Variable Types and Basics of Subsetting and Dealing with Data

We already have considerable experience in dealing with univariate and miltivariate data. In this section, we pull out some general principles associated to skills that you have already acquired.

We will use the data set shown below, available in `combine1516nd.csv`, as our running example. It shows the data from the NFL combine for all players from Notre Dame who participated in the combine and played in the AFL or NFL from years 2015 and 2016.

```
> options(width=80)
> combine1516nd<-read.csv("combine1516nd.csv",header = TRUE)
> combine1516nd
```

```
  Year          Player Pos Height  Wt FortyYD Vertical BenchReps BroadJump
1 2016      Sheldon Day  DT     73 293    5.07     30.0        21       102
2 2016 Keivarae Russell  CB     71 192    4.49       NA        17        NA
3 2016     C.J. Prosise  RB     72 220    4.48     35.5        NA       121
4 2016      Will Fuller  WR     72 186    4.32     33.5        10       126
5 2016   Ronnie Stanley  OT     78 312    5.20     28.5        NA        NA
6 2015     Kyle Brindza   K     73 236    5.17       NA        14        NA
7 2015       Ben Koyack  TE     77 255    4.79       NA        NA        NA
  ThreeCone Shuttle            Draftedby
1      7.44    4.50 Jacksonville Jaguars
2        NA      NA   Kansas City Chiefs
3        NA      NA     Seattle Seahawks
4      6.93    4.27       Houston Texans
5      8.03    4.90     Baltimore Ravens
6        NA      NA                 <NA>
7        NA      NA Jacksonville Jaguars
```

Common Notation

- **Case:** A case is one of several items of interest, often members of a population of people, games, days, depending on the nature of the data. It usually corresponds to a single row of the data. In our example above, each player is a case.
- **Variable** This is a measurement or characteristic of a case (it varies from case to case). The data shows shows several variables, `Name, Position, Height, Weight, Speed for the Forty Yard Dash`, etc....

Data Types: There are several ways to categorize data. We will refer to the following data types:

- **Factor Data** Some variables record categories which can be used to group data. We can use `factors` to store these variables in R and this will help us summarize the data by category if we need to. In the example above the player's position should be regarded as a factor variable. We can use factor variables to group data and make numerical summaries of the group data.

- **Character data** Some data used to characterize a case may not be suitable to categorize the data, for example player name or id, numbers or telephone numbers. **Character data and Factor data** are both descriptive data types which are considered non-numeric in nature. (for example it would not make sense to calculate the average of student id numbers).

- **Discrete numerical Data** This is data that is numerical but the possible values are distinct from each other and can be "listed" on a possibly infinite list. `BenchReps` is a discrete variable, since the number recorded is always a whole number. In the example above, height could be measured on a continuum, however, they have rounded off to the nearest inch so it is discrete.

- **Continuous Data:** Continuous data is numerical data which could concieveably come from a continuum of values. Theoretically the variable could take on any value in some interval of real numbers. We see that the speed for the `FortyYD, ThreeCone` and `Shuttle` are all continuous. `Height, Weight` and `Broadjump` are theoretically continuous, however since they have rounded the data to whole numbers they are recorded as discrete data.

- **Data and Time Data:** This data can be presented in many different ways and on different scales. It can be considered as a measurement but sometimes it is also descriptive in nature. Some software automatically processes anything remotely resembling data and time data in some default way when the data is imported and one must be careful to take note of this.

- Logical Data takes the values `TRUE` and `FALSE`. It can be trated as numerical data in the sense that if you apply a function such as `sum()`, R will automatically convert the `TRUE` values to 1 and the `FALSE` values to 0.

