Chapter 1

Basics of R Programming

EXAM PA LEARNING OBJECTIVES

1. Topic: Predictive Analytics Problems and Tools

Learning Objectives

The Candidate will be able to articulate the types of problems that can be addressed by predictive modeling and be able to work with RStudio to implement basic R packages and commands.

Learning Outcomes

b) Write and execute basic commands in R using RStudio.

(Note: We will defer a) to Part II of this study manual.)

4. Topic: Data Types and Exploration

Learning Outcomes

c) Understand basic methods of handling missing data.

Chapter overview: In Exam PA, you are expected to use R to manipulate data, create graphs for visualizing relationships between variables, and implement predictive models. Based on the computer output, you will develop promising predictive models for making predictions and interpret your findings. A basic understanding of R programming is therefore a prerequisite to doing well in the exam. In this chapter, we will familiarize ourselves with some basic R commands and functions that will be heavily used in the remainder of this study manual. We begin in Section 1.1, where we set up our R toolkit and learn about the three most commonly used types of data. Section 1.2 expands the discussion in Section 1.1 and presents four important data structures that R use to store information. The subtle differences between these data structures are highlighted. Section 1.3 applies what we know about data structures to solve a number of typical data management problems. Section 1.4 concludes this chapter with a discussion of `for` loops, which are important tools for doing simulation. After completing this chapter, you will become comfortable with the R environment and be ready to use R to construct predictive models in subsequent chapters.

It merits mention that Exam PA is not an exam on R programming *per se*; R is used mostly as a means to perform data exploration and run predictive models. As stated in the preface, sample code chunks are provided in the exam to help you accomplish more involved programming tasks

so that your focus is not on writing long code, but on analysis and interpretation[i] (I highly doubt if you will be asked to write a `for` loop from scratch). For this reason, this preparatory chapter, written specifically for students taking Exam PA, is necessarily limited in scope and focuses on a very small set (say, 1%) of tools in R that allow you to accomplish almost all (say, 99%) of the programming tasks you will likely encounter in the exam. Still, a fundamental understanding of R programming is useful for making sense of what the given code chunks do and figuring out how to modify the code (e.g., change the parameters, if needed) to suit our purpose.

1.1 Getting Started

Developed in the early 1990s, R is an extremely versatile open-source programming language for statistical computing, graphics, and, in recent years, document preparation. For Exam PA, the most conspicuous merits of R are three-fold:

- Unlike many commercial statistical software platforms, R is free to use. This point is important as it ensures that every PA candidate can access and practice using the software platform required for the exam for free. (Well, you still have to pay $1125 just to access the PA modules!!)
- R has superior graphics capabilities. It allows you to create elegant, informative, and customizable graphs which would be difficult, if not impossible, to create in other programming languages.
- R outperforms many other software platforms due to the wide collection of add-on packages that can significantly enhance the functionality of R. More impressively, these packages can be downloaded from online repositories for free.

In this introductory section, we will walk through the installation of R and RStudio (a good companion to R), the basic features of RStudio, and the common data types in R.

1.1.1 Basic Infrastructure

First things first, let's install R and RStudio on your computer (if you have not already done so).

Installing R. R can be freely downloaded and installed from *The Comprehensive R Archive Network* (CRAN) at `https://cran.rstudio.com`. The top of the web page shows three links for downloading R:

- Download R for Linux
- Download R for (Mac) OS X
- Download R for Windows

Choose the link that works for the operating system of your computer. To install R on Windows, for example, click "Download R for Windows", then the "base" link, and finally "Download R 3.6.2 for Windows" (3.6.2 may be replaced by a more updated version of R later). An installer program will be downloaded. Running the program and stepping through the installation wizard (the default

[i]See Slide 3 of PA Module 1.

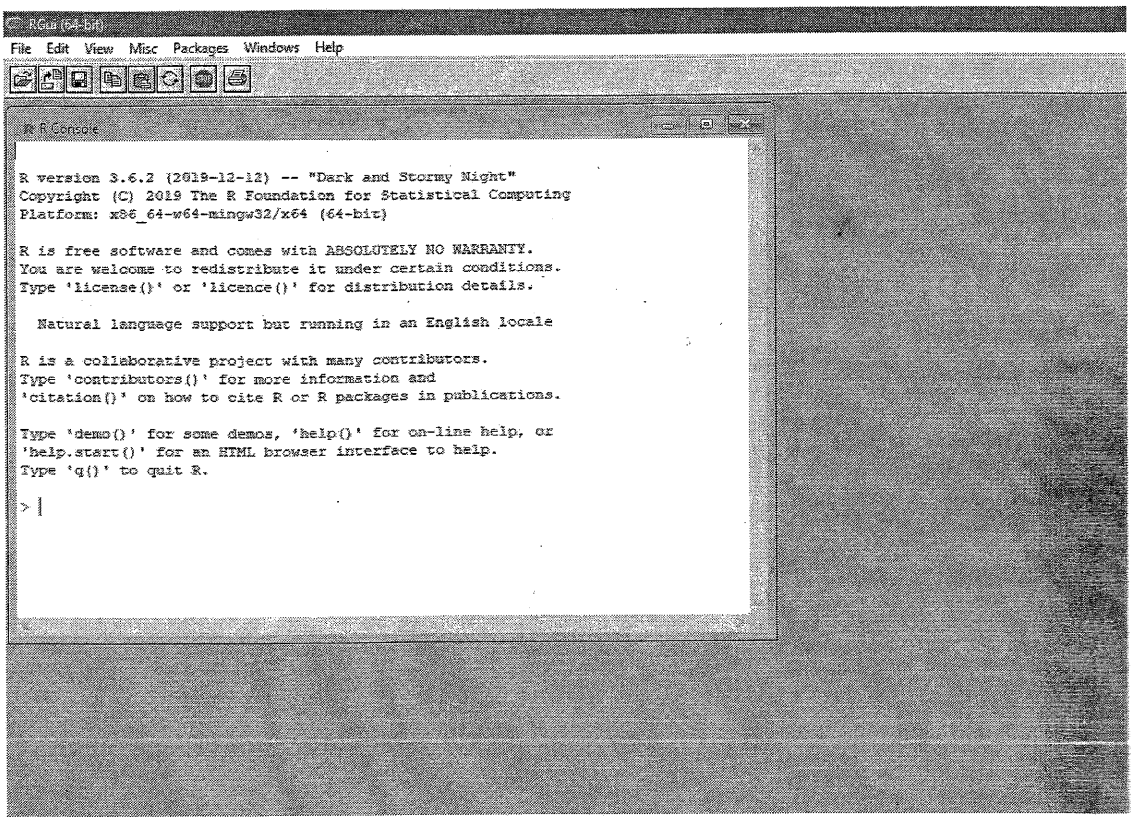


Figure 1.1.1: The basic command line interface of R.

choices will do) will install R into your program files and place a shortcut in your Start menu. When you open R (by clicking its icon on your desktop, for example), you should see a window like Figure 1.1.1. If so, R has been successfully installed.

Installing RStudio. Instead of using R's primitive command line interface in Figure 1.1.1, in Exam PA we will work with R within *RStudio*, which is the most popular *integrated development environment* (IDE) for R and is available for free. An IDE is a software application that provides comprehensive facilities to programmers. Typically, it has a graphical user interface (GUI) and consists of at least a code editor, a compiler or interpreter, and a debugger. The RStudio IDE in particular offers the following valuable features:

- Code highlighting and prediction as well as syntax error detection, helping you write correct code much more efficiently than in R's command line interface.
- The ability to generate dynamic reports where R code and output are integrated into a text document.
- Management of installed packages, environments, and command history all within one workspace.

The relationship between R and RStudio can be succinctly put this way:

> R is the language to speak in and RStudio provides a convenient platform for you to talk to your computer in this language.

