Data Wrangling in R

Sönke Ehret, NYU Data Services

Last Updated: December 2016

I. Brief Overview of Data Type

Vectors

```
  v1 <- rnorm(100, 75, 15)
  v2 <- as.factor(rep(c('A', 'B', 'C', 'D'), times = 25))
  v3 <- rnorm(100, 1, .5)
```

To index a vector use [ ].

```
  v2[1:10]
##  [1] A B C D A B C D A B
## Levels: A B C D
  v2[c(4, 8, 12)]
## [1] D D D
## Levels: A B C D
```

Data Frames

```
  mydata <- data.frame(v1, v2, v3)
  mydata$v1
```

```
##   [1] 101.05001  72.81603  93.36961  83.88685  76.66412  77.60697  80.97990
##   [8]  62.03844  59.63273  75.38857  52.48667 100.54537  80.27961  53.26272
##  [15]  64.43410  76.27284  97.66639  84.58751  81.73240  79.09204  66.31012
##  [22]  74.63010  78.55848  86.28033 100.34931  75.08491  67.67332  94.05037
##  [29]  89.72897  47.28097  81.92576  77.94164  74.60641  73.46427  97.09112
##  [36]  64.01737  93.22099  72.49035  86.10074  73.63221  70.19864  69.60963
##  [43]  61.66402  84.12589  77.57750  50.62337 100.83322  76.38099  71.32689
##  [50]  90.23057  59.11319  89.93094  62.24789  77.57863  79.92889  61.73315
##  [57]  66.56792  49.21559  78.53761  41.21103  86.24450  74.14343  69.96950
##  [64]  81.93106  63.87411  71.57936  66.90349  94.95945  63.36475  94.49545
##  [71]  87.03514  77.75044  54.66610  95.68763  43.08908  89.68859  95.98686
##  [78]  78.06820  75.34120  73.61378  90.14061  75.09652  46.03198  94.35198
##  [85]  76.86424  97.91090  74.70257  75.50034 107.57978  58.42153  90.24141
##  [92]  69.60777  67.30974  91.49058  74.40085  59.60893  69.42880  96.31694
##  [99]  55.80217  60.62463
```

```
  mydata[1:10, c('v1', 'v2')]
```

```
##           v1 v2
## 1  101.05001  A
## 2   72.81603  B
## 3   93.36961  C
## 4   83.88685  D
## 5   76.66412  A
## 6   77.60697  B
```

```
## 7   80.97990  C
## 8   62.03844  D
## 9   59.63273  A
## 10  75.38857  B
```

Data Type Conversion

```
  as.numeric()
  as.character()
  as.vector()
  as.matrix()
  as.data.frame()
  as.factor()
```

II. Data Management

A. Working with Strings

```
  dna <- 'ACAAAGATGCCATTGTCCCCCGGCCTCCTGCTGCTGCTGCTCTCCGGGGCCACGGCCACCGCTGCCCTGCCCCTGGAGGGTGGCCCCACCGGC
```

Counting Characters

The *'length'* function counts the number of string elements in the vector. The *'nchar'* function counts the number of characters in the string.

```
  length(dna)
## [1] 1
  nchar(dna)
## [1] 147
```

Splitting Strings

The *'strsplit'* function, takes the string and splits at a specified character. This function returns a list. By keeping the split argument "", this splits after every character.

```
  sp_dna <- strsplit(dna, '')
  class(sp_dna)
## [1] "list"
  table(sp_dna)
## sp_dna
##  A  C  G  T
## 29 53 45 20
```

By changing the split argument to 'A', this cuts the string after every 'A' and eliminates the 'A's as well.

```
  strsplit(dna, 'A')
```

```
## [[1]]
##  [1] ""
##  [2] "C"
##  [3] ""
```

```
##  [4] ""
##  [5] "G"
##  [6] "TGCC"
##  [7] "TTGTCCCCCGGCCTCCTGCTGCTGCTGCTCTCCGGGGCC"
##  [8] "CGGCC"
##  [9] "CCGCTGCCCTGCCCCTGG"
## [10] "GGGTGGCCCC"
## [11] "CCGGCCG"
## [12] "G"
## [13] "C"
## [14] "GCG"
## [15] "GC"
## [16] "T"
## [17] "TGC"
## [18] "GG"
## [19] ""
## [20] "GCGGC"
## [21] "GG"
## [22] ""
## [23] "T"
## [24] ""
## [25] "GG"
## [26] ""
## [27] ""
## [28] ""
## [29] "GC"
## [30] "GCCTCCTG"
```

Use the *'substring'* function to parse certain indices of the string:

```
  substr(dna, 20, 30)
```

```
## [1] "CCGGCCTCCTG"
```

Matching Strings

The *'grep'* function returns the string which has the pattern inside of it. The *'regexpr'* function shows you what index the pattern begins. The *'gregexpr'* function shows you the indexes of a match within the string.

```
  grep('AA', dna, value = TRUE)
## [1] "ACAAAGATGCCATTGTCCCCCGGCCTCCTGCTGCTGCTGCTCTCCGGGGCCACGGCCACCGCTGCCCTGCCCCTGGAGGGTGGCCCCACCGGCCG
  regexpr('AA', dna)
## [1] 3
## attr(,"match.length")
## [1] 2
## attr(,"useBytes")
## [1] TRUE
  gregexpr('AA', dna)
## [[1]]
## [1]   3 116 126 129 133 135
## attr(,"match.length")
## [1] 2 2 2 2 2 2
## attr(,"useBytes")
## [1] TRUE
```

What if there is more than one match in a string? Both the *'stringr'* and *'stringi'* packages can help answer

this question.

```
  library(stringr)
## Warning: package 'stringr' was built under R version 3.3.2
  str_locate_all(dna, 'AA')
## [[1]]
##      start end
## [1,]     3   4
## [2,]   116 117
## [3,]   126 127
## [4,]   129 130
## [5,]   133 134
## [6,]   135 136
```

```
  library(stringi)
## Warning: package 'stringi' was built under R version 3.3.2
  stri_count_fixed(dna, 'AA')
## [1] 6
```

Paste

The *'paste'* functions combine multiple vectors together ('concatenation')

```
  paste('X', 1:5, sep = '.')
```

```
## [1] "X.1" "X.2" "X.3" "X.4" "X.5"
```

```
  paste('X', 1:5, sep = '.', collapse = '')
```

```
## [1] "X.1X.2X.3X.4X.5"
```

```
  paste0('X', 1:5, sep = '.')
```

```
## [1] "X1." "X2." "X3." "X4." "X5."
```

