Data Manipulation: Part 1

This tutorial introduces the dplyr package for general purpose data manipulation. dplyr is written by Hadley Wickham and can be installed from CRAN using the install.packages() function. The GitHub page for the project is found here: https://github.com/hadley/dplyr. It makes processing data frames easier and quicker.

This tutorial will cover the following functions in the dplyr package:

* [select()](#select), selecting columns
* [arrange()](#arrange), reordering rows
* [filter()](#filter), filtering rows

It also covers the following advanced topics:

* [What does %in% do?](#in)
* [dplyr and databases](#databases)

In the folder we provided you will also find a printed version of the RStudio cheatsheet for data wrangling found here: https://www.rstudio.com/resources/cheatsheets/. All of the functions from dplyr that we cover in this course can be found on that cheatsheet.

The data we will be using for demonstrations in this tutorial can be found in the region5air package. That package is not on CRAN, but can be found on GitHub: https://github.com/NateByers/region5air. If you have not installed this package, below is code that will install it using the install\_github() function from the devtools package.

library(devtools)  
install.github("NateByers/region5air")

Once the package has been installed, we need to library it and load the airdata data frame into our global environment.

library(region5air)  
data(airdata)  
head(airdata)

## site data\_status action\_code datetime parameter  
## 1 840170890005 0 10 20141231T0100-0600 44201  
## 2 840170311601 0 10 20141231T0100-0600 44201  
## 3 840170314002 0 10 20141231T0100-0600 44201  
## 4 840170310001 0 10 20141231T0100-0600 44201  
## 5 840171110001 0 10 20141231T0100-0600 44201  
## 6 840170971007 0 10 20141231T0100-0600 44201

For information about this data set, type ?airdata.

## Select columns with select()

Using base R functions, if we wanted to select a few columns from airdata, we would need to either use numbers or names inside of the [ , ] function. In dplyr we simply use the select() function.

As with all dplyr functions, the first parameter is the data frame that you want to perform the operation on. In our case we want to select a few columns from the airdata data frame, so it will look something like this: select(airdata, ...). But what are the column names? We use the names function to find out.

names(airdata)

## [1] "site" "data\_status" "action\_code" "datetime" "parameter"   
## [6] "duration" "frequency" "value" "unit" "qc"   
## [11] "poc" "lat" "lon" "GISDatum" "elev"   
## [16] "method\_code" "mpc" "mpc\_value" "uncertainty" "qualifiers"

Suppose we wanted to create a data frame with just columns "site", "datetime", "parameter", and "value". This is how we would use the select() function to do that.

library(dplyr)  
d <- select(airdata, site, datetime, parameter, value)  
head(d)

## site datetime parameter value  
## 1 840170890005 20141231T0100-0600 44201 0.022  
## 2 840170311601 20141231T0100-0600 44201 0.021  
## 3 840170314002 20141231T0100-0600 44201 0.018  
## 4 840170310001 20141231T0100-0600 44201 0.021  
## 5 840171110001 20141231T0100-0600 44201 0.023  
## 6 840170971007 20141231T0100-0600 44201 0.026

As with all dplyr functions, the output is always a data frame (or data-frame- like) object.

You can also select consecutive columns by separating the first column and the last column with a colon. We've seen this before. We can use the colon to create a consecutive series of integers.

1:3

## [1] 1 2 3

We can use that short hand to select the first three columns of a data frame.

b <- airdata[, 1:3]  
head(b)

## site data\_status action\_code  
## 1 840170890005 0 10  
## 2 840170311601 0 10  
## 3 840170314002 0 10  
## 4 840170310001 0 10  
## 5 840171110001 0 10  
## 6 840170971007 0 10

In select() we can pick a first column by name and a last column by name and separate them with a colon.

d <- select(airdata, site, datetime:value)  
head(d)

