

Week 6: Making Untidy Data Tidy

Ellen Bledsoe

2025-09-30

Making Untidy Data Tidy

One major task of working with data (especially other people's data!) is to tidy up the dataset so that we can successfully calculate our descriptive statistics with `group_by` and `summarize`, make our data visualizations with `ggplot2`, and eventually conduct our statistical analyses.

This week's lessons are all about ways to make untidy data tidy. First, let's remind ourselves what our "tidy" data means.

“**TIDY DATA** is a standard way of mapping the meaning of a dataset to its structure.”
—HADLEY WICKHAM

In tidy data:

- each variable forms a column
- each observation forms a row
- each cell is a single measurement

each column a variable

id	name	color
1	floof	gray
2	max	black
3	cat	orange
4	donut	gray
5	merlin	black
6	panda	calico

each row an observation

Wickham, H. (2014). Tidy Data. Journal of Statistical Software 59 (10). DOI: 10.18637/jss.v059.i10

Unfortunately, a lot of existing data doesn't follow these rules. The way in which datasets are untidy are always unique.

However, to analyze the data, we typically need data to be in a tidy format. We can use a number of functions from the `tidyr` package in the `tidyverse` to help make the data tidy.

Set-up

Packages

First, let's load the `tidyverse`.

```
library(tidyverse)
```

```
## -- Attaching core tidyverse packages ----- tidyverse 2.0.0 --
## v dplyr      1.1.4      v readr      2.1.5
## v forcats    1.0.0      v stringr    1.5.2
## v ggplot2    4.0.0      v tibble     3.3.0
## v lubridate  1.9.4      v tidyr      1.3.1
```

```
## v purrr      1.1.0
## -- Conflicts ----- tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()   masks stats::lag()
## i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to become errors
```

File Paths

Now, let's read in our data. In this lesson, we are going to start incorporating a little bit of information about file paths into our work.

So far, our data files have been in the same place ("working directory" or folder) as our Quarto files. Now, however, we have our data in a data folder.

In order to tell R which data file we want to use (or where to save a data file we want to download), we have to point R to the exact location, which is no longer the same place as where our Quarto file is stored.

To do so, we must tell our to first look in the `data` folder and *then* tell it which file we want to read in.

The data file we are using is called `macroplot_data.txt`. When we open the file and take a look at it, we can see that it is tab-delimited, meaning we will want to use the `read_tsv()` function.

```
macroplots <- read_tsv("data/macroplot_data.txt")
```

```
## Rows: 61965 Columns: 7
## -- Column specification -----
## Delimiter: "\t"
## chr (2): Plot ID, Sp Code
## dbl (5): TreeGirth1, TreeGirth2, TreeGirth3, TreeGirth4, TreeGirth5
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
# view the dataframe
```

```
macroplots
```

```
## # A tibble: 61,965 x 7
##   `Plot ID` `Sp Code` TreeGirth1 TreeGirth2 TreeGirth3 TreeGirth4 TreeGirth5
##   <chr>     <chr>      <dbl>      <dbl>      <dbl>      <dbl>      <dbl>
## 1 BSP70    Acaccaes         25         0         0         0         0
## 2 BSP70    <NA>            25         0         0         0         0
## 3 BSP70    <NA>            28         0         0         0         0
## 4 BSP70    <NA>            38         0         0         0         0
## 5 BSP70    <NA>            54         0         0         0         0
## 6 BSP100   Acaccate         12         0         0         0         0
## 7 BSP100   <NA>             13         0         0         0         0
## 8 BSP100   <NA>             14         0         0         0         0
## 9 BSP100   <NA>             15         0         0         0         0
## 10 BSP100  <NA>             16         0         0         0         0
## # i 61,955 more rows
```

Data Structure

Let's take a look. These data are tree girths from the Western Ghats, a mountainous region on the western edge of the Indian peninsula. This area is considered one of the top biodiversity hotspots in the world.

