Intro to R

R is a functional programming language, which means that most of what one does is apply functions to objects.

We will begin with a brief introduction to R objects and how functions work, and then focus on getting data into R, manipulating that data in R, and generating basic summary statistics.

Getting Help

R has very good built-in documentation that describes what functions do.

To get help about a particular function, use ? followed by the function name, like so:

?read.table

Vectors

Let's start by creating one of the simplest R objects, a vector of numbers.

```
v1<-c(1,2,3,4,5)
```

v1 is an *object* which we created by using the `<-` operator (less followed by a dash).

v1 now contains the output of the *function* `c(1,2,3,4,5)` (c, for **c**ombined, combines elements into a vector

Let's display the contents of v1:

```
print(v1)
```

```
## [1] 1 2 3 4 5
```

We can also just type the name of an object (in this case `v1`) to display it.

Let's make a few more objects:

```
x<-10
y<-11
letters<-c("a", "b", "c", "d")
```

We might want to get a list of all the objects we've created at this point. In R, the function ls() returns a character vector with the names of all the objects in a specified environment (by default, the set of user-defined objects and functions).

```
ls()
```

```
## [1] "letters" "v1"      "x"       "y"
```

We can manipulate objects like so:

```
x+5
```

```
## [1] 15
```

```
print(x)
```

```
## [1] 10
```

```
x*2
```

```
## [1] 20
```

```
print(x)
```

```
## [1] 10
```

Note that the value of `x` is not modified here. We did not save the output to a new object, so it is printed to the screen.

If we want to update the value of `x`, we need to use our assignment operator:

```
x<-x+5
print(x)
```

```
## [1] 15
```

R handles vector math for us automatically:

```
v1*x
```

```
## [1] 15 30 45 60 75
```

```
v2<-v1*x
print(v2)
```

```
## [1] 15 30 45 60 75
```

Object Types

All objects have a type. Object types are a complex topic and we are only going to scratch the surface today. To slightly simplify, all data objects in R are either atomic vectors (contain only a single type of data), or data structures that combine atomic vectors in various ways. We'll walk through a few examples. First, let's consider a couple of the vectors that we've already made.

```
class(x)
```

```
## [1] "numeric"
```

```
class(letters)
```

```
## [1] "character"
```

Numeric and *character* data types are two of the most common we'll encounter, and are just what they sound like. Another useful type is *logical* data, as in TRUE/FALSE. We can create a logical vector directly like so:

```
logic1<-c(TRUE, TRUE, FALSE, FALSE)
print(logic1)
```

```
## [1]  TRUE  TRUE FALSE FALSE
```

Or we can use logical tests.

```
logic2<-v1>2
print(logic2)
```

```
## [1] FALSE FALSE  TRUE  TRUE  TRUE
```

Note that the logical test here (>2) is applied independently to each element in the vector. We'll come back to this when we talk about data subsets.

A final thing to note about logical vectors: they can be converted to numeric with TRUE=1 and FALSE=0. So a simple way to find how many elements in a logical vector are true is sum().

```
sum(logic2)
```

```
## [1] 3
```

Data Frames

So far we've been talking about atomic vectors, which only contain a single data type (every element is logical, or character, or numeric). However, data sets will usually have multiple different data types: numeric for continunous data, character for categorical data and sample labels. Depending on how underlying types are combined, we can have four different "higher-level" data types in R:

Dimensions	Homogeneous	Heterogeneous
1-D	atomic vector	list
2-D	matrix	data frame

We'll focus on data frames, as those are the data type most commonly found in real biological analysis. A data frame is a collection of vectors, which can be (but don't have to be) different types, but all have to have the same length. Let's make a couple of toy data frames.

One way to do this is with the data.frame() function.

```
df1<-data.frame(label=c("rep1", "rep2", "rep3", "rep4"), data=c(23, 34, 15, 19))
class(df1)
```

```
## [1] "data.frame"
```

```
print(df1)
```

```
##   label data
## 1  rep1   23
## 2  rep2   34
## 3  rep3   15
## 4  rep4   19
```

str() gives lots of information about the data type of the consituent parts of a data frame

```
str(df1)
```

```
## 'data.frame':    4 obs. of  2 variables:
##  $ label: Factor w/ 4 levels "rep1","rep2",..: 1 2 3 4
##  $ data : num  23 34 15 19
```

Note that label is a *factor* – this is a special kind of character vector. By default, when creating a data frame or reading data from a file, R will convert strings (character vectors) to factors. Factors can be useful, but also introduce some pitfalls, which we can come back to if we have time. A good source of information on factors is here (http://swcarpentry.github.io/r-novice-inflammation/01-supp-factors.html).

