

# Linear Models for Data Science

Jeffrey Woo

2024-06-28



# Contents

<b>Preface</b>	<b>5</b>
Who is this book for? . . . . .	5
Data sets used . . . . .	6
Chapters . . . . .	6
Other resources . . . . .	6
<b>1 Data Wrangling with R</b>	<b>9</b>
1.1 Introduction . . . . .	9
1.2 Data Wrangling using Base R Functions . . . . .	10
1.3 Data Wrangling Using dplyr Functions . . . . .	18
<b>2 Introduction</b>	<b>27</b>
<b>3 Literature</b>	<b>29</b>
3.1 New section . . . . .	29
<b>4 Methods</b>	<b>31</b>
4.1 math example . . . . .	31
<b>5 Applications</b>	<b>33</b>
5.1 Example one . . . . .	33
5.2 Example two . . . . .	33
<b>6 Final Words</b>	<b>35</b>



# Preface

## Who is this book for?

There are many books on linear models, with various expectations for different levels of familiarity with statistical, mathematical, and coding concepts. These books generally fall into one of two camps:

1. Little to no familiarity with statistical and mathematical concepts, but fairly familiar to coding. These books tend to be written for programmers who want to get into data science. These books tend to explain linear models while trying to avoid statistical and mathematical concepts as much, only covering these concepts if absolutely necessary. These books tend to present linear models in a recipe format giving readers directions on what to do to build their models.

The drawback of such books is that readers do not get much understanding of the underlying concepts of linear models. It is impossible to give directions covering every possible scenario in the real world as real data are messy. Practitioners of data science often have to think outside the box in order to make linear models work for their particular data, and it is difficult to do so without understanding the mathematical framework of linear models.

2. Familiarity with mathematical notation and introductory statistical concepts such as statistical inference, and little to no familiarity with coding. These books tend to be written for mathematicians (or anyone with a strong background in mathematics) who want to get into data science. These books cover the mathematical framework of linear models thoroughly.

The drawback of such books is that readers must be comfortable with mathematical notation. This limits the audience for such books to people with fairly thorough training in mathematics. People without such training will get lost trying to read such books, and do not understand why we need to know the mathematical foundations to use linear models in data science.

This book is meant to be readable by both groups of readers. Some foundational mathematical knowledge will be presented, but will be written so that is

readable by anyone. This book will also explain what these knowledge mean in the context of data science. Practical advice, based on the foundational mathematical knowledge, will also be given.

This book accompanies the course STAT 6021: Linear Models for Data Science, for the Masters of Data Science (MSDS) program at the University of Virginia School of Data Science.

As introductory statistics and introductory programming are pre-requisites for entering the MSDS program, this book assumes basic knowledge of statistical inference and coding. Review materials covering these concepts are provided separately for enrolled students.

## Data sets used

I have tried to use as many open source data sets as much as possible so that readers can work on the various examples I have provided on their own. However, some data sets may not be open source and have come from my experience teaching this class since 2019 (and variations of the class since 2013), and have used some data sets that were shared by other statistics and data science educators. It is my goal to eventually use only open source data sets.

## Chapters

The chapters for the book is as follows:

## Other resources

Some other resources that readers may want to check out:

- *OpenIntro Statistics*, 4th ed. Diez, Cetinkaya-Rundel, Barr, OpenIntro. Get free PDF version at <https://leanpub.com/os>, just set the price that you want to pay to \$0. This is a good book for introductory statistics.
- *Linear Models with R*, 2nd ed. Faraway. This is probably one of the few books that balances between the two camps that I wrote about earlier. It does require familiarity with matrices and linear algebra though.
- *Introduction to Linear Regression Analysis*, 5th or 6th ed. Montgomery, Peck, Vining. You may be able to access an e-version of the book through your university library if you are affiliated with a university. This book is mathematically rigorous so is useful to those who are interested in mathematical proofs that is not covered.
- *Applied Linear Statistical Models* (ALSM), Kutner, Nachtsheim, Neter, Li, 5th ed. This book covers a wide range of topics in linear models and is also mathematically rigorous.

- *Applied Linear Regression Models* (ALRM), Kutner, Nachtsheim, Neter, 4th ed. ALRM is the same as the first 14 chapters of ALSM. The second part of ALSM covers topics in Design of Experiments, which I highly recommend if you are interested in those topics.





# Chapter 1

## Data Wrangling with R

### 1.1 Introduction

The data structure that we will be dealing with most often will be data frames. When we read data in to R, they are typically stored as a data frame. A data frame can be viewed like an EXCEL spreadsheet, where data are stored in rows and columns. Before performing any analysis, we want the data frame to have this basic structure:

- Each row of the data frame corresponds to an observation.
- Each column of the data frame corresponds to a variable.