Other String Functions

There are many other useful string functions and string packages.

```
string1 <- 'NYU Data Services'
tolower(string1)
```

```
## [1] "nyu data services"
```

```
toupper(string1)
```

```
## [1] "NYU DATA SERVICES"
```

```
toString(c(1, 3, 4))
```

```
## [1] "1, 3, 4"
```

B. Working with Dates

When you read in dates in R, the dates are read in as strings. Use the *'as.Date'* function to convert to a date that R will understand

```
dates <- c('11/28/2011', '12/07/2012', '08/01/2013', '02/09/2015')
class(dates)
## [1] "character"
real_dates <- as.Date(dates, format = '%m/%d/%Y')
class(real_dates)
## [1] "Date"
real_dates
## [1] "2011-11-28" "2012-12-07" "2013-08-01" "2015-02-09"
```

R only displays and understands dates in the following format: 'YYYY-MM-DD'. However, if you want to change the way the date is displayed, you can use the *'format'* function to display the date but know that this goes back to a character string and not an R date.

```
other_format <- format(real_dates, '%A %B %d, %Y')
class(other_format)
```

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

R can use the *'Sys.Date'* function to access today's date. Difference in dates can be

```
today <- Sys.Date()
```

Difference in dates can be calculated as follows:

```
dif <- today - real_dates
class(dif)
## [1] "difftime"
dif
## Time differences in days
## [1] 1836 1461 1224  667
```

C. Selecting Variables and Cases

We will be using the UScereal dataset from the *'MASS'* package:

```
  library(MASS)
```

```
## Warning: package 'MASS' was built under R version 3.3.2
```

```
  data(UScereal)
  head(UScereal)
```

```
##                            mfr calories   protein      fat   sodium
## 100% Bran                    N 212.1212 12.121212 3.030303 393.9394
## All-Bran                     K 212.1212 12.121212 3.030303 787.8788
## All-Bran with Extra Fiber    K 100.0000  8.000000 0.000000 280.0000
## Apple Cinnamon Cheerios      G 146.6667  2.666667 2.666667 240.0000
## Apple Jacks                  K 110.0000  2.000000 0.000000 125.0000
## Basic 4                      G 173.3333  4.000000 2.666667 280.0000
##                              fibre    carbo   sugars shelf potassium
## 100% Bran                30.303030 15.15152 18.18182     3 848.48485
## All-Bran                 27.272727 21.21212 15.15151     3 969.69697
## All-Bran with Extra Fiber 28.000000 16.00000  0.00000     3 660.00000
## Apple Cinnamon Cheerios   2.000000 14.00000 13.33333     1  93.33333
## Apple Jacks               1.000000 11.00000 14.00000     2  30.00000
## Basic 4                   2.666667 24.00000 10.66667     3 133.33333
##                            vitamins
```

```
## 100% Bran                enriched
## All-Bran                 enriched
## All-Bran with Extra Fiber enriched
## Apple Cinnamon Cheerios  enriched
## Apple Jacks              enriched
## Basic 4                  enriched
```

The *'which'* function reports a numeric vector of indices:

```
  which(UScereal$mfr == 'K')
```

```
##  [1]  2  3  5 15 16 18 19 22 23 24 26 36 40 42 43 46 49 51 53 56 57
```

Then use the *'which'* function in the row index of the data frame:

```
  UScereal[which(UScereal$mfr == 'K'), c('mfr', 'calories')]
```

```
##                           mfr calories
## All-Bran                    K 212.1212
## All-Bran with Extra Fiber   K 100.0000
## Apple Jacks                 K 110.0000
## Corn Flakes                 K 100.0000
## Corn Pops                   K 110.0000
## Cracklin' Oat Bran          K 220.0000
## Crispix                     K 110.0000
## Froot Loops                 K 110.0000
## Frosted Flakes              K 146.6667
## Frosted Mini-Wheats         K 125.0000
## Fruitful Bran               K 179.1045
## Just Right Fruit & Nut      K 186.6667
## Mueslix Crispy Blend        K 238.8060
## Nut&Honey Crunch            K 179.1045
## Nutri-Grain Almond-Raisin   K 208.9552
## Product 19                  K 100.0000
## Raisin Bran                 K 160.0000
## Raisin Squares              K 180.0000
## Rice Krispies               K 110.0000
## Smacks                      K 146.6667
## Special K                   K 110.0000
```

Alternatively you can use the *'subset'* function

```
subset(UScereal, calories > 250, c('mfr', 'calories'))
```

```
##                      mfr calories
## Grape-Nuts             P 440.0000
## Great Grains Pecan     P 363.6364
## Oatmeal Raisin Crisp   G 260.0000
```

With the dplyr package

Let's introduce the piping operator %>%. This symbol can be read as 'then'. Sometimes it is also referred to as the magrittr symbol. f(x, y) can be rewritten as x %>% f(y)

The *'filter'* function allows you to subset rows and the *'select'* function displays only certain columns.

```
library(dplyr)
```

```
## Warning: package 'dplyr' was built under R version 3.3.2
```

```
##
## Attaching package: 'dplyr'
## The following object is masked from 'package:MASS':
##
##     select
## The following objects are masked from 'package:stats':
##
##     filter, lag
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
UScereal %>% filter(calories > 250)
```

```
##   mfr calories   protein      fat   sodium     fibre    carbo   sugars
## 1   P 440.0000 12.000000 0.000000 680.0000 12.000000 68.00000 12.00000
## 2   P 363.6364  9.090909 9.090909 227.2727  9.090909 39.39394 12.12121
## 3   G 260.0000  6.000000 4.000000 340.0000  3.000000 27.00000 20.00000
##   shelf potassium vitamins
## 1     3  360.0000 enriched
## 2     3  303.0303 enriched
## 3     3  240.0000 enriched
```

```
UScereal %>% filter(calories > 250) %>% select(mfr, calories)
```

```
##   mfr calories
## 1   P 440.0000
## 2   P 363.6364
## 3   G 260.0000
```