Word of Warning on Character and Factor Variables When R imports a csv file, it imports Character variables as factor variables. It is important to convert the character variables from factor variables if you need to work with them. To check a variable type you can use the `class()` function and you can convert a variable to another class, as shown below

```
> class(combine1516nd$Player)
```

```
[1] "factor"
```

```
> #to convert to a character variable
> combine1516nd$Player<-as.character(combine1516nd$Player)
> class(combine1516nd$Player)
```

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

Coercion We saw before that we can define vectors for two different types of data, character data and numerical data, however, we cannot mix data types in a single vector. If there is mixed data in the vector, R will automatocally coerce all of the data to the most basic type, usually character data.

```
> v<-c("2", 4,1,5)
> v
```

```
[1] "2" "4" "1" "5"
```

```
> class(v)
```

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

Note also that when importing data R sometimes converts numerical data to character data. We can check the data type of all variables in an imported file with using the `class()` function:

```
> sapply(combine1516nd,class)
```

```
       Year      Player         Pos      Height          Wt     FortyYD 
  "integer" "character"    "factor"   "integer"   "integer"   "numeric" 
   Vertical   BenchReps   BroadJump   ThreeCone     Shuttle   Draftedby 
  "numeric"   "integer"   "integer"   "numeric"   "numeric"    "factor" 
```

Missing Values: One should always consider what to do with missing values when dealing with data, sometimes we need to replace them with 0 or as in the case with our modified Borda count, we needed to replace them by a number. Sometimes we do not wish to include them at all when evaluating a reductive function such as the sum or the mean, since replacing them with 0 will distort our summary. Let's see what happens if I want to calculate the means of the numerical variables in columns 4-11.

```
> options(width=80)
> apply(combine1516nd[,4:11], 2,mean)
```

```
   Height         Wt    FortyYD   Vertical  BenchReps  BroadJump  ThreeCone
 73.714286 242.000000   4.788571         NA         NA         NA         NA
   Shuttle
        NA
```

We see that if there is a value of `NA` in a column, we cannot calculate the mean. R represents missing values by `NA` (meaning not available). If we create a numerical vector with missing data, R will give us an error message and will not create the vector. On the other hand R will accept the vector with the missing data replaced by NA(not available). Be very careful to use `NA` and not `"NA"` for missing data. This indicates that the value may exist, but is not in the data set or it may not exist.

When we import data, we can have blank spaces in character vectors and if we convert character data to numerical data (with `as.numeric`, R will automaticallly replace empty spaces to NA values

```
> #B<-c(21,17,,10, ,14, )
> #You get an error message for the above command
> B<-c(21,17,NA,10,NA,14,NA)
> B

[1] 21 17 NA 10 NA 14 NA

> class(B)

[1] "numeric"

> BR<-c("21","17","" ,"10","" ,"14","")
> BR

[1] "21" "17" ""   "10" ""   "14" ""

> class(BR)

[1] "character"

> BR<-as.numeric(BR)
> BR

[1] 21 17 NA 10 NA 14 NA

> class(BR)

[1] "numeric"
```

If we try to apply a function such as `sum()` or `mean()` to a data set with missing values, R will return a value `NA` since it cannot compute the function because of the missing values.

```
> sum(BR)
```

```
[1] NA
```

Many such functions in R are defined with an argument that allows you to specify what to do with missing values. For `sum()` and `mean()` you can have R remove missing values using `na.rm`.

```
> options(width=80)
> mean(BR, na.rm=TRUE)
```

```
[1] 15.5
```

```
> apply(combine1516nd[,4:11], 2,mean, na.rm=TRUE)
```

```
   Height         Wt    FortyYD   Vertical  BenchReps  BroadJump  ThreeCone
73.714286 242.000000   4.788571  31.875000  15.500000 116.333333   7.466667
  Shuttle
 4.556667
```

NULL is a value reserved for a situation where some requested action is undefined or unavailable. For example if I create a function with no formula:

```
> f<-function(x){}
> f(BR)
```

```
NULL
```

`NaN` (not a number) will be returned when the calculation leads to a number that is not defined. $\pm$`Inf` will be returned if the result of a calculation is not defined but is $\pm\infty$.

```
> sqrt(-2)
```

```
[1] NaN
```

```
> 0/0
```

```
[1] NaN
```

```
> 1/0
```

```
[1] Inf
```

```
> -1/0
```

```
[1] -Inf
```

