Introduction to R

Pachalo Chizala

2023-09-24

Background

This is a living document… It gets updated as we move. You can add a section This document is developed in R with R Markdown, and hence it is an R Markdown document. Markdown is a simple formatting syntax for authoring HTML, PDF, and MS Word documents. For more details on using R Markdown see. http://rmarkdown.rstudio.com. You combine code with text. No need to copy tables from one software such as STATA and paste them to MS Word.

Introduction to R

In this journey, we seek to build our confidence working with R including data management and analysis using a statistical software called R- just R. The approach is not just to teach you R, but to use it while on the job. Learning R by skill

Let's get back to our business

Dive straight in R

Using R as a calculator

We can use R as a calculator. It follows the rules of arithmetic

```
#R works as calculator
2+8
```

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

```
2*4+6-5/2.5
```

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

```
(2*4)+6-5/2.5
```

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

```
5^2
```

```
## [1] 25
```

```
5**2
```

```
## [1] 25
```

We can use R to compare expressions including numbers and letters, i.e.

```
#Logical operations
4==2       #Is 4 equal to 2?
```

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

```r
4==4       #Is 4 equal to 4
```

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

```r
4>=3       #Is 4 greater than or equal to 3
```

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

```r
"a"=="b"   #Is a equal to b
```

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

```r
"a">"b"    #Is greater than b
```

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

Objects, functions and Variables

From time to time we are faced with situations where we have to reference to some value. For instance, you might add two number 9 and 34, and keep the result for later calculations. In R, we can keep this result in an variable object named "a". You can view the contents of this variable by just typing the variable name and place enter. Check the following examples

```r
a = 9+34
b = 4*5+7
a
```

```
## [1] 43
```

```r
b
```

```
## [1] 27
```

```r
a + b
```

```
## [1] 70
```

```r
(a+b)/a
```

```
## [1] 1.627907
```

```r
a=a^4-sqrt(a)
a
```

```
## [1] 3418794
```

R is so fluid and very flexible. Just take caution that it overwrites an existing object (or variable) without issuing a warning. You might have noted this in the last commands.

```r
b = sqrt(a*runif(1)*10)
if (a > b) {print("a is greater than b")} else {print("a is less than b")}
```

```
## [1] "a is greater than b"
```

Suppose you are tasked to collect information such as name, age and sex of your colleagues at your place of work. You collect and write them on a paper. Probably, the next question would be "how do I feed then into R?".

To this far, we need existing functions to help ease the work at hand, and (almost) all the work we will lay our hands on in R. To this far, we have already used three functions, namely, sqrt(), runif() and print(). One important but simply named function is the c() function. The c() function concatenates different entries, separated by commas, into a vector of the entries.

Now that we know the c() function, we can make use of it. Take note that you need to save your vector of values to an object of name of your choice. It is, however, important to name your objects or variables with meaningful names.

```
name <- c("Patrick", "Gregory", "Bernard", "Lesla", "Bridget", "Rico", "Temwa", "Andrew", "Cecil","Mart
age <- c(23,29,31,21,34,38,28,33,25,NA,30,35,33,30,29,NA)
sex <- c(1,1,1,2,2,1,2,1,2,2,2,1,2,2,2,1)
```

If you just want to view the entries for each variable/object, you can print the contents using the function print() or just typing name of the object.

It would definitely be useful to check further if the list matches the number of colleagues you collected data for.

```
#Print values contained in each variable name, age, and sex
print(name)
```

```
##  [1] "Patrick" "Gregory" "Bernard" "Lesla"   "Bridget" "Rico"    "Temwa"
##  [8] "Andrew"  "Cecil"   "Martha"  "Merriam" "William" "Martha"  "Mada"
## [15] "Sara"    NA
```

```
age
```

```
##  [1] 23 29 31 21 34 38 28 33 25 NA 30 35 33 30 29 NA
```

```
sex
```

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

```
#Returns number of items
length(name)
```

```
## [1] 16
```

```
length(age)
```

```
## [1] 16
```

```
length(sex)
```

```
## [1] 16
```

Data frames

Your thoughts are as good as mine if you ever wondered if it were possible to have variables, name, age and sex appear in a structured spreadsheet-like display. Possibly you would have guessed to use the same c() function to concatenate the variables… as in c(name, age, sex).. but this won't give the required result. And since c() only stores values of the same type, you can guess what type of values you will now have.

There are, of course, a number of functions that combine vectors including cbind() and rbind(), but one you will likely come to work more with is the data.frame() function. Let's dive into it…

```
cbind(name, age, sex)       #Combines the variables column-wise
```

```
##       name      age  sex
##  [1,] "Patrick" "23" "1"
##  [2,] "Gregory" "29" "1"
##  [3,] "Bernard" "31" "1"
##  [4,] "Lesla"   "21" "2"
##  [5,] "Bridget" "34" "2"
##  [6,] "Rico"    "38" "1"
##  [7,] "Temwa"   "28" "2"
##  [8,] "Andrew"  "33" "1"
##  [9,] "Cecil"   "25" "2"
## [10,] "Martha"  NA   "2"
## [11,] "Merriam" "30" "2"
## [12,] "William" "35" "1"
## [13,] "Martha"  "33" "2"
## [14,] "Mada"    "30" "2"
## [15,] "Sara"    "29" "2"
## [16,] NA        NA   "1"
```

```
rbind(name, age, sex)       #Combines the variables low-wise.
```

```
##      [,1]      [,2]      [,3]      [,4]    [,5]      [,6]   [,7]    [,8]    
## name "Patrick" "Gregory" "Bernard" "Lesla" "Bridget" "Rico" "Temwa" "Andrew"
## age  "23"      "29"      "31"      "21"    "34"      "38"   "28"    "33"    
## sex  "1"       "1"       "1"       "2"     "2"       "1"    "2"     "1"     
##      [,9]    [,10]    [,11]     [,12]     [,13]    [,14]  [,15]  [,16]
## name "Cecil" "Martha" "Merriam" "William" "Martha" "Mada" "Sara" NA   
## age  "25"    NA       "30"      "35"      "33"     "30"   "29"   NA   
## sex  "2"     "2"      "2"       "1"       "2"      "2"    "2"    "1"  
```

```
data.frame(name, age, sex)  #Combines the variables column-wise. This is the best.
```

```
##       name age sex
## 1  Patrick  23   1
## 2  Gregory  29   1
## 3  Bernard  31   1
## 4    Lesla  21   2
## 5  Bridget  34   2
## 6     Rico  38   1
## 7    Temwa  28   2
## 8   Andrew  33   1
## 9    Cecil  25   2
## 10  Martha  NA   2
## 11 Merriam  30   2
## 12 William  35   1
## 13  Martha  33   2
## 14    Mada  30   2
## 15    Sara  29   2
## 16    <NA>  NA   1
```

```
#Saving the data frame to an object called mydata
mydata <- data.frame(name, age, sex)
mydata
```

```
##       name age sex
## 1  Patrick  23   1
## 2  Gregory  29   1
## 3  Bernard  31   1
## 4    Lesla  21   2
## 5  Bridget  34   2
## 6     Rico  38   1
## 7    Temwa  28   2
## 8   Andrew  33   1
## 9    Cecil  25   2
## 10  Martha  NA   2
## 11 Merriam  30   2
## 12 William  35   1
## 13  Martha  33   2
## 14    Mada  30   2
## 15    Sara  29   2
## 16    <NA>  NA   1
```

Now we see better! In R, NA implies a missing value (Not applicable). Currently, we would say that we don't yet have Martha's age, and there is a certain male with no name and age. We will likely ask you to go back and fetch for these values!

Again, note now that we have two Marthas, and, obviously, it will not be easy to distinguish them using names. However, we can assign each person a unique identification number.

```
id <- 1:length(name)
mydata <- data.frame(id,mydata)
mydata
```

```
##    id    name age sex
## 1   1 Patrick  23   1
## 2   2 Gregory  29   1
## 3   3 Bernard  31   1
## 4   4   Lesla  21   2
## 5   5 Bridget  34   2
## 6   6    Rico  38   1
## 7   7   Temwa  28   2
## 8   8  Andrew  33   1
## 9   9   Cecil  25   2
## 10 10  Martha  NA   2
## 11 11 Merriam  30   2
## 12 12 William  35   1
## 13 13  Martha  33   2
## 14 14    Mada  30   2
## 15 15    Sara  29   2
## 16 16    <NA>  NA   1
```

I hope you are able to follow what's happening here! Let me repeat the statement I once said above, "R is so fluid and very flexible"; you can achieve one task a million ways. Of course, this can also be a little confusing at times. We will soon look at how we can achieve the same task in a different way. For now, I would encourage that you experiment the use of a:b, seq(a,b,c) where a,b and c are numeric values.

But now, we have each person assigned a unique identification number.

Having a feel of our data

Now we have our data, but how do we get a feel of it? If you were handed down a data set,you would want to know what type of data that it is, what variables are in it, and number of cases it contains.