D. Sorting

The *'sort'* function only works for vectors:

```
sort(UScereal$calories)
```

```
##  [1]  50.00000  73.33333  82.70677  88.00000  97.34513 100.00000 100.00000
##  [8] 100.00000 100.00000 100.00000 100.00000 110.00000 110.00000 110.00000
## [15] 110.00000 110.00000 110.00000 110.00000 110.00000 110.00000 110.00000
## [22] 110.00000 110.00000 113.63636 113.63636 120.00000 125.00000 133.33333
## [29] 133.33333 134.32836 134.32836 134.32836 134.32836 140.00000 146.66667
## [36] 146.66667 146.66667 146.66667 146.66667 146.66667 146.66667 146.66667
## [43] 149.25373 149.25373 160.00000 160.00000 160.00000 173.33333 179.10448
## [50] 179.10448 179.10448 179.10448 180.00000 186.66667 200.00000 200.00000
## [57] 208.95522 212.12121 212.12121 220.00000 220.00000 238.80597 260.00000
## [64] 363.63636 440.00000
```

When you want to sort a dataset by a variable, you can use the *'order'* function, which will return indices similar to the *'which'* function.

```
order(UScereal$calories)
```

```
##  [1] 47 37 35 10 52  3 15 41 46 60 64  5 13 14 16 17 19 22 39 53 57 58 62
## [24] 28 30 33 24 20 21  7  8 54 55 59  4 23 27 29 34 56 61 65 38 63  9 11
## [47] 49  6 25 26 42 45 51 36 48 50 43  1  2 12 18 40 44 32 31
```

```r
UScereal[order(UScereal$calories), c('mfr', 'calories')]
```

```
##                                       mfr  calories
## Puffed Rice                             Q  50.00000
## Kix                                     G  73.33333
## Honey-comb                              P  82.70677
## Cheerios                                G  88.00000
## Rice Chex                               R  97.34513
## All-Bran with Extra Fiber               K 100.00000
## Corn Flakes                             K 100.00000
## Multi-Grain Cheerios                    G 100.00000
## Product 19                              K 100.00000
## Total Whole Grain                       G 100.00000
## Wheaties                                G 100.00000
## Apple Jacks                             K 110.00000
## Cocoa Puffs                             G 110.00000
## Corn Chex                               R 110.00000
## Corn Pops                               K 110.00000
## Count Chocula                           G 110.00000
## Crispix                                 K 110.00000
## Froot Loops                             K 110.00000
## Lucky Charms                            G 110.00000
## Rice Krispies                           K 110.00000
## Special K                               K 110.00000
## Total Corn Flakes                       G 110.00000
## Trix                                    G 110.00000
## Golden Crisp                            P 113.63636
## Grape Nuts Flakes                       P 113.63636
## Honey Graham Ohs                        Q 120.00000
## Frosted Mini-Wheats                     K 125.00000
## Crispy Wheat & Raisins                  G 133.33333
## Double Chex                             R 133.33333
## Bran Chex                               R 134.32836
## Bran Flakes                             P 134.32836
## Shredded Wheat 'n'Bran                  N 134.32836
## Shredded Wheat spoon size               N 134.32836
## Total Raisin Bran                       G 140.00000
## Apple Cinnamon Cheerios                 G 146.66667
## Frosted Flakes                          K 146.66667
## Fruity Pebbles                          P 146.66667
## Golden Grahams                          G 146.66667
## Honey Nut Cheerios                      G 146.66667
## Smacks                                  K 146.66667
## Triples                                 G 146.66667
## Wheaties Honey Gold                     G 146.66667
## Life                                    Q 149.25373
## Wheat Chex                              R 149.25373
## Cap'n'Crunch                            Q 160.00000
## Cinnamon Toast Crunch                   G 160.00000
## Raisin Bran                             K 160.00000
## Basic 4                                 G 173.33333
## Fruit & Fibre: Dates Walnuts and Oats   P 179.10448
## Fruitful Bran                           K 179.10448
## Nut&Honey Crunch                        K 179.10448
```

```
## Post Nat. Raisin Bran                  P 179.10448
## Raisin Squares                         K 180.00000
## Just Right Fruit & Nut                 K 186.66667
## Quaker Oat Squares                     Q 200.00000
## Raisin Nut Bran                        G 200.00000
## Nutri-Grain Almond-Raisin              K 208.95522
## 100% Bran                              N 212.12121
## All-Bran                               K 212.12121
## Clusters                               G 220.00000
## Cracklin' Oat Bran                     K 220.00000
## Mueslix Crispy Blend                   K 238.80597
## Oatmeal Raisin Crisp                   G 260.00000
## Great Grains Pecan                     P 363.63636
## Grape-Nuts                             P 440.00000
```