Any time that a tree had more than one stem (trunk), the diameter of each stem got entered into a new column.

Is this good data structure? Why or why not? What do we need to fix?

Before we get started, let's add a `treeid` column to our data frame using the `mutate` function from `dplyr`. We want one `treeid` for each row because there is one tree for each row. We can use the `.before` argument to determine where the new column is located in the data frame.

Because the `Plot ID` column has a space in its name, we can't reference it as we normally would. Instead, we can put backticks around it.

```
macroplots <- macroplots %>%  
  mutate(TreeID = 1:n(), .before = `Plot ID`)  
  # use `backticks` around Plot ID because of the space  
  # the .before argument determines where the new column will be placed
```

```
macroplots
```

```
## # A tibble: 61,965 x 8  
##   TreeID `Plot ID` `Sp Code` TreeGirth1 TreeGirth2 TreeGirth3 TreeGirth4  
##   <int> <chr>    <chr>      <dbl>      <dbl>      <dbl>      <dbl>  
## 1     1 BSP70    Acaccaes      25         0         0         0  
## 2     2 BSP70    <NA>         25         0         0         0  
## 3     3 BSP70    <NA>         28         0         0         0  
## 4     4 BSP70    <NA>         38         0         0         0  
## 5     5 BSP70    <NA>         54         0         0         0  
## 6     6 BSP100   Acaccate     12         0         0         0  
## 7     7 BSP100   <NA>         13         0         0         0  
## 8     8 BSP100   <NA>         14         0         0         0  
## 9     9 BSP100   <NA>         15         0         0         0  
## 10    10 BSP100   <NA>         16         0         0         0  
## # i 61,955 more rows  
## # i 1 more variable: TreeGirth5 <dbl>
```

```
tail(macroplots)
```

```
## # A tibble: 6 x 8  
##   TreeID `Plot ID` `Sp Code` TreeGirth1 TreeGirth2 TreeGirth3 TreeGirth4  
##   <int> <chr>    <chr>      <dbl>      <dbl>      <dbl>      <dbl>  
## 1 61960 BSP8    <NA>         12         0         0         0  
## 2 61961 BSP8    <NA>         16         0         0         0  
## 3 61962 BSP8    <NA>         32         24         21         0  
## 4 61963 BSP8    <NA>         33         0         0         0  
## 5 61964 BSP8    <NA>         41         0         0         0  
## 6 61965 BSP8    <NA>         52         39         0         0  
## # i 1 more variable: TreeGirth5 <dbl>
```

Renaming Columns

After a while, putting backticks around column names is going to get tedious. Thankfully, the `tidyverse` has a helpful function for renaming columns.

The argument in `rename()` is the new name of the column equal to the old name.

```
macroplots <- macroplots %>%  
  rename(PlotID = `Plot ID`, SpCode = `Sp Code`)  
head(macroplots)
```

```
## # A tibble: 6 x 8  
##   TreeID PlotID SpCode   TreeGirth1 TreeGirth2 TreeGirth3 TreeGirth4 TreeGirth5
```

```
##      <int> <chr> <chr>          <dbl>      <dbl>      <dbl>      <dbl>      <dbl>
## 1         1 BSP70 Acaccaes         25         0         0         0         0
## 2         2 BSP70 <NA>          25         0         0         0         0
## 3         3 BSP70 <NA>          28         0         0         0         0
## 4         4 BSP70 <NA>          38         0         0         0         0
## 5         5 BSP70 <NA>          54         0         0         0         0
## 6         6 BSP100 Acaccate      12         0         0         0         0
```

That is much better!

Completing Data with Gaps

When recording data, people might write out a value once and then leave the following rows blank, assuming that all of the following rows are the same value until a new value is present.

This is human-readable but not computer-readable.

