Data manipulation tutorial

Experiential Data science for Undergraduate Cross-disciplinary Education

Dr. Kim Dill-McFarland, U. of British Columbia

Contents

Data manipulation tutorial

Learning objectives

- Load tabular data using the tidyverse
- Subset and clean data in `dplyr` (filter, select, rename, arrange, mutate)
- Summarize data in `dplyr` (group_by, summarize)
- Transform data frames using `tidyr` (gather, spread) and `dplyr` (*_join)
- Link multiple tidyverse functions using pipes `%>%`

Setup

If you would like to follow along, open a new RStudio session and create a Project. Download the data file `Saanich_Data.csv` using the following command in RStudio. If you would like to save code/notes, also start a new R script.

```
write.csv(
  read.csv("https://raw.githubusercontent.com/EDUCE-UBC/workshop_data/master/Saanich_Data.csv"),
  "Saanich_Data.csv", row.names=FALSE)
```

Not sure what a Project is? Be sure to include the "RStudio tutorial" in your materials!

If you would like to learn more about Saanich Inlet and these data, checkout our description.

The R tidyverse

Base R gets the job done when it comes to data manipulation, but it is computationally slow and easily becomes difficult to read. For example, this code take data in `dat` and selects several variables (columns) as well as filters to rows where oxygen data is present (not NA).

```
dat[apply(!is.na(dat[,"WS_O2"]), 1, any),
        c("Cruise", "Date", "Depth", "WS_O2", "WS_NO3", "WS_H2S")]
```

There is a better way!

Currently, the most popular alternative for data wrangling is the package `dplyr` in the tidyverse. This package is so good at what it does, and integrates so well with other popular tools like `ggplot2`, that it has rapidly become the de-facto standard and it is what we will focus on in this tutorial.

Compared to base R, `dplyr` code is much more readable because all operations are based on using `dplyr` functions or *verbs* (select, filter, mutate…) rather than base R's more difficult to read indexing system (brackets, parentheses…).

Typical data wrangling tasks in `dplyr`:

- `filter` out a subset of observations (rows)
- `select` a subset of variables (columns)
- `rename` variables
- `arrange` the observations by sorting variable(s) in ascending or descending order
- `mutate` all values of a variable (apply a transformation)

Each verb works similarly:

- input data frame in the first argument
- other arguments can refer to variables as if they were local objects
- output is another data frame

Install `tidyverse`

In order to use the functions in the tidyverse like those in the `dplyr` package, you must first install these packages. (More information available in our 'RStudio tutorial'.)

```r
install.packages("tidyverse")
```

*Please note that if you have **R v3.3 or older**, you may not be able to install **`tidyverse`**. In this case, you need to separately install each package within the tidyverse. This includes:* `readr`, `tibble`, `dplyr`, `tidyr`, `stringr`, `ggplot2`, `purr`, `forcats`

Load `tidyverse`

Installing a package downloads the relevant files onto your computer, but it does not allow R/RStudio to access the functions therein. Thus, you must load packages into RStudio using the `library()` function.

This may seem tedious but it is necessary every time you open a new RStudio session. Otherwise, the program would load every package you had ever downloaded every time it is opened. With 10s of 1000s of packages out there, this would significantly slow down RStudio. Thus, only load the packages you will need for a given project.

For this tutorial, you will need:

R v3.4 or newer

```r
library(tidyverse)
```

R v3.3 or older

```r
library(readr)
library(dplyr)
```

```
## Warning: package 'dplyr' was built under R version 3.5.2
```

```r
library(tidyr)
```

```
## Warning: package 'tidyr' was built under R version 3.5.2
```

Load data

Read in tabular data

To start analyzing data, we need to read it into RStudio. There a number of functions to do this; we will start with the most generic one in the tidyverse, `read_delim()`. As with all R functions, you specify the function name and then enclose any specific parameters or arguments within parentheses.

```
Function(arugument1=..., argument2=..., ...)
```