With the dplyr package

The *'arrange'* function in the

```
library(dplyr)
UScereal %>% arrange(mfr, desc(calories))
```

```
##    mfr  calories    protein       fat    sodium     fibre    carbo
## 1    G 260.00000  6.0000000 4.0000000 340.00000  3.000000 27.00000
## 2    G 220.00000  6.0000000 4.0000000 280.00000  4.000000 26.00000
## 3    G 200.00000  6.0000000 4.0000000 280.00000  5.000000 21.00000
## 4    G 173.33333  4.0000000 2.6666667 280.00000  2.666667 24.00000
## 5    G 160.00000  1.3333333 4.0000000 280.00000  0.000000 17.33333
## 6    G 146.66667  2.6666667 2.6666667 240.00000  2.000000 14.00000
## 7    G 146.66667  1.3333333 1.3333333 373.33333  0.000000 20.00000
## 8    G 146.66667  4.0000000 1.3333333 333.33333  2.000000 15.33333
## 9    G 146.66667  2.6666667 1.3333333 333.33333  0.000000 28.00000
## 10   G 146.66667  2.6666667 1.3333333 266.66667  1.333333 21.33333
## 11   G 140.00000  3.0000000 1.0000000 190.00000  4.000000 15.00000
## 12   G 133.33333  2.6666667 1.3333333 186.66667  2.666667 14.66667
## 13   G 110.00000  1.0000000 1.0000000 180.00000  0.000000 12.00000
## 14   G 110.00000  1.0000000 1.0000000 180.00000  0.000000 12.00000
## 15   G 110.00000  2.0000000 1.0000000 180.00000  0.000000 12.00000
## 16   G 110.00000  2.0000000 1.0000000 200.00000  0.000000 21.00000
## 17   G 110.00000  1.0000000 1.0000000 140.00000  0.000000 13.00000
## 18   G 100.00000  2.0000000 1.0000000 220.00000  2.000000 15.00000
## 19   G 100.00000  3.0000000 1.0000000 200.00000  3.000000 16.00000
## 20   G 100.00000  3.0000000 1.0000000 200.00000  3.000000 17.00000
## 21   G  88.00000  4.8000000 1.6000000 232.00000  1.600000 13.60000
## 22   G  73.33333  1.3333333 0.6666667 173.33333  0.000000 14.00000
## 23   K 238.80597  4.4776119 2.9850746 223.88060  4.477612 25.37313
## 24   K 220.00000  6.0000000 6.0000000 280.00000  8.000000 20.00000
## 25   K 212.12121 12.1212121 3.0303030 787.87879 27.272727 21.21212
## 26   K 208.95522  4.4776119 2.9850746 328.35821  4.477612 31.34328
## 27   K 186.66667  4.0000000 1.3333333 226.66667  2.666667 26.66667
## 28   K 180.00000  4.0000000 0.0000000   0.00000  4.000000 30.00000
## 29   K 179.10448  4.4776119 0.0000000 358.20896  7.462687 20.89552
## 30   K 179.10448  2.9850746 1.4925373 283.58209  0.000000 22.38806
## 31   K 160.00000  4.0000000 1.3333333 280.00000  6.666667 18.66667
## 32   K 146.66667  1.3333333 0.0000000 266.66667  1.333333 18.66667
```

```
## 33   K 146.66667  2.6666667 1.3333333  93.33333  1.333333 12.00000
## 34   K 125.00000  3.7500000 0.0000000   0.00000  3.750000 17.50000
## 35   K 110.00000  2.0000000 0.0000000 125.00000  1.000000 11.00000
## 36   K 110.00000  1.0000000 0.0000000  90.00000  1.000000 13.00000
## 37   K 110.00000  2.0000000 0.0000000 220.00000  1.000000 21.00000
## 38   K 110.00000  2.0000000 1.0000000 125.00000  1.000000 11.00000
## 39   K 110.00000  2.0000000 0.0000000 290.00000  0.000000 22.00000
## 40   K 110.00000  6.0000000 0.0000000 230.00000  1.000000 16.00000
## 41   K 100.00000  8.0000000 0.0000000 280.00000 28.000000 16.00000
## 42   K 100.00000  2.0000000 0.0000000 290.00000  1.000000 21.00000
## 43   K 100.00000  3.0000000 0.0000000 320.00000  1.000000 20.00000
## 44   N 212.12121 12.1212121 3.0303030 393.93939 30.303030 15.15152
## 45   N 134.32836  4.4776119 0.0000000   0.00000  5.970149 28.35821
## 46   N 134.32836  4.4776119 0.0000000   0.00000  4.477612 29.85075
## 47   P 440.00000 12.0000000 0.0000000 680.00000 12.000000 68.00000
## 48   P 363.63636  9.0909091 9.0909091 227.27273  9.090909 39.39394
## 49   P 179.10448  4.4776119 2.9850746 238.80597  7.462687 17.91045
## 50   P 179.10448  4.4776119 1.4925373 298.50746  8.955224 16.41791
## 51   P 146.66667  1.3333333 1.3333333 180.00000  0.000000 17.33333
## 52   P 134.32836  4.4776119 0.0000000 313.43284  7.462687 19.40299
## 53   P 113.63636  2.2727273 0.0000000  51.13636  0.000000 12.50000
## 54   P 113.63636  3.4090909 1.1363636 159.09091  3.409091 17.04545
## 55   P  82.70677  0.7518797 0.0000000 135.33835  0.000000 10.52632
## 56   Q 200.00000  8.0000000 2.0000000 270.00000  4.000000 28.00000
## 57   Q 160.00000  1.3333333 2.6666667 293.33333  0.000000 16.00000
## 58   Q 149.25373  5.9701493 2.9850746 223.88060  2.985075 17.91045
## 59   Q 120.00000  1.0000000 2.0000000 220.00000  1.000000 12.00000
## 60   Q  50.00000  1.0000000 0.0000000   0.00000  0.000000 13.00000
## 61   R 149.25373  4.4776119 1.4925373 343.28358  4.477612 25.37313
## 62   R 134.32836  2.9850746 1.4925373 298.50746  5.970149 22.38806
## 63   R 133.33333  2.6666667 0.0000000 253.33333  1.333333 24.00000
## 64   R 110.00000  2.0000000 0.0000000 280.00000  0.000000 22.00000
## 65   R  97.34513  0.8849558 0.0000000 212.38938  0.000000 20.35398
##       sugars shelf potassium vitamins
## 1  20.000000     3 240.00000 enriched
## 2  14.000000     3 210.00000 enriched
## 3  16.000000     3 280.00000 enriched
## 4  10.666667     3 133.33333 enriched
## 5  12.000000     2  60.00000 enriched
## 6  13.333333     1  93.33333 enriched
## 7  12.000000     2  60.00000 enriched
## 8  13.333333     1 120.00000 enriched
## 9   4.000000     3  80.00000 enriched
## 10 10.666667     1  80.00000 enriched
## 11 14.000000     3 230.00000     100%
## 12 13.333333     3 160.00000 enriched
## 13 13.000000     2  55.00000 enriched
## 14 13.000000     2  65.00000 enriched
## 15 12.000000     2  55.00000 enriched
## 16  3.000000     3  35.00000     100%
## 17 12.000000     2  25.00000 enriched
## 18  6.000000     1  90.00000 enriched
## 19  3.000000     3 110.00000     100%
## 20  3.000000     1 110.00000 enriched
```

```
## 21  0.800000     1  84.00000 enriched
## 22  2.000000     2  26.66667 enriched
## 23 19.402985     3 238.80597 enriched
## 24 14.000000     3 320.00000 enriched
## 25 15.151515     3 969.69697 enriched
## 26 10.447761     3 194.02985 enriched
## 27 12.000000     3 126.66667     100%
## 28 12.000000     3 220.00000 enriched
## 29 17.910448     3 283.58209 enriched
## 30 13.432836     2  59.70149 enriched
## 31 16.000000     2 320.00000 enriched
## 32 14.666667     1  33.33333 enriched
## 33 20.000000     2  53.33333 enriched
## 34  8.750000     2 125.00000 enriched
## 35 14.000000     2  30.00000 enriched
## 36 12.000000     2  20.00000 enriched
## 37  3.000000     3  30.00000 enriched
## 38 13.000000     2  30.00000 enriched
## 39  3.000000     1  35.00000 enriched
## 40  3.000000     1  55.00000 enriched
## 41  0.000000     3 660.00000 enriched
## 42  2.000000     1  35.00000 enriched
## 43  3.000000     3  45.00000     100%
## 44 18.181818     3 848.48485 enriched
## 45  0.000000     1 208.95522     none
## 46  0.000000     1 179.10448     none
## 47 12.000000     3 360.00000 enriched
## 48 12.121212     3 303.03030 enriched
## 49 14.925373     3 298.50746 enriched
## 50 20.895522     3 388.05970 enriched
## 51 16.000000     2  33.33333 enriched
## 52  7.462687     3 283.58209 enriched
## 53 17.045455     1  45.45455 enriched
## 54  5.681818     3  96.59091 enriched
## 55  8.270677     1  26.31579 enriched
## 56 12.000000     3 220.00000 enriched
## 57 16.000000     2  46.66667 enriched
## 58  8.955224     2 141.79104 enriched
## 59 11.000000     2  45.00000 enriched
## 60  0.000000     3  15.00000     none
## 61  4.477612     1 171.64179 enriched
## 62  8.955224     1 186.56716 enriched
## 63  6.666667     3 106.66667 enriched
## 64  3.000000     1  25.00000 enriched
## 65  1.769912     1  26.54867 enriched
```

