Chapter 1

Manipulating Data Frames

Most of the time, our data is in the form of a data frame and we are interested in exploring the relationships. This chapter explores how to manipulate data frames and methods.

1.1 Classical functions for summarizing rows and columns

1.1.1 summary()

The first method is to calculate some basic summary statistics (minimum, 25th, 50th, 75th percentiles, maximum and mean) of each column. If a column is categorical, the summary function will return the number of observations in each category.

```
# use the iris data set which has both numerical and categorical variables
data( iris )
str(iris)    # recall what columns we have
```

```
## 'data.frame': 150 obs. of  5 variables:
##  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
##  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 ...
```

```
# display the summary for each column
summary( iris )
```

```
##   Sepal.Length    Sepal.Width     Petal.Length    Petal.Width
##  Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100
##  1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300
##  Median :5.800   Median :3.000   Median :4.350   Median :1.300
##  Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199
##  3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800
##  Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500
##        Species
##  setosa    :50
##  versicolor:50
##  virginica :50
##
##
##
```

1.1.2 apply()

The summary function is convenient, but we want the ability to pick another function to apply to each column and possibly to each row. To demonstrate this, suppose we have data frame that contains students grades over the semester.

```
# make up some data
grades <- data.frame(
  l.name = c('Cox', 'Dorian', 'Kelso', 'Turk'),
  Exam1 = c(93, 89, 80, 70),
  Exam2 = c(98, 70, 82, 85),
  Final = c(96, 85, 81, 92) )
```

The `apply()` function will apply an arbitrary function to each row (or column) of a matrix or a data frame and then aggregate the results into a vector.

```
# Because I can't take the mean of the last names column,
# remove the name column
scores <- grades[,-1]
scores

##   Exam1 Exam2 Final
## 1    93    98    96
## 2    89    70    85
## 3    80    82    81
## 4    70    85    92

# Summarize each column by calculating the mean.
apply( scores,       # what object do I want to apply the function to
       MARGIN=2,     # rows = 1, columns = 2, (same order as [rows, cols]
       FUN=mean      # what function do we want to apply
     )

## Exam1 Exam2 Final
## 83.00 83.75 88.50
```

To apply a function to the rows, we just change which margin we want. We might want to calculate the average exam score for person.

```
apply( scores,       # what object do I want to apply the function to
       MARGIN=1,     # rows = 1, columns = 2, (same order as [rows, cols]
       FUN=mean      # what function do we want to apply
     )

## [1] 95.66667 81.33333 81.00000 82.33333
```

This is useful, but it would be more useful to concatenate this as a new column in my `grades` data frame.

```
average <- apply(
  scores,       # what object do I want to apply the function to
  MARGIN=1,     # rows = 1, columns = 2, (same order as [rows, cols]
  FUN=mean      # what function do we want to apply
)
grades <- cbind( grades, average )
grades

##   l.name Exam1 Exam2 Final  average
## 1    Cox    93    98    96 95.66667
## 2 Dorian    89    70    85 81.33333
## 3  Kelso    80    82    81 81.00000
## 4   Turk    70    85    92 82.33333
```

There are several variants of the `apply()` function, and the variant I use most often is the function `sapply()`, which will apply a function to each element of a list or vector and returns a corresponding list or vector of results.

1.2 Package `dplyr`

```
library(dplyr)    # load the dplyr package!
```

Many of the tools to manipulate data frames in R were written without a consistent syntax and are difficult use together. To remedy this, Hadley Wickham (the writer of `ggplot2`) introduced a package called `plyr` which was quite useful. As with many projects, his first version was good but not great and he introduced an improved version that works exclusively with `data.frames` called `dplyr` which we will investigate. The package `dplyr` strives to provide a convenient and consistent set of functions to handle the most common data frame manipulations and a mechanism for chaining these operations together to perform complex tasks.

