Week 6: Making Untidy Data Tidy

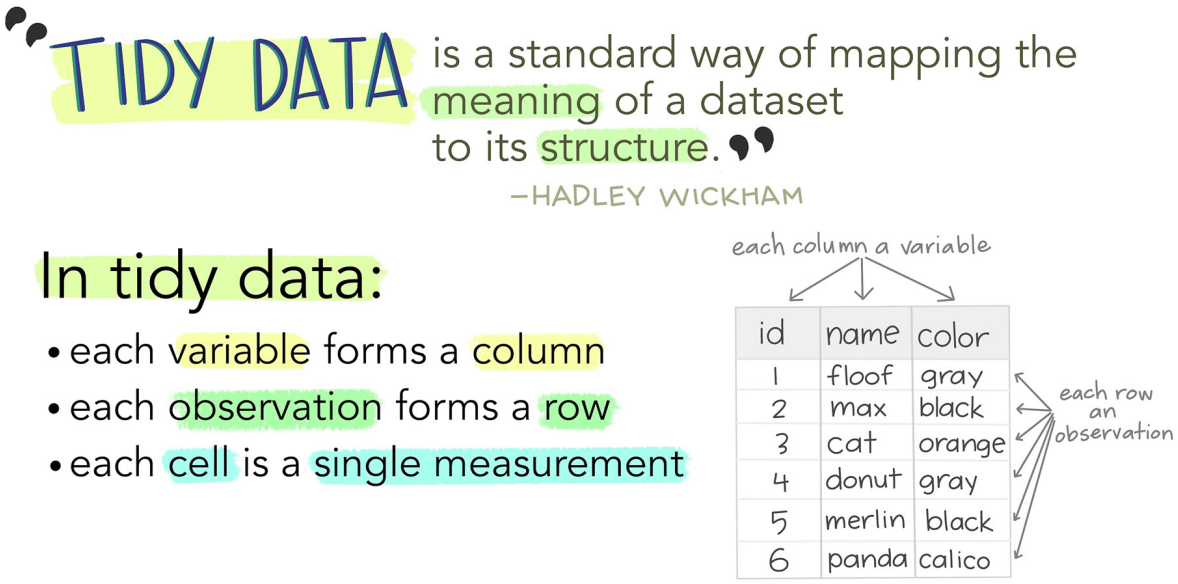
Ellen Bledsoe

2024-02-19

Making Untidy Data Tidy

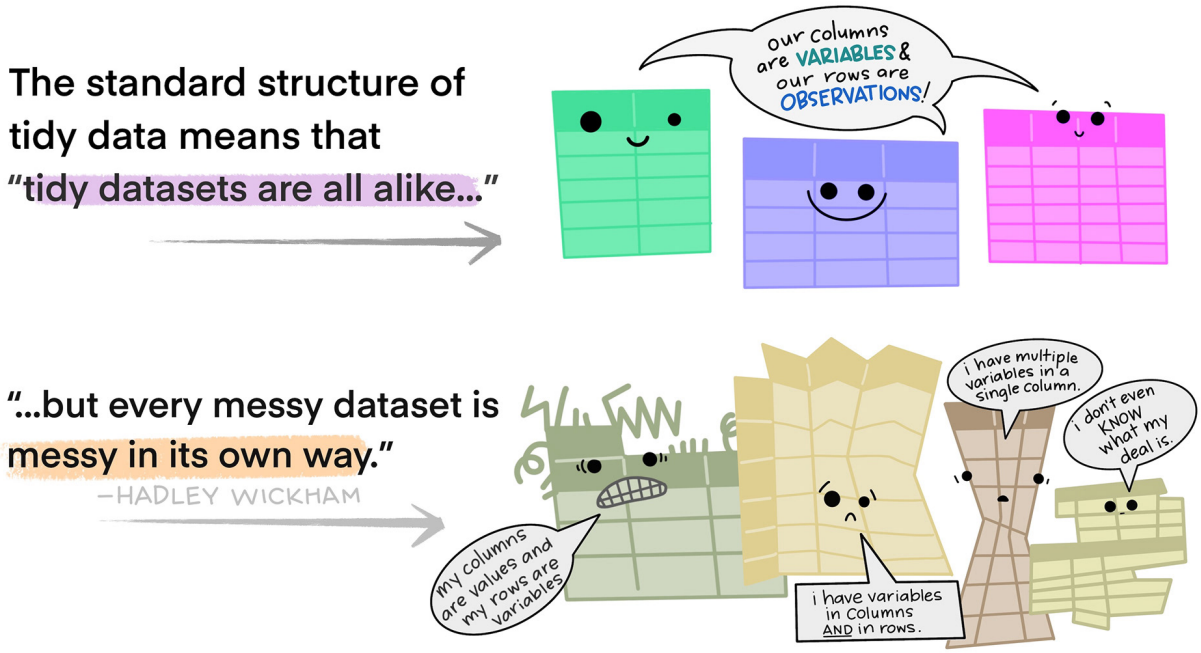
This week's lessons are about how to take untidy data tidy.