E. Reshape

Using the *'reshape'* function to go from a long dataset to a wide dataset:

```
health <- data.frame(id = rep(1:10, each = 4, len = 40), trial = rep(c(1:4), 10), score = rnorm(40, 3
health[1:10, ]
```

```
##    id trial    score
```

```
## 1   1     1 4.140982
## 2   1     2 3.020545
## 3   1     3 2.975466
## 4   1     4 3.096974
## 5   2     1 3.023033
## 6   2     2 3.207919
## 7   2     3 2.007311
## 8   2     4 3.172049
## 9   3     1 1.693609
## 10  3     2 2.724354
```

```
  health_wide <- reshape(health, v.names = 'score', idvar = 'id', timevar = 'trial', direction = 'wide')
  health_wide
```

```
##    id  score.1  score.2   score.3  score.4
## 1   1 4.140982 3.020545 2.9754661 3.096974
## 5   2 3.023033 3.207919 2.0073114 3.172049
## 9   3 1.693609 2.724354 3.2838113 1.068860
## 13  4 2.486737 2.987145 2.7476061 2.527867
## 17  5 3.415448 2.895011 3.6017524 1.703617
## 21  6 2.471317 3.654095 0.9852765 2.565640
## 25  7 2.000525 1.352490 2.6625290 3.382465
## 29  8 2.233548 3.522468 2.5442499 2.986930
## 33  9 2.854598 3.076775 3.2429805 3.136879
## 37 10 2.348740 2.522202 1.5637669 2.699693
```

Using the *'spead'* and *'gather'* functions in the *'tidyr'* packages:

```
  library(tidyr)
```

```
## Warning: package 'tidyr' was built under R version 3.3.2
```

```
  spread(health, key = trial, value = score)
```

```
##    id        1        2         3        4
## 1   1 4.140982 3.020545 2.9754661 3.096974
## 2   2 3.023033 3.207919 2.0073114 3.172049
## 3   3 1.693609 2.724354 3.2838113 1.068860
## 4   4 2.486737 2.987145 2.7476061 2.527867
## 5   5 3.415448 2.895011 3.6017524 1.703617
## 6   6 2.471317 3.654095 0.9852765 2.565640
## 7   7 2.000525 1.352490 2.6625290 3.382465
## 8   8 2.233548 3.522468 2.5442499 2.986930
## 9   9 2.854598 3.076775 3.2429805 3.136879
## 10 10 2.348740 2.522202 1.5637669 2.699693
```

```
  gather(health_wide, key = trial, value = score, score.1:score.4)
```

```
##    id   trial     score
## 1   1 score.1 4.1409819
## 2   2 score.1 3.0230328
## 3   3 score.1 1.6936095
## 4   4 score.1 2.4867368
## 5   5 score.1 3.4154477
## 6   6 score.1 2.4713169
## 7   7 score.1 2.0005246
## 8   8 score.1 2.2335478
```

```
## 9   9 score.1 2.8545976
## 10 10 score.1 2.3487405
## 11  1 score.2 3.0205452
## 12  2 score.2 3.2079188
## 13  3 score.2 2.7243544
## 14  4 score.2 2.9871447
## 15  5 score.2 2.8950112
## 16  6 score.2 3.6540948
## 17  7 score.2 1.3524895
## 18  8 score.2 3.5224675
## 19  9 score.2 3.0767746
## 20 10 score.2 2.5222024
## 21  1 score.3 2.9754661
## 22  2 score.3 2.0073114
## 23  3 score.3 3.2838113
## 24  4 score.3 2.7476061
## 25  5 score.3 3.6017524
## 26  6 score.3 0.9852765
## 27  7 score.3 2.6625290
## 28  8 score.3 2.5442499
## 29  9 score.3 3.2429805
## 30 10 score.3 1.5637669
## 31  1 score.4 3.0969742
## 32  2 score.4 3.1720493
## 33  3 score.4 1.0688595
## 34  4 score.4 2.5278673
## 35  5 score.4 1.7036169
## 36  6 score.4 2.5656402
## 37  7 score.4 3.3824654
## 38  8 score.4 2.9869302
## 39  9 score.4 3.1368788
## 40 10 score.4 2.6996927
```

F. Merging

```
  data1 <- data.frame(id = rep(1:5, 3),
                      year = rep(2000:2002, each = 5),
                      group = sample(c('A', 'B', 'C'), 15, replace = TRUE))
  data2 <- data.frame(id = rep(1:5, each = 4), year = rep(2000:2003, 5), score = rnorm(20, 50, 15))
```

```
browseURL("http://guides.nyu.edu/quant/merge")
```

