Dates, Loops, Controls, and Functions in R

In this lecture, we will be exploring how to use R to simplify reading in these files, how to use “for”, “if”, and “while” statements to control the flow of your code, and how to wrap everything up in a well defined function.

Sequences and Repetitions in R

There are many times you’ll need to generate a simple vector of numbers in R. The two most often forms these vectors take are a simple sequence, generated using either a colon for a simple sequence, or the seq() command for a more complex vector, or a repetivite vector generated using the rep() command.

```
1:10
```

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

```
seq(from = 1, to = 10)
```

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

```
rep(x = 1, times = 10)
```

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

```
rep(x = c(1, 2, 3), times = 10)
```

```
##  [1] 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3
```

Repetition is straightforard. Sequences can be generated in more useful ways using the flags in the seq() command.

```
seq(from = 0, to = 100, by = 10)
```

```
##  [1]   0  10  20  30  40  50  60  70  80  90 100
```

```
seq(from = 0, length = 11, by = 10)
```

```
##  [1]   0  10  20  30  40  50  60  70  80  90 100
```

Remember, both seq() and rep() return vectors. This means they can be assigned and manipulated like any other vector

```
seq(from = 10, to = 100, by = 10) + 5
```

```
##  [1]  15  25  35  45  55  65  75  85  95 105
```

```r
seq(from = 10, to = 100, by = 10) * 3.5
```

```
##  [1]  35  70 105 140 175 210 245 280 315 350
```

```r
seq(from = 10, to = 100, by = 10)[c(2, 4, 6)]
```

```
## [1] 20 40 60
```

In Class Exercise:

Use both seq() and rep() to generate the vector: 1 20 3 40 5 60 7 80 9 100 1 20 3 40 5 60 7 80 9 100

Dates in R

R treats Dates as a unique variable type, represented as: “YYYY-MM-DD”. The as.Date() function can convert strings into this date format. Since dates are a variable type, they can be assigned to object like any other variable, and R can do most operations (that makes sense) on dates, simply as part of the basic R package.

```r
dates <- c(as.Date("2010-01-01"), as.Date("2010-12-31"))

print(dates)
```

```
## [1] "2010-01-01" "2010-12-31"
```

```r
mean(dates)
```

```
## [1] "2010-07-02"
```

```r
min(dates)
```

```
## [1] "2010-01-01"
```

```r
max(dates)
```

```
## [1] "2010-12-31"
```

```r
as.Date("2010-02-10") + 10
```

```
## [1] "2010-02-20"
```

R can convert many, many different looking strings into a date class using the “format” argument in as.Date:

```r
as.Date("1980-02-10")
```

```
## [1] "1980-02-10"
```

```r
as.Date("19800210", format = "%Y%m%d")
```

```
## [1] "1980-02-10"
```

```r
as.Date("10Feb80", format = "%d%b%y")
```

```
## [1] "1980-02-10"
```

```r
as.Date("9/18/2016", format = "%m/%d/%Y")
```

```
## [1] "2016-09-18"
```

Symbol	Meaning
%d	Day as a number
%a	Abbreviated Weekday
%A	Unabbreviated Weekday
%m	Month as a number
%b	Abbreviated month
%B	Unabbreviated month
%y	Two-digit year
%Y	Four-digit year

Format is both an argument and a function itself:

```r
Sys.Date()
```

```
## [1] "2017-02-24"
```

```r
format(Sys.Date(), "%d %B %Y")
```

```
## [1] "24 February 2017"
```

```r
format(Sys.Date(), "%d-%b")
```

```
## [1] "24-Feb"
```

```r
format(Sys.Date(), "%Y")
```

```
## [1] "2017"
```

```r
paste0("carsDataSet_", format(Sys.Date(), "%Y%m%d"), ".csv")
```

```
## [1] "carsDataSet_20170224.csv"
```

R can even import from Excel, with Excel's wierd 5-digit dates.

```r
# From Windows Excel
as.Date(30829, origin = "1899-12-30")
```

```
## [1] "1984-05-27"
```

```r
# From iOS Excel
as.Date(29367, origin = "1904-01-01")
```

```
## [1] "1984-05-27"
```

R can even use the seq command to generate strings of dates, which can be helpful when cleaning data.

```r
dates <- seq(as.Date("2015-01-01"), to = as.Date("2015-12-31"), by = "month")
print(dates)
```

```
##  [1] "2015-01-01" "2015-02-01" "2015-03-01" "2015-04-01" "2015-05-01"
##  [6] "2015-06-01" "2015-07-01" "2015-08-01" "2015-09-01" "2015-10-01"
## [11] "2015-11-01" "2015-12-01"
```

```r
mean(dates)
```

```
## [1] "2015-06-16"
```

```r
min(dates)
```

```
## [1] "2015-01-01"
```

```r
max(dates)
```

```
## [1] "2015-12-01"
```

Rather technical note: In the case of two digit years, R (currently) assumes that years 00-68 are 2000 - 2068, and years 69-99 are 1969 - 1999.