A more common way to create a data frame is by reading from a file. There are a few functions to do this in R: read.table() and read.csv() are some of the most common.

```
bus<-read.table(file="http://software.rc.fas.harvard.edu/ngsdata/workshops/2015_March/mbta_bus.tsv", header=TRUE,
 sep="\t", stringsAsFactors=T)

#note that R can read data directly from the web -- no need to download the file first

head(bus)
```

```
##   route      type cost.per.pax ridership pax.per.trip
## 1     1       Key         0.63     13306          416
## 2     4  Commuter         4.38       459           21
## 3     5 Community         2.99       156            5
## 4     7     Local         1.52      3893          177
## 5     8     Local         2.02      3844           85
## 6     9     Local         1.21      5980          187
```

head() displays the first few rows of a data frame and is great for examining large objects. Let's also look at str()

```
str(bus)
```

```
## 'data.frame':    163 obs. of  5 variables:
##  $ route       : Factor w/ 163 levels "1","10","100",..: 1 74 104 128 142 151 2 10 25 26 ...
##  $ type        : Factor w/ 5 levels "Community","Commuter",..: 4 2 1 5 5 5 5 5 5 4 ...
##  $ cost.per.pax: num  0.63 4.38 2.99 1.52 2.02 1.21 1.75 1.84 2.66 1.11 ...
##  $ ridership   : int  13306 459 156 3893 3844 5980 3184 3312 1285 6227 ...
##  $ pax.per.trip: int  416 21 5 177 85 187 398 237 22 111 ...
```

summary() is a good way to see some basic information about a data frame

```
summary(bus)
```

```
##      route            type      cost.per.pax     ridership
##  1      :  1   Community:  1   Min.   :0.570   Min.   :   30.0
##  10     :  1   Commuter : 13   1st Qu.:1.350   1st Qu.:  755.5
##  100    :  1   Express  : 10   Median :2.150   Median : 1481.0
##  101    :  1   Key      : 15   Mean   :2.433   Mean   : 2427.1
##  104    :  1   Local    :124   3rd Qu.:3.055   3rd Qu.: 3101.0
##  105    :  1                   Max.   :9.760   Max.   :15018.0
##  (Other):157
##   pax.per.trip
##  Min.   :   0.00
##  1st Qu.:   9.00
##  Median :  21.00
##  Mean   :  78.55
##  3rd Qu.:  48.00
##  Max.   :2550.00
##
```

Subsetting Data

Now let's look at ways to get subsets of data. R uses `[]` to index rows and columns, like so:

```
bus[1,]
```

```
##   route type cost.per.pax ridership pax.per.trip
## 1     1  Key         0.63     13306          416
```

```
head(bus[,1])
```

```
## [1] 1 4 5 7 8 9
## 163 Levels: 1 10 100 101 104 105 106 108 109 11 110 111 112 114 116 ... CT3
```

```
bus[1,1]
```

```
## [1] 1
## 163 Levels: 1 10 100 101 104 105 106 108 109 11 110 111 112 114 116 ... CT3
```

```
bus[15,2]
```

```
## [1] Local
## Levels: Community Commuter Express Key Local
```

```
bus[1:5,4]
```

```
## [1] 13306   459   156  3893  3844
```

```
bus[c(1,3,5),c(2,3)]
```

```
##        type cost.per.pax
## 1       Key         0.63
## 3 Community         2.99
## 5     Local         2.02
```

Of course, usually we want to select a set of rows from a data frame matching some criteria. We can use the subset() function for this. Let's start by just getting the data for key bus routes.

```
key.bus<-subset(bus, type=="Key")
```

