Script for Module 2: Data Wrangling

1 Introduction

[Switch to Slide 1] Welcome to the module on data wrangling. In this module, we'll discuss a several common sources of frustration in data analysis. We'll start off by taking about data cleaning, and then we'll discuss summarizing information about different groups in your dataset, reshaping your data using the split-apply-combine paradigm, merging datasets, and computing a value for each observation or variable in a dataset.

[Switch to slide 2] Throughout this module we'll be using the Hubway dataset. Hubway is a Boston-area bike share company that released a large dataset as part of a visualization challenge. We'll be using two files from the released dataset. *trips.csv* contains more than 550,000 observations, which represent trips taken on a bike in the dataset between 2011 and 2012. For each trip, we are provided the time and date of the start and end of the trip, as well as the duration of the trip in seconds. In Hubway, a bike can be picked up at one station and returned to another one, so we're also provided with the start and end ids of the stations used for the trip, as well as the id of the bicycle that was used for the trip. Hubway has two sorts of users: casual users, who don't pay a monthly fee, and registered users who do. For all the registered users, we are additional provided with their zip code, age, and gender. *stations.csv* contains the id we get for the endpoints of each trip, as well as the name and location, of each Hubway station.

2 Loading the Data and Data Cleaning

[Switch to the R console] In this first part of the lesson, we're going to load our data and perform some basic data cleaning. This is typically the way you'll start when you're handed a dataset. Let's start by loading the datasets. Change your R directory using either "File → Change dir..." on Windows or "Apple-D" on Mac, selecting the DataWrangling folder. It's often more convenient to just keep strings as strings instead of the default in R (which is to load them as factor variables), we we'll pass the `stringsAsFactors=FALSE` argument to the `read.csv` function:

```
stations = read.csv("../Hubway/stations.csv", stringsAsFactors=FALSE)
trips = read.csv("../Hubway/trips.csv", stringsAsFactors=FALSE)
```

Let's summarize our data frames starting with the `str` and `summary` commands applied to our `stations` data frame.

```
str(stations)
# 'data.frame': 95 obs. of  4 variables:
#  $ id  : int  3 4 5 6 7 8 9 10 11 12 ...
#  $ name: chr  "Colleges of the Fenway" "Tremont St. at Berkeley St."
# "Northeastern U / North Parking Lot" "Cambridge St. at Joy St." ...
#  $ lat : num  42.3 42.3 42.3 42.4 42.4 ...
#  $ lng : num  -71.1 -71.1 -71.1 -71.1 -71 ...

summary(stations)
#        id            name                lat             lng
#  Min.   : 3.00   Length:95          Min.   :42.31   Min.   :-71.15
#  1st Qu.:26.50   Class :character   1st Qu.:42.34   1st Qu.:-71.11
#  Median :51.00   Mode  :character   Median :42.35   Median :-71.09
#  Mean   :50.74                      Mean   :42.35   Mean   :-71.09
#  3rd Qu.:74.50                      3rd Qu.:42.36   3rd Qu.:-71.06
#  Max.   :98.00                      Max.   :42.39   Max.   :-71.04
```

The `id` variable holds the station identifies, `name` holds the pretty names of each station, and `lat` and `lng` hold, respectively, the latitude and longitude of the stations. From the `summary` output, we can see that there are no missing values for any of the variables.

Now, let's summarize the trips data frame using the same commands:

```
str(trips)
# 'data.frame': 552073 obs. of  11 variables:
#  $ id               : int  8 9 10 11 12 13 14 15 16 17 ...
#  $ duration         : int  9 220 56 64 12 19 24 7 8 1108 ...
#  $ start_date       : chr  "2011-07-28 10:12:00-04" "2011-07-28 10:21:00-04"
#  "2011-07-28 10:33:00-04" "2011-07-28 10:35:00-04" ...
#  $ start_station    : int  23 23 23 23 23 23 23 23 23 47 ...
#  $ end_date         : chr  "2011-07-28 10:12:00-04" "2011-07-28 10:25:00-04"
#  "2011-07-28 10:34:00-04" "2011-07-28 10:36:00-04" ...
#  $ end_station      : int  23 23 23 23 23 23 23 23 23 40 ...
#  $ bike_nr          : chr  "B00468" "B00554" "B00456" "B00554" ...
#  $ subscription_type: chr  "Registered" "Registered" "Registered"
#  "Registered" ...
#  $ zip_code         : chr  "97217" "02215" "02108" "02116" ...
#  $ birth_date       : int  1976 1966 1943 1981 1983 1951 1971 1971 1983 1994 ...
#  $ gender           : chr  "Male" "Male" "Male" "Female" ...

summary(trips)
#        id            duration         start_date        start_station
#  Min.   :     8   Min.   :       0   Length:552073      Min.   : 3.00
#  1st Qu.:154977   1st Qu.:     411   Class :character   1st Qu.:22.00
#  Median :315388   Median :     687   Mode  :character   Median :38.00
#  Mean   :312921   Mean   :    1695                      Mean   :37.72
```

```
#  3rd Qu.:468523   3rd Qu.:    1204                      3rd Qu.:52.00
#  Max.   :623517   Max.   :11994458                      Max.   :98.00
#                                                        NA's   :15
#    end_date          end_station      bike_nr          subscription_type
#  Length:552073      Min.   : 3.00   Length:552073      Length:552073
#  Class :character   1st Qu.:22.00   Class :character   Class :character
#  Mode  :character   Median :38.00   Mode  :character   Mode  :character
#                     Mean   :37.68
#                     3rd Qu.:52.00
#                     Max.   :98.00
#                     NA's   :45
#    zip_code           birth_date        gender
#  Length:552073      Min.   :1932     Length:552073
#  Class :character   1st Qu.:1969     Class :character
#  Mode  :character   Median :1979     Mode  :character
#                     Mean   :1976
#                     3rd Qu.:1985
#                     Max.   :1995
#                     NA's   :199113
```

In this data frame, `id` is an identifier for each trip, `duration` is the duration in of a trip in seconds (we can see that the median trip length is between 11 and 12 minutes), `start_date` and `end_date` are timestamps of the start and end of the trip, `start_station` and `end_station` are the identifiers for the stations of the start and end of the trip, `bike_nr` is an identifier for the physical bicycle used for the trip, `subscription_type` indicates whether the person who took the trip is a casual or registered user, `zip_code` is the zip code of registered users, `birth_date` is the birth year of registered users, and `gender` is the gender of registered users.

Visually, we can identify a few issues with this data:

- There are 15 trips for which `start_station` is missing and 45 trips for which `end_station` is missing.
- There is at least one big outlier in the duration column, which is many orders of magnitude larger than the mean. Such outliers might throw off various statistical analyses we might perform.
- The timestamps are currently stored as strings, which will make them hard to manipulate.