Sometimes, data are not structured in this way, and we will have to transform the data to take on this basic data structure. This process is called data wrangling. The most common and basic operations to transform the data are:

- Selecting a subset of columns of the data frame.
- Selecting a subset of rows of the data frame based on some criteria.
- Change column names.
- Find missing data.
- Create new variables based on existing variables.
- Combine multiple data frames.

We will explore two approaches to data wrangling:

- Using functions that already come pre-loaded with R (sometimes called base R).
- Using functions from the `dplyr` package.

These two approaches are quite different but can achieve the same goals in data wrangling. Each user of R usually ends up with their own preferred way of performing data wrangling operations, but it is important to know both approaches so you are able to work with a broader audience.

## 1.2 Data Wrangling using Base R Functions

We will use the dataset `ClassDataPrevious.csv` as an example. The data were collected from an introductory statistics class at UVa from a previous semester. Download the dataset from Canvas and read it into R.

```
Data<-read.csv("ClassDataPrevious.csv", header=TRUE)
```

We check the number of rows and columns in this dataframe.

```
dim(Data)
```

```
## [1] 298 8
```

There are 298 rows and 8 columns: so we have 298 students and 8 variables. We can also check the names for each column.

```
colnames(Data)
```

```
## [1] "Year"      "Sleep"     "Sport"     "Courses"   "Major"     "Age"       "Computer"
## [8] "Lunch"
```

The variables are:

1. **Year**: the year the student is in
2. **Sleep**: how much sleep the student averages a night (in hours)
3. **Sport**: the student's favorite sport
4. **Courses**: how many courses the student is taking in the semester
5. **Major**: the student's major
6. **Age**: the student's age (in years)
7. **Computer**: the operating system the student uses (Mac or PC)
8. **Lunch**: how much the student usually spends on lunch (in dollars)

### 1.2.1 View specific row(s) and/or column(s) of a data frame

We can view specific rows and/or columns of any data frame using the square brackets `[]`, for example:

```
Data[1,2] ##row index first, then column index
```

```
## [1] 8
```

The row index is listed first, then the column index, in the square brackets. This means the first student sleeps 8 hours a night. We can also view multiple rows and columns, for example:

```
Data[c(1,3,4),c(1,5,8)]
```

```
##      Year                Major Lunch
## 1 Second                Commerce    11
## 3 Second Cognitive science and psychology    10
```

```
## 4 First Pre-Comm 4
```

to view the 1st, 5th, and 8th variables for observations 1, 3, and 4.

There are several ways to view a specific column. For example, to view the 1st column (which is the variable called `Year`):

```
Data$Year ##or
Data[,1] ##or
Data[,c(2:8)]
```

Note the comma separates the indices for the row and column. An empty value before the comma means we want all the rows, and then the specific column. To view multiple columns, for example the first four columns:

```
Data[,1:4]
Data[,c(1,2,3,4)]
```

To view the values of certain rows, we can use

```
Data[c(1,3),]
```

to view the values for observations 1 and 3. An empty value after the comma means we want all the columns for those specific rows.

## 1.2.2 Select observations by condition(s)

We may want to only analyze certain subsets of our data, based on some conditions. For example, we may only want to analyze students whose favorite sport is soccer. The `which()` function in R helps us find the indices associated with a condition being met. For example:

```
which(Data$Sport=="Soccer")
```

```
## [1] 3 20 25 26 31 32 33 38 44 46 48 50 51 64 67 71 87 92 98
## [20] 99 118 122 124 126 128 133 136 137 143 146 153 159 165 174 197 198 207 211
## [39] 214 226 234 241 255 259 260 266 274 278 281 283 294 295
```

informs us which rows belong to observations whose favorite sport is soccer, i.e. the 3rd, 20th, 25th (and so on) students. We can create a new data frame that contains only students whose favorite sport is soccer:

```
SoccerPeeps<-Data[which(Data$Sport=="Soccer"),]
dim(SoccerPeeps)
```

```
## [1] 52 8
```

We are extracting the rows which satisfy the condition, favorite sport being soccer, and storing these rows into a new data frame called `SoccerPeeps`. We can see that this new data frame has 52 observations.

Suppose we want to have a data frame that satisfies two conditions: that the favorite sport is soccer and they are 2nd years at UVa. We can type:

```
SoccerPeeps_2nd<-Data[which(Data$Sport=="Soccer" & Data$Year=="Second"),]
dim(SoccerPeeps_2nd)
```

```
## [1] 25 8
```

This new data frame `SoccerPeeps_2nd` has 25 observations.

