

# R Programming: Zero to Pro

Yang Feng and Jianan Zhu

2021-08-07



# Contents

<b>Preface</b>	<b>5</b>
<b>1 Introduction</b>	<b>7</b>
1.1 Installation of R, RStudio and R Packages . . . . .	7
1.2 Use R as a Fancy Calculator . . . . .	14
<b>2 R Objects</b>	<b>19</b>
2.1 Object Assignment . . . . .	20
2.2 Numeric Vector, Character Vector, & Logical Vector . . . . .	23
2.3 Create Vectors with Patterns . . . . .	32
2.4 Sort, Rank, & Order . . . . .	40
2.5 Statistical functions on vectors . . . . .	46
<b>3 Data Visualization</b>	<b>51</b>
3.1 Introduction to Datasets . . . . .	51
3.2 Scatterplots . . . . .	57
3.3 Aesthetics in ggplot . . . . .	60
3.4 Smoothline Fits . . . . .	80
3.5 Multiple geoms and Aesthetics . . . . .	88
3.6 Global and Local Aesthetic Mappings . . . . .	94
<b>4 Data Import and Export</b>	<b>101</b>
4.1 Exporting Data to Delimited Files . . . . .	101
4.2 Importing Data from Delimited Files . . . . .	103

4.3	Exporting and Importing Data from Excel Files . . . . .	108
4.4	Working with Data from SPSS, SAS, and Stata Files . . . . .	111
4.5	Save and Restore Objects and Workspace . . . . .	113
<b>5</b>	<b>Data Manipulation</b>	<b>117</b>
5.1	Filter Observations and Objects Masking . . . . .	118
<b>6</b>	<b>Tidy Data</b>	<b>121</b>
6.1	Convert Between Names and Values . . . . .	121
<b>7</b>	<b>Strings</b>	<b>125</b>
<b>8</b>	<b>Statistics</b>	<b>127</b>
8.1	Normal Distribution . . . . .	127
8.2	Other Distributions . . . . .	134
8.3	Random Permutation and Random Sampling . . . . .	135
8.4	Covariance and Correlation . . . . .	137
<b>9</b>	<b>Writing Complicated Codes</b>	<b>139</b>
<b>10</b>	<b>A Case Study: 24 Solver</b>	<b>141</b>

# Preface

This book is for anyone who is interested in learning R and Data Science. It is designed for people with zero background in programming.

We also have a companion R package named **r02pro**, containing the data sets used as well as interactive exercises for each part.



# Chapter 1

## Introduction

This chapter begins with the installation of R, RStudio, and R Packages in Section 1.1, and shows how to use R as a fancy calculator in Section 1.2.

### 1.1 Installation of R, RStudio and R Packages

#### 1.1.1 Download and Install

As a first step, you need to download R and RStudio, whose links are as follows. For both software, you need to choose the version that corresponds to your operation system.

Download R: <https://cloud.r-project.org/>

Download RStudio: <https://rstudio.com/products/rstudio/download/#download>



For a step-by-step demonstration, you can refer to the YouTube video via the following [link](#).

RStudio is an *Integrated Development Environment* for R, which is powerful yet easy to use. Throughout this book, you will use RStudio instead of R to learn R programming. Next, let's get started with a quick tour of RStudio.

#### 1.1.2 RStudio Interface

After opening RStudio for the first time, you may find that the font and button size is a bit small. Let's see how to customize the appearance.

### a. Customize appearance

On the RStudio menu bar, you can click *Tools*, and then click on *Global Options* as shown in the following figure.

Then, you will see a window pops up like Figure 1.1. After clicking on *Appearance*, you can see several drop-down menus including *Zoom* and *Editor font size*, among other choices shown.

- *Zoom* controls the overall scale for all elements in RStudio interface, including the sizes of menu, buttons, as well as the fonts.
- *Editor font size* controls the size of the font only in the code editor.

After adjusting the appearance, you need to click on *Apply* to save our settings.

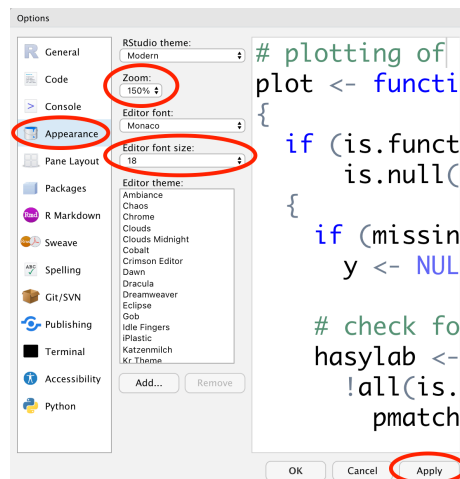


Figure 1.1: Zoom and Editor font size

Here, we change the *Zoom* to 150% and set the *Editor font size* to 18.

### b. Four panels of RStudio

Now, the RStudio interface is clearer with bigger font size. Although RStudio has four panels, not all of them are visible to us at the beginning (Figure 1.2).

In Figure 1.2, we have labeled three useful buttons as 1, 2, and 3. By clicking buttons 1 and 3, you can reveal the two hidden panels.<sup>1</sup> By clicking button 2, we can clear the content in the bottom left panel as shown in the following figure.

<sup>1</sup>Note that you may see different panels hidden when you open RStudio for the first time, depending on the RStudio version. However, you can always reveal the hidden panels by clicking the corresponding buttons like Buttons 1 and 3 in Figure 1.2.



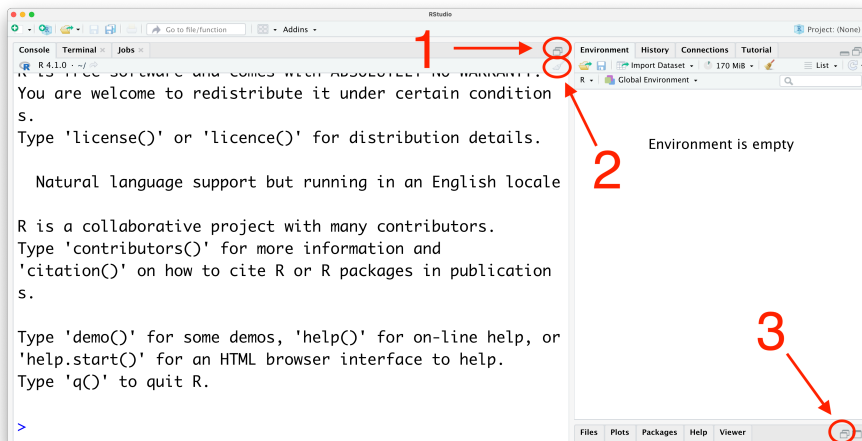


Figure 1.2: Unfold panels

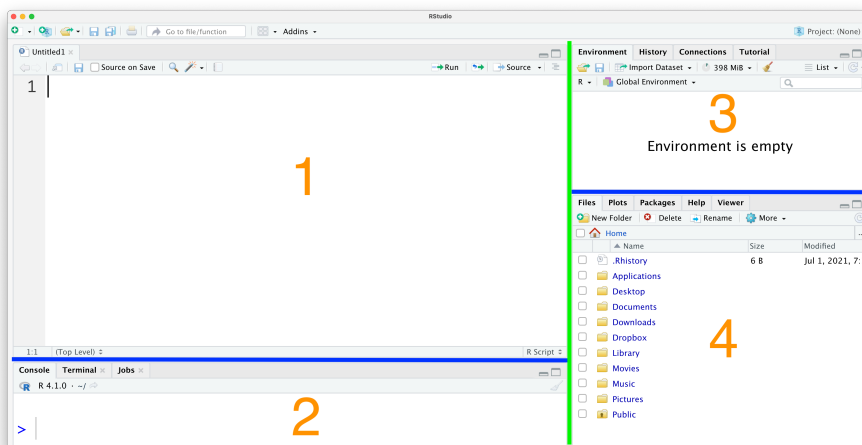


Figure 1.3: Four panels

Now, let's take a close look at all four panels, which are labeled as 1-4 in Figure 1.3. You can change the size of each panel by dragging the two blue slides up or down and the green slide left or right.

- Panels 1 and 2 are located to the left of the green line, and are collectively called the **Code Area**. We will introduce them next.
- Panels 3 and 4 are located to the right of the green line, and are collectively called **R Support Area**. We will introduce these two panels in later sections. **Add the section numbers when available**

### c. Console

Now, let's introduce the panel 2 in Figure 1.3, which is usually called the **Console**.

By clicking the mouse on the line after the `>` symbol, you can see a blinking cursor, indicating that R is ready to accept codes. Let's type `1 + 2` and press Return (on Mac) or Enter (on Windows).



It is a good habit to add spaces around an operator to increase readability of the code.

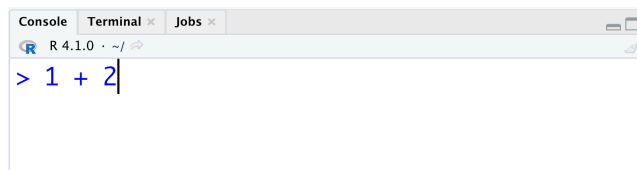


Figure 1.4: Writing code in the console

Hooray! You have successfully ran our first piece of R code and gotten the correct answer 3. Note that the blinking cursor now appears on the next line, ready to accept a new line of code.

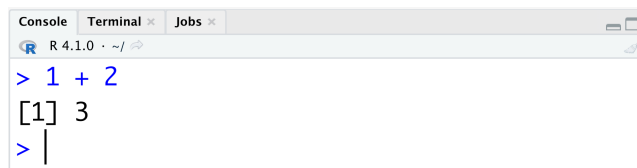


Figure 1.5: R code(2)

Although the console may work well for some quick calculations, you need to resort to the panel 1 in Figure 1.3 (usually called the **Editor**) to save our work and run multiple lines of code at the same time.

#### d. Save R codes as scripts

The **Editor** panel is the go-to place to write complicated R codes, which you can save as R scripts for repeated use in the future.

Firstly, we will introduce how to run codes in scripts. Let's go to the editor and type  $1 + 2$ . To run this line of code, you can click the *Run* button. The keyboard shortcut of running this line of code is  $\text{Cmd} + \text{Return}$  on Mac or  $\text{Ctrl} + \text{Enter}$  on Windows.

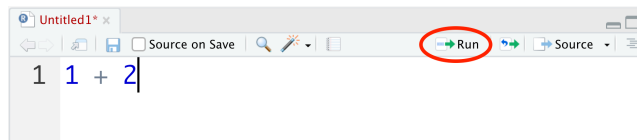


Figure 1.6: script

RStudio will then send the line of code to the console and execute the code.

After finishing writing codes in the editor, you can save them as a script. To do that, you can click the *Save* button as shown in the Figure 1.7. The keyboard shortcut of saving files is  $\text{Cmd} + \text{S}$  on Mac or  $\text{Ctrl} + \text{S}$  on Windows.

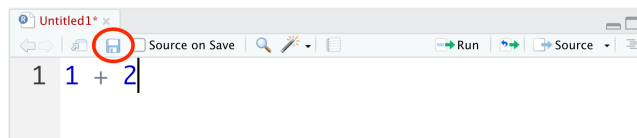


Figure 1.7: Save (I)

Then you would see a pop-up file dialog box, asking you for a file name and location to save it to. Let's call it `lesson1.1` here.

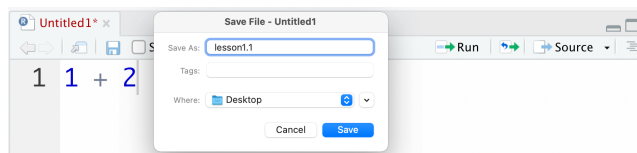


Figure 1.8: Save (II)

After saving files successfully, you can confirm the name of the R script on the top.

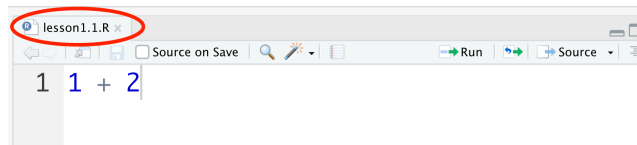


Figure 1.9: Save (III)

Lastly, if you want to create a new R script, we can click the + button on the menu, then select *R Script*. Note that there are quite a few other options including *R Markdown*, which will be introduced in **Section ???**. Then you will see a new file created.

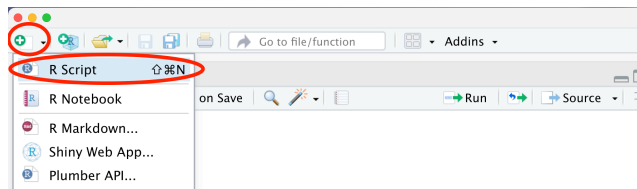


Figure 1.10: create a new script

### 1.1.3 Install and load R packages

Now, you have had a basic understanding of RStudio, it is time to introduce **R packages**, which greatly extend the capabilities of base R. There are a large number of publicly available R packages. As of July 2021, there are more than 17K R packages on Comprehensive R Archive Network (CRAN), with many others located in Bioconductor, GitHub, and other repositories.

To install an R package, you need to use a built-in R **function**, which is `install.packages()`. A **function** takes in **arguments** (inputs) and performs a specific task. After the function name, we always need to put **a pair of parentheses** with the arguments inside.

While there are many built-in R functions, R packages usually contain many useful functions as well, and we can also write our own functions, which will be introduced in Section ???.

With `install.packages()`, the argument is the package name with a pair of quotation marks around it. The task it performs is installing the specific package into R. Here, you will install the companion package for this book, named `r02pro`, a.k.a. *R Zero to Pro*. The `r02pro` package contains several data sets that will be used throughout the book, and interactive exercises for each subsection.

```
install.packages("r02pro")
```



If you miss the right parenthesis, R will show a plus on the next line (as shown in Figure 1.11), waiting for more input to complete the command. If this happens, you can either enter the right parenthesis, or press ESC to escape this command. When you see a blinking cursor after the > symbol, you can write new codes again.

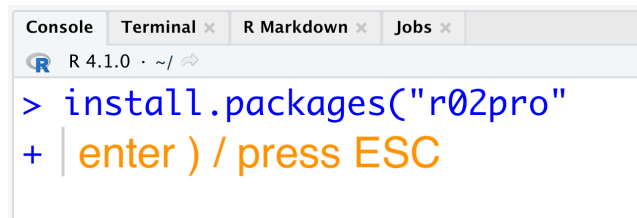


Figure 1.11: Miss the right parenthesis

After a package is installed, you still need to load it into R before using it. To load a package, we use the `library()` function with the package name as its argument. Here, the quotation marks are not necessary.

```
library(r02pro)
```

Note that once a package is installed, you don't need to install it again on the same machine. However, when starting a new R session, you would need to load the package again.



Quotation marks are necessary for installing R packages, but are not necessary for loading packages. If we install packages without quotation marks. We will see an error message, showing *object not found*.

```
install.packages(r02pro)
```

```
#> Error in endsWith(pkgs, ".tgz"): non-character object(s)
```

### 1.1.4 Exercise setup

Having installed and loaded the `r02pro` package, let's introduce how to do the interactive exercise. To setup the exercise, we use the `r02pro()` function with the subsection number as the argument. For example, to do the exercise for Section 1.1, we can run the following code.

```
r02pro(1.1)
```

Upon running the code, a new browser window containing the interactive exercise will pop up. You can then do the exercise. The majority of the exercises ask you to write R codes to accomplish tasks. When finishing writing codes in the corresponding box, you can press the *Run Code* button to run it. **Do we need to have the screenshot for this?**

## 1.2 Use R as a Fancy Calculator

While R is super powerful, it is, first of all, a very fancy calculator.

### 1.2.1 Add comments using “#”

The first item we will cover is about adding comments. In R, you can add comments using the pound sign `#`. In each line, anything after `#` are comments, which will be ignored by R. Let's see an example,

```
6 - 1 / 2 #first calculate 1/2=0.5, then 6-0.5=5.5
```

```
#> [1] 5.5
```

Just looking at the resulting value 5.5, you may not know the detail of the calculation process. The comment informs you the operation order: the division is calculated before the subtraction.

In general, adding comments to codes is a very good practice, as it greatly increases readability and make collaboration easier. We will also add many comments in our codes to help you learn R.

### 1.2.2 Basic calculation

Now let's start to use R as a calculator! You can use R to do addition, subtraction, multiplication division, and combine multiple basic operations. You can also calculate the square root, absolute value and the sign of a number.

Operation	Explanation
$1 + 2$	addition
$1 - 2$	subtraction
$2 * 4$	multiplication
$2 / 4$	division
$6 - 1 / 2$	multiple operations
<code>sqrt(100)</code>	square root
<code>abs(-3)</code>	absolute value
<code>sign(-3)</code>	sign

While the first seven operations in the table look intuitive, you may be wondering, what does the `sign()` function mean here? Is it a stop sign?



Sometimes, you may have no idea how a particular function works. Fortunately, R provides a detailed documentation for each function. There are three ways to ask for help in R.

- Use question mark followed by the function name, e.g. `?sign`
- Use help function, e.g. `help(sign)`
- Use the help window in RStudio, as shown in Figure 1.12. The help window is the panel 4 of Figure 1.3 in Section 1.1. Then type in the function name in the box to the right of the magnifying glass and press return.

### 1.2.3 Approximation

After learning about doing basic calculations, let's move on to do approximation in R. When you do division, for example, when computing  $7 / 3$ , the answer is not a whole number since 7 is not divisible by 3. Under these circumstances, approximation operators are very handy to use. Let's take  $7 / 3$  as the example.

*a. Get the integer part and the remainder*

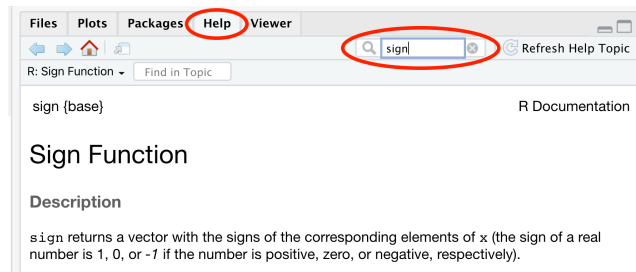


Figure 1.12: Ask for help

Code	Name
<code>7%/%3 = 2</code>	integer division
<code>7%%3 = 1</code>	modulus

We all know that  $7 = 3 * 2 + 1$ . So the *integer division* will pick up the integer part, which is 2 here; and the *modulus* will get the remainder, which is 1.

**b. Get the nearby integer**

```
floor(7 / 3)
ceiling(7 / 3)
```

Since  $2 \leq 7/3 \leq 3$ , you can use the `floor` function to find the *largest integer*  $\leq 7/3$ , which is 2; and the `ceiling` function gives the *smallest integer*  $\geq 7/3$ , which is 3.

**c. Round to the nearest number**

```
round(7 / 3)
round(7 / 3, digits = 3)
```

The `round` function follows the **rounding principle**. By default, you will get the nearest integer to  $7 / 3$ , which is 2. If you want to control the approximation accuracy, you can add a `digits` argument to specify how many digits you want after the decimal point. Here you will get 2.333 after adding `digits = 3`.

### 1.2.4 Power & logarithm

You can also use R to do *power* and *logarithmic* operations.

Generally, you can use `^` to do power operations. For example,  $10^5$  will give us 10 to the power of 5. Here, 10 is the *base* value, and 5 is the *exponent*.



The result is 100000, but it is shown as `1e+05` in R. That's because R uses the so-called *scientific notation*.