We'll handle each of these data issues. First, let's deal with the missing start and end stations. Since this affects a very small number of all the trips, it probably just makes sense to remove these observations. We can use the `subset` function to obtain a limited version of the data frame. We can use the `is.na` function to check if a variable is missing a value in R.

```
trips = subset(trips, !is.na(start_station) & !is.na(end_station))
```

In this command, we used to exclamation mark to negate the output of the `is.na` function, and we can used the ampersand (`&`) to limit to observations where `start_station` is not missing *and* `end_station` is not missing. If we instead wanted to perform a logical "or" operation, we could have used the pipe symbol (`|`). We overwrote the old `trips` variable. Let's check the number of observations in this new data frame using the `nrow` function:

```
nrow(trips)
# [1] 552020
```

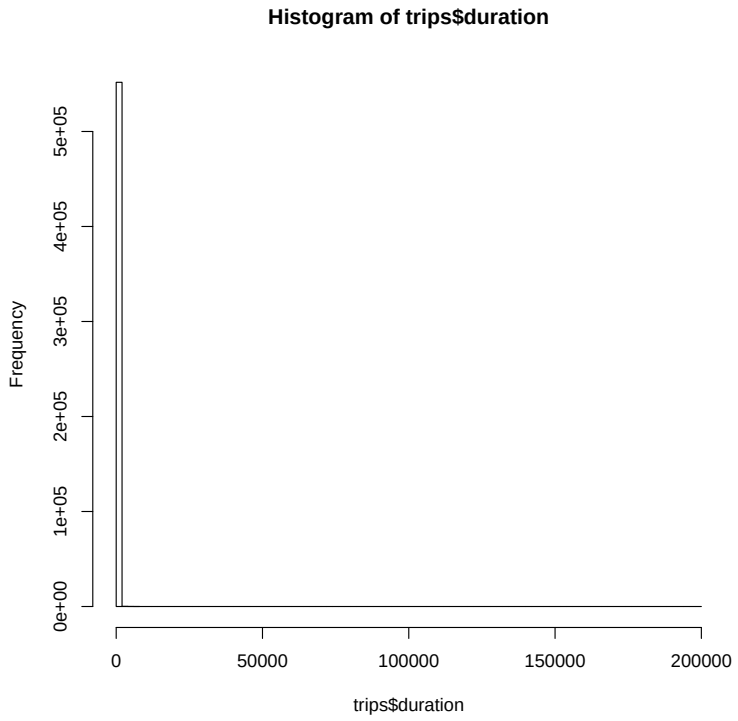
We started with 552,073 observations and now have 552,020, so we dropped 53 observations with that operation.

Next, let's turn our attention to the `duration` variable. At a glance, it's tough to decode a variable that's encoded in seconds, because the numbers are so large. Let's start by converting this variable into minutes by dividing by 60:

```
trips$duration = trips$duration / 60
```

Earlier, we noticed that the `duration` variable had at least one outlier. Let's see how this outlier affects a simple visualization of the distribution of trip durations via a histogram. We'll use 100 breaks for our histogram to get a finer-granularity breakdown of the durations:

```
hist(trips$duration, breaks=100)
```



Because of the outliers, we really can't tell anything from this histogram. We can subset our data frame to remove all the trips with duration exceeding one day. Depending on the type of analysis we're doing, this might be the wrong decision — it might be better to cap the trip duration at a maximum of one day, log-transform the durations, or do nothing at all.

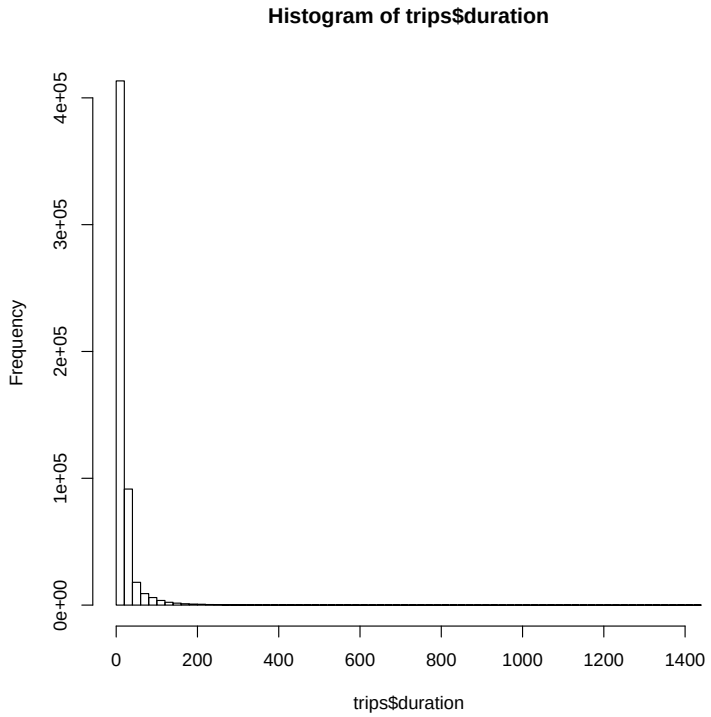
```
trips = subset(trips, duration <= 24*60)
```

Again we can check the number of trips using the `nrow` function:

```
nrow(trips)
# [1] 551611
```

As we can see, we've removed an additional 409 trips that had duration in excess of one day. Let's check how this affected our simple visualization of the distribution of trip durations:

```
hist(trips$duration, breaks=100)
```



Now, the x-axis is much less compressed, and we can better understand the distribution of trips. As we can see, trips still tend to be quite short.

We're now ready to tackle our last data cleaning problem, which is that our timestamps are stored as strings. We really want to load them into a timestamp data structure so we can access important information like the day of the week, which couldn't really be extracted with string processing. In R, we can use the `strptime` function to convert text to a timestamp data structure. To learn more about this function, we can run:

```
?strptime
```

From this help page, we can see that the first argument is going to be the object to be converted, and the second argument is a format identifier describing how the timestamp is stored in the string. Reading the specification of that identifier and reviewing the sample

timestamps we saw when we ran `str(start_date)`, we see that the four-digit year will be matched by `%Y`, the numeric month will be matched by `%m`, the day of the month will be matched by `%d`, the hour of the day will be matched by `%H`, the minute of the hour will be matched by `%M`, and the second of the hour will be matched by `%S`. Though our timestamps also include time zone information, we can ignore this information since they will all be the same. Putting it together, we can run the following command to parse the `start_date` variable with:

```
trips$start_date = strptime(trips$start_date, "%Y-%m-%d %H:%M:%S")
```

We can take a look at what we've created using the `str` function:

```
str(trips$start_date)
#  POSIXlt[1:551611], format: "2011-07-28 10:12:00" "2011-07-28 10:21:00" ...
```

We can see that we've created an object of type `POSIXlt`. To learn more about this we can again check out the help pages:

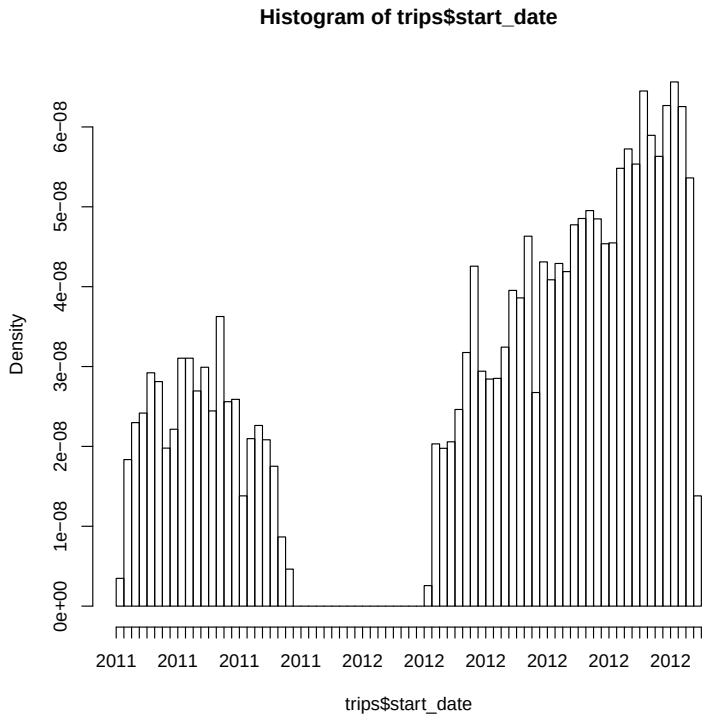
```
?POSIXlt
```

From this help page, we can see the various values we can extract from the `POSIXlt` object. For instance, we can read that `trips`*start_date*`wday` will contain the day of the week of the start of each trip, encoded as the numbers 0–6, starting from Sunday. Before it would have been hard to get the breakdown of trips by day of the week, but now it's a simple application of the `table` command:

```
table(trips$start_date$wday)
#     0     1     2     3     4     5     6
# 73956 82261 77222 77616 80061 83601 76894
```

We can see that Friday (day 5) is the most common for trips and Sunday (day 0) is the least common. However, the frequency of trips by day of the week is quite level. We can also use the `hist` function to show the distribution of trips by date:

```
hist(trips$start_date, breaks=100)
```



Though the x-axis labels are non-ideal, we can see that there's a period of time where there were no trips at the end of 2011 and beginning of 2012. This makes sense because Hubway didn't operate during these winter months. The steady increasing during 2012 suggests Hubway is gaining traction over time. Again, this plot would have been very difficult to make without converting our text timestamps.