For example, we load the file `Saanich_Data.csv` like so.

```
read_delim(file="Saanich_Data.csv")
```

```
## Error in read_delim(file = "Saanich_Data.csv"): argument "delim" is missing, with no default
```

Here, we've told R our `file`. Since we are using a Project, R looks for this file in the Project directory. If we wanted to specify a file not in our Project folder, we would need to provide the full path like if the file were on my computer's Desktop.

```
read_delim(file="/Users/kim/Desktop/Saanich_Data.csv")
```

In either case, we get an **Error**, which means that the function did not complete. This occurs because we did not provide `read_delim` with enough information to successfully read in these data.

This is in contrast to a **warning**, which means that the function completed but there were one or more issues.

Add arugments

We can specify other features of the data using additional arguments in the `read_delim()` function. In the case of these data, we want to use:

- delim: tells R that the data are comma-delimited `","`. If you had a tab-delimited file, you would use `sep="\t"`.

```
read_delim(file="Saanich_Data.csv", delim=",")
```

```
## Parsed with column specification:
## cols(
##   .default = col_double(),
##   Date = col_date(format = "")
## )
```

```
## See spec(...) for full column specifications.
```

```
## # A tibble: 1,605 x 29
##    Longitude Latitude Cruise Date       Depth WS_O2   PO4    SI WS_NO3
##        <dbl>    <dbl>  <dbl> <date>     <dbl> <dbl> <dbl> <dbl>  <dbl>
##  1     -124.     48.6      1 2006-02-18 0.01     NA  2.42    NA   26.7
##  2     -124.     48.6      1 2006-02-18 0.025    NA  2.11    NA   23.2
##  3     -124.     48.6      1 2006-02-18 0.04     NA  2.14    NA   19.5
##  4     -124.     48.6      1 2006-02-18 0.055    NA  2.49    NA   22.6
##  5     -124.     48.6      1 2006-02-18 0.07     NA  2.24    NA   23.1
##  6     -124.     48.6      1 2006-02-18 0.08     NA  2.80    NA   23.1
##  7     -124.     48.6      1 2006-02-18 0.09     NA  2.93    NA   23.1
##  8     -124.     48.6      1 2006-02-18 0.1      NA  3.42    NA   21.0
##  9     -124.     48.6      1 2006-02-18 0.112    NA  4.08    NA  15.7
## 10     -124.     48.6      1 2006-02-18 0.125    NA  4.29    NA   9.72
## # ... with 1,595 more rows, and 20 more variables: Mean_NH4 <dbl>,
## #   Std_NH4 <dbl>, Mean_NO2 <dbl>, Std_NO2 <dbl>, WS_H2S <dbl>,
## #   Std_H2S <dbl>, Cells.ml <dbl>, Mean_N2 <dbl>, Std_n2 <dbl>,
## #   Mean_O2 <dbl>, Std_o2 <dbl>, Mean_co2 <dbl>, Std_co2 <dbl>,
## #   Mean_N2O <dbl>, Std_N2O <dbl>, Mean_CH4 <dbl>, Std_CH4 <dbl>,
## #   Temperature <dbl>, Salinity <dbl>, Density <dbl>
```

With these added arguments, we see that our data are now read in and correctly formatted in multiple columns with column (variable) names. Note that the tidyverse has automatically detected and formatted a column in these data that appear to be dates.

Argument names and order

You will see in the preceding command that for every argument, we provide the name, `=`, and then the necessary input for that argument. However, if you write arguments in the function's default order, you can leave out the argument name.

```
read_delim("Saanich_Data.csv", ",")
```

```
## Parsed with column specification:
## cols(
##   .default = col_double(),
##   Date = col_date(format = "")
## )
```

```
## See spec(...) for full column specifications.
```

Default orders are listed in a function's help page (accessed with `?`). Removing argument names should be done *with caution* as it makes the code less readable and runs the risk of breaking if a function updates or changes in the future. And if you provide arguments out of order, the argument name *must* be provided.