**scientific notation:** a common way to express numbers which are too large or too small to be conveniently written in decimal form. Generally, it expresses numbers in forms of  $m \times 10^n$  and R uses the **e notation**. Note that the **e notation** has nothing to do with the natural number  $e$ . Let's see some examples,

$$1 \times 10^5 = 1\text{e}+05 \quad (1.1)$$

$$2 \times 10^4 = 2\text{e}+04 \quad (1.2)$$

$$1.2 \times 10^{-3} = 1.2\text{e}-03 \quad (1.3)$$

In mathematics, the *logarithmic operations* are inverse to the power operations. If  $b^y = x$  and you only know  $b$  and  $x$ , you can do logarithm operations to solve  $y$  using the general form  $y = \log(x, b)$ , which is called the logarithm of  $x$  with base  $b$ .

In R, logarithm functions with base value of 10, 2, or the natural number  $e$  have shortcuts `log10()`, `log2()`, and `log()`, respectively. Let's see an example of `log10()`, the logarithm function with base 10.

```
10^6
log10(1e6) #log10() = log(x, 10)
```

Next, let's see `log2()`, the logarithm function with base 2.

```
2^10
log2(1024) #log2() = log(x, 2)
```

Before moving on to the natural logarithm, note that the natural number  $e$  needs to be written as `exp(1)` in R. When you want to do power operations on  $e$ , you can simply change the argument in the function `exp()`, for example, `exp(3)` is  $e$  to the power of 3. Here, `log()` without specifying the **base** argument represents the logarithm function with base  $e$ .

```
exp(1)
exp(3)
log(exp(3)) #log() = log(x, exp(1))
```

### 1.2.5 Trigonometric function

R also provides the common trigonometric functions.

```
cos(pi)
acos(-1)
```

Here, `acos()` is the inverse function of `cos()`. If we set  $\cos(a) = b$ , then we will get  $\text{acos}(b) = a$ .

```
sin(pi/2)
asin(1)
```

Similarly, `asin()` is the inverse function of `sin()`. If we set  $\sin(a) = b$ , then we will get  $\text{asin}(b) = a$ .

```
tan(pi/4)
atan(1)
```

Also, `atan()` is the inverse function of `tan()`. If we set  $\tan(a) = b$ , then we will get  $\text{atan}(b) = a$ .

### 1.2.6 Exercise

You can run the following code to do the exercise.

```
r02pro(1.2)
```

## Chapter 2

# R Objects

In R, **everything is an object**. An object can contain values of different types including numbers, characters, and any intermediate results from operations. For example, 5 is an object with the value 5.

In the last chapter, we have seen the power of R as a fancy calculator. However, in order to do more complicated and interesting tasks, we may need to store intermediate results for future use.

Let's take a look at a concrete example. Say if you want to do the following calculations involving  $\exp(3) / \log(20,3) * 7$ .

```
(exp(3) / log(20,3) * 7) + 3 #addition  
(exp(3) / log(20,3) * 7) - 3 #subtraction  
(exp(3) / log(20,3) * 7) / 3 #division
```

You need to type the expression three times, which is a bit cumbersome. In this chapter, we will introduce how to assign the value of the object  $\exp(3) / \log(20,3) * 7$  to a name for future use. Then, for any operation involving  $\exp(3) / \log(20,3) * 7$ , you can just use the corresponding object name instead.

In R, there are a few different object types, which we will cover each of them in detail in this Chapter.

## 2.1 Object Assignment

### 2.1.1 Assignment Operation with <-

Firstly, we will introduce how to assign value(s) (of objects) to a name via the *assignment operator*. Let's start with a simple example,

```
x_numeric <- 5
```

The assignment operation has three components.

- From left to right the first component `x_numeric` is the **object name**, which has certain naming rules which we will discuss shortly in Section 2.1.3.
- The second component is the **assignment operator** `<-`, which is a combination of the less than sign `<` immediately followed by the minus sign `-`.
- The final component is the **value** to be assigned to the name, which is 5 here.



There is no space between `<` and `-` in the assignment operator `<-`. Note that although `=` may also appear to be working as the assignment operator, it is not recommended as `=` is usually reserved for specifying the value of arguments in a function call, which will be introduced in Section 2.3.

After running the code above, you will see no output in the console, unlike the case when we ran `1 + 2` which gives us the answer 3 (as shown in the Figure 2.1). You may be wondering, did we successfully make our first assignment operation?

```
R 4.1.0 · ~/
> x_numeric <- 5
> 1 + 2
[1] 3
```

Figure 2.1: No output

To verify it, you can run the code with just the object name to check its value.

```
x_numeric
```

```
#> [1] 5
```

Great! You get the value 5, indicating that you have successfully assigned the value 5 to `x_numeric`.

In addition to assigning the value to a name, you can also assign the value of any R expression to it. In this case, R will first calculate the value of the expression and assign the value to the name. Let's see the following example.

```
y_numeric <- exp(3) / log(20,3) * 7  
y_numeric
```

```
#> [1] 51.56119
```

Using the object name `y_numeric`, you can do the same three calculations introduced at the beginning of this chapter as follows.

```
y_numeric + 3  
y_numeric - 3  
y_numeric / 3
```

Clearly, using the object name, we greatly simplify our code and avoid the redundancy.

Note that R object names are **case-sensitive**. For example, you have defined `x_numeric`, but if you type `X_numeric`, you will get an error message as follow.

```
X_numeric
```

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

### 2.1.2 Review objects in environment

After creating the objects `x_numeric` and `y_numeric`, they also appear in the **Environment**, located in the top right 2.2 panel (**panel3 in Figure 1.3**). You can check all the named objects and their values in this area. It is helpful to monitor the environment from time to time to make sure everything looks fine.

You can also see the list of all the objects you have defined using function `ls()`.

```
ls()
```

```
#> [1] "Code"      "d"          "Explanation" "name"       "Name"
#> [6] "norm_dat"  "norm_dat_1" "norm_dat_2"  "norm_dat_3" "Operation"
#> [11] "Pattern"   "Section"    "x"           "x_numeric"  "y_numeric"
```

### 2.1.3 Object naming rule

Now you have created two objects named `x_numeric` and `y_numeric`. In general, R is very flexible in the name you give to an object however, there are three important rules you need to follow.

**a. Must start with a letter or . (period)**

If starting with period, the second character can't be a number.

**b. Can only contain letters, numbers, \_ (underscore), and . (period)**

One recommended naming style is to only use lower case letters and numbers, and use underscore to separate words within a name. So you can use relatively longer names that is more readable.

**c. Can not use special keywords as names.** For example, `TRUE <- 12` is not permitted as `TRUE` is a special keyword in R. You can see from the following that this assignment operation leads to an error message.

```
TRUE <- 12
```

```
#> Error in TRUE <- 12: invalid (do_set) left-hand side to assignment
```

Some commonly used keywords that cannot be used as names are listed as below.

break	NA
else	NaN
FALSE	next
for	repeat
function	return
if	TRUE
Inf	while

### 2.1.4 Object types

In this section, you have learned about how to assign a value to a name. The objects we used are of *numeric* type. In addition, an object may contain more than

## 2.2. NUMERIC VECTOR, CHARACTER VECTOR, & LOGICAL VECTOR<sup>23</sup>

one values. Also it can also be of other types than *numeric*, including *character*, *logical* and the combination of different types. Depending on the structure, the object belongs to one particular type. We will give a comprehensive treatment to the following object types in this chapter.

Type	Section
Vector	\@ref(vector)
Matrix	\@ref(matrix)
Array	\@ref(array)
Data Frame	\@ref(dataframe)
List	\@ref(list)

While some of the object types look more intuitive than others, you have nothing to worry about since we have this whole chapter devoted to the details of R objects. Objects are the building blocks of R programming and it will be time well spent mastering every object type.

### 2.1.5 Exercise

You can run the following code to do the exercise.

```
r02pro(2.1)
```

## 2.2 Numeric Vector, Character Vector, & Logical Vector

In the last section, you have had a basic understanding of R objects and how to do object assignments. From this section, we will start to introduce different types of R objects one by one. The first R object we want to introduce is called **vector**. **Vector** is the simplest object type in R, which contains one or more values of the **same type**. We will introduce numeric vector, character vector, and logical vector in this section. Let's begin with numeric vector.

### 2.2.1 Numeric vector

#### a. Create numeric vectors

A **numeric vector** is a type of vector that only contains *numbers*. For example, 6 is a numeric vector with length 1. After assigning 6 to the name **x1**, you can refer to **x1** in the following calculations.

```
6                                #a vector with length 1
x1 <- 6                          #x1 is a named vector with length 1
x1                              #check the value of x1
```

But can a numeric vector contain more than one numbers? The answer is a big YES! In R, you can use the `c()` function (`c` is short for combine) to combine values into a vector.

```
c(1, 3, 3, 5, 5)                #use c() to combine values into a vector with length 5
y1 <- c(1, 3, 3, 5, 5)          #y1 is a named vector with length 5
y1                              #check the value of y1
length(y1)                     #length of a vector
```

In this example, you have created a length-5 vector using the `c()` function with arguments containing the five numbers separated by comma. You can also assign the values to the name `y1`. You can verify the contents of `y1` and check the length of it through the `length()` function.

If you write several numeric vectors in `c()`, you will also create a numeric vector with more than one numbers. For example, you can create a numeric vector with values from two numeric vectors.

```
c(c(1,2), c(3,4))               #use c() to combine several numeric vectors into a numeric vector
z1 <- c(c(1,2), c(3,4))
z1
length(z1)
```

After creating numeric vectors, you can use the function `class()` to check its type,

```
class(x1)
class(y1)
class(z1)
```

From the results, you will know that `x1`, `y1` and `z1` are both of numeric type, which is the reason why they are called *numeric vectors*.

### ***b. Operations between two vectors***

Since numeric vectors are made of numbers, you can do **arithmetic operations** between them, just like the fancy calculator in Section 1.2. If two vectors are of the **same length**, the calculation is done elementwisely. In other words, R will



## 2.2. NUMERIC VECTOR, CHARACTER VECTOR, & LOGICAL VECTOR<sup>25</sup>

perform the operation separately for each element. First, let's create another vector of length 1 and do addition with `x1`.

```
x2 <- 3
x1 + x2
```

```
#> [1] 9
```

Then obviously you will get 9!

Similarly, you can create another vector `y2` of the same length as vector `y1`. Then, you can do operations between `y1` and `y2`.

```
y2 <- c(2, 4, 1, 3, 2)
y1 + y2
```

```
#> [1] 3 7 4 8 7
```

The result is yet another length-5 vector. To check the calculation was indeed done elementwisely, you can verify that the first element is  $1 + 2 = 3$ , and the second element is  $3 + 4 = 7$ , etc.

Since the calculation is done elementwisely, we normally would want the two vectors to have the *same length*. However, there is a **recycling** rule in R, which is sometimes quite useful and enables us to write simpler code. Specifically, if one vector is shorter than the other vector, R will *recycle* (repeat) the shorter vector until it matches in length with the longer one. This recycling is particularly helpful for an operator between a length>1 vector and a length-1 vector. Let's see an example.

```
y1 + x1
```

```
#> [1] 7 9 9 11 11
```

From the result, you can see that each element in `y1` is added by 6. The following are a few additional examples you can try.

```
y1 * x2
y1 / 5
y2 - x1
```

## 2.2.2 Character vector character/letter

### a. Create character vectors

Now, let's move to character vectors. In a **character vector**, each value is a **string**. A **string** is a sequence of characters (letters, numbers, or symbols) surrounded by a pair of double quotes (") or single quotes (' '). To be consistent, we will stick with double quotes in this book. Let's first create a character vector with just one string. You can then check the value of this vector by typing its name and verify the vector type by using `class()`.

```
sheepstudio <- "sheep@007"  
sheepstudio  
class(sheepstudio)
```



Double quotes need to be paired in strings. If you miss the right double quote, R will show a plus on the next line, waiting for you to finish the command. If this happens, you can either enter the matching double quote, or press ESC to escape this command.

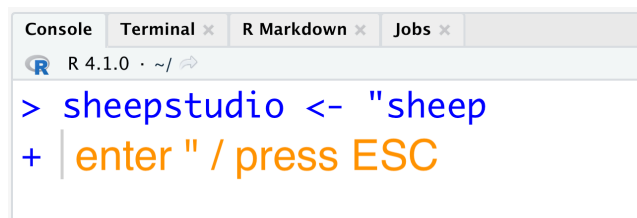


Figure 2.2: Miss the right quotation mark

Similar to a numeric vector, you can use the `c()` function to combine several strings to create a character vector. You can verify the number of strings in the character vector by using `length()`, and `nchar()` can help you get the number of characters in each string.

```
animals <- c("sheep@29", "pig$29", "monkey")  
animals  
length(animals)  
nchar(animals)
```

Note that if you have a vector consisted of numbers with surrounding double quotes, it is also a character vector.

## 2.2. NUMERIC VECTOR, CHARACTER VECTOR, & LOGICAL VECTOR27

```
num_vec <- c(4, 29)
char_vec <- c("4", "29")
class(num_vec)
```

```
#> [1] "numeric"
```

```
class(char_vec)
```

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

### *b. Concatenate several strings into a single string*

Next, we will introduce how to concatenate several strings into a single string. To do this, you can use the `paste()` function. First, let's create a character vector with four strings,

```
four_strings <- c("This", "is", "Sheep@29", "$Studio")
length(four_strings) #verify the number of strings
```

Then use `paste()` instead of `c()`,

```
one_long_string <- paste("This", "is", "Sheep@29", "$Studio")
one_long_string
class(one_long_string)
length(one_long_string) #verify the number of strings
```

From the results, you can see that `one_long_string` is a character vector with length 1, and the value of `one_long_string` is a single string with space between the individual strings.

You may notice that the default separator between the individual strings is space. Actually you can change the the separator by setting the `sep` argument in `paste()`. For example, you can separate the strings with comma,

```
comma <- paste("This", "is", "Sheep@29", "$Studio", sep = ",")
comma
```

```
#> [1] "This,is,Sheep@29,$Studio"
```

If you don't want to use a separator, you can use the `paste0()` function.

```
nosep <- paste0("This", "is", "Sheep@29", "$Studio")
nosep
```

```
#> [1] "ThisisSheep@29$Studio"
```

### c. *Change case*

In character vectors, each string can contain both uppercase and lowercase letters. You can unify the cases of all letters inside a vector. To convert all letters to lower case, you can use the `tolower()` function,

```
tolower(four_strings)
```

```
#> [1] "this"      "is"        "sheep@29" "$studio"
```

The opposite function of `tolower()` is `toupper()`, which converts all letters to upper case,

```
toupper(four_strings)
```

```
#> [1] "THIS"      "IS"        "SHEEP@29" "$STUDIO"
```

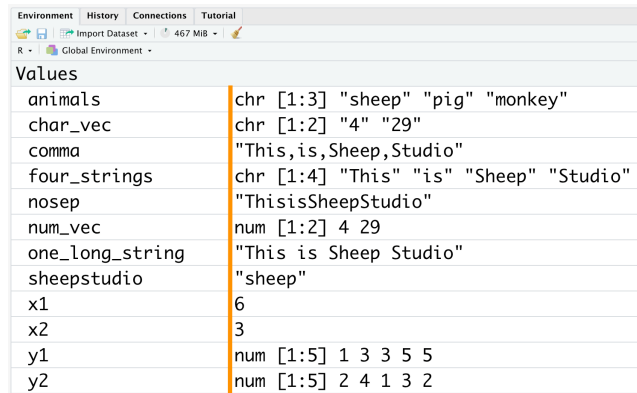
## 2.2.3 Logical vector

So far we have created several numeric vectors and character vectors. You can see all the objects you have defined by using the `ls()` function.

```
ls()
```

```
#> [1] "animals"      "char_vec"      "Code"          "comma"
#> [5] "d"            "Explanation"   "four_strings"  "key_mat"
#> [9] "Keys"         "name"          "Name"          "norm_dat"
#> [13] "norm_dat_1"   "norm_dat_2"    "norm_dat_3"    "nosep"
#> [17] "num_vec"      "one_long_string" "Operation"      "Pattern"
#> [21] "Section"      "sheepstudio"   "Type"          "x"
#> [25] "x_numeric"    "x1"            "x2"            "y_numeric"
#> [29] "y1"           "y2"            "z1"
```

## 2.2. NUMERIC VECTOR, CHARACTER VECTOR, & LOGICAL VECTOR29



The screenshot shows the R Environment panel with the following objects and values:

Object Name	Value
animals	chr [1:3] "sheep" "pig" "monkey"
char_vec	chr [1:2] "4" "29"
comma	"This,is,Sheep,Studio"
four_strings	chr [1:4] "This" "is" "Sheep" "Studio"
nosep	"ThisisSheepStudio"
num_vec	num [1:2] 4 29
one_long_string	"This is Sheep Studio"
sheepstudio	"sheep"
x1	6
x2	3
y1	num [1:5] 1 3 3 5 5
y2	num [1:5] 2 4 1 3 2

Figure 2.3: Environment

As introduced in Section 2.1, another way to check the named objects is via the environment panel as shown in Figure 2.3.

We can see that the environment panel has two columns, with the first column showing the list of object names, and the second column showing the corresponding information for each object. The information includes the vector type (*chr* is short for character and *num* is short for numeric), the vector length, and the first few values of the vector. Note that if the vector is of length 1 (for example *x1*), the environment will not show the type or the length.

Before introducing the *logical vector*, let's first learn a function called `is.numeric()`, which checks whether an object is of numeric type,

```
is.numeric(y1) #Is y1 of numeric type?
```

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

Similar to `is.numeric()`, you can also use `is.character()` function to check if the given object is of character type.

```
is.character(y1) #Is y1 of character type?
```

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

You may notice that results are `TRUE` or `FALSE` from the above codes. Actually, **logical vectors** are vectors that only use `TRUE` or `FALSE` as values. Note that `TRUE` and `FALSE` are logical constants in R. Similarly, you can use `is.logical()` to check if the argument is of logical type, or you can use `class()` to find out the exact type.

```
logic1 <- c(TRUE, FALSE, TRUE) #you can also use the c() function to create a logical  
is.logical(logic1)  
class(logic1)
```

You can also use T to represent TRUE and F to represent FALSE in logical vectors.

```
logic2 <- c(T, F, F)  
is.logical(logic2)  
class(logic2)
```

It is worth to point out that you don't want to put a pair of double quotes around TRUE or FALSE when you use them as logical values. If you do that, a character vector will be generated instead.

```
char <- c("TRUE", "FALSE", "TRUE")  
is.logical(char)  
class(char)
```

Note that the keywords TRUE and FALSE are case sensitive, and all letters inside them need to be in **upper case**. If you change any letter to the lower case, you will get an error, because True is neither a logical constant nor a defined object.

```
tlogic <- True
```

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

### 2.2.4 The coercion rule

So far, we have been considering vectors that have values of the same type, namely, numbers, strings, or logical values. In practice, if we have values with a mix of different types in a vector, R will *unify* all values into the most complex one, which is usually called the **coercion rule**. Specifically, R use the following order of complexity (from simple to complex).

$$\text{logical} < \text{numeric} < \text{character}$$

Let's see a few examples about the coercion. The first example mixes logical values with numbers.

## 2.2. NUMERIC VECTOR, CHARACTER VECTOR, & LOGICAL VECTOR31

```
mix_1 <- c(TRUE, 7, 24, FALSE)
mix_1
```

```
#> [1] 1 7 24 0
```

```
class(mix_1)
```

```
#> [1] "numeric"
```

We can see that `TRUE` will be converted to 1 and `FALSE` will be converted to 0 when they appear with numbers.

The second example mixes numbers with strings.

```
mix_2 <- c(8, "happy", 26, "string")
mix_2
```

```
#> [1] "8"      "happy"  "26"     "string"
```

```
class(mix_2)
```

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

We can see both 8 and 26 are converted into strings.