Finally, we'll convert over the `end_date` variable:

```
trips$end_date = strptime(trips$end_date, "%Y-%m-%d %H:%M:%S")
```

[Switch to Slide 3] There are a lot of situations in which data cleaning is important, and it's tedious to review them so we'll go over a few common data cleaning needs and commands. As we saw already `strptime` is a useful function for reading in date/time information. To find/replace patterns in a string or otherwise manipulate strings in R, the `grepl`, `gsub`, and `strsplit` functions are useful, as well as the `stringr` package. To convert between strings, numeric values, and factors, the `as.character`, `as.numeric`, and `as.factor` functions are helpful. A common gotcha comes up when converting a factor with numeric level names into those level names (for instance, a factor with level names '2", '2.5", and '3" getting converted into the numbers 2, 2.5, and 3). In this case, you would run the command `as.numeric(as.character(fac))` instead of `as.numeric(fac)`. Finally, as we saw earlier, we use the `is.na` function to check for missing values, not the `== NA` syntax that might seem more natural.

3 Leveraging built-in functions to summarize data

[Switch to Slide 4] We're now going to switch gears and begin talking about obtaining summary information for groups of observations in our data. We're going to be using `tapply` a number of times during this module, so even though you were introduced to this function in the Introduction to R module we'll review its use here. Let's consider the small dataset displayed on the slide, where observations represent people, and for each person we have a country value and an income value. Rows in the dataset are colored based on the country of the individual.

We're interested in computing the average income of individuals from each country. The `tapply` function allows us to summarize our data by group. It takes three arguments, and it operates by grouping the first argument by the second argument and then applying the third argument. The first two arguments are data, in this case the `income` and `country` variables, and the third argument is a function, in this case the built-in `mean` function. We can see that `tapply` first breaks up our `income` data into three sets of two values each, based on the `country` values, and then applies the `mean` function to each set, resulting in one average value for each country.

[Switch to the R console] Using our `trips` data frame, let's consider a new question: "What is the average duration of a trip by subscription type?" To remember the variable names of our `trips` data frame, we can use the `names` function:

```
names(trips)
#  [1] "id"                "duration"          "start_date"
#  [4] "start_station"     "end_date"          "end_station"
#  [7] "bike_nr"           "subscription_type" "zip_code"
# [10] "birth_date"        "gender"
```

We can use `table` to obtain a breakdown of the `subscription_type` variable, which is the variable we're going to use to group our data:

```
table(trips$subscription_type)
#     Casual Registered
#     196044     355567
```

We have a pretty large number of each group. How can we use `tapply` to obtain the average trip duration for each type of user? *[Get answer from student in the class]*

```
tapply(trips$duration, trips$subscription_type, mean)
#     Casual Registered
#   39.56056   11.72258
```

Interesting — we can see that registered users on average take trips of roughly 12 minutes, while casual users have an average trip length of 40 minutes. We can obtain the full summary of the trip duration for each group just by using `summary` instead of `mean` as the third argument to `tapply`:

```
tapply(trips$duration, trips$subscription_type, summary)
# $Casual
#    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#    0.05   12.25   20.63   39.56   34.80 1440.00
#
# $Registered
#     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
#    0.000    5.767    8.850   11.720   13.770 1417.000
```

We can see that 25th, 50th, and 75th percentile trip durations are all much longer for casual users than for registered users.

[Switch to Slide 5] Now, we'll have a quick in-class exercise make sure everybody's comfortable with using `tapply` with built-in functions. Using `tapply`, determine the average and sum of trip durations by gender. Further, determine the average trip duration by day of the week at the start of the trip and by month of the year of the end of the trip. For those of you who finish early, there's also a bonus exercise — determine the proportion users who are casual users by start location, and determine the start locations with the lowest and highest proportions. A few tips for this bonus: use `==` instead of `=` to check for the equality between two values. Also, the `mean` function of applied to a vector of TRUE/FALSE values will return the proportion that are true — it treats TRUE and 1 and FALSE as 0 in this calculation.

[During the in-class exercise, circulate to provide guidance to students.]

[Switch to the R console] Now, let's quickly go over the answers to the exercises. *[Wherever possible, get students give answers.]*

For the first part, we want to operate on the duration of trips, grouped by the gender of the person making the trip. We'll use the `mean` and `sum` functions, yielding:

```
tapply(trips$duration, trips$gender, mean)
#            Female     Male
# 39.56056 13.47773 11.14336
tapply(trips$duration, trips$gender, sum)
#          Female    Male
# 7755610 1189086 2979078
```

We see that we're actually getting three values: one for male, one for female, and one for casual users, since we don't know their genders. Casual users by far have the longest average trips, but we see that females have longer average trip durations than males. However, their summed durations are much smaller because more trips are completed by males than females.

For the next two questions, we're operating on the same data (the `duration` variable) and we use the `mean` function for each operation, but we're now grouping by two the day of week and month of year. From `?POSIXlt`, we know that we use `wday` to get the day of week and `mon` to get the month of the year, yielding the following calls:

```
tapply(trips$duration, trips$start_date$wday, mean)
```

```
#        0        1        2        3        4        5        6
# 31.38572 19.36453 17.17478 17.52617 16.69096 19.37939 30.77810
tapply(trips$duration, trips$end_date$mon, mean)
#        2        3        4        5        6        7        8        9
# 20.54531 18.49368 18.61186 17.35814 20.13478 24.69731 21.83348 28.36619
#       10       11
# 21.39851 16.87500
```

We can see that trips have much longer durations on weekends (presumably because there are more casual users on weekends). They seem to be slightly longer in the summer than the spring. There was no average value computed for January and December because there were no trips in those months.

On the bonus exercise, as suggested by the hint, we want to operate on the TRUE/FALSE vector of whether each trip was taken by a casual user. We want to group by the start location, which is the `start_station` variable and use the `mean` function to compute the proportion of users from that start location who are casual. Finally, we want to see the largest and smallest values, so we can use the `sort` function to sort from smallest to largest proportion. Putting it all together we have:

```
sort(tapply(trips$subscription_type == "Casual", trips$start_station, mean))
#        51        14        25        11        39        26        48
# 0.1488266 0.1532551 0.1866667 0.1883225 0.1890872 0.1940739 0.1975309
#        62        57        16        22        64        63        83
# 0.2008329 0.2056638 0.2074623 0.2079361 0.2120192 0.2161722 0.2229965
# ...
# ...
#        70        94        73        15        98        71        93
# 0.5681217 0.5719844 0.5815972 0.5835718 0.5947955 0.5965909 0.6000000
#        65        74        97        92
# 0.6697080 0.6745531 0.7106227 0.7215190
```

We can see that station 92 has 72.2% casual users, while station 51 has just 14.9%, suggesting there's different "types" of stations with respect to user registration status.