For example, this gives an error because the `delim` argument should be second and instead, we have `TRUE`, which is not an allowable input for `delim`.

```
read_delim("Saanich_Data.csv", TRUE, ",")
```

```
## Error in guess_header_(datasource, tokenizer, locale): Expecting a single string value: [type=logica
```

In contrast, specifying the argument names allows you to write them out of order. We will add the `col_names` argument (which is TRUE by default so this does not alter the function's output) to demonstrate this.

```
read_delim("Saanich_Data.csv", col_names=TRUE, delim=",")
```

Other `read_` functions

If you look at the `read_delim` help page, you will see several other similar functions. These allow you to read in data using fewer arguments. For example, `read_csv` reads in a comma-delimited file without the need to specify `delim=","`.

```
read_csv("Saanich_Data.csv")
```

```
## Parsed with column specification:
## cols(
##   .default = col_double(),
##   Date = col_date(format = "")
## )
## See spec(...) for full column specifications.
```

```
## # A tibble: 1,605 x 29
##    Longitude Latitude Cruise Date       Depth WS_O2   PO4    SI WS_NO3
##        <dbl>    <dbl>  <dbl> <date>     <dbl> <dbl> <dbl> <dbl>  <dbl>
##  1     -124.     48.6      1 2006-02-18 0.01     NA  2.42    NA  26.7
##  2     -124.     48.6      1 2006-02-18 0.025    NA  2.11    NA  23.2
##  3     -124.     48.6      1 2006-02-18 0.04     NA  2.14    NA  19.5
##  4     -124.     48.6      1 2006-02-18 0.055    NA  2.49    NA  22.6
##  5     -124.     48.6      1 2006-02-18 0.07     NA  2.24    NA  23.1
##  6     -124.     48.6      1 2006-02-18 0.08     NA  2.80    NA  23.1
##  7     -124.     48.6      1 2006-02-18 0.09     NA  2.93    NA  23.1
##  8     -124.     48.6      1 2006-02-18 0.1      NA  3.42    NA  21.0
##  9     -124.     48.6      1 2006-02-18 0.112    NA  4.08    NA  15.7
## 10     -124.     48.6      1 2006-02-18 0.125    NA  4.29    NA   9.72
## # ... with 1,595 more rows, and 20 more variables: Mean_NH4 <dbl>,
## #   Std_NH4 <dbl>, Mean_NO2 <dbl>, Std_NO2 <dbl>, WS_H2S <dbl>,
## #   Std_H2S <dbl>, Cells.ml <dbl>, Mean_N2 <dbl>, Std_n2 <dbl>,
## #   Mean_O2 <dbl>, Std_o2 <dbl>, Mean_co2 <dbl>, Std_co2 <dbl>,
## #   Mean_N2O <dbl>, Std_N2O <dbl>, Mean_CH4 <dbl>, Std_CH4 <dbl>,
## #   Temperature <dbl>, Salinity <dbl>, Density <dbl>
```

We will use this version to load in data for the remainder of this tutorial since it is the shortest command.

Save data to the Environment

All of the examples above simply print the data table into the console. This does not allow us to work further with these data! So, we need to name the data as an object in R so that it is saved in the local Environment and used in subsequent analyses.

We do this by prefacing `read_csv` with the name we want to give the table and either `=` or `<-` to assign that name to whatever `read_csv` reads in.

```
dat = read_csv("Saanich_Data.csv")
```

```
## Parsed with column specification:
## cols(
##   .default = col_double(),
##   Date = col_date(format = "")
## )
```

```
## See spec(...) for full column specifications.
```

```
#Which is equivalent to

dat <- read_csv("Saanich_Data.csv")
```

```
## Parsed with column specification:
## cols(
##   .default = col_double(),
##   Date = col_date(format = "")
## )
## See spec(...) for full column specifications.
```

Now, we see the `dat` object show up in our Environment in the upper right. If we click on that object in the Environment pane, we can see the table. Row and columns names are in grey boxes and data are in white boxes.