The final example mixes logical values, numbers and strings.

```
mix_3 <- c(97, TRUE, "pig")
mix_3
```

```
#> [1] "97"    "TRUE"  "pig"
```

```
class(mix_3)
```

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

We can see that both 97 and `TRUE` are converted to strings.

### 2.2.5 Exercise

You can run the following code to do the exercise.

```
r02pro(2.2)
```

## 2.3 Create Vectors with Patterns

Now you are familiar with numeric vector, character vector and logical vector, and you can create them from scratch using the `c()` function. However, in many applications, we may want to create vectors with certain patterns. In this section, we will introduce several commonly used functions for generating vectors with patterns.

### 2.3.1 Create equally-spaced numeric vectors via :

One of the commonly used patterns associated with numeric vectors is numeric vectors composed of **equally-spaced** integers, where the differences between adjacent values in the vectors are all 1 or  $-1$ .

Suppose we want to create a vector with consecutive integers from 1 to 5. The first method is to write all numbers down in `c()`,

```
pattern1 <- c(1,2,3,4,5)
```

You can see that it is not too cumbersome to enumerate all 5 integers when creating `pattern1`. Let's imagine if we want to create a vector containing 100 consecutive integers. Do we have a faster way than writing all 100 integers down? The answer is Yes!

You can use the **colon operator** `:`, which is frequently used in everyday programming. (Note that you don't need to use `c()` with `:`)

```
pattern2 <- 1:5 #consecutive integers from 1 to 5
```

In addition to creating vectors with consecutive integers that is increasing, `:` can also be used to create vectors with integers in decreasing sequences.

```
pattern3 <- 6:2 #decreasing sequence from 6 to 2  
pattern4 <- 3:-3 #decreasing sequence from 3 to -3
```



Powerful the `:` operator is, it can only generate equally-spaced numeric vectors with increment 1 or -1. If you want to generate numeric vectors with different increments, you can use the more powerful `seq()` function.

### 2.3.2 Create equally-spaced numeric vectors via `seq()`

A very efficient way to create **equally-spaced** numeric vectors is to use the `seq()` function, which is short for sequence.

#### a. Create sequences with *by* argument

To use the `seq()` function, you can specify the start value of the sequence in the `from` argument, the limit end value in the `to` argument, and the increment in the `by` argument.

```
seq(from = 1, to = 5, by = 1)
```

Here, the vector starts with 1, increases by 1 at each step, and ends at 5. Note that the `from` and `by` arguments are optional in `seq()`. If you don't specify their values, `seq()` will use the default value 1 for both arguments.

```
seq(to = 5)
```



Now you have had four methods to create vectors with consecutive integers.

```
(1,2,3,4,5,6)           #write all numbers down
1 6                     #use colon operator
  (from = 1, to = 6, by = 1) #use seq()
  (to = 6)                #use seq()
```

Next, let's change the increment to 2 and you will get a numeric vector with 1, 3 and 5 as its values.

```
seq(from = 1, to = 5, by = 2)
```

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

If you only change the limit end value to 6, you still get the same sequence, since the next value in the sequence would be 7 which is larger than the limit end value 6.

```
seq(from = 1, to = 6, by = 2)
```

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

Unlike `:`, you can set values of three arguments in `seq()` as decimal numbers. Here, you will get a sequence starting with 1.1, each element in this sequence will be added by 0.7 on the basis of the previous element, until the biggest number smaller than 6.2.

```
seq(from = 1.1, to = 6.2, by = 0.7)
```

```
#> [1] 1.1 1.8 2.5 3.2 3.9 4.6 5.3 6.0
```

You can also create a decreasing sequence by using a smaller `to` value than the `from` value, coupled with a negative value in the `by` argument.

```
seq(from = 1.5, to = -1, by = -0.5)
```

If a positive value is used in the `by` argument in a decreasing sequence, you will see an error message.

```
seq(from = 1.5, to = -1, by = 0.5)
```

```
#> Error in seq.default(from = 1.5, to = -1, by = 0.5): wrong sign in 'by' argument
```

### *b. Create sequences with `length.out` argument*

Instead of setting the increment, you can also specify the `length.out` argument, which creates a sequence with equal space in the specified length. R will automatically calculate the interval between two neighboring numbers according to values of three arguments in `seq()`.

```
seq(from = 1, to = 5, length.out = 9)
```

Here, you will get a sequence of length 9 from 1 to 5.

You can also create a decreasing sequence by using the `length.out` argument.

```
seq(from = 5, to = -5, length.out = 9)
```



Unlike creating sequences with `by` argument, if you specify the `length.out` argument in `seq()`, the start value and end value of the sequence you get will be exactly match the input arguments.

### *c. Create sequences with both `by` and `length.out` arguments*

Lastly, if you provide both the `by` and `length.out` arguments, only one of `from` and `to` is needed. With one value (the start value or the limit end value) fixed, `seq()` will create a vector with specified increment and length.

If you only have the `from` argument, you will get a sequence starting from the value you set with the increment in the `by` argument, until you get a sequence with specified length.

```
seq(from = 1, by = 2, length.out = 5)
```

If you only have the `to` argument, you will get a sequence end with the value you set with the increment in the `by` argument, until you get a sequence with specified length.

```
seq(to = 1, by = 2, length.out = 5)
```

One last thing regarding `seq()` is that you can *at most* provide three arguments. For example, you will see an error when running the following example since all four arguments are specified.

```
seq(from = 1, to = 3, by = 1, length.out = 3)
```

```
#> Error in seq.default(from = 1, to = 3, by = 1, length.out = 3): too many arguments
```

### 2.3.3 Create a matching numeric vectors via `seq_along()`

Now, we introduce one function related to `seq()`. Let's first create a numeric vector,

```
extend <- seq(from = 2, to = 8, length.out = 9)
```

From the `seq()` above, you know that the length of this vector is 9. Next, let's put this numeric vector in `seq_along()`.

```
seq_along(extend)
```

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

`seq_along()` takes a vector as its argument, and generates consecutive integers from 1 to the length of the input vector. The `seq_along()` function is commonly used when writing loops, which will be covered at a later time.



You can also use `1:length(extend)` to get the same result as `seq_along(extend)`.

```
1          (extend)
```

### 2.3.4 Create numeric vectors via `sequence()`

Sometimes, you may want to combine multiple equally-spaced integer sequences into a single vector. To do this, you can use the function `sequence()`. The most common usage of `sequence()` is to supply a vector of integers as its input.

```
comp_seq1 <- sequence(c(2, 3, 5))
comp_seq1
```

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

From the result, we can see that it firstly create equally-spaced vectors 1:2, 1:3, and 1:5, then combine all vectors into a single one. This avoids the trouble of writing something like `c(1:2, 1:3, 1:5)`.

More generally, we can construct a vector of more complex integer sequences with additional arguments, namely `sequence(nvec, from, by)`. Here, the `nvec`, `from` and `by` are integer vectors of the corresponding `length.out`, `from`, and `by` arguments of each equally-spaced sequence. Let's see the following example.

```
comp_seq2 <- sequence(nvec = c(4, 3, 2), from = c(1, 2, -1), by = c(2, -1, 2))
comp_seq2
```

```
#> [1] 1 3 5 7 2 1 0 -1 1
```

Now, `sequence()` generate three different equally-spaced integer sequences and combine them to a single vector. We can reproduce the vector using `c()` and three calls of the `seq()` function.

```
comp_seq3 <- c(seq(from = 1, by = 2, length.out = 4),
               seq(from = 2, by = -1, length.out = 3),
               seq(from = -1, by = 2, length.out = 2))
comp_seq3
```

```
#> [1] 1 3 5 7 2 1 0 -1 1
```

### 2.3.5 Create numeric, character and logical vectors with repetition

Another commonly used pattern associated with vectors is **repetition**. Note that while the equally-spaced pattern only makes sense for numeric vectors, the repetition pattern work for all three kinds of vectors.

To do repetition, you can use the `rep()` function, which works by repeating the first argument for the number of times indicated in the second argument.

Firstly, let's create a numeric vector with repetition.

```
num1 <- rep(2, 4)
num1
```

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

Since the first argument is 2 and the second argument is 4, 2 is repeated for 4 times, resulting a length-4 vector with all elements 2.

The first argument can also be a numeric vector with several values.

```
num2 <- rep(c(1, 4, 2), 3)
num2
```

```
#> [1] 1 4 2 1 4 2 1 4 2
```

Here, then `rep()` will repeat the whole vector `c(1, 4, 2)` three times. Note that the vector is **repeated as a whole**, not elementwisely.

You may be wondering what happens the second argument also has several numbers? Let's try together.

```
num3 <- rep(c(1,5,7), c(3,2,1))
num3
```

```
#> [1] 1 1 1 5 5 7
```

When the second argument is also a vector, R will do an **element repeat** operation by repeating each element in the first argument the number of times indicated in the corresponding location of the second argument, and combine the repeated vectors to a single vector. In this example, 1 is repeated 3 times, 5 is repeated twice, and 7 is repeated once. It is equivalent to

```
c(rep(1,3), rep(5,2), rep(7,1))
```

The `rep()` function works the same way if the first argument is a character vector.

```
animals1 <- rep(c("sheep", "pig", "monkey"), 2)
animals1
animals2 <- rep(c("sheep", "pig", "monkey"), c(3, 2, 1))
animals2
```

You can also use logical vectors in the first argument.

```
logic <- rep(c(TRUE, FALSE), c(3,2))
logic
```

### 2.3.6 Getting unique elements and their frequencies

So far, you have learned how to create vectors with different patterns. Sometimes, you may want to get the unique elements of a vector and their corresponding frequencies. Let's use `num3` as an example. **(Don't forget to use `ls()` or check the environment panel to find all objects you have defined),**

```
num3          #check the values
```

```
#> [1] 1 1 1 5 5 7
```

You can use `unique()` to show all unique elements in vectors

```
unique(num3)   #get the unique elements
```

```
#> [1] 1 5 7
```

From the result, you know the unique elements in `num3` are 1,5, and 7. To get the frequency of each element, you can use the `table()` function.

```
table(num3)    #get the frequency table
```

```
#> num3  
#> 1 5 7  
#> 3 2 1
```

Here, the first row is the name of the object, the second row shows all unique elements, and the third row is the corresponding frequency of each element in the same column. In `num3`, there are three 1s, two 5s and one 7.

`unique()` and `table()` work similarly for character vectors and logical vectors. You can try the following codes.

```
animals  
unique(animals)  
table(animals)  
logic  
unique(logic)  
table(logic)
```

### 2.3.7 Exercise

You can run the following code to do the exercise.

```
r02pro(2.3)
```

## 2.4 Sort, Rank, & Order

In the past two sections, you have mastered how to create vectors of different types including numeric, character and logical. In addition, you know how to create vectors with patterns. A vector usually contains more than one elements. Sometimes, you want to order the elements in various ways. In this section, we will introduce important functions that relate to ordering elements in a vector.

### 2.4.1 Numeric vectors

Let's start with numeric vectors. Firstly, let's create a numeric vector which will be used throughout this section.

```
x <- c(2, 3, 2, 0, 4, 7)
x #check the value of x
```

#### *a. Sort vectors*

The first function we will introduce is `sort()`. By default, the `sort()` function **sorts** elements in vector in the ascending order, namely from the smallest to largest.

```
sort(x)
```

```
#> [1] 0 2 2 3 4 7
```

If you want to sort the vector in the descending order, namely from the largest to smallest, you can set a second argument `decreasing = TRUE`.

```
sort(x, decreasing = TRUE)
```

```
#> [1] 7 4 3 2 2 0
```



### *b. Ranks of vectors*

Next, let's talk about ranks. The `rank()` function gives the **ranks** for each element of the vector, namely the corresponding positions in the **ascending order**.

```
rank(x)
```

```
#> [1] 2.5 4.0 2.5 1.0 5.0 6.0
```

If you check the value of `x`, you can see that the smallest value of `x` is 0, which corresponds to the fourth element. Thus, the fourth element has rank 1. The second smallest value of `x` is 2, which is shared at the first and the third elements, resulting a **tie** (elements with the same value will result in a tie). Normally, these two elements would have ranks 2 and 3. To break the tie, the `rank()` function assigns all the elements involving in the tie (the first and third elements in this example) the same rank, which is average of all their ranks (the average of 2 and 3), by default. In addition to this default behavior for handling ties, `rank()` also provides other options by setting the `ties.method` argument.

If you set `ties.method = "min"`, all the tied elements will have the *minimum rank* instead of the average rank. In this case, the minimum rank is 2.

```
rank(x, ties.method = "min")
```

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

If you want to break the ties by the order it appears in the vector, you can set `ties.method = "first"`. Then the earlier appearing elements will have smaller ranks than the later ones. In this example, the first element will have rank 2 and the third element has rank 3, since the first element appears earlier than the third element. There are other options for handling ties, which you can look up in the documentaion of `rank()` if interested.

```
rank(x, ties.method = "first")
```

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



Unlike `sort()`, you can't get positions in the descending order from the `rank()` function, which means you can't add `decreasing = TRUE` in `rank()`.

### c. Order of vectors

The next item we want to introduce is the `order()` function. Note that the function name `order` could be a bit misleading since ordering elements also has the same meaning of sorting. However, although it is related to sorting, it is a very *different* function from `sort()`.

Let's recall the values of `x` and apply `order()` to `x`.

```
x
```

```
#> [1] 2 3 2 0 4 7
```

```
order(x)
```

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

From the result, you can see that the `order()` function returns **indices** for the elements in the ascending order, namely from the smallest to the largest. For example, the first output is 4, indicating the 4th element in `x` is the smallest. The second output is 1, showing the 1st element in `x` is the second smallest.



Unlike `rank()`, the `order()` function breaks the ties by the appearing order by default.

If you want the indices corresponding to the descending order, then you can set `decreasing = TRUE` just like what we did in the `sort()` function.

```
order(x, decreasing = TRUE)
```

So far, we have covered `sort()`, `rank()` and `order()` functions for numeric vectors. It is helpful to provide a brief summary.

- The `sort()` function sorts elements in vectors.
- The `rank()` function will give ranks for each element of the vector.
- The `order()` function returns indices for the elements.

## 2.4.2 Character vectors

Now, let's move to character vectors. For character vectors, R uses the **lexicographical ordering**, which is sometimes called dictionary order since it is the order used in a dictionary. Similar to numeric vectors, let's first prepare a character vector. Note that the strings in character vectors can contain letters, numbers, or symbols.

```
char_vec <- c("a", "A", "B", "b", "ab", "aC", "1c", ".a", "1a", "2a", ".a", "&u", "3", "_4")
```

### a. Ordering rules

First, let's discuss the ordering of a single character, including symbols, digits and letters. There are a few important ordering rules as follows.

- symbols < digits < letters: symbols appears first, followed by digits, and letters come last.
- symbols are ordered in the following way.

```
syms <- c(" ", ",", ";", "_", "(", ")", "!", "[", "]", "{", "}", "-", "*", "/", "#", "$", "%", "^", "&", "~", "@", "+",
sort(syms)
```

```
#> [1] " " "_" "-" ", " "; " !" "?" "." "(" ")" "[" "]" "{" "}" "@" "*" "/" "&" "#"
#> [20] "%" "`" "^" "+" "<" "=" ">" "|" "$"
```

- digits are in an ascending order: the smaller digits appear earlier than the bigger ones.

```
nums <- 0:9
sort(nums)
```

```
#> [1] 0 1 2 3 4 5 6 7 8 9
```

- letters are alphabetically ordered: for the same letter the lower case comes first  $a < A < b < B < \dots < z < Z$ .

```
lower_letters <- letters
upper_letters <- LETTERS
all_letters <- c(lower_letters, upper_letters)
sort(all_letters)
```

```
#> [1] "a" "A" "b" "B" "c" "C" "d" "D" "e" "E" "f" "F" "g" "G" "h" "H" "i" "I" "j"
#> [20] "J" "k" "K" "l" "L" "m" "M" "n" "N" "o" "O" "p" "P" "q" "Q" "r" "R" "s" "S"
#> [39] "t" "T" "u" "U" "v" "V" "w" "W" "x" "X" "y" "Y" "z" "Z"
```

You can combine symbols, numbers and letters into a single vector and get the order.

```
all_chars <- c(syms, nums, all_letters)
sort(all_chars)
```

### *b. Sort vectors*

As before, you can apply `sort()` on character vectors. Basically, it orders by the first character, moves to the second character if there are ties in the first character (same first character), and look at more characters until the ties are broken or run out of characters.

```
sort(char_vec)
```

```
#> [1] "_4" ".a" ".a" "&u" "1a" "1c" "2a" "3" "a" "A" "ab" "aC" "b" "B"
```

We have the following observations.

- According to the ordering rule of symbols, `_4` is the first, `.a` should be the second and `&u` is the third.
- `1a` and `1c` have the same first character, since `a` comes before `c`, `1a` comes before `1c`.
- `ab` and `aC` have the same first character, since `b` comes before `C` (regardless of the case), `ab` comes before `aC`.

Of course, we can also have the order reversed by setting `decreasing = TRUE`.

```
sort(char_vec, decreasing = TRUE)
```

### *c. Ranks of vectors*

Similarly, you can look at the ranks for each element according to the ordering rules. Here, the element with rank 1 is `_4` and `.a` has rank 2. Just like numeric vectors, if you have elements with the same value in character vectors, the rank of these elements will be the same (the average of the corresponding ranks) by default.

```
rank(char_vec)
```

```
#> [1]  9.0 10.0 14.0 13.0 11.0 12.0  6.0  2.5  5.0  7.0  2.5  4.0  8.0  1.0
```

As expected, you can set the `ties.method` argument in `rank()` to use other methods for breaking ties.

```
rank(char_vec, ties.method = "min")  
rank(char_vec, ties.method = "first")
```

#### *d. Order of vectors*

Again, you can get the order of each element in character vectors with the same `order()` function like that for numeric vectors.

```
order(char_vec)
```

```
#> [1] 14  8 11 12  9  7 10 13  1  2  5  6  4  3
```

The `decreasing` argument still works for `order()`!

```
order(char_vec, decreasing = TRUE)
```

```
#> [1]  3  4  6  5  2  1 13 10  7  9 12  8 11 14
```

### 2.4.3 Logical vectors

Since there are only two possible values `TRUE` and `FALSE` for logical vectors, it is straightforward to sort them with the knowledge of `FALSE < TRUE`. You can try the following example.

```
logi_vec <- c(TRUE, FALSE, FALSE, TRUE, TRUE)  
sort(logi_vec)  
rank(logi_vec)  
order(logi_vec)
```

### 2.4.4 Exercise

You can run the following code to do the exercise.

```
r02pro(2)
```

## 2.5 Statistical functions on vectors

In this section, we will continue talking about functions on vectors, and focus on various summary statistics.

### 2.5.1 Numeric vectors

Speaking of statistics, statistics on numeric vectors. Let's first create a numeric vector which will be used throughout this section( )

```
h <- c(3, 2, 75, 0, 0, 100)
h #check the value of h
```

Next, we will divide statistical functions into several groups, and we will introduce groups one by one.

#### ***Group A***

```
min(h)
max(h)
range(h)
```

First, you can get the minimum value and maximum value of a numeric vector, and `range()` produces a length-2 vector with both the minimum (the left element) and maximum (the right element).

#### ***Group B***

```
which.min(h)
which.max(h)
```

After getting the minimum value and the maximum value, you can get the location of the minimum value, if there are multiple elements with the minimum

value, `which.min()` returns the first location. Here, the fourth element and the fifth element in `h` both have the minimum value, but you will get the result of 4 from `which.min()`. Similarly, `which.max()` tells you the location of the maximum value.