We can also set conditions based on numeric variables, for example, we want students who sleep more than eight hours a night

```
Sleepy<-Data[which(Data$Sleep>8),]
```

We can also create a data frame that contains students who satisfy at least one out of two conditions, for example, the favorite sport is soccer or they sleep more than 8 hours a night:

```
Sleepy_or_Soccer<-Data[which(Data$Sport=="Soccer" | Data$Sleep>8),]
```

### 1.2.3 Change column name(s)

For some datasets, the names of the columns are complicated or do not make sense. We should always give descriptive names to columns that make sense. For this dataset, the names are self-explanatory so we do not really need to change them. As an example, suppose we want to change the name of the 7th column from `Computer` to `Comp`:

```
names(Data)[7]<- "Comp"
```

To change the names of multiples columns (for example, the 1st and 7th columns), type:

```
names(Data)[c(1,7)]<-c("Yr", "Computer")
```

### 1.2.4 Find and remove missing data

There are a few ways to locate missing data. Using the `is.na()` function directly on a data frame produces a lot of output that can be messy to view:

```
is.na(Data)
```

On the other hand, using the `complete.cases()` function is more pleasing to view:

```
Data[!complete.cases(Data),]
```

##	Yr	Sleep	Sport	Courses	Major
## 103	Second	NA	Basketball	7	psychology and youth and social innovation
## 206	Second	8	None	4	Cognitive Science

```
##      Age Computer Lunch
## 103  19      Mac    10
## 206  19      Mac    NA
```

The code above will extract rows that are not complete cases, in other words, rows that have missing entries. The output informs us observation 103 has a missing value for `Sleep`, and observation 206 has a missing value for `Lunch`.

If you want to remove observations with a missing value, you can use one of the following two lines of code to create new data frames with the rows with missing values removed:

```
Data_nomiss<-na.omit(Data) ##or
Data_nomiss2<-Data[complete.cases(Data),]
```

**A word of caution:** these lines of code will remove the entire row as long as at least a column has missing entries. As noted earlier, observation 103 has a missing value for only the `Sleep` variable. But this observation still provides information on the other variables, which are now removed.

### 1.2.5 Summarizing variable(s)

Very often, we want to obtain some characteristics of our data. A common way to summarize a numerical variable is to find its mean. We have four numerical variables in our data frame, which are in columns 2, 4, 6, and 8. To find the mean of all four numerical variables, we can use the `apply()` function:

```
apply(Data[,c(2,4,6,8)],2,mean)
```

```
##      Sleep    Courses      Age      Lunch
##      NA  5.016779 19.573826      NA
```

```
apply(Data[,c(2,4,6,8)],2,mean,na.rm=T)
```

```
##      Sleep    Courses      Age      Lunch
## 155.559259  5.016779 19.573826 156.594175
```

Notice that due to the missing values, the first line has `NA` for some of the variables. The second line includes an optional argument, `na.rm=T`, which will remove the observations with an `NA` value for the variable from the calculation of the mean.

There are at least 3 arguments that are supplied to the `apply()` function:

1. The first argument is a data frame containing all the variables which we want to find the mean of. In this case, we want columns 2, 4, 6, and 8 of the data frame `Data`.
2. The second argument takes on the value 1 or 2. Since we want to find the mean of columns, rather than rows, we type 2. If want to mean of a row, we will type 1.

3. The third argument specifies the name of the function you want to apply to the columns of the supplied data frame. In this case, we want the mean. We can change this to find the median, standard deviation, etc, of these numeric variables if we want to.

We notice the means for some of the variables are suspiciously high, so looking at the medians will be more informative.

```
apply(Data[,c(2,4,6,8)],2,median,na.rm=T)
```

```
##   Sleep Courses      Age   Lunch
##    7.5      5.0    19.0     9.0
```

### 1.2.6 Summarizing variable by groups

Sometimes we want to summarize a variable by groups. Suppose we want to find the median amount of sleep separately for 1st years, 2nd years, 3rd years, and 4th years get. We can use the `tapply()` function:

```
tapply(Data$Sleep,Data$Yr,median,na.rm=T)
```

```
##   First Fourth Second   Third
##    8.0     7.0    7.5    7.0
```

This informs us the median amount of sleep first years get is 8 hours a night; for fourth years the median amount is 7 hours a night.