Since we are using an R project, we can close RStudio, save our `RScript` and `RData`, and when we re-open using the `.Rproj` file, our data will still be in the Environment! Try it and see!

Subset and clean data

Our `dat` data frame consists of 1605 rows (observations) and 29 columns (variables). However, there are many manipulations needed to clean-up these data prior to visualization or analysis.

Conditional statements

A conditional statement queries data and returns TRUE or FALSE based on the statement.

Some common statements include:

- `a == b` for 'a is equal to b'
  - Note that this is different from the single `=` used in functions!
- `a != b` for 'a is not equal to b'
- `a > b` for 'a is greater than b'
- `a >= b` for 'a is greater than or equal to b'
  - Similarly for `<` and `<=`
- `a %in% b` for 'a is in b'
- `is.na()` for 'is missing data'
- `!is.na()` for 'is not missing data'
  - Note that the `!` is often used to be a statement into its negative form such as `!=` and `!is.na()`

Logical operators

You can string together multiple conditional statements using logical operators. These most commonly include:

- `&` for 'and'
- `|` for 'or'

Filter

You can use `filter` to select specific rows using conditional statements and logical operators.

Below, we filter the data such that we only retain data with oxygen measurements (not NA). *Remember that using `!` means 'not'.*

```r
dat <- filter(dat, !is.na(WS_O2))
```

Note that this variable is a capital O as in Oxygen, not a 0 as in zero. Tab-completion can help you here!

Select

You can use the `select` function to focus on a subset of variables (columns). Let's select the variables that we will need for this tutorial, including:

- cruise #
- date
- depth in kilometers
- oxygen (O_2) in uM
- nitrate (NO_3) in uM

- hydrogen sulfide (H_2S) in uM

```
dat <- select(dat,
              Cruise, Date, Depth,
              WS_O2, WS_NO3, WS_H2S)
```

Note that we can spread a single function across several lines as long as the preceding line ends with a comma.

Rename

You can use the `rename` function to assign new names to your variables. This can be very useful when your variable names are very long and tedious to type. The setup is always **new_name = old_name**

Here, we remove the "WS_" prefixes from some of the variables and add units.

```
dat <- rename(dat, O2_uM=WS_O2, NO3_uM=WS_NO3, H2S_uM=WS_H2S)
```

This is a good point to touch on best practices for units in data. It is best to either name the units in the variable name, such as O2_uM for oxygen in micromolar, or to create a separate column for units, such as O2 = 3.4 and O2_units = uM. You should *never* place the units in the same variable as the measurement because then you cannot plot the variable as a number!

Arrange

Arrange reorders rows in ascending order of the provided variables. If we want to instead use descending order, we can add the `desc()` function. For example, let's order the data by hydrogen sulfide (`H2S_uM`).

```
dat <- arrange(dat, H2S_uM)
```

Mutate

Use `mutate(y=x)` to apply a transformation to some variable x and assign it to the variable name y. As with the rename function, it is always **new_name = old_name**.

Here, we multiply Depth by 1000 to convert its units from kilometers to meters.

```
dat <- mutate(dat, Depth_m=Depth*1000)
```

Link functions with %>%

Thus far, we have cleaned the data by repeatedly overwriting the `dat` object in R. This means that we may run into issues if we loose our place in the script. For example, if we try to re-run our renaming step, it fails since we've already renamed these variables and the original names no longer exist in `dat`.

```
dat <- rename(dat, O2_uM=WS_O2, NO3_uM=WS_NO3, H2S_uM=WS_H2S)
```

```
## Error in .f(.x[[i]], ...): object 'WS_O2' not found
```

Instead, we can start from our original `dat` object and chain commands together using the `%>%` operator, which is called a pipe. This is done by nesting the functions within one another. In generic terms, this is

```
g(f(x))
```

where the data `x` are first processed using the function `f()` and then the resulting data are fed directly into the function `g()`.