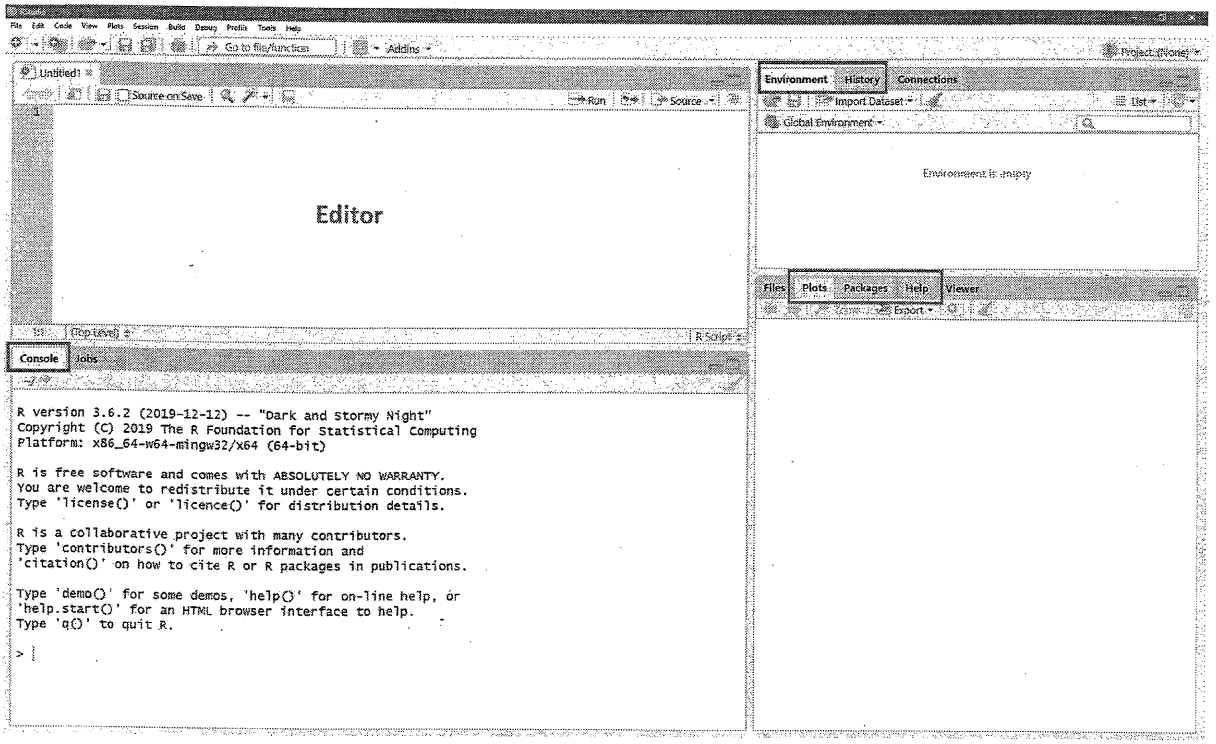


Figure 1.1.2: The basic interface of RStudio.

RStudio can be downloaded and installed from

`https://www.rstudio.com/products/rstudio/download/`.

Look for "RStudio Desktop," select the version corresponding to the operating system of your computer, and follow the instructions there to get RStudio set up.

Layout of RStudio. When you first launch RStudio (by clicking its icon on your desktop, for example), you will see a screen that looks like Figure 1.1.2 with four distinct parts:

- *Lower left pane—console:* This is the direct command line interface to R. All output from running commands and scripts will be shown here. To directly run a command here, place the cursor on the command prompt indicated by the right arrow >, type the command, and press `Enter`. Unless your command is extremely short and simple, you would be better off typing and saving your commands in the editor pane.

- *Upper left pane—editor:* The editor provides different kinds of files where you can compose and save commands for further manipulation. If your editor is closed for whatever reason, you can make it appear by going to the menu bar and selecting `File > New File >`, followed by the type of file where you want to do the editing. The simplest choice is an R script, a plain text file where you can write, edit, and run all or part of your R code. This creates a reproducible record of your work for future use.

- *Lower right pane:* This is where you can find information about plots, packages, and help files.

- ▷ The `Plots` tab displays all of the plots you make if you run commands in the R console or in an R script in the editor pane. The plots can be saved and exported to external files. This tab is not particularly important for Exam PA as the plots generated by running commands in R Markdown files will be shown directly underneath your commands.
- ▷ When you click the `Packages` tab, you can see a list of installed R packages (packages will be further discussed on page 9). To load an installed package into your current R session, simply check the box for that package and it will be available for use. To install a new, external package, click `Install` and type the name of the package. Note that you only have to install a new package once.
- ▷ The `Help` tab is where you can find help documentations on R functions and installed packages. To get help on a function, type the command (either in the R console or in the editor pane)

  `help("<function_name>")` or simply `?<function_name>`,

  where `<function_name>` is the name of the function of interest. To access the documentation of an installed package, use the command

  `help(package = "<package_name>")`,

  where `<package_name>` is the name of the package of interest. At the PA test center, help files on functions in the base installation as well as pre-installed packages are available, but you are strongly advised to be familiar with the functions discussed in these materials prior to the exam.

- *Upper right pane:* This pane contains information about the workspace and command history. In the `Environment` tab, you can see a list of objects that you have imported or created in the current environment. If you have created a data object, then clicking on the name of the object will open a new tab in the editor showing how the object looks. This can ensure that the data object is set up properly. The `History` tab records all of the commands you have run and allows you to reuse or modify old commands.

  (Note: You will not see the `Connections` tab in the exam as there will be no Internet connections.)

You will get fully comfortable with these four panes as you work through this study manual.

R Markdown files. In Exam PA, you are specifically required to type and save your code in an *R Markdown* (Rmd) file, which will be graded together with your report in Word. An Rmd file is simply a plain text document in which text is interspersed with chunks of R code. When the file is compiled, a new document with the code chunks replaced by their output is generated. Most people turn to R Markdown as the preferred means of communicating results due to its capability to generate dynamic reports in different formats (e.g., PDF, HTML, slideshows), where R code and output are embedded with texts for expository convenience. The results produced are internally tied to the code, so changing the data input automatically updates the output. In contrast, if you typeset your report in an external word-processing document like Word and your data or code changes, you will have to take the trouble to rerun your code in R and copy and paste the output from R to your Word file manually (not to mention that Word files are no match for reports generated by R Markdown in terms of typesetting quality!).[ii]

[ii] This study manual is typeset in LaTeX, not in Word!

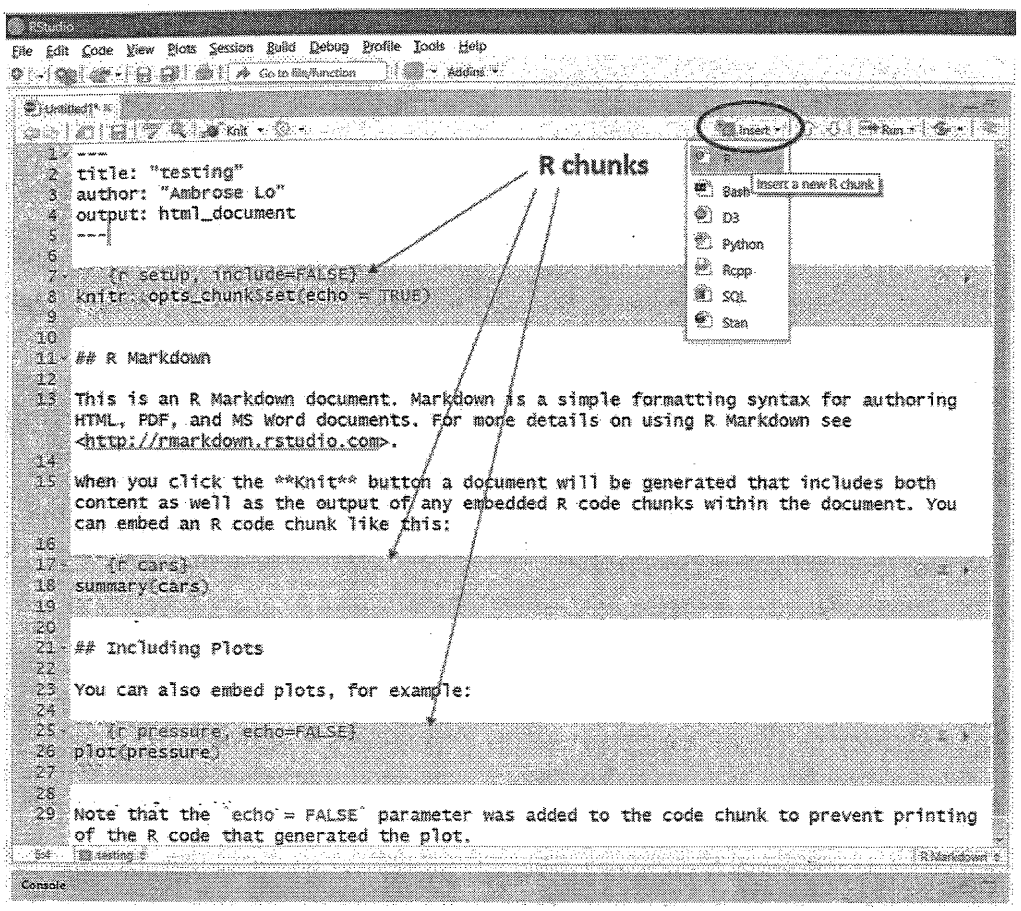
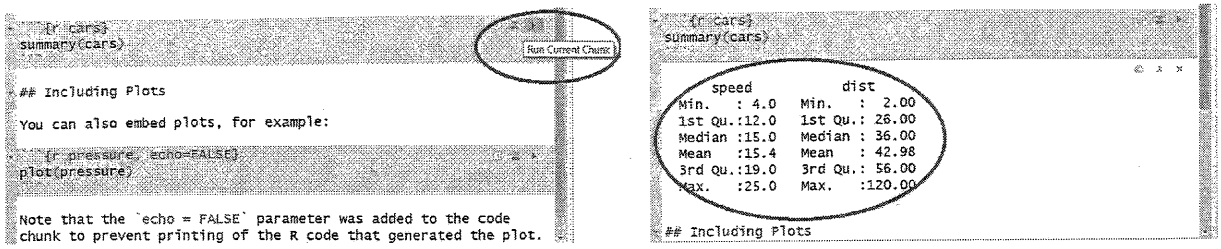


Figure 1.1.3: The typical structure of an R Markdown file.

