Aggregating and analyzing data with dplyr

DC contributors and Dmytro Fishman

Data Manipulation using dplyr

Bracket subsetting is handy, but it can be cumbersome and difficult to read, especially for complicated operations. Enter `dplyr`. `dplyr` is a package for making data manipulation easier.

Packages in R are basically sets of additional functions that let you do more stuff. The functions we've been using so far, like `str()` or `data.frame()`, come built into R; packages give you access to more of them. Before you use a package for the first time you need to install it on your machine, and then you should to import it in every subsequent R session when you'll need it.

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

You might get asked to choose a CRAN mirror – this is basically asking you to choose a site to download the package from. The choice doesn't matter too much; we recommend the RStudio mirror.

```r
library("dplyr")    ## load the package
```

What is `dplyr`?

The package `dplyr` provides easy tools for the most common data manipulation tasks. It is built to work directly with data frames. The thinking behind it was largely inspired by the package `plyr` which has been in use for some time but suffered from being slow in some cases.`dplyr` addresses this by porting much of the computation to C++. An additional feature is the ability to work with data stored directly in an external database. The benefits of doing this are that the data can be managed natively in a relational database, queries can be conducted on that database, and only the results of the query returned.

This addresses a common problem with R in that all operations are conducted in memory and thus the amount of data you can work with is limited by available memory. The database connections essentially remove that limitation in that you can have a database of many 100s GB, conduct queries on it directly, and pull back just what you need for analysis in R.

Selecting columns and filtering rows

We're going to learn some of the most common `dplyr` functions: `select()`, `filter()`, `mutate()`, `group_by()`, and `summarize()`. To select columns of a data frame, use `select()`. The first argument to this function is the data frame (`temperature`), and the subsequent arguments are the columns to keep.

```r
select(temperature, year, country_id, AverageTemperatureFahr)
```

To choose rows, use `filter()`:

```r
filter(temperature, year == 1995)
```

Pipes

But what if you wanted to select and filter at the same time? There are three ways to do this: use intermediate steps, nested functions, or pipes. With the intermediate steps, you essentially create a temporary data frame and use that as input to the next function. This can clutter up your workspace with lots of objects. You can also nest functions (i.e. one function inside of another). This is handy, but can be difficult to read if too many functions are nested as the process from inside out. The last option, pipes, are a fairly recent addition to R. Pipes let you take the output of one function and send it directly to the next, which is useful when you need to many things to the same data set. Pipes in R look like `%>%` and are made available via the `magrittr` package installed as part of `dplyr`.

```
temperature %>%
  filter(AverageTemperatureFahr < 10) %>%
  select(City, country_id, AverageTemperatureFahr, year, month)
```

In the above we use the pipe to send the `temperature` data set first through `filter`, to keep rows where `AverageTemperatureFahr` was less than 5 (which is -15 degrees of Celsius), and then through `select` to keep the `City`, `country_id`, `AverageTemperatureFahr`, `year` and `month` columns. When the data frame is being passed to the `filter()` and `select()` functions through a pipe, we don't need to include it as an argument to these functions anymore.

If we wanted to create a new object with this smaller version of the data we could do so by assigning it a new name:

```
temperature_sml <- temperature %>%
  filter(AverageTemperatureFahr < 10) %>%
  select(City, country_id, AverageTemperatureFahr, year, month)

temperature_sml
```

Note that the final data frame is the leftmost part of this expression.

> **Challenge**
>
> Using pipes, subset the data to include measurements collected after 2011, and retain the columns `year`, `City`, and `Latitude`

```
## Answer
temperature %>%
    filter(year > 2011) %>%
    select(year, City, Latitude)
```

Mutate

Frequently you'll want to create new columns based on the values in existing columns, for example to do unit conversions, or find the ratio of values in two columns. For this we'll use `mutate()`.

To create a new column of AverageTemperature in degrees Celsius:

```
temperature %>%
  mutate(AverageTemperatureCelsius = (AverageTemperatureFahr-32)*(5/9))
```

If this runs off your screen and you just want to see the first few rows, you can use a pipe to view the `head()` of the data (pipes work with non-dplyr functions too, as long as the `dplyr` or `magrittr` packages are loaded).

```
temperature %>%
  mutate(AverageTemperatureCelsius = (AverageTemperatureFahr-32)*(5/9)) %>%
  head
```

The first few rows are full of NAs, so if we wanted to remove those we could insert a `filter()` in this chain:

```
temperature %>%
  filter(!is.na(AverageTemperatureFahr)) %>%
  mutate(AverageTemperatureCelsius = (AverageTemperatureFahr-32)*(5/9)) %>%
  head
```

`is.na()` is a function that determines whether something is or is not an `NA`. The `!` symbol negates it, so we're asking for everything that is not an `NA`.