One to One, Inner Join

A one to one merge is when there is one observation (row) in dataset A that is matched up to 1 observation (row) in dataset B.

```
  data_merge <- merge(data1, data2, by = c('id', 'year'))
  data_merge
```

```
##    id year group    score
## 1   1 2000     C 58.86556
## 2   1 2001     B 34.32589
```

```
## 3   1 2002     B 40.05248
## 4   2 2000     C 41.87053
## 5   2 2001     B 31.62113
## 6   2 2002     A 56.54560
## 7   3 2000     A 26.28220
## 8   3 2001     A 64.91624
## 9   3 2002     C 40.00351
## 10  4 2000     B 72.19469
## 11  4 2001     A 57.47751
## 12  4 2002     B 34.53607
## 13  5 2000     C 44.96319
## 14  5 2001     B 61.64751
## 15  5 2002     B 66.32431
```

One to One, Full Outer Join

With the all argument set to TRUE, the non matching rows are added to the dataset.

```
data_merge <- merge(data1, data2, by = c('id', 'year'), all = TRUE)
data_merge
```

```
##    id year group     score
## 1   1 2000     C  58.86556
## 2   1 2001     B  34.32589
## 3   1 2002     B  40.05248
## 4   1 2003  <NA>  50.21129
## 5   2 2000     C  41.87053
## 6   2 2001     B  31.62113
## 7   2 2002     A  56.54560
## 8   2 2003  <NA>  34.99158
## 9   3 2000     A  26.28220
## 10  3 2001     A  64.91624
## 11  3 2002     C  40.00351
## 12  3 2003  <NA>  39.83918
## 13  4 2000     B  72.19469
## 14  4 2001     A  57.47751
## 15  4 2002     B  34.53607
## 16  4 2003  <NA> 108.05264
## 17  5 2000     C  44.96319
## 18  5 2001     B  61.64751
## 19  5 2002     B  66.32431
## 20  5 2003  <NA>  47.36880
```

To do a left or right outer join, we would change the argument to be all.x = TRUE or all.y = TRUE. The code would be the same for both the one to many and many to many merges.

Using the dplyr package

```
library(dplyr)
inner_merge <- data1 %>% inner_join(data2, by = c("id", "year"))
outer_merge <- data1 %>% full_join(data2, by = c("id", "year"))
```

G. Apply Functions and Aggregate Statistics

The *'apply'* function applies the functions over margins of an array or matrix. When the margin argument is set to 1, the functions is run across rows and when margin is set to 2, the function is run across columns.

```
  apply(UScereal[, c(2:8, 9)], MARGIN = 1, FUN = mean)
##                               100% Bran
##                                85.98106
##                                All-Bran
##                               135.22348
##             All-Bran with Extra Fiber
##                               54.37500
##               Apple Cinnamon Cheerios
##                               52.79167
##                             Apple Jacks
##                               33.12500
##                                Basic 4
##                               62.54167
##                              Bran Chex
##                               59.45336
##                             Bran Flakes
##                               61.19590
##                            Cap'n'Crunch
##                               61.41667
##                               Cheerios
##                               42.92500
##                 Cinnamon Toast Crunch
##                               59.58333
##                               Clusters
##                               69.62500
##                             Cocoa Puffs
##                               39.87500
##                              Corn Chex
##                               52.25000
##                             Corn Flakes
##                               52.12500
##                              Corn Pops
##                               28.62500
##                           Count Chocula
##                               39.87500
##                     Cracklin' Oat Bran
##                               69.62500
##                                Crispix
##                               45.00000
##                 Crispy Wheat & Raisins
##                               44.70833
##                             Double Chex
##                               53.04167
##                             Froot Loops
##                               33.12500
##                          Frosted Flakes
##                               56.29167
##                    Frosted Mini-Wheats
##                               20.09375
```

```
## Fruit & Fibre: Dates Walnuts and Oats
##                               58.58396
##                           Fruitful Bran
##                               73.88246
##                          Fruity Pebbles
##                               45.58333
##                            Golden Crisp
##                               24.69886
##                          Golden Grahams
##                               69.58333
##                       Grape Nuts Flakes
##                               38.30114
##                              Grape-Nuts
##                              153.37500
##                     Great Grains Pecan
##                               84.08712
##                       Honey Graham Ohs
##                               46.12500
##                     Honey Nut Cheerios
##                               64.62500
##                              Honey-comb
##                               29.82425
##                  Just Right Fruit & Nut
##                               57.87500
##                                    Kix
##                               33.33333
##                                   Life
##                               51.74254
##                            Lucky Charms
##                               39.87500
##                    Mueslix Crispy Blend
##                               65.30037
##                    Multi-Grain Cheerios
##                               43.37500
##                       Nut&Honey Crunch
##                               63.12313
##               Nutri-Grain Almond-Raisin
##                               74.25560
##                    Oatmeal Raisin Crisp
##                               82.87500
##                   Post Nat. Raisin Bran
##                               66.60634
##                             Product 19
##                               56.25000
##                            Puffed Rice
##                                8.37500
##                      Quaker Oat Squares
##                               65.87500
##                             Raisin Bran
##                               61.08333
##                         Raisin Nut Bran
##                               66.87500
##                          Raisin Squares
```

```
##                               29.12500
##                             Rice Chex
##                               41.71792
##                          Rice Krispies
##                               53.50000
##                 Shredded Wheat 'n'Bran
##                               21.76679
##              Shredded Wheat spoon size
##                               21.76679
##                                 Smacks
##                               34.91667
##                             Special K
##                               45.87500
##                     Total Corn Flakes
##                               42.50000
##                     Total Raisin Bran
##                               46.25000
##                     Total Whole Grain
##                               41.12500
##                                Triples
##                               64.87500
##                                   Trix
##                               34.87500
##                            Wheat Chex
##                               66.72948
##                               Wheaties
##                               41.00000
##                    Wheaties Honey Gold
##                               56.45833
  apply(UScereal[, c(2:8, 9)], MARGIN = 2, FUN = mean)
##   calories    protein        fat     sodium      fibre      carbo
## 149.408258   3.683705   1.422538 237.838364   3.870844  19.967620
##     sugars      shelf
##  10.050842   2.169231
```

The *'lapply'* function applies a function over a list or a vector and returns a list. Other variations include *'sapply'* and *'vapply'* which differ on their return data types.

```
  lapply(UScereal[, c(2:8, 9)], FUN = sd)
```

```
## $calories
## [1] 62.41187
##
## $protein
## [1] 2.642618
##
## $fat
## [1] 1.64724
##
## $sodium
## [1] 130.6296
##
## $fibre
## [1] 6.133404
##
```

```
## $carbo
## [1] 8.468468
##
## $sugars
## [1] 5.835239
##
## $shelf
## [1] 0.8398145
```