Assignment Functions The `names()` function can be used in 2 ways. it can be used to set or assign names to the variiables in a data set and it can be used to get names of the variables in a data set.

```
> names(combine1516nd)
```

```
 [1] "Year"      "Player"    "Pos"       "Height"    "Wt"        "FortyYD"
 [7] "Vertical"  "BenchReps" "BroadJump" "ThreeCone" "Shuttle"   "Draftedby"
```

```
> a<-c(1,2,3)
> b<-c(4,5,6)
> d<-c(7,8,9)
> df<-data.frame(a,b,d)
> df
```

```
  a b d
1 1 4 7
2 2 5 8
3 3 6 9
```

```
> names(df)<-c("x","y","z")
> df
```

```
  x y z
1 1 4 7
2 2 5 8
3 3 6 9
```

There are a number of functions in R with this double feature. Note the set feature is characterized by the format `fun(x)<-value` whereas the get feature looks like `fun(x)`.

Indexing: To access the `k` th item in a vector `x`, we use square brackets `x[k]`. To access several items simultaneously, we use a vector of indices and to access all but a specified set of indices, we use the - notation:

```
> H<-c(1,2,3,4,5,6,7,8,9,10)
> H[4]
```

```
[1] 4

> H[c(2,4,5)]

[1] 2 4 5

> H[-4]

[1]  1  2  3  5  6  7  8  9 10

> H[-c(2,4,5)]

[1]  1  3  6  7  8  9 10

> H[10]

[1] 10

> H[]

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

If we do not specify an index, we get the full vector.

Subsets of a data frame To get the (i, j) entry of a data frame `df` (in row i and column j), we use `df[i,j]`, to look at the ith row, we use `df[i,]` and to look at the jth column, we use `df[,j]`

```
> combine1516nd[1,2]

[1] "Sheldon Day"

> combine1516nd[1,]

  Year      Player Pos Height  Wt FortyYD Vertical BenchReps BroadJump
1 2016 Sheldon Day  DT     73 293    5.07       30        21       102
  ThreeCone Shuttle             Draftedby
1      7.44     4.5 Jacksonville Jaguars

> combine1516nd[,2]

[1] "Sheldon Day"      "Keivarae Russell" "C.J. Prosise"     "Will Fuller"
[5] "Ronnie Stanley"   "Kyle Brindza"     "Ben Koyack"
```

To get columns 2,4, and 6, we use `v<-c(2,4,6)` and `df[,v]`. Likewise to get rows 2,4, and 6, we use `df[v,]`.

```
> options(width=60)
> v<-c(2,4,6)
> combine1516nd[,v]

            Player Height FortyYD
1      Sheldon Day     73    5.07
2 Keivarae Russell     71    4.49
3     C.J. Prosise     72    4.48
4      Will Fuller     72    4.32
5   Ronnie Stanley     78    5.20
6     Kyle Brindza     73    5.17
7       Ben Koyack     77    4.79

> combine1516nd[v,]

  Year           Player Pos Height  Wt FortyYD Vertical
2 2016 Keivarae Russell  CB     71 192    4.49       NA
4 2016      Will Fuller  WR     72 186    4.32     33.5
6 2015     Kyle Brindza   K     73 236    5.17       NA
  BenchReps BroadJump ThreeCone Shuttle          Draftedby
2        17        NA        NA      NA Kansas City Chiefs
4        10       126      6.93    4.27     Houston Texans
6        14        NA        NA      NA               <NA>
```

We can also call out columns and rows by name; To look at the players' names and Height and weight, we could us

```
> combine1516nd[,c("Player","Height","Wt")]

            Player Height  Wt
1      Sheldon Day     73 293
2 Keivarae Russell     71 192
3     C.J. Prosise     72 220
4      Will Fuller     72 186
5   Ronnie Stanley     78 312
6     Kyle Brindza     73 236
7       Ben Koyack     77 255
```

Using Logical statements to subset We can also pick out the cases which satisfy cretain conditions with logical statements. I have attached 3 pages of a book entitled "A First Course in Statistical Programming with R" by *W. Braun and D. Murdoch*. Which explains a bit about Boolean Algebra and Logical Operations in R. Recall our table of operators

Arithmetic Operators

Operator	Description
+	addition
-	subtraction
*	multiplication
/	division
^ or **	exponentiation
x %% y	modulus (x mod y) 5%%2 is 1
x %/% y	integer division 5%/%2 is 2

Logical Operators

Operator	Description
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	exactly equal to
!=	not equal to
!x	Not x
x | y	x OR y
x & y	x AND y
isTRUE(x)	test if X is TRUE

For example, if I want to make a new data frame conatining only the data for players with height above 75 in, I use the following statements top subset the data:

```
> options(width=80)
> c1<-combine1516nd[combine1516nd$Height>75,]
> c1

  Year          Player Pos Height  Wt FortyYD Vertical BenchReps BroadJump
5 2016 Ronnie Stanley  OT     78 312    5.20     28.5        NA        NA
7 2015     Ben Koyack  TE     77 255    4.79       NA        NA        NA
  ThreeCone Shuttle             Draftedby
5      8.03     4.9     Baltimore Ravens
7        NA      NA Jacksonville Jaguars
```

Players with height above 75 in <u>and</u> time for the forty yard dash < 5sec is given by

```
> options(width=60)
> c2<-combine1516nd[(combine1516nd$Height>75)&(combine1516nd$FortyYD<5),]
> c2
```

```
  Year     Player Pos Height  Wt FortyYD Vertical BenchReps
7 2015 Ben Koyack  TE     77 255    4.79       NA        NA
  BroadJump ThreeCone Shuttle             Draftedby
7        NA        NA      NA Jacksonville Jaguars
```

Players with height above 75 in or time for the forty yard dash < 5sec is given by

```
> options(width=60)
> c2<-combine1516nd[(combine1516nd$Height>75)|(combine1516nd$FortyYD<5),]
> c2
```

```
  Year           Player Pos Height  Wt FortyYD Vertical
2 2016 Keivarae Russell  CB     71 192    4.49       NA
3 2016     C.J. Prosise  RB     72 220    4.48     35.5
4 2016      Will Fuller  WR     72 186    4.32     33.5
5 2016   Ronnie Stanley  OT     78 312    5.20     28.5
7 2015       Ben Koyack  TE     77 255    4.79       NA
  BenchReps BroadJump ThreeCone Shuttle
2        17        NA        NA      NA
3        NA       121        NA      NA
4        10       126      6.93    4.27
5        NA        NA      8.03    4.90
7        NA        NA        NA      NA
              Draftedby
2   Kansas City Chiefs
3     Seattle Seahawks
4       Houston Texans
5     Baltimore Ravens
7 Jacksonville Jaguars
```

Assigning new values to parts of a vector: We can assign the value `m` to the `k`th position in a vector `x` using the command `x[k]<-m`.

```
> x<-c(3, 2, 1, 5)
> x[3]<-14
> x
```

```
[1]  3  2 14  5
```

We can assign values to multiple positions simultaneously with vectors of indices:

```
> x[c(1,4)]<-c(20,17)
> x
```

```
[1] 20  2 14 17
```

We can also reduce the vector `x` to a subset of itself by redefining it

```
> x<-x[5:7]
> x
```

```
[1] NA NA NA
```

We set several values of a vector equal to a single value by just specifying the single value. (When the right hand side ahs less values than needed R recycles them):

```
> y<-1:10
> y
```

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

```
> y[1:5]<-1
> y
```

```
 [1]  1  1  1  1  1  6  7  8  9 10
```

```
> y[6:9]<-c(2,4)
> y
```

```
 [1]  1  1  1  1  1  2  4  2  4 10
```

```
> y[6:10]<-c(3,5)
> y
```

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

As we saw when using the modified Borda count, we can replace the missing values in a vector with preferred values, bu subsetting the vector with a logical statement.