## site datetime parameter duration frequency value  
## 1 840170890005 20141231T0100-0600 44201 60 0 0.022  
## 2 840170311601 20141231T0100-0600 44201 60 0 0.021  
## 3 840170314002 20141231T0100-0600 44201 60 0 0.018  
## 4 840170310001 20141231T0100-0600 44201 60 0 0.021  
## 5 840171110001 20141231T0100-0600 44201 60 0 0.023  
## 6 840170971007 20141231T0100-0600 44201 60 0 0.026

## Arrange rows with arrange()

In dplyr ordering a data frame is very easy. We use the arrange() function. As always, the data frame is the first parameter. Then we provide the columns by which to arrange the data frame. If we wanted to arrangeairdata by first ordering the rows by "site" and then by datetime, we would do this:

head(airdata)

## site data\_status action\_code datetime parameter  
## 1 840170890005 0 10 20141231T0100-0600 44201  
## 2 840170311601 0 10 20141231T0100-0600 44201  
## 3 840170314002 0 10 20141231T0100-0600 44201  
## 4 840170310001 0 10 20141231T0100-0600 44201  
## 5 840171110001 0 10 20141231T0100-0600 44201  
## 6 840170971007 0 10 20141231T0100-0600 44201

airdata <- arrange(airdata, site, datetime)  
head(airdata)

## site data\_status action\_code datetime parameter  
## 1 840170310001 0 10 20130403T1400-0600 44201  
## 2 840170310001 0 10 20130403T1500-0600 44201  
## 3 840170310001 0 10 20130403T1600-0600 44201  
## 4 840170310001 0 10 20130403T1700-0600 44201  
## 5 840170310001 0 10 20130403T1800-0600 44201  
## 6 840170310001 0 10 20130403T1900-0600 44201

Again, the output of a dplyr function is always some modified version of the data frame that you supplied in the first parameter.

If we wanted to arrange the data frame by descending order, we simply wrap the column with the desc() function.

airdata <- arrange(airdata, desc(site), datetime)  
head(airdata)

## site data\_status action\_code datetime parameter  
## 1 840550590025 0 10 20130513T1300-0600 44201  
## 2 840550590025 0 10 20130513T1400-0600 44201  
## 3 840550590025 0 10 20130513T1500-0600 44201  
## 4 840550590025 0 10 20130513T1600-0600 44201  
## 5 840550590025 0 10 20130513T1700-0600 44201  
## 6 840550590025 0 10 20130513T1800-0600 44201

## Filter rows with filter()

To filter a data frame we need to use a logical vector. For example, if we wanted to filter airdata by pollutant and POC, we would do something like this in base R.

b <- airdata[airdata$parameter == 44201 & airdata$poc == 1, ]  
  
# show the dimensions of the data frame--  
# the first number is the total number of rows, the second is the total number of columns  
dim(b)

## [1] 260911 20

The result of airdata$parameter == 44201 & airdata$poc == 1 is a logical vector as long as the number of rows in airdata. You must put in those dollar signs and separate the two comparisons with &. With dplyr you can use the filter() function without dollar signs and separate the logical comparisons with a comma, which is equivalent to &.

d <- filter(airdata, parameter == 44201, poc == 1)  
dim(d)

## [1] 260911 20

If you want to use two logical comparisons and combine them with an OR, you must use the | symbol. Suppose we want to subset down to all records with parameter 6201 and with a POC of 1 *or* 2.

d <- filter(airdata, parameter == 44201, poc == 1 | poc == 2)  
dim(d)

## [1] 284665 20

For review, here is a table of logical operators in R.

|  |  |
| --- | --- |
| Operator | Description |
| < | less than |
| <= | less than or equal to |
| > | greater than |
| >= | greater than or equal to |
| == | exactly equal to |
| != | not equal to |
| x | y | x OR y |
| x & y | x AND y |
| x %in% y | compare consecutive elements of x to all elements of y |

## Additional comments on dplyr

* Some functions don't appear to be that much easier to use than base R (like select() or arrange()). But dplyr provides a suite of functions *with the same syntax* so that you can easily remember them.
* dplyr is fast.
* You can use dplyr with databases.