The str() function re

```
str(mydata)
```

```
## 'data.frame':    16 obs. of  4 variables:
##  $ id  : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ name: chr  "Patrick" "Gregory" "Bernard" "Lesla" ...
##  $ age : num  23 29 31 21 34 38 28 33 25 NA ...
##  $ sex : num  1 1 1 2 2 1 2 1 2 2 ...
```

```
dim(mydata)
```

```
## [1] 16  4
```

```
names(mydata)
```

```
## [1] "id"   "name" "age"  "sex"
```

```
head(mydata)      #print the first default 6 rows
```

```
##   id    name age sex
## 1  1 Patrick  23   1
## 2  2 Gregory  29   1
## 3  3 Bernard  31   1
## 4  4   Lesla  21   2
## 5  5 Bridget  34   2
## 6  6    Rico  38   1
```

```
head(mydata,5)    #print the first n=5 rows
```

```
##   id    name age sex
## 1  1 Patrick  23   1
## 2  2 Gregory  29   1
## 3  3 Bernard  31   1
## 4  4   Lesla  21   2
## 5  5 Bridget  34   2
```

```
tail(mydata)
```

```
##    id    name age sex
## 11 11 Merriam  30   2
## 12 12 William  35   1
## 13 13  Martha  33   2
## 14 14    Mada  30   2
## 15 15    Sara  29   2
## 16 16    <NA>  NA   1
```

```
tail(mydata,3)
```

```
##    id name age sex
## 14 14 Mada  30   2
## 15 15 Sara  29   2
## 16 16 <NA>  NA   1
```

Work practice

Suppose that your friend asks you to assign the first 10 of your colleagues to department A and the rest to department B. Sure you can achieve this a number of ways, but your final output should use the function rep(). Without any explanation, you may compare with the following:

```
mydata <- data.frame(mydata,dept=rep(c("A","B"),c(10,6)))
```

Labeling the levels (categories) of sex.

```
mydata$sex <- factor(mydata$sex, levels = c(1,2), labels = c("Male","Female"))
```

You must remember to remove objects that you are not working on

```
rm(age, name, sex, a, b)
```

Indexing data frame entries

Look at the data sheet of our mydata. It sure looks like a matrix, one with rows and columns. The rows represents cases.... How would we find the name that is in row number 4?

```
mydata[4,2]
```

```
## [1] "Lesla"
```

You likely may be wondering why mydata[4,2]. The 4 is for the row number, as you might have guessed, and 2 is the column number. Names are in column 2. If we want all information about person on the fourth, all we need to do is never specify the row number! That makes sense, right? Yeah, I know, R is intuitive.

And of course, we are not restricted to getting one row only. Try and practice the following until you get the idea:

```
mydata[c(5,4,15),]
```

```
##    id    name age    sex dept
## 5   5 Bridget  34 Female    A
## 4   4   Lesla  21 Female    A
## 15 15    Sara  29 Female    B
```

```
mydata[c(5,4,1),c(2,4)]
```

```
##      name    sex
## 5 Bridget Female
## 4   Lesla Female
## 1 Patrick   Male
```

And of course, it may sometimes not be practical to keep referencing to row and column numbers. It is possible to specify names of the column, but since rows doesn't have names, we will have to stick to row numbers or other ways.

```
mydata[1:3,"name"]
```

```
## [1] "Patrick" "Gregory" "Bernard"
```

```
mydata[1:3,c("name", "sex")]
```

```
##      name  sex
## 1 Patrick Male
## 2 Gregory Male
## 3 Bernard Male
```

```
mydata[,"name"]
```

```
##  [1] "Patrick" "Gregory" "Bernard" "Lesla"   "Bridget" "Rico"    "Temwa"
##  [8] "Andrew"  "Cecil"   "Martha"  "Merriam" "William" "Martha"  "Mada"
## [15] "Sara"    NA
```

Accessing variables inside a data frame

Many a times we are so interested in manipulating values of variables contained in a data frame. Of course, we have noticed mydata[,"name"] does the job, however, there is a better way! Here is how! We must specify the name of the data frame, append a dollar sign ($), and then the variable name, i.e. mydata$age.

```
mydata[,"age"]
```

```
##  [1] 23 29 31 21 34 38 28 33 25 NA 30 35 33 30 29 NA
```

```
mydata$age
```

```
##  [1] 23 29 31 21 34 38 28 33 25 NA 30 35 33 30 29 NA
```

```
mydata$name[4]
```

```
## [1] "Lesla"
```

```
#mean(age)   #This won't work now! Why? And what's your guess on this one below?
mean(mydata[,"age"])
```

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

```
mean(mydata[,"age"], na.rm = T) #The use of na.rm has been emphasized enough
```

```
## [1] 29.92857
```

```
mean(mydata["age"], na.rm = T)  #Why won't this work?
```

```
## Warning in mean.default(mydata["age"], na.rm = T): argument is not numeric or
## logical: returning NA
```

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

An alternative (probably the the best) way to reference a variable in a data frame, is to use a dollar ($) sign as below

```
mydata$age
```

```
##  [1] 23 29 31 21 34 38 28 33 25 NA 30 35 33 30 29 NA
```

```
head(mydata$age,7)
```

```
## [1] 23 29 31 21 34 38 28
```

```
mydata$age[6]
```

```
## [1] 38
```

```
mydata$age[c(6,3)]
```

```
## [1] 38 31
```

```
mean(mydata$age, na.rm = T)
```

```
## [1] 29.92857
```

```
table(mydata$sex)
```

```
## 
##   Male Female 
##      7      9
```

```
#Table of proportions

prop.table(1)
```

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

```
prop.table(c(1,1))
```

```
## [1] 0.5 0.5
```

```
prop.table(c(1,1,2))  #I hope now you get the idea. Check this
```

```
## [1] 0.25 0.25 0.50
```

```
1/sum(c(1,1,2)) #First element
```

```
## [1] 0.25
```

```
1/sum(c(1,1,2)) #Second elemenet
```

```
## [1] 0.25
```

```
2/sum(c(1,1,2)) #Third element
```

```
## [1] 0.5
```

```
c(1,1,2)/4      #How cool!!
```

```
## [1] 0.25 0.25 0.50
```

```
c(1,1,2)/sum(c(1,1,2))
```

```
## [1] 0.25 0.25 0.50
```

```
a <- c(1,1,2)
a/sum(a)
```

```
## [1] 0.25 0.25 0.50
```

Let's get back to our problem

```
prop.table(c(1,1,2))
```

```
## [1] 0.25 0.25 0.50
```

```
prop.table(a)
```

```
## [1] 0.25 0.25 0.50
```

Just doing the same thing over and over. The idea is, if you pass a vector of numbers to prop.table, the function will calculate the the proportion of each value to the sum of all elements in the vector. Assume that 1 represents "male" and 2 represents "female". This means, we have 2 males and 1 female. So, we can get the proportions as below.

```
prop.table(c(2,1))  #Does this make sense?
```

```
## [1] 0.6666667 0.3333333
```

We know we can get these counts of 2 males and 1 female using table function

```
table(c(1,1,2))      #Don't get confused with the output. Then,
```

```
##
## 1 2
## 2 1
```

```
prop.table(table(c(1,1,2))) #Does this make sense?
```

```
##
##         1         2
## 0.6666667 0.3333333
```

We know, if a = c(1,1,2), then

```
prop.table(table(a))
```

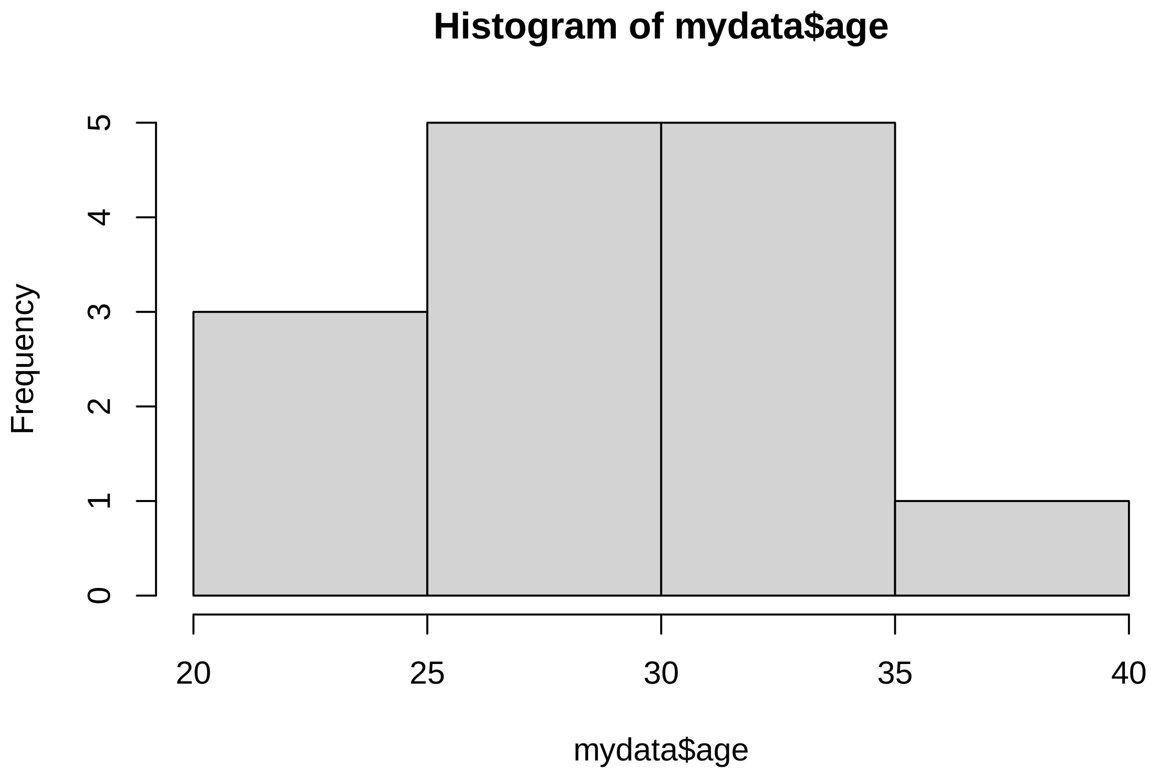
```
## a
##         1         2
## 0.6666667 0.3333333
```