### *Group C*

```
cummin(h)
cummax(h)
```

In addition to the standard minimum, we also have the cumulative minimum function, called `cummin()`. It returns a vector of the same length as the input vector, with the value at each location being the minimum of all preceding elements until that location in the original vector. For example, the first element is 3 since the minimum of the first element is always itself, the second element is 1 since the minimum of the first two elements 3 and 2 is 2, and so on. Note that once we reaches the minimum value of the vector, the remaining cumulative minimum will remain the minimum value. There is also a corresponding function for computing the cumulative maximum, called `cummax()`.

### *Group D*

```
sum(h)
cumsum(h)
```

Next, let's look at the `sum()` function, which produces the sum of all elements of the vector. For the numeric vector `h`, the summation is  $3+2+75+0+0+100$ , which is 180. Similarly, if you want to have the cumulative summation, you can use `cumsum()`, which works by summing up the elements cumulatively up to each location. For example, the first element is 3 since there is only one element to do summation, the second element is 5 since the summation of the first two elements 3 and 2 is 5, you can check the value of each element by yourself.

### *Group E*

```
prod(h)
cumprod(h)
```

We also have the `prod()` function, computing the product of all elements of `h`. Since there is 0 in `h`, the result of all elements of `h` will be 0. Again, we have the cumulative product function working by producting the elements cumulatively up to each location.

### *Group F*

```
sort(h)
```

```
#> [1] 0 0 2 3 75 100
```

Before introducing this group, let's first review the `sort()` function. By default, this function can sort elements from smallest to largest. We have introduced it in ...

```
mean(h)  
median(h)
```

Then in this group, we have the `mean()` function, which returns the average of all elements. And the `median()` function returns the middle number when all elements are listed in order from smallest to largest. (get ordered elements from `order()`) If the vector length is odd, the middle number is the value of the middle element after sorting, for example, the median corresponds to the 3rd element for vectors with 5 elements. If the vector length is even, the middle number is the average of two middle elements after sorting, take `h` for example, after sorting, you will see that 2 and 3 are in middle, the median is then defined as the average of these two elements, equaling 2.5.

*Group G*



Operation	Explanation
<code>min(h)</code>	get the minimum value
<code>max(h)</code>	get the maximum value
<code>range(h)</code>	get both the minimum value and the maximum value
<code>which.min(h)</code>	get the location of the minimum value
<code>which.max(h)</code>	get the location of the maximum value
<code>cummin(h)</code>	get the minimum value of all preceding elements until each location
<code>cummax(h)</code>	get the maximum value of all preceding elements until each location
<code>sum(h)</code>	get the sum of all elements
<code>cumsum(h)</code>	sum up the elements cumulatively up to each location
<code>prod(h)</code>	get the product of all elements
<code>cumprod(h)</code>	product cumulatively up to each location
<code>mean(h)</code>	get the average of all elements
<code>median(h)</code>	get the middle number when all elements are listed in order from smallest to largest
<code>quantile(h, probs = 0.95)</code>	
<code>IQR(h)</code>	
<code>summary(h)</code>	
<code>var(h)</code>	
<code>sum((h-mean(h))^2)/(length(h)-1)</code>	
<code>sd(h)</code>	
<code>sqrt(var(h))</code>	
<code>lag(h)</code>	
<code>lag(h, 2)</code>	
<code>diff(h)</code>	
<code>diff(h, lag = 2)</code>	



Unlike `sort()`, you can't get positions in the descending order from the `rank()` function, which means you can't add `decreasing = TRUE` in `rank()`.

### 2.5.2 Character vectors

### 2.5.3 Logical vectors

### 2.5.4 Exercise

You can run the following code to do the exercise.

```
r02pro(2)
```



## Chapter 3

# Data Visualization

### 3.1 Introduction to Datasets

Before entering the colorful world of data visualization, let's first introduce the data set we will be using throughout this chapter. The data set is a part of the Ames Housing Price data, containing 165 observations and 12 features including the sale date and price.

The dataset **sahp** is located in the R package **r02pro**, the companion package of this book. Besides the **r02pro** package, we will also extensively use the **tidyverse** package for visualization in this chapter. First, let's load these two packages.

```
library(r02pro)
library(tidyverse)
```

```
#> -- Attaching packages ----- tidyverse 1.3.0 --

#> v ggplot2 3.3.2      v purrr   0.3.4
#> v tibble  3.0.4      v dplyr   1.0.2
#> v tidyr   1.1.2      v stringr 1.4.0
#> v readr   1.4.0      v forcats 0.5.0

#> -- Conflicts ----- tidyverse_conflicts() --
#> x dplyr::filter() masks stats::filter()
#> x dplyr::lag()    masks stats::lag()
```

After loading the two packages, you can type **sahp** to have a quick look of the dataset.

```
sahp
```

```
#> # A tibble: 165 x 12
#>   dt_sold   bedroom bathroom gar_car oa_qual liv_area lot_area house_style
#>   <date>     <dbl>    <dbl>  <dbl>  <dbl>   <dbl>   <dbl>  <chr>
#> 1 2010-03-25     3      2.5     2      6    1479    13517 2Story
#> 2 2009-04-10     4      3.5     2      7    2122    11492 2Story
#> 3 2010-01-15     3      2      1      5    1057     7922 1Story
#> 4 2010-04-19     3      2.5     2      5    1444     9802 2Story
#> 5 2010-03-22     3      2      2      6    1445    14235 1.5Fin
#> 6 2010-06-06     2      2.5     2      6    1888    16492 1Story
#> 7 2006-06-14     2      3      2      6    1072     3675 SFoyer
#> 8 2010-05-08     3      2      2      5    1188    12160 1Story
#> 9 2007-06-14     2      1      1      5     924    15783 1Story
#> 10 2007-09-01     5      2.5     2      5    2080    11606 2Story
#> # ... with 155 more rows, and 4 more variables: kit_qual <chr>,
#> #   heat_qual <chr>, central_air <chr>, sale_price <dbl>
```

You can see that **sahp** is a *tibble* with 165 observations and 12 variables. By default, the output only shows the first 10 observations in the tibble along with the first few variables that can fit the window. To view the full dataset, you can use the `view()` function, which will open the dataset in a new window.

```
view(sahp)
```

To view the top rows of the dataset, you can use the `head()` function, which produces the first 6 observations by default. You can also set an optional second argument to pick any given number of observations.

```
head(sahp)
head(sahp, 15)
```

To get a first impression on the dataset, you can use the `summary()` function introduced in **Section??**.

```
summary(sahp)
```

In the output, we get the summary statistics for each variable. For numeric variables, we get the minimum, 1st quartile, median, mean, 3rd quartile, and the maximum. It also shows the number of NAs for a particular variable. For

character variables, we only get the length of the vector, the class, and the mode.

Although the types of each variable are shown in the result when typing `sahp`, a more detailed list can be found with the function `str()`.

```
str(sahp)
```

```
#> tibble [165 x 12] (S3: tbl_df/tbl/data.frame)
#> $ dt_sold      : Date[1:165], format: "2010-03-25" "2009-04-10" ...
#> $ bedroom      : num [1:165] 3 4 3 3 3 2 2 3 2 5 ...
#> $ bathroom     : num [1:165] 2.5 3.5 2 2.5 2 2.5 3 2 1 2.5 ...
#> $ gar_car       : num [1:165] 2 2 1 2 2 2 2 2 1 2 ...
#> $ oa_qual       : num [1:165] 6 7 5 5 6 6 6 5 5 5 ...
#> $ liv_area      : num [1:165] 1479 2122 1057 1444 1445 ...
#> $ lot_area      : num [1:165] 13517 11492 7922 9802 14235 ...
#> $ house_style   : chr [1:165] "2Story" "2Story" "1Story" "2Story" ...
#> $ kit_qual      : chr [1:165] "Good" "Good" "Good" "Average" ...
#> $ heat_qual     : chr [1:165] "Excellent" "Excellent" "Average" "Good" ...
#> $ central_air   : chr [1:165] "Y" "Y" "Y" "Y" ...
#> $ sale_price    : num [1:165] 130 NA 109 174 138 ...
```

The `str()` function gives a list of each component, the corresponding type, the length, and the first several values.

### 3.1.1 Are two-story houses more expensive than one-story ones?

Let's try to answer this question by doing some analysis. First, let's create the logical vectors corresponding to two-story and one-story houses.

```
story_2 <- sahp$house_style == "2Story"
story_1 <- sahp$house_style == "1Story"
```

Then, we create two vectors containing the prices of the two groups, respectively.

```
sale_price_2 <- sahp$sale_price[story_2]
sale_price_1 <- sahp$sale_price[story_1]
```

Finally, we can run the `summary()` function on both vectors.

```
summary(sale_price_2)
```

```
#>   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.   NA's
#>   55.0   137.9   174.0   197.8   231.5   545.2     1
```

```
summary(sale_price_1)
```

```
#>   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#>   44.0   129.0   160.0   183.0   224.2   465.0
```

From these summaries, it is clear that the corresponding statistic is larger for two-story houses compared with one-story ones, for all 6 measures. As a result, we can draw the conclusion that the two-story houses indeed have a higher sale price than the one-story ones.

### 3.1.2 Converting Data Types

When you import a data set into R, some variables may not have the desired types. In this case, it would be useful to convert them into the types you want before conducting further data analysis.

#### *a. Convert a character vector to an unordered factor*

Let's look at the variable `house_style` in `sahp`. We can see from the output of `str(sahp)` that it is of `chr` type. Let's confirm this and get its summary.

```
is.character(sahp$house_style)
```

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

```
summary(sahp$house_style)
```

```
#>   Length    Class    Mode
#>   165 character character
```

As briefly mentioned before, using the `summary()` function on a character vector doesn't provide us much useful information. Let's find the unique values of this vector and get the frequency table.

```
unique(sahp$house_style)
```

```
#> [1] "2Story" "1Story" "1.5Fin" "SFoyer" "SLvl"
```

```
table(sahp$house_style)
```

```
#>
#> 1.5Fin 1Story 2Story SFoyer  SLvl
#>    21    81    50     5     8
```

We can see that there are five house styles along with their frequencies. It turns out to be particularly useful to convert this type of variable into a *factor* type. Let's use the function `as.factor()` and run the summary function again.

```
sahp$house_style <- factor(sahp$house_style)
summary(sahp$house_style)
```

```
#> 1.5Fin 1Story 2Story SFoyer  SLvl
#>    21    81    50     5     8
```

### ***b. Convert a character vector to an ordered factor***

Now, let's take a look at another variable called `kit_qual`, measuring the kitchen quality. Again, let's check the unique values.

```
unique(sahp$kit_qual)
```

```
#> [1] "Good"      "Average"   "Fair"      "Excellent"
```

In addition to having four different quality values, they have an internal ordering among them. In particular, we know `Fair < Average < Good < Excellent`. To reflect this, you can convert this variable in to an *ordered factor* using the `factor()` function. In particular, the `ordered = TRUE` argument reflects that we want to create an ordered factor.

```
sahp$kit_qual <- factor(sahp$kit_qual, ordered = TRUE, levels = c("Fair", "Average", "Good", "Excellent"))
summary(sahp$kit_qual)
```

```
#>      Fair   Average   Good Excellent
#>         9      85      57         14
```

```
str(sahp$kit_qual)
```

```
#> Ord.factor w/ 4 levels "Fair"<"Average"<...: 3 3 3 2 2 3 2 2 2 1 ...
```

### *c. Convert a character vector to a logical vector*

Lastly, let's look at the variable `central_air`, representing whether the house has central AC or not. As before, let's get the unique elements.

```
unique(sahp$central_air)
```

```
#> [1] "Y" "N"
```

Intuitively, you can create a logical vector representing whether the house has central AC.

```
sahp$central_air <- sahp$central_air == "Y"
summary(sahp$central_air)
str(sahp$central_air)
```

Sometimes, you may also want to create additional variables from the existing ones. For example, we know the overall quality of the house ranges from 2 to 10.

```
table(sahp$oa_qual)
```

Maybe we want to call a house of good quality if `oa_qual` is larger than 5. We can then create a new logical variable as follows.

```
sahp$good_qual <- sahp$oa_qual > 5
```

Finally, let's look at the structure of `sahp` again to confirm the actions we did.

```
str(sahp)
```



## 3.2 Scatterplots

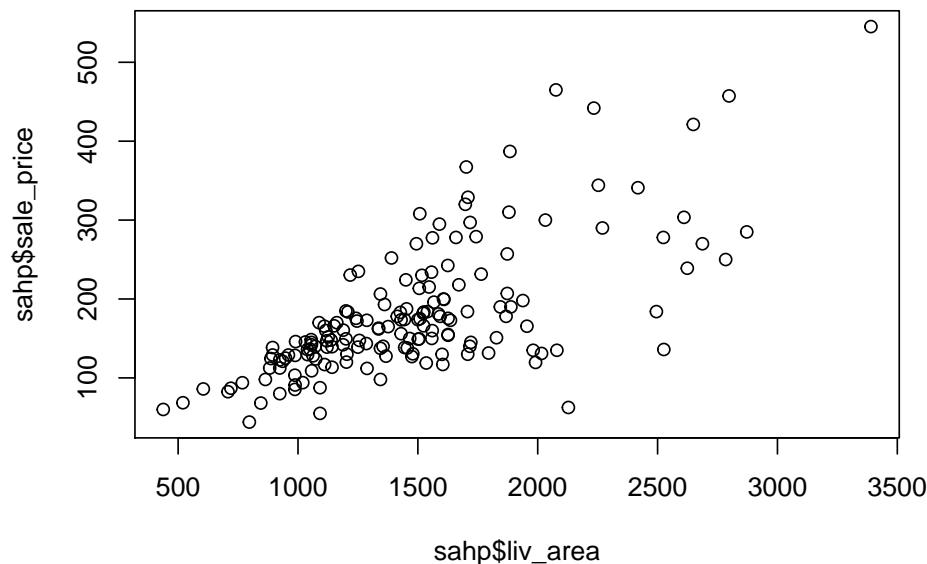
From this section, you will learn various kinds of plots, that involves one or more variables in a data set. Considering the housing prices, a natural question you may have is that are the bigger houses more expensive?

To answer this question, you need to look at the relationship between `liv_area` and the `sale_price` in the `sahp` data set. To visualize the relationship between two continuous variables, the most commonly used plot is the **scatterplot**, which is a 2-dimensional plot with a collection of all the datapoints, where the x-axis and y-axis correspond to the two variables, respectively.

### 3.2.1 Using the `plot()` function

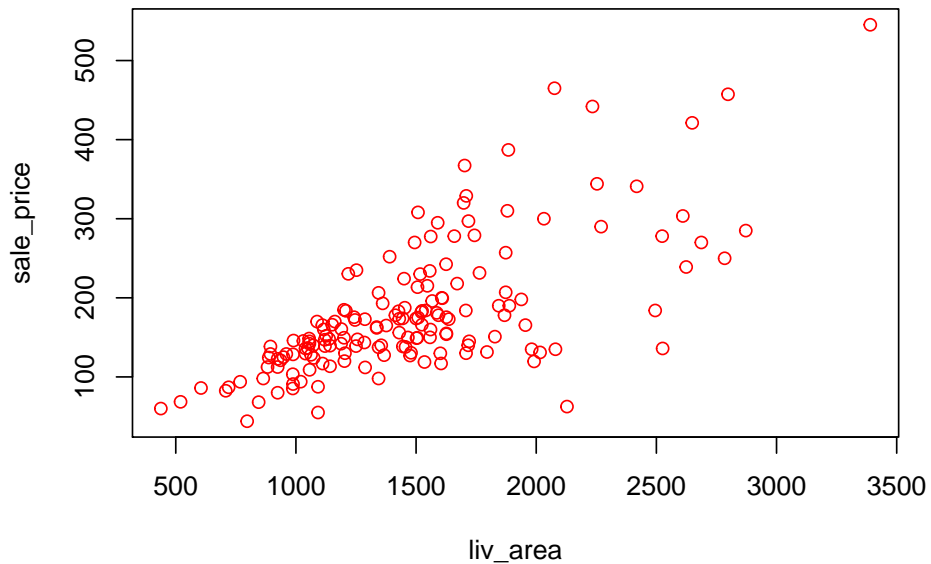
In base R, we can use the `plot()` function to generate this scatterplot with the first argument being the variable on the x-axis and the second argument being the variable on the y-axis.

```
library(r02pro)
plot(sahp$liv_area, sahp$sale_price)
```



From the scatterplot, we can see a clear increasing trend between `sale_price` and `liv_area`, which is consistent with our intuition. The `plot()` function provides a rich capability of customization. For example, we can set the labels on the x-axis and y-axis, change the color of the points, etc.

```
plot(sahp$liv_area, sahp$sale_price, xlab = "liv_area", ylab = "sale_price", col = "red")
```

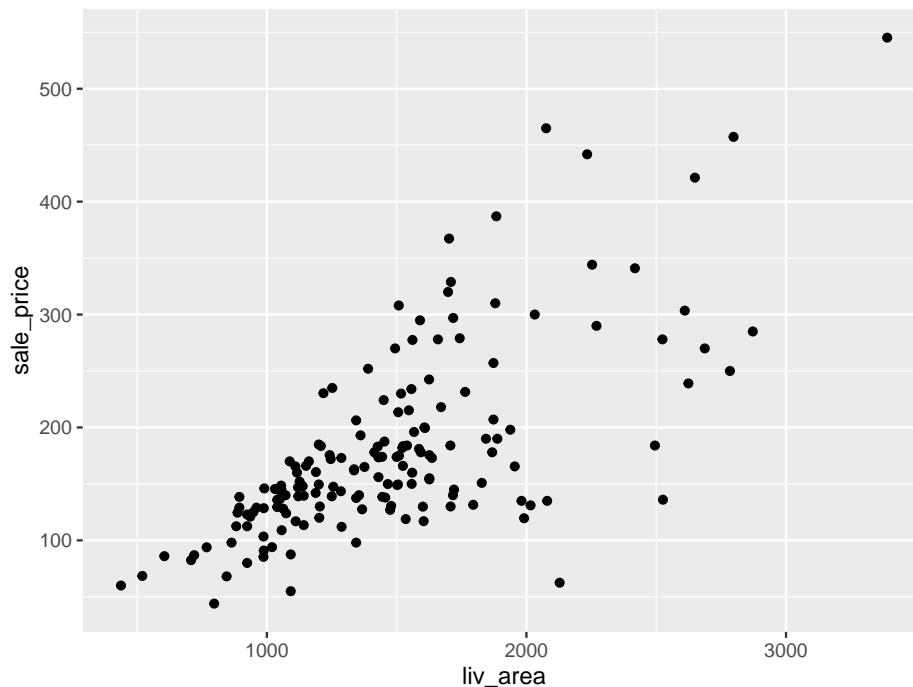


### 3.2.2 Using the ggplot() function

Although the `plot()` function gets the work done, the **ggplot2** package provides a superior user experience which allows us to create complex plots with ease. Since the **ggplot2** package is a member of the **tidyverse** package, you don't need to install it separately if **tidyverse** was already installed. Let's first load the package **ggplot2** and create a scatterplot.

```
library(ggplot2)
ggplot(data = sahp) + geom_point(mapping = aes(x = liv_area, y = sale_price))
```

```
#> Warning: Removed 1 rows containing missing values (geom_point).
```

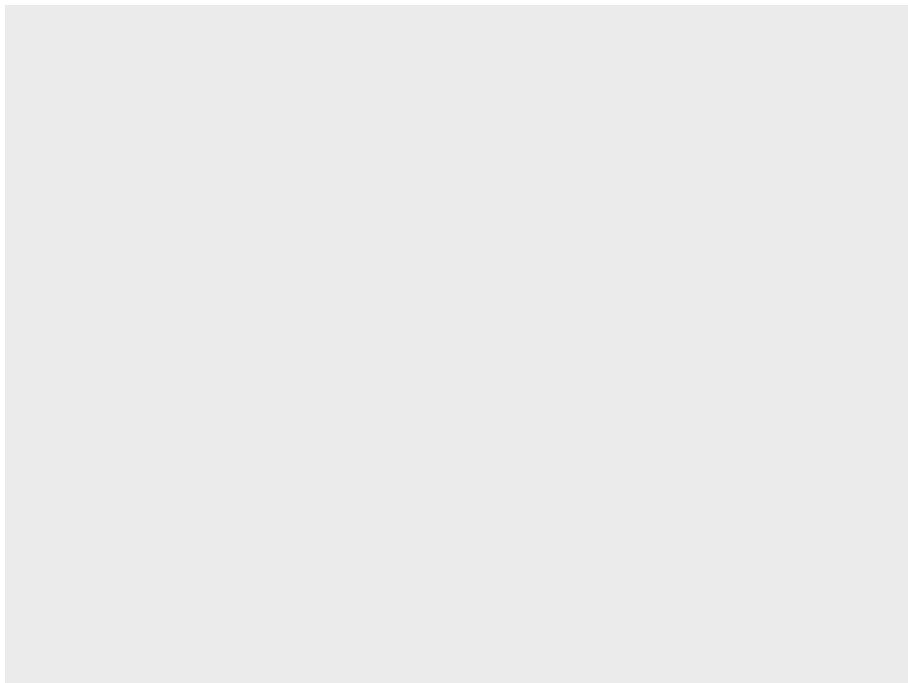


Aside from the expected scatterplot, you can see a warning message “Removed 1 rows containing missing values (geom\_point).” This indicates that there is 1 row in **sahp** that contains missing values and it was removed during the plotting process. The removal of missing values is a default behavior for all plots generated by the **ggplot2** package.