Sometimes we might want something more complicated. Let's say we want to get data for all "local" bus routes with at least 1000 daily riders.

```
local.bus<-subset(bus, type=="Local" & ridership > 1000)
```

We can also use [] (see `?Extract`) to subset data in more complicated ways. The key insight here is that we can use logical vectors as indexes. So for example:

```
high.ridership<-bus$ridership > 5000
class(high.ridership)
```

```
## [1] "logical"
```

```
print(high.ridership)
```

```
##   [1]  TRUE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE  TRUE  TRUE
##  [12] FALSE FALSE FALSE FALSE  TRUE  TRUE FALSE FALSE FALSE  TRUE FALSE
##  [23] FALSE FALSE  TRUE FALSE  TRUE FALSE FALSE FALSE FALSE  TRUE FALSE
##  [34] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE
##  [45] FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE  TRUE  TRUE
##  [56] FALSE  TRUE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE
##  [67] FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [78] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [89]  TRUE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [100] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [111] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [122] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [133] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [144] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [155] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

```
bus[high.ridership,]
```

```
##    route  type cost.per.pax ridership pax.per.trip
## 1      1   Key         0.63     13306          416
## 6      9 Local         1.21      5980          187
## 10    15   Key         1.11      6227          111
## 11    16 Local         0.99      5100         2550
## 16    22   Key         1.05      8151          170
## 17    23   Key         1.00     11687          899
## 21    28   Key         0.80     13550          387
## 25    32   Key         1.02     10130          103
## 27    34 Local         2.43      5959           60
## 32    39   Key         0.72     15018          406
## 44    57   Key         0.86     12081          263
## 50    66   Key         0.79     13630          147
## 54    70 Local         1.96      7021          195
## 55    71   Key         1.35      5488          157
## 57    73   Key         1.75      5834           46
## 61    77   Key         1.95      7253          165
## 68    86 Local         1.07      5844           76
## 89   111   Key         1.06     11695           38
## 92   116   Key         0.88      5077           26
```

We don't need to store the logical vector in an object, so we can get the same result with this:

```
bus[bus$ridership > 5000,]
```

```
##    route  type cost.per.pax ridership pax.per.trip
## 1      1   Key         0.63     13306          416
## 6      9 Local         1.21      5980          187
## 10    15   Key         1.11      6227          111
## 11    16 Local         0.99      5100         2550
## 16    22   Key         1.05      8151          170
## 17    23   Key         1.00     11687          899
## 21    28   Key         0.80     13550          387
## 25    32   Key         1.02     10130          103
## 27    34 Local         2.43      5959           60
## 32    39   Key         0.72     15018          406
## 44    57   Key         0.86     12081          263
## 50    66   Key         0.79     13630          147
## 54    70 Local         1.96      7021          195
## 55    71   Key         1.35      5488          157
## 57    73   Key         1.75      5834           46
## 61    77   Key         1.95      7253          165
## 68    86 Local         1.07      5844           76
## 89   111   Key         1.06     11695           38
## 92   116   Key         0.88      5077           26
```

We might want to add some new variables to our dataset. For example, perhaps we want to know the cost per day to operate each bus line. We can calculate this from the ridership and cost.per.pax numbers.

```
bus$cost.per.day<-bus$cost.per.pax * bus$ridership
head(bus)
```

```
##   route      type cost.per.pax ridership pax.per.trip cost.per.day
## 1     1       Key         0.63     13306          416      8382.78
## 2     4  Commuter         4.38       459           21      2010.42
## 3     5 Community         2.99       156            5       466.44
## 4     7     Local         1.52      3893          177      5917.36
## 5     8     Local         2.02      3844           85      7764.88
## 6     9     Local         1.21      5980          187      7235.80
```

Let's also compute the revenue per day for each bus line. This is actually a bit complicated because not all buses charge the same rate. For local/key/community lines, it is $1.60, but for express/commuter lines it is either $3.65 or $5.25. So we do this in a few steps. Let's first make a fare column that lists the fare for each line.

```
bus$fare<-ifelse(bus$type != "Commuter" & bus$type != "Express", 1.60, 3.65)
bus$fare[bus$route=="352" | bus$route=="354" | bus$route=="505"] = 5.25
```

Now we can calculate revenue per line as fare * ridership.