Now, we get back to our data,

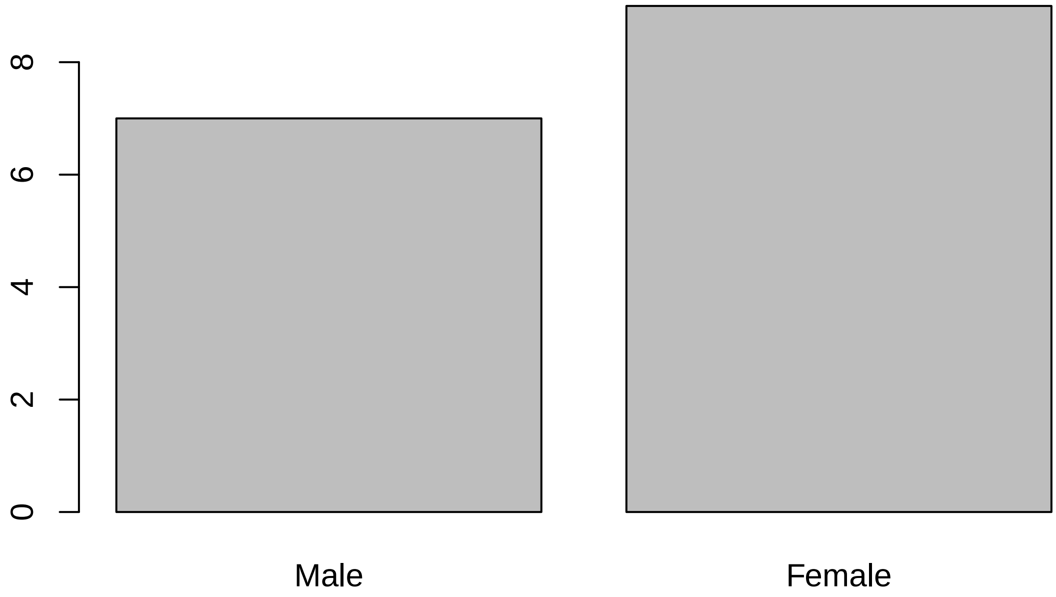
```
prop.table(table(mydata$sex))
```

```
##
##   Male Female
## 0.4375 0.5625
```

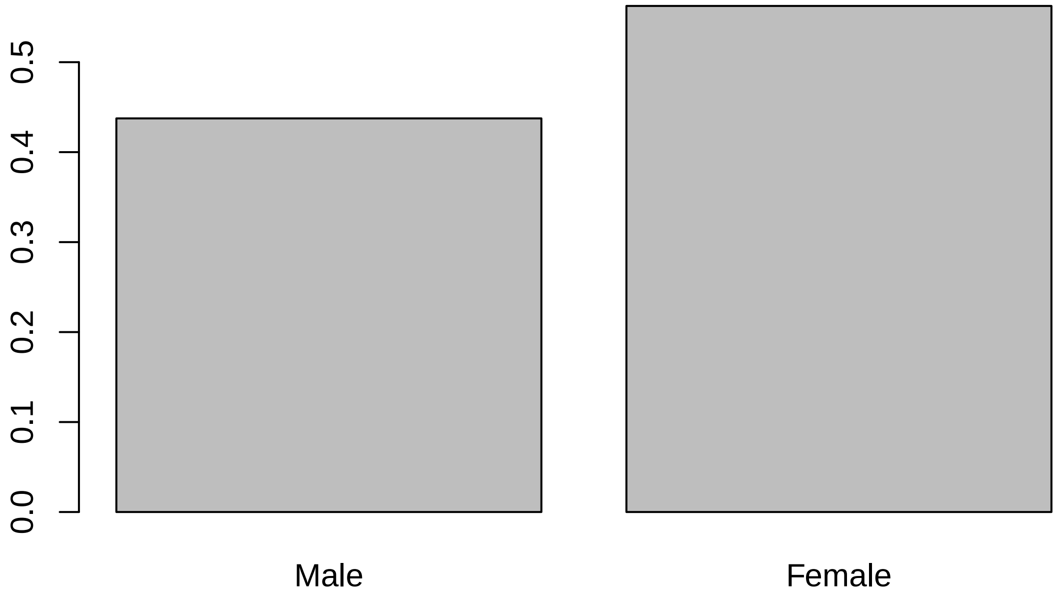
```
#Basic graphs
hist(mydata$age)
```



```
barplot(table(mydata$sex))   #Barplot of frequencies of sex categories
```



```r
barplot(prop.table(table(mydata$sex)))  #Barplot of proportions of sex categories
```



```
#Note the chain.... function of a function of a function of a function.... ;)
#Will get back to graphs later. Now, how would you select only males. Revisit
#the indexing section.

mydata[mydata$sex==1,]
```

```
## [1] id   name age  sex  dept
## <0 rows> (or 0-length row.names)
```

```
mydata[mydata$sex==2,]
```

```
## [1] id   name age  sex  dept
## <0 rows> (or 0-length row.names)
```

```
mydata[mydata$sex==2,c("name","sex","age")]
```

```
## [1] name sex  age
## <0 rows> (or 0-length row.names)
```

```
mydata$age[mydata$sex==1]                      #Print ages for males
```

```
## numeric(0)
```

Practice 1 grepl() is a function used to test if a string is contained in another string. Use the candidates data set to attempt the following

Descriptive Statistics

With a data file of 10 individuals, it could be easier to list or report age of each person. This is usually not feasible. Therefore, it is imperative to find a number that typically reflects all the values. ### Quantitative Variables ### Measures of Central Tendancy #### Min #### Max #### Median #### Mean/Average

Qualitative Variables

It is not uncommon to tabulate frequencies for certain categories of variable. For example, we may want to know how many males are in our data set; or the proportion of .

```
table.sex <- table(mydata$sex)
table.sex
```

```
##
##   Male Female
##      7      9
```

```
prop.table(table.sex)*100
```

```
##
##   Male Female
##  43.75  56.25
```

Cross tabulation

```
options(digits = 3)

tbl.sex.dept <- with(mydata, table(sex,dept))
tbl.sex.dept
```

```
##         dept
## sex       A B
##   Male    5 2
##   Female  5 4
```

```
prop.table(tbl.sex.dept,1)*100  #Row proportions
```

```
##         dept
## sex         A    B
##   Male   71.4 28.6
##   Female 55.6 44.4
```

```
prop.table(tbl.sex.dept,2)*100  #Column proportions
```

```
##         dept
## sex         A    B
##   Male   50.0 33.3
##   Female 50.0 66.7
```

The mean() function calculates and returns an average of values in a variable.

```
mean(mydata$age)
```

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

If an NA is returned instead, it probably could be that one of the entries is a missing a value. It is possible to check if a vector or data frame has missing values. We can use is.na(mydata$age) for the variable age or is.na(mydata). Try! You will note that R outputs a TRUE for any value which is NA otherwise it outputs a

FALSE. One nice property with boolean output is that the FALSE values can also be read as zero and the TRUE as ones. So, if we want a summary output, we can just sum all the TRUE (1s) as

```
sum(is.na(mydata))
```

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

```
sum(is.na(mydata$age))
```

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

So, now we are convinced that that the mean() function could not return a value because the variable age has NAs. The NAs can be ignored by passing another parameter to the mean function.

```
mean(mydata$age, na.rm = T)   # Average
```

```
## [1] 29.9
```

```
sd(mydata[,"age"], na.rm = T) # Standard deviation
```

```
## [1] 4.68
```

Further, we can get the mean for males and females separately, or at one go when using the aggregate function.

```
mean(mydata$age[mydata$sex=="Male"], na.rm = T)
```

```
## [1] 31.5
```

```
mean(mydata$age[mydata$sex=="Female"], na.rm = T)
```

```
## [1] 28.8
```

```
aggregate(age~sex, data = mydata, mean)      #Mean by sex
```

```
##      sex  age
## 1   Male 31.5
## 2 Female 28.8
```

Practice 2

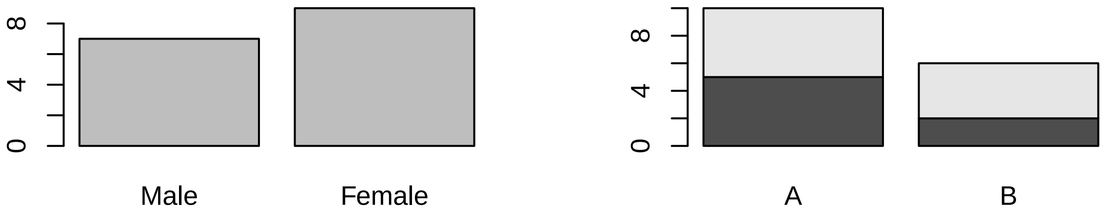
Graphs

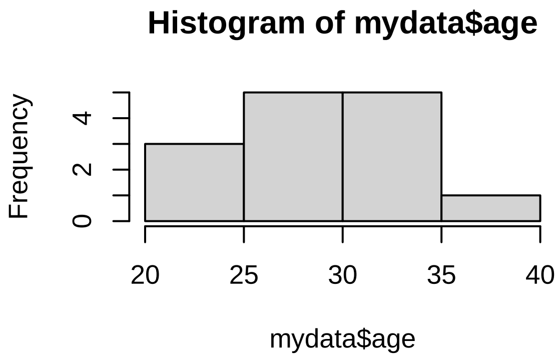
Not all times do numbers do a good job when telling a story. Graphs are best it at too. Graphs provide a pictorial view of the pattern of the situation. Let’s revisit the mydata data frame. We know there are 7 males and their corresponding average age of 31.5 and 9 females and their corresponding average age of 28.75.