In tidyverse syntax using a pipe, the above is equivalent to

```
f(x) %>% g()
```

This condenses code and improves readability. Of note, the keyboard shortcut for the pipe in R is Cmd+Shft+M (Mac) or Ctrl+Shft+M (PC).

So, instead of reading in the data, saving as `dat`, and then filtering like so,

```
dat <- read_csv("Saanich_Data.csv")

dat <- filter(dat, !is.na(WS_O2))
```

We can pipe the data frame read into R with `read_delim` directly into our first cleaning step to `filter` out observations without any oxygen data. Note how we do not input the `dat` data as the first argument of filter anymore. This is because the pipe is providing the data input in lieu of providing it as an explicit argument.

```
dat <- read_csv("Saanich_Data.csv") %>%
  filter(!is.na(WS_O2))
```

We can continue to string together functions until all of our previous data cleaning steps (from filter to mutate) are encompassed in 1 piped tidyverse command.

```
dat <-
  read_csv("Saanich_Data.csv") %>%
  filter(!is.na(WS_O2)) %>%
  select(Cruise, Date, Depth, WS_O2, WS_NO3, WS_H2S) %>%
  rename(O2_uM=WS_O2, NO3_uM=WS_NO3, H2S_uM=WS_H2S) %>%
  arrange(H2S_uM) %>%
  mutate(Depth_m=Depth*1000)
```

```
## Parsed with column specification:
## cols(
##   .default = col_double(),
##   Date = col_date(format = "")
## )
```

```
## See spec(...) for full column specifications.
```

We could even go as far back as the Setup instructions and link the reading of the data from the web into our find tidyverse command.

```
dat <-
  read_csv("https://raw.githubusercontent.com/EDUCE-UBC/workshop_data/master/Saanich_Data.csv") %>%
  filter(!is.na(WS_O2)) %>%
  select(Cruise, Date, Depth, WS_O2, WS_NO3, WS_H2S) %>%
  rename(O2_uM=WS_O2, NO3_uM=WS_NO3, H2S_uM=WS_H2S) %>%
  arrange(H2S_uM) %>%
  mutate(Depth_m=Depth*1000)
```

```
## Parsed with column specification:
## cols(
##   .default = col_double(),
##   Date = col_date(format = "")
## )
```

```
## See spec(...) for full column specifications.
```

Summarize data

Another `dplyr` function that effectively utilizes pipes is `summarize` (or `summarise`). This function is handy when we want to calculate descriptive statistics for groups of observations in the data. This is done by first applying the `group_by` verb and then feeding the grouped data into `summarise`.

For example, we can calculate the mean, standard deviation, and sample size of oxygen concentrations by depth as follows.

```
dat %>%
  group_by(Depth_m) %>%
  summarise(Mean_O2=mean(O2_uM),
            SD_O2=sd(O2_uM),
            n=n())
```

```
## # A tibble: 24 x 4
##    Depth_m Mean_O2 SD_O2     n
##      <dbl>   <dbl> <dbl> <int>
##  1      10   211.   55.8    78
##  2      20   187.   50.3    78
##  3      40   159.   38.3    78
##  4      60   138.   45.2    78
##  5      75   105.   54.8    77
##  6      80    32.3  NA       1
##  7      85    81.9  48.8    77
##  8      90    68.2  42.1    77
##  9      97    49.1  34.7    77
## 10     100    43.6  33.1    78
## # ... with 14 more rows
```

You can find a list of other common statistical functions in the stats package that comes pre-installed with R.

Transform data frames

Gather and spread

Tabular data can come in long/narrow/skinny, short/wide/fat, or a combination of these two forms.