Ironically and very sadly, in Exam PA the conversion of Rmd files to external documents is disabled, making the Rmd file you compose merely a container of R code and essentially no different from an R script. No knowledge of Rmd syntax is required and perhaps all you have to know about Rmd is how to insert new R chunks to separate your code from your comments. Still, to help you get familiar with Rmd files, all of the R commands that accompany this study manual will be saved as Rmd files with clearly numbered R chunks.

When you open an Rmd file (try `File > New File > R Markdown...`), you will see texts with a white background together with texts highlighted with a gray background, called R *chunks*, as in Figure 1.1.3. R chunks are where R commands are stored, executed, and separated from other texts. To insert a new R chunk, select `Insert > R` at the top right corner of the file. A new chunk with three backticks (‘‘‘) signifying the beginning and end of the chunk is produced. To run all of the code in an R chunk, click the green triangle at the top right corner of the chunk and the output (if any) is shown right below each chunk; see Figure 1.1.4. If you want to run only part of the code in an R chunk, highlight the commands you want to run and click `Run > Run Selected Line(s)` at the top right corner of the Rmd file, or simply press `Ctrl + Enter` (for Windows).

When you click the Rmd file you want to work with, this will automatically open RStudio and at the same time load the file.



(a) Clicking the right triangle runs the code in an R chunk. (b) Output produced by running an R chunk.

Figure 1.1.4: How to run an R chunk and where the output is produced in an Rmd file.

R packages. One of the most important reasons for R's skyrocketing popularity is the abundance of user-contributed packages available for download and installation. These packages make R highly extensible. In R, a *package* is simply a collection of R functions, datasets, and pre-written code designed for a particular purpose. If you think of a statistical technique, chances are that someone has written a package for it and contributed it to CRAN.[iii] Rather than reinvent the wheel, we can build on the efforts of others.

We have covered how to use the GUI provided by RStudio to install and load packages. An alternative that is preferred from a programming point of view is to include the following commands in your Rmd file: (`<package_name>` is the name of the package you want to install or load)

- To install a new package:

```
install.packages("<package_name>")
```

- To load an installed package:

```
library(<package_name>)
```

  Quotation marks are optional.

  (Note: Many people refer to R packages as "libraries," because you use the R function `library()` to load a package. Using the terms packages and libraries interchangeably is innocuous in most instances.)

Explicitly including the commands above has the advantage that everyone who runs your code will install and load the packages automatically.

Despite the large collection of R packages available, the computers at the PA test center will only have a limited set of packages installed to ensure that all candidates have the same set of tools for solving a given problem. There will be no Internet connection at the test center and hence no opportunity to install additional packages. Moreover, package versions will be frozen. The available packages are: (those that play the most important role in Exam PA are underlined)

boot	data.table	ggplot2	pdp	rpart
broom	devtools	glmnet	pls	rpart.plot
caret	dplyr	gridExtra	plyr	tidyverse
cluster	e1071	ISLR	pROC	xgboost
coefplot	gbm	MASS	randomForest	

[iii] See `https://cran.r-project.org/web/packages/index.html`.

As the SOA states in the PA exam syllabus,

> "[i]t is not expected that candidates are familiar with each and every one of them. It is expected that candidates can use a *selection* of these packages to perform the tasks covered in the supporting modules."

Throughout this study manual, we will only make use of the packages above (even if there are more efficient packages for performing the same modeling task) and provide ample illustrations of those underlined.

1.1.2 Data Types

Now that R and RStudio have been installed, it is time to do some very simple programming.

Variable assignments. R, as a statistical programming language, has tools to do essentially all math operations. To begin with, let's ask R to do in CHUNK 1 the easiest math problem of all times: `1 + 1`. (Note: Read page viii of the preface on how to access the Rmd files that accompany this manual, if you have not done so.)

```
# CHUNK 1
1 + 1

## [1] 2
```

You should get 2 as the output (if not, there must be something fundamentally wrong with your R!). Notice that a `[1]` is printed next to the result. This is just a message from R that this line begins with the first value of your result. Later, our commands may return multiple-line output. For example: (we will learn the colon `:` operator in Subsection 1.2.1)

```
21:50

##  [1] 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43
## [24] 44 45 46 47 48 49 50
```

Here the number `[24]` means that the second line starts with the 24th value in the whole sequence of output. In most cases, we can safely ignore the numbers in square brackets.

More often than not, we want to store the results of a calculation for future use. In R, the assignment operator `<-`[iv] (a less than sign followed by a dash) is used to assign an expression to a variable.[v] Run CHUNK 2 to store the result of `1 + 1` in a variable named `a`.

```
# CHUNK 2
a <- 1 + 1
a

## [1] 2
```

[iv] In most cases, the equality sign `=` can also be used to do object assignments and produces the same result as `<-`. However, the equality sign is not recommended by most R programmers.

[v] Though not commonly used, you can reverse the direction of the assignment operator, so `1 + 1 -> a` produces the same result as `a <- 1 + 1`.

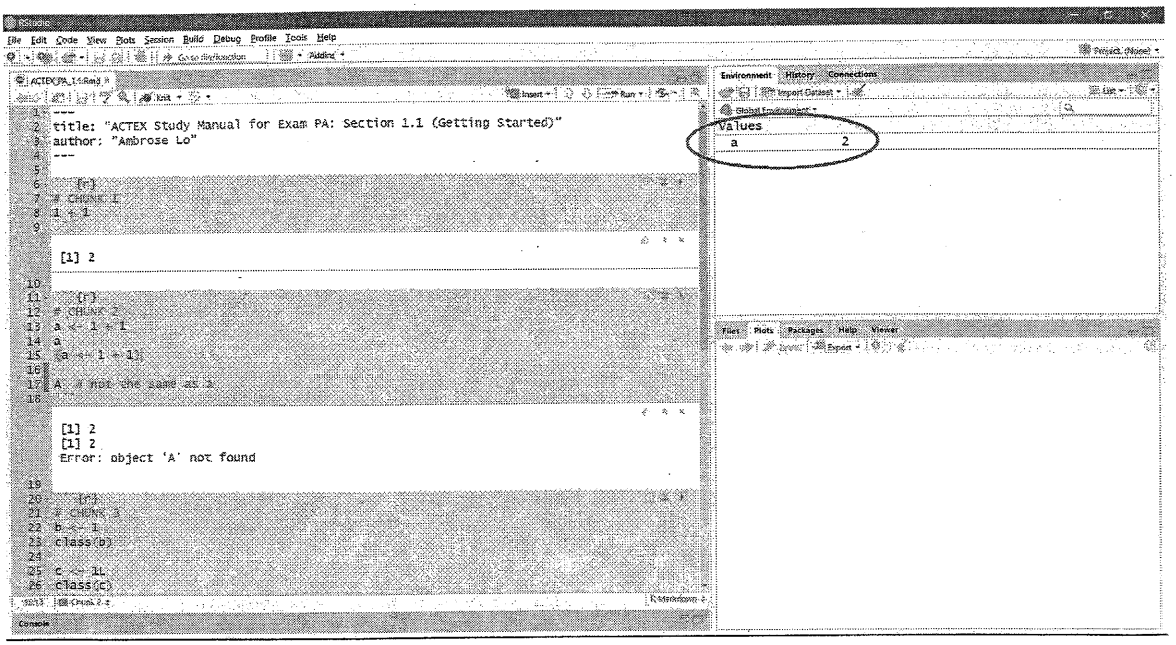


Figure 1.1.5: The environment pane tracks all of the R objects you create.

```
(a <- 1 + 1)

## [1] 2

A  # not the same as a

## Error in eval(expr, envir, enclos): object 'A' not found
```

After running these commands, you can see in the environment pane that a variable named `a` with a value of 2 has been created (see Figure 1.1.5).