We can fill in these gaps using the `fill()` function. The default direction is “down,” but other directions are possible.

```
macroplots <- macroplots %>%
  fill(SpCode)
head(macroplots)
```

```
## # A tibble: 6 x 8
##   TreeID PlotID SpCode   TreeGirth1 TreeGirth2 TreeGirth3 TreeGirth4 TreeGirth5
##   <int> <chr> <chr>         <dbl>      <dbl>      <dbl>      <dbl>      <dbl>
## 1         1 BSP70 Acaccaes         25         0         0         0         0
## 2         2 BSP70 Acaccaes         25         0         0         0         0
## 3         3 BSP70 Acaccaes         28         0         0         0         0
## 4         4 BSP70 Acaccaes         38         0         0         0         0
## 5         5 BSP70 Acaccaes         54         0         0         0         0
## 6         6 BSP100 Acaccate      12         0         0         0         0
```

Wide Data vs. Long Data

Data are often structured in a way that doesn’t allow us to (easily) perform calculations or plot the data. We, therefore, need to wrangle data between “wide” format and “long format” to get the data in a structure that better serves our purposes.

One format isn’t necessary better than the other in terms of “tidy” data, although you would often hear people use tidy data and long format data interchangeably. There are some instances where the “longest” form of the data is not actually considered tidy data.

Here, we are going to focus on converting between the types using two functions from the `tidyr` package (part of the `tidyverse`, of course!): `pivot_longer()` and `pivot_wider()`.

Like with the `join` functions, visualizing how these functions work takes a bit of practice. For GIFs demonstrating them, head over to the trusty tidyexplain site.

Wide Data

In this example of wide data, the data table has one row per tree species with columns for each site.

Tree species	Site A	Site B	Site C	Site D
<i>Acer rubrum</i>	15	8	30	27
<i>Quercus alba</i>	29	17	14	42
<i>Pinus taeda</i>	10	19	25	23

Long Data

In this example of long data, the data table has one row per tree species per site, and columns represent unique variables. In this example, the long format is considered tidy data.

Tree species	Site	DBH (cm)
<i>Acer rubrum</i>	A	15
<i>Acer rubrum</i>	B	8
<i>Acer rubrum</i>	C	30
<i>Acer rubrum</i>	D	27
<i>Quercus alba</i>	A	29
<i>Quercus alba</i>	B	17
<i>Quercus alba</i>	C	14
<i>Quercus alba</i>	D	42
<i>Pinus taeda</i>	A	10
<i>Pinus taeda</i>	B	19
<i>Pinus taeda</i>	C	25
<i>Pinus taeda</i>	D	23

One common issue with wide data is that data is spread over multiple columns that should be in one. For example, in the tables above, the “Sites” as column names are violating the rule of not having data in column names. Instead, that data should be in one column, like in the long data.

To get the data in this form, we can use a function from `tidyr` called `pivot_longer`.

Pivot Longer

The `pivot_longer()` function takes the following arguments (as well as many other optional arguments). This function was previously called `spread()`, which you might still see in other people’s code sometimes.

- the data frame
- columns to include (or not include)
- `names_to`: the name of the new column to put the previous column names in
- `values_to`: the name of the new column to put the values from the cells in

Any redundant or unnecessary columns will be automatically removed.

```
macro_long <- macroplots %>%
  pivot_longer(TreeGirth1:TreeGirth5, # columns to include in the pivot
               names_to = "Stem",    # name of new column to hold previous column names
               values_to = "Girth")  # name of new column to hold values from cells
```

```
macro_long
```

```
## # A tibble: 309,825 x 5
```

```
##      TreeID PlotID SpCode   Stem      Girth
##      <int> <chr>  <chr>    <chr>    <dbl>
##  1         1 BSP70  Acaccaes TreeGirth1    25
##  2         1 BSP70  Acaccaes TreeGirth2     0
##  3         1 BSP70  Acaccaes TreeGirth3     0
##  4         1 BSP70  Acaccaes TreeGirth4     0
##  5         1 BSP70  Acaccaes TreeGirth5     0
##  6         2 BSP70  Acaccaes TreeGirth1    25
##  7         2 BSP70  Acaccaes TreeGirth2     0
##  8         2 BSP70  Acaccaes TreeGirth3     0
##  9         2 BSP70  Acaccaes TreeGirth4     0
## 10         2 BSP70  Acaccaes TreeGirth5     0
## # i 309,815 more rows
```