For example, this wide data

```
##   sample_ID year_2015 year_2016 year_2017
## 1         1     0.288     1.140     1.051
## 2         2     0.788     0.246     0.957
## 3         3     0.409     0.728     1.457
## 4         4     0.883     1.092     0.953
```

contains the same information as this long data.

```
##    sample_ID      Year Value
## 1          1 year_2015 0.288
## 2          2 year_2015 0.788
## 3          3 year_2015 0.409
## 4          4 year_2015 0.883
## 5          1 year_2016 1.140
## 6          2 year_2016 0.246
## 7          3 year_2016 0.728
## 8          4 year_2016 1.092
## 9          1 year_2017 1.051
## 10         2 year_2017 0.957
## 11         3 year_2017 1.457
## 12         4 year_2017 0.953
```

In their pure forms, wide data have as few rows and as many columns as possible for a given data set (while long data are the opposite - many rows, few columns).

Practically speaking, however, most data are a combination of the two types. For example, the `dat` data are in *partially* wide format. Observe that each variable is given its own column (wide) but time is already gathered into 1 column (long).

```
dat
```

```
## # A tibble: 1,249 x 7
##    Cruise Date       Depth O2_uM NO3_uM H2S_uM Depth_m
##     <dbl> <date>     <dbl> <dbl>  <dbl>  <dbl>   <dbl>
##  1     18 2008-02-13 0.01  225.    18.9      0      10
##  2     18 2008-02-13 0.02  221.    24.0      0      20
##  3     18 2008-02-13 0.04  202.    29.2      0      40
##  4     18 2008-02-13 0.06  198.    24.0      0      60
##  5     18 2008-02-13 0.075 194.    23.8      0      75
##  6     18 2008-02-13 0.085 150.    23.0      0      85
##  7     18 2008-02-13 0.09  122.    21.9      0      90
##  8     18 2008-02-13 0.097 108.    20.3      0      97
##  9     18 2008-02-13 0.1    99.8   19.4      0     100
## 10     18 2008-02-13 0.11   39.6   19.4      0     110
## # ... with 1,239 more rows
```

`gather` and `spread` are two functions from the `tidyr` package that enable easy conversion between wide/long formats. The first will `gather` your wide data into one column containing all of the variable names (the "Key" column) and another column containing all of the values (the "Value" column). The second function will `spread` your long data into the wide data format.

Here, we gather the three geochemical variables (O2_uM, NO3_uM, H2S_uM) from wide to long format. We'll save this as `dat2` so that we can easily compare the outcome to `dat`.

```
dat2 <- gather(dat, key="Key", value="Value", O2_uM, NO3_uM, H2S_uM)

dat2
```

```
## # A tibble: 3,747 x 6
##    Cruise Date       Depth Depth_m Key   Value
##     <dbl> <date>     <dbl>   <dbl> <chr> <dbl>
##  1     18 2008-02-13 0.01       10 O2_uM 225.
##  2     18 2008-02-13 0.02       20 O2_uM 221.
##  3     18 2008-02-13 0.04       40 O2_uM 202.
##  4     18 2008-02-13 0.06       60 O2_uM 198.
##  5     18 2008-02-13 0.075      75 O2_uM 194.
##  6     18 2008-02-13 0.085      85 O2_uM 150.
##  7     18 2008-02-13 0.09       90 O2_uM 122.
##  8     18 2008-02-13 0.097      97 O2_uM 108.
##  9     18 2008-02-13 0.1       100 O2_uM  99.8
## 10     18 2008-02-13 0.11      110 O2_uM  39.6
## # ... with 3,737 more rows
```

We see that our transformed data is indeed much longer (*i.e.* has more rows).

```r
dim(dat)
```

```
## [1] 1249    7
```

```r
dim(dat2)
```

```
## [1] 3747    6
```

And that the new Key column contains the 3 variables names that were gathered.

```r
unique(dat2$Key)
```

```
## [1] "O2_uM"  "NO3_uM" "H2S_uM"
```