We would tell the story differently using graphs. , say, by using barplot() function for a bar graph, ie barplot(c(7,9)). The idea is that we pass a table as an an argument.

```
par(mfrow = c(2, 2))
barplot(table.sex)
barplot(tbl.sex.dept)

hist(mydata$age)
```





Practice 3

#Data management/manipulation

Merging data

Often times in household surveys, interviewers are conducted at household level, and later on at individual level. There now comes an opportunity to learn in greater depth about the individuals, but poses a challenge to connect back the individuals to the household from which they belong. To ease the process of interconnecting these data files, we must have or create a common identifier in both data sets.

Let's assume we have two separate but related data sets. To demonstrate the concept, we will create a small data set so that we easily observe the changes. We will merge this data with our mydata data frame.

```
options(width = 100)
#Creating some data
set.seed(65767)
id <- sample(1:16, 7, replace = F )
ed <- factor(sample(1:3,7,replace = T), levels = c(1,2,3), labels = c("none","formal","informal"))
edu.data <- data.frame(id, ed)
edu.data
```

```
##   id       ed
## 1 10   formal
## 2  6     none
## 3  1 informal
## 4 11 informal
## 5 14     none
## 6  3     none
## 7  9     none
```

We note that person id number 10 had attended "formal" education, and we know from our mydata that the name of the person with this id is Martha. We would like to join these two data sets together, and so, we use the merge() function. The two data sets must have a number of common variables, of course they don't necessarily to have the same name.

```
merge(mydata,edu.data, by = "id")
```

```
##   id    name age    sex dept       ed
## 1  1 Patrick  23   Male    A informal
## 2  3 Bernard  31   Male    A     none
## 3  6    Rico  38   Male    A     none
## 4  9   Cecil  25 Female    A     none
## 5 10  Martha  NA Female    A   formal
## 6 11 Merriam  30 Female    B informal
## 7 14    Mada  30 Female    B     none
```

```
merge(mydata,edu.data, by = "id", all.x = T)
```

```
##    id    name age    sex dept       ed
## 1   1 Patrick  23   Male    A informal
## 2   2 Gregory  29   Male    A     <NA>
## 3   3 Bernard  31   Male    A     none
## 4   4   Lesla  21 Female    A     <NA>
## 5   5 Bridget  34 Female    A     <NA>
## 6   6    Rico  38   Male    A     none
## 7   7   Temwa  28 Female    A     <NA>
## 8   8  Andrew  33   Male    A     <NA>
## 9   9   Cecil  25 Female    A     none
## 10 10  Martha  NA Female    A   formal
## 11 11 Merriam  30 Female    B informal
## 12 12 William  35   Male    B     <NA>
## 13 13  Martha  33 Female    B     <NA>
## 14 14    Mada  30 Female    B     none
## 15 15    Sara  29 Female    B     <NA>
## 16 16    <NA>  NA   Male    B     <NA>
```

```
merge(mydata,edu.data, by = "id", all.y = T)
```

```
##   id    name age    sex dept       ed
## 1  1 Patrick  23   Male    A informal
## 2  3 Bernard  31   Male    A     none
## 3  6    Rico  38   Male    A     none
## 4  9   Cecil  25 Female    A     none
## 5 10  Martha  NA Female    A   formal
## 6 11 Merriam  30 Female    B informal
## 7 14    Mada  30 Female    B     none
```

```
merge(mydata,edu.data, by = "id", all = T)
```

```
##    id    name age    sex dept       ed
## 1   1 Patrick  23   Male    A informal
## 2   2 Gregory  29   Male    A     <NA>
## 3   3 Bernard  31   Male    A     none
## 4   4   Lesla  21 Female    A     <NA>
## 5   5 Bridget  34 Female    A     <NA>
## 6   6    Rico  38   Male    A     none
## 7   7   Temwa  28 Female    A     <NA>
## 8   8  Andrew  33   Male    A     <NA>
## 9   9   Cecil  25 Female    A     none
## 10 10  Martha  NA Female    A   formal
## 11 11 Merriam  30 Female    B informal
## 12 12 William  35   Male    B     <NA>
## 13 13  Martha  33 Female    B     <NA>
## 14 14    Mada  30 Female    B     none
## 15 15    Sara  29 Female    B     <NA>
## 16 16    <NA>  NA   Male    B     <NA>
```

Section: Sorting/Ordering data.

Ordering data is another important process in data processing and management. Its not so straightforward but the concept is much more clearer. Say, if ordering a data set in id; what it means is that you are rearranging the positions of rows or cases.

```
edu.data$id
```

```
## [1] 10  6  1 11 14  3  9
```

```
sort(edu.data$id)
```

```
## [1]  1  3  6  9 10 11 14
```

```
order(edu.data$id)
```

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

I will like to take you back to indexing elements in a data frame. When we specify row numbers as in mydata[c(2,6,3),], R returns all columns for row number 2, then 6 and then 3, in that same order. Similarly, we want the order of. . . . .

Look closely to the outputs. The order() output tells us that the lowest value is on row number 3, second on 6 and so on and so forth.

```
mydata[c(2,6,3),]
```

```
##   id    name age  sex dept
## 2  2 Gregory  29 Male    A
## 6  6    Rico  38 Male    A
## 3  3 Bernard  31 Male    A
```

```
mydata[order(mydata$sex),]              #Sort data by sex
```

```
##    id    name age    sex dept
## 1   1 Patrick  23   Male    A
## 2   2 Gregory  29   Male    A
## 3   3 Bernard  31   Male    A
## 6   6    Rico  38   Male    A
## 8   8  Andrew  33   Male    A
## 12 12 William  35   Male    B
## 16 16    <NA>  NA   Male    B
## 4   4   Lesla  21 Female    A
## 5   5 Bridget  34 Female    A
## 7   7   Temwa  28 Female    A
## 9   9   Cecil  25 Female    A
## 10 10  Martha  NA Female    A
## 11 11 Merriam  30 Female    B
## 13 13  Martha  33 Female    B
## 14 14    Mada  30 Female    B
## 15 15    Sara  29 Female    B
```

```
mydata[order(mydata$sex, mydata$age),]  #Sort data by sex and age
```

```
##    id    name age    sex dept
## 1   1 Patrick  23   Male    A
## 2   2 Gregory  29   Male    A
## 3   3 Bernard  31   Male    A
## 8   8  Andrew  33   Male    A
## 12 12 William  35   Male    B
## 6   6    Rico  38   Male    A
## 16 16    <NA>  NA   Male    B
## 4   4   Lesla  21 Female    A
## 9   9   Cecil  25 Female    A
## 7   7   Temwa  28 Female    A
## 15 15    Sara  29 Female    B
## 11 11 Merriam  30 Female    B
## 14 14    Mada  30 Female    B
## 13 13  Martha  33 Female    B
## 5   5 Bridget  34 Female    A
## 10 10  Martha  NA Female    A
```

#Section: More useful functions #Section: More about graphs #Section: User-defined functions

Tidyverse function

The tidyverse package actually contains other packages (dplyr, ggplot2, etc.) and you'll see that when you load the tidyverse package using library(). Remember the package must be installed to your device before it can be loaded into your libraries! For help on installing packages, refer to Section

```
library(tidyverse)
```

```
## -- Attaching core tidyverse packages -------------------------------------------- tidyverse 2.0.0 --
## v dplyr     1.1.3     v readr     2.1.4
## v forcats   1.0.0     v stringr   1.5.0
## v ggplot2   3.4.3     v tibble    3.2.1
## v lubridate 1.9.2     v tidyr     1.3.0
## v purrr     1.0.2
## -- Conflicts ---------------------------------------------------------- tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
## i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to become error
```

pipes

The pipe operator, (%>%), feeds the results of one operation into the next operation. It is more handy when there is a sequence of operations on a data frame. The advantage of using the pipe operator is that it makes code extremely easy to read.