First, let's remind ourselves what out "tidy" data means.



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

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   1.5.0
## v ggplot2   3.4.2     v tibble    3.2.1
## v lubridate 1.9.2     v tidyr     1.3.0
## v purrr     1.0.1
## -- 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
```

Now, let's read in our data for this lesson.

```
macroplots <- read_tsv("http://datacarpentry.org/semester-biology/data/Macroplot_data_Rev.txt")
```

```
## Rows: 61965 Columns: 7
## -- Column specification --------------------------------------------------------
## Delimiter: "\t"
## chr (2): PlotID, SpCode
## 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.
```

```
head(macroplots)
```

```
## # A tibble: 6 x 7
##   PlotID SpCode   TreeGirth1 TreeGirth2 TreeGirth3 TreeGirth4 TreeGirth5
##   <chr>  <chr>         <dbl>      <dbl>      <dbl>      <dbl>      <dbl>
## 1 BSP70  Acaccaes         25          0          0          0          0
## 2 BSP70  Acaccaes         25          0          0          0          0
## 3 BSP70  Acaccaes         28          0          0          0          0
## 4 BSP70  Acaccaes         38          0          0          0          0
## 5 BSP70  Acaccaes         54          0          0          0          0
## 6 BSP100 Acaccate         12          0          0          0          0
```

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?

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

```
macroplots <- macroplots %>%
  mutate(TreeID = 1:n(), .before = PlotID)
  # the .before argument determines where the new column will be placed

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

One of the key ways that we need to wrangle data is between "wide" format and "long format."

Wide Data

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

Long Data

Tree species	Site	DBH (cm)
Acer rubrum	A	15
Acer rubrum	B	8
Acer rubrum	C	30
Acer rubrum	D	27
Quercus alba	A	29
Quercus alba	B	17
Quercus alba	C	14
Quercus alba	D	42
Pinus taeda	A	10
Pinus taeda	B	19
Pinus taeda	C	25
Pinus taeda	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 we as many other optional arguments).

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

Any redundant or unnecessary columns will be automatically removed.

```
macro_long <- macroplots %>%
  pivot_longer(TreeGirth1:TreeGirth5,
               names_to = "Stem",
               values_to = "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      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
```

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

Replace Values with NAs

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
```

`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

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

```
## # A tibble: 65,889 x 5
##    TreeID PlotID SpCode   Stem  Girth
##     <int> <chr>  <chr>    <chr> <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
##  7      7 BSP100 Acaccate 1        13
##  8      8 BSP100 Acaccate 1        14
##  9      9 BSP100 Acaccate 1        15
## 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 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
```

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 just 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 it's actual type (numeric), we can add the optional argument `convert = TRUE` to `extract`.

```
macro_long <- macro_long %>%
  extract(Stem, "Stem", "TreeGirth(.)", convert = TRUE)

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
- new column names
- the separator value, character, or position

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

head(macro_long)
```

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

```
# 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())
```

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

```
head(stem_counts)
```

```
## # A tibble: 6 x 4
## # Groups:   PlotID, Genus [5]
##   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 x 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.