4 Leveraging user-defined functions to summarize data

Great — now we're experts at using `tapply` with built-in functions. But sometimes that's not enough. Let's say we wanted to figure out the two busiest days of the week for each type of user (using `start_date` to compute the day of the week). This is easy enough with a two-way frequency table:

```
table(trips$subscription_type, trips$start_date$wday)
#                    0     1     2     3     4     5     6
#   Casual       44449 24092 18226 19252 19776 25232 45017
#   Registered   29507 58169 58996 58364 60285 58369 31877
```

From the table, we can read that the two busiest days for casual users are Saturday (day 6) and Sunday (day 0). For registered users, the most common are Thursday (day 4) and Tuesday (day 2). However, there are a few problems with this approach. First, we haven't gotten the values programmatically, which would be necessary if we wanted to use them later in our code. Secondly, this would be impractical if we had asked for most common 2 days by start location, since there are 95 stations.

[Switch to Slide 6] Let's think about how we might approach this with the `tapply` function. The data in the computation is the day of the week of the trip (the `wday` variable on the slide), and we want to group by the subscription type of the user (the `type` variable on the slide). After grouping `wday` by `type`, all that remains is to apply some a function, which we call `get.top.2` on the slide, that computes the two most common days from each group. Unfortunately, no such built-in function exists in R, so we'll need to build our own user-defined version of this function.

[Switch to the R console] Let's start by thinking about how to find the two most common days in the whole data frame. We can output the frequency table by day for the whole data frame with the `table` function:

```
table(trips$start_date$wday)
#     0     1     2     3     4     5     6
# 73956 82261 77222 77616 80061 83601 76894
```

We can read that our most common days in the whole data frame are Friday (day 5) and Monday (day 1). Let's see if we can now get that programmatically. We're going to want to sort our table since we need the biggest values. As we saw in the bonus exercise, we can use the `sort` function, but we'll want the largest values at the beginning so we'll use the `decreasing=TRUE` argument. We'll save the sorted result into a variable called `tab`:

```
tab = sort(table(trips$start_date$wday), decreasing=TRUE)
tab
#     5     1     4     3     2     6     0
# 83601 82261 80061 77616 77222 76894 73956
```

We just want the most common two days, so we can access the two most common values:

```
tab[1:2]
#     5     1
# 83601 82261
```

The actual values here are counts of the numbers of trips for each day, but we're interested in the day itself. We can see this being outputted, and we can access it with the `names` function:

```
names(tab[1:2])
# [1] "5" "1"
```

The quotes around the results mean that they're strings instead of numbers, so we can convert them to numbers with the `as.numeric` function:

```
as.numeric(names(tab[1:2]))
# [1] 5 1
```

[In a new file called top2.R] Now that we have code to compute the top-2 days for the whole data frame, we'll want to write our `get.top.2` function for use in `tapply`. Typically, it's a good idea to write and store functions we develop in a separate script file, which by convention has a .R file extension. We'll write this function in a new file called `top2.R`, and we'll go over different ways to load this code into the R console. Recall that our function takes as input a vector of days of the week, and is expected to return the two most common days in that vector. We'll call that argument `wdays`. We'll just copy our code that we used to compute the most common two days across the whole data frame, and we'll replace `trips`$start_date$`wday` with our argument `wday`. Finally, we'll add a `return` statement around the code that extracts the two most common days, as those are the values that we want our function to return.

```
get.top.2 = function(wdays) {
  tab = sort(table(wdays), decreasing=TRUE)
  return(as.numeric(names(tab[1:2])))
}
```

[Switch to the R console] Now that we've saved our code in the file `top2.R`, we have two options to load it into our R console so we can start using our `get.top.2` function. Either we could select the code and hit Control-R in Windows or Command-Enter on a Mac, which copies the code to the terminal, or we could use the `source` function to load all the code in the file into our console:

```
source("top2.R")
```

To test out our function, we can start by calling it on the `wday` values from the whole data frame:

```
get.top.2(trips$start_date$wday)
# [1] 5 1
```

This is the expected result for the whole data frame. Now that we've defined `get.top.2`, we can use it just like any of the built-in functions. We can now use `get.top.2` for the original task, which was computing the two most common days of the week for each subscription type:

```
tapply(trips$start_date$wday, trips$subscription_type, get.top.2)
# $Casual
# [1] 6 0
#
# $Registered
# [1] 4 2
```

We can check our results using the `table` command:

```
table(trips$subscription_type, trips$start_date$wday)
#                   0     1     2     3     4     5     6
#   Casual      44449 24092 18226 19252 19776 25232 45017
#   Registered 29507 58169 58996 58364 60285 58369 31877
```

This confirms that we've successfully `tapply` with our user-defined function `get.top.2`, because days 6 and 0 are indeed the most common for casual users, and days 4 and 2 are most common for registered users.

Though we developed the `get.top.2` function to identify the two most common travel days broken down by registration status, we can easily use this function in other contexts. For instance, we can use it determine the two most common days of the week for trips from each start location:

```
tapply(trips$start_date$wday, trips$start_station, get.top.2)
# $`3`
# [1] 1 3
#
# $`4`
# [1] 0 6
# ...
# ...
# $`97`
# [1] 0 6
#
# $`98`
# [1] 0 5
```

Here, we see that some start locations, like location 3, are most associated with weekday travel, while other like ids 4, 97, and 98 are most associated with weekend travel.

You could imagine wanting to actually build a new data frame out of this – it would have one variable for the station id and two more variables for the top 2 days. We will see how to do that in the next section, when we'll talk more formally about the split-apply-combine paradigm.

[Switch to Slide 7] Before we get to split-apply-combine, we'll have a quick in-class exercise make sure everybody's comfortable with writing and using user-defined functions. Hubway charges a fee for any trip at least 30 minutes long. Use `tapply` to compute the proportion of trips from each start location that is at least 30 minutes long. For this part, write a function called `prop.above.30` that inputs a vector of trip durations and outputs the proportion of them that are 30 minutes or greater. If you're having trouble getting started on the code, you can use the template code in `exercise2_start.R`. For those of you who finish early, there's also a bonus exercise — compute the most common subscription type (registered, casual, or a tie) for each start/end location pair. Treat a trip from A to B as different from a trip from B to A. How many of each type of pair are there? As a hint, you can use the `paste` function for building a vector of start/end pairs. You can use `if`, `else if`, and `else` as displayed on the slides.

[During the in-class exercise, circulate to provide guidance to students.]