There are at least 3 arguments that are supplied to the `tapply()` function:

1. The first argument contains the vector which we want to summarize.
2. The second argument contains the factor which we use to subset our data. In this example, we want to subset according to `Yr`.
3. The third argument is the function which we want to apply to each subset of our data.
4. The fourth argument is optional, in this case, we want to remove observations with missing values from the calculation of the mean.

Notice the output orders the factor levels in alphabetical order. For our context, it is better to rearrange the levels to First, Second, Third, Fourth using the `factor()` function:

```
Data$Yr<-factor(Data$Yr, levels=c("First","Second","Third","Fourth"))
```

```
levels(Data$Yr)
```

```
## [1] "First" "Second" "Third" "Fourth"
```

```
tapply(Data$Sleep,Data$Yr,median,na.rm=T) ##much nicer
```

```
## First Second Third Fourth
##      8.0    7.5    7.0    7.0
```

This output makes a lot more sense for this context.

If we want to summarize a variable on groups formed by more than one variable, we need to adjust the second argument in the `tapply()` function by creating a list. Suppose we want to find the median sleep hour based on the `Yr` and the preferred operating system of the observations,

```
tapply(Data$Sleep,list(Data$Yr,Data$Computer),median,na.rm=T)
```

```
##           Mac    PC
## First  NA 8.0 7.50
## Second  7 7.5 7.50
## Third  NA 7.5 7.00
## Fourth NA 7.0 7.25
```

Interestingly, it looks like there were observations who did not specify which operating system they use, hence the extra column in the output.

### 1.2.7 Create a new variable based on existing variable(s)

Depending on the context of our analysis, we may need to create new variables based on existing variables. There are a few variations of this task, based on the type of variable you want to create, and the type of variable it is based on.

#### 1.2.7.1 Create a numeric variable based on another numeric variable

The variable `Sleep` is in number of hours. Suppose we need to convert the values of `Sleep` to number of minutes, we can simply perform the following mathematical operation:

```
Sleep_mins<-Data$Sleep * 60
```

and store the transformed variable into a vector called `Sleep_mins`.

#### 1.2.7.2 Create a binary variable based on a numeric variable

Suppose we want to create a binary variable (categorical variable with two levels), called `deprived`. An observation will obtain a value of “yes” if they sleep for less than 7 hours a night, and “no” otherwise. The `ifelse()` function is useful in creating binary variables:

```
deprived<-ifelse(Data$Sleep<7, "yes", "no")
```

There are 3 arguments associated with the `ifelse()` function:

1. The first argument is the condition that we wish to use.

2. The second argument is the value of the observation if the condition is true.
3. The third argument is the value of the observation if the condition is false.

### 1.2.7.3 Create a categorical variable based on a numeric variable

Suppose we want to create a categorical variable based on the number of courses a student takes. We will call this new variable `CourseLoad`, which takes on the following values:

- `light` if 3 courses or less,
- `regular` if 4 or 5 courses,
- `heavy` if more than 5 courses .

The `cut()` function is used in this situation

```
CourseLoad<-cut(Data$Courses, breaks = c(-Inf, 3, 5, Inf),
               labels = c("light", "regular", "heavy"))
```

There are three arguments that are applied to the `cut()` function:

1. The first argument is the vector which you are basing the new variable on.
2. The argument `breaks` lists how you want to set the intervals associated with `Data$Courses`. In this case, we are creating three intervals: one from  $(-\infty, 3]$ , another from  $(3, 5]$ , and the last interval from  $(5, \infty]$ .
3. The argument `labels` gives the label for `CourseLoad` associated with each interval.

### 1.2.7.4 Collapse levels

Sometimes, a categorical variable has more levels than we need for our analysis, and we want to collapse some levels. For example, the variable `Yr` has four levels: First, Second, Third, and Fourth. Perhaps we are more interested in comparing upperclassmen and underclassmen, so we want to collapse First and Second years into underclassmen, and Third and Fourth years into upperclassmen:

```
levels(Data$Yr)

## [1] "First" "Second" "Third" "Fourth"

new.levels<-c("und", "und", "up", "up")
Year2<-factor(new.levels[Data$Yr])
levels(Year2)

## [1] "und" "up"
```

The levels associated with the variable `Yr` are ordered as First, Second, Third, Fourth. The character vector `new.levels` has `und` as the first two characters,



and `upper` as the last two characters to correspond to the original levels in the variable `Yr`. The new variable is called `Year2`.