mutate, group_by, summarize, filter, select, arrange

```r
options(width = 300)
set.seed(356863)
region <- sample(1:3,1000, prob = c(0.2,0.5,0.3), replace = T)
distr <- c(sample(1:7,200,replace = T),sample(1:10,504,replace = T),sample(1:15,296,replace = T))
resid <- c(rep(c(1,2),c(39,161)),rep(c(1,2),c(203,301)),rep(c(1,2),c(75,221)))
bdywgt <- rnorm(1000,33,9)

#Tibble (a data frame version)
sample.data <- tibble(region,distr, resid, bdywgt)
rm(region,distr,resid,bdywgt)

sample.data <- sample.data %>%
  mutate(dist_code = region*100+distr)

#Summarize
sample.data %>%
  group_by(region) %>%
  summarise(meanwgt = mean(bdywgt),
            freqn   = NROW(bdywgt))
```

```
## # A tibble: 3 x 3
##   region meanwgt freqn
##    <int>   <dbl> <int>
## 1      1    32.3   177
## 2      2    32.3   519
## 3      3    33.3   304
```

```r
seed <- read.csv("seed.csv")
str(seed)
```

```
## 'data.frame':    64 obs. of  4 variables:
##  $ Blocks  : int  1 2 3 4 1 2 3 4 1 2 ...
##  $ cultivar: chr  "vicland1" "vicland1" "vicland1" "vicland1" ...
##  $ seedchem: chr  "control" "control" "control" "control" ...
##  $ response: num  42.9 41.6 28.9 30.8 53.3 69.6 45.4 35.1 62.3 58.3 ...
```

```r
seed[sample(1:nrow(seed),10),]
```

```
##    Blocks cultivar seedchem response
## 60      4  clinton    Agrox     51.8
## 42      2  clinton   panoge     46.1
## 23      3 vicland2  ceresan     42.4
## 29      1   Branch  ceresan     70.3
## 16      4   Branch  control     52.7
## 1       1 vicland1  control     42.9
## 25      1  clinton  ceresan     63.4
## 8       4 vicland2  control     35.1
## 32      4   Branch  ceresan     58.5
## 62      2   Branch    Agrox     69.4
```

```r
table(seed$Blocks)
```

```
## 
##  1  2  3  4
## 16 16 16 16
```

```
table(seed$cultivar)
```

```
##
##   Branch  clinton vicland1 vicland2
##       16       16       16       16
```

```
table(seed$seedchem)
```

```
##
##   Agrox ceresan control  panoge
##      16      16      16      16
```

```
mean(seed$response)
```

```
## [1] 52.8
```

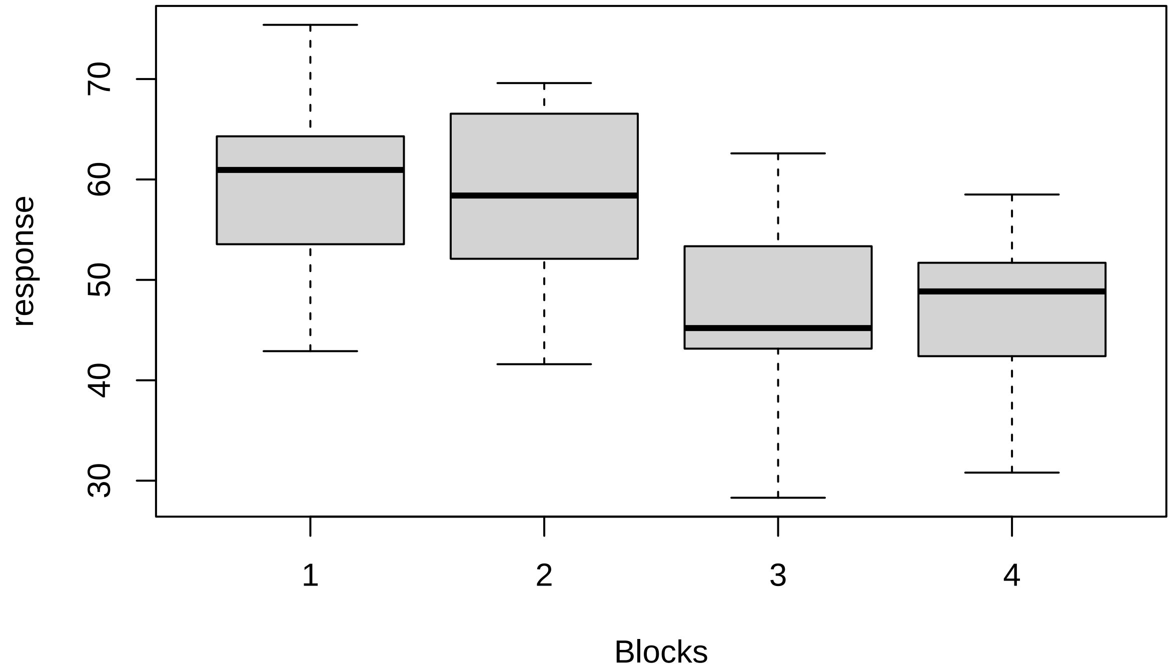
```
aggregate(response~Blocks,data=seed, mean)
```

```
##   Blocks response
## 1      1     59.8
## 2      2     58.5
## 3      3     46.2
## 4      4     46.5
```

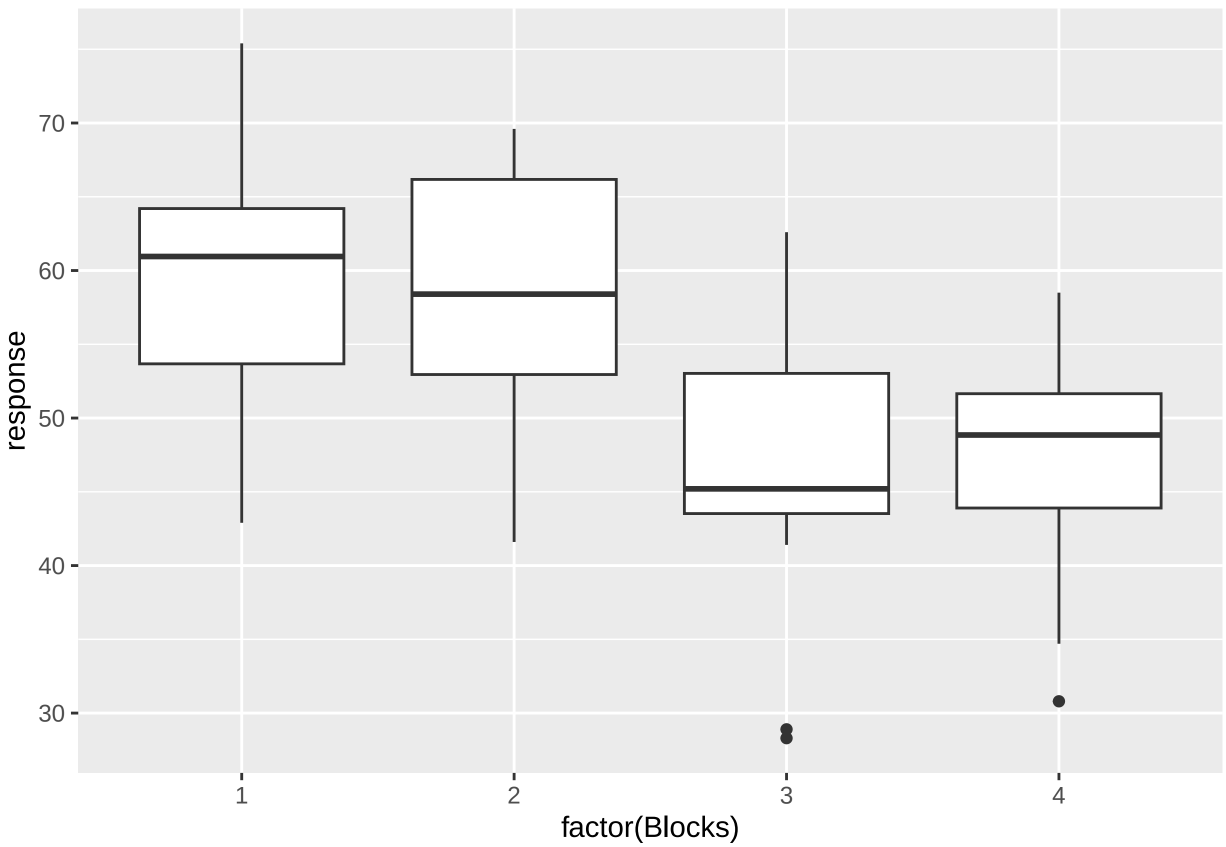
```
my.aov <- aov(response~Blocks,data=seed)
summary(my.aov)
```

```
##             Df Sum Sq Mean Sq F value  Pr(>F)
## Blocks       1   2188    2188    25.6 4.1e-06 ***
## Residuals   62   5309      86
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

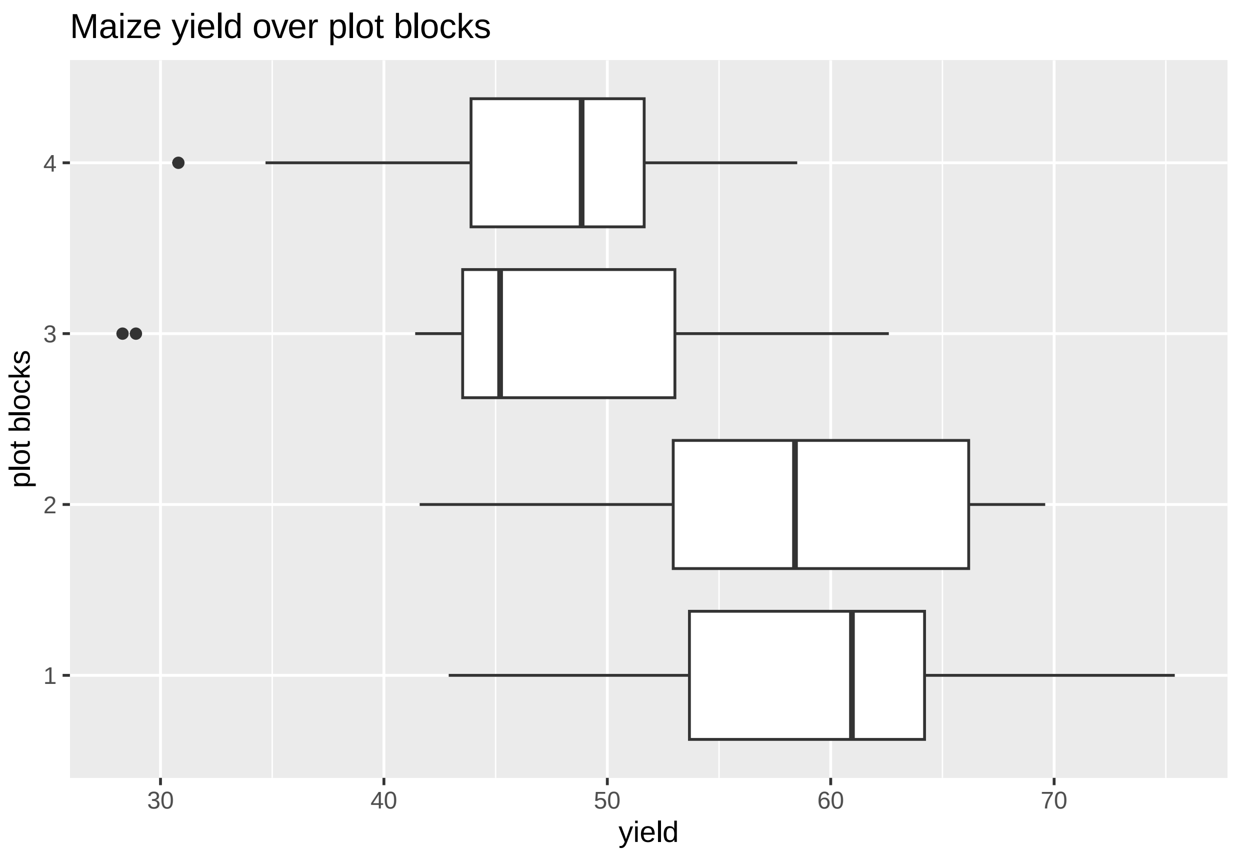
```
#Boxplot
boxplot(response~Blocks, data=seed)
```