Although the code in CHUNK 2 is very simple, there are a few points we should draw your attention to:

- The name of a variable can be any combination of alphanumeric characters together with periods (`.`) and underscores (`_`), except that it cannot start with a number or an underscore. Special symbols such as `^`, `!`, `$`, `*` are also not allowed in any part of a name.
- An assignment statement such as `a <- 1 + 1` does not produce any output by itself. To retrieve the value of the variable just created, we can retype the name of the variable, as we did in the second line of the code. As a shortcut, we can enclose the assignment expression in parentheses, which automatically displays the result immediately after execution.
- Beware that R is case-sensitive, so typing the upper case letter `A` instead of `a` will return an error—`A` is not defined!
- Anything that follows the hash `#` symbol will not be run, so we can use it to precede inline comments, which improve the readability of your code.

Typical data types. In R, we can use a wide variety of objects to store different types of data representing different sorts of information (e.g., numbers and texts). In Exam PA, the three most commonly used data types are numeric, character, and logical.

- *Numeric:* Numeric data is arguably the most common data type in Exam PA and includes both integer and decimal numbers, positive and negative. By default, numeric data is saved as a double (a computer science term meaning numbers that can have decimal places). To specifically create an integer in R, append the integer with an uppercase L. Run CHUNK 3 to see how this works.

```
# CHUNK 3
b <- 1
class(b)

## [1] "numeric"

c <- 1L
class(c)

## [1] "integer"
```

The `class()` function (functions are discussed further in Subsection 1.2.5) returns the data type of an object.[vi] As we can see, `b` is a numeric variable and `c` is an integer, a special type of numeric variable.

- *Character:* Character data is for storing small pieces of text (e.g., names, addresses, comments) surrounded by quotes. In Exam PA, character data can be used to represent categorical variables. Run CHUNK 4 to create two character variables.

```
# CHUNK 4
d <- "I love you!"
d

## [1] "I love you!"

class(d)

## [1] "character"

e <- "123"  # character data, not numeric data
e

## [1] "123"

class(e)

## [1] "character"
```

[vi] An R object is, loosely speaking, anything that can be assigned a value in R.

Note that although `e` is a character string of numeric values, it is still a character variable instead of a numeric variable, which is confirmed by running the command `class(e)`.

- *Logical:* Logical (a.k.a. Boolean) variables are used to store the special values `TRUE` and `FALSE` (all letters in uppercase). In Exam PA, logical variables often arise through logical comparisons (e.g., `1 < 2`; see Table 1.1 on page 46), equal to `TRUE` if the comparison statement is true and `FALSE` otherwise, and can be used as "indicators" or "flags" that highlight whether a characteristic or event is true or not for particular observations or variables in our dataset. When arithmetic operators are applied to logical variables, `TRUE` is treated as numerically equivalent to 1 and `FALSE` numerically equivalent to 0. Run CHUNK 5 to see some manipulations of logical variables.

```
# CHUNK 5
1 < 2

## [1] TRUE

class(1 < 2)

## [1] "logical"

TRUE * 3   # equal to 1 * 3

## [1] 3

FALSE + 4  # equal to 0 + 4

## [1] 4

T          # equal to TRUE

## [1] TRUE

T <- 100   # override T by another value
T

## [1] 100
```

It should be pointed out that although R provides `T` and `F` as shorthand for `TRUE` and `FALSE`, respectively, these two abbreviated symbols can be inadvertently overwritten as in the last part of the code in CHUNK 5 and it is best to spell out `TRUE` and `FALSE` in full.

In Section 1.3, we will learn different data structures to store different types of data in a single object.

Sidebar: R programming style. You may have noticed that we have put white space around operators such as <- and +. Doing so does not affect whether or not the code works, but is considered good coding practice. Throughout this study manual, we will follow Google's R Style Guide that is mentioned in PA Module 1. Here we highlight some of the key points in the style guide:

- Put spaces around all binary operators such as =, +, -, <-.

  Good: `x < 1`

  Bad: `x<1`

- Do not put a space before a comma, but always put one after a comma.

  Good: `df[, 2]`

  Bad: `df[ , 2]` and `df[ ,2]`

- Do not place spaces around code in parentheses or square brackets, with the exception that a space is always placed after a comma.

  Good: `if (x <= 1)` and `df[1, ]`

  Bad: `if ( x <= 1 )` and `df[1,]`

- Short comments can be put after code preceded by two spaces, #, followed by one space, e.g.,

```
a <- 1 + 1  # to save the result of 1 + 1 in an object named a
```

- Function definitions should first list arguments without default values, followed by those with default values.

Note that graders for Exam PA will not be evaluating how closely you follow these programming style guidelines (ironically, many of the Rmd files provided by the SOA defy these guidelines!). However, writing code with these guidelines in mind will make your code more readable and help graders understand your work better.

1.2 Data Structures

There are a wealth of structures in R that we can use to hold data. These *data structures* (not to be confused with "data types" in the preceding section) are distinguished by the type of data they can hold, their structural flexibility, and the notation used to extract individual elements. In this section, we will cover a number of data structures that are particularly important in Exam PA. Here is an overview:

Name	Dimension	Individual elements/components of same data type?
Vectors	One-dimensional	Yes
Matrices	Two-dimensional	Yes
Data frames	Two-dimensional	Potentially different
Lists	One-dimensional	Potentially different

1.2.1 Vectors

Vector operations. A distinguishing property of R is that it is a *vectorized* programming language, meaning that operations are applied to a vector element-wise without the need for looping through the entire vector. In R, a *vector* is defined as an ordered (i.e., the order matters) collection of elements, all of which are of the same type (e.g., numeric, character, logical). The easiest way to create a vector is through the `c()` function (standing for "concatenate" or "combine"), which strings the comma-separated arguments inside the parentheses together to form a vector. For the purpose of illustration, let's build four vectors in CHUNK 1.

```
# CHUNK 1
a <- c(1, 2, 3, 4, 5)
b <- c(5, 4, 3, 2, 1)
c <- c("A", "B", "C")
d <- c(TRUE, FALSE, FALSE, TRUE, TRUE)
```

Here `a` and `b` are numeric vectors of the same length (which is 5), `c` is a character vector, and `d` is a logical vector. A shorthand for creating `a` is `1:5`, where the colon operator `:` generates a sequence of consecutive integers in either direction. Similarly, `b` can be created by the command `5:1`. More generally, the `seq(from, to, by)` function returns a sequence of numbers (not necessarily integers) starting from the number indicated by the `from` argument and ending with the number indicated by the `end` argument, with the `by` argument specifying the increment of the sequence. For instance:

```
# CHUNK 1 (Cont.)
seq(0, 1, 0.25)

## [1] 0.00 0.25 0.50 0.75 1.00
```

The length of a vector can be found by the `length()` function. Note that unlike in linear algebra, there is no such thing as a row vector or a column vector in R.

```
# CHUNK 1 (Cont.)
length(a)

## [1] 5

length(c)

## [1] 3
```

To add, multiply, or raise each element of a numeric vector by a common factor, it suffices to apply the operation directly to the vector itself. No loops are needed. CHUNK 2 contains a series of examples.

```
# CHUNK 2
a + 2   # add 2 to each element of a

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

```
a * 3   # multiply each element of a by 3

## [1]  3  6  9 12 15

a^2     # square each element of a

## [1]  1  4  9 16 25

a + b   # usual vector addition

## [1] 6 6 6 6 6

a * b   # element-wise product of a and b; not dot product

## [1] 5 8 9 8 5

a / b   # element-wise division of a and b

## [1] 0.2 0.5 1.0 2.0 5.0

a <= b  # determine if each element of a is <= corresponding element of b

## [1]  TRUE  TRUE  TRUE FALSE FALSE

c(a, b) # combine a and b end to end to form a longer vector

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

Note that `a <= b` returns a logical vector of the same length as `a` and `b` that contains `TRUE` or `FALSE` corresponding to each element-wise comparison.