### 1.2.8 Combine data frames

We have created four new variables, `Sleep_mins`, `deprived`, `CourseLoad`, and `Year2`, based on previously existing variables. Since these variables are all based on the same observations, we can combine them with an existing data frame using the `data.frame()` function:

```
Data<-data.frame(Data,Sleep_mins,deprived,CourseLoad,Year2)
head(Data)
```

```
##      Yr Sleep      Sport Courses      Major Age Computer
## 1 Second      8 Basketball      6      Commerce 19      Mac
## 2 Second      7      Tennis      5      Psychology 19      Mac
## 3 Second      8      Soccer      5 Cognitive science and psychology 21      Mac
## 4 First       9 Basketball      5      Pre-Comm 19      Mac
## 5 Second      4 Basketball      6      Statistics 19      PC
## 6 Third       7      None      4      Psychology 20      PC
##  Lunch Sleep_mins deprived CourseLoad Year2
## 1     11         480       no      heavy  und
## 2     10         420       no    regular  und
## 3     10         480       no    regular  und
## 4      4         540       no    regular  und
## 5      0         240      yes     heavy  und
## 6     11         420       no    regular  up
```

Notice that since we listed the four new variables after `Data` in the `data.frame()` function, they appear after the original columns in the data frame.

Alternatively, we can use the `cbind()` function which gives the same data frame:

```
Data2<-cbind(Data,Sleep_mins,deprived,CourseLoad,Year2)
```

If you are combining data frames which have different observations but the same columns, we can merge them using `rbind()`:

```
dat1<-Data[1:3,1:3]
dat3<-Data[6:8,1:3]
res.dat2<-rbind(dat1,dat3)
head(res.dat2)
```

```
##      Yr Sleep      Sport
## 1 Second      8 Basketball
## 2 Second      7      Tennis
## 3 Second      8      Soccer
## 6 Third       7      None
## 7 Second      7 Basketball
```

```
## 8 First      7 Basketball
```

### 1.2.9 Export data frame in R to a .csv file

To export a data frame to a .csv file, type:

```
write.csv(Data, file="newdata.csv", row.names = FALSE)
```

A file newdata.csv will be created in the working directory. Note that by default, the argument `row.names` is set to be `TRUE`. This will add a column in the dataframe which is an index number. I do not find this step useful in most analyses so I almost always set `row.names` to be `FALSE`.

### 1.2.10 Sort data frame by column values

To sort your data frame in ascending order by Age:

```
Data_by_age<-Data[order(Data$Age),]
```

To sort in descending order by Age:

```
Data_by_age_des<-Data[order(-Data$Age),]
```

To sort in ascending order by Age first, then by Sleep:

```
Data_by_age_sleep<-Data[order(Data$Age, Data$Sleep),]
```

## 1.3 Data Wrangling Using dplyr Functions

In the previous section, we were performing data wrangling operations using functions that are built in with base R. In this module, we will be using functions mostly from a package called `dplyr`, which can perform the same operations as well.

Before performing data wrangling operations, let us clear our environment, so that previously declared objects are removed. This allows us to start with a clean slate, which is often desirable when starting on a new analysis. This is done via:

```
rm(list = ls())
```

The `dplyr` package is a subset of the `tidyverse` package, so we can access these functions after installing and loading either package. After installing the `tidyverse` package, load it by typing:

```
##library(dplyr) or  
library(tidyverse)
```

The `dplyr` package was developed to make the syntax more intuitive to a broader range of R users, primarily through the use of pipes. However, the code involved

with functions from `dplyr` tends to be longer than code involved with base R functions, and there are more functions to learn with `dplyr`.

You will find a lot of articles on the internet by various R users about why each of them believes one approach to be superior to the other. I am not fussy about which approach you use as long as you can perform the necessary operations. It is to your benefit to be familiar with both approaches so you can work with a broader range of R users.

We will continue to use the dataset `ClassDataPrevious.csv` as an example. Download the dataset from Canvas and read it into R:

```
Data<-read.csv("ClassDataPrevious.csv", header=TRUE)
```

In the examples below, we are performing the same operations as in the previous section, but using `dplyr` functions instead of base R functions.

### 1.3.1 Select specific column(s) of a data frame

The `select()` function is used to select specific columns. There are a couple of ways to use this function. First:

```
select(Data,Year)
```

to select the column `Year` from the data frame called `Data`.

#### 1.3.1.1 Pipes

Alternatively, we can use pipes:

```
Data%>%  
  select(Year)
```

Pipes in R are typed using `%>%` or by pressing Ctrl + Shift + M on your keyboard. To think of the operations above, we can read the code as

1. take the data frame called `Data`
2. and then select the column named `Year`.

We can interpret a pipe as “and then”. Commands after a pipe should be placed on a new line (press enter). Pipes are especially useful if we want to execute several commands in sequence, which we will see in later examples.

