Data Wrangling with R

Jonathan Gilligan

August 30, 2017

Data in R

Kinds of variables

R is capable of analyzing many different kinds of data. Some of the most important kinds of data that we may work with are:

- **Integer data**, which represents discrete quantities, such as counting events or objects.
- **Real number data**, which represents quantities that can have fractional values. Most of the data we will work with in this course, such as temperatures, altitudes, amounts of rainfall, and so forth, will be real-number data. This kind of data is also referred to frequently as "floating point" data or (for obscure reasons having to do with computer hardware) as "double" or "double-precision" data.
- **Character data**, which represents text. Examples include names of months, or categories (such as the name of a city or country). This kind of data is also referred to as "string data".
- **factor data**, which represents variables that can only take on certain discrete values. R treats factor data as a kind of augmented character data.

  The difference between character data and factor data is that factor data has an explicit set of allowed values and has an integer number associated with each of those values.

  For instance, if I have a factor variable with the allowed values "up" and "down", then I could not assign it a value "left" or "right", whereas a character variable can be assigned any arbitrary text, such as "second star to the right and straight on til morning."

  There are two kinds of factors: ordered and unordered. The difference is that the legal values for ordered factors have a specific order, so you can say that one comes before or after another (or is greater than or less than another), whereas unordered factors don't have any natural ordering.

  Examples of ordered variables might be the months of the year, or the days of the week, or a grouping like "small", "medium", "large", or "bad", "fair", "good".

  Examples of unordered variables might be lists of states, gender, religion, cities, sports teams, or other descriptive characteristics that don't have a natural order to them.
- **date and time data**, which represents calendar dates, times of day, or a combination, such as 9:37 PM on January 17, 1984.

For the most part, R handles different data types sensibly so you don't need to worry about them, but sometimes when R is reading data in from files, or when you want to convert one kind of variable to another, you will need to think about these.

The most common cases where you will need to think about this is when you are reading data in from files. Sometimes it is ambiguous whether to treat something from a file as character data, numerical data, or a date. In such cases, you may need to give R guidance about how to interpret data. The functions for reading data in from files, such as `read_csv` and `read_table` allow you to specify whether a given column of data in a table is integer, double-precision (floating point), character, date, etc.

R also provides functions for converting data. The `as.character` function takes data that might be character, factor, or numeric, and represent it as text (characters).

`as.numeric` or `as.integer` will allow you to convert a character variable to a number. For instance, `as.numeric("3.14")` converts a text variable "3.14" into a numeric variable 3.14.

`as.integer` is very useful when we want to convert an ordered factor to an integer that corresponds to the order of that value. For instance, if I have an ordered factor `f` with legal values corresponding to the months of the year ("Jan", "Feb", ..., "Dec"), if `f` has the value `Mar`, then `as.integer(f)` will have the integer value 3.

Vectors, Lists, Data Frames, and Tibbles.

In statistics, you generally don't just work with one number at a time, but with collections of numbers. R provides many ways to work with collections of numbers.

Vectors

The simplest is a **vector**. A vector is a collection of values that are all of the same kind: a collection of integers, a collection of floating point values, a collection of character values, a collection of factor values, etc.

You specify vectors like this: `x = c(1, 2, 5, 9, 3, 4, 2, 7, 5)`. You can access elements of vectors by indexing their position within the vector, so `x[3]` will be 5 and `x[4]` will be 9.

You can also give the elements of a vector names: `ages = c(Sam = 27, Ben = 20, Sarah = 25, Deborah = 31)` allows you to use `ages["Ben"]`, which will be 20.

All of the elements of a vector have to be the same kind, so `x = c(1, 2, "three")` will not allow the vector to mix numbers and characters and R will transform all of the values to character. The result is `"1", "2", "three"`, and `x[1] + x[2]` will give an error because R doesn't know how to add two character variables. However, `as.numeric(x[1]) + as.numeric(x[2])` will yield 3.