[Switch to exercise2_start.R] Now, let's quickly go over the answers to the exercises. *[Wherever possible, get students give answers.]*

In the first part, we want to write a function that takes in a vector of trip durations and return the proportion of those durations that are 30 or larger. We can use the `mean` function to compute the proportion of durations greater than or equal to 30:

```
prop.above.30 = function(durations) {
  return(mean(durations >= 30))
}
```

[Switch to the R console] Now, we can load our `prop.above.30` function using the `source` function and compute the proportion of long trips from each start location:

```
source("exercise2_start.R")
tapply(trips$duration, trips$start_station, prop.above.30)
#          3          4          5          6          7          8          9
# 0.08548931 0.07604052 0.09934721 0.13089947 0.15352287 0.16573661 0.13909260
#         10         11         12         13         14         15         16
# 0.09434479 0.06709917 0.08601736 0.06602043 0.04233026 0.35317687 0.07690710
# ...
# ...
#         88         89         90         91         92         93         94
# 0.06890756 0.10971787 0.09174312 0.05847076 0.60759494 0.26666667 0.12451362
#         95         96         97         98
# 0.06959707 0.08097166 0.25641026 0.16171004
```

This example was good practice with creating user-defined functions, but we could have avoided writing our own function by using `tapply` in a different way. We could have instead passed a vector of TRUE/FALSE values to `tapply` as the first argument as used the `mean` function:

```
tapply(trips$duration >= 30, trips$start_station, mean)
#          3          4          5          6          7          8          9
# 0.08548931 0.07604052 0.09934721 0.13089947 0.15352287 0.16573661 0.13909260
#         10         11         12         13         14         15         16
# 0.09434479 0.06709917 0.08601736 0.06602043 0.04233026 0.35317687 0.07690710
# ...
# ...
#         88         89         90         91         92         93         94
# 0.06890756 0.10971787 0.09174312 0.05847076 0.60759494 0.26666667 0.12451362
#         95         96         97         98
# 0.06959707 0.08097166 0.25641026 0.16171004
```

[Switch to exercise2_start.R] For the bonus exercise, we'll write a function called `most.common`, which is passed a vector of subscription types and returns the most common type. As suggested in the slides, we'll use `if`, `else if`, and `else` to determine which of the three cases

holds for a passed vector, using the `sum` function over logical vectors to determine the number of `TRUE` values:

```
most.common = function(x) {
  if (sum(x == "Registered") > sum(x == "Casual")) {
    return("Registered")
  } else if (sum(x == "Registered") < sum(x == "Casual")) {
    return("Casual")
  } else {
    return("Tie")
  }
}
```

[Switch to the R console] Now, we can load our function and test it on the whole `trips` data frame to make sure it operates properly:

```
source("exercise2_start.R")
most.common(trips$subscription_type)
# [1] "Registered"
```

The last part of this bonus is to compute the groups for `tapply`. As suggested in the hint, we can use the `paste` function to combine the `start_station` and `end_station` variables together:

```
pair.results = tapply(trips$subscription_type,
                      paste(trips$start_station, trips$end_station), most.common)
pair.results
#        10 10        10 11        10 12        10 13        10 14        10 15
#     "Casual" "Registered" "Registered" "Registered" "Registered"     "Casual"
#        10 16        10 17        10 18        10 19        10 20        10 21
# "Registered"     "Casual"     "Casual" "Registered" "Registered" "Registered"
# ...
# ...
#        98 81        98 84        98 87        98 88         98 9        98 90
# "Registered" "Registered"     "Casual" "Registered"     "Casual" "Registered"
#        98 91        98 94        98 95        98 97        98 98
#        "Tie"     "Casual" "Registered"     "Casual"     "Casual"
```

To compute the number of each outcome, we can use the `table` function on the `pair.results` variable:

```
table(pair.results)
#     Casual Registered        Tie
#       2445       4511        399
```

As we would expect, most pairs have registered users as the most common travelers, but there are still more than 2,000 for which casual users are the most common.

5 Split-Apply-Combine

[Switch to Slide 8] The `trips` data frame has rows that represent bike trips, but we saw last time how we could split the data up based on some other variable (in our case `start_station`) and compute multiple values for that group (in our case the first and second most common day of the week for travel). We would like to combine this into a data frame with rows representing start stations, and columns representing the variables we built for each start station. This methodology of splitting a dataset into subsets, computing summary values for each subset, and finally combining into a new data frame is called split-apply-combine.

The first step of split-apply-combine is splitting the data into subsets. In this step we use the `split` function, which inputs as its first argument a data frame (in our case the `trips` data frame) and a value used to split the data frame (in this slide, a variable called `station`). The output of this function is a data structure called a `list`, which stores one new data frame for each value of the `station` variable. These new data frames are subsets of the original data frame for which all observations have the same `station value`. On the slide, `split` returns a list of two data frames. The first one contains the four observations in `trips` with a `station` value of 1, and the second data frame contains the six observations in `trips` with a `station` value of 2. These outputted data frames contain all the same variables as `trips`.

[Switch to Slide 9] The next step in split-apply-combine is to apply a function to each of the data frames stored in our list. This function will input and output a data frame. In our case, we want a function called `get.top.2.df`, which inputs the subset for a specific station and returns a 1-row data frame containing the station identifier and the most common two days of the week for trips leaving that station. The `lapply` function takes a list as its first argument and a function as its second argument. It returns a list containing the result of apply its second argument to each element stored inside its first argument. In our case, `lapply` will return a list of 1-row data frames.

[Switch to Slide 10] The final step of split-apply-combine is to combine this processed list into a final data frame. Normally, we would do this by calling the `rbind` function and passing it each of the elements in the processed list, which we called `spl2` on the slide. The access a particular element of a list you use double square brackets instead of single square brackets, so this would look like `rbind(spl2[[1]], spl2[[2]], ...)`. However, we often won't know how many elements there are in the processed list, and it would be annoying to type them all out. Luckily, there's a function called `do.call` that saves us a lot of typing. `do.call` calls its first argument on all of the elements stored in its second argument, which much be a list. Therefore, can use `do.call` with the `rbind` function to combine all our processed data frames into a final data frame.

[Switch to the R console] Let's use split-apply-combine to compute our data frame of the top two days of the week for each start location. We'll start by splitting our `trips` data frame based on the `start_station` variable:

```
spl = split(trips, trips$start_station)
```

`spl` is a list containing one data frame for each start station in our `trips` data frame. We can use the `summary` function to look at individual data frames stored in `spl`:

```
summary(spl[[1]])
#        id            duration          start_date
#  Min.   :    85   Min.   :   0.050   Min.   :2011-07-28 12:31:00
#  1st Qu.:142639   1st Qu.:   7.267   1st Qu.:2011-11-11 16:30:00
#  Median :273336   Median :  11.850   Median :2012-05-25 15:12:00
#  Mean   :288185   Mean   :  18.167   Mean   :2012-04-14 10:38:13
#  3rd Qu.:426229   3rd Qu.:  19.100   3rd Qu.:2012-07-30 18:41:00
#  Max.   :623416   Max.   :1143.717   Max.   :2012-10-01 19:48:00
#
#  start_station    end_date                    end_station
#  Min.   :3     Min.   :2011-07-28 12:32:00   Min.   : 3.00
#  1st Qu.:3     1st Qu.:2011-11-11 16:40:00   1st Qu.:14.00
#  Median :3     Median :2012-05-25 15:16:00   Median :32.00
#  Mean   :3     Mean   :2012-04-14 10:56:24   Mean   :30.58
#  3rd Qu.:3     3rd Qu.:2012-07-30 18:59:00   3rd Qu.:45.00
#  Max.   :3     Max.   :2012-10-01 20:00:00   Max.   :98.00
#
#    bike_nr          subscription_type    zip_code          birth_date
#  Length:4445        Length:4445        Length:4445        Min.   :1945
#  Class :character   Class :character   Class :character   1st Qu.:1973
#  Mode  :character   Mode  :character   Mode  :character   Median :1981
#                                                           Mean   :1978
#                                                           3rd Qu.:1985
#                                                           Max.   :1994
#                                                           NA's   :1307
#     gender
#  Length:4445
#  Class :character
#  Mode  :character
```

This looks quite similar to the summary output for the whole trips data frame, except it's much smaller (4445 observations) and all of the observations have the same value for the `start_station` variable: 3. Other elements in `spl` will be similar but will represent a different start station.