```
> z<-c(1,NA,2,NA,3,4,5)
> z
```

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

```
> is.na(z)
```

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

```
> z[is.na(z)]
```

```
[1] NA NA
```

```
> z[is.na(z)]<-0
> z
```

```
[1] 1 0 2 0 3 4 5
```

As you can see, R creates a logical vector with the command `is.na(z)` and `z[is.na(z)]` gives the positions of the `TRUE` values in that vector. The command `z[is.na(z)]<-0` replaces the values in the positions indicated by the vector `z[is.na(z)]` by 0. We juat need the last command here to replace the `NA` values, but we can understand what is going opn behind the scenes by examining the vectors created in each step.

Categorical Data Types: Character vs Factor data.

Character Data: We already know how to create character vectors:

```
> v<-c("Math", "is", "awesome")
> v
```

```
[1] "Math"    "is"      "awesome"
```

Factors: Factors are used to categorize data. we can create a factor with the `factor()` function. For example the following factor categorizes the subject.

```
> Name<-c("Tom","Peyton","Ely","Jerry","Randy")
> class(Name)
```

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

```
> Pos<-factor(c("QB","QB","QB","WR","WR"))
> class(Pos)
```

```
[1] "factor"
```

```
> Pos
```

```
[1] QB QB QB WR WR
Levels: QB WR
```

```
> FortyYD<-c(4.9,5.1,5.2,4.9,4.1)
> class(FortyYD)
```

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

```
> Vertical<-c(30,35.5,32.1,39.1,33.1)
> class(Vertical)
```

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

```
> dreamteam<-data.frame(Name,Pos,FortyYD,Vertical)
> dreamteam
```

```
    Name Pos FortyYD Vertical
1    Tom  QB     4.9     30.0
2 Peyton  QB     5.1     35.5
3    Ely  QB     5.2     32.1
4  Jerry  WR     4.9     39.1
5  Randy  WR     4.1     33.1
```

We see that when we call a factor vector, we get the vector along with a list of the **levels** of the factor vector. The levels are just the possible values of the factor. We can get a list of the levels of a factor vector with the function `levels()`.

```
> levels(Pos)
```

```
[1] "QB" "WR"
```

We cannot change a value in a factor vector unless we are changing it to an already existing level. If we try, `R` will insert a value of `NA`. Suppose we want to switch Tom to Running Back position (RB).

```
> Pos[1]<-"RB"
> Pos
```

```
[1] <NA> QB   QB   WR   WR
Levels: QB WR
```

In order to make the above change, we must first add a level. (We first rectify the damage we have done :))

```
> #switch back to original Pos vector
> Pos<-factor(c("QB","QB","QB","WR","WR"))
> class(Pos)
```

```
[1] "factor"
```

```
> #Add a new level and switch Tomn's playing Position
> levels(Pos)<-c(levels(Pos),"RB")
> levels(Pos)
```

```
[1] "QB" "WR" "RB"

> Pos

[1] QB QB QB WR WR
Levels: QB WR RB

> Pos[1]<-"RB"
> Pos

[1] RB QB QB WR WR
Levels: QB WR RB
```

To redefine the levels(switch the labels to new ones) we can specify the new levels as a vector with the same length as the old vector and ordered appropriately so that the position of the new levels match the position of their predecessor. Suppose we want to use the labels `"QBack"` and `"WideR"` instead of `"QB"` and `"WR"`, we redefine the labels as follows:

```
> #switch back to original Pos vector
> Pos<-factor(c("QB","QB","QB","WR","WR"))
> class(Pos)

[1] "factor"

> levels(Pos)

[1] "QB" "WR"

> #Switch names in the levels vector
> levels(Pos)<-c("QBack","WideR")
> Pos

[1] QBack QBack QBack WideR WideR
Levels: QBack WideR

> levels(Pos)

[1] "QBack" "WideR"
```

The `tapply` function. One of the nice things about factors is that you can apply functions by factor. For example, if I want to get a list of the means of the variable `FortyYD` by position in the data frame `dreamteam` above, I can use the `tapply` function from the `apply` group.

```
> r1<-with(dreamteam, tapply(FortyYD, Pos, mean))
> r1

      QB       WR
5.066667 4.500000
```

The `with` command above, saves you the bother of writing the name of the data frame each time you call one of the variables. You can also use `tapply` as follows:

```
> tapply(dreamteam$Vertical,dreamteam$Pos,mean)

      QB       WR
32.53333 36.10000
```

Logical Data: We have already considered logical data which takes either of two values `TRUE` or `FALSE`. Logical data is produced by a number of `is` functions and comaprison operators.