Lists

Lists are a lot like vectors, but they can contain different kinds of variables. They can even contain lists and vectors. `x = list(1, 2, "three", list(4, 5, 6))` has four elements. The first two are the numbers 1 and 2; the third is the character string "three", and the fourth is the list `(4,5,6)`.

Just as we can have named vectors, we can have named lists: `ages = list(Sam = 27, Ben = 20, Sarah = 25, Deborah = 31)`. There is a nice shortcut to getting the elements of a named list, using a dollar sign: `ages$Sam` is 27.

Data frames and tibbles

We will not use lists very much in this class. We will use vectors a little bit, but what we will use *a lot* are tables of data. You have probably worked a lot with spreadsheets and other data analysis tools that organize data in a table with rows and columns. This is a very natural and common way to work.

R provides a structure called a `data.frame` for working with tabular data, but the package `tidyverse` introduces an improved version of the `data.frame` called a `tibble` (think of it as a kind of data table).

`data.frames` and tibbles have rows and columns. A row represents a set of quantities, such as measurements or observations, that go together in some way. Different rows in a tibble represent different sets of these quantities.

For instance, if I am measuring the height and weight of a number of people, then I would have a row for each person and each row would have a column for the person's name or identity code, a column for their height, and a column for their weight.

If I am measuring the average temperature and average precipitation for a number of cities, then I would have a column for the city, a column for the temperature, and a column for the precipitation.

Each column of a `data.frame` or tibble should correspond to a specific kind of data (integer, floating point, character, factor, date, etc.). A column is a kind of vector, so it has to obey the restrictions that apply to vectors.

To get some experience with tibbles, let's load a couple of data sets that I have prepared. If you have cloned the directory for this document (from https://github.com/gilligan-ees-3310/lab_02_documentation), you can load the datasets, which contain daily weather summaries for Nashville and Chicago, by running the code below:

```
nashville_weather = readRDS('data/nashville_weather.Rds')
chicago_weather = readRDS('data/chicago_weather.Rds')
```

Here is an example of the first few rows of a tibble with weather data for Nashville from 1948–2017:

```
head(nashville_weather)
```

```
## # A tibble: 6 x 6
##            id       date  prcp  tmin  tmax      location
##         <chr>     <date> <dbl> <dbl> <dbl>         <chr>
## 1 USW00013897 1948-01-01  67.8  -1.1  18.3 Nashville, TN
## 2 USW00013897 1948-01-02   0.0  -1.7   1.7 Nashville, TN
## 3 USW00013897 1948-01-03   0.0  -3.3   9.4 Nashville, TN
## 4 USW00013897 1948-01-04   0.0   1.7   8.9 Nashville, TN
## 5 USW00013897 1948-01-05   0.0  -2.2   8.9 Nashville, TN
## 6 USW00013897 1948-01-06   0.0  -1.1  10.6 Nashville, TN
```

There are 6 columns: the weather station ID for the Nashville Airport, the date of the measurement, the daily precipitation (in millimeters), the daily minimum and maximum temperatures (Celsius), and the name of the location.

The tibble also shows the kind of variable that each column represents: `id` and `location` are character data, `date` is Date data, and `prcp`, `tmin`, and `tmax` are double-precision floating point data (i.e., real numbers).

In some ways, a tibble or `data.frame` is like a names list of vectors, where each vector is a column and its name is the name of the column. We can access individual columns using the dollar sign, just as with regular named lists:

```
precipitation = nashville_weather$prcp

head(precipitation)
```

```
## [1] 67.8  0.0  0.0  0.0  0.0  0.0
```

RStudio has a nice feature that lets you examine a tibble or `data.frame` as though it were a spreadsheet. To examine one of R's built-in data sets, which has data on hurricanes in the Atlantic from 1975–2015: `View(dplyr::storms)`

Some other useful functions:

- You can get a list of the names of a named vector, a named list, or the columns of a tibble or `data.frame` with the `names` function: `names(x)`, where x is a vector, list, tibble, or `data.frame`.
- You can get the length of a vector or list with the `length` function, and you can get the number of rows and columns in a tibble using the `dim` function:

```
x = c(1, 2, 3, 4, 5)
print("Length of x is")
```

```
## [1] "Length of x is"
```

```
print(length(x))
```

```
## [1] 5
```

```
print("Dimensions of dplyr::storms is ")
```

```
## [1] "Dimensions of dplyr::storms is "
```

```
dim(dplyr::storms)
```

```
## [1] 10010    13
```

That's 10010 rows and 13 columns. You can also get just the number of rows or the number of columns with `nrow()` and `ncol()`.

The Tidyverse

The "tidyverse" is a collection of packages written by Hadley Wickham to make it easy to work with data frames. Wickham developed an improved kind of data frame that has features that are lacking in the basic R `data.frame`, and he developed a collection of tools for manipulating, analyzing, and graphing data from tibbles and regular `data.frames`.

To use the tidyverse, we need to load the package using R's `library` function. If tidyverse is not installed on your computer, you will get an error message and you will have to run `install.packages("tidyverse")` before you can proceed.

When you load `tidyverse`, it automatically loads a bunch of useful packages for manipulating and analyzing data: `tibble`, `dplyr`, `tidyr`, `purrr`, `readr`, and `ggplot2`.

If you have the `pacman` package installed, it can help you avoid these error messages: after you load `pacman` with `library(pacman)`, then you can load other packages using `p_load(tidyverse)` (you can substituate any other package name for "tidyverse"): pacman will first see whether you have that package on your computer; if you do, pacman will load it, and if you don't pacman will install the package from the Comprehensive R Archive Network (CRAN) and then load it.

In the code below, I will also load the `lubridate` package, which is part of `tidyverse` but is not loaded automatically when you load `tidyverse`. `lubridate` provides useful functions for working with dates, which will come in handy as we work with the weather data.

```
library(tidyverse)
library(lubridate)
# alternately, I could do the following:
# library(pacman)
# p_load(tidyverse, lubridate)
```

One part of the `tidyverse` is the package `dplyr`, which has many useful tools for modifying and manipulating tibbles:

- `select` lets you choose a subset of columns from a tibble
- `rename` lets you rename columns
- `filter` lets you choose a subset of rows from a tibble
- `arrange` lets you sort the rows with respect to the values of different columns
- `mutate` lets you modify the values of columns or add new columns
- `summarize` lets you generate summaries of columns (e.g., the mean, maximum, or minimum value of that column)

- `group_by` and `ungroup` let you perform calculations with grouping (e.g., in combination with summarize, you can group by year to produce separate summaries for each year)
- `bind_rows` to combine multiple tibbles that have the same kinds of columns by stacking one above the other.

There is a lot more, but these functions will be enough to keep us busy for now and they will allow us to do some powerful analysis.

Let's start with `select`: You can select columns to keep or columns to delete.

Here are the first few rows of `nashville_weather`

```
head(nashville_weather)
```

```
## # A tibble: 6 x 6
##            id       date  prcp  tmin  tmax      location
##         <chr>     <date> <dbl> <dbl> <dbl>         <chr>
## 1 USW00013897 1948-01-01  67.8  -1.1  18.3 Nashville, TN
## 2 USW00013897 1948-01-02   0.0  -1.7   1.7 Nashville, TN
## 3 USW00013897 1948-01-03   0.0  -3.3   9.4 Nashville, TN
## 4 USW00013897 1948-01-04   0.0   1.7   8.9 Nashville, TN
## 5 USW00013897 1948-01-05   0.0  -2.2   8.9 Nashville, TN
## 6 USW00013897 1948-01-06   0.0  -1.1  10.6 Nashville, TN
```