```
#ggplot version
my.blocks.resp <-  ggplot(seed, aes(x = factor(Blocks), y = response))
my.blocks.resp + geom_boxplot()
```



```
my.blocks.resp + geom_boxplot() + coord_flip() + labs(x = "plot blocks", y = "yield", title = "Maize yi
```



Visualization with ggplot2

```
seed <- read.csv("seed.csv")
str(seed)
```

```
## 'data.frame':    64 obs. of  4 variables:
##  $ Blocks  : int  1 2 3 4 1 2 3 4 1 2 ...
##  $ cultivar: chr  "vicland1" "vicland1" "vicland1" "vicland1" ...
##  $ seedchem: chr  "control" "control" "control" "control" ...
##  $ response: num  42.9 41.6 28.9 30.8 53.3 69.6 45.4 35.1 62.3 58.3 ...
```

```
seed[sample(1:nrow(seed),10),]
```

```
##    Blocks cultivar seedchem response
## 55      3 vicland2    Agrox     44.1
## 38      2 vicland2   panoge     65.8
## 13      1   Branch  control     75.4
## 30      2   Branch  ceresan     67.3
## 34      2 vicland1   panoge     53.8
## 26      2  clinton  ceresan     50.4
## 19      3 vicland1  ceresan     43.9
## 14      2   Branch  control     65.6
## 64      4   Branch    Agrox     47.4
## 54      2 vicland2    Agrox     57.4
```

```r
table(seed$Blocks)
```

```
## 
##  1  2  3  4 
## 16 16 16 16
```

```r
table(seed$cultivar)
```

```
## 
##   Branch  clinton vicland1 vicland2 
##       16       16       16       16
```

```r
table(seed$seedchem)
```

```
## 
##   Agrox ceresan control  panoge 
##      16      16      16      16
```

```r
mean(seed$response)
```

```
## [1] 52.8
```

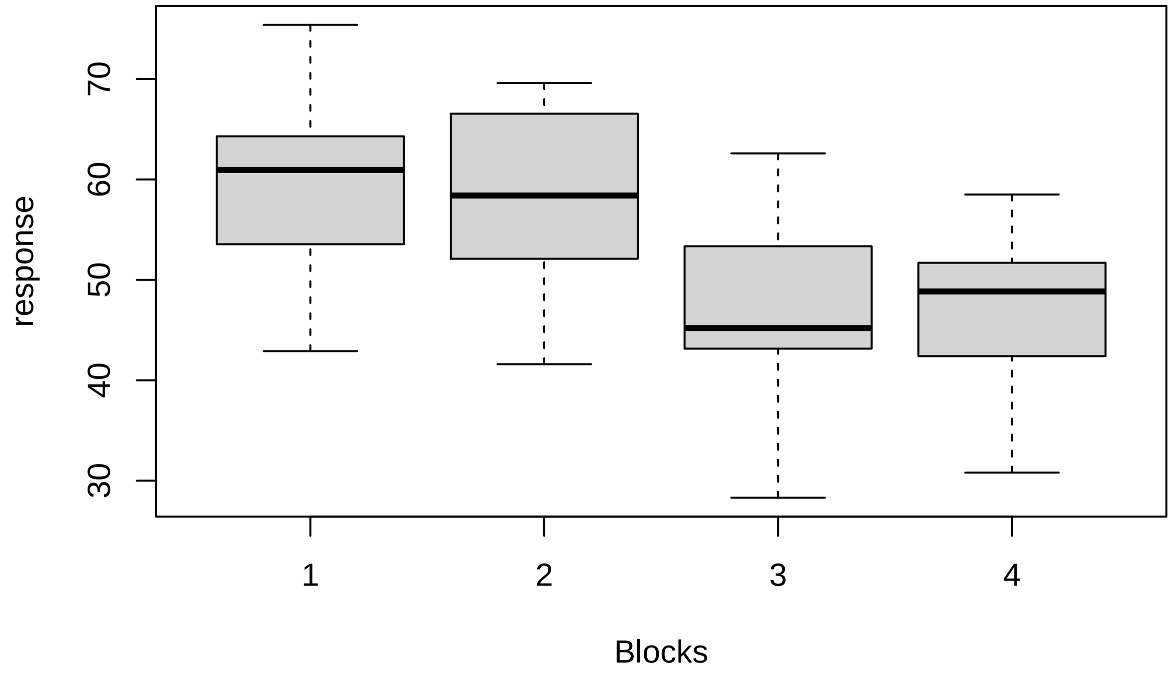
```r
aggregate(response~Blocks,data=seed, mean)
```

```
##   Blocks response
## 1      1     59.8
## 2      2     58.5
## 3      3     46.2
## 4      4     46.5
```

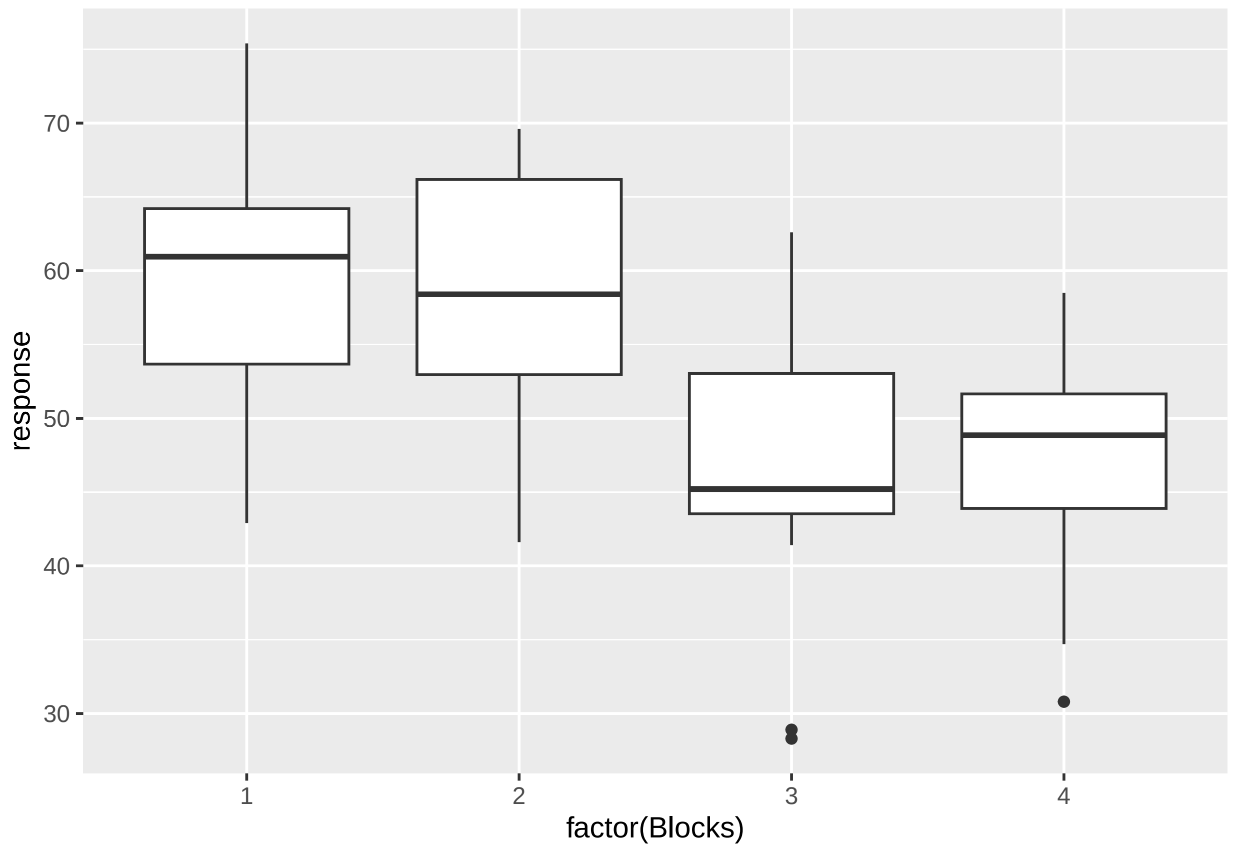
```r
my.aov <- aov(response~Blocks,data=seed)
summary(my.aov)
```