Things become a bit more complex when we deal with two vectors of unequal length. The way R handles this is to recycle the shorter vector by repeating its elements to match the longer vector. We have already seen one such example when typing `a + 2`. Here the scalar `2` can be considered a one-element vector that gets recycled to form the 5-element vector `c(2, 2, 2, 2, 2)`, which is added to `a` to produce the final result. CHUNK 3 contains a more interesting example:

```
# CHUNK 3
a + 1:3

## Warning in a + 1:3:  longer object length is not a multiple of shorter object length

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

To match the length of `a`, `1:3` is expanded to the 5-element vector `c(1, 2, 3, 1, 2)` and R is computing the sum of `a` and `c(1, 2, 3, 1, 2)`. A schematic diagram showing how the two vectors line up is given in Figure 1.2.1. You can also see the warning message

`longer object length is not a multiple of shorter object length`,

which is produced if the length of the longer vector is not a multiple of that of the shorter vector, as in this case.

a (longer vector)		1:3 (shorter vector)		a (longer vector)		1:3 (shorter vector)		
1		1		1		1		2
2		2		2		2		4
3	+	3	→	3	+	3	=	6
4				4		1 (recycled)		5
5				5		2 (recycled)		7

Figure 1.2.1: A pictorial illustration of how vectors of unequal length are dealt with in R.

Vectors are the backbone of R programming and play a central role in virtually every corner of R in general. In Exam PA in particular, vectors are often used to store the values of a variable in a statistical model, as we will see in the remainder of this study manual.

Extracting subsets of vectors. R has effective indexing features that allow us to extract or exclude the elements of a data object for subsequent analysis. In the case of a vector, there are several common ways to do *subsetting* (also known as *filtering*). In every case, we append an *index vector* in square brackets `[]` immediately following the name of the given vector, i.e.,

`<vector> [ <index_vector> ].`

- *A vector of positive integers:* When we use a vector of positive integers[vii] as the index vector, only elements corresponding to the positive integers in the index vector are extracted and concatenated, in that order, to form a sub-vector. To select only one particular element, use a one-element vector, that is, a scalar. Let's look at two examples in CHUNK 4.

```
# CHUNK 4
a[2]        # second element of a

## [1] 2

a[c(2, 4)] # second and fourth elements of a

## [1] 2 4
```

- *A vector of negative integers:* If a vector of negative integers serves as the index vector, then the elements corresponding to the negative subscripts are omitted from the whole vector. For example:

```
# CHUNK 4 (Cont.)
b[-1]        # all but the first element of b

## [1] 4 3 2 1
```

[vii] The values in the index vector must be one of `1, 2, ..., length(x)`, where `x` is the vector whose elements are to be extracted. Note that unlike in other programming languages, indices in R start at 1 rather than 0.

```
b[-(2:4)]   # remove the second to fourth elements of b

## [1] 5 1
```

Using a vector of negative integers is more efficient than using a vector of positive integers when you want to include the majority of the elements of a vector.

- [IMPORTANT!] *A logical vector:* A more subtle, but surprisingly useful way to take out the elements of a vector is to use a logical vector of the same length as the index vector. Only elements corresponding to `TRUE` in the index vector are extracted. This way of extracting the elements of an R object is known as *logical subsetting*. For the numeric vector `a` and logical vector `d` constructed earlier, consider:

```
# CHUNK 4 (Cont.)
a[d]

## [1] 1 4 5
```

Since only the first, fourth, and fifth elements of `d` are `TRUE`, `a[d]` returns the first, fourth, and fifth elements of `a`.

Logical subsetting may appear weird at first sight—why use such an indirect way to pull out the elements of a vector? We will defer a full explanation to Section 1.3, where we will see that logical subsetting proves extremely useful in numerous data management tasks.

Example 1.2.1. (Left truncation using a logical vector) Consider the following R commands:

```
x <- c(1, 10, 8, 2, 6)
x[x > 4]
```

Can you guess the final output? Run CHUNK 5 to check your answer.

Solution. The command `x > 4` returns a 5-element logical vector whose elements are equal to `TRUE` (resp. `FALSE`) if the corresponding element of `x` is strictly larger than (resp. less than or equal to) 4. Here is how the logical vector is defined.

```
x > 4

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

Only the second, third, and fifth elements are `TRUE`. It follows that when this logical vector serves as the index vector, only the second, third, and fifth elements of `x` are returned.

```
x[x > 4]

## [1] 10  8  6
```

□

Remark. In the language of Exam C/STAM, the above commands can be used to perform left truncation with a truncation level of 4. If we think of `x` as a loss vector, then only losses above 4 (like a deductible) are kept in the left-truncated vector `x[x > 4]`.

Factors. *Factors* are a special type of character vector with predefined *levels*; its distinct values are recorded, unlike a plain character vector. Factors are important in Exam PA because they determine how *categorical* variables (e.g., gender, ethnicity, socioeconomic class) are stored in R, how they are represented in graphical displays, and, most importantly, how they should be incorporated into predictive models.

To create a factor, we apply the `factor()` function to a vector. Run CHUNK 6 to see a factor vector for gender with three levels, `M` (Male), `F` (Female), and `O` (Others).

```
# CHUNK 6
x <- c("M", "F", "F", "O", "M", "M")  # a character vector
x

## [1] "M" "F" "F" "O" "M" "M"

x.factor <- factor(x)  # now a factor
x.factor

## [1] M F F O M M
## Levels: F M O

levels(x.factor)

## [1] "F" "M" "O"

length(levels(x.factor))

## [1] 3

as.numeric(x.factor)

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

When you type the name of a factor (`x.factor` in this case), R prints out all of its elements in order, followed by its distinct levels. To see the possible levels of a factor, use the `levels()` function. The command `length(levels(<factor>))` counts the number of distinct levels and is a measure of the dimension of a factor variable.

Technically, R assigns a unique integer to each distinct level of a factor and ties it back to the original character string representation. For `x.factor`, it is stored internally as 2, 1, 1, 3, 2, 2, with the three integers 1, 2, and 3 corresponding to `F`, `M`, and `O` (in alphabetical order), respectively. This integer representation of `x.factor` can be revealed by applying the `as.numeric()` function to the factor.

Factors can also be created from numeric vectors, in which case the levels indicated by the different numeric values are merely class labels; they do not convey any sense of relative magnitude. As we will see in the next chapter, whether a numeric vector is treated as a factor has a huge impact on how it is represented graphically. Run CHUNK 7 to see an example of a factor with two levels indicated by `0` and `1`.

```
# CHUNK 7
y <- c(0, 1, 1, 0, 1, 1)
y

## [1] 0 1 1 0 1 1

y.factor <- factor(y)
y.factor

## [1] 0 1 1 0 1 1
## Levels: 0 1
```

Although not commonly done in Exam PA, the levels of a factor can be ordered if desirable, in which case the factor represents an *ordinal* categorical variable, by setting the `order` argument of the `factor()` function to `TRUE`, as in CHUNK 8.

```
# CHUNK 8
z <- c("Good", "Excellent", "Bad", "Good", "Bad", "Good")
z.factor <- factor(z, order = TRUE)
z.factor

## [1] Good      Excellent Bad       Good      Bad       Good
## Levels: Bad < Excellent < Good

z.factor <- factor(z, levels = c("Bad", "Good", "Excellent"), order = TRUE)
z.factor

## [1] Good      Excellent Bad       Good      Bad       Good
## Levels: Bad < Good < Excellent
```

Note that by default the three distinct levels of `z` are ordered alphabetically as `Bad < Excellent < Good`, which is weird. To override the default, we have to explicitly spell out the desired order through the `levels` argument.

1.2.2 Matrices

Creation of matrices. *Matrices* generalize vectors by providing a two-dimensional container of data of the same type (e.g., numeric, character, logical). In R, matrices can be set up by the `matrix()` function, whose general syntax is as follows:

```
matrix(data = <vector>, nrow = <nrow>, ncol = <ncol>, byrow = FALSE),
```

where the `data` argument supplies the elements of the matrix, `nrow` and `ncol` determine the number of rows and columns, respectively, and the `byrow` option specifies whether the elements of the matrix are filled in by row or by column (default). Let's construct four matrices in CHUNK 9.

```
# CHUNK 9
A <- matrix(1:8, ncol = 2)
A

##      [,1] [,2]
## [1,]    1    5
## [2,]    2    6
## [3,]    3    7
## [4,]    4    8

B <- matrix(9:16, nrow = 4, byrow = TRUE)
B

##      [,1] [,2]
## [1,]    9   10
## [2,]   11   12
## [3,]   13   14
## [4,]   15   16

C <- matrix(c(2, 1, 1, 4), ncol = 2)
C

##      [,1] [,2]
## [1,]    2    1
## [2,]    1    4

D <- matrix(LETTERS[1:6], nrow = 3)
D

##      [,1] [,2]
## [1,] "A"  "D"
## [2,] "B"  "E"
## [3,] "C"  "F"
```

Note that `A` is a numeric matrix with 8 elements and 2 columns. The number of rows is therefore $8/2 = 4$ and is calculated by R automatically. Similarly, `B` has the same dimension (4×2), but has its elements filled in by row, and `C` has $4/2 = 2$ rows. Finally, D is a 3×2 character matrix containing the first six English letters, all in uppercase, generated by the special vector `LETTERS[1:6]` (`letters[1:6]` will return the first six letters all in lowercase).

There are not many instances in Exam PA in which we will deal with matrices manually, so our treatment of matrices here is rather brief. The most important use of matrices in the exam is to apply the `model.matrix()` function to extract the design matrix for a (generalized) linear model, as we will see in Section 3.4 of this manual.

Matrix attributes. While the `length()` function returns the length of a vector, its counterpart for matrices include `nrow()`, `ncol()`, `dim()`, which compute the number of rows, columns and the dimension of a matrix, respectively. Run CHUNK 10 to calculate these attributes for the four matrices defined in CHUNK 9.