```
> is.numeric("Name")

[1] FALSE

> is.na(4)

[1] FALSE

> 3<4

[1] TRUE

> "two"==2

[1] FALSE

> sqrt(3)*sqrt(3)==3

[1] FALSE

> all.equal(sqrt(3)*sqrt(3),3)

[1] TRUE
```

We can also check which entries in a vector satisfy a logical statement

```
> Wt<-combine1516nd$Wt
> Wt>200

[1]  TRUE FALSE  TRUE FALSE  TRUE  TRUE  TRUE

> Wt==220

[1] FALSE FALSE  TRUE FALSE FALSE FALSE FALSE

> Wt>200&Wt<250

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

The functions `any`, `all`, `which`, and `%in%`. The functions `any` and `all` return whether any or all of the values in a vector satisfy a condition.

```
> any(Wt>250)

[1] TRUE

> all(Wt>250)

[1] FALSE
```

The `which` function tells us which indices satisfy the condition

```
> which(Wt<250&Wt>200)

[1] 3 6

> combine1516nd$Player[c(3,6)]

[1] "C.J. Prosise" "Kyle Brindza"

> #we could just do all of this with one command
> combine1516nd$Player[which(combine1516nd$Wt<250&combine1516nd$Wt>200)]

[1] "C.J. Prosise" "Kyle Brindza"

> #or something very compact:
> with(combine1516nd,Player[which(Wt<250&Wt>200)])

[1] "C.J. Prosise" "Kyle Brindza"
```

To check if a value is in a vector, we use `%in%`, `any` or `match` (which works for more than one value).

```
> 210 %in% Wt

[1] FALSE

> any(210==Wt)

[1] FALSE

> match(c(293,236),Wt)

[1] 1 6
```

We can apply numerical functions to logical vectors. In this case `R` coerces the logical vector to a numerical vector with 1 replacing `TRUE` and 0 replacing `FALSE`. This is especially useful if you wish to count the number of entries in a vector that satisfy some condition, we can sum a logical vector. For example to check how many players in our data set have a weight between 200 and 250 pounds we use

```
> sum(Wt<250&Wt>200)

[1] 2
```

As we saw before, we can use logical vectors to subset a data frame or a vector.

```
> Wt[Wt<mean(Wt)]

[1] 192 220 186 236

> combine1516nd[combine1516nd$Wt<mean(combine1516nd$Wt),]

  Year           Player Pos Height  Wt FortyYD Vertical
2 2016 Keivarae Russell  CB     71 192    4.49       NA
3 2016     C.J. Prosise  RB     72 220    4.48     35.5
4 2016      Will Fuller  WR     72 186    4.32     33.5
6 2015     Kyle Brindza   K     73 236    5.17       NA
  BenchReps BroadJump ThreeCone Shuttle           Draftedby
2        17        NA        NA      NA Kansas City Chiefs
3        NA       121        NA      NA   Seattle Seahawks
4        10       126      6.93    4.27     Houston Texans
6        14        NA        NA      NA               <NA>
```

As shown above, we can also use a logical vector to take out `NA` values.

```
> BJ<-combine1516nd$BroadJump
> BJ
```

```
[1] 102  NA 121 126  NA  NA  NA
```

```
> is.na(BJ)
```

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

```
> BJ1<-BJ[!is.na(BJ)]
> BJ1
```

```
[1] 102 121 126
```

```
> mean(BJ1)
```

```
[1] 116.3333
```

We can also use one variable to subset another.

```
> BJ<-combine1516nd$BroadJump
> BJ
```

```
[1] 102  NA 121 126  NA  NA  NA
```

```
> Wt<-combine1516nd$Wt
> Wt
```

```
[1] 293 192 220 186 312 236 255
```

```
> BJ[Wt>200]
```

```
[1] 102 121  NA  NA  NA
```

Example It is common, when making a model with a regression or otherwise, to split your data in half, and use one half to make the model and the other to test it. If your data is spreadf throughout a season, you may not wish to split the data into the first half and second half off the season, since the game may be played differently as the season progresses. In this case, it is common to take random samples or to take every second case(even numbered cases) for the model data (`run`) and the rest (odd numbered cases) for the test data (`test`).