We can then undo this by spreading our data from long to wide format.

```r
dat2 <- spread(dat2, key="Key", value="Value")
dat2
```

```
## # A tibble: 1,249 x 7
##    Cruise Date       Depth Depth_m H2S_uM NO3_uM O2_uM
##     <dbl> <date>     <dbl>   <dbl>  <dbl>  <dbl> <dbl>
##  1     18 2008-02-13 0.01       10      0   18.9 225.
##  2     18 2008-02-13 0.02       20      0   24.0 221.
##  3     18 2008-02-13 0.04       40      0   29.2 202.
##  4     18 2008-02-13 0.06       60      0   24.0 198.
##  5     18 2008-02-13 0.075      75      0   23.8 194.
##  6     18 2008-02-13 0.085      85      0   23.0 150.
##  7     18 2008-02-13 0.09       90      0   21.9 122.
##  8     18 2008-02-13 0.097      97      0   20.3 108.
##  9     18 2008-02-13 0.1       100      0   19.4  99.8
## 10     18 2008-02-13 0.11      110      0   19.4  39.6
## # ... with 1,239 more rows
```

So that it is now back to the original shape.

```r
dim(dat)
```

```
## [1] 1249    7
```

```r
dim(dat2)
```

```
## [1] 1249    7
```

Note that you can name the key and value columns anything that you want. For example, in the above case, we could have named the key and value based on what we know about these data.

```r
gather(dat, key="Geochemical", value="uM", O2_uM, NO3_uM, H2S_uM)
```

```
## # A tibble: 3,747 x 6
##    Cruise Date       Depth Depth_m Geochemical    uM
##     <dbl> <date>     <dbl>   <dbl> <chr>       <dbl>
##  1     18 2008-02-13 0.01       10 O2_uM       225.
##  2     18 2008-02-13 0.02       20 O2_uM       221.
##  3     18 2008-02-13 0.04       40 O2_uM       202.
##  4     18 2008-02-13 0.06       60 O2_uM       198.
##  5     18 2008-02-13 0.075      75 O2_uM       194.
##  6     18 2008-02-13 0.085      85 O2_uM       150.
##  7     18 2008-02-13 0.09       90 O2_uM       122.
##  8     18 2008-02-13 0.097      97 O2_uM       108.
##  9     18 2008-02-13 0.1       100 O2_uM        99.8
## 10     18 2008-02-13 0.11      110 O2_uM        39.6
## # ... with 3,737 more rows
```

You can see several applications of gather/spread in our 'Data visualization tutorial'.

*_join

Oftentimes you will need to work with data across multiple tables. These tables may function separately or they may need to be combined. If they need to be combined, `dplyr`'s suite of joining functions can be applied. Theses include:

- inner_join(x,y)
    - Keeps rows found in *both* x and y; keeps all columns from *both* x and y
    - Combine all data in x and y, keeping only rows found in both x and y
- semi_join(x,y)
    - Keeps rows found in *both* x and y; keeps all columns from x
    - Filter data in x to include only rows found in both x and y
- anti_join(x,y)
    - Keeps rows from x that are NOT found in y; keeps all columns from x
    - Filter data in x to include only rows NOT also found in y
- left_join(x,y)
    - Keeps all rows from x; keeps all columns from x and y
    - Combine all data in x and y, keeping only rows from x
- right_join(x,y)
    - Keeps all rows from y; keeps all columns from x and y
    - Combine all data in x and y, keeping only rows from y
- full_join(x,y)
    - Keeps all rows and all columns from both x and y
    - Combine all data in x and y, adding NAs where rows do not match

To practice this, we will artificially split up our data into separate data frames for oxygen and nitrate. We will also arrange the data by its geochemical variable (so that their rows are not in the same order) and filter out zeroes (so that not all rows in one are present in the other).

```r
dat_O2 <- dat %>%
  select(Cruise, Date, Depth_m, O2_uM) %>%
  arrange(O2_uM) %>%
  filter(O2_uM != 0)

dat_NO3 <- dat %>%
  select(Cruise, Date, Depth_m, NO3_uM) %>%
  arrange(NO3_uM) %>%
  filter(NO3_uM != 0)
```