The author of the `dplyr` package has put together a very nice introduction to the package that explains in more detail how the various pieces work and I encourage you to read it at some point. http://cran.rstudio.com/web/packages/d

The pipe command `%>%` allows for very readable code. The idea is that the `%>%` operator works by translating the command `a %>% f(b)` to the expression `f(a,b)`. This operator works on any function and was introduced in the `magrittr` package. The beauty of this comes when you have a suite of functions that takes input arguments of the same type as their output. In `dplyr`, all the functions below take a `data.frame` as its first argument and outputs an appropriately modified `data.frame`. This will allow me to chain together commands in a readable fashion. For example if we wanted to start with `x`, and first apply function `f()`, then `g()`, and then `h()`, the usual R command would be `h(g(f(x)))` which is hard to read because you have to start reading at the innermost set of parentheses. Using the pipe command `%>%`, this sequence of operations becomes `x %>% f() %>% g() %>% h()`.

1.2.1 Verbs

The foundational operations to perform on a data frame are:

- Subsetting - Returns a `data.frame` with only particular columns or rows
  - `select` - Selecting a subset of columns by name or column number.
  - `filter` - Selecting a subset of rows from a data frame based on logical expressions.
  - `slice` - Selecting a subset of rows by row number.
- `arrange` - Re-ordering the rows of a data frame.
- `mutate` - Add a new column that is some function of another column.

- `summarise` - calculate some summary statistic of a column of data. This collapses a set of rows into a single row.

Each of these operations is a function in the package `dplyr`. These functions all have a similar calling syntax, the first argument is a data frame, subsequent arguments describe what to do with the input data frame and you can refer to the columns without using the `df$column` notation. All of these functions will return a data frame.

1.2.1.1 Subsetting with `select`, `filter`, and `slice`

These function allows you select certain columns and rows of a data frame.

select()

Often you only want to work with a small number of columns of a data frame. It is relatively easy to do this using the standard `[,col.name]` notation, but is often pretty tedious.

```
# recall what the grades are
grades
```

```
##   l.name Exam1 Exam2 Final  average
## 1    Cox    93    98    96 95.66667
## 2 Dorian    89    70    85 81.33333
## 3  Kelso    80    82    81 81.00000
## 4   Turk    70    85    92 82.33333
```

I could select the columns Exam columns by hand, or by using an extension of the `:` operator

```
grades %>% select( Exam1, Exam2 )    # Exam1 and Exam2=
```

```
##   Exam1 Exam2
## 1    93    98
## 2    89    70
## 3    80    82
## 4    70    85
```

```
grades %>% select( Exam1:Final )     # Columns Exam1 through Final
```

```
##   Exam1 Exam2 Final
## 1    93    98    96
## 2    89    70    85
## 3    80    82    81
## 4    70    85    92
```

```
grades %>% select( -Exam1 )          # Negative indexing by name works
```

```
##   l.name Exam2 Final  average
## 1    Cox    98    96 95.66667
## 2 Dorian    70    85 81.33333
## 3  Kelso    82    81 81.00000
## 4   Turk    85    92 82.33333
```

```
grades %>% select( 1:2 )            # Can select column by column position

##   l.name Exam1
## 1    Cox    93
## 2 Dorian    89
## 3  Kelso    80
## 4   Turk    70
```

The `select()` command has a few other tricks. There are functional calls that describe the columns you wish to select that take advantage of pattern matching. I generally can get by with `starts_with()`, `ends_with()`, and `contains()`, but there is a final operator `matches()` that takes a regular expression.

```
grades %>% select( starts_with('Exam') )    # Exam1 and Exam2

##   Exam1 Exam2
## 1    93    98
## 2    89    70
## 3    80    82
## 4    70    85
```

filter()

It is common to want to select particular rows where we have some logically expression to pick the rows.

```
# recall what the grades are
grades

##   l.name Exam1 Exam2 Final  average
## 1    Cox    93    98    96 95.66667
## 2 Dorian    89    70    85 81.33333
## 3  Kelso    80    82    81 81.00000
## 4   Turk    70    85    92 82.33333
```

```
# select students with Final grades greater than 90
grades %>% filter(Final > 90)

##   l.name Exam1 Exam2 Final  average
## 1    Cox    93    98    96 95.66667
## 2   Turk    70    85    92 82.33333
```

You can have multiple logical expressions to select rows and they will be logically combined so that only rows that satisfy **all** of the conditions are selected.[1]