```
##             Df Sum Sq Mean Sq F value  Pr(>F)    
## Blocks       1   2188    2188    25.6 4.1e-06 ***
## Residuals   62   5309      86                    
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

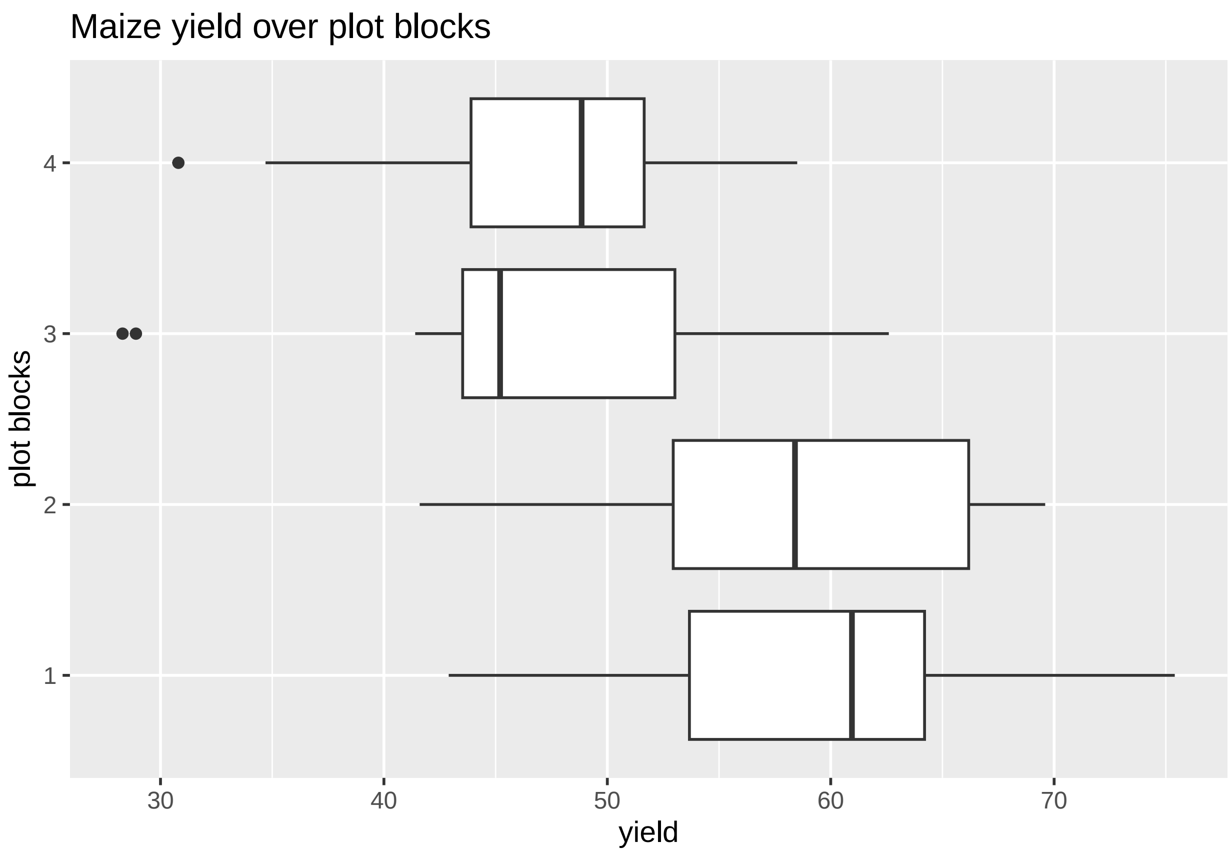
```r
#Graphs
boxplot(response~Blocks, data=seed)
```



```
my.blocks.resp <-  ggplot(seed, aes(x = factor(Blocks), y = response))
my.blocks.resp + geom_boxplot()
```



```
my.blocks.resp + geom_boxplot() + coord_flip() + labs(x = "plot blocks", y = "yield", title = "Maize yi
```



With ggplot2, data and aesthetic mappings are supplied in ggplot(), then layers are added on with +. This is an important pattern, and as you learn more about ggplot2 you'll construct increasingly sophisticated plots by adding on more types of components.

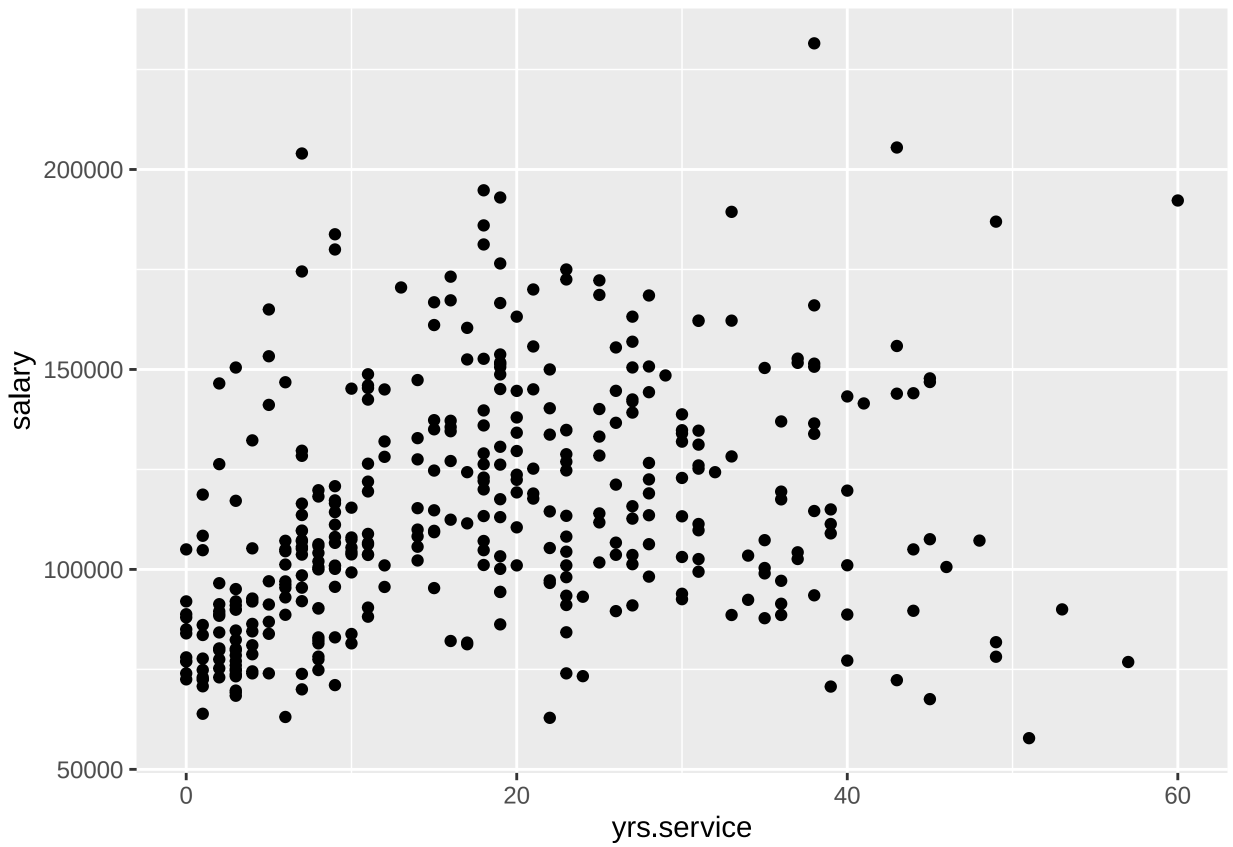
Almost every plot maps a variable to x and y, so naming these aesthetics is tedious, so the first two unnamed arguments to aes() will be mapped to x and y. This means that the following code is identical to the example above:

#Applied Inferential Statistics

Linear regression

```
salaries <- read.csv("WorkSalaries.csv")
View(salaries)
```

```
mylm.plot <- ggplot(salaries, aes(yrs.service, salary))
mylm.plot + geom_point()
```



```r
my.lm <- lm(salary~yrs.service, data = salaries)
summary(my.lm)
```