Let's get rid of the `id` column, since we don't really care about the ID number, that meteorological agencies use to identify the weather station. and let's get rid of the `location` column because we know that the data set is from Nashville, so having that information repeated on each row is a waste of space. To do this, we just call select, specifying the tibble or `data.frame` to operate on, and then give a list of columns to eliminate, with a minus sign in front of each:

```
x = select(nashville_weather, -id, -location)
head(x)
```

```
## # A tibble: 6 x 4
##         date  prcp  tmin  tmax
##       <date> <dbl> <dbl> <dbl>
## 1 1948-01-01  67.8  -1.1  18.3
## 2 1948-01-02   0.0  -1.7   1.7
## 3 1948-01-03   0.0  -3.3   9.4
## 4 1948-01-04   0.0   1.7   8.9
## 5 1948-01-05   0.0  -2.2   8.9
## 6 1948-01-06   0.0  -1.1  10.6
```

Alternately, instead of telling select which columns to get rid of, we can tell it which columns to keep:

```
x = select(nashville_weather, date, prcp, tmin, tmax)
head(x)
```

```
## # A tibble: 6 x 4
##         date  prcp  tmin  tmax
##       <date> <dbl> <dbl> <dbl>
## 1 1948-01-01  67.8  -1.1  18.3
## 2 1948-01-02   0.0  -1.7   1.7
## 3 1948-01-03   0.0  -3.3   9.4
## 4 1948-01-04   0.0   1.7   8.9
## 5 1948-01-05   0.0  -2.2   8.9
## 6 1948-01-06   0.0  -1.1  10.6
```

We can specify a range of consecutive columns by giving the first and last with a colon between them:

```
x = select(nashville_weather, date:tmax)
head(x)
```

```
## # A tibble: 6 x 4
##         date  prcp  tmin  tmax
##       <date> <dbl> <dbl> <dbl>
## 1 1948-01-01  67.8  -1.1  18.3
## 2 1948-01-02   0.0  -1.7   1.7
## 3 1948-01-03   0.0  -3.3   9.4
## 4 1948-01-04   0.0   1.7   8.9
## 5 1948-01-05   0.0  -2.2   8.9
## 6 1948-01-06   0.0  -1.1  10.6
```

This is a general R thing: we can specify a range of numbers in a similar way:

```
1:10
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

`rename` lets us rename columns:

```
x = rename(nashville_weather, weather_station = id, city = location)
head(x)
```

```
## # A tibble: 6 x 6
##   weather_station       date  prcp  tmin  tmax          city
##             <chr>     <date> <dbl> <dbl> <dbl>         <chr>
## 1     USW00013897 1948-01-01  67.8  -1.1  18.3 Nashville, TN
## 2     USW00013897 1948-01-02   0.0  -1.7   1.7 Nashville, TN
## 3     USW00013897 1948-01-03   0.0  -3.3   9.4 Nashville, TN
## 4     USW00013897 1948-01-04   0.0   1.7   8.9 Nashville, TN
## 5     USW00013897 1948-01-05   0.0  -2.2   8.9 Nashville, TN
## 6     USW00013897 1948-01-06   0.0  -1.1  10.6 Nashville, TN
```

`filter` lets us select only rows that match a condition:

```
x = filter(nashville_weather, year(date) > 2015 & tmax < 0)
head(x)
```

```
## # A tibble: 6 x 6
##            id       date  prcp  tmin  tmax      location
##         <chr>     <date> <dbl> <dbl> <dbl>         <chr>
## 1 USW00013897 2016-01-18   0.0 -11.0  -3.8 Nashville, TN
## 2 USW00013897 2016-01-19   0.0  -9.9  -2.1 Nashville, TN
## 3 USW00013897 2016-01-23   0.0  -8.2  -1.0 Nashville, TN
## 4 USW00013897 2016-02-09   1.3  -6.0  -2.7 Nashville, TN
## 5 USW00013897 2016-02-10   0.5  -7.1  -0.5 Nashville, TN
## 6 USW00013897 2016-12-15   0.0  -6.0  -1.0 Nashville, TN
```

In the code above, I used the `year` function from the `lubridate` package to extract just the year from a date.

One thing that is important to know about making comparisons in `filter` expressions: to specify that two things are equal, you write `==` with two equal signs. A single equal sign is for assigning a value to a variable and two equal signs are for comparisons. You can also use "<=" for less than or equal to and ">=" for greter than or equal to.