`unite` Columns into One

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

We specify the data, the name of the new column, and the columns to combine.

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

```
## # A tibble: 6 x 3
## # Groups:   PlotID [1]
##   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
```

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
## # Groups:   PlotID [24]
##    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
## # Groups:   PlotID [0]
## # i 3 variables: PlotID <chr>, SpeciesID <chr>, Count <int>
```

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

```
## # A tibble: 24 x 3
## # Groups:   PlotID [24]
##    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
## # Groups:   PlotID [96]
##    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
## # Groups:   PlotID [96]
##    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>, ...
```

Completing Data with Gaps

In some datasets, 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.

```
gappy_data <- read.csv("http://www.datacarpentry.org/semester-biology/data/gappy-data.csv",
                       na.strings = "")
gappy_data
```

```
##    Species Individual Mass
## 1     acac          1   28
## 2     <NA>          2   26
## 3     <NA>          3   13
## 4     <NA>          4   33
## 5     <NA>          5   46
## 6     <NA>          6   40
## 7     <NA>          7   27
## 8     <NA>          8   42
## 9     <NA>          9   26
## 10    <NA>         10   22
## 11    fomo          1   13
## 12    <NA>          2   15
## 13    <NA>          3   12
## 14    <NA>          4    9
## 15    <NA>          5   20
## 16    <NA>          6   18
## 17    <NA>          7   16
## 18    <NA>          8   17
## 19    <NA>          9   12
## 20    <NA>         10    9
## 21    fala          1   52
## 22    <NA>          2   48
## 23    <NA>          3   48
## 24    <NA>          4   62
## 25    <NA>          5   35
## 26    <NA>          6   40
## 27    <NA>          7   42
## 28    <NA>          8   55
```

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

```
complete_data <- gappy_data %>%
  fill(Species)
head(complete_data)
```

```
##   Species Individual Mass
## 1    acac          1   28
## 2    acac          2   26
## 3    acac          3   13
## 4    acac          4   33
## 5    acac          5   46
## 6    acac          6   40
```

Renaming Columns

Last but very much not least, we often need to change the names of columns.

Let's read in a new .csv file with some less-than-desirable column names.

```
rodents <- read_csv("https://raw.githubusercontent.com/bleds22e/santa-cruz-rodents/main/data/capture_ra
```

```
## Rows: 51 Columns: 15
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr  (11): Site, Trap ID, Species, Status (R/N), Sex, Total Weight, Tail len...
## dbl   (3): Bag weight, Animal Weight, Hind foot length
## date  (1): Date
##
## 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.
```

```
colnames(rodents)
```

```
##  [1] "Date"              "Site"              "Trap ID"
##  [4] "Species"           "Status (R/N)"      "Sex"
##  [7] "Total Weight"      "Bag weight"        "Animal Weight"
## [10] "Hind foot length"  "Tail length"       "Hair sample (Y/N)"
## [13] "Position (R/L)"    "Handler"           "Notes"
```

We can select a few columns: date, site, and trap ID.

```
rodents %>% select(Date, Site, Trap ID)
```

Well, that doesn't work. How do we get around it? We use backticks to deal with names with spaces or special charaters.

```
rodents %>% select(Date, Site, `Trap ID`)
```

```
## # A tibble: 51 x 3
##    Date       Site     `Trap ID`
##    <date>     <chr>    <chr>
##  1 2022-11-14 Heritage 4C
##  2 2022-11-14 Heritage 4D
##  3 2022-11-14 Heritage 4I
##  4 2022-11-14 Heritage 2H
##  5 2022-11-14 Heritage 4J
##  6 2022-11-14 Heritage 2F
##  7 2022-11-15 Heritage 4C
##  8 2022-11-15 Heritage 4H
##  9 2022-11-15 Heritage 1H
## 10 2022-11-15 Heritage 1B
## # i 41 more rows
```

While function, this get gets annoying after a while. 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.

```
rodents <- rodents %>%
  rename(TrapID = `Trap ID`)
```