[1]The logicals are joined together using `&` (and) operator or the `|` (or) operator and you may explicitly use other logicals. For example a factor column `type` might be used to select rows where type is either one or two via the following: `type==1 | type==2`.

```
# select students with Final grades above 90 and
# average score also above 90
grades %>% filter(Final > 90, average > 90)

##   l.name Exam1 Exam2 Final  average
## 1    Cox    93    98    96 95.66667

# we could also use an "and" condition
grades %>% filter(Final > 90 & average > 90)

##   l.name Exam1 Exam2 Final  average
## 1    Cox    93    98    96 95.66667
```

slice()

When you want to filter rows based on row number, this is called slicing.

```
# grab the first 2 rows
grades %>% slice(1:2)

##   l.name Exam1 Exam2 Final  average
## 1    Cox    93    98    96 95.66667
## 2 Dorian    89    70    85 81.33333
```

1.2.1.2 arrange()

We often need to re-order the rows of a data frame. For example, we might wish to take our grade book and sort the rows by the average score, or perhaps alphabetically. The `arrange()` function does exactly that. The first argument is the data frame to re-order, and the subsequent arguments are the columns to sort on. The order of the sorting column determines the precedent... the first sorting column is first used and the second sorting column is only used to break ties.

```
grades %>% arrange(l.name)

##   l.name Exam1 Exam2 Final  average
## 1    Cox    93    98    96 95.66667
## 2 Dorian    89    70    85 81.33333
## 3  Kelso    80    82    81 81.00000
## 4   Turk    70    85    92 82.33333
```

The default sorting is in ascending order, so to sort the grades with the highest scoring person in the first row, we must tell `arrange` to do it in descending order using `desc(column.name)`.

```
grades %>% arrange(desc(Final))

##   l.name Exam1 Exam2 Final  average
## 1    Cox    93    98    96 95.66667
## 2   Turk    70    85    92 82.33333
## 3 Dorian    89    70    85 81.33333
## 4  Kelso    80    82    81 81.00000
```

In a more complicated example, consider the following data and we want to order it first by Treatment Level and secondarily by the y-value. I want the Treatment level in the default ascending order (Low, Medium, High), but the y variable in descending order.

```
# make some data
dd <- data.frame(
  Trt = factor(c("High", "Med", "High", "Low"),
               levels = c("Low", "Med", "High")),
  y = c(8, 3, 9, 9),
  z = c(1, 1, 1, 2))
dd

##    Trt y z
## 1 High 8 1
## 2  Med 3 1
## 3 High 9 1
## 4  Low 9 2

# arrange the rows first by treatment, and then by y (y in descending order)
dd %>% arrange(Trt, desc(y))

##    Trt y z
## 1  Low 9 2
## 2  Med 3 1
## 3 High 9 1
## 4 High 8 1
```

1.2.1.3 mutate()

I often need to create a new column that is some function of the old columns. This was often cumbersome. Consider code to calculate the average grade in my grade book example.

```
grades$average <- (grades$Exam1 + grades$Exam2 + grades$Final) / 3
```

Instead, we could use the `mutate()` function and avoid all the `grades$` nonsense.[2]

```
grades %>% mutate( average = (Exam1 + Exam2 + Final)/3 )

##   l.name Exam1 Exam2 Final  average
## 1    Cox    93    98    96 95.66667
## 2 Dorian    89    70    85 81.33333
## 3  Kelso    80    82    81 81.00000
## 4   Turk    70    85    92 82.33333
```

You can do multiple calculations within the same `mutate()` command, and you can even refer to columns that were created in the same `mutate()` command.

```
grades %>% mutate( average = (Exam1 + Exam2 + Final)/3,
                   grade = cut(average, c(0,60,70,80,90,100), c('F','D','C','B','A')) )

##   l.name Exam1 Exam2 Final  average grade
## 1    Cox    93    98    96 95.66667     A
## 2 Dorian    89    70    85 81.33333     B
## 3  Kelso    80    82    81 81.00000     B
## 4   Turk    70    85    92 82.33333     B
```

[2]There is another way to do this. The command `with(df, expression)` will attach the dataframe `df` to the current environment, then evaluate the expression, and then detach the dataframe. However, to assign the result back to the data frame, I still end up typing the name of the dataframe twice.