```
bus$revenue.per.day = bus$fare * bus$ridership
```

Now let's explore this dataset a bit. What is the mean ridership on MBTA bus lines?

```
mean(bus$ridership)
```

```
## [1] 2427.055
```

What if we wanted to get the means for all the (numeric) columns? We could type each by hand, but R has a shortcut – the apply function.

```
apply(bus[,c(3,4,5,6)], MARGIN=2, mean)
```

```
## cost.per.pax    ridership pax.per.trip cost.per.day
##     2.433067  2427.055215    78.552147  3755.512883
```

Apply can be used to apply a function (in this case mean) to all the columns (MARGIN=2) or to all the rows (MARGIN=1)

Writing Data

We've added several variables now, and we might want to write our updated dataset to a file. We do this with the write.table() function, which writes out a data.frame.

```
write.table(bus, file="mbta_bus_updated.tsv", quote=F, sep="\t", row.names=F, col.names=T)
```

But where did R put the file on our computer? R does all file operations without a full path in a working directory. RStudio has a preference to set the default working directory, which is typically something like /Users/Tim/R.

To see the current working directory, use:

```
getwd()
```

```
## [1] "/Users/williams03/a/workshops/2015.March/Intro_to_R"
```

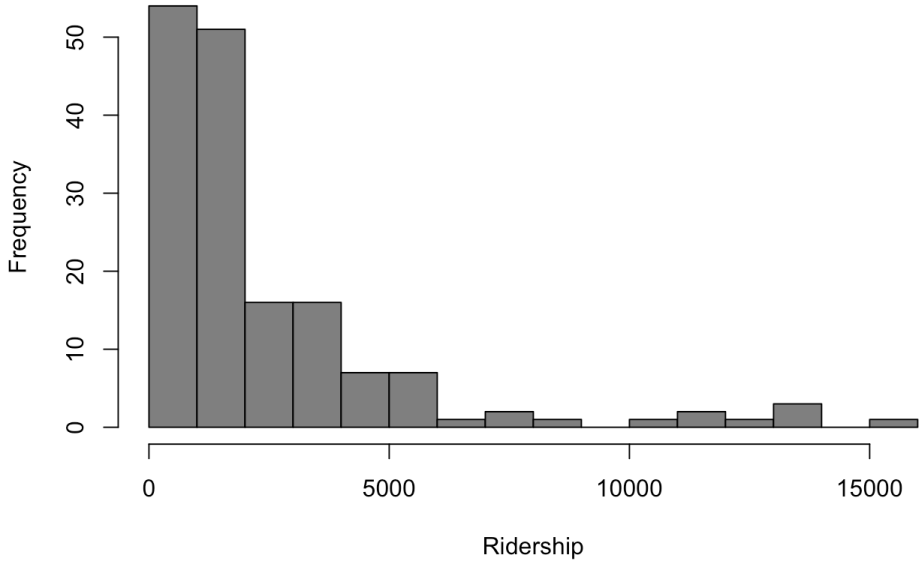
We can change the working directory with `setwd()`.

Plotting

Now, let's do some visualizations. We'll start with something simple – making a histogram.

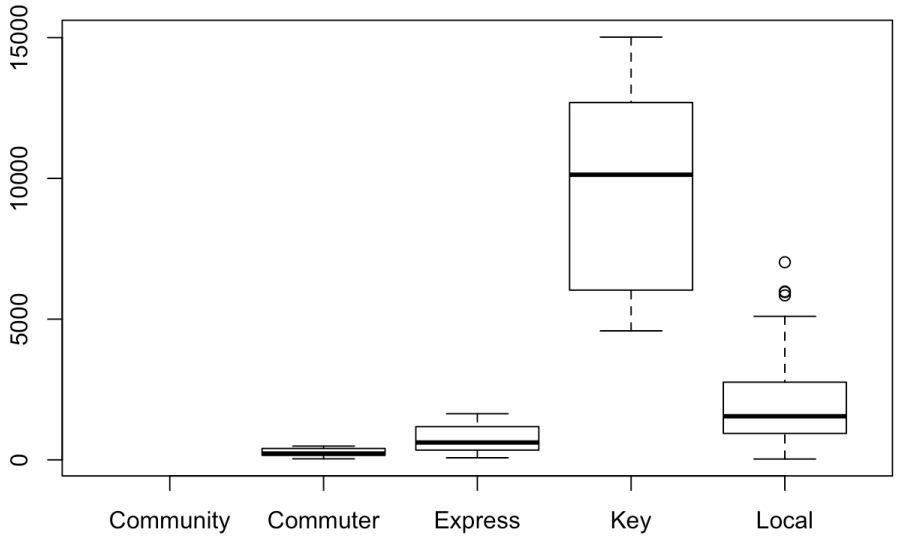
How about the distribution? We'll use the hist() function to plot a histogram