```
# CHUNK 10
nrow(A)

## [1] 4

ncol(B)

## [1] 2

dim(C)

## [1] 2 2

dim(D)

## [1] 3 2
```

These attributes are computed usually only when the matrix of interest is so large in size that they cannot be read off by inspection.

Matrix operations. Just like vectors, element-wise operations on matrices can be performed in the usual way, provided that the dimensions of the matrices are compatible. A matrix of the same dimension is produced. For example:

```
# CHUNK 11
A + B  # element-wise addition of A and B

##      [,1] [,2]
## [1,]   10   15
## [2,]   13   18
## [3,]   16   21
## [4,]   19   24

A * B  # element-wise product of A and B

##      [,1] [,2]
## [1,]    9   50
## [2,]   22   72
## [3,]   39   98
## [4,]   60  128

A == B

##       [,1]  [,2]
## [1,] FALSE FALSE
## [2,] FALSE FALSE
## [3,] FALSE FALSE
## [4,] FALSE FALSE
```

To perform matrix multiplication rather than element-by-element multiplication, the operator %*% is used. Of course, we have to make sure that the matrices being multiplied are dimension-compatible.

```
# CHUNK 12
A %*% B  # returns an error

## Error in A %*% B: non-conformable arguments

A %*% C

##      [,1] [,2]
## [1,]    7   21
## [2,]   10   26
## [3,]   13   31
## [4,]   16   36
```

Matrix transposition and inversion are rarely used in Exam PA, but it is a good idea to know that they can be accomplished by the `t()` and `solve()` functions, respectively.

```
# CHUNK 12 (Cont.)
t(A)

##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    5    6    7    8

solve(C)

##            [,1]       [,2]
## [1,]  0.5714286 -0.1428571
## [2,] -0.1428571  0.2857143
```

Matrices are two-dimensional in structure and, just like vectors, they are allowed to contain data of the same type only, which is still a huge limitation. For multi-dimensional data, we can use *arrays* (which are not used in any of the PA modules; please read Section 5.4 of *R for Everyone* if you are interested). To allow for different types of data across different columns, we can use data frames, the subject of the next subsection.

Example 1.2.2. (Calculating least squares estimates by matrix manipulation) You are fitting a multiple linear regression model

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \varepsilon$$

to the following dataset:

i	Y_i	X_{i1}	X_{i2}
1	14	2	4
2	11	3	2
3	14	0	6
4	18	8	4
5	12	4	3
6	9	1	2

Perform the following two tasks in R.

(a) Set up the response vector $\mathbf{Y}$ and design matrix $\mathbf{X}$ corresponding to the dataset above.

(b) Use the matrix formula $\hat{\beta} = (\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{Y}$ to calculate the least squares estimates of $\beta_0, \beta_1, \beta_2$.

Solution. The R commands are collected in CHUNK 13. Let's look at the code line by line.

(a) The response vector can be created by the following command:

```
Y <- c(14, 11, 14, 18, 12, 9)
```

To set up the design matrix, we first create the three columns.

```
X0 <- rep(1, 6)  # intercept column; same as c(1, 1, 1, 1, 1, 1)
X1 <- c(2, 3, 0, 8, 4, 1)
X2 <- c(4, 2, 6, 4, 3, 2)
```

The first column is a vector of 1's corresponding to the intercept of the model. It is created by the `rep(x, n)` function, which, in its simplest form, repeats or replicates the `x` vector `n` times. The other two columns represent the values of X_1 and X_2. Now we combine these three vectors via `c(X0, X1, X2)` and use the `matrix()` function to produce the design matrix.

```
X <- matrix(c(X0, X1, X2), nrow = 6)
X
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    4
## [2,]    1    3    2
## [3,]    1    0    6
## [4,]    1    8    4
## [5,]    1    4    3
## [6,]    1    1    2
```

(b) Using the matrix formula $\hat{\beta} = (\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{Y}$, we can calculate the least squares estimates of $\beta_0, \beta_1, \beta_2$ by the following commands:

```
B <- solve(t(X) %*% X) %*% t(X) %*% Y
B
```

```
##           [,1]
## [1,] 5.2505543
## [2,] 0.8137472
## [3,] 1.5166297
```

Thus $\hat{\beta}_0 = 5.2506$, $\hat{\beta}_1 = 0.8137$, and $\hat{\beta}_2 = 1.5166$.

□

Remark. In Chapter 3, we will learn how to use the `lm()` function to fit a linear model and the `coef()` function to extract the least squares estimates.

```
m <- lm(Y ~ X1 + X2)
coef(m)

## (Intercept)          X1          X2
##   5.2505543   0.8137472   1.5166297
```

1.2.3 Data Frames

Basic properties. A substantially more general version of matrices, *data frames* are central to the use of many R graphics and modeling functions and are justifiably the most important data structure in Exam PA.[viii] From a technical point of view, they can be thought of as an ordered collection of vectors combined column by column. These vectors must be of the same length, but, unlike matrices, can be of *different types*, i.e., some columns can be numeric while some can be character or logical. This allows for a wide variety of data types all stored within the same data structure for analysis. For example, in CHUNK 14 we use the `data.frame()` function to create a data frame with three columns named `x`, `y`, and `z`, the first two of which are numeric vectors and the last one is a character vector.

```
# CHUNK 14
x <- 6:10
y <- 11:15
z <- c("one", "two", "three", "four", "five")
df <- data.frame(x, y, z)  # construct a data frame with x, y, and z as columns
df

##    x  y     z
## 1  6 11   one
## 2  7 12   two
## 3  8 13 three
## 4  9 14  four
## 5 10 15  five
```

From a statistical point of view, each column of a data frame can be interpreted as observed values of the same *variable* (that's why each column must have only one mode of data) and each row represents the set of measurements taken or characteristics for the same *observation* (a.k.a. record) corresponding to these variables, much like an Excel spreadsheet. The variables across different columns are allowed to be of different types and capture different kinds of information; see the

[viii] Module 4 of the PA modules refers to a data frame as structured data.

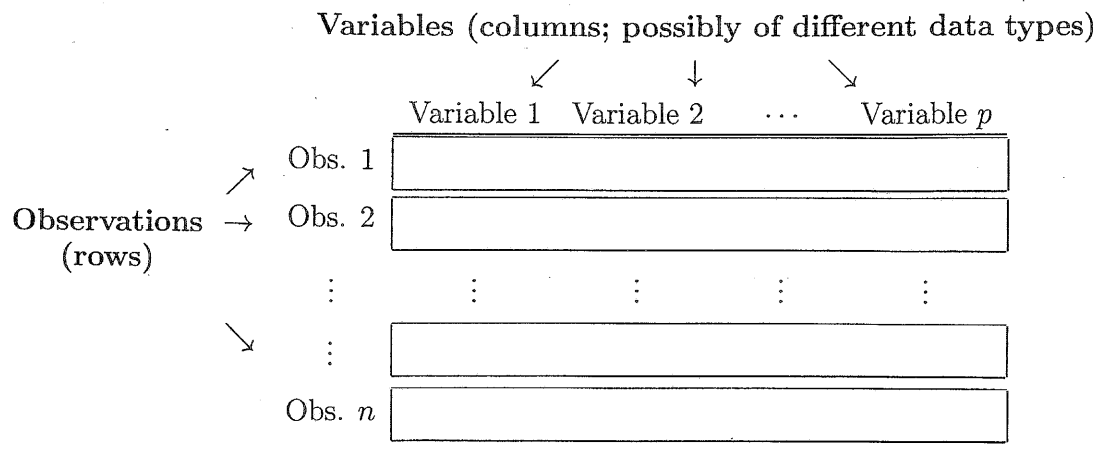


Figure 1.2.2: A pictorial illustration of the structure of a generic data frame.

pictorial illustration in Figure 1.2.2. For instance, if we use a data frame to construct a dataset for a book of life insurance policies, then its columns (variables) may represent information such as the names of policyholders, the amount insured, and the age of the policyholders, and each row (observation) may carry the information for a particular policy. This observation-by-variable arrangement parallels the usual rectangular way we think of a dataset and makes a data frame almost synonymous with a dataset. In the rest of this manual, the terms "dataset" and "data frame" will be used interchangeably.

The `data.frame()` function is the most primitive way to construct a data frame and is used mostly when the data frame in question is relatively small or we want to create one on the fly (some statistical functions such as `predict()` require a data frame as an argument). In Exam PA, almost always you will read in an external CSV file to create a data frame with thousands of observations. For such a large data frame, it is often useful to use the `head(<data frame>, n)` function to print out only the first few rows to get a sense of the data. By default, only the first six rows will be printed, but this can be changed by the `n` argument. The `tail(<data frame>, n)` function is dual to `head()` and prints out the last few rows. Let's try out the `head()` and `tail()` functions in CHUNK 15.