In-Class Exercise:

Use the seq() command to generate a vector of every month-end date in 2010.

Loops and Control in R

R has many different ways for you to simplify your code, as well as ways for you to control how your code executes. The most often used are "for", "while" loops which allow you to automate repetitive tasks, and "if/else", which allows you to set conditional statements for control.

For loops

All loops are built in a three-part structure: 1. Initializing the data 2. The test condition 3. The function to be performed

To demonstrate, let's take a look at building the first 10 numbers of the Fibonacci sequence, first without a loop, then with one.

```
fib <- c(0, 1)
fib <- c(fib, fib[2] + fib[1])
fib <- c(fib, fib[3] + fib[2])
fib <- c(fib, fib[4] + fib[3])
fib <- c(fib, fib[5] + fib[4])
fib <- c(fib, fib[6] + fib[5])
fib <- c(fib, fib[7] + fib[6])
fib <- c(fib, fib[8] + fib[7])
fib <- c(fib, fib[9] + fib[8])
print(fib)
```

```
##  [1]  0  1  1  2  3  5  8 13 21 34
```

This is clunky, takes up a lot of space, and is hard to understand as a single functional program. Now let's look at it with a loop, keeping in mind the structure from above.

```
# Initalize the data
fib <- c(0, 1)

# Set up the condition
for(i in 2:9) {

  # The function itself
  nextFib <- fib[i] + fib[i - 1]
  fib <- c(fib, nextFib)
}
print(fib)
```

```
##  [1]  0  1  1  2  3  5  8 13 21 34
```

This does the exact same thing as the lines and lines of code above, but in just five lines, instead of nine.

Loops can also be very useful for cleaning data in data frames. Let's take a look at some data from the labor market, and how we can use loops to analyse it. Open the labordata.csv file you should have pulled from gitHub. This data is pull from FRED, the economic data repository maintained by the St. Louis Federal Reserve (and an excellent source of data for class projects, hint hint hint). "unrate" is the official unemployment rate (U3) reported by the BLS, and "unlevel" is the number of unemployed people (in thousands).

Using a for loop, let's look at the absolute change in the unemployment rate over time.

```
labordata <- read.csv("labordata.csv", stringsAsFactors = FALSE)
head(labordata)
```

```
##       date unrate unlevel
## 1 1/1/2001    4.2    6023
## 2 2/1/2001    4.2    6089
## 3 3/1/2001    4.3    6141
## 4 4/1/2001    4.4    6271
## 5 5/1/2001    4.3    6226
## 6 6/1/2001    4.5    6484
```