```
hist(bus$ridership, col="gray50", breaks=15, xlab="Ridership", main="")
```



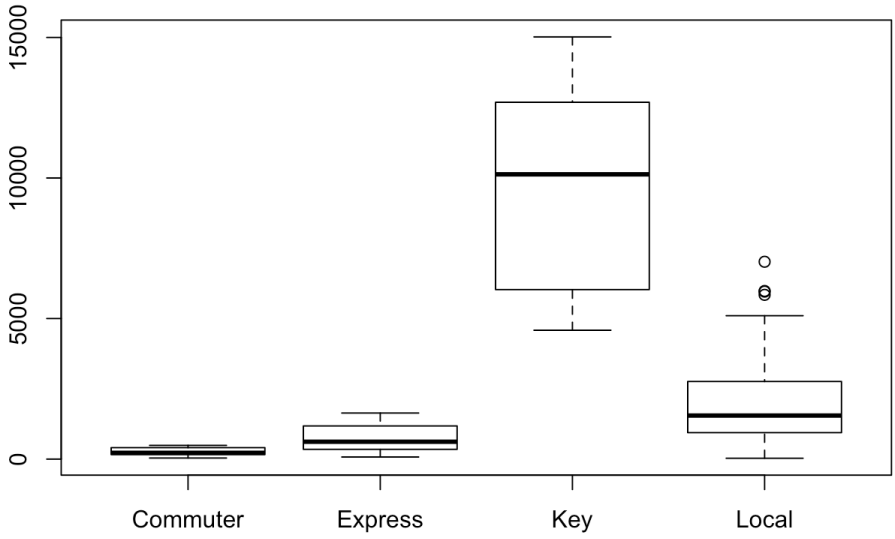
Another very useful way to summarize data is with a boxplot. Here, we'll look at ridership for each bus type. However, we'll ignore the one bus line with type==Community.

```
boxplot(ridership ~ type, data=bus[bus$type!="Community",])
```



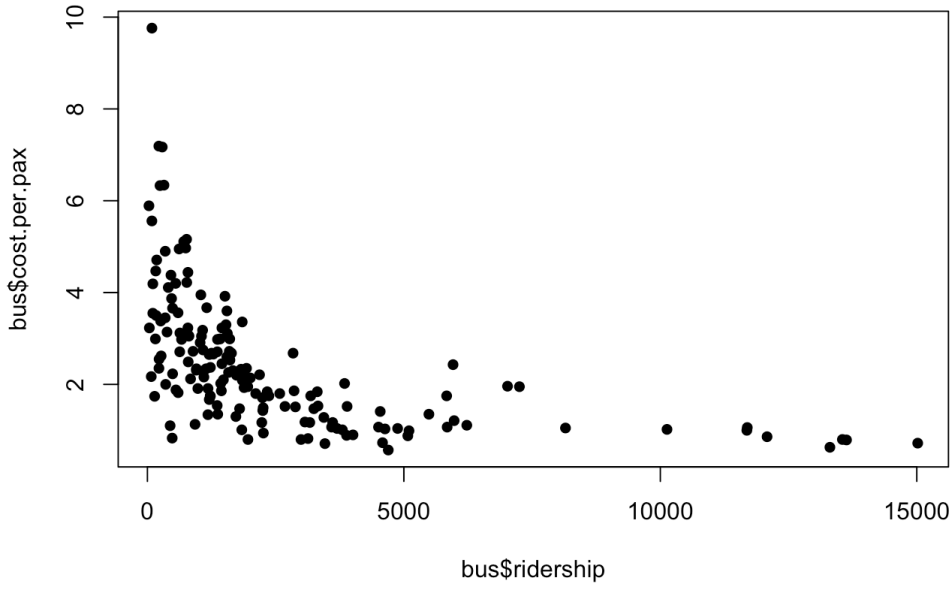
Note that Community still shows up on the axis, even though we are not plotting the data for that bus line. This is because 'type' is stored as a factor, and all levels of a factor (even those with no actual data) are plotted. One simple way to remove factor levels that don't appear in the dataset is with droplevels().