## Exercises

Exercises for this tutorial can be found here: http://rpubs.com/NateByers/datamanip1exercises.

## Advanced Topics

### What does %in% do?

%in% is one of the more difficult logical operators to understand, but it's extremely useful. Here's a demonstration:

c("b", "c", "z") %in% c("a", "b", "c", "d", "e", "f")

## [1] TRUE TRUE FALSE

The operator takes the first element of the vector on the left and checks to see if the element is in the vector on the right. Since "b" is in the vector on the right, the first value of the returned vector is TRUE. "c" is also in the vector on the right, so the second value in the returned vector is TRUE. But "z" is not in the vector on the right. The third value is FALSE, and the length of the returned vector equals the length of the vector on the left, which is 3.

Now let's switch the vectors around.

c("a", "b", "c", "d", "e", "f") %in% c("b", "c", "z")

## [1] FALSE TRUE TRUE FALSE FALSE FALSE

The length of the output vector is 6, equal to the length of the vector on the left. The output vector is TRUE for letters that are in the vector on the right and FALSE for the letters that are not in the vector on the right.

### dplyr and Databases

You can use dplyr functions directly on databases. The functions will actually translate what you are doing in R to an SQL statement and pass that statement to the database. This is convenient if you have a large database and small RAM. Instead of pulling a large amount of data into RAM and then subsetting it, you can subset in the database itself and only pull a smaller amount of data into R.

dplyr has built-in support for SQLite, MySQL, MariaDB, and PostgreSQL. Let's build another simple SQLite database for demonstration.

# get the Cities table from the AQS codes site and replace . with \_ in names  
cities <- read.csv("https://aqs.epa.gov/aqsweb/codes/data/CityNamesByState.csv",  
 skip = 1, stringsAsFactors = FALSE)  
names(cities) <- sub("\\.", "\_", names(cities))  
head(cities)

## State\_Name State\_Code City\_Code City\_Name County\_Code County\_Name  
## 1 Alabama 01 124 Abbeville 67 Henry  
## 2 Alabama 01 460 Adamsville 73 Jefferson  
## 3 Alabama 01 484 Addison 133 Winston  
## 4 Alabama 01 676 Akron 65 Hale  
## 5 Alabama 01 820 Alabaster 117 Shelby  
## 6 Alabama 01 988 Albertville 95 Marshall

# how big is this thing?  
object.size(cities)

## 2236160 bytes

library(RSQLite)  
  
# create an SQLite database and save connection in db\_cities object  
db\_cities <- dbConnect(drv = SQLite(), dbname = "dplyrTest.sqlite")  
  
# load the Cities table  
dbWriteTable(db\_cities, "Cities", cities)

## [1] TRUE

Now we make a dplyr connection to the database and the Cities table.

# dplyr database connection  
dplyr\_db <- src\_sqlite("dplyrTest.sqlite")  
  
# dplyr table connection  
cities\_tbl <- tbl(dplyr\_db, "Cities")

Now we'll filter down to Michigan cities by using the dplyr function filter().

mich\_cities <- filter(cities\_tbl, State\_Name == "Michigan")  
head(mich\_cities)

## State\_Name State\_Code City\_Code City\_Name County\_Code County\_Name  
## 1 Michigan 26 380 Addison 91 Lenawee  
## 2 Michigan 26 440 Adrian 91 Lenawee  
## 3 Michigan 26 620 Ahmeek 83 Keweenaw  
## 4 Michigan 26 700 Akron 157 Tuscola  
## 5 Michigan 26 860 Alanson 47 Emmet  
## 6 Michigan 26 980 Albion 25 Calhoun

Let's see how much space this object requires.

object.size(mich\_cities)

## 5912 bytes

Now we'll remove the dplyr connections, disconnect from the database, and the database from our working directory.

rm(dplyr\_db, cities\_tbl)  
dbDisconnect(db\_cities)

## [1] TRUE

unlink("dplyrTest.sqlite")