### 1.3.2 Select observations by condition(s)

The `filter()` function allows us to subset our data based on some conditions, for example, to select students whose favorite sport is soccer:

```
filter(Data, Sport=="Soccer")
```

We can create a new data frame called `SoccerPeeps` that contains students whose favorite sport is soccer:

```
SoccerPeeps<-Data%>%
  filter(Sport=="Soccer")
```

Suppose we want to have a data frame, called `SoccerPeeps_2nd`, that satisfies two conditions: that the favorite sport is soccer and they are 2nd years at UVa:

```
SoccerPeeps_2nd<-Data%>%
  filter(Sport=="Soccer" & Year=="Second")
```

We can also set conditions based on numeric variables, for example, we want the students who sleep more than eight hours a night:

```
Sleepy<-Data%>%
  filter(Sleep>8)
```

We can also create a data frame that contains observations as long as they satisfy at least one out of two conditions: the favorite sport is soccer or they sleep more than 8 hours a night:

```
Sleepy_or_Soccer<-Data%>%
  filter(Sport=="Soccer" | Sleep>8)
```

### 1.3.3 Change column name(s)

It is straightforward to change the names of columns using the `rename()` function. For example:

```
Data<-Data%>%
  rename(Yr=Year, Comp=Computer)
```

allows us to change the name of two columns: from `Year` and `Computer` to `Yr` and `Comp`.

### 1.3.4 Summarizing variable(s)

The `summarize()` function allows us to summarize a column. Suppose we want to find the mean value of the numeric columns: `Sleep`, `Courses`, `Age`, and `Lunch`:

```
Data%>%
  summarize(mean(Sleep, na.rm = T), mean(Courses), mean(Age), mean(Lunch, na.rm = T))
```

```
##   mean(Sleep, na.rm = T) mean(Courses) mean(Age) mean(Lunch, na.rm = T)
## 1           155.5593       5.016779   19.57383           156.5942
```

The output looks a bit cumbersome. We can give names to each summary

```
Data%>%
  summarize(avgSleep=mean(Sleep,na.rm = T),avgCourse=mean(Courses),avgAge=mean(Age),
            avgLun=mean(Lunch,na.rm = T))

##   avgSleep avgCourse  avgAge  avgLun
## 1 155.5593  5.016779 19.57383 156.5942
```

As mentioned previously, the means look suspiciously high for a couple of variables, so looking at the medians may be more informative:

```
Data%>%
  summarize(medSleep=median(Sleep,na.rm = T),medCourse=median(Courses),
            medAge=median(Age),medLun=median(Lunch,na.rm = T))

##   medSleep medCourse medAge medLun
## 1      7.5         5     19      9
```

**Note:** For a lot of functions in the `dplyr` package, using American spelling or British spelling works. So we can use `summarise()` instead of `summarize()`.

### 1.3.5 Summarizing variable by groups

Suppose we want to find the median amount of sleep 1st years, 2nd years, 3rd years, and 4th years get. We can use the `group_by()` function:

```
Data%>%
  group_by(Yr)%>%
  summarize(medSleep=median(Sleep,na.rm=T))

## # A tibble: 4 x 2
##   Yr      medSleep
##   <chr>      <dbl>
## 1 First         8
## 2 Fourth        7
## 3 Second       7.5
## 4 Third         7
```

The way to read the code above is

1. Get the data frame called `Data`,
2. and then group the observations by `Yr`,
3. and then find the median amount of sleep by each `Yr` and store the median in a vector called `medSleep`.

As seen previously, the ordering of the factor levels is in alphabetical order. For our context, it is better to rearrange the levels to First, Second, Third, Fourth. We can use the `mutate()` function whenever we want to transform or create a new variable. In this case, we are transforming the variable `Yr` by reordering the factor levels with the `fct_relevel()` function:

```
Data<- Data%>%
  mutate(Yr = Yr%>%
    fct_relevel(c("First", "Second", "Third", "Fourth")))
```

1. Get the data frame called `Data`,
2. and then transform the variable called `Yr`,
3. and then reorder the factor levels.

Then, we use pipes, the `group_by()`, and `summarize()` functions like before:

```
Data%>%
  group_by(Yr)%>%
  summarize(medSleep=median(Sleep,na.rm=T))
```

```
## # A tibble: 4 x 2
##   Yr      medSleep
##   <fct>      <dbl>
## 1 First         8
## 2 Second       7.5
## 3 Third         7
## 4 Fourth         7
```

This output makes a lot more sense for this context.