```
# CHUNK 15
head(df, n = 3)  # print out the first three rows

##   x  y     z
## 1 6 11   one
## 2 7 12   two
## 3 8 13 three

tail(df, n = 2)  # print out the last two rows

##    x  y    z
## 4  9 14 four
## 5 10 15 five
```

Attributes. A data frame has a number of attributes that can be extracted by functions.

- *Number of rows and columns:* Just like matrices, the number of rows, columns, and the dimension of a data frame can be checked by the `nrow()`, `ncol()`, and `dim()` functions, respectively. In a statistical analysis where the dataset is hosted in a data frame, `nrow()` has a special meaning: It represents the sample size of the data. Run CHUNK 16 to compute these quantities for the `df` data frame.

```
# CHUNK 16
nrow(df)

## [1] 5

ncol(df)

## [1] 3

dim(df)

## [1] 5 3
```

- *Column names:* The `names()` (or `colnames()`) function, when applied to a data frame, returns a character vector that contains the names of the columns of the data frame. If column names are not assigned during the initialization stage, then the `names()` function allows us to do the naming at a later stage.

```
# CHUNK 16 (Cont.)
df <- data.frame(6:10, 11:15, c("one", "two", "three", "four", "five"))
df

##   X6.10 X11.15 c..one....two....three....four....five..
## 1     6     11                                      one
## 2     7     12                                      two
## 3     8     13                                    three
## 4     9     14                                     four
## 5    10     15                                     five

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

##    x  y     z
## 1  6 11   one
## 2  7 12   two
## 3  8 13 three
## 4  9 14  four
## 5 10 15  five
```

Subsetting a data frame. Like vectors, individual elements of a data frame can be accessed by subscripts specified inside the square brackets `[]`. However, two[ix] index vectors, the first one for rows and the second one for columns, are required due to the two-dimensional nature of a data frame. As in the case of vectors, the index vectors can be vectors of positive integers, negative integers, or logical values. Consider the following examples in CHUNK 17:

```
# CHUNK 17
df[2, 1]                          # (2,1)-element of df

## [1] 7

df[3, 2:3]                        # third row, columns 2 through 3

##    y     z
## 3 13 three

df[, 1]                           # first column of df

## [1]  6  7  8  9 10

df[2:4, ]                         # second to fourth rows of df

##   x  y     z
## 2 7 12   two
## 3 8 13 three
## 4 9 14  four

df[, c(TRUE, TRUE, FALSE)]  # first two columns of df

##    x  y
## 1  6 11
## 2  7 12
## 3  8 13
## 4  9 14
## 5 10 15
```

In the command `df[, 1]`, the row index vector is left out (make sure to include the empty row subscript!), which means that all of the five rows are returned. In general, "blank" means "all." The result is that the entire first column of the data frame is extracted. The same idea applies to `df[2:4, ]` to access the second to fourth rows of the data frame. The last example `df[, c(TRUE, TRUE, FALSE)]` involves logical subsetting. Since only the first two elements of the column index vector are `TRUE`, it follows that the command returns a new data frame that contains the first two columns of the original data frame.

There are alternative ways to access specific columns of a data frame that are absent for vectors but are used quite often in Exam PA. One way is the dollar sign $ notation which can be used to identify a *particular column by name* (assuming that the columns of the data frame have been

[ix] Technically, you can also supply just one subscript representing the columns. The resulting object is a sub-data frame rather than a vector. See page 62 of *R for Everyone* if you are interested.

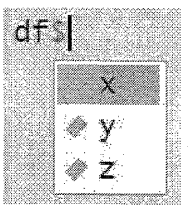
named, which is almost always the case in Exam PA). To extract multiple columns by name, supply a character vector of column names to the column subscript and leave the row subscript blank. Consider the following commands in CHUNK 18:

```
# CHUNK 18
df$y                  # no quotes should go around the column name

## [1] 11 12 13 14 15

df[, c("x", "z")]  # quotes should be used

##    x     z
## 1  6   one
## 2  7   two
## 3  8 three
## 4  9  four
## 5 10  five
```

Using the dollar sign to access an individual column of a data frame is often preferable to using the square bracket notation with the precise column number specified (i.e., `df[, j]` for some `j`). By identifying the column by name, it intuitively tells us which variable is being dealt with. We can then run a function like `mean()` or `median()` to study the distributional properties of the variable.

```
# CHUNK 18 (Cont.)
mean(df$y)

## [1] 13
```

The dollar sign notation also dispenses with the need to count which column in the data frame corresponds to the variable in question. For a small data frame like `df`, it is easy to see at a glance the column numbers. The dataset in Exam PA, however, can involve 20 or more variables. In that case, extracting a variable by name makes it easy for us to get the right component even if we forget the column numbers or if the column numbers change (which can happen when new variables are added to the dataset). In fact, as soon as you have typed the dollar sign $, RStudio cleverly shows a prompt listing all of the variables in the data frame (press `Tab` to make the prompt appear at once).



1.2.4 Lists

Basic properties. A *list* is an ordered collection of objects, known as *components*, which can be any of the objects we have seen so far (i.e., a vector, matrix, data frame, or even another list). Different components can belong to different structures of data, so, for example, a character vector,

a numeric vector, a matrix, and a data frame can co-exist in the same list. This provides a list with huge flexibility with data storage. A list is created by the `list()` function, where the arguments are the components of the list. CHUNK 19 presents an example.

```
# CHUNK 19
l <- list(p = "This is a list",
          q = 4:6,
          r = matrix(1:9, nrow = 3),
          s = data.frame(x = c(1, 4, 9)))
l

## $p
## [1] "This is a list"
##
## $q
## [1] 4 5 6
##
## $r
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
##
## $s
##   x
## 1 1
## 2 4
## 3 9
```

Here `l` is a four-component list. The first is a character string (i.e., a one-element character vector), the second is a three-element numeric vector, the third is 3×3 numeric matrix, and the fourth is a one-column data frame. These four components are named as `p`, `q`, `r`, and `s` to facilitate easy retrieval.

Although there are few instances in which we will set up a list manually in Exam PA, lists are still important as the outputs of many model fitting functions we will use in the exam (e.g., `lm()`, `glm()`, `rpart()`, `prcomp()`) are of different data structures and therefore can only be stored in an all-purpose storage tool like a list (see Example 1.2.3 below). Pulling out relevant components from a list to learn about different aspects of the predictive model we fit is a necessary skill we have to learn.

Subsetting a list. Unlike the previous data structures, the components of a list are accessed using *double* square brackets `[[]]`, within which either the component number or the component name (supplied as a character string) is specified. Run CHUNK 20 to see some examples:

```
# CHUNK 20
l[[1]]     # first component of the list

## [1] "This is a list"
```

```
l[["r"]]   # third component (named r) of the list by name

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

l[[2]][3]  # third element of the second component of the list

## [1] 6

l$s        # fourth component of the list by name

##   x
## 1 1
## 2 4
## 3 9
```

We can view the third example `l[[2]][3]` in two stages. First, we take the second component of the list, namely `q`, as what `l[[2]]` does. Then, since `q` itself is a three-element numeric vector, `l[[2]][3]` is the same as `q[3]` and extracts the third element of `q`, namely, 6.

The last example illustrates the use of the dollar sign `$` notation to access the components of a list by name, just as we did for data frames. In fact, this is the usual way we take out components of a list.

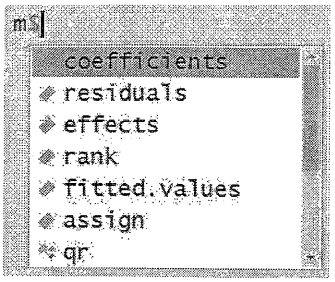
Example 1.2.3. (Example 1.2.2 revisited: Listing the components of a list) In Example 1.2.2, we considered a multiple linear regression model for Y on two predictors X_1 and X_2. As we will learn in Chapter 3, such a model can be easily fitted by the following commands involving the `lm()` function ("`lm`" stands for "linear models"):