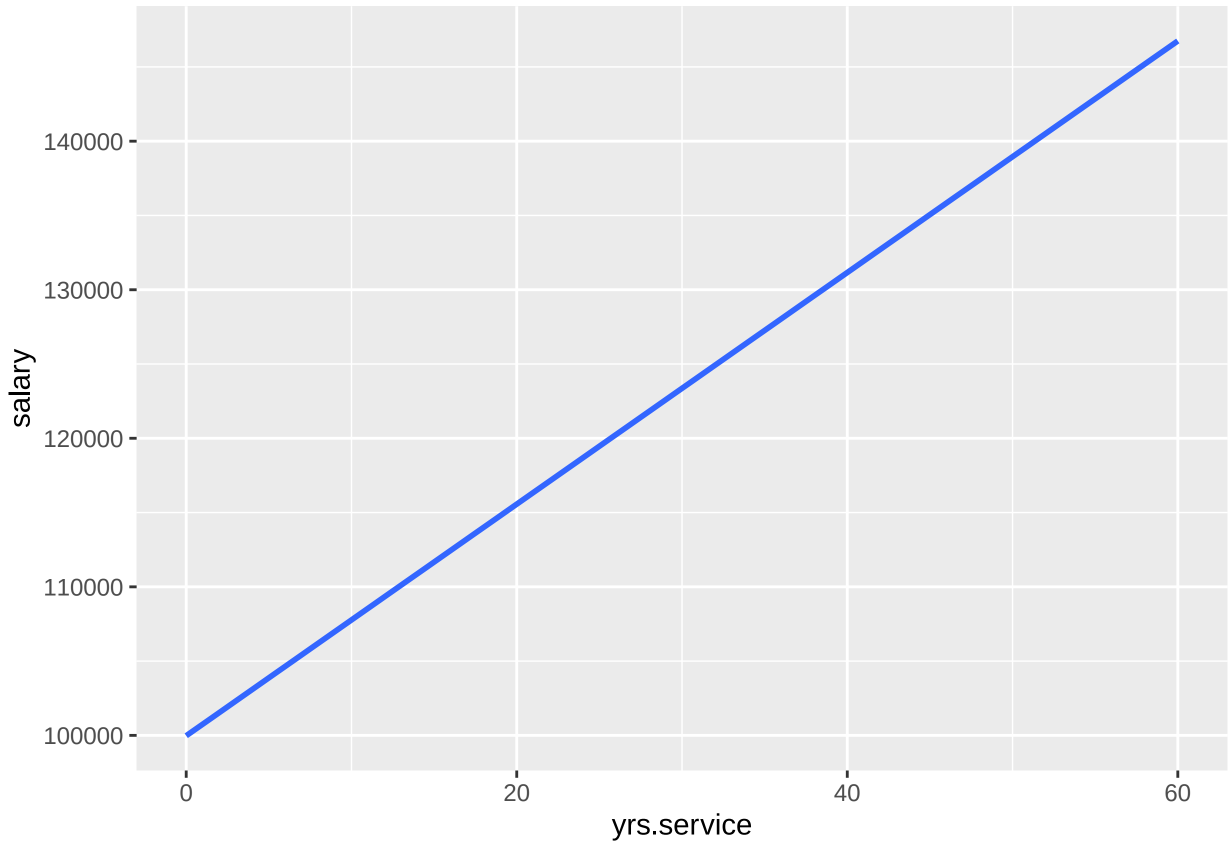
```
##
## Call:
## lm(formula = salary ~ yrs.service, data = salaries)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -81933 -20511  -3776  16417 101947
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)    99975       2417   41.37  < 2e-16 ***
## yrs.service      780        110    7.06  7.5e-12 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 28600 on 395 degrees of freedom
## Multiple R-squared:  0.112,  Adjusted R-squared:  0.11
## F-statistic: 49.8 on 1 and 395 DF,  p-value: 7.53e-12
```

```r
ls(my.lm)
```

```
##  [1] "assign"        "call"          "coefficients"  "df.residual"   "effects"       "fitted.values"
```

```
mylm.plot +
  geom_smooth(method = lm, se = F, fullrange = T)
```

```
## `geom_smooth()` using formula = 'y ~ x'
```



Programming with User functions

User functions

Sampling

During the course of the journey we have used functions which would be more beneficial to sampling of elements from a population. ### Simple random sampling

Simple random sampling ensures that every has an equal probability of being selected from the population. There are numerous way we can achieve this,

```
sample(1:100,50)
```

```
## [1] 75 60 26 91 25 76 66 63 35 93 22 64  2 85 86 20 33 21 42 61 15 96 28 34
```

Systematic random sampling

Simulations

Survival Tables

The cohort component projection method projects the population into the future by age (usually 5-year age groups) and sex. Survival rates are used to calculate the number of people that will be alive at a future date in time.

In many countries, life tables are based on an average of age-specific death rates for a 3-year time period, generally around a census taking. In many cases, the life tables are prepared every 10 years. For example, a country or state would collect age-specific death rates for 1999, 2000, and 2001. The census for year 2000 would be used for the base population.

Projections

This chapter describes a variant of the cohort component method which can be used to make a projection either of the national population or of urban and rural populations. The method is capable of projecting the structure of the population by age and sex along with various indicators of population size, structure and change.

Mathematical

The mathematical method is quick, simple, and requires little in the way of data. It is the approach of choice for many projections of the whole populations of countries.

Cohort

The component method is much more cumbersome than the mathematical method, and has heavy data requirements. It is more time-consuming than the mathematical method, although the advent of computers has made it a great deal quicker than it used to be. It has the great advantage over the mathematical method that detailed aspects of the population structure can be forecast

The major strength of this technique is its ability to project a population in a straightforward and unambiguous manner. The technique does not embody restrictive or arbitrary assumptions and generates results which faithfully reflect the initial population structure and the fertility, mortality and migration conditions specified by the user. It yields projection results which are indispensable to any planning exercise seeking to take the future population change into account. These features make this technique fundamental for integrating population factors into development planning.

```
age_int <- c(0,1,seq(5,95,5))
nqx <- c(0.02592,0.0042,0.00232,0.00201,0.00443,0.00611,0.00632,0.00654,0.01098,0.01765,0.02765,0.04387
lx <- c(100000)

for (i in 2:length(nqx))
{
  lx[i] <- round(lx[i-1] - lx[i-1]*nqx[i-1])
}

ndx <- round(nqx * lx)
```

Another example

```
x   <- c(0,1,seq(5,75,5))
n   <- c(1,4,rep(5,(length(x)-2)))
nMx <- c(0.1072,0.0034,0.0010,0.0007,0.0017,0.0030,0.0036,0.0054,0.0054,0.0146,0.0128,0.0269,0.0170,0.04
nkx <- c(0.33,1.56,rep(2.5,length(nMx)-2))

nqx <- round((n*nMx)/(1 + (n - nkx)*nMx),4)

lx <- c(100000)
for (i in 2:length(nqx))
{
  lx[i] <- round(lx[i-1] - lx[i-1]*nqx[i-1])
  #Lx[i] <- (lx[i-1] + lx[i])*2.5
}

ndx <- round(nqx*lx)
nLx <- n * lx - ndx*(n - nkx)

Tx <- NA

for (i in 1:length(nqx))
{
  Tx[i] <- sum(nLx[i:length(nqx)])
  #Lx[i] <- (lx[i-1] + lx[i])*2.5
}

ex <- Tx/lx
as.data.frame(cbind(x,n,nMx,nkx, nqx,lx,ndx,nLx,Tx,ex))
```

```
##     x n    nMx  nkx    nqx     lx   ndx    nLx      Tx    ex
## 1   0 1 0.1072 0.33 0.1000 100000 10000  93300 5536915 55.37
## 2   1 4 0.0034 1.56 0.0135  90000  1215 357035 5443615 60.48
## 3   5 5 0.0010 2.50 0.0050  88785   444 442815 5086580 57.29
## 4  10 5 0.0007 2.50 0.0035  88341   309 440933 4643765 52.57
## 5  15 5 0.0017 2.50 0.0085  88032   748 438290 4202833 47.74
## 6  20 5 0.0030 2.50 0.0149  87284  1301 433168 3764543 43.13
## 7  25 5 0.0036 2.50 0.0178  85983  1530 426090 3331375 38.74
## 8  30 5 0.0054 2.50 0.0266  84453  2246 416650 2905285 34.40
## 9  35 5 0.0054 2.50 0.0266  82207  2187 405568 2488635 30.27
## 10 40 5 0.0146 2.50 0.0704  80020  5633 386018 2083068 26.03
## 11 45 5 0.0128 2.50 0.0620  74387  4612 360405 1697050 22.81
## 12 50 5 0.0269 2.50 0.1260  69775  8792 326895 1336645 19.16
## 13 55 5 0.0170 2.50 0.0815  60983  4970 292490 1009750 16.56
## 14 60 5 0.0433 2.50 0.1954  56013 10945 252703  717260 12.81
## 15 65 5 0.0371 2.50 0.1698  45068  7653 206208  464558 10.31
## 16 70 5 0.0785 2.50 0.3281  37415 12276 156385  258350  6.90
## 17 75 5 0.0931 2.50 0.3776  25139  9492 101965  101965  4.06
```

#SIMILATIONS