Now, let’s walk through the mechanism of **ggplot2**. In a nutshell, **ggplot2** implements the **grammar of graphics**, a coherent system for describing and building graphs. A more detailed description on the grammar of graphics can be found in Wickham (2010).

Let’s break it down into two steps. In **ggplot2**, we always start with the function **ggplot()** with a data frame or tibble as its argument.

```
ggplot(data = sahp)
```



After running this code, you can see an empty plot. This is because ggplot does not yet know which variables or what type of plots you want to create. To generate a scatterplot, you can use add a layer using the `+` operator followed by the `geom_point()` function. The `geom_point()` is one of the many available geoms in ggplot.

Inside `geom_point()`, you need to set the value of the `mapping` argument. The `mapping` argument takes a functional form as `mapping = aes()`, where the `aes` is short for aesthetics. For example, you can use `aes()` to tell ggplot to use which variable on the x-axis, which variable on the y-axis. Let's take another look at this example.

```
ggplot(data = sahp) + geom_point(mapping = aes(x = liv_area, y = sale_price))
```

Here, inside the `aes()` function, we set `x = liv_area` and `y = sale_price`, indicating that the variable `liv_area` will appear on the x-axis and `sale_price` will appear on the y-axis.

### 3.3 Aesthetics in ggplot

Knowing how to generate a scatterplot using `geom_point()`, let's discuss one of the most important aspects in a `geom`, namely, the aesthetics. Aesthetics

include various parameters that you can change that affect the appearances of a plot. Some commonly used aesthetics include color, size, shape, and so on.

Note that although we will introduce aesthetics via the example of scatterplot, they are used for all kinds of plots which will be covered at a later time.

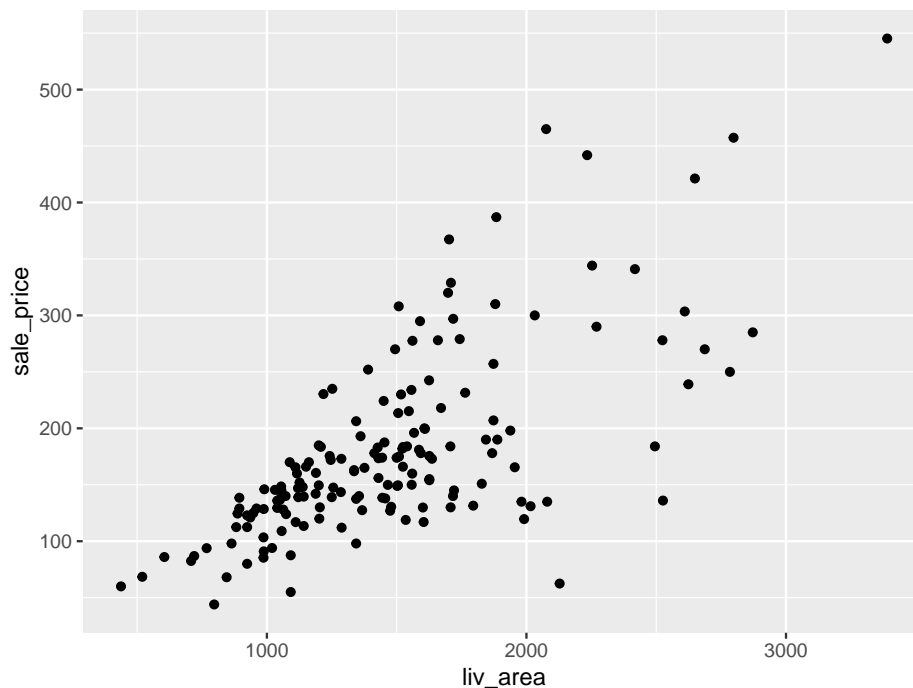
### 3.3.1 Global Aesthetics

First, we discuss **global aesthetics**, which change certain features of a plot globally.

Let's first review the code we used to generate the scatterplot between `liv_area` and `sale_price`.

```
library(ggplot2)
library(r02pro)
ggplot(data = sahp) + geom_point(mapping = aes(x = liv_area, y = sale_price))
```

```
#> Warning: Removed 1 rows containing missing values (geom_point).
```



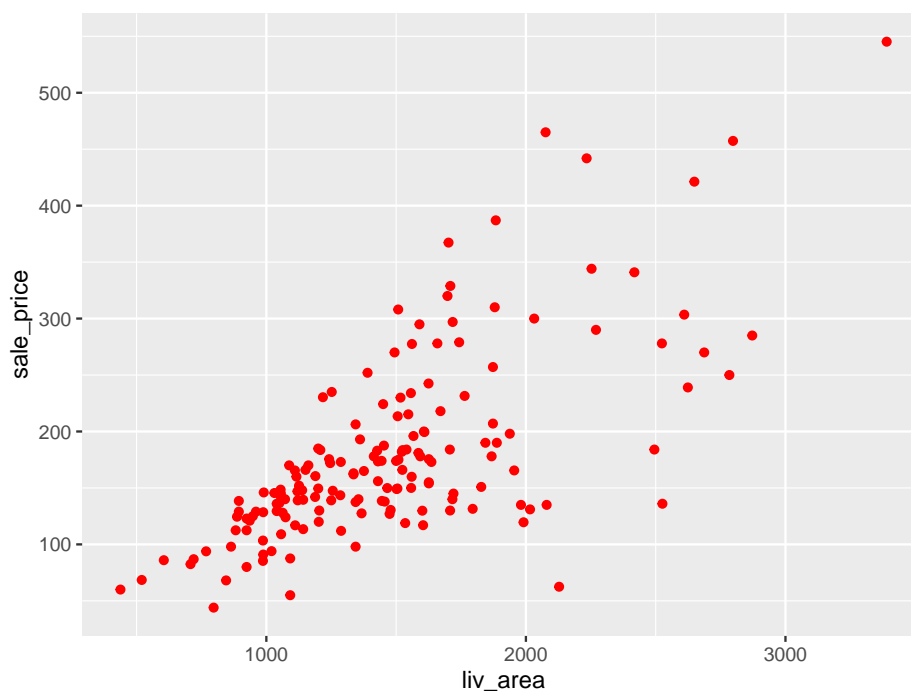
Now, let's see how to set **global aesthetics** in `geom_point()`.

#### a. Color

To change the color of all points, you can set the `color` argument in the `geom_point()` function. Note that it is placed outside of the `aes()` function.

```
ggplot(data = sahp) + geom_point(mapping = aes(x = liv_area, y = sale_price), color = 'red')
```

```
#> Warning: Removed 1 rows containing missing values (geom_point).
```



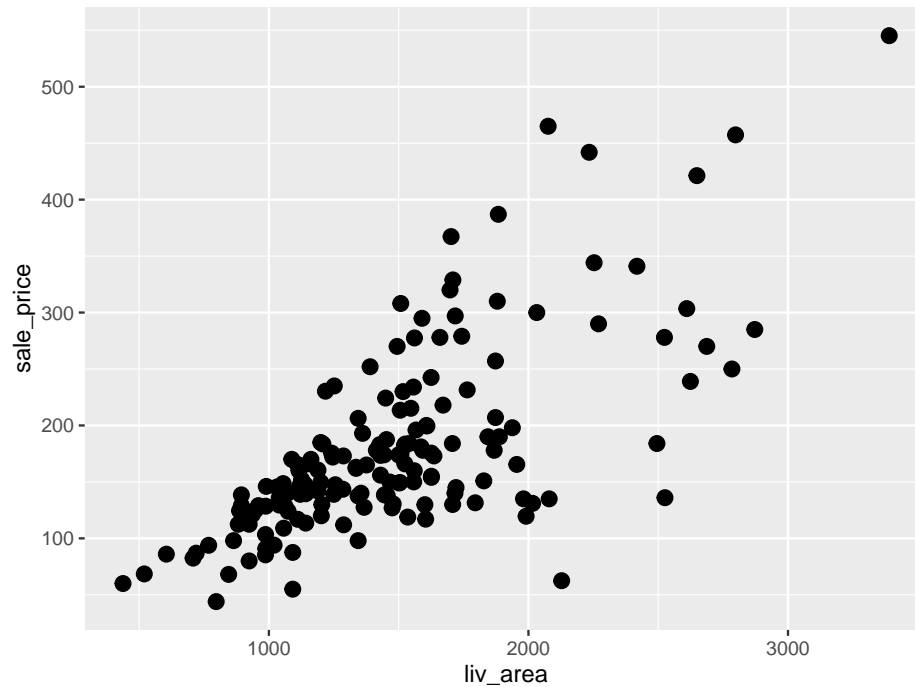
Clearly, all points are changed to red.

### *b. Size*

Similarly, you can set the `size` element in the `geom_point()` function to change the size of the all points.

```
ggplot(data = sahp) + geom_point(mapping = aes(x = liv_area, y = sale_price), size = 3)
```

```
#> Warning: Removed 1 rows containing missing values (geom_point).
```

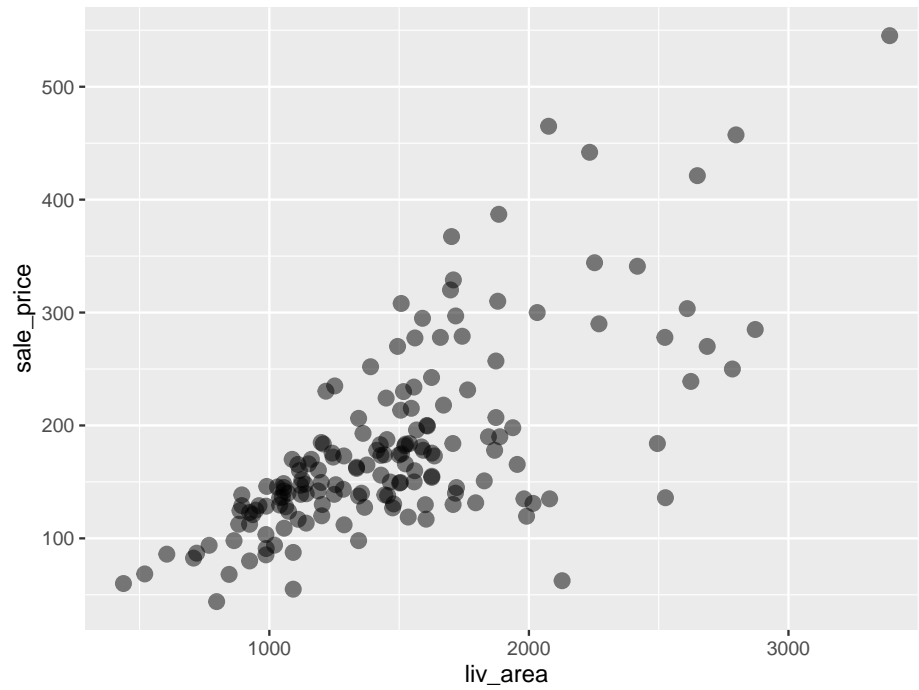


You may notice that the points are now bigger than before. Looking at the plot, many points are overlapping with each other, which is sometimes called **overplotting**. To solve this issue, you can change the transparency level of the points by setting the `alpha` argument.

### c. Transparency

```
ggplot(data = sahp) + geom_point(mapping = aes(x = liv_area, y = sale_price), size = 3, alpha = 0.5)
```

```
#> Warning: Removed 1 rows containing missing values (geom_point).
```



By setting `alpha = 0.5`, the points become more visible and the overplotting problem is largely alleviated.

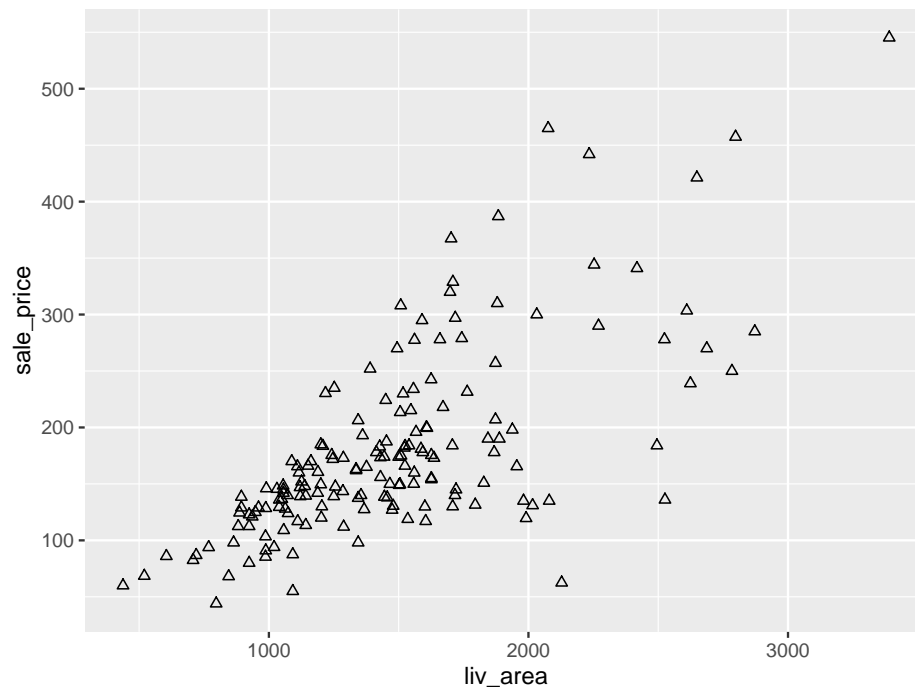
#### *d. Shape*

Lastly, we can also change the shape of the points from the default one (circle) to other shapes by the `shape` argument in `geom_point()`.

```
ggplot(data = sahp) + geom_point(mapping = aes(x = liv_area, y = sale_price), shape = 2)
```

```
#> Warning: Removed 1 rows containing missing values (geom_point).
```



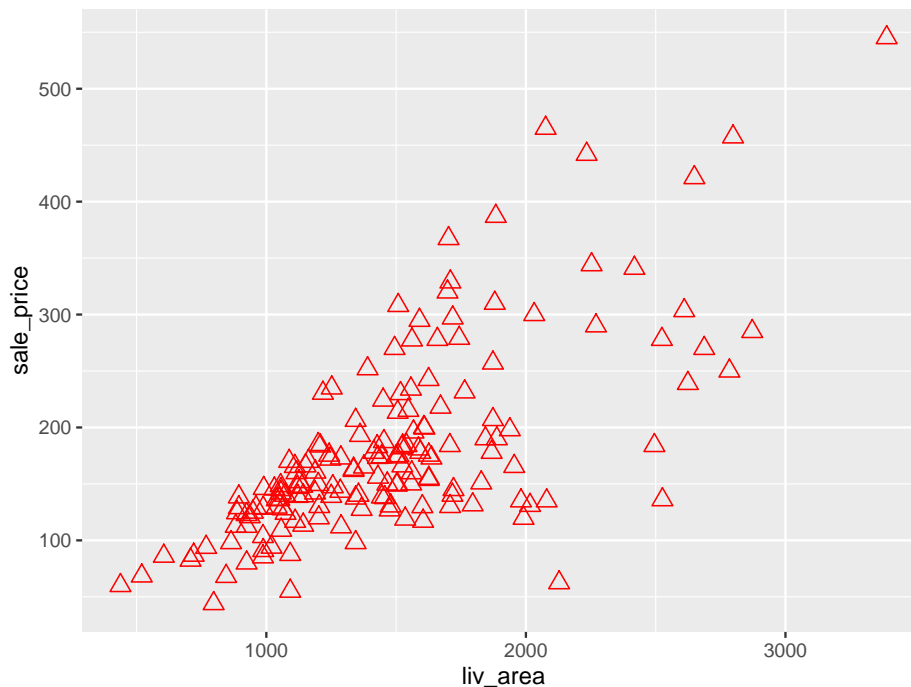


#### *e. Multiple Aesthetics*

Of course, we can combine multiple global aesthetics in the same plot.

```
ggplot(data = sahp) + geom_point(mapping = aes(x = liv_area, y = sale_price), color = "red", size = 100)
```

```
#> Warning: Removed 1 rows containing missing values (geom_point).
```



Here, we have all points red, size of 3, and of triangle shape.

### 3.3.2 Map Discrete Variables to Aesthetics

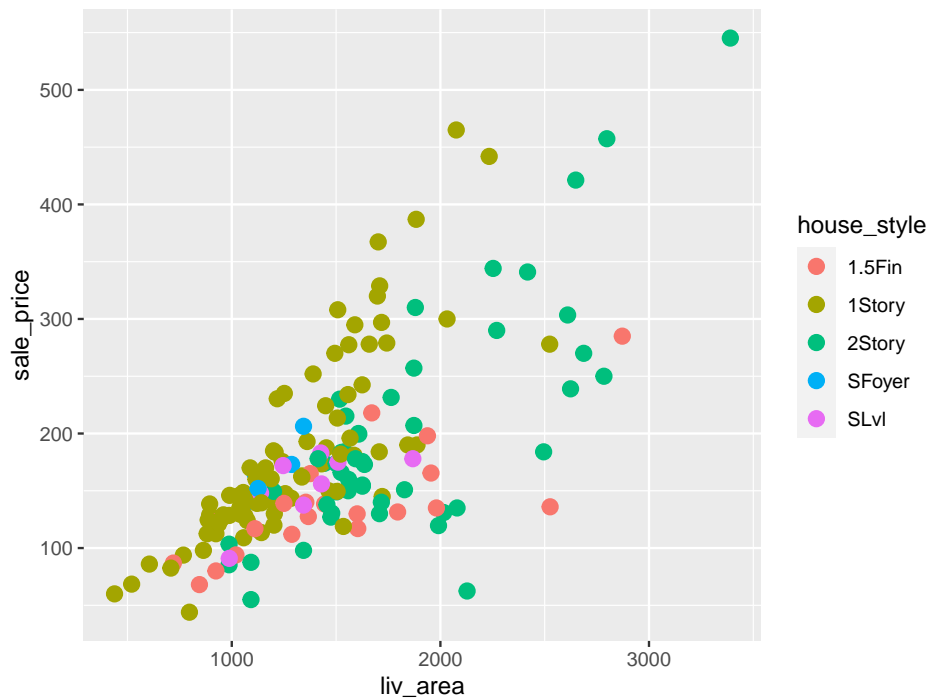
Knowing how to use global aesthetics to change the global appearance of a plot, you may want to differentiate different groups with different values of aesthetics. For example, you want to use different colors according to the different `house_style` in the scatterplot. To do this, you can map a discrete variable (say `house_style`), to an aesthetic (say `color`) by setting `color = house_style` as an argument in the `aes()` function.