```
boxplot(ridership ~ type, data=droplevels(bus[bus$type!="Community",]))
```



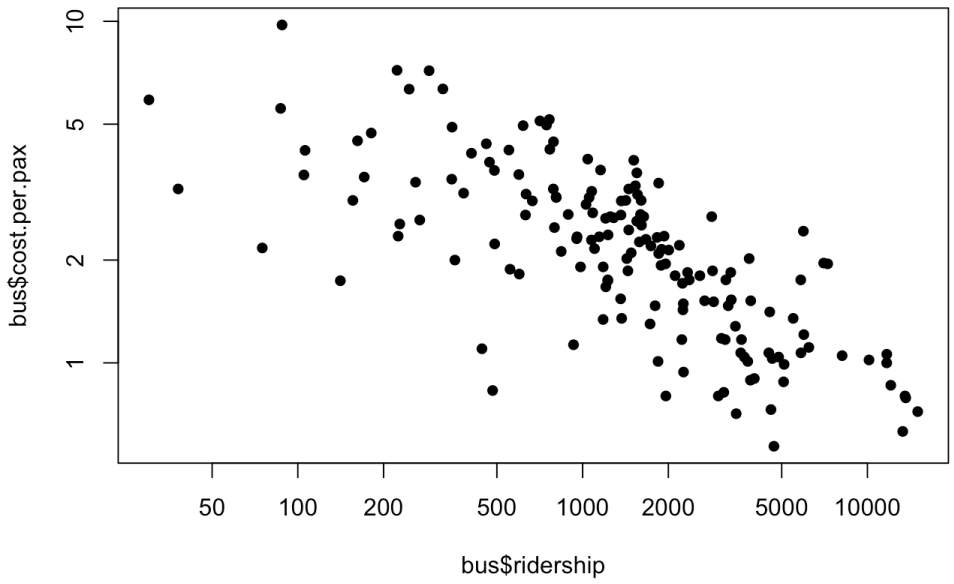
Finally, let's look at an X-Y plot, in this case comparing ridership to cost per passenger.

```
plot(x=bus$ridership, y=bus$cost.per.pax, pch=16)
```



Looks like a non-linear relationship, so let's try a log-log plot

```
plot(x=bus$ridership, y=bus$cost.per.pax, pch=16, log="xy")
```



Alternatively:

```
plot(log10(bus$cost.per.pax) ~ log10(bus$ridership), pch=16)
```

Statistics

Now let's demonstrate a few statistical tests.

Our boxplot suggests that 'key' bus routes have much higher ridership than other bus routes. Let's test that using a T-test.

```
#note that we can use a logical test as a grouping variable in a formula
t.test(ridership ~ type=="Key", data=bus)
```

```
##
##  Welch Two Sample t-test
##
## data:  ridership by type == "Key"
## t = -8.3121, df = 14.435, p-value = 7.043e-07
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -9905.928 -5851.498
## sample estimates:
## mean in group FALSE  mean in group TRUE
##            1702.020            9580.733
```

We might want to do this non-parameterically using a Wilcoxon Rank Sum Test.

```
wilcox.test(ridership ~ type=="Key", data=bus)
```

```
##
##  Wilcoxon rank sum test with continuity correction
##
## data:  ridership by type == "Key"
## W = 22, p-value = 4.281e-10
## alternative hypothesis: true location shift is not equal to 0
```

Our plot suggested that log(ridership) and log(cost per passenger) are strongly correlated. Let's test that with cor.test()

```
cor.test(log10(bus$ridership), log10(bus$cost.per.pax))
```

```
##
##  Pearson's product-moment correlation
##
## data:  log10(bus$ridership) and log10(bus$cost.per.pax)
## t = -13.3327, df = 161, p-value < 2.2e-16
## alternative hypothesis: true correlation is not equal to 0
## 95 percent confidence interval:
##  -0.7901266 -0.6421777
## sample estimates:
##       cor
## -0.7243894
```

Now let's work through a final example that involves both data manipulation and some more statistical tests. Let's say we want to find out how many bus lines are profitable (make more in fare revenue than they cost to operate)? What would you do?