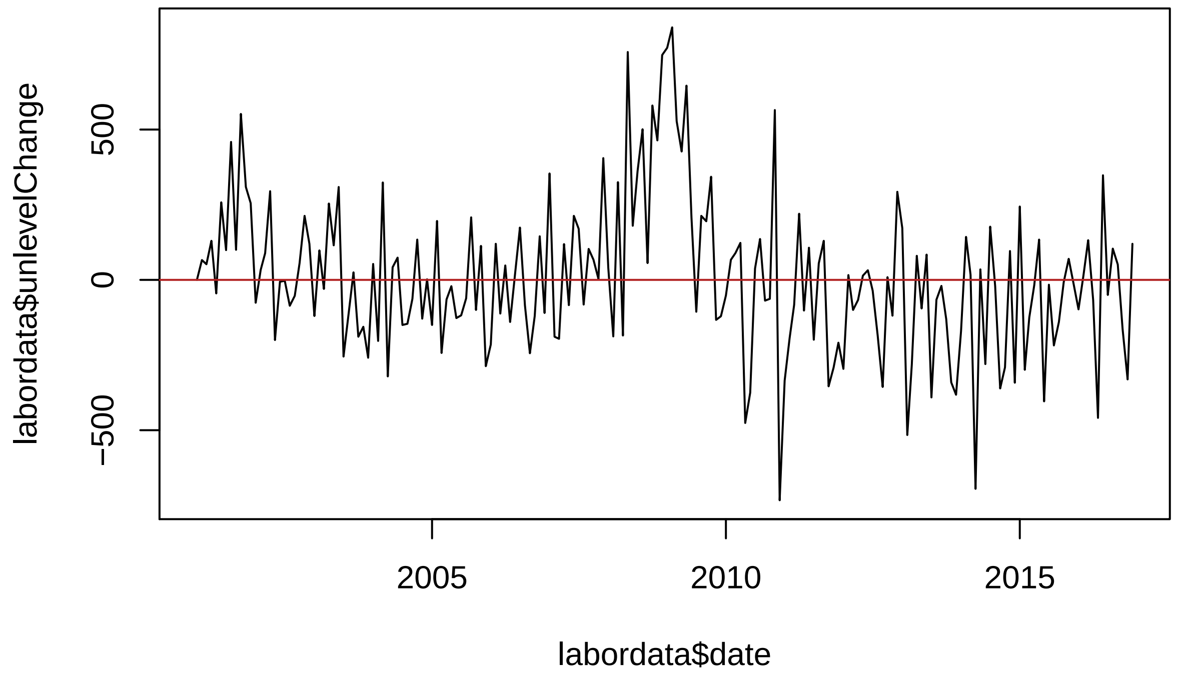
```
# fix the date
labordata$date <- as.Date(labordata$date, format = "%m/%d/%Y")
nrow(labordata)
```

```
## [1] 193
```

193 rows is far, far too many to work with line by line. Loops will save us.

```
# Initalize the data
labordata$unlevelChange <- 0
# Set the condition
for(i in 2:nrow(labordata)) {
  # Run the code
  labordata$unlevelChange[i] <- labordata$unlevel[i] - labordata$unlevel[i - 1]
}

plot(labordata$date, labordata$unlevelChange, type = 'l')
abline(h = 0, col = "firebrick")
```



In Class Exercise 2:

Using the example above, write code that calculates the percent change in the unemployment level from month-to-month. Call this column unlevelPercentChange.

For loops can also be run across columns, not just down rows. For loops can also be nested inside each other, allowing for a broader range of controls

While Loops

While loops are very similar to for loops, except a for loop is executed over a pre-defined vector, but a while loop is executed until a condition is met.

Let's take another look at the Fibonnaci sequence.

```r
# Initalize the data
fib <- c(0, 1)
i = 2

while(length(fib) <= 10) {
  fib <- c(fib, fib[i] + fib[i - 1])
  i <- i + 1
}

print(fib)
```

```
##  [1]  0  1  1  2  3  5  8 13 21 34 55
```

VERY IMPORTANT: Note that a for loop is executed over a preset vector, which means it necessarily has a finite number of iterations. A while loop looks for a condition to be true, which means you have to be very careful to set up your while loop. It is very easy to set up a loop where the condition is never true, and thus create a loop which will never terminate.

In Class Exercise:

Use a while loop to find the greatest number in the Fibonnaci sequence that is still less than 10000.

If/else

If statements give you a way for your code to make choices. The code inside an if statement is only executed if the stated condition is met (recall "mutate" from the dplyr package").

```r
# Initalize the data
labordata$health <- NA

for(i in 1:nrow(labordata)){
  if(labordata$unlevelChange[i] < 0) {
    labordata$health[i] <- "HEALTHY!"
  } else if(labordata$unlevelChange[i] >= 0) {
    labordata$health[i] <- "WARNING!"
  }
}
```

```r
head(labordata, 10)
```

```
##          date unrate unlevel unlevelChange   health
## 1  2001-01-01    4.2    6023             0 WARNING!
## 2  2001-02-01    4.2    6089            66 WARNING!
## 3  2001-03-01    4.3    6141            52 WARNING!
## 4  2001-04-01    4.4    6271           130 WARNING!
## 5  2001-05-01    4.3    6226           -45 HEALTHY!
## 6  2001-06-01    4.5    6484           258 WARNING!
## 7  2001-07-01    4.6    6583            99 WARNING!
## 8  2001-08-01    4.9    7042           459 WARNING!
## 9  2001-09-01    5.0    7142           100 WARNING!
## 10 2001-10-01    5.3    7694           552 WARNING!
```