The *'tapply'* function allows you to choose a factor variable so that you can run the function over a grouping variable. This function returns an array.

```
tapply(UScereal$calories, UScereal$mfr, summary)
```

```
## $G
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   73.33  110.00  136.70  137.80  146.70  260.00
##
## $K
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   100.0   110.0   146.7   149.7   180.0   238.8
##
## $N
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   134.3   134.3   134.3   160.3   173.2   212.1
##
## $P
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   82.71  113.60  146.70  194.80  179.10  440.00
##
## $Q
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    50.0   120.0   149.3   135.9   160.0   200.0
##
## $R
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   97.35  110.00  133.30  124.90  134.30  149.30
```

The *'by'* function applies a function to a dataframe split by factors. You can have more than one factor.

```
by(UScereal$calories, UScereal$mfr, summary)
```

```
## UScereal$mfr: G
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   73.33  110.00  136.70  137.80  146.70  260.00
## ------------------------------------------------------------
## UScereal$mfr: K
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   100.0   110.0   146.7   149.7   180.0   238.8
## ------------------------------------------------------------
## UScereal$mfr: N
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   134.3   134.3   134.3   160.3   173.2   212.1
## ------------------------------------------------------------
## UScereal$mfr: P
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
```

```
##   82.71  113.60  146.70  194.80  179.10  440.00
## ------------------------------------------------------------
## UScereal$mfr: Q
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    50.0   120.0   149.3   135.9   160.0   200.0
## ------------------------------------------------------------
## UScereal$mfr: R
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   97.35  110.00  133.30  124.90  134.30  149.30
```

```
by(UScereal$calories, list(UScereal$mfr, UScereal$shelf), summary)
```

```
## : G
## : 1
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    88.0   100.0   123.3   121.3   146.7   146.7
## ------------------------------------------------------------
## : K
## : 1
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   100.0   107.5   110.0   116.7   119.2   146.7
## ------------------------------------------------------------
## : N
## : 1
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   134.3   134.3   134.3   134.3   134.3   134.3
## ------------------------------------------------------------
## : P
## : 1
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   82.71   90.44   98.17   98.17  105.90  113.60
## ------------------------------------------------------------
## : Q
## : 1
## NULL
## ------------------------------------------------------------
## : R
## : 1
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   97.35  106.80  122.20  122.70  138.10  149.30
## ------------------------------------------------------------
## : G
## : 2
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   73.33  110.00  110.00  117.10  128.30  160.00
## ------------------------------------------------------------
## : K
## : 2
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   110.0   110.0   125.0   134.4   153.3   179.1
## ------------------------------------------------------------
## : N
## : 2
## NULL
## ------------------------------------------------------------
```

```
## : P
## : 2
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   146.7   146.7   146.7   146.7   146.7   146.7
## ------------------------------------------------------------
## : Q
## : 2
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   120.0   134.6   149.3   143.1   154.6   160.0
## ------------------------------------------------------------
## : R
## : 2
## NULL
## ------------------------------------------------------------
## : G
## : 3
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   100.0   133.3   146.7   164.8   200.0   260.0
## ------------------------------------------------------------
## : K
## : 3
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   100.0   127.3   183.3   173.6   211.3   238.8
## ------------------------------------------------------------
## : N
## : 3
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   212.1   212.1   212.1   212.1   212.1   212.1
## ------------------------------------------------------------
## : P
## : 3
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   113.6   145.5   179.1   235.0   317.5   440.0
## ------------------------------------------------------------
## : Q
## : 3
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    50.0    87.5   125.0   125.0   162.5   200.0
## ------------------------------------------------------------
## : R
## : 3
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   133.3   133.3   133.3   133.3   133.3   133.3
```

- Other apply functions include:
  - The 'eapply' function applies a function over values in an enviornment.
  - The 'mapply' function applies a function to multiple lists or vector arguments.
  - The 'rapply' function recursively applies a function to a list.

  - The *'ddply'* function in the *'plyr'* package puts the results of aggreagate statistics into a dataframe.

With the dplyr package

```
library(dplyr)
UScereal %>% group_by(mfr) %>% summarize(avg.cal = mean(calories))
```

```
## # A tibble: 6 × 2
##      mfr  avg.cal
##   <fctr>    <dbl>
## 1      G 137.7879
## 2      K 149.6710
## 3      N 160.2593
## 4      P 194.7578
## 5      Q 135.8507
## 6      R 124.8521
```

```
UScereal %>% group_by(mfr) %>% summarize(avg.cal = mean(calories), count = n())
```

```
## # A tibble: 6 × 3
##      mfr  avg.cal count
##   <fctr>    <dbl> <int>
## 1      G 137.7879    22
## 2      K 149.6710    21
## 3      N 160.2593     3
## 4      P 194.7578     9
## 5      Q 135.8507     5
## 6      R 124.8521     5
```

```
UScereal %>% group_by(mfr) %>% mutate(avg.cal = mean(calories), count = n())
```

```
## Source: local data frame [65 x 13]
## Groups: mfr [6]
##
##       mfr calories   protein      fat   sodium     fibre    carbo
##    <fctr>    <dbl>     <dbl>    <dbl>    <dbl>     <dbl>    <dbl>
## 1       N 212.1212 12.121212 3.030303 393.9394 30.303030 15.15152
## 2       K 212.1212 12.121212 3.030303 787.8788 27.272727 21.21212
## 3       K 100.0000  8.000000 0.000000 280.0000 28.000000 16.00000
## 4       G 146.6667  2.666667 2.666667 240.0000  2.000000 14.00000
## 5       K 110.0000  2.000000 0.000000 125.0000  1.000000 11.00000
## 6       G 173.3333  4.000000 2.666667 280.0000  2.666667 24.00000
## 7       R 134.3284  2.985075 1.492537 298.5075  5.970149 22.38806
## 8       P 134.3284  4.477612 0.000000 313.4328  7.462687 19.40299
## 9       Q 160.0000  1.333333 2.666667 293.3333  0.000000 16.00000
## 10      G  88.0000  4.800000 1.600000 232.0000  1.600000 13.60000
## # ... with 55 more rows, and 6 more variables: sugars <dbl>, shelf <int>,
## #   potassium <dbl>, vitamins <fctr>, avg.cal <dbl>, count <int>
```