As a reminder, the colon specifies all columns starting at `TreeGirth1` and ending at `TreeGirth5`

Replace Values with NA

As you might have noticed, there are still 0s where there were no stems. We probably don't want those zeros to still be there. Instead, we might want them to be `NA` values.

To do so, we can use a function (from `dplyr`) called `na_if`. This function allows us to replace certain values with `NA`. We can use this function within a `mutate` function.

The arguments in `na_if` are the vector (column) you want the values replaced in and the value to be replaced.

```
macro_long <- macro_long %>%
  mutate(Girth = na_if(Girth, 0))

head(macro_long)
```

```
## # A tibble: 6 x 5
##   TreeID PlotID SpCode   Stem      Girth
##   <int> <chr>  <chr>    <chr>    <dbl>
## 1         1 BSP70  Acaccaes TreeGirth1    25
## 2         1 BSP70  Acaccaes TreeGirth2    NA
## 3         1 BSP70  Acaccaes TreeGirth3    NA
## 4         1 BSP70  Acaccaes TreeGirth4    NA
## 5         1 BSP70  Acaccaes TreeGirth5    NA
## 6         2 BSP70  Acaccaes TreeGirth1    25
```

Drop Rows with NA Values

Often, we might want to remove these rows altogether. To remove rows that have `NA` values in a specific column, we could use the `filter` function and `!is.na()`, as we have in the past. Alternatively, we could use the `drop_na()` function, which does the same thing.

```
macro_long <- macro_long %>%
  drop_na(Girth)

head(macro_long)
```

```
## # A tibble: 6 x 5
##   TreeID PlotID SpCode   Stem      Girth
##   <int> <chr>  <chr>    <chr>    <dbl>
## 1         1 BSP70  Acaccaes TreeGirth1    25
## 2         2 BSP70  Acaccaes TreeGirth1    25
## 3         3 BSP70  Acaccaes TreeGirth1    28
```

```
## 4      4 BSP70  Acaccaes TreeGirth1    38
## 5      5 BSP70  Acaccaes TreeGirth1    54
## 6      6 BSP100 Acaccate TreeGirth1    12
```

Let's Practice!

In the assignment, work on Question 1a and 1b.

`extract()` Values from a Column

There are a number of ways that we can work with character strings, and we will cover many of those more in depth in another lesson. For now, we will stick to one helpful function called `extract()`.

Let's say we want the `Stem` column to have only the number of the stem (1-5) rather than "TreeGirth1".

`extract()` will extract one or more values from a column. It uses something called "regular expressions." We might get into details about regular expression later, but we won't worry about them too much for now.

The arguments for `extract` are:

- the data frame
- the name of the column from which we want to extract something
- the name(s) of the new column(s)
- the regular expression specifying what to extract (we will talk more about regular expressions later)

```
macro_long %>%
  extract(Stem, "Stem", "TreeGirth(.)")
```

```
## # A tibble: 65,889 x 5
##   TreeID PlotID SpCode   Stem   Girth
##   <int> <chr>   <chr>   <chr> <dbl>
## 1      1    1 BSP70  Acaccaes 1      25
## 2      2    2 BSP70  Acaccaes 1      25
## 3      3    3 BSP70  Acaccaes 1      28
## 4      4    4 BSP70  Acaccaes 1      38
## 5      5    5 BSP70  Acaccaes 1      54
## 6      6    6 BSP100 Acaccate 1      12
## 7      7    7 BSP100 Acaccate 1      13
## 8      8    8 BSP100 Acaccate 1      14
## 9      9    9 BSP100 Acaccate 1      15
## 10    10   10 BSP100 Acaccate 1      16
## # i 65,879 more rows
```

```
head(macro_long)
```

```
## # A tibble: 6 x 5
##   TreeID PlotID SpCode   Stem   Girth
##   <int> <chr>   <chr>   <chr> <dbl>
## 1      1    1 BSP70  Acaccaes TreeGirth1    25
## 2      2    2 BSP70  Acaccaes TreeGirth1    25
## 3      3    3 BSP70  Acaccaes TreeGirth1    28
## 4      4    4 BSP70  Acaccaes TreeGirth1    38
## 5      5    5 BSP70  Acaccaes TreeGirth1    54
## 6      6    6 BSP100 Acaccate TreeGirth1    12
```

Here, `TreeGirth.` means the phrase "TreeGirth" followed by a single value. The `()` indicate what part of this string to extract, so just the number at the end of the string.

This gives us the result we want, with only the stem number in the `Stem` column.

You may notice that this number is on the left side of the column, not the right. That's because the number is still stored as a character, because it was extracted from a character string.

To convert it to its actual data class (numeric), we can add the optional argument `convert = TRUE` to `extract`.

```
macro_long <- macro_long %>%
  extract(col = Stem,           # column from which to extract values
         into = "Stem",        # name of column to put extracted values in
         regex = "TreeGirth(.)", # "regular expression" for extracting values
         convert = TRUE)       # convert extracted values to their data class

head(macro_long)
```

```
## # A tibble: 6 x 5
##   TreeID PlotID SpCode   Stem Girth
##   <int> <chr>  <chr>   <int> <dbl>
## 1     1  BSP70  Acaccaes     1    25
## 2     2  BSP70  Acaccaes     1    25
## 3     3  BSP70  Acaccaes     1    28
## 4     4  BSP70  Acaccaes     1    38
## 5     5  BSP70  Acaccaes     1    54
## 6     6  BSP100 Acaccate     1    12
```

Adding the `convert = TRUE` argument is a helpful addition when extracting numbers so you can then work with the column as numbers.

`separate()` a Column into Multiple Columns

In the `SpCode` column, the Genus and Species information for each tree are combined in a single column. If we want to group by genera, for example, having these data separated might be useful.

We can do so using the `separate()` function, which takes the following arguments:

- the data frame
- the name of the column to separate (`col`)
- new column names (`into`)
- the separator value, character, or position (if character, it must be in quotation marks, e.g., “-”; `sep`)

```
macro_long <- macro_long %>%
  separate(SpCode, c("Genus", "Species"), sep = 4)

macro_long
```

```
## # A tibble: 65,889 x 6
##   TreeID PlotID Genus Species   Stem Girth
##   <int> <chr>  <chr> <chr>   <int> <dbl>
## 1     1  BSP70  Acac  caes     1    25
## 2     2  BSP70  Acac  caes     1    25
## 3     3  BSP70  Acac  caes     1    28
## 4     4  BSP70  Acac  caes     1    38
## 5     5  BSP70  Acac  caes     1    54
## 6     6  BSP100 Acac  cate     1    12
## 7     7  BSP100 Acac  cate     1    13
## 8     8  BSP100 Acac  cate     1    14
## 9     9  BSP100 Acac  cate     1    15
## 10    10  BSP100 Acac  cate     1    16
## # i 65,879 more rows
```


if separated by a - or a space, put in quotation marks (e.g., "-")

Pivot Wider

Occasionally, we need to convert data the other way: from long to wide.

For example, this is fairly common for analyses of community-level data. Many packages will want cross-tab (or wide) data, such as a site by species matrix.