Else is often used in conjuction with if statements to provide a catchall so that no data slips through the code. Else conditions are triggered when none of the other if statements are.

```r
labordata$health <- NA

for(i in 1:nrow(labordata)){
  if(labordata$unlevelChange[i] < 0) {
    labordata$health[i] <- "HEALTHY!"
  } else if(labordata$unlevelChange[i] > 0) {
    labordata$health[i] <- "WARNING!"
  } else {
    labordata$health[i] <- "NEUTRAL"
  }
}

head(labordata, 15)
```

```
##          date unrate unlevel unlevelChange   health
## 1  2001-01-01    4.2    6023             0  NEUTRAL
## 2  2001-02-01    4.2    6089            66 WARNING!
## 3  2001-03-01    4.3    6141            52 WARNING!
## 4  2001-04-01    4.4    6271           130 WARNING!
## 5  2001-05-01    4.3    6226           -45 HEALTHY!
## 6  2001-06-01    4.5    6484           258 WARNING!
## 7  2001-07-01    4.6    6583            99 WARNING!
## 8  2001-08-01    4.9    7042           459 WARNING!
## 9  2001-09-01    5.0    7142           100 WARNING!
## 10 2001-10-01    5.3    7694           552 WARNING!
## 11 2001-11-01    5.5    8003           309 WARNING!
## 12 2001-12-01    5.7    8258           255 WARNING!
## 13 2002-01-01    5.7    8182           -76 HEALTHY!
## 14 2002-02-01    5.7    8215            33 WARNING!
## 15 2002-03-01    5.7    8304            89 WARNING!
```

In this case, "NEUTRAL" will be given for any values which are 0 or missing. It's important to be aware of all possible cases that are falling under your "else" statements.

In-Class Exercise:

Let's use if statements to examine GDP Data and find recessions. In the file "gpddata", gdpc1 is the quarterly measure of real GDP, in chained 2009 dollars.

```
# Read in and format the data properly
gdpdata <- read.csv("gdpdata.csv")
gdpdata$date <- as.Date(gdpdata$date, format = "%m/%d/%Y")
```

With the data read in, we're ready to find periods of recession. The defintion of a recession is at least two consecutive quarters of negative GDP growth.

Create a column called "recession" that has a value of TRUE during a recession, and FALSE otherwise.

Writing Functions

All of the above examples are useful, but they are difficult to use flexibly. For example, the code used to generate the Fibonnaci sequence, as written, can only generate the first 10 digits. Instead of having to edit code every time we want to look at a different input, we can use arguments in our function calls.

```
# Finds the first 10 terms of the fibonnaci sequence
fibonnaci <- function(terms) {
  fib = c(0, 1)
  i = 2

  while(length(fib) < terms) {
  fib <- c(fib, fib[i] + fib[i - 1])
  i <- i + 1
  }

  return(fib)
}
```

To run the fuction, just type the name of the function, with the number of terms in the parantheses as an argument, same as any other function you've used so far.

```
fibonnaci(terms = 10)
```

```
##  [1]  0  1  1  2  3  5  8 13 21 34
```

```
fibonnaci(terms = 15)
```

```
##  [1]   0   1   1   2   3   5   8  13  21  34  55  89 144 233 377
```

Note that to use this function, we assign a value to terms inside the parentheses. You can't assign a value to a variable "terms" and have the function "know" what you mean. You could assign a value to another variable like so:

```
x = 15
fibonnaci(x)
```

```
##  [1]   0   1   1   2   3   5   8  13  21  34  55  89 144 233 377
```

Be careful doing this, though. This makes code harder to read.

Functions can be set to use the same output methods that you're already familiar with, including write.csv, or creating charts. Oftentimes you'll want to return a specific value, vector, or other data structure. The "return()" command you see above is used to export objects from a function.

In Class Exercise:

Convert the code above to take in a value, called "maxValue" and calculate the largest value of the fibonacci sequence less than that value.

If statements can also be used to make sure that your functions are being given arguments that make sense.

```r
fibonnaci(-10)
```

```
## [1] 0 1
```

```r
fibonnaci(3.14159)
```

```
## [1] 0 1 1 2
```

