp_f2534" "new_sp_f3544"
[16] "new_sp_f4554" "new_sp_f5564" "new_sp_f65"   "new_sn_m014"  "new_sn_m1524"
[21] "new_sn_m2534" "new_sn_m3544" "new_sn_m4554" "new_sn_m5564" "new_sn_m65"
[26] "new_sn_f014"  "new_sn_f1524" "new_sn_f2534" "new_sn_f3544" "new_sn_f4554"
[31] "new_sn_f5564" "new_sn_f65"   "new_ep_m014"  "new_ep_m1524" "new_ep_m2534"
[36] "new_ep_m3544" "new_ep_m4554" "new_ep_m5564" "new_ep_m65"   "new_ep_f014"
```

```
[41] "new_ep_f1524" "new_ep_f2534" "new_ep_f3544" "new_ep_f4554" "new_ep_f5564"
[46] "new_ep_f65"   "newrel_m014"  "newrel_m1524" "newrel_m2534" "newrel_m3544"
[51] "newrel_m4554" "newrel_m5564" "newrel_m65"   "newrel_f014"  "newrel_f1524"
[56] "newrel_f2534" "newrel_f3544" "newrel_f4554" "newrel_f5564" "newrel_f65"
```

In this data frame, the underlying variables `type`, `cases`, `age` and `gender` are contained in the column headings. Let us take the column `new_sp_m1524`. `sp` stands for the type that has a positive lung smear. `m` stands for male and `1524` stands for the age group from 15 to 24 years.

Another problem with this data frame is that the column names are not well formatted. In some columns `new` is followed by an underscore `new_sp_m3544`, in others not, `newrel_m2534`.

Here you can see how this data frame can be tidied with `pivot_longer`. We will break it down further next.

```
(who_cleaned <- who %>%
  pivot_longer(
    cols = new_sp_m014:newrel_f65,
    names_pattern = "new_?([a-z]{2,3})_([a-z])(\\d+)",
    names_to = c("type", "sex", "age"),
    values_to = "cases"
  ) %>%
  mutate(
    age = case_when(
      str_length(age) == 2 ~ age,
      str_length(age) == 3 ~ str_replace(age, "(^.)", "\\1-"),
      str_length(age) == 4 ~ str_replace(age, "(^.{2})", "\\1-"),
      TRUE ~ age
    )
  )
 )
```

```
# A tibble: 405,440 x 8
   country     iso2  iso3   year type  sex   age   cases
   <chr>       <chr> <chr> <int> <chr> <chr> <chr> <int>
 1 Afghanistan AF    AFG    1980 sp    m     0-14     NA
 2 Afghanistan AF    AFG    1980 sp    m     15-24    NA
 3 Afghanistan AF    AFG    1980 sp    m     25-34    NA
 4 Afghanistan AF    AFG    1980 sp    m     35-44    NA
 5 Afghanistan AF    AFG    1980 sp    m     45-54    NA
 6 Afghanistan AF    AFG    1980 sp    m     55-64    NA
 7 Afghanistan AF    AFG    1980 sp    m     65       NA
```

```
 8 Afghanistan AF    AFG    1980 sp    f     0-14     NA
 9 Afghanistan AF    AFG    1980 sp    f     15-24    NA
10 Afghanistan AF    AFG    1980 sp    f     25-34    NA
# ... with 405,430 more rows
```

Let's start with `pivot_longer`. The main difference from our previous example is that the regular expression for `names_pattern` is more complex. The regular expression captures three groups. Each group is converted into a new column.

- The first group `([a-z]{2,3})` is converted into a column representing the type of case
- The second group `([a-z])` is converted to the gender column
- The third group `(\\d+)` is translated into the column `age`

See also how we solved the problem with the underscore in the regular expression by making it optional `new_?`.

```
(who_tidy_first_step <- who %>%
  pivot_longer(
    cols = new_sp_m014:newrel_f65,
    names_pattern = "new_?([a-z]{2,3})_([a-z])(\\d+)",
    names_to = c("type", "sex", "age"),
    values_to = "cases"
  ))
```

```
# A tibble: 405,440 x 8
   country     iso2  iso3   year type  sex   age   cases
   <chr>       <chr> <chr> <int> <chr> <chr> <chr> <int>
 1 Afghanistan AF    AFG    1980 sp    m     014      NA
 2 Afghanistan AF    AFG    1980 sp    m     1524     NA
 3 Afghanistan AF    AFG    1980 sp    m     2534     NA
 4 Afghanistan AF    AFG    1980 sp    m     3544     NA
 5 Afghanistan AF    AFG    1980 sp    m     4554     NA
 6 Afghanistan AF    AFG    1980 sp    m     5564     NA
 7 Afghanistan AF    AFG    1980 sp    m     65       NA
 8 Afghanistan AF    AFG    1980 sp    f     014      NA
 9 Afghanistan AF    AFG    1980 sp    f     1524     NA
10 Afghanistan AF    AFG    1980 sp    f     2534     NA
# ... with 405,430 more rows
```

Next, we need to clean up the `age` column. It should be cleaned as follows:

- `014` -> `0-14`

- 1524 -> 15-24
- 65 -> 65

We can do this conversion with `mutate` and `case_when`:

```
(who_tidy_second_step <- who_tidy_first_step %>%
   mutate(
    age = case_when(
      str_length(age) == 2 ~ age,
      str_length(age) == 3 ~ str_replace(age, "(^.)", "\\1-"),
      str_length(age) == 4 ~ str_replace(age, "(^.{2})", "\\1-"),
      TRUE ~ age
    )
  ))
```

```
# A tibble: 405,440 x 8
   country     iso2  iso3   year type  sex   age   cases
   <chr>       <chr> <chr> <int> <chr> <chr> <chr> <int>
 1 Afghanistan AF    AFG    1980 sp    m     0-14     NA
 2 Afghanistan AF    AFG    1980 sp    m     15-24    NA
 3 Afghanistan AF    AFG    1980 sp    m     25-34    NA
 4 Afghanistan AF    AFG    1980 sp    m     35-44    NA
 5 Afghanistan AF    AFG    1980 sp    m     45-54    NA
 6 Afghanistan AF    AFG    1980 sp    m     55-64    NA
 7 Afghanistan AF    AFG    1980 sp    m     65       NA
 8 Afghanistan AF    AFG    1980 sp    f     0-14     NA
 9 Afghanistan AF    AFG    1980 sp    f     15-24    NA
10 Afghanistan AF    AFG    1980 sp    f     25-34    NA
# ... with 405,430 more rows
```

You can see that the data frame is quite large (405,440 rows and 8 columns). However, most of the values are `NA`. Fortunately, we can easily remove these `NA`s with the `values_drop_na` parameter by setting it to `TRUE`:

```
(who_cleaned_small <- who %>%
  pivot_longer(
    cols = new_sp_m014:newrel_f65,
    names_pattern = "new_?([a-z]{2,3})_([a-z])(\\d+)",
    names_to = c("type", "sex", "age"),
    values_to = "cases",
    values_drop_na = TRUE
  ) %>%
```

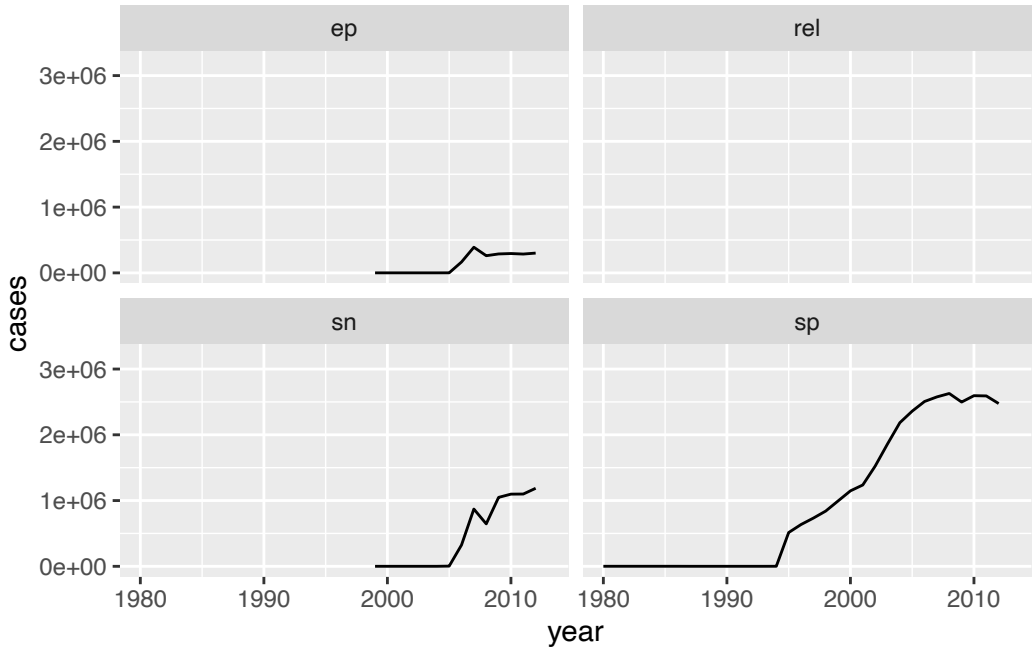
```
    mutate(
      age = case_when(
        str_length(age) == 2 ~ age,
        str_length(age) == 3 ~ str_replace(age, "(^.)", "\\1-"),
        str_length(age) == 4 ~ str_replace(age, "(^.{2})", "\\1-"),
        TRUE ~ age
      )
    ))
```