```
x = filter(nashville_weather, date == ymd("2016-06-10"))
x
```

```
## # A tibble: 1 x 6
```

```
##            id       date  prcp  tmin  tmax      location
##         <chr>     <date> <dbl> <dbl> <dbl>         <chr>
## 1 USW00013897 2016-06-10     0  15.6  33.3 Nashville, TN
```

In the code above, I used the `ymd` function from `lubridate` to translate a character value "2016-06-10" to the date value for June 10, 2016.

You can combine conditions in `filter` by using `&` to indicate "and" and `|` to indicate "or".

We can sort the rows of a tibble or `data.frame` with the `arrange` function:

```
x = arrange(nashville_weather, desc(tmax), tmin)
head(x,10)
```

```
## # A tibble: 10 x 6
##             id       date  prcp  tmin  tmax      location
##          <chr>     <date> <dbl> <dbl> <dbl>         <chr>
##  1 USW00013897 2012-06-29   0.0  21.1  42.8 Nashville, TN
##  2 USW00013897 1952-07-27   0.0  20.0  41.7 Nashville, TN
##  3 USW00013897 1952-07-28   0.0  22.8  41.7 Nashville, TN
##  4 USW00013897 2012-06-30   0.0  26.7  41.7 Nashville, TN
##  5 USW00013897 1952-06-30  15.2  23.3  41.1 Nashville, TN
##  6 USW00013897 2007-08-16   0.0  24.4  41.1 Nashville, TN
##  7 USW00013897 2012-06-28   0.0  17.8  40.6 Nashville, TN
##  8 USW00013897 1988-07-08   0.0  20.0  40.6 Nashville, TN
##  9 USW00013897 1954-09-05   0.0  20.6  40.6 Nashville, TN
## 10 USW00013897 2012-07-06   0.0  22.2  40.6 Nashville, TN
```

This sorts the rows in descending order of `tmax` (i.e., so the largest values are at the top), and where multiple rows have the same value of `tmax`, then it sorts them in ascending order of `tmin`. Observe the three rows with `tmax` = 41.7, the two rows where `tmax` = 41.1, and the four rows where `tmax` = 40.6.

We can use `mutate` to modify the values of columns or to create new columns. The temperatures in Nashville weather are in Celsius and the precipitation is in millimeters. Let's convert these to Fahrenheit and inches, respectively, and then let's create a `trange` column that will have the difference between the maximum and minimum temperature:

```
x = mutate(nashville_weather, prcp = prcp / 25.4, tmin = tmin * 9./5. + 32,
           tmax = tmax * 9./5. + 32, trange = tmax - tmin)
head(x)
```

```
## # A tibble: 6 x 7
##            id       date     prcp  tmin  tmax      location trange
##         <chr>     <date>    <dbl> <dbl> <dbl>         <chr>  <dbl>
## 1 USW00013897 1948-01-01 2.669291 30.02 64.94 Nashville, TN  34.92
## 2 USW00013897 1948-01-02 0.000000 28.94 35.06 Nashville, TN   6.12
## 3 USW00013897 1948-01-03 0.000000 26.06 48.92 Nashville, TN  22.86
## 4 USW00013897 1948-01-04 0.000000 35.06 48.02 Nashville, TN  12.96
## 5 USW00013897 1948-01-05 0.000000 28.04 48.02 Nashville, TN  19.98
## 6 USW00013897 1948-01-06 0.000000 30.02 51.08 Nashville, TN  21.06
```

Summaries are useful for finding averages and extreme values. Let's find the maximum and minimum temperatures and the most extreme rainfall in the whole data set:

```
x = summarize(nashville_weather, prcp.max = max(prcp), tmin.min = min(tmin), tmax.max = max(tmax))
x
```

```
## # A tibble: 1 x 3
##   prcp.max tmin.min tmax.max
```

```
##      <dbl>    <dbl>    <dbl>
## 1       NA       NA       NA
```

nashville_weather has 25446 rows, but summarize reduces it to a single summary row.

You can use summary to generate multiple summary quantities from a column:

```
x = summarize(nashville_weather, prcp.max = max(prcp), prcp.min = min(prcp))
x
```

```
## # A tibble: 1 x 2
##   prcp.max prcp.min
##      <dbl>    <dbl>
## 1       NA       NA
```

We can also generate grouped summaries:

```
x = ungroup(summarize(group_by(nashville_weather, year(date)), prcp.max = max(prcp), prcp.tot = sum(prcp
head(x)
```

```
## # A tibble: 6 x 3
##   `year(date)` prcp.max prcp.tot
##          <dbl>    <dbl>    <dbl>
## 1         1948     67.8   1179.3
## 2         1949     78.2   1323.6
## 3         1950     84.6   1633.0
## 4         1951     97.3   1483.8
## 5         1952    117.6   1011.3
## 6         1953     47.8   1050.2
```

This provides the maximum one-day precipitation and the total annual precipitation for each year

Note how difficult it is to read that grouped summary expression: the group_by function is inside summarize, which is inside ungroup.

The tidyverse offers us a much nicer way to put these kind of complicated expressions together using what it calls the "pipe" operator, %>%. The pipe operator chains operations together, taking the result of the one on the left and inserting it into the one on the right.

We can use the pipe operator to rewrite the expression above as

```
x = nashville_weather %>% group_by(year(date)) %>% summarize(prcp.max = max(prcp), prcp.tot = sum(prcp))
  ungroup()
head(x)
```

```
## # A tibble: 6 x 3
##   `year(date)` prcp.max prcp.tot
##          <dbl>    <dbl>    <dbl>
## 1         1948     67.8   1179.3
## 2         1949     78.2   1323.6
## 3         1950     84.6   1633.0
## 4         1951     97.3   1483.8
## 5         1952    117.6   1011.3
## 6         1953     47.8   1050.2
```

Now the expression is easier to read: First we group nashville_weather by year, then we summarize it by calculating the maximum daily precipitation and the yearly total for each year, and finally, after we summarize we ungroup.

You can combine any set of the tidyverse functions using the pipe operator, so you could select columns, filter rows, mutate the values of columns, and summarize, using the %>% pipe operator to connect all of

the different operations in sequence.

The final `dplyr` command we're going to look at is `bind_rows`, which lets us combine tibbles:

```
weather = bind_rows(nashville_weather, chicago_weather)
```

This creates a single tibble that has all of the rows from `nashville_weather` on top and all the rows from `chicago_weather` on the bottom. Because the two tibbles have the same columns, the columns are matched up.

We can operate on this combined tibble:

```
x = weather %>%
  mutate(year = year(date), t.range = tmax - tmin) %>%
  group_by(year, location) %>%
  summarize(prcp.max = max(prcp), prcp.tot = sum(prcp), t.range.max = max(t.range)) %>%
  ungroup() %>%
  arrange(year, location)
tail(x)
```

```
## # A tibble: 6 x 5
##    year      location prcp.max prcp.tot t.range.max
##   <dbl>         <chr>    <dbl>    <dbl>       <dbl>
## 1  2015   Chicago, IL     98.3   1168.8        18.3
## 2  2015 Nashville, TN     59.9   1290.7        23.9
## 3  2016   Chicago, IL     50.8   1066.7        23.9
## 4  2016 Nashville, TN     52.8   1086.1        22.2
## 5  2017   Chicago, IL       NA       NA          NA
## 6  2017 Nashville, TN       NA       NA          NA
```