This doesn't return any value that makes sense. With If statements to check the validity of the inputs, and the stop() function to terminate the program, we can correct this problem.

```r
fibonnaci <- function(terms = 2) {
  if(!is.numeric(terms)) {
    stop("Please enter a numeric value")
  }
  if(round(terms) != terms) {
    stop("Please enter a whole number")
  }
  if(terms <= 0) {
    stop("Please enter a positive value")
  }
  fib = c(0, 1)
  i = 2

  while(length(fib) < terms) {
    fib <- c(fib, fib[i] + fib[i - 1])
    i <- i + 1
  }

  return(fib)

}

fibonnaci(terms = "A")
```

```
## Error in fibonnaci(terms = "A"): Please enter a numeric value
```

```r
fibonnaci(terms = 1.10)
```

```
## Error in fibonnaci(terms = 1.1): Please enter a whole number
```

```r
fibonnaci(terms = -10)
```

```
## Error in fibonnaci(terms = -10): Please enter a positive value
```

```r
fibonnaci(terms = 10)
```

```
##  [1]  0  1  1  2  3  5  8 13 21 34
```

In Class Exercise:

Write a function that takes in a date (and only a date) and uses If/else statements to return the quarter of the year that date is in as a character, denoted Q1, Q2, Q3, or Q4. The quarters are:

- Q1: Jan, Feb, Mar
- Q2: Apr, May, Jun
- Q3: Jul, Aug, Sep
- Q4: Oct, Nov, Dec

Reading in multiple files

Oftentimes when working with data, you'll need to read in many, many files to assemble your final dataset.Let's take a look at state level employment data.

```r
akna <- read.csv("stateData/AKNA.csv", stringsAsFactors = FALSE)
head(akna)
```

```
##         DATE  AKNA
## 1 1990-01-01 230.0
## 2 1990-02-01 232.1
## 3 1990-03-01 234.4
## 4 1990-04-01 236.3
## 5 1990-05-01 236.2
## 6 1990-06-01 239.4
```

We have the data for one state, it would obviously be very annoying to copy/paste/edit this line 50 times more for the rest of the country. This can be automated using loops along with the list.files() command.

```r
stateEmployFiles<- list.files("stateData/")
stateEmployFiles
```

```
##  [1] "AKNA.csv" "ALNA.csv" "ARNA.csv" "AZNA.csv" "CANA.csv" "CONA.csv"
##  [7] "CTNA.csv" "DCNA.csv" "DENA.csv" "FLNA.csv" "GANA.csv" "HINA.csv"
## [13] "IANA.csv" "IDNA.csv" "ILNA.csv" "INNA.csv" "KSNA.csv" "KYNA.csv"
## [19] "LANA.csv" "MANA.csv" "MDNA.csv" "MENA.csv" "MINA.csv" "MNNA.csv"
## [25] "MONA.csv" "MSNA.csv" "MTNA.csv" "NCNA.csv" "NDNA.csv" "NENA.csv"
## [31] "NHNA.csv" "NJNA.csv" "NMNA.csv" "NVNA.csv" "NYNA.csv" "OHNA.csv"
## [37] "OKNA.csv" "ORNA.csv" "PANA.csv" "RINA.csv" "SCNA.csv" "SDNA.csv"
## [43] "TNNA.csv" "TXNA.csv" "UTNA.csv" "VANA.csv" "VTNA.csv" "WANA.csv"
## [49] "WINA.csv" "WVNA.csv" "WYNA.csv"
```

This gives us the file names. We can use an argument in list.files to make this easier still.

```r
stateEmployFiles <- list.files("stateData/", full.names = TRUE)
stateEmployFiles
```

```
##  [1] "stateData/AKNA.csv" "stateData/ALNA.csv" "stateData/ARNA.csv"
##  [4] "stateData/AZNA.csv" "stateData/CANA.csv" "stateData/CONA.csv"
##  [7] "stateData/CTNA.csv" "stateData/DCNA.csv" "stateData/DENA.csv"
## [10] "stateData/FLNA.csv" "stateData/GANA.csv" "stateData/HINA.csv"
## [13] "stateData/IANA.csv" "stateData/IDNA.csv" "stateData/ILNA.csv"
## [16] "stateData/INNA.csv" "stateData/KSNA.csv" "stateData/KYNA.csv"
## [19] "stateData/LANA.csv" "stateData/MANA.csv" "stateData/MDNA.csv"
## [22] "stateData/MENA.csv" "stateData/MINA.csv" "stateData/MNNA.csv"
## [25] "stateData/MONA.csv" "stateData/MSNA.csv" "stateData/MTNA.csv"
## [28] "stateData/NCNA.csv" "stateData/NDNA.csv" "stateData/NENA.csv"
## [31] "stateData/NHNA.csv" "stateData/NJNA.csv" "stateData/NMNA.csv"
## [34] "stateData/NVNA.csv" "stateData/NYNA.csv" "stateData/OHNA.csv"
## [37] "stateData/OKNA.csv" "stateData/ORNA.csv" "stateData/PANA.csv"
## [40] "stateData/RINA.csv" "stateData/SCNA.csv" "stateData/SDNA.csv"
## [43] "stateData/TNNA.csv" "stateData/TXNA.csv" "stateData/UTNA.csv"
## [46] "stateData/VANA.csv" "stateData/VTNA.csv" "stateData/WANA.csv"
## [49] "stateData/WINA.csv" "stateData/WVNA.csv" "stateData/WYNA.csv"
```

Now that we have the file names, and where they are on the computer, we can use a for loop to bring them all in and combine them into one file.