The next step is to write our `get.top.2.df` function. As usual, we'll write our user-defined function in a script. This function is quite similar to `get.top.2`, which we wrote before, but it inputs data frames instead of vectors of days of the week, and it outputs 1-row data frames instead of a vector of the two most common days. *[Switch to top2.R and develop new function together, starting by copying get.top.2 and changing its name]*

```
get.top.2.df = function(dat) {
  tab = sort(table(dat$start_date$wday), decreasing=TRUE)
  top2 = as.numeric(names(tab[1:2]))
  return(data.frame(start_station=dat$start_station[1], day1=top2[1],
                    day2=top2[2]))
}
```

We changed a few things here. We called the input `dat` to remind ourselves that it's a data frame, we had to extract the day of the week from the `start_date` variable in the passed data frame, and we had to construct our data frame to output. Because all observations in `dat` have the same `start_station` value, we simply extracted the first value for our returned data frame.

[Switch to the R console] Let's load up our function using the `source` function:

```
source("top2.R")
```

Now, let's test out the function on the data frame stored in our `spl` variable, again using double square brackets to access that element.

```
get.top.2.df(spl[[1]])
#   start_station day1 day2
# 1             3    1    3
```

So the function correctly extracted the station id corresponding to this subset of our `trips` data frame as 3, and it identified the most common days of the week for trips as Monday (day 1) and Wednesday (day 3). We can verify this with with `table` command:

```
table(spl[[1]]$start_date$wday)
#   0   1   2   3   4   5   6
# 443 739 714 730 689 713 417
```

This confirms that we got the right result. We can run it on one more subset as well:

```
get.top.2.df(spl[[2]])
#   start_station day1 day2
# 1             4    0    6
table(spl[[2]]$start_date$wday)
#    0    1    2    3    4    5    6
# 1338 1158 1013 1138 1059 1208 1279
```

For start station 4, `get.top.2.df` identified that Sunday (day 0) and Saturday (day 6) are the most common days of the week for trips, and we confirmed this with the `table` function.

Now that we feel comfortable that `get.top.2.df` is working as we intended, we'll apply it to all of the data frames stored in `spl` with the `lapply` function, storing the result in a variable called `spl2`:

```
spl2 = lapply(spl, get.top.2.df)
```

Now, we can check out the values stored in `spl2`, which is a list, using the double square bracket notation:

```
spl2[[1]]
#   start_station day1 day2
# 1             3    1    3
spl2[[2]]
#   start_station day1 day2
# 1             4    0    6
```

As we can see, these match the computations we did just a moment ago where we applied `get.top.2.df` to the elements individually.

Finally, we'll use the `do.call` function to call the `rbind` function will all the elements stored in `spl2`, storing the result in a variable called `station.info`:

```
station.info = do.call(rbind, spl2)
```

We can use the `head` function to check out the first few rows of this data frame:

```
head(station.info)
#   start_station day1 day2
# 3             3    1    3
# 4             4    0    6
# 5             5    1    5
# 6             6    1    2
# 7             7    4    5
# 8             8    0    6
```

Finally, we can use the `table` command to determine the number of stations for which each day of the week is the most common and second most common day for trips:

```
table(station.info$day1)
#  0  1  2  3  4  5  6
# 18 17  3  3 10 15 29

table(station.info$day2)
#  0  1  2  3  4  5  6
# 24 12  7  4 11 19 18
```

As we can see, Saturday (day 6) has the largest number of stations where it's the most common day, and Sunday (day 0) has the largest number of stations where it's the second most common day.

[Switch to Slide 11] To get a bit of practice with split-apply-combine, let's do an in-class exercise. From the `trips` data frame, create a data frame called `bicycle.info`, where reach row corresponds to a bicycle. Include a variable called `bike.nr`, the identifier of the bicycle, `mean.duration`, the mean duration of the trips taken on this bike, `sd.duration`, the standard deviation of trips taken on this bike, and `num.trips`, the number of trips taken on this bike. If you're having trouble getting started on the code, you can use the template code in `exercise3_start.R`. For those of you who finish early, there's also a bonus exercise

— add a variable called `multi.day` that has the number of trips on each bike that started on one day and finished on another, `common.start`, the most common start location for each bike, and `comment.end`, the most common end location for each bike.

[During the in-class exercise, circulate to provide guidance to students.]

[Switch to exercise3_start.R] Now, let's quickly go over the answers to the exercises. *[Wherever possible, get students give answers.]*

In the first part, we want to write a function called `process.bike` that takes as input a subset of the `trips` data frame corresponding to a single bike and return a 1-row data frame with summary information about that bike. In `exercise3_start.R`, we already have the skeleton of that function, which extracts the `bike.nr` and `mean.duration` variables from the passed data frame. We can compute `sd.duration` using the `sd` function, which computes standard deviations, and we can compute the `num.trips` variable using the `nrow` function, which computes the number of rows in the passed data frame. Putting it all together we, have:

```
process.bike = function(x) {
  bike.nr = x$bike_nr[1]
  mean.duration = mean(x$duration)
  sd.duration = sd(x$duration)
  num.trips = nrow(x)
  return(data.frame(bike.nr, mean.duration, sd.duration, num.trips))
}
```

[Switch to the R console] Now, let's load our function using the `source` function and split up our data frame using the `split` function, apply `process.bike` to each subset using the `lapply` function, and combine it using the `do.call` function:

```
source("exercise3_start.R")
spl = split(trips, trips$bike_nr)
spl2 = lapply(spl, process.bike)
bicycle.info = do.call(rbind, spl2)
```

Now we can use `head` to look at the first few rows of our new bike-specific data frame:

```
head(bicycle.info)
#                       bike.nr mean.duration sd.duration num.trips
# ? (0x241EC230) ? (0x241EC230)      33.17014   60.414725       192
# ? (0x67D8478F) ? (0x67D8478F)      25.73062   40.001034       129
# ? (0xA533D104) ? (0xA533D104)      38.82852  107.237058       142
# B00000                 B00000      11.86333    7.971572        45
# B00001                 B00001      18.05180   32.528162       933
# B00002                 B00002      23.10213   60.710399       703
```

[Switch to exercise3_start.R] For the bonus, we want to expand this function to count the number of multi-day trips taken by a particular bike, as well as the most common start and end locations. To compute if `start_date` and `end_date` are on different days, we can

check if the day of the year, `yday`, or the year, `year`, differ between the two dates. For the most common start and end locations, we can sort a frequency table in decreasing order and take the first element in the names of the table, as we did when we were computing the two most common days of the week. In code, this is:

```
process.bike = function(x) {
  bike.nr = x$bike_nr[1]
  mean.duration = mean(x$duration)
  sd.duration = sd(x$duration)
  num.trips = nrow(x)
  multi.day = sum(x$start_date$yday != x$end_date$yday |
                  x$start_date$year != x$end_date$year)
  tab = sort(table(x$start_station), decreasing=TRUE)
  common.start = as.numeric(names(tab))[1]
  tab = sort(table(x$end_station), decreasing=TRUE)
  common.end = as.numeric(names(tab))[1]
  return(data.frame(bike.nr, mean.duration, sd.duration, num.trips,
                    multi.day, common.start, common.end))
}
```

[Switch to the R console] After re-loading our script file, we can use `lapply` with the updated `process.bike` function and `do.call` with `rbind` to construct an updated `bicycle.info` variable:

```
source("exercise3_start.R")
spl2 = lapply(spl, process.bike)
bicycle.info = do.call(rbind, spl2)
```

As before, we can investigate the first few rows of this new data frame with the `head` function:

```
head(bicycle.info)
#                       bike.nr mean.duration sd.duration num.trips multi.day
# ? (0x241EC230) ? (0x241EC230)      33.17014   60.414725       192         2
# ? (0x67D8478F) ? (0x67D8478F)      25.73062   40.001034       129         0
# ? (0xA533D104) ? (0xA533D104)      38.82852  107.237058       142         1
# B00000                 B00000      11.86333    7.971572        45         0
# B00001                 B00001      18.05180   32.528162       933         8
# B00002                 B00002      23.10213   60.710399       703         5
#                 common.start common.end
# ? (0x241EC230)            22         22
# ? (0x67D8478F)            22         52
# ? (0xA533D104)            35         35
# B00000                    47         47
# B00001                    22         22
# B00002                    22         22
```

6 Merging data and the `apply` function

[Switch to Slide 12] So far we have not taken advantage of the lat/long information for the start and end locations of our trips because trips and stations are separate. Let's change that by merging together our data. Our eventual goal from this will be to compute the distance of each trip (as the crow flies).