Let's demonstrate with an example calculating the number of stems per species. First, we need to group and summarize the data.

```
stem_counts <- macro_long %>%  
  group_by(PlotID, Genus, Species) %>%  
  summarize(Count = n()) |>  
  ungroup()
```

```
## `summarise()` has grouped output by 'PlotID', 'Genus'. You can override using  
## the `.groups` argument.
```

```
head(stem_counts)
```

```
## # A tibble: 6 x 4  
##   PlotID Genus Species Count  
##   <chr>  <chr> <chr>   <int>  
## 1 BSP1  Acac  sinu     20  
## 2 BSP1  Alan  salv      1  
## 3 BSP1  Albi  lebb      2  
## 4 BSP1  Albi  proc      2  
## 5 BSP1  Allo  cobb      6  
## 6 BSP1  Alse  seme     24
```

To get the site by species matrix, we need to make the dataset wider. Since the species ID will end up as column names, we need to bring the columns back together.

Let's Practice!

Work on Question 2 in the assignment.

unite Columns into One

To bring multiple columns together into one column, we use a function that does the opposite of `separate()`: `unite()`.

The `unite()` function takes the following arguments:

- the data frame
- the name of the new column to hold the united data (`col`); this should be written in quotation marks
- the columns to be united
- the separator value, character, or position (`sep`)

```
stem_counts <- stem_counts %>%  
  unite("SpeciesID", Genus, Species, sep = "_")  
head(stem_counts)
```

```
## # A tibble: 6 x 3  
##   PlotID SpeciesID Count  
##   <chr>  <chr>      <int>  
## 1 BSP1  Acac_sinu    20
```

```
## 2 BSP1 Alan_salv 1
## 3 BSP1 Albi_lebb 2
## 4 BSP1 Albi_proc 2
## 5 BSP1 Allo_cobb 6
## 6 BSP1 Alse_seme 24
```

Replacing Values

There is also one species code for unknown species. While in long format, we would likely convert this to an NA value, because it will become a column name, we probably want to make it something interpretable.

To do this, we can use a combination of the `mutate` function with the `replace` function. The `replace` function is part of the `base` package that comes build into R.

The `replace` function, when used inside the `mutate` function, the arguments are:

- the name of the column
- the condition that, if met, means the value will be replaced
- the value to use as the replacement

```
filter(stem_counts, SpeciesID == "UnID_")
```

```
## # A tibble: 24 x 3
##   PlotID SpeciesID Count
##   <chr>   <chr>    <int>
## 1 BSP1   UnID_      1
## 2 BSP12  UnID_      3
## 3 BSP15  UnID_      1
## 4 BSP16  UnID_      3
## 5 BSP18  UnID_      2
## 6 BSP22  UnID_      1
## 7 BSP24  UnID_      1
## 8 BSP25  UnID_      4
## 9 BSP27  UnID_     14
## 10 BSP29 UnID_      1
## # i 14 more rows
```

```
stem_counts <- stem_counts %>%
  mutate(SpeciesID = replace(SpeciesID, SpeciesID == "UnID_", "Unknown"))

filter(stem_counts, SpeciesID == "UnID_")
```

```
## # A tibble: 0 x 3
## # i 3 variables: PlotID <chr>, SpeciesID <chr>, Count <int>

filter(stem_counts, SpeciesID == "Unknown")
```

```
## # A tibble: 24 x 3
##   PlotID SpeciesID Count
##   <chr>   <chr>    <int>
## 1 BSP1   Unknown     1
## 2 BSP12  Unknown     3
## 3 BSP15  Unknown     1
## 4 BSP16  Unknown     3
## 5 BSP18  Unknown     2
## 6 BSP22  Unknown     1
## 7 BSP24  Unknown     1
## 8 BSP25  Unknown     4
```

```
## 9 BSP27 Unknown 14
## 10 BSP29 Unknown 1
## # i 14 more rows
```

Now we can go ahead and convert the data into a wider format.