Now that we have all the functions in hand, we combine them to create a new dataset.

```r
stateEmployData <- read.csv(stateEmployFiles[1])

head(stateEmployData)
```

```
##         DATE  AKNA
## 1 1990-01-01 230.0
## 2 1990-02-01 232.1
## 3 1990-03-01 234.4
## 4 1990-04-01 236.3
## 5 1990-05-01 236.2
## 6 1990-06-01 239.4
```

```r
nrow(stateEmployData)
```

```
## [1] 324
```

As before, we first need to initialize the data before running it through a loop.

```
stateEmployData$state <- names(stateEmployData[2])
colnames(stateEmployData) <- c("date", "employ", "state")

for(i in 2:length(stateEmployFiles)) {
  tempData <- read.csv(stateEmployFiles[i], stringsAsFactors = FALSE)
  tempData$state <- names(tempData[2])
  colnames(tempData) <- c("date", "employ", "state")

  stateEmployData <- rbind(stateEmployData, tempData)
}
nrow(stateEmployData)
```

```
## [1] 16524
```

Aggregating data

Now that we have a complete dataset, we can start analyzing it. Obviously, with 16524 rows, 324 days, and 51 states, we have a lot of different ways to think about and examine this data, some much more complex than others. Let's look at the average number of employed people by state over the entire time series. First, let's think about using a for loop.

```
states <- unique(stateEmployData$state)
stateMeans <- data.frame(state = states[1], meanEmploy = mean(stateEmployData$employ[stateEmployData$sta

for(i in 2:length(states)) {
  tempStateMean = data.frame(state = states[i], meanEmploy = mean(stateEmployData$employ[stateEmployData

  stateMeans <- rbind(stateMeans, tempStateMean)
}
head(stateMeans)
```

```
##   state meanEmploy
## 1  AKNA   295.0022
## 2  ALNA  1867.3519
## 3  ARNA  1125.8787
## 4  AZNA  2211.5164
## 5  CANA 14194.1256
## 6  CONA  2117.6747
```

While this obviously works, it is, again, clunky and very hard to read. There is an easier way, in the R function "aggregate".

```
stateMeansAggregate <- aggregate(x = stateEmployData$employ,
                                 by = list(stateEmployData$state),
                                 FUN = mean)
colnames(stateMeansAggregate) <- c("state", "meanEmploy")

head(stateMeansAggregate)
```

```
##   state meanEmploy
## 1  AKNA   295.0022
## 2  ALNA  1867.3519
## 3  ARNA  1125.8787
## 4  AZNA  2211.5164
## 5  CANA 14194.1256
## 6  CONA  2117.6747
```

One line, no loop, easier to read, think about, and debug. Aggregate works by taking in at least three arguments: x is the vector you want to aggregate down, by is the variable (or variables) that you are aggregating on, and FUN is the function to perform.
Note that aggregate returns generic column names when you run it. You'll need to rename the columns into meaningful titles.

In Class Exercise:

Use aggregate to give the nationwide total level of employment for each date in the dataset.

In Class Exercise:

Using the "Quarter" function you wrote earlier, assign a Quarter value to the stateEmployData dataset (the unaggregated data). Hint: You may need a for loop to run the function over all the dates. Once you have this set, use the new column and the aggregate command to create a dataset of average state-level employment by quarter. Second hint: You'll also need a "year" column to make this work.

HOMEWORK

1. Write a function, called is.leapyear() that takes in a 2- or 4- digit year value (and only such a value), called "yearIn", and determines whether or not it is a leap year. The rules for a leap year are: If a year is a multiple of 4, it is a leap year, UNLESS the year is also a multiple of 100, UNLESS the year is also a multiple of 400. The function should return TRUE or FALSE.
2. Using the GDP data, add a column to the labor market data that denotes when periods of recession occured. Note that the labor market data is monthly, the GDP data is quarterly. Loops and if statements will make this easier.