Week 3: Combining Steps with Pipes

Ellen Bledsoe

2025-02-04

Combining Steps with Pipes

Because we are starting a new session in R, we need to load in our packages again. In Posit Cloud, you often don't *actually* need to do this, but it is good practice, regardless.

Load both `dplyr` and `readr`. If you need to read in the .csv files again, do so here, as well.

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

```
surveys <- read_csv("surveys.csv")
```

```
## Rows: 35549 Columns: 9
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (2): species_id, sex
## dbl (7): record_id, month, day, year, plot_id, hindfoot_length, weight
##
## 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.
```

```
species <- read_csv("species.csv")
```

```
## Rows: 54 Columns: 4
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (4): species_id, genus, species, taxa
##
## 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.
```

```
plots <- read_csv("plots.csv")
```

```
## Rows: 24 Columns: 2
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (1): plot_type
## dbl (1): plot_id
##
## 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.
```

When Order Matters

When we run code in a code chunk or in a script/file, R runs the code in order, from beginning to end (first line first, then the second line, and so on).

Let's look at an example: Obtain the data for only DS, sorted by year, with only the year and and weight columns

```
# why doesn't this code work?
ds_weight_by_year <- select(ds_data_by_year, year, weight)
ds_data <- filter(surveys, species_id == "DS", !is.na(weight))
ds_data_by_year <- arrange(ds_data, year)
```

Why doesn't the code above work?

The order of object creation is incorrect. In the first line of code, the `select()` function trying to pull columns from an object that hasn't been created yet.

We can put the code in the correct order, and the code chunk with run smoothly.

```
ds_data <- filter(surveys, species_id == "DS", !is.na(weight))
ds_data_by_year <- arrange(ds_data, year)
ds_weight_by_year <- select(ds_data_by_year, year, weight)
```

Let's Practice!

Work on Questions 2 and 3 in the Assignment.

The pipe (|> or %>%)

You'll notice that in the code above, we made a bunch on intermediate objects (`ds_data`, `ds_data_by_year`). Instead of doing that, we can use a pipe operator to chain functions together.

You can think of the pipe as automatically sending the output from the first line into the next line as the input.

Using pipes is helpful for a lot of reasons, including:

1. removing the clutter of creating a lot of intermediate objects in your work space, which reduces the chance of errors caused by using the wrong input object
2. makes things more human-readable (in addition to computer-readable)

The shortcut for typing a pipe is `Ctrl` + `Shift` + `M` (or `Cmd` + `Shift` + `M` on a Mac)

Let's start with a very small example, one we probably won't actually use in our own code.

```
x = c(1, 2, 3)
mean(x)
```

```
## [1] 2
```

Instead of putting `x` as an argument in the `mean()` function, we could instead pipe the vector into the function.

```
x %>% mean()
```

```
## [1] 2
```

This works the same way as above; `x` becomes the first argument in mean.

If we want to add other arguments, they get added to the function call. The `mean()` function already knows that our first argument in going to be `x`.

```
x = c(1, 2, 3, NA)
mean(x, na.rm = TRUE)
```

```
## [1] 2
```

```
# with a pipe
x |> mean(na.rm = TRUE)
```

```
## [1] 2
```

Let's try this with some of the `dplyr` functions we have learned.

```
# one pipe
surveys |>
  filter(species_id == "DS")
```

```
## # A tibble: 2,504 x 9
##    record_id month   day  year plot_id species_id sex   hindfoot_length weight
##        <dbl> <dbl> <dbl> <dbl>   <dbl> <chr>      <chr>           <dbl>  <dbl>
##  1        11     7    16  1977       5 DS         F                  53     NA
##  2        17     7    16  1977       3 DS         F                  48     NA
##  3        20     7    17  1977      11 DS         F                  48     NA
##  4        30     7    17  1977      10 DS         F                  52     NA
##  5        42     7    18  1977      18 DS         F                  46     NA
##  6        58     7    18  1977      12 DS         M                  45     NA
##  7        73     8    19  1977       3 DS         F                  44     NA
##  8        76     8    19  1977       9 DS         F                  47     NA
##  9        80     8    19  1977       1 DS         M                  48     NA
## 10        91     8    20  1977      11 DS         F                  50     NA
## # i 2,494 more rows
```

```
# recreate the ds_weight_by_year df with pipes
surveys |>
  filter(species_id == "DS", !is.na(weight)) |>
  arrange(year) |>
  select(year, weight)
```

```
## # A tibble: 2,344 x 2
##     year weight
##    <dbl>  <dbl>
##  1  1977    117
##  2  1977    121
```

```
##  3  1977    115
##  4  1977    120
##  5  1977    118
##  6  1977    126
##  7  1977    132
##  8  1977    113
##  9  1977    122
## 10  1977    107
## # i 2,334 more rows
```

```
# need to use assignment arrow to save the changes to the object, as always
ds_weight_by_year <- surveys |>
  filter(species_id == "DS", !is.na(weight)) |>
  arrange(year) |>
  select(year, weight)
```

Base R pipe vs. `magrittr` pipe

So far, we have been using the pipe that is now "native" to R, meaning it is automatically loaded into R; it isn't part of a package that needs to be loaded. It looks like `|>`.

The first pipe created to be used in R looked like `%>%` and had to be loaded from an R package if you wanted to use it. It is part of the `magrittr` package and automatically gets loaded when you load a package from the `tidyverse`, such as `dplyr`.

Even though I learned to use `%>%`, I am starting to transition to the native R pipe, `|>`, in my own work and in teaching. It is even what the creator of the `tidyverse` recommends!

These pipes operate in nearly the same way, and you can use either one you'd like. If you want to change to the base R pipe in a Posit Cloud project or on your own computer, you can do so:

Tools -> Global Options -> Code -> Use native pipe operator

Let's Practice

Try your hand at the Portal Data Manipulation Pipes (Questions 4 and 5).

Placeholders

Occasionally (not in your assignment, for the record), you will need to pipe the result of a line to something other than the first argument.

To do so, we use a placeholder. This is one major difference between the 2 pipes.

- For `|>`, the placeholder is `_`
- For `%>%`, the placeholder is `.`

We can demonstrate by fitting a linear model at the end of our `dplyr` pipeline.

The `lm` function takes a formula as the first argument (names of columns to use for the dependent and independent variables). The second argument tells it where the data is (our data frame, the thing we are piping).

```r
# lm(dependent_column ~ independent_column, data = <dataframe>)

# base R pipe
surveys |>
  filter(species_id == "DS", !is.na(weight)) |>
  arrange(year) |>
  select(year, weight) |>
  lm(weight ~ year, data = _)
```

```
##
## Call:
## lm(formula = weight ~ year, data = select(arrange(filter(surveys,
##     species_id == "DS", !is.na(weight)), year), year, weight))
##
## Coefficients:
## (Intercept)         year
##   -709.1968       0.4184
```

```r
# magrittr pipe
surveys %>%
  filter(species_id == "DS", !is.na(weight)) %>%
  arrange(year) %>%
  select(year, weight) %>%
  lm(weight ~ year, data = .)
```

```
##
## Call:
## lm(formula = weight ~ year, data = .)
##
## Coefficients:
## (Intercept)         year
##   -709.1968       0.4184
```