1.2.1.4 summarise()

By itself, this function is quite boring, but will become useful later on. Its purpose is to calculate summary statistics using any or all of the data columns. Notice that we get to chose the name of the new column. The way to think about this is that we are collapsing information stored in multiple rows into a single row of values.

```
# calculate the mean of exam 1
summarise( grades, mean.E1=mean(Exam1))

##   mean.E1
## 1      83
```

We could calculate multiple summary statistics if we like.

```
# calculate the mean of each of the exams
grades %>% summarise( mean.E1=mean(Exam1), stddev.E1=sd(Exam2) )

##   mean.E1 stddev.E1
## 1      83      11.5
```

If we want to apply the same statistic to each column, we use the `summarise_each()` command. We have to be a little careful here because the function you use has to work on every column (that isn't part of the grouping structure (see `group_by()`)).

```
# calculate the mean and stddev of each column
grades %>% summarise_each( funs(mean, sd) )

## Warning in mean.default(structure(1:4, .Label = c("Cox", "Dorian", "Kelso", :
argument is not numeric or logical:  returning NA
## Warning in var(if (is.vector(x) || is.factor(x)) x else as.double(x), na.rm = na.rm):
Calling var(x) on a factor x is deprecated and will become an error.
##  Use something like 'all(duplicated(x)[-1L])' to test for a constant vector.

##   l.name_mean Exam1_mean Exam2_mean Final_mean average_mean l.name_sd
## 1          NA         83      83.75       88.5     85.08333  1.290994
##   Exam1_sd Exam2_sd Final_sd average_sd
## 1 10.23067     11.5 6.757712   7.078266
```

Miscellaneous functions

There are some more function that are useful but aren't as commonly used. For sampling the functions `sample_n()` and `sample_frac()` will take a subsample of either `n` rows or of a fraction of the data set. The function `n()` returns the number of rows in the data set. Finally `rename()` will rename a selected column.

1.2.2 Split, apply, combine

Aside from unifying the syntax behind the common operations, the major strength of the `dplyr` package is the ability to split a data frame into a bunch of sub-dataframes, apply a sequence of one or more of the operations we just described, and then combine results back together. We'll consider data from an experiment from spinning wool into yarn. This experiment considered two different types of wool (A or B) and three different levels of tension on the thread. The response variable is the number of breaks in the resulting yarn. For each of the 6 `wool:tension` combinations, there are 9 replicated observations.

```
data(warpbreaks)
str(warpbreaks)

## 'data.frame': 54 obs. of  3 variables:
##  $ breaks : num  26 30 54 25 70 52 51 26 67 18 ...
##  $ wool   : Factor w/ 2 levels "A","B": 1 1 1 1 1 1 1 1 1 1 ...
##  $ tension: Factor w/ 3 levels "L","M","H": 1 1 1 1 1 1 1 1 1 2 ...
```

The first we must do is to create a data frame with additional information about how to break the data into sub-dataframes. In this case, I want to break the data up into the 6 wool-by-tension combinations. Initially we will just figure out how many rows are in each wool-by-tension combination.

```
# what variable(s) shall we group one
grouped.warpbreaks <- warpbreaks %>% group_by( wool, tension)

# n() is a function that returns how many rows are in the
# currently selected sub-dataframe
summarise(grouped.warpbreaks, n = n() )

## Source: local data frame [6 x 3]
## Groups: wool [?]
##
##     wool tension     n
##   (fctr)  (fctr) (int)
## 1      A       L     9
## 2      A       M     9
## 3      A       H     9
## 4      B       L     9
## 5      B       M     9
## 6      B       H     9
```