> **Challenge**
>
> Create a new dataframe from the temperature data that meets the following criteria: contains only the `country_id`,`City`, `year` columns and a column that contains values that are half the `AverageTemperatureCelsius` values (e.g. a new column `AverageTemperatureCelsius_half`). In this `AverageTemperatureCelsius_half` column, there are no NA values and all values are < -7.
>
> **Hint**: think about how the commands should be ordered to produce this data frame!

```
## Answer
temperature %>%
    filter(!is.na(AverageTemperatureFahr)) %>%
    mutate(AverageTemperatureCelsius_half = (AverageTemperatureFahr-32)*(5/9)/2) %>%
    filter(AverageTemperatureCelsius_half < -7) %>%
    select(country_id, City, year, AverageTemperatureCelsius_half)
```

Let's finally update our `data.frame` with `AverageTemperatureCelsius` and `AverageTemperatureUncertaintyCelsius`

```
temperature <- temperature %>%
  mutate(AverageTemperatureCelsius = (AverageTemperatureFahr-32)*(5/9)) %>%
  mutate(AverageTemperatureUncertaintyCelsius = (AverageTemperatureUncertaintyFahr-32)*(5/9))
```

Split-apply-combine data analysis and the summarize() function

Many data analysis tasks can be approached using the "split-apply-combine" paradigm: split the data into groups, apply some analysis to each group, and then combine the results. `dplyr` makes this very easy through the use of the `group_by()` function. `group_by()` splits the data into groups upon which some operations can be run. For example, if we wanted to group by Country and find the number of rows of data for each Country, we would do:

```
temperature %>%
  group_by(Country) %>%
  tally()
```

Here, `tally()` is the action applied to the groups created to `group_by()` and counts the total number of records for each category. `group_by()` is often used together with `summarize()` which collapses each group into a single-row summary of that group. So to view mean `AverageTemperatureFahr` by Country:

```
temperature %>%
  filter(!is.na(Country)) %>%
  group_by(Country) %>%
  summarize(mean_temperature = mean(AverageTemperatureCelsius, na.rm = TRUE))
```

You can group by multiple columns too:

```
temperature %>%
  filter(!is.na(Country)) %>%
  group_by(Country, City) %>%
  summarize(mean_temperature = mean(AverageTemperatureCelsius, na.rm = TRUE))
```

All of a sudden this isn't running off the screen anymore. That's because `dplyr` has changed our `data.frame` to a `tbl_df`. This is a data structure that's very similar to a data frame; for our purposes the only difference is that it won't automatically show tons of data going off the screen.

You can also summarize multiple variables at the same time:

```
temperature %>%
   filter(!is.na(Country)) %>%
   group_by(Country, City) %>%
   summarize(mean_temperature = mean(AverageTemperatureCelsius, na.rm = TRUE),
   min_temperature = min(AverageTemperatureCelsius, na.rm = TRUE))
```

Challenge

How many times was each `City` was measured?

```
## Answer
temperature %>%
    group_by(City) %>%
    tally
```

Challenge

Use `group_by()` and `summarize()` to find the mean, min, and max AverageTemperatureUncertaintyCelsius length for each City

```
## Answer
temperature %>%
    filter(!is.na(AverageTemperatureUncertaintyCelsius)) %>%
    filter(!is.na(City)) %>%
    group_by(City) %>%
    summarize(
        mean_Uncertainty = mean(AverageTemperatureUncertaintyCelsius),
        min_Uncertainty = min(AverageTemperatureUncertaintyCelsius),
        max_Uncertainty = max(AverageTemperatureUncertaintyCelsius)
    )
```

Challenge

What was the lowest temperature measured in each year? Return the columns `year`, `Country`, `City`, and `AverageTemperatureCelsius`.

```
## Answer
temperature %>%
    filter(!is.na(AverageTemperatureCelsius)) %>%
    filter(!is.na(Country)) %>%
    group_by(year) %>%
    filter(AverageTemperatureCelsius == max(AverageTemperatureCelsius)) %>%
    select(year, Country, City, month, AverageTemperatureCelsius) %>%
    arrange(year)
```

Handy dplyr cheatsheet

Much of this lesson was copied or adapted from Jeff Hollister's materials