```
Y <- c(14, 11, 14, 18, 12, 9)
X1 <- c(2, 3, 0, 8, 4, 1)
X2 <- c(4, 2, 6, 4, 3, 2)
m <- lm(Y ~ X1 + X2)
```

The resulting object, named `m`, is a list with various components.

List the components of `m` and choose the component that appears to contain the least squares coefficient estimates. Check your answers against the code in CHUNK 21.

Solution. We can see a list of components of `m` by typing the command `m$`. Then RStudio will show a prompt listing all of the components of `m`.



Alternatively and more formally, we can apply the `names()` function to `m`:

```
names(m)

##  [1] "coefficients"  "residuals"     "effects"       "rank"
##  [5] "fitted.values" "assign"        "qr"            "df.residual"
##  [9] "xlevels"       "call"          "terms"         "model"
```

Out of the 12 components, the `coefficients` component appears the most relevant. Accessing this component indeed produces the least squares coefficient estimates:

```
m$coefficients

## (Intercept)          X1          X2
##   5.2505543   0.8137472   1.5166297
```

The results are identical to what we got in Example 1.2.2. □

1.2.5 Sidebar: Functions

In the above discussions, we have seen some examples of built-in functions in R such as `c()`, `seq()`, `data.frame()`, `list()`, and `lm()`. Functions are the mainstay of R. They allow us to automate complex tasks by a single function call and significantly reduce repetition in our code. In fact, even basic arithmetic operators like + are a function in disguise. Before moving on, it pays to take a closer look at functions in R. As the R Markdown file for Section 1.2 of PA Module 1 says,

> "[w]hile it is unlikely you will need to write functions for your exam, it is *worth understanding how R functions work.*"

In the exam, you may be provided with a newly defined function that accomplishes a certain task that cannot be done by existing functions in the base R installation. In the December 2018 Exam PA, for example, you are given a function that computes the loglikelihood of a Poisson model (see Problem 1.5.2). The ability to make good use of this function to evaluate the predictive performance of different models is the key to completing the exam satisfactorily.

Basic structure. Here is the structure of a generic function in R:

```
<function_name> <- function(<arg_1>, <arg_2>, ...) {
  <statement_1>
  <statement_2>
  ...
  return(<value>)
}
```

Here

`<function_name>` is the name of the function created,

`<arg_1>, <arg_2>, ...` are the arguments of the function separated by commas,

`<statement_1>, <statement_2>, ...` are a series of statements that form the body of the function enclosed in a pair of curly braces {},[x] and

`return(<value>)` is the final value returned, which can be of any data type, ranging from a scalar to a list. (If the `return` command is omitted, then the value of the last line of code in the body is returned.)

The function takes the arguments as inputs, processes the inputs according to the body of the function, and returns an output specified on the last line. Notice that RStudio automatically indents the code in the body of the function. Although not technically required, the indentation can make our code easier to read.

To see a toy example of a user-defined R function, consider the `sumDiff()` function constructed in CHUNK 22 that returns a list comprising the sum and difference of two vectors supplied as arguments.

```
# CHUNK 22
sumDiff <- function(x, y) {
  s <- x + y
  d <- x - y
  return(list(sum = s, diff = d))
}

sumDiff(1, 2)$sum

## [1] 3

sumDiff(1:3, 1:3)

## $sum
## [1] 2 4 6
##
## $diff
## [1] 0 0 0
```

In the first example `sumDiff(1, 2)$sum`, we are accessing the `sum` component of the list returned by the function. The second example produces the entire list `sumDiff(1:3, 1:3)` containing both the `sum` and `diff` components.

[x]The curly braces are optional if the body of the function has only one line, but this is rare.

Note that except for the return value, objects created during the execution of a function are local objects that are not accessible outside the function. If you try to retrieve the variable `s` after running the `sumDiff()` function, you will get an error, as CHUNK 23 shows.

```
# CHUNK 23
sumDiff(1, 2)

## $sum
## [1] 3
##
## $diff
## [1] -1

s  # object not found

## Error in eval(expr, envir, enclos): object 's' not found
```

Specifying functional arguments. When we call a function, its arguments can be specified in two ways:

- By *name*: Explicitly pair the name of each argument with its value, i.e., use the call

  `<function_name>(arg_1 = val_1, arg_2 = val_2, ...).`

- By *position*: Omit the names of the arguments, but use the order in which the arguments are listed in the function definition to determine which arguments go with which values, i.e., use the call

  `<function_name>(val_1, val_2, ...).`

Run CHUNK 24 to see both ways of invoking a function.

```
# CHUNK 24
sumDiff(x = 2, y = 1)$diff  # by name

## [1] 1

sumDiff(y = 1, x = 2)$diff  # order does not matter

## [1] 1

sumDiff(y = c(0, 3, 2), c(1, 5, 3))$diff  # only one argument named

## [1] 1 2 1

sumDiff(9:7, 1:3)$diff  # by position

## [1] 8 6 4
```

In the first two examples, we explicitly spell out the names of the two arguments `x` and `y` followed by their values. When parameters are passed by name, the order of the arguments does not matter. In the third example, only the first argument is named and R will automatically match the second, unnamed argument with the remaining argument, namely `x`. In the last example, arguments are passed by position and two unnamed arguments are supplied. In the definition of `sumDiff()` (recall the definition in CHUNK 22), the arguments are specified in the order of `x` followed by `y`, so the first argument `9:7` is mapped to `x` and the second argument `1:3` is mapped to `y`. In general, it is a good practice to pass arguments by name to avoid passing the wrong value to the wrong argument, especially when the function has multiple arguments.

Notice that when a function is called with no arguments specified (either because the function has no arguments or its default argument values are good enough), the empty parentheses `()` are still required.

Default arguments. If a function has a large number of arguments, having to spell out all of the argument values (by name or by position) is tiresome. It will make our life much easier if some arguments of the function are given commonly used *default values* so that these arguments need not be specified when the function is called, but we reserve the right to override the default values if necessary. In R, default values of a function's arguments are specified inside `function()` when the function is first defined. If the default values have to be overridden, then we specify the desired values by name (i.e., using the `argument = argument value` format) when the function is invoked. Run CHUNK 25 to modify the `sumDiff()` function above in a way that the default output type is a list (notice the command `list = TRUE` inside `function()`) containing the sum and difference of the two argument vectors, but can be changed to a data frame by setting `list = FALSE` in the function call.

```
# CHUNK 25
sumDiff.mod <- function(x, y, list = TRUE) {
  s <- x + y
  d <- x - y
  if (list) {
    return(list(sum = s, diff = d))
  } else {
    return(data.frame(sum = s, diff = d))
  }
}

sumDiff.mod(1:3, 4:6)

## $sum
## [1] 5 7 9
##
## $diff
## [1] -3 -3 -3

sumDiff.mod(1:3, 4:6, list = FALSE)

##   sum diff
```

```
## 1   5   -3
## 2   7   -3
## 3   9   -3
```

When defining the modified function, we have made use of the `if-else` construct to control the programming flow. Its general syntax is:

```
if (<condition>) {
  <statement_1>
  <statement_2>
  ...
} else {
  <statement_1>
  <statement_2>
  ...
}
```

Here `<condition>` is a logical variable immediately following the `if` command. If `<condition>` equals `TRUE`, then all statements inside the following curly braces `{}` are executed (you may omit curly braces when there is only a single statement to execute). If `<condition>` fails, then all statements inside the curly braces that follow the `else` command are executed. In the first call of the `sumDiff.mod()` function, a list with two components, `sum` and `diff`, is returned. In the second call, we override the default and set `list = FALSE` so that the function outputs a data frame with `sum` and `diff` as its variables (columns).

To see an example of a function with numeric default values, let's revisit the `head()` function that we introduced when learning data frames. Consulting its help page (type `?head`), we see that this function returns by default the first six rows of a vector, matrix, or data frame, but the number of rows to display can be changed by the `n` argument.

```
# CHUNK 26
head(sumDiff.mod(1:8, 8:1, list = FALSE))          # show first six rows

##   sum diff
## 1   9   -7
## 2   9   -5
## 3   9   -3
## 4   9   -1
## 5   9    1
## 6   9    3

head(sumDiff.mod(1:8, 8:1, list = FALSE), n = 3)   # show first three rows

##   sum diff
## 1   9   -7
## 2   9   -5
## 3   9   -3
```