Using the same `summarise` function, we could calculate the group mean and standard deviation for each wool-by-tension group.

```
warpbreaks %>% group_by(wool, tension) %>%
  summarise( n           = n(),             # I added some formatting to tell the
             mean.breaks = mean(breaks),    # reader I am calculating several
             sd.breaks   = sd(breaks)       # statistics.
)

## Source: local data frame [6 x 5]
## Groups: wool [?]
##
##     wool tension     n mean.breaks sd.breaks
##   (fctr)  (fctr) (int)       (dbl)     (dbl)
## 1      A       L     9    44.55556 18.097729
## 2      A       M     9    24.00000  8.660254
## 3      A       H     9    24.55556 10.272671
## 4      B       L     9    28.22222  9.858724
## 5      B       M     9    28.77778  9.431036
## 6      B       H     9    18.77778  4.893306
```

If instead of summarizing each split, we might want to just do some calculation and the output should have the same number of rows as the input data frame. In this case I'll tell `dplyr` that we are *mutating* the

data frame instead of summarizing it. For example, suppose that I want to calculate the residual value

$$e_{ijk} = y_{ijk} - \bar{y}_{ij\cdot}$$

where $\bar{y}_{ij\cdot}$ is the mean of each `wool:tension` combination.

```
temp <- warpbreaks %>%
   group_by(wool, tension) %>%
   mutate(resid = breaks - mean(breaks))
head( temp )  # show the first couple of rows of the result

## Source: local data frame [6 x 4]
## Groups: wool, tension [1]
##
##   breaks   wool tension      resid
##    (dbl) (fctr)  (fctr)      (dbl)
## 1     26      A       L -18.555556
## 2     30      A       L -14.555556
## 3     54      A       L   9.444444
## 4     25      A       L -19.555556
## 5     70      A       L  25.444444
## 6     52      A       L   7.444444
```

1.2.3 Chaining commands together

Suppose we have the results of a small 5K race. The data given to us is in the order that the runners signed up but we want to calculate the results for each gender, calculate the placings, and the sort the data frame by gender and then place. We can think of this process as having three steps: 1) Splitting 2) Ranking 3) Re-arranging.

```
# input the initial data
race.results <- data.frame(
  name=c('Bob', 'Jeff', 'Rachel', 'Bonnie', 'Derek', 'April','Elise','David'),
  time=c(21.23, 19.51, 19.82, 23.45, 20.23, 24.22, 28.83, 15.73),
  gender=c('M','M','F','F','M','F','F','M')
)

# how should I group?
grouped.results <- race.results %>% group_by( gender)

# calculate the rankings using the rank() function
temp.df1 <- grouped.results %>% mutate( place = rank(time) )

# arrange the rows based on gender and then place
temp.df2 <- temp.df1 %>% arrange( gender, place )
```

```
# output the result
temp.df2

## Source: local data frame [8 x 4]
## Groups: gender [2]
##
##     name  time gender place
##   (fctr) (dbl) (fctr) (dbl)
## 1 Rachel 19.82      F     1
## 2 Bonnie 23.45      F     2
## 3  April 24.22      F     3
## 4  Elise 28.83      F     4
## 5  David 15.73      M     1
## 6   Jeff 19.51      M     2
## 7  Derek 20.23      M     3
## 8    Bob 21.23      M     4
```

It would be nice if I didn't have to save all these intermediate results because keeping track of `temp1` and `temp2` gets pretty annoying if I keep changing the order of how things or calculated or add/subtract steps. The way this is typically handled in R is to to just nest one command inside the next. The same set of commands could be run as follows:

```
arrange(
  mutate(
    group_by(
      race.results,          # using race.results
      gender),               # group by gender
    place = rank( time )),   # mutate to calculate the place column
  gender, place)             # arrange the result by gender and place

## Source: local data frame [8 x 4]
## Groups: gender [2]
##
##     name  time gender place
##   (fctr) (dbl) (fctr) (dbl)
## 1 Rachel 19.82      F     1
## 2 Bonnie 23.45      F     2
## 3  April 24.22      F     3
## 4  Elise 28.83      F     4
## 5  David 15.73      M     1
## 6   Jeff 19.51      M     2
## 7  Derek 20.23      M     3
## 8    Bob 21.23      M     4
```