#### *a. Color*

Now, let's map `house_style` to `color`.

```
ggplot(data = sahp) + geom_point(mapping = aes(x = liv_area, y = sale_price, color = house_style))
```

```
#> Warning: Removed 1 rows containing missing values (geom_point).
```

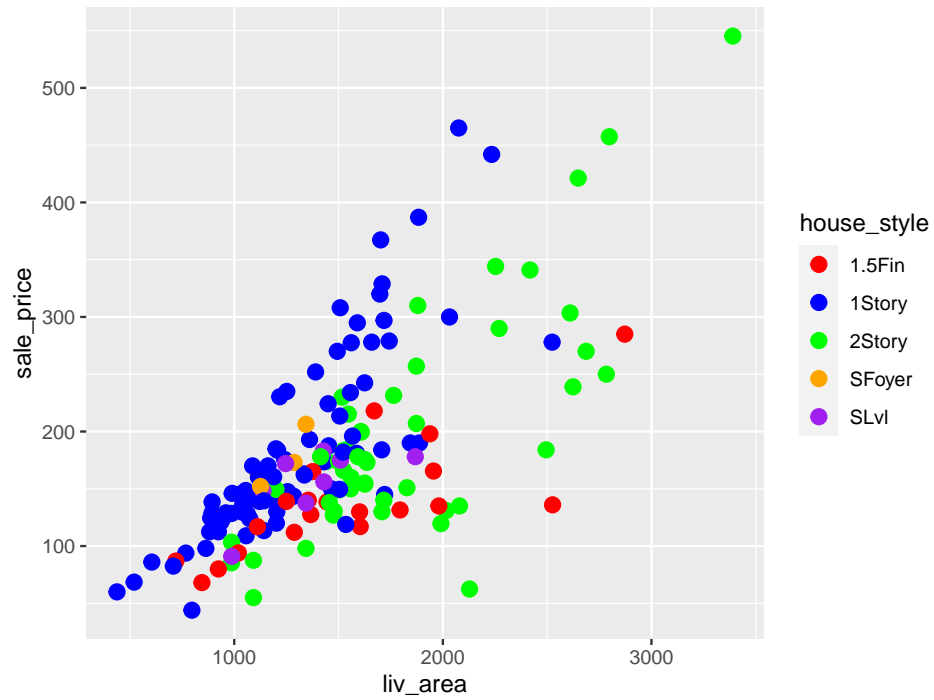


From this figure, we can clearly see houses of different styles in distinct colors. In addition, `ggplot` automatically created a legend to show the correspondence between the house styles and colors.

Sometimes, you may want to use specific colors for different values of the factor. To customize the colors, you can add a layer to the `ggplot` with function `scale_colour_manual` with argument `values` containing a character vector consisting of the colors.

```
ggplot(data = sahp) + geom_point(mapping = aes(x = liv_area, y = sale_price, color = house_style))
```

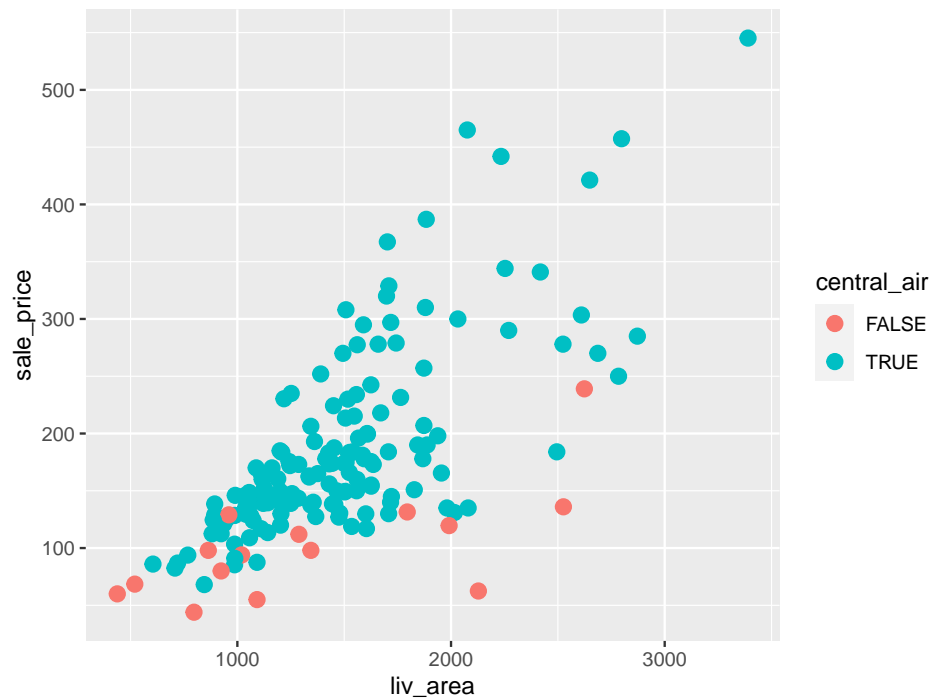
```
#> Warning: Removed 1 rows containing missing values (geom_point).
```



Similarly, you can also map `central_air` to color.

```
ggplot(data = sahp) + geom_point(mapping = aes(x = liv_area, y = sale_price, color = central_air))
```

```
#> Warning: Removed 1 rows containing missing values (geom_point).
```



The plot tells us the majority of the houses have central AC and the ones without it have relatively lower sale price.

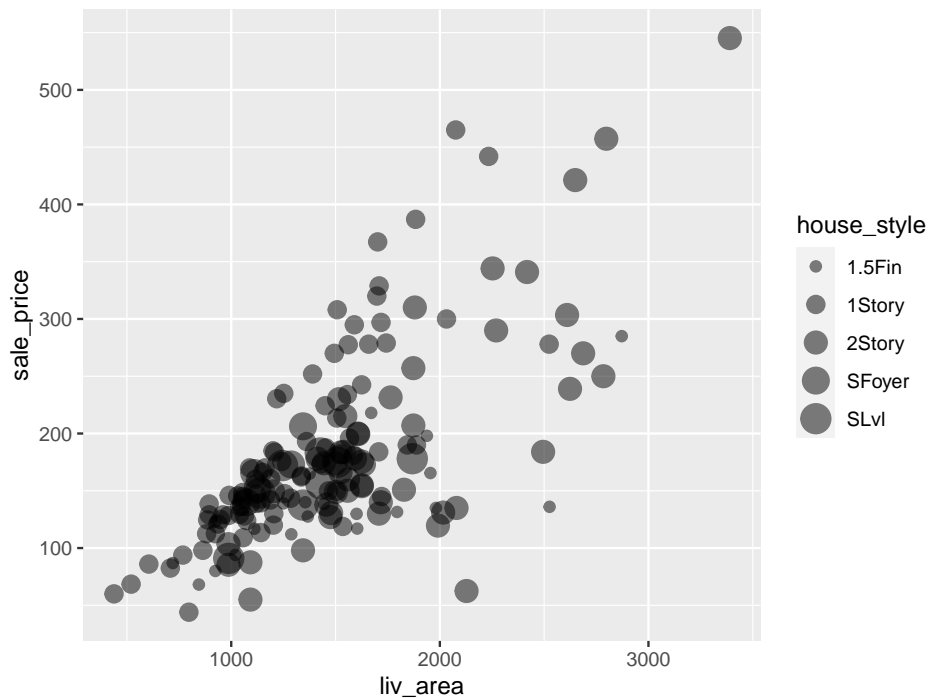
#### *b. Size*

In addition to color, you can also map a discrete variable to the size aesthetic.

```
ggplot(data = sahp) + geom_point(mapping = aes(x = liv_area, y = sale_price, size = house_style),
```

```
#> Warning: Using size for a discrete variable is not advised.
```

```
#> Warning: Removed 1 rows containing missing values (geom_point).
```



You can see from the plot that the sizes of the points are now different according to the `house_style`. To alleviate the overplotting issue, we added a global aesthetic `alpha = 0.5`, making all points more transparent.

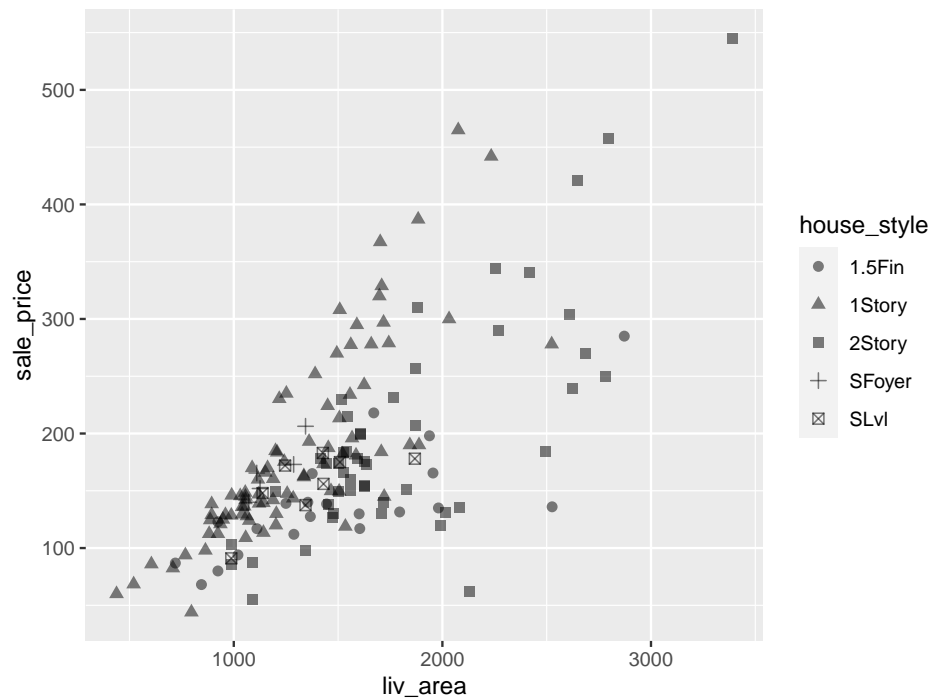
There is a warning message: “Using size for a discrete variable is not advised.” The reason is that different sizes may implicitly indicate a particular ordering of the groups, which are usually not clear for a discrete variable.

### c. Shape

We can also map a discrete variable to the `shape` aesthetic.

```
ggplot(data = sahp) + geom_point(mapping = aes(x = liv_area, y = sale_price, shape = h
```

```
#> Warning: Removed 1 rows containing missing values (geom_point).
```



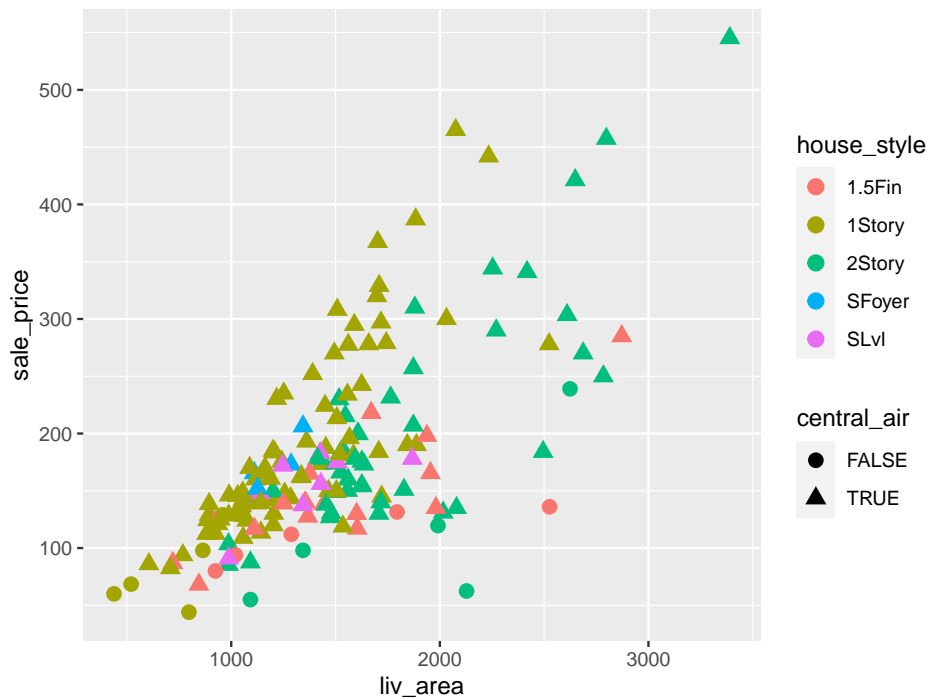
Again, we added global aesthetics `size` and `alpha` to make the points more visible.

#### *d. Multiple mappings*

Just like global aesthetics, you can also have multiple mappings for aesthetics, and mix them with the global aesthetic when necessary.

```
ggplot(data = sahp) + geom_point(mapping = aes(x = liv_area, y = sale_price, color = house_style,
```

```
#> Warning: Removed 1 rows containing missing values (geom_point).
```



Here, we can see the points have different colors according to `house_style` and are of different shapes depending on the value of `central_air`. Note that there are two legends on the plot showing the color and shape, respectively.

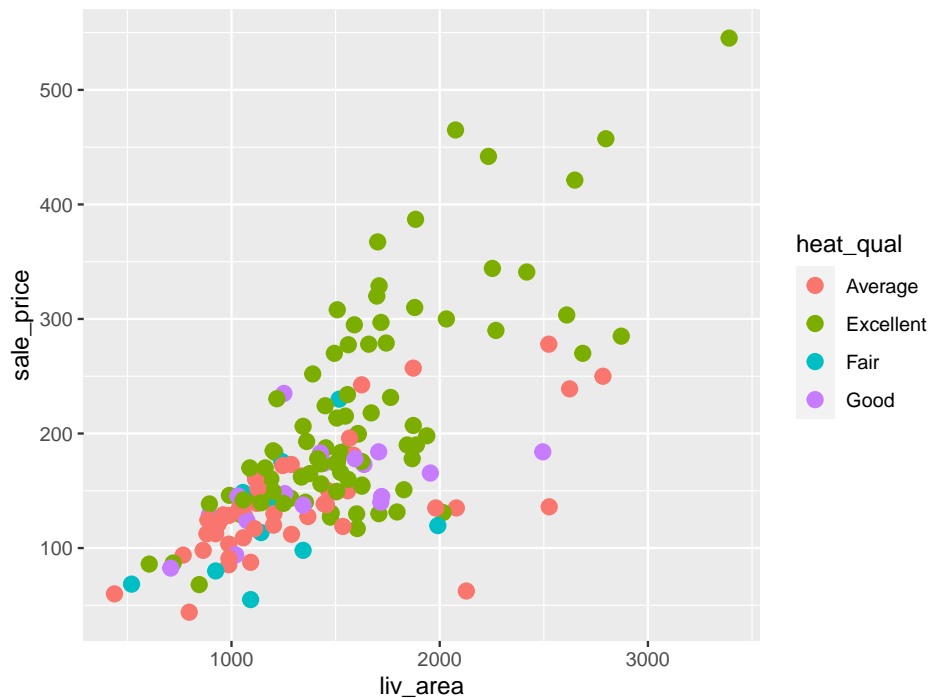
### 3.3.3 Change Legend Order via Factors

Let's first generate a scatterplot between `liv_area` and `sale_price` where we map the `heat_qual` (heating quality) to the `color` aesthetic.

```
ggplot(data = sahp) + geom_point(mapping = aes(x = liv_area, y = sale_price, color = h
```

```
#> Warning: Removed 1 rows containing missing values (geom_point).
```

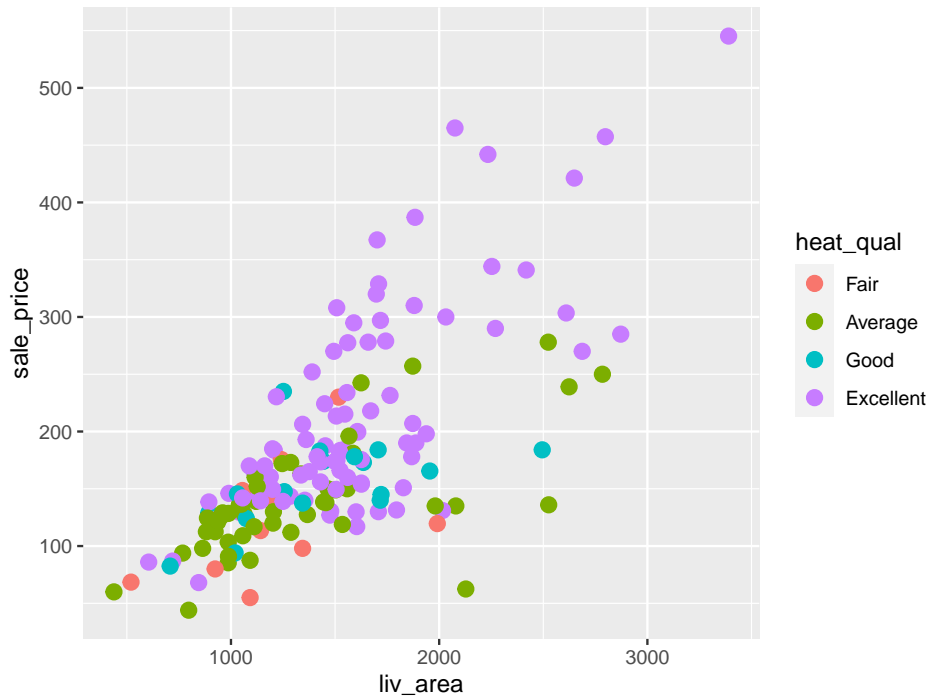




Looking at the legend, you can see that different `heat_qual` values are in alphabetical order as introduced in Section 2.4 when we introduced the ordering of character vectors. Sometimes, you may want to arrange these values in a different order in the plot, for example from the worst to the best. To achieve this, you can use the `factor()` function with the argument `levels` which specifies the desired order.

```
sahp$heat_qual <- factor(sahp$heat_qual, levels = c("Fair", "Average", "Good", "Excellent"))
ggplot(data = sah) + geom_point(mapping = aes(x = liv_area, y = sale_price, color = heat_qual),
```

```
#> Warning: Removed 1 rows containing missing values (geom_point).
```



After changing the `heat_qual` variable to a factor with desired levels, you can see the order in the legend changes accordingly.