```
bus$cost.per.day < bus$revenue.per.day
```

```
##   [1]  TRUE FALSE FALSE  TRUE FALSE  TRUE FALSE FALSE FALSE  TRUE  TRUE
##  [12]  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE
##  [23]  TRUE  TRUE  TRUE FALSE FALSE FALSE  TRUE FALSE FALSE  TRUE  TRUE
##  [34] FALSE  TRUE FALSE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE  TRUE
##  [45] FALSE FALSE FALSE FALSE  TRUE  TRUE FALSE FALSE  TRUE FALSE  TRUE
##  [56] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE
##  [67] FALSE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE FALSE  TRUE FALSE FALSE
##  [78] FALSE FALSE FALSE FALSE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE
##  [89]  TRUE FALSE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE FALSE FALSE FALSE
## [100] FALSE FALSE FALSE  TRUE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE
## [111] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [122] FALSE FALSE  TRUE  TRUE FALSE  TRUE FALSE  TRUE FALSE FALSE FALSE
## [133]  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE
## [144] FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE
## [155] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE
```

```
#this is a logical vector; remember sum() will give us the number of elements that are TRUE
sum(bus$cost.per.day < bus$revenue.per.day)
```

```
## [1] 63
```

Now we might want to know if different types of buses are more or less likely to be profitable. Let's start by making a new column in our data frame that is TRUE if a bus route is profitable and FALSE otherwise. How would you do that?

```
bus$profitable = bus$cost.per.day < bus$revenue.per.day
```

Now we can use the table() function to look for associations between categorical variables.

```
table(bus$profitable, bus$type)
```

```
##
##         Community Commuter Express Key Local
##   FALSE         1        8       2   2    87
##   TRUE          0        5       8  13    37
```

```
#to test non-independence using an RxC chisquare test, use chisq.test
chisq.test(table(bus$profitable, bus$type))
```

```
## Warning in chisq.test(table(bus$profitable, bus$type)): Chi-squared
## approximation may be incorrect
```

```
##
##  Pearson's Chi-squared test
##
## data:  table(bus$profitable, bus$type)
## X-squared = 26.4862, df = 4, p-value = 2.525e-05
```

Notice we got a warning message here. In this case, this means that our data violates the assumptions of the standard Chi-squared approximation (likely because there are several cells with low counts). In general, it is always wise to understand the cause of any warnings you get.

Finally, we'll end with a note about missing data. R stores missing data with the special value `NA`. While our toy dataset here has no missing values, this will usually not be the case with real data. You can read more about missing data and data types here (http://swcarpentry.github.io/r-novice-inflammation/01-supp-data-structures.html)

Preparation for next time

For next Tuesday, we will be using packages from Bioconductor. Packages extend R in various ways – Bioconductor, in particular, contains a very useful set of packages for handling various kinds of genomic and high-throughput biological data.

The easiest way to install packages in RStudio is using the pull-down menu, but you can also use the function install.packages(). For example, install.packages("maps") will install a package for using R to draw maps.

Packages, however, are not automatially loaded. To load a package, use library(packagename). For example,

```
library(maps) #loads R package to draw geographic maps
```

Installing packages from Bioconductor (http://www.bioconductor.org/install/) is a little more complicated and requires running a helper script from the Bioconductor web site, but this is actually pretty easy:

```
source("http://bioconductor.org/biocLite.R")
```

Now we can use the biocLite() function to install Bioconductor packages.

Important: prior to next Tuesday, please install DESeq2 and edgeR packages from Bioconductor, with these commands:

```
biocLite("DESeq2")
biocLite("edgeR")
```

This can take several minutes to run as if you have never installed any Bioconductor packages you will have to install a number of prerequisites.