`pivot_wider()`

Using `pivot_wider()` will spread values from one column into column names. It will then fill in the appropriate values. It has many arguments, but these are the required ones:

- the data frame
- the name of column to use for column names
- the name of column that contains the values to fill in the cells

```
stem_counts %>%
  pivot_wider(names_from = SpeciesID, values_from = Count)
```

```
## # A tibble: 96 x 400
##   PlotID Acac_sinu Alan_salv Albi_lebb Albi_proc Allo_cobb Alse_seme Apor_lind
##   <chr>      <int>      <int>      <int>      <int>      <int>      <int>      <int>
## 1 BSP1         20         1         2         2         6        24        82
## 2 BSP10        NA         NA         NA         NA        11         4        11
## 3 BSP100        1         NA         NA         NA         NA        NA        NA
## 4 BSP101        1        14         NA         NA         NA         2        NA
## 5 BSP102        NA         NA         NA         NA         NA        NA         1
## 6 BSP104        NA         NA         NA         NA         NA        NA        NA
## 7 BSP11         NA         NA         NA         NA        12        12       243
## 8 BSP12         NA         NA         NA         NA        36         5       164
## 9 BSP13         NA         NA         NA         NA         NA        10        31
## 10 BSP14        2         NA         NA         1         NA        NA        NA
## # i 86 more rows
## # i 392 more variables: Arto_hete <int>, Arto_hirs <int>, Bauh_mala <int>,
## #   Bauh_phoe <int>, Bomb_ceib <int>, Buch_lanz <int>, Bute_mono <int>,
## #   Call_tome <int>, Caly_flor <int>, Cant_dico <int>, Care_arbo <int>,
## #   Cari_cara <int>, Cary_uren <int>, Cass_fist <int>, Catu_dume <int>,
## #   Cinn_veru <int>, Dalb_horr <int>, Dalb_lati <int>, Dill_pent <int>,
## #   Dios_mela <int>, Dios_mont <int>, Dios_sylv <int>, Elae_kolo <int>, ...
```

Ok, that is a lot of NA values...

In this case, it would likely make sense to replace these NA values with 0. We can do this in a couple ways. With `pivot_wider()`, there is an optional argument to specify what value to use for any empty cells after the conversion.

```
stem_counts %>%
  pivot_wider(names_from = SpeciesID,
              values_from = Count,
              values_fill = 0)
```

```
## # A tibble: 96 x 400
##   PlotID Acac_sinu Alan_salv Albi_lebb Albi_proc Allo_cobb Alse_seme Apor_lind
##   <chr>      <int>      <int>      <int>      <int>      <int>      <int>      <int>
## 1 BSP1         20         1         2         2         6        24        82
## 2 BSP10         0         0         0         0        11         4        11
## 3 BSP100        1         0         0         0         0         0         0
## 4 BSP101        1        14         0         0         0         2         0
## 5 BSP102         0         0         0         0         0         0         1
```

```

## 6 BSP104      0      0      0      0      0      0      0
## 7 BSP11       0      0      0      0     12     12    243
## 8 BSP12       0      0      0      0     36      5    164
## 9 BSP13       0      0      0      0      0     10     31
## 10 BSP14      2      0      0      1      0      0      0
## # i 86 more rows
## # i 392 more variables: Arto_hete <int>, Arto_hirs <int>, Bauh_mala <int>,
## #   Bauh_phoe <int>, Bomb_ceib <int>, Buch_lanz <int>, Bute_mono <int>,
## #   Call_tome <int>, Caly_flor <int>, Cant_dico <int>, Care_arbo <int>,
## #   Cari_cara <int>, Cary_uren <int>, Cass_fist <int>, Catu_dume <int>,
## #   Cinn_veru <int>, Dalb_horr <int>, Dalb_lati <int>, Dill_pent <int>,
## #   Dios_mela <int>, Dios_mont <int>, Dios_sylv <int>, Elae_kolo <int>, ...

```

You should now be able to complete the assignment!