This is extremely hard to read because the commands are separated from the arguments (e.g. the `arrange` function call was at the top, but the columns to arrange by are on the last line). To get around this, the author of `dplyr` gives us an operator to combine these simple operations smoothly. The composition operation `%>%` takes the following `A %>% f(B)` and converts it to the statement `f(A, B)`. This allows us to write the following code that does exactly what the above two code chunks did.

```
race.results %>%                    # what data frame am I interested in
  group_by(gender) %>%              # break things by gender
  mutate(place=rank(time)) %>%      # calculate the placings within each gender
  arrange(gender, place)            # arrange the result by gender and place

## Source: local data frame [8 x 4]
## Groups: gender [2]
##
##     name  time gender place
##   (fctr) (dbl) (fctr) (dbl)
## 1 Rachel 19.82      F     1
## 2 Bonnie 23.45      F     2
## 3  April 24.22      F     3
## 4  Elise 28.83      F     4
## 5  David 15.73      M     1
## 6   Jeff 19.51      M     2
## 7  Derek 20.23      M     3
## 8    Bob 21.23      M     4
```

If I only wanted the top three finishers in each gender, we could simply add a `filter` command after the `place` column was calculated.

```
race.results %>%                    # what data frame am I interested in
  group_by(gender) %>%              # break things by gender
  mutate(place=rank(time)) %>%      # calculate the placings within each gender
  filter( place <= 3 ) %>%          # only get the top 3 finishers within each gender
  arrange(gender, place)            # arrange the result by gender and place

## Source: local data frame [6 x 4]
## Groups: gender [2]
##
##     name  time gender place
##   (fctr) (dbl) (fctr) (dbl)
## 1 Rachel 19.82      F     1
## 2 Bonnie 23.45      F     2
## 3  April 24.22      F     3
## 4  David 15.73      M     1
## 5   Jeff 19.51      M     2
## 6  Derek 20.23      M     3
```

1.3 Exercises

1. The dataset `ChickWeight` which tracks the weights of 48 baby chickens (chicks) feed four different diets.

   (a) Load the dataset using

   ```
   data(ChickWeight)
   ```

   (b) Look at the help files for the description of the columns.

   (c) Remove all the observations except for the weights on day 10 and day 20.

   (d) Calculate the mean and standard deviation for each diet group on days 10 and 20.

2. The *OpenIntro* textbook on statistics includes a data set on body dimensions.

(a) Load the file using

```
Body <- read.csv('http://www.openintro.org/stat/data/bdims.csv')
```

(b) The column `sex` is coded as a `1` if the individual is male and `0` if female. This is a non-intuitive labeling system. Create a new column `sex.MF` that uses labels `Male` and `Female`.

(c) The columns `wgt` and `hgt` measure weight and height in kilograms and centimeters (respectively). Use these to calculate the Body Mass Index (BMI) for each individual where

$$BMI = \frac{Weight\,(kg)}{[Height\,(m)]^2}$$

Notice you should get values between 18 to 30.

(d) Double check that your calculated BMI column is correct by examining the summary statistics of the column.

(e) The function `cut` takes a vector of continuous numerical data and creates a factor based on your give cut-points.

```
# Define a continuous vector to convert to a factor
x <- 1:10

# divide range of x into three groups of equal length
cut(x, breaks=3)

##  [1] (0.991,4] (0.991,4] (0.991,4] (0.991,4] (4,7]     (4,7]     (4,7]
##  [8] (7,10]    (7,10]    (7,10]
## Levels: (0.991,4] (4,7] (7,10]

# divide x into four groups, where I specify all 5 break points
cut(x, breaks = c(0, 2.5, 5.0, 7.5, 10))

##  [1] (0,2.5]  (0,2.5]  (2.5,5]  (2.5,5]  (2.5,5]  (5,7.5]  (5,7.5]
##  [8] (7.5,10] (7.5,10] (7.5,10]
## Levels: (0,2.5] (2.5,5] (5,7.5] (7.5,10]

# divide x into 3 groups, but give them a nicer
# set of group names
cut(x, breaks=3, labels=c('Low','Medium','High'))

##  [1] Low    Low    Low    Low    Medium Medium Medium High   High   High
## Levels: Low Medium High
```

Create a new column of in the data frame that divides the `age` into decades (10-19, 20-29, 30-39, etc). Notice the oldest person in the study is 67.

(f) Find the average BMI for each Sex and Age group.