For example to create a data set `run` with the even numbered cases of our data set `combine1516nd` , and a data set `test` containing the odd numbered cases, we can do the following:

```
> v<-1:nrow(combine1516nd)
> v
```

```
[1] 1 2 3 4 5 6 7

> v1<-v%%2==0
> v2<-v%%2!=0
> v1

[1] FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE

> v2

[1]  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE

> run<-combine1516nd[v1,]
> test<-combine1516nd[v2,]
> run

  Year           Player Pos Height  Wt FortyYD Vertical
2 2016 Keivarae Russell  CB     71 192    4.49       NA
4 2016      Will Fuller  WR     72 186    4.32     33.5
6 2015     Kyle Brindza   K     73 236    5.17       NA
  BenchReps BroadJump ThreeCone Shuttle          Draftedby
2        17        NA        NA      NA Kansas City Chiefs
4        10       126      6.93    4.27     Houston Texans
6        14        NA        NA      NA               <NA>

> test

  Year         Player Pos Height  Wt FortyYD Vertical
1 2016    Sheldon Day  DT     73 293    5.07     30.0
3 2016   C.J. Prosise  RB     72 220    4.48     35.5
5 2016 Ronnie Stanley  OT     78 312    5.20     28.5
7 2015     Ben Koyack  TE     77 255    4.79       NA
  BenchReps BroadJump ThreeCone Shuttle
1        21       102      7.44     4.5
3        NA       121        NA      NA
5        NA        NA      8.03     4.9
7        NA        NA        NA      NA
             Draftedby
1 Jacksonville Jaguars
3     Seattle Seahawks
5     Baltimore Ravens
7 Jacksonville Jaguars
```

(d) Multiply each observation by -2, and assign the result to `srm2`. Find the mean, median, range, and variance of `srm2`. How do the statistics change now?
(e) Plot a histogram of the `solar.radiation`, `sr10`, and `srm2`.

2 Calculate $\sum_{n=1}^{15} \min(2^n, n^3)$. [Hint: the `min()` function will give the wrong answer.]

3 Calculate $\sum_{n=1}^{15} \max(2^n, n^3)$.

2.7 Logical vectors and relational operators

We have used the `c()` function to put numeric vectors together as well as character vectors. R also supports logical vectors. These contain two different elements: `TRUE` and `FALSE`, as well as `NA` for missing.

2.7.1 Boolean algebra

To understand how R handles `TRUE` and `FALSE`, we need to understand a little *Boolean algebra*. The idea of Boolean algebra is to formalize a mathematical approach to logic.

Logic deals with statements that are either true or false. We represent each statement by a letter or variable, e.g. A is the statement that the sky is clear, and B is the statement that it is raining. Depending on the weather where you are, those two statements may both be true (there is a "sunshower"), A may be true and B false (the usual clear day), A false and B true (the usual rainy day), or both may be false (a cloudy but dry day).

Boolean algebra tells us how to evaluate the truth of compound statements. For example, "A and B" is the statement that it is both clear and raining. This statement is true only during a sunshower. "A or B" says that it is clear or it is raining, or both: anything but the cloudy dry day. This is sometimes called an *inclusive or*, to distinguish it from the *exclusive or* "A xor B", which says that it is either clear or raining, but *not* both. There is also the "not A" statement, which says that it is not clear.

There is a very important relation between Boolean algebra and set theory. If we interpret A and B as sets, then we can think of "A and B" as the set of elements which are in A and are in B, i.e. the intersection $A \cap B$. Similarly "A or B" can be interpreted as the set of elements that are in A or are in B, i.e. the union $A \cup B$. Finally, "not A" is the complement of A, i.e. A^c.

Because there are only two possible values (true and false), we can record all Boolean operations in a table. On the first line of Table 2.1 we list the basic Boolean expressions, on the second line the equivalent way to code them in R, and in the body of the table the results of the operations.