III. Functions

Writing your own functions in R can be extremely useful.

```
<function_name> <-  function(arg1, arg2, ...) {
                      function_body
                      return(any_value_to_return)
                }
```

Objects defined within the function are local to the function.

```
addTwoNums <- function(a, b) {
  tmp <- a + b
  return(tmp)
  # Alternatively either of the below would substitue for the above
  # return(a + b)
  # a + b
}

addTwoNums(5, 7)
```

```
## [1] 12
```

You can set the arguments to have default values.

```
addTwoNums <- function(a, b = 2) {
  return(a + b)
}
addTwoNums(1)
```

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

```
addTwoNums(1, 6)
```

```
## [1] 7
```

You can set an argument to be a list of values.

```
addTwoNums(a = c(1, 3, 4))
```

```
## [1] 3 5 6
```

When you want to return more than one value from a function, you must store these values in a vector/list/data frame etc...

```
myOperations <- function(a, b) {
  add <- a + b
  subtract <- a - b
  multiply <- a * b
  divide <- a / b
  mylist <- list(add, subtract, multiply, divide)
  return(mylist)
}

myOperations(5, 10)
```

```
## [[1]]
## [1] 15
##
## [[2]]
## [1] -5
##
## [[3]]
## [1] 50
##
## [[4]]
## [1] 0.5
```

IV. If/Else Statements

Writing conditional if/else if/else statements are an important piece to writing your own functions and loops in R.

```
x <- 10
if (x > 10) {
  print ('Greater than 10')
} else if (x == 10) {
  print ('Equal to 10')
} else if (x < 10 && x >= 0) {
  print ('Between 0 and 10')
} else {
  print ('Less than 0')
}
```

```
## [1] "Equal to 10"
```

The *'ifelse'* function is great and efficient when you only have an 'either or' type condition.

```
x <- seq(1:4)
ifelse(x < 4, 'less than 4', 'more than equal to 4')
```

```
## [1] "less than 4"          "less than 4"          "less than 4"
## [4] "more than equal to 4"
```

V. Loops

A. For Loop

For loops are used to exectute repetitive over different variables/observations/values.

```
for (i in c(1, 2, 4, 5))  {
  out <- i + 1
  print(out)
}
```

```
## [1] 2
## [1] 3
## [1] 5
## [1] 6
```

B. While Loop

While loops are nice when you don't know the exact number of iterations that will occur, and you only know that you want to continue to exectute until a statement is FALSE.

```
i <- 1
while (i <= 5) {
  i <- i + 2
  print(i)
}
```

```
## [1] 3
## [1] 5
```

```
## [1] 7
```

C. Repeat Loop

Similar to a while loop, the *repeat* loop executes code until a contraint is met. The only way out of the repeat loop is the *break* statement.

```
  i <- 2
  repeat {
    print(i)
    i <- i + 2
    if (i > 6) break
  }
```

```
## [1] 2
## [1] 4
## [1] 6
```

D. How to Avoid Using Loops in R

Loops are not always the most efficient method in R. One way to avoid using loops is to first create a function and then run that function through an apply function or an ifelse statement.

Using the merged dataset, created in exericse 3.

```
  exercise2 <- read.csv('Exercise 2.csv')
  exercise3 <- read.csv('Exercise 3.csv')
  wide <- spread(exercise2, key = Treatment, value = Result)
  merged <- merge(exercise3, wide, by = 'Participant')
```

Using a for loop to create a dummy variable:

```
  for (i in 1:nrow(merged)){
    if (merged$After[i] - merged$Before[i] > 3) {
      merged$forloop[i] <- 1
    }
    else {
      merged$forloop[i] <- 0
    }
  }

merged
```

```
##    Participant    Sex Age State After Before forloop
## 1         1234   Male  45    NY    95     90       1
## 2         1491 Female  40    NY    76     75       0
## 3         2231 Female  45    NY    74     74       0
## 4         2569   Male  33    PA    91     79       1
## 5         2849 Female  32    NY    82     79       0
## 6         3334   Male  25    PA    80     79       0
## 7         3465   Male  18    NY    95     92       0
## 8         4163   Male  22    NY    93     92       0
## 9         4679   Male  61    NY    89     81       1
## 10        4781   Male  64    NY    95     88       1
## 11        4978 Female  29    PA    84     81       0
```

```
## 12        5825 Female  33    PA    88     87       0
## 13        6499 Female  37    PA    89     89       0
## 14        6579   Male  32    PA    86     81       1
## 15        6593   Male  39    PA    93     82       1
## 16        7263   Male  34    PA    74     75       0
## 17        7647   Male  46    PA    90     85       1
## 18        8888 Female  27    NY    74     69       1
## 19        8914 Female  65    PA    82     86       0
## 20        9546 Female  51    NY    97     90       1
```

Using an *'ifelse'* function to create a dummy variable:

```
merged$ifelse <- ifelse(merged$After - merged$Before > 3, 1, 0)
```

Using a for loop to clean and merge datasets together:

```
setwd('/Users/katieanderson/Desktop/Data Management in R/data')

for(data_file in list.files(path = './data')[grep('.csv$', list.files())]) {
  temp <- read.csv(file = data_file)  # Open file
  temp$X <- NULL # Get rid of the variable named 'X'
  temp$avg_var1_country_year <- ave(temp$var1, temp$country, FUN = mean)
    # Add rows to data set if it exists, if not, create the dataset to get started
    if(exists('all_data1')) {
     all_data1 <- rbind(all_data1, temp)
  } else {
      all_data1 <- temp # first dataset
  }
}
```

Using the *'lapply'* function and a user written function to clean and merge datasets together:

```
# Step1: Create a function to open and clean a data set
clean_dat <- function(filename) {
  dat <- read.csv(file = filename)
  dat$X <- NULL # get rid of the variable named 'X'
  dat$avg_var1_country_year <- ave(dat$var1, dat$country, FUN = mean) # compute the average of var1 f
  return(dat)
}

# Step2: Create a list containing all clean datasets
dat_list <- lapply(list.files()[grep('.csv$', list.files())], clean_dat)

# Step3: Merge all datasets
all_data2 <- do.call(rbind, dat_list)
```