```
# A tibble: 76,046 x 8
   country     iso2  iso3   year type  sex   age   cases
   <chr>       <chr> <chr> <int> <chr> <chr> <chr> <int>
 1 Afghanistan AF    AFG    1997 sp    m     0-14      0
 2 Afghanistan AF    AFG    1997 sp    m     15-24    10
 3 Afghanistan AF    AFG    1997 sp    m     25-34     6
 4 Afghanistan AF    AFG    1997 sp    m     35-44     3
 5 Afghanistan AF    AFG    1997 sp    m     45-54     5
 6 Afghanistan AF    AFG    1997 sp    m     55-64     2
 7 Afghanistan AF    AFG    1997 sp    m     65        0
 8 Afghanistan AF    AFG    1997 sp    f     0-14      5
 9 Afghanistan AF    AFG    1997 sp    f     15-24    38
10 Afghanistan AF    AFG    1997 sp    f     25-34    36
# ... with 76,036 more rows
```

This data frame has only 76,046 lines. A reduction of 81%.

Now that we have this data frame, we can track cases over time:

```
who_cleaned_small %>%
  ggplot(aes(x = year, y = cases)) +
  stat_summary(
    fun = sum,
    geom = "line"
  ) +
  facet_wrap(vars(type))
```



16.4 Variables are stored in both rows and columns

In our last use case, an underlying variable is stored in both columns and rows. Consider this data frame:

```
weather_data <- tribble(
  ~id,          ~year,   ~month,  ~element,  ~d1, ~d2,     ~d3, ~d4,   ~d5, ~d6,
  "MX17004",    2010,        1,   "tmax",    NA,   NA,      NA,  NA,    NA,  NA,
  "MX17004",    2010,        1,   "tmin",    NA,   NA,      NA,  NA,    NA,  NA,
  "MX17004",    2010,        2,   "tmax",    NA,   27.3,  24.1,  NA,    NA,  NA,
  "MX17004",    2010,        2,   "tmin",    NA,   14.4,  14.4,  NA,    NA,  NA,
  "MX17004",    2010,        3,   "tmax",    NA,   NA,      NA,  NA,  32.1,  NA,
  "MX17004",    2010,        3,   "tmin",    NA,   NA,      NA,  NA,  14.2,  NA,
  "MX17004",    2010,        4,   "tmax",    NA,   NA,      NA,  NA,    NA,  NA,
  "MX17004",    2010,        4,   "tmin",    NA,   NA,      NA,  NA,    NA,  NA,
  "MX17004",    2010,        5,   "tmax",    NA,   NA,      NA,  NA,    NA,  NA,
  "MX17004",    2010,        5,   "tmin",    NA,   NA,      NA,  NA,    NA,  NA,
)  %>%
  mutate(across(d1:d6, as.numeric))
```

This data frame shows temperature data from a weather station in Mexico (see Wickham, 2014, p. 10f). Minimum and maximum temperatures were recorded daily. The columns `d1` through

`d31` (some omitted here for readability) represent days, and the `year` and `month` columns represent the year and month, respectively.

What is striking about this data frame is that the underlying variable `date` is spread across rows and columns. Since the `date` variable should be in one column, this data is not tidy.

To solve this problem, we need to do two things. First, we need to make the data frame longer. Second, we need to create the `date` column. Here is how we could do this:

```
weather_data_cleaned <- weather_data %>%
  pivot_longer(
    cols = d1:d6,
    names_to = "day",
    names_prefix = "d",
    values_to = "value",
    values_drop_na = TRUE
  ) %>%
  unite(
    col = date,
    year, month, day,
    sep = "-"
  ) %>%
  mutate(
    date = as.Date(date, format = "%Y-%m-%d")
  ) %>% print()
```

```
# A tibble: 6 x 4
  id      date       element value
  <chr>   <date>     <chr>   <dbl>
1 MX17004 2010-02-02 tmax     27.3
2 MX17004 2010-02-03 tmax     24.1
3 MX17004 2010-02-02 tmin     14.4
4 MX17004 2010-02-03 tmin     14.4
5 MX17004 2010-03-05 tmax     32.1
6 MX17004 2010-03-05 tmin     14.2
```

The parameters in `pivot_longer` should already be familiar. We make the data frame longer with the columns `d1` to `d6`. We remove the prefix from these column values with `names_prefix` and we drop rows containing `NA`s.

The second step is to create the `date` column.

Summary

- Un-tidy data is usually the result of working with spreadsheet programs optimized for data entry
- To make messy data tidy, we usually have to lengthen it with `pivot_longer`.
- When we make data frames longer, we increase the number of rows and decrease the number of columns. We also increase the number of values in the data frame.
- There are four common use cases when making data frames longer: Column headers are values of one variable, not variable names, multiple variables are stored in columns, multiple variables are stored in one column, variables are stored in both rows and columns, variables are stored in both rows and column.
- If your column names contain more than one variable, you need to use the parameters `names_pattern` and `.value` in `names_to`.