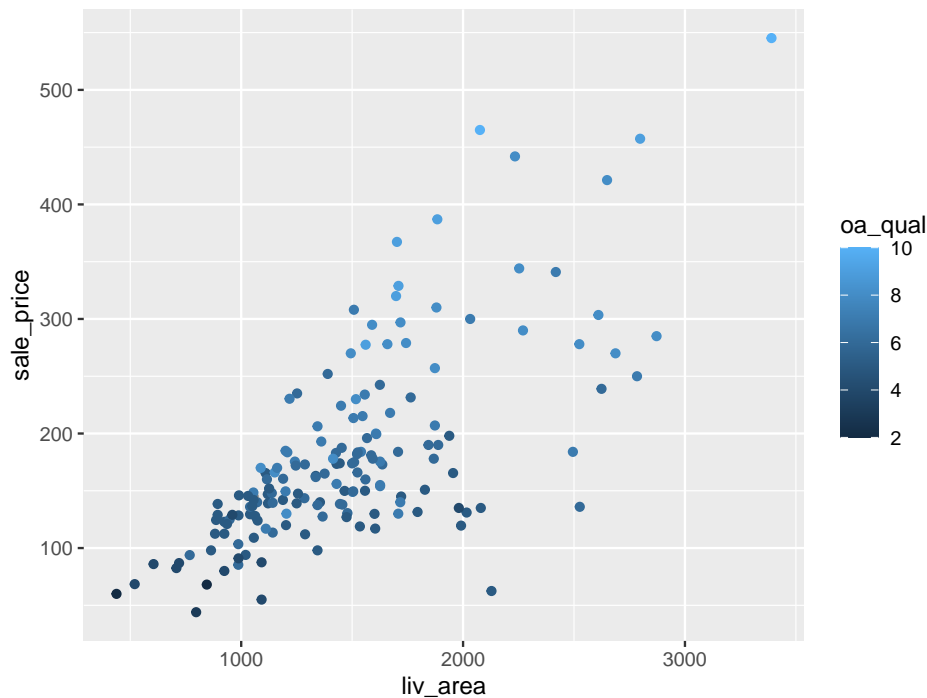
### 3.3.4 Map Continuous Variables to Aesthetics

Knowing how to map discrete variables to aesthetics, it is natural to ask whether we can also map continuous variables to aesthetics. The answer is positive.

*a. Color* Let's again start with the color aesthetic by mapping `oa_qual` to color.

```
ggplot(data = sahp) + geom_point(mapping = aes(x = liv_area, y = sale_price, color = oa_qual))
```

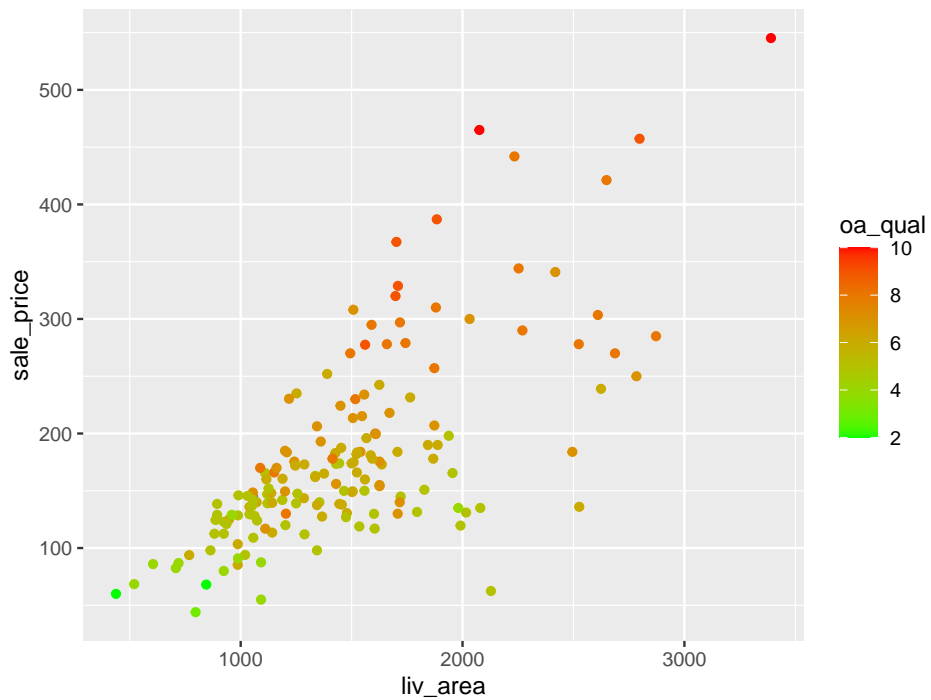
```
#> Warning: Removed 1 rows containing missing values (geom_point).
```



Here, we can see the color of all points vary from dark blue to light blue, depending on the value of `oa_qual`. Instead of showing different colors in the discrete variable case, the legend now displays a bar showing a continuous color scale according to the value of `oa_qual`. To customize the color scale, you can add another layer using the function `scale_color_continuous` with arguments `low` and `high` being two colors corresponding to the colors when the variable is of low and high values, respectively.

```
ggplot(data = sahp) + geom_point(mapping = aes(x = liv_area, y = sale_price, color = oa_qual)) +
```

```
#> Warning: Removed 1 rows containing missing values (geom_point).
```



Here, the low value of `oa_qual` is mapped to green color and the high value of `oa_qual` is mapped to red color. You can also try out the following examples.

```
ggplot(data = sahp) + geom_point(mapping = aes(x = liv_area, y = sale_price, color = oa_qual))
```

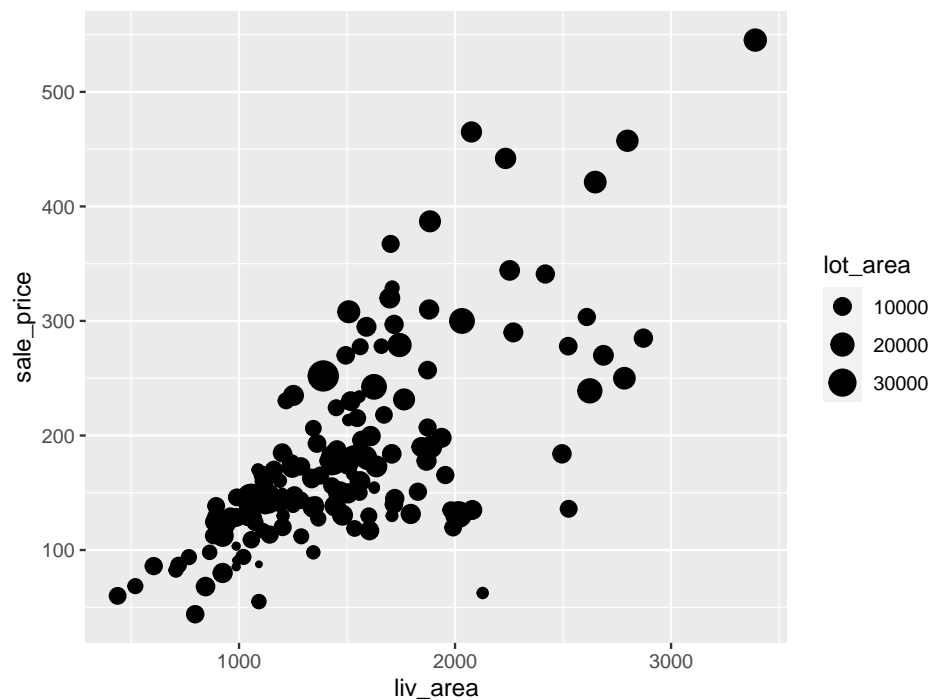
```
ggplot(data = sahp) + geom_point(mapping = aes(x = liv_area, y = sale_price, color = oa_qual))
```

### *b. Size*

In addition to the color aesthetic, we can also map continuous variables to the size aesthetic.

```
ggplot(data = sahp) + geom_point(mapping = aes(x = liv_area, y = sale_price, size = liv_area))
```

```
#> Warning: Removed 1 rows containing missing values (geom_point).
```



In this example, you can see the points corresponding to larger `lot_area` values are larger than those corresponding to smaller `lot_area` values. Note that although the legend only shows three different sizes, the actual size of the point is continuous corresponding to the value of `lot_area`.

How about the shape aesthetic? Can we map a continuous variable to it? Let's try it.

```
ggplot(data = sahp) + geom_point(mapping = aes(x = liv_area, y = sale_price, shape = lot_area)) #
```

```
#> Error: A continuous variable can not be mapped to shape
```

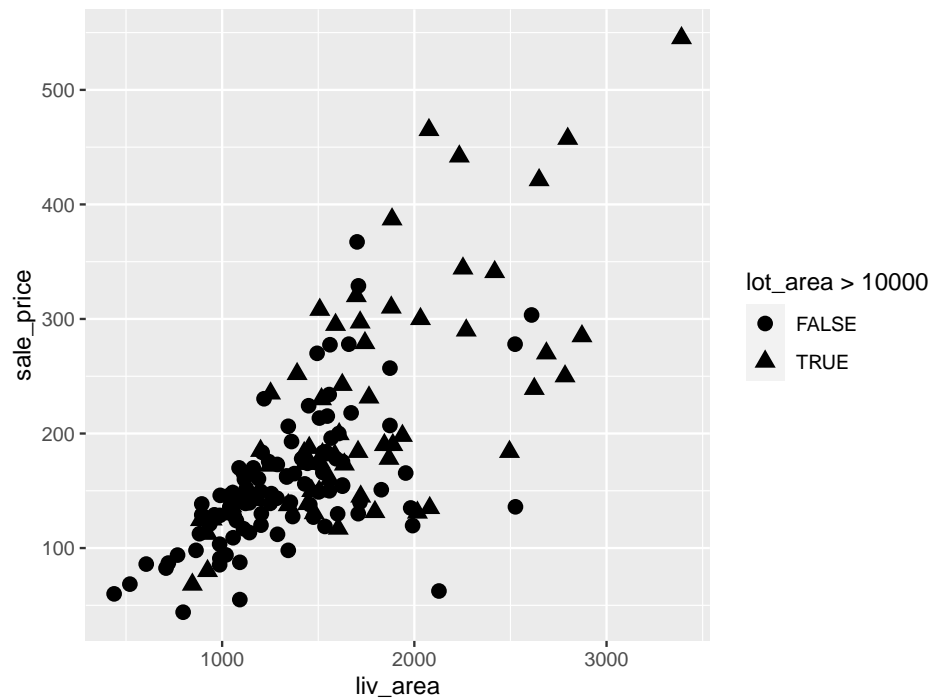
You see an error message: “A continuous variable can not be mapped to shape”. The reason is intuitive: the shape can’t be naturally changed continuously.

### 3.3.5 Map Converted Logical Variable to Aesthetics

Lastly, you can also create logical variables on the fly and map them to aesthetics. For example, if you want to differentiate the points according to whether the value of `lot_area` is larger than `1e4`, a logical variable `lot_area > 1e4` can be created.

```
ggplot(data = sahp) + geom_point(mapping = aes(x = liv_area, y = sale_price, shape = lot_area > 1e4))
```

```
#> Warning: Removed 1 rows containing missing values (geom_point).
```

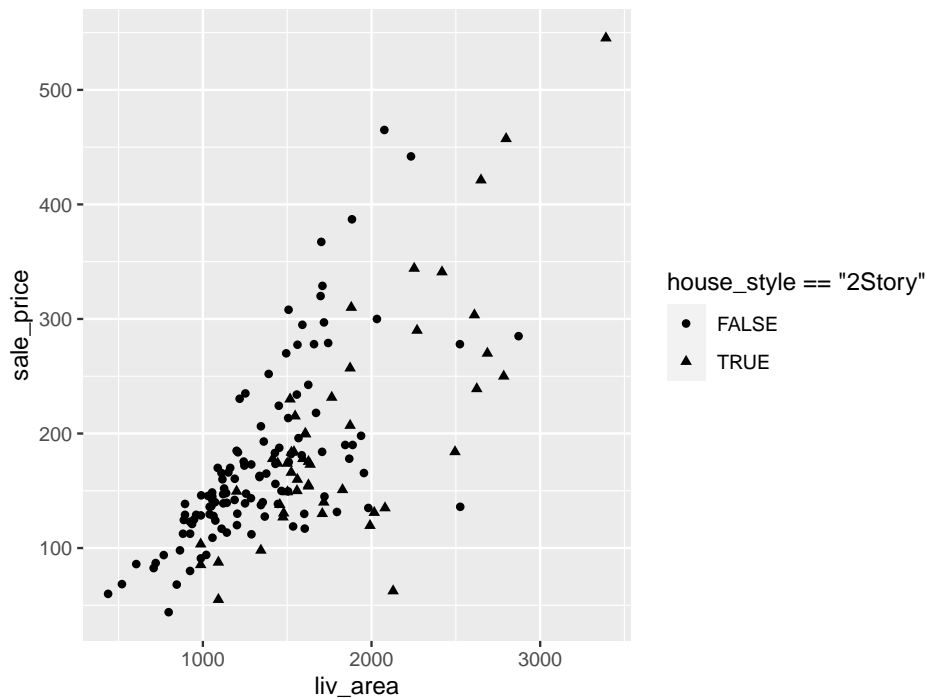


We can see the houses with `lot_area` larger than `1e4` are of different color from those with less than `1e4` in lot area.

Let's see another example where we want to highlight the different between two-story houses from the other types.

```
ggplot(data = sahp) + geom_point(mapping = aes(x = liv_area, y = sale_price, shape = house_style
```

```
#> Warning: Removed 1 rows containing missing values (geom_point).
```



Now, the two-story houses are triangles and other houses are circles.

Clearly, you can easily create new logical variables using any logical operations on existing variables, and map them into any aesthetics just like the existing categorical variables.<sup>1</sup>

## 3.4 Smoothline Fits

Now, you know how to create scatterplots with many possible customizations via specifying different aesthetics. In addition to scatterplots, a very useful type of plots that can capture the trend of pairwise relationship is the **smoothline fits**.

### 3.4.1 Creating Smoothline Fits using `geom_smooth()`

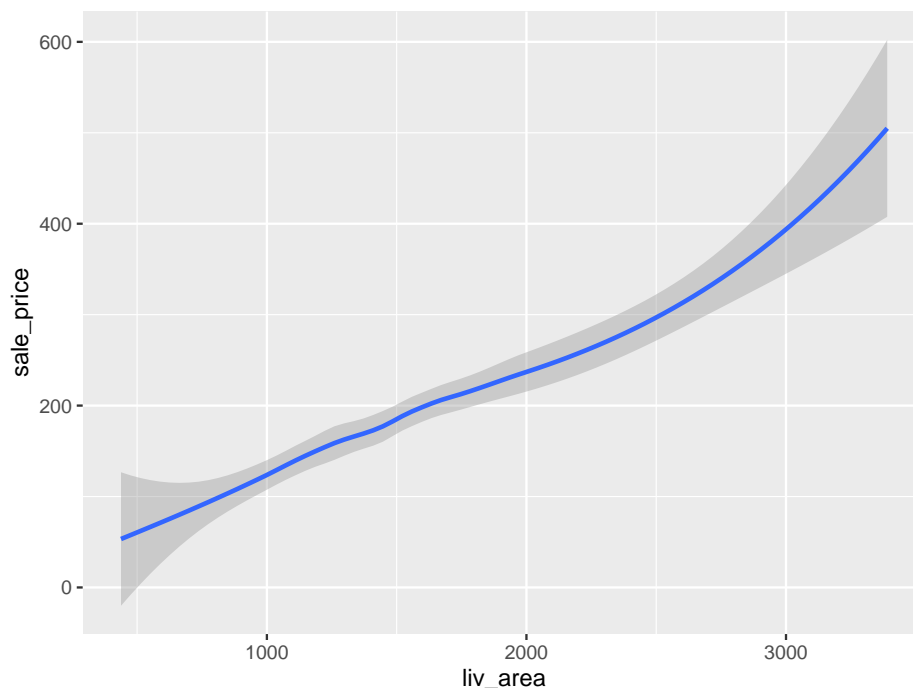
To create a smoothline fit, you can use the `geom_smooth()` function in the **ggplot2** package. Let's say you want to find the trend between the sale price and the living area of a house.



```
library(ggplot2)
library(r02pro)
ggplot(data = sahp) + geom_smooth(mapping = aes(x = liv_area, y = sale_price))
```

```
#> `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```

```
#> Warning: Removed 1 rows containing non-finite values (stat_smooth).
```



Perhaps it is helpful to review the code for generating a scatterplot between `liv_area` and `sale_price`.

```
ggplot(data = sahp) + geom_point(mapping = aes(x = liv_area, y = sale_price))
```

We can see that the only difference is the use of different geoms. In fact, the mechanism of `geom_smooth()` is that it fits a smooth line according to the points of the given variable pair. By default, it uses the **loess** method (locally estimated scatterplot smoothing), which is a popular nonparametric regression technique. In addition to the smoothline, it also generates a shaded area, representing the confidence interval around the fitted smoothline. To hide this shaded area, you can add the argument `se = FALSE` as a global aesthetic.

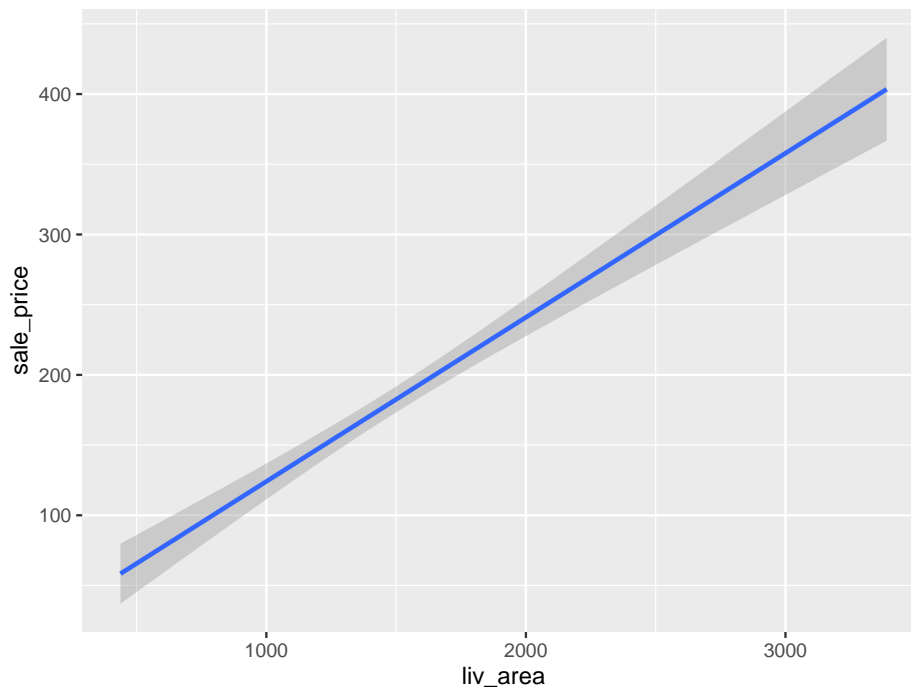
```
ggplot(data = sahp) + geom_smooth(mapping = aes(x = liv_area, y = sale_price), se = FALSE)
```

In addition to the default loess method for smoothline fit, `geom_smooth()` also provides other smoothing methods. For example, we can set `method = "lm"` to fit a linear line.

```
ggplot(data = sahp) + geom_smooth(mapping = aes(x = liv_area, y = sale_price), method = "lm")
```

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

```
#> Warning: Removed 1 rows containing non-finite values (stat_smooth).
```



### 3.4.2 Aesthetics in Smoothline Fits

As in scatterplots, you can also set global aesthetics as well as map variables to aesthetics in smoothline fits. Let's begin with mapping variables to aesthetics. We first define a new logical vector `good_qual` which is `TRUE` when `oa_qual > 5`.