To summarize a variable on groups formed by more than one variable, we just add the other variables in the `group_by()` function:

```
Data%>%
  group_by(Yr,Comp)%>%
  summarize(medSleep=median(Sleep,na.rm=T))
```

```
## `summarise()` has grouped output by 'Yr'. You can override using the `.groups`
## argument.
```

```
## # A tibble: 9 x 3
## # Groups:   Yr [4]
##   Yr      Comp medSleep
##   <fct> <chr>      <dbl>
## 1 First "Mac"         8
## 2 First "PC"         7.5
## 3 Second ""          7
## 4 Second "Mac"       7.5
## 5 Second "PC"       7.5
## 6 Third "Mac"       7.5
## 7 Third "PC"         7
## 8 Fourth "Mac"         7
## 9 Fourth "PC"       7.25
```

### 1.3.6 Create a new variable based on existing variable(s)

As mentioned in the previously, the `mutate()` function is used to transform a variable or to create a new variable. There are a few variations of this task, based on the type of variable you want to create, and the type of variable it is based on.

#### 1.3.6.1 Create a numeric variable based on another numeric variable

The variable `Sleep` is in number of hours. Suppose we need to convert the values of `Sleep` to number of minutes, we can simply perform the following mathematical operation:

```
Data<-Data%>%  
  mutate(Sleep_mins = Sleep*60)
```

and store the transformed variable called `Sleep_mins` and add `Sleep_mins` to the data frame called `Data`.

#### 1.3.6.2 Create a binary variable based on a numeric variable

Suppose we want to create a binary variable, called `deprived`. An observation will obtain a value of `yes` if they sleep for less than 7 hours a night, and `no` otherwise. We will then add this variable `deprived` to the data frame called `Data`:

```
Data<-Data%>%  
  mutate(deprived=ifelse(Sleep<7, "yes", "no"))
```

#### 1.3.6.3 Create a categorical variable based on a numeric variable

Suppose we want to create a categorical variable based on the number of courses a student takes. We will call this new variable `CourseLoad`, which takes on the following values:

- `light` if 3 courses or less,
- `regular` if 4 or 5 courses,
- `heavy` if more than 5 courses

and then add `CourseLoad` to the data frame `Data`. We can use the `case_when()` function from the `dplyr` package, instead of the `cut()` function:

```
Data<-Data%>%  
  mutate(CourseLoad=case_when(Courses <= 3 ~ "light",  
                              Courses >3 & Courses <=5 ~ "regular",  
                              Courses > 5 ~ "heavy"))
```

Notice how the names of the categorical variable are supplied after a specific condition is specified.

### 1.3.6.4 Collapsing levels

Sometimes, a categorical variable has more levels than we need for our analysis, and we want to collapse some levels. For example, the variable `Yr` has four levels: First, Second, Third, and Fourth. Perhaps we are more interested in comparing between upperclassmen and underclassmen, so we want to collapse First and Second Yrs into underclassmen, and Third and Fourth Yrs into upperclassmen. We will use the `fct_collapse()` function:

```
Data<-Data%>%
  mutate(UpUnder=fct_collapse(Yr,under=c("First","Second"),up=c("Third","Fourth")))
```

We are creating a new variable called `UpUnder`, which is done by collapsing `First` and `Second` into a new factor called `under`, and collapsing `Third` and `Fourth` into a new factor called `up`. `UpUnder` is also added to the data frame `Data`.

### 1.3.7 Combine data frames

To combine data frames which have different observations but the same columns, we can combine them using `bind_rows()`:

```
dat1<-Data[1:3,1:3]
dat3<-Data[6:8,1:3]
res.dat2<-bind_rows(dat1,dat3)
head(res.dat2)
```

```
##      Yr Sleep      Sport
## 1 Second     8 Basketball
## 2 Second     7      Tennis
## 3 Second     8      Soccer
## 4 Third      7         None
## 5 Second     7 Basketball
## 6 First      7 Basketball
```

`bind_rows()` works the same way as `rbind()`. Likewise, we can use `bind_cols()` instead of `cbind()`.

### 1.3.8 Sort data frame by column values

To sort your data frame in ascending order by `Age`:

```
Data_by_age<-Data%>%
  arrange(Age)
```

To sort in descending order by `Age`:

```
Data_by_age_des<-Data%>%
  arrange(desc(Age))
```



To sort in ascending order by `Age` first, then by `Sleep`:

```
Data_by_age_sleep<-Data%>%  
  arrange(Age,Sleep)
```



## Chapter 2

# Introduction

You can label chapter and section titles using `{#label}` after them, e.g., we can reference Chapter 2. If you do not manually label them, there will be automatic labels anyway, e.g., Chapter 4.