We can see that we now have two data frames of different sizes with rows in differing orders.

```r
dat_O2
```

```
## # A tibble: 1,104 x 4
##    Cruise Date       Depth_m O2_uM
##     <dbl> <date>       <dbl> <dbl>
##  1     89 2013-11-13     200 0.079
##  2    100 2014-10-08     185 0.628
##  3     91 2014-01-15     165 0.653
##  4     92 2014-02-12     150 0.654
##  5     95 2014-05-14     150 0.682
## 6      91 2014-01-15     150 0.697
## 7      92 2014-02-12     165 0.733
## 8      91 2014-01-15     185 0.751
## 9      95 2014-05-14     135 0.761
## 10     99 2014-09-10     135 0.793
## # ... with 1,094 more rows
```

```r
dat_NO3
```

```
## # A tibble: 989 x 4
##    Cruise Date       Depth_m NO3_uM
##     <dbl> <date>       <dbl>  <dbl>
##  1     45 2010-05-12      10  0.011
##  2     23 2008-07-16     165  0.043
##  3     46 2010-06-16     135  0.046
##  4     40 2009-12-09     185  0.069
##  5     20 2008-04-09     200  0.117
##  6     40 2009-12-09     200  0.127
##  7     92 2014-02-12     200  0.128
##  8     57 2011-05-18     135  0.132
##  9     48 2010-08-11     135  0.147
## 10     31 2009-03-11     185  0.2
## # ... with 979 more rows
```

Thus, we cannot recombine these data by simply pasting 1 table next to the other. Instead, we can use joining.

If we wanted to recreate the original data in `dat`, we would use `full_join` to keep all rows and all columns from both data frames.

For all of the joining functions, you can either allow R to automatically detect the matching columns

```r
full_join(dat_O2, dat_NO3)
```

```
## Joining, by = c("Cruise", "Date", "Depth_m")
```

```
## # A tibble: 1,138 x 5
##    Cruise Date       Depth_m O2_uM NO3_uM
##     <dbl> <date>       <dbl> <dbl>  <dbl>
##  1     89 2013-11-13     200 0.079  0.886
##  2    100 2014-10-08     185 0.628 NA
##  3     91 2014-01-15     165 0.653  0.233
##  4     92 2014-02-12     150 0.654  6.74
##  5     95 2014-05-14     150 0.682  3.63
##  6     91 2014-01-15     150 0.697  3.20
##  7     92 2014-02-12     165 0.733 NA
##  8     91 2014-01-15     185 0.751 NA
##  9     95 2014-05-14     135 0.761 17.7
## 10     99 2014-09-10     135 0.793 NA
## # ... with 1,128 more rows
```

or explicitly state which columns to compare.

```r
full_join(dat_O2, dat_NO3, by=c("Cruise", "Date", "Depth_m"))
```

```
## # A tibble: 1,138 x 5
##    Cruise Date       Depth_m O2_uM NO3_uM
##     <dbl> <date>       <dbl> <dbl>  <dbl>
##  1     89 2013-11-13     200 0.079  0.886
##  2    100 2014-10-08     185 0.628 NA
##  3     91 2014-01-15     165 0.653  0.233
##  4     92 2014-02-12     150 0.654  6.74
##  5     95 2014-05-14     150 0.682  3.63
##  6     91 2014-01-15     150 0.697  3.20
##  7     92 2014-02-12     165 0.733 NA
##  8     91 2014-01-15     185 0.751 NA
##  9     95 2014-05-14     135 0.761 17.7
## 10     99 2014-09-10     135 0.793 NA
## # ... with 1,128 more rows
```

In either case, we see that we've combined the oxygen and nitrate data back into 1 data frame, though there are now NAs where we deleted 0 data from the original `dat`.