The concept of merging data frames, which is essentially the same as database joins for those of you familiar with database terminology, centers around combining data from two data frames based on a shared id. In our case, the `start_station` and `end_station` variables can both be matched to the `id` variable in the `stations` data frame. To join together the data frames in R, we use the `merge` function, and the first two arguments are the two data frames to be merged. The `by.x` argument identifies the name of the variable in the first data frame to match, and the `by.y` argument identifies the name of the variable in the second data frame to match. By default, `merge` will match all variables that share names. The output of the `merge` function is a data frame that contains a row for every matched pair of rows between the two data frames. The outputted rows will contain all the variables from both of the merged data frames, so in the example on the slide we have both the `wday` variable from the `trips` data frame and the `lat` variable from the `stations` data frame. You'll notice on the slide that the `trips` data frame had a row that was not matched, with `station` value 4, and the `stations` data frame also had a row that was not matched, with `id` value 3. With the default arguments, `merge` will drop all rows that are not matched, so we don't see either of these in the final output. In database terminology, this is called an "inner join."

[Switch to Slide 13] To retain all the unmatched rows from one or both of the data frames, we can use the `all.x` and `all.y` arguments to the `merge` function. For instance, on this slide we've set the `all.x` argument to `TRUE`, which means we'll keep rows from the first argument, even if they're not matched by any row in the second argument. In the case that they're not matched, all the variables from the second data frame are set to the `NA`, or missing, value. In database terminology, this is called a "left outer join." If we set `all.x` to `FALSE` and `all.y` to `TRUE`, we would have a "right outer join," and if we set both arguments to `TRUE` we would have a "full outer join." Let's now switch back to R and use the `merge` function to link in information about start and stop station for each trip.

[Switch to the R console] First, we'll merge the `start_station` variable in the `trips` data frame with the `id` variable in the `stations` data frame, storing the result in a data frame called `merged`. Because we already removed all the trips with a missing start or end station, we expect all of our trips to be successfully matched to a row in the `stations` data frame with an inner join:

```
merged = merge(trips, stations, by.x="start_station", by.y="id")
```

We can check the number of rows in `merged` to make sure it matches the number of rows in `trips`:

```
nrow(trips)
# [1] 551611
nrow(merged)
# [1] 551611
```

Good — we didn't lose any observations when merging. Using the `names` function, we can look at the new variables that were added to the information stored in the `trips` data frame:

```
names(merged)
#  [1] "start_station"     "id"                "duration"
#  [4] "start_date"        "end_date"          "end_station"
#  [7] "bike_nr"           "subscription_type" "zip_code"
# [10] "birth_date"        "gender"            "name"
# [13] "lat"               "lng"
```

In addition to the 11 variables we had for each trip, we've merged in the `name`, `lat`, and `lng` variables from the `stations` data frame. We also want to merge in the information for the end stations, so we'll merge the `merged` data frame we just created with `stations` again, this time matching the `end_station` variable to the `id` variable. We can overwrite `merged` with the result of this second merge:

```
merged = merge(merged, stations, by.x="end_station", by.y="id")
```

Again, we can check the number of rows in the result to make sure we haven't lost any data:

```
nrow(merged)
# [1] 551611
```

We can again check the variables in our output data frame using the `names` function:

```
names(merged)
#  [1] "end_station"       "start_station"     "id"
#  [4] "duration"          "start_date"        "end_date"
#  [7] "bike_nr"           "subscription_type" "zip_code"
# [10] "birth_date"        "gender"            "name.x"
# [13] "lat.x"             "lng.x"             "name.y"
# [16] "lat.y"             "lng.y"
```

Because the `merge` function wanted to create a second `name`, `lat`, and `lng` variable for the end station, it renamed these three variables — `name.x`, `lat.x`, and `lng.x` are for the start location, and `name.y`, `lat.y`, and `lng.y` are for the end location. The next step is to compute the distance for trip. To do this, we'll learn one more function — the `apply` function.

[Switch to Slide 14] Previously, we've looked at the `tapply` function and the split-apply-combine paradigm, which both involve summarizing information about groups of data. However, sometimes we want to compute compute a value for every observation or every variable in our dataset. In these cases, we use the `apply` function. The first argument to `apply` is a matrix of data — if you recall from the Introduction to R module, a matrix is like a data frame except it only contains one type of data, usually all numbers. In the example on the

slide, we have a matrix with four columns containing the latitude and longitude of the start and end locations of each trip. The third argument to `apply` is the function we want to call, and the second argument is an indicator of the "margin" of the matrix on which we want to apply the function — 1 refers to rows and 2 refers to columns. In our case, we want to call our function `lat.long.dist` on every row of our matrix, returning the distances of every trip in our dataset. Therefore, we'll be using 1 for the second argument to `apply`.

[Switch to the R console] We saw that we need to provide the `apply` function with a matrix of values. The `merged` data frame has a number of non-numeric values, like the `POSIXlt` variables and the names of the stations. As a result, we'll select just the columns we need: `lat.x`, `lat.y`, `lng.x`, and `lng.y`. We can limit the columns rather easily in R:

```
lat.long = merged[c("lat.x", "lat.y", "lng.x", "lng.y")]
```

The `lat.long` variable is still a data frame, but we can convert it to a matrix with the `as.matrix` function. This actually isn't strictly necessary, since `apply` will automatically convert our first argument to a matrix.

```
lat.long = as.matrix(lat.long)
```

We can take a look at our matrix with the `summary` function, just like we would with a data frame:

```
summary(lat.long)
#      lat.x           lat.y           lng.x            lng.y
#  Min.   :42.31   Min.   :42.31   Min.   :-71.15   Min.   :-71.15
#  1st Qu.:42.35   1st Qu.:42.35   1st Qu.:-71.09   1st Qu.:-71.09
#  Median :42.35   Median :42.35   Median :-71.07   Median :-71.07
#  Mean   :42.35   Mean   :42.35   Mean   :-71.08   Mean   :-71.08
#  3rd Qu.:42.36   3rd Qu.:42.36   3rd Qu.:-71.06   3rd Qu.:-71.06
#  Max.   :42.39   Max.   :42.39   Max.   :-71.04   Max.   :-71.04
```

Now that we have our matrix `lat.long` defined, we need to define our `lat.long.dist` function, which computes the great-circle distance between the start and end locations. It's usually a bad idea to try to implement such a function yourself using a formula like the Haversine formula for great circle distances, because R comes with many functions that will provide functionality like this. I found a package called `Imap` that has a convenient interface for us through a function called `gdist`. The first step to using this function is to install and load the `Imap` package. We can do this by first calling the `install.packages` command, selecting a mirror near us, and then using the `library` function to load the package:

```
install.packages("Imap")
library(Imap)
```