Figures and tables with captions will be placed in `figure` and `table` environments, respectively.

```
par(mar = c(4, 4, .1, .1))  
plot(pressure, type = 'b', pch = 19)
```

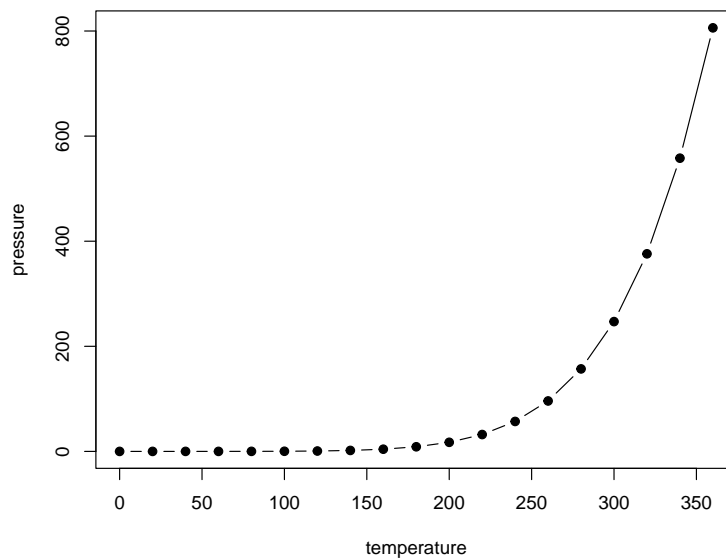


Figure 2.1: Here is a nice figure!

Table 2.1: Here is a nice table!

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
5.1	3.5	1.4	0.2	setosa
4.9	3.0	1.4	0.2	setosa
4.7	3.2	1.3	0.2	setosa
4.6	3.1	1.5	0.2	setosa
5.0	3.6	1.4	0.2	setosa
5.4	3.9	1.7	0.4	setosa
4.6	3.4	1.4	0.3	setosa
5.0	3.4	1.5	0.2	setosa
4.4	2.9	1.4	0.2	setosa
4.9	3.1	1.5	0.1	setosa
5.4	3.7	1.5	0.2	setosa
4.8	3.4	1.6	0.2	setosa
4.8	3.0	1.4	0.1	setosa
4.3	3.0	1.1	0.1	setosa
5.8	4.0	1.2	0.2	setosa
5.7	4.4	1.5	0.4	setosa
5.4	3.9	1.3	0.4	setosa
5.1	3.5	1.4	0.3	setosa
5.7	3.8	1.7	0.3	setosa
5.1	3.8	1.5	0.3	setosa

Reference a figure by its code chunk label with the `fig:` prefix, e.g., see Figure 2.1. Similarly, you can reference tables generated from `knitr::kable()`, e.g., see Table 2.1.

```
knitr::kable(
  head(iris, 20), caption = 'Here is a nice table!',
  booktabs = TRUE
)
```

You can write citations, too. For example, we are using the **bookdown** package (Xie, 2023) in this sample book, which was built on top of R Markdown and **knitr** (Xie, 2015).

## Chapter 3

# Literature

Here is a review of existing methods.

### 3.1 New section

Add some text here. BLAH BLAH



## Chapter 4

# Methods

We describe our methods in this chapter.

Math can be added in body using usual syntax like this

### 4.1 math example

$p$  is unknown but expected to be around  $1/3$ . Standard error will be approximated

$$SE = \sqrt{\left(\frac{p(1-p)}{n}\right)} \approx \sqrt{\frac{1/3(1-1/3)}{300}} = 0.027$$

You can also use math in footnotes like this<sup>1</sup>.

We will approximate standard error to  $0.027^2$

---

<sup>1</sup>where we mention  $p = \frac{a}{b}$

<sup>2</sup> $p$  is unknown but expected to be around  $1/3$ . Standard error will be approximated

$$SE = \sqrt{\left(\frac{p(1-p)}{n}\right)} \approx \sqrt{\frac{1/3(1-1/3)}{300}} = 0.027$$





## Chapter 5

# Applications

Some *significant* applications are demonstrated in this chapter.

### 5.1 Example one

### 5.2 Example two



## Chapter 6

# Final Words

We have finished a nice book.



# Bibliography

Xie, Y. (2015). *Dynamic Documents with R and knitr*. Chapman and Hall/CRC, Boca Raton, Florida, 2nd edition. ISBN 978-1498716963.

Xie, Y. (2023). *bookdown: Authoring Books and Technical Documents with R Markdown*. R package version 0.34.