2.7.2 Logical operations in R

One of the basic types of vector in R holds logical values. For example, a logical vector may be constructed as

```
a <- c(TRUE, FALSE, FALSE, TRUE)
```

Table 2.1 *Truth table for Boolean operations*

Boolean R	*A* `A`	*B* `B`	not *A* `!A`	not *B* `!B`	*A* and *B* `A & B`	*A* or *B* `A | B`
	TRUE	TRUE	FALSE	FALSE	TRUE	TRUE
	TRUE	FALSE	FALSE	TRUE	FALSE	TRUE
	FALSE	TRUE	TRUE	FALSE	FALSE	TRUE
	FALSE	FALSE	TRUE	TRUE	FALSE	FALSE

The result is a vector of four logical values. Logical vectors may be used as indices:

```
b <- c(13, 7, 8, 2)
b[a]

## [1] 13  2
```

The elements of `b` corresponding to `TRUE` are selected.

If we attempt arithmetic on a logical vector, e.g.

```
sum(a)

## [1] 2
```

then the operations are performed after converting `FALSE` to 0 and `TRUE` to 1, so by summing we count how many occurrences of `TRUE` there are in the vector.

There are two versions of the Boolean operators. The usual versions are `&`, `|`, and `!`, as listed in the previous section. These are all vectorized, so we see, for example,

```
!a

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

If we attempt logical operations on a numerical vector, 0 is taken to be `FALSE`, and any non-zero value is taken to be `TRUE`:

```
a & (b - 2)

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

The operators `&&` and `||` are similar to `&` and `|`, but behave differently in two respects. First, they are *not* vectorized: only one calculation is done. Secondly, they are guaranteed to be evaluated from left to right, with the right-hand operand evaluated only if necessary. For example, if `A` is `FALSE`, then `A && B` will be `FALSE` regardless of the value of `B`, so `B` needn't be evaluated. This can save time if evaluating `B` would be very slow, and may make calculations easier, for example if evaluating `B` would cause an error when `A` was `FALSE`. This behavior is sometimes called *short-circuit evaluation*.

2.7.3 Relational operators

It is often necessary to test relations when programming. R allows testing of equality and inequality relations using the relational operators: `<`, `>`, `==`, `>=`, `<=`, and `!=`.[4] Some simple examples follow:

[4] Be careful with tests of equality. Because R works with only a limited number of decimal places rounding error can accumulate, and you may find surprising results, such as `49 * (4 / 49)` not being equal to `4`.

```
threeM <- c(3, 6, 9)
threeM > 4    # which elements are greater than 4

## [1] FALSE  TRUE  TRUE

threeM == 4   # which elements are exactly equal to 4

## [1] FALSE FALSE FALSE

threeM >= 4   # which elements are greater than or equal to 4

## [1] FALSE  TRUE  TRUE

threeM != 4   # which elements are not equal to 4

## [1] TRUE TRUE TRUE

threeM[threeM > 4] # elements of threeM which are greater than 4

## [1] 6 9

four68 <- c(4, 6, 8)
four68 > threeM # four68 elements exceed corresponding threeM elements

## [1]  TRUE FALSE FALSE

four68[threeM < four68] # print them

## [1] 4
```

Exercises

1 Use R to identify the elements of the sequence $\{2^1, 2^2, \ldots, 2^{15}\}$ that exceed the corresponding elements of the sequence $\{1^3, 2^3, \ldots, 15^3\}$.

2 More complicated expressions can be constructed from the basic Boolean operations. Write out the truth table for the *xor* operator, and show how to write it in terms of *and*, *or*, and *not*.

3 Venn diagrams can be used to illustrate set unions and intersections. Draw Venn diagrams that correspond to the *and*, *or*, *not*, and *xor* operations.

4 DeMorgan's laws in R notation are `!(A & B) == (!A) | (!B)` and `!(A | B) == (!A) & (!B)`. Write these out in English using the *A* and *B* statements above, and use truth tables to confirm each equality.

5 Evaluation of a square root is achieved using the `sqrt()` function, but a warning will be issued when the argument is negative. Consider the following code which is designed to test whether a given value is