Now that we've loaded our package, we can look at the arguments to the `gdist` function from its help page:

```
?gdist
```

We see that we'll want to provide our `lng.x`, `lat.x`, `lng.y`, and `lat.y` variables, in that order, as the first four arguments to `gdist`. We can also set the units to kilometers for this calculation. Let's start by trying to plug in the values for the first trip:

```
lat.long[1,]
#     lat.x     lat.y     lng.x     lng.y
#  42.34476  42.34002 -71.09788 -71.10081
gdist(-71.09788, 42.34476, -71.10081, 42.34002, units="km")
# [1] 0.5792399
```

However, we're going to need to programmatically extract the four parameters from our row of `lat.long`. With a data frame, we would use the dollar sign to get at variables. However, for a matrix we instead use square brackets and the name of variable:

```
gdist(lat.long[1,]["lng.x"], lat.long[1,]["lat.x"],
      lat.long[1,]["lng.y"], lat.long[1,]["lat.y"], units="km")
#     lat.x
# 0.5795105
```

[Switch to latlong.R] We can slightly adapt this code to make it our `lat.long.dist` function, which inputs a row of our `lat.long` matrix and outputs the distance of that trip in kilometers. We just need to add the function header and replace `lat.long[1,]` with `x`, the name of our input variable:

```
lat.long.dist = function(x) {
  return(gdist(x["lng.x"], x["lat.x"], x["lng.y"], x["lat.y"], units="km"))
}
```

We can load up our code with the `source` function:

```
source("latlong.R")
```

We want this function to be run on the rows of lat.long, so we can test it on a row. If there are errors, this makes it very manageable to debug, since it's just being called once, on the row you selected.

```
lat.long.dist(lat.long[1,])
#     lat.x
# 0.5795105
```

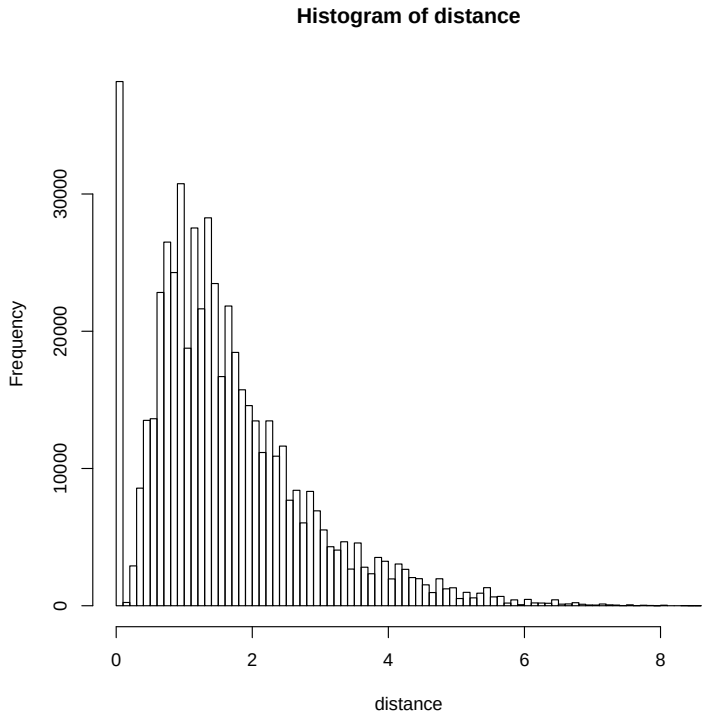
The function looks like it's working properly, so now we'll use it in a call to `apply`, using 1 as the margin to operate on the rows of `lat.long`. We'll store our result in a variable called `distance`.

```
distance = apply(lat.long, 1, lat.long.dist)
```

[It takes about two minutes to run the function, so it might be good time management to go over the final in-class exercise while the code is running]

Now that we're done running the code, we can plot the histogram of trip distance:

```
hist(distance, breaks=100)
```



We can see that there are peaks in the distribution of trip durations — one is at 0, which is trips where the user started and ended at the same station, and the other is trips of length 1 km.

Finally, we can copy the distance variable we created into our data frame for use in further analyses.

```
trips$distance = distance
```

[Switch to Slide 15] We're going to finish with a final in-class exercise. Hubway charges a variable amount for a bike ride, depending on the duration of the ride. There is no charge for users who take trips between 0 and 30 minutes, a $2 charge for trips 30–60 minutes, and a $6 fee for trips 60–90 minutes. Every 30 minutes after that incurs an additional $8 fee, up to a maximum fee of $100. The rate for registered users is 75% that of casual users, so for instance registered users have a max fee of $75). Use the `apply` function to compute the fee associated with each trip. The fee depends on both the duration of the trip and the registration status the person taking the trip.

As a hint hint, if you create a matrix with the `registration_status` and `duration` variables, it will because a matrix of text because `registration_status` is text. It will be easier to work with a matrix of numbers, so you should add a variable that takes `TRUE/FALSE`

or 1/0 values based on whether a user is registered. If you're having trouble getting started on the code, you can use the template code in `exercise4_start.R`.

[During the in-class exercise, circulate to provide guidance to students.]

[Switch to exercise4_start.R] Now, let's go over the answers to the final exercise. *[Wherever possible, get students give answers.]*

The first part of this file builds a variable called `is.registered`, which is `TRUE` if a user is registered and `FALSE` if not:

```
trips$is.registered = trips$subscription_type == "Registered"
```

Next, we'll limit `trips` to the two variables we need for our computation, `is.registered` and `duration`, storing the result in a variable called `fee.info`:

```
fee.info = trips[c("is.registered", "duration")]
```

Next, we'll fill in the missing parts of the `get.fee` function, which inputs a row of `fee.info` containing the registration status of the user and duration of a single trip. We'll add the formulae on the slides to compute the base fee based on the trip duration, and we'll scale the fee based the variable `multiplier`, which is set based on whether the user is registered.

```
get.fee = function(x) {
  # We'll multiply the fee by the multiplier variable.
  # Change the if clause to check if is.registered is 1.
  multiplier = 1
  if (x["is.registered"] == 1) {
    multiplier = 0.75
  }

  if (x["duration"] < 30) {
    return(0*multiplier)
  } else if (x["duration"] < 60) {
    return(2*multiplier)
  } else if (x["duration"] < 420) {
    return((8*floor(x["duration"]/30)-10)*multiplier)
  } else {
    return(100*multiplier)
  }
}
```

We can load up this new function with `source`:

```
source("exercise4_start.R")
```

We can test it out on a few rows. Let's start by testing it out on row 1:

```
fee.info[1,]
#   is.registered duration
# 1          TRUE     0.15
get.fee(fee.info[1,])
# [1] 0
```

The trip had a duration of less than one minute, and `get.fee` correctly identified that there is no fee associated with the trip. Now let's check row 36:

```
fee.info[36,]
#    is.registered duration
# 36          TRUE    32.55
get.fee(fee.info[36,])
# [1] 1.5
```

Because the trip duration is between 30 and 60 minutes, there is a base fee of $2. However, this user is registered, so instead we calculate their fee as $1.50.

Now that we feel comfortable `get.fee` is operating properly, we can use `apply` to create a variable in our `trips` data frame with the fee for each trip:

```
trips$fee = apply(fee.info, 1, get.fee)
```

We can use `sum` to figure out to total fees over the time period of interest:

```
sum(trips$fee)
# [1] 704676.5
```

That concludes the module on data wrangling. I hope you all feel more comfortable with cleaning, summarizing, and reshaping your data!