```
sahp$good_qual <- sahp$oa_qual > 5
```

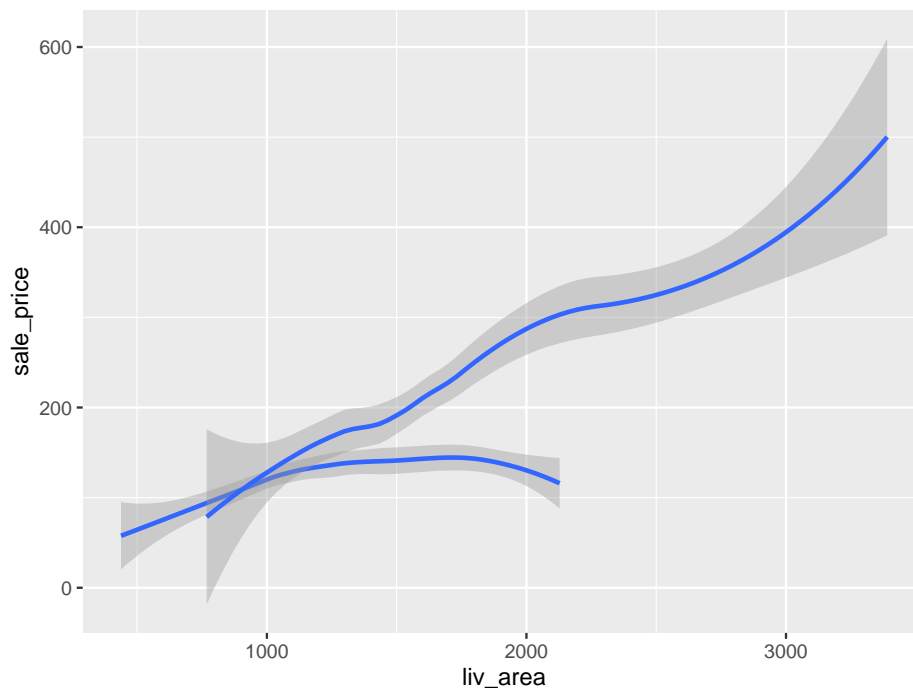
#### a. Group

When we map a variable to the `group` aesthetic, `geom_smooth` will first divide all the data points into different *groups* according to the variable value, and then fit a separate smoothline for each group.

```
ggplot(data = sah) + geom_smooth(mapping = aes(x = liv_area, y = sale_price, group = good_qual))
```

```
#> `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```

```
#> Warning: Removed 1 rows containing non-finite values (stat_smooth).
```



You can see that two smoothlines are generated. However, it is not clear from the plot which group each smoothline corresponds to. To make the two smoothlines different, you can map the variable to other aesthetics.

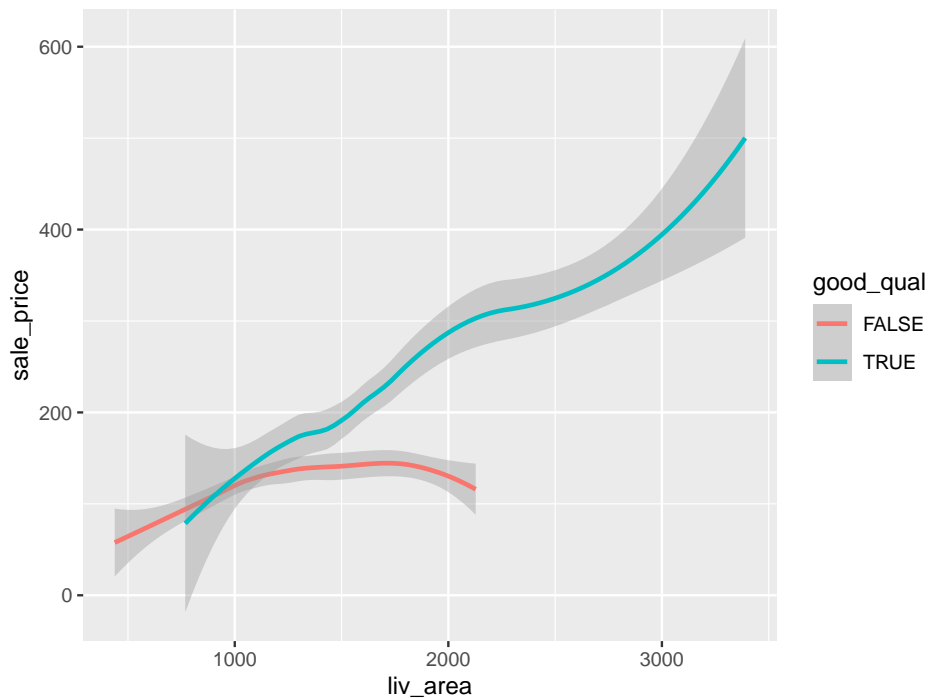
#### b. Color

As in `geom_point()`, we can map the variable to the color aesthetic.

```
ggplot(data = sahp) + geom_smooth(mapping = aes(x = liv_area, y = sale_price, color = good_qual))
```

```
#> `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```

```
#> Warning: Removed 1 rows containing non-finite values (stat_smooth).
```



This is a more informative plot than the one using `group` aesthetic as you can see the two smoothlines have different colors according to the `good_qual` variable.<sup>8</sup>

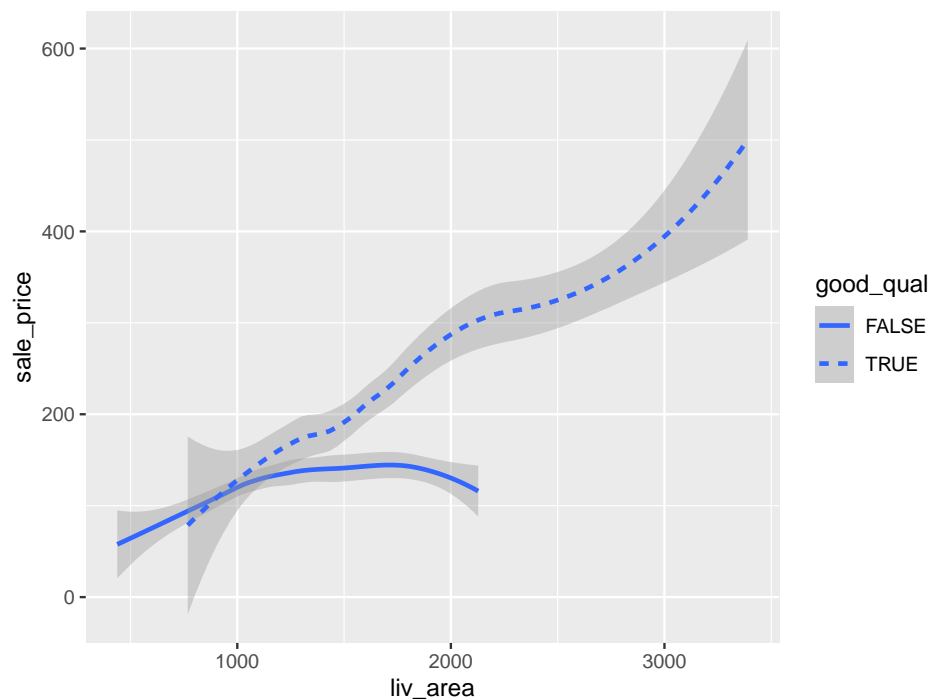
### c. *Linetype*

Another useful aesthetic that was not applicable in `geom_point()` is `linetype`, which controls the linetypes for each smoothline.

```
ggplot(data = sahp) + geom_smooth(mapping = aes(x = liv_area, y = sale_price, linetype = good_qual))
```

```
#> `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```

```
#> Warning: Removed 1 rows containing non-finite values (stat_smooth).
```



The plot shows a dashed line for the smoothline corresponding to `good_qual == TRUE`, and a solid line for the smoothline corresponding to `good_qual == FALSE`.

#### d. Size

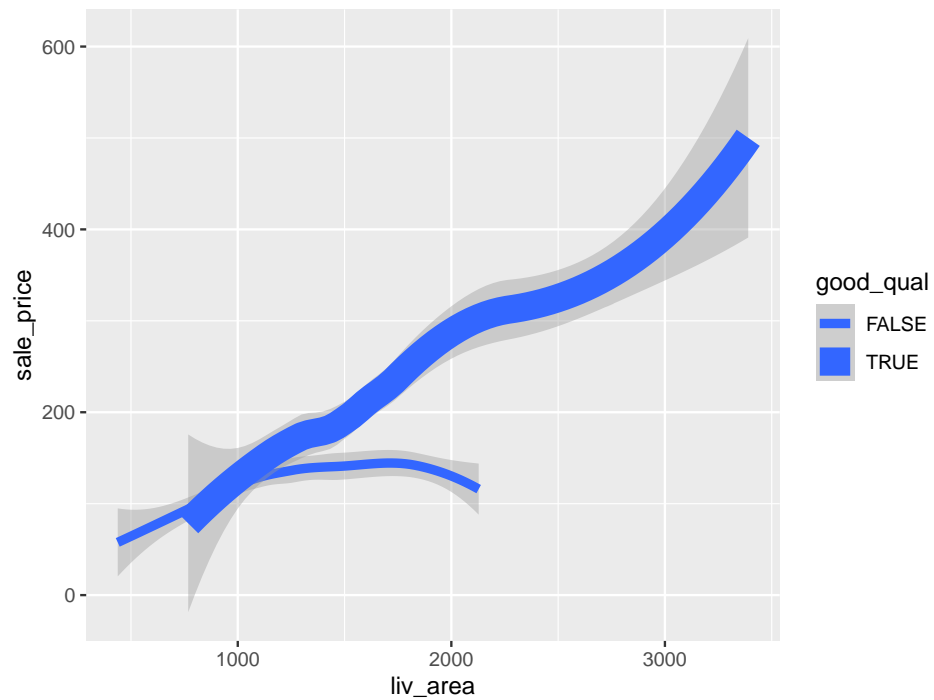
You can also map `good_qual` to the `size` aesthetic, which controls the width of each smoothline fit.

```
ggplot(data = sahp) + geom_smooth(mapping = aes(x = liv_area, y = sale_price, size = good_qual))
```

```
#> Warning: Using size for a discrete variable is not advised.
```

```
#> `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```

```
#> Warning: Removed 1 rows containing non-finite values (stat_smooth).
```



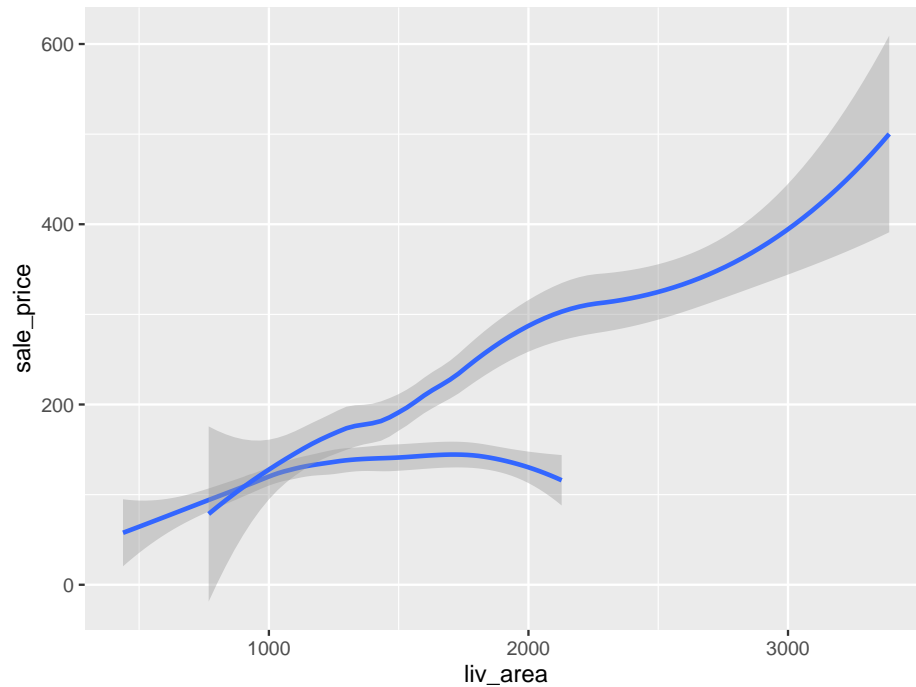
It is worth to mention that `shape` is not a valid aesthetic for `geom_smooth` as it doesn't make sense to talk about the shape of a line.

```
ggplot(data = sahp) + geom_smooth(mapping = aes(x = liv_area, y = sale_price, shape = good_qual))
```

```
#> Warning: Ignoring unknown aesthetics: shape
```

```
#> `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```

```
#> Warning: Removed 1 rows containing non-finite values (stat_smooth).
```



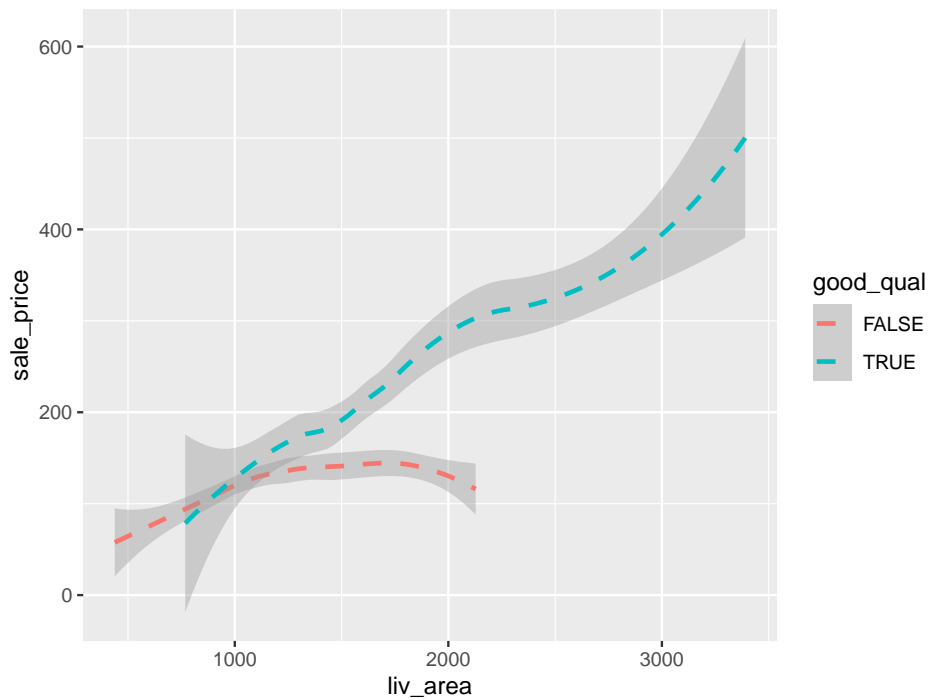
When you tried to map a variable to the `shape` aesthetic, `geom_smooth()` will show a warning message “Warning: Ignoring unknown aesthetics: shape”, and use the `group` aesthetic instead.

Naturally, we can also have global aesthetic and it is straightforward to combine multiple aesthetics in the same plot.

```
ggplot(data = sahp) + geom_smooth(mapping = aes(x = liv_area, y = sale_price, color = good_qual),
```

```
#> `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```

```
#> Warning: Removed 1 rows containing non-finite values (stat_smooth).
```



### 3.5 Multiple geoms and Aesthetics

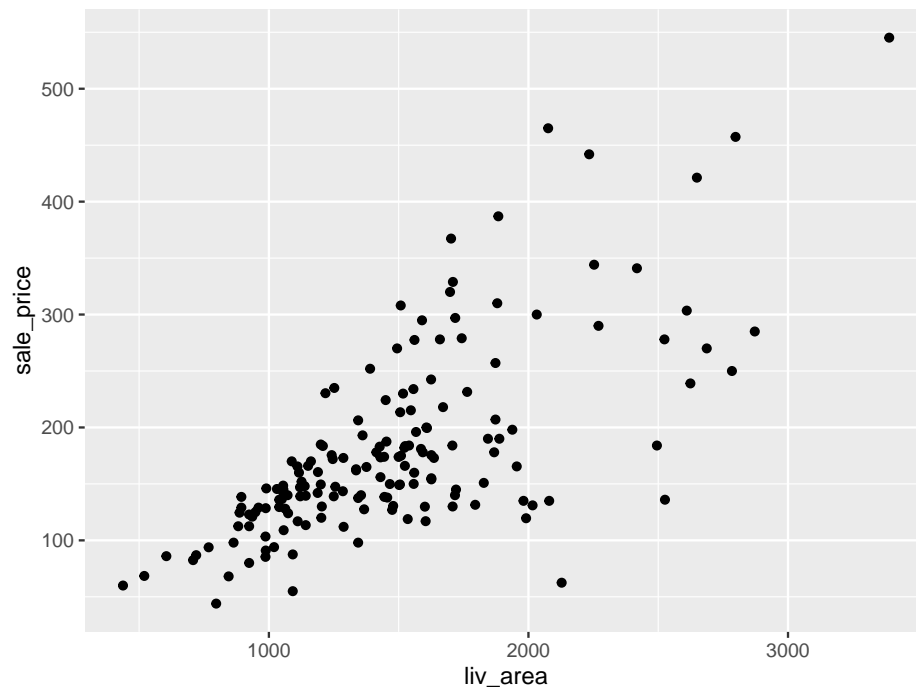
So far, you have learned to create scatterplots using `geom_point()` and smooth-line fits using `geom_smooth()`. It is sometimes useful to combine multiple geoms in the same plot.

Let's first review the scatterplot and smoothline fit between `liv_area` and `sale_price`.

```
library(r02pro)
library(tidyverse)
ggplot(data = sahp) + geom_point(mapping = aes(x = liv_area, y = sale_price))
```

```
#> Warning: Removed 1 rows containing missing values (geom_point).
```

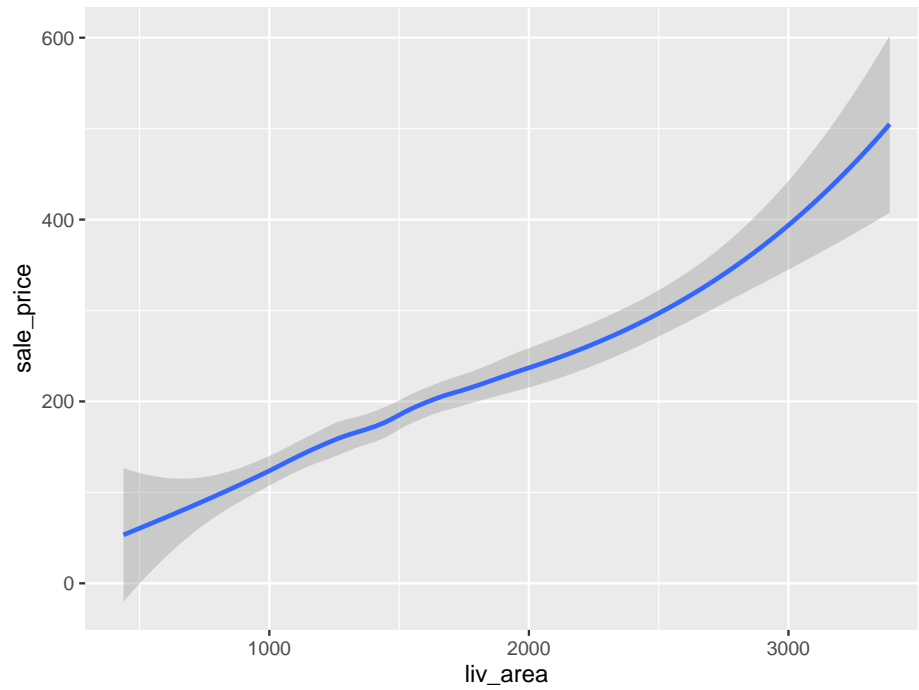




```
ggplot(data = sahp) + geom_smooth(mapping = aes(x = liv_area, y = sale_price))
```

```
#> `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```

```
#> Warning: Removed 1 rows containing non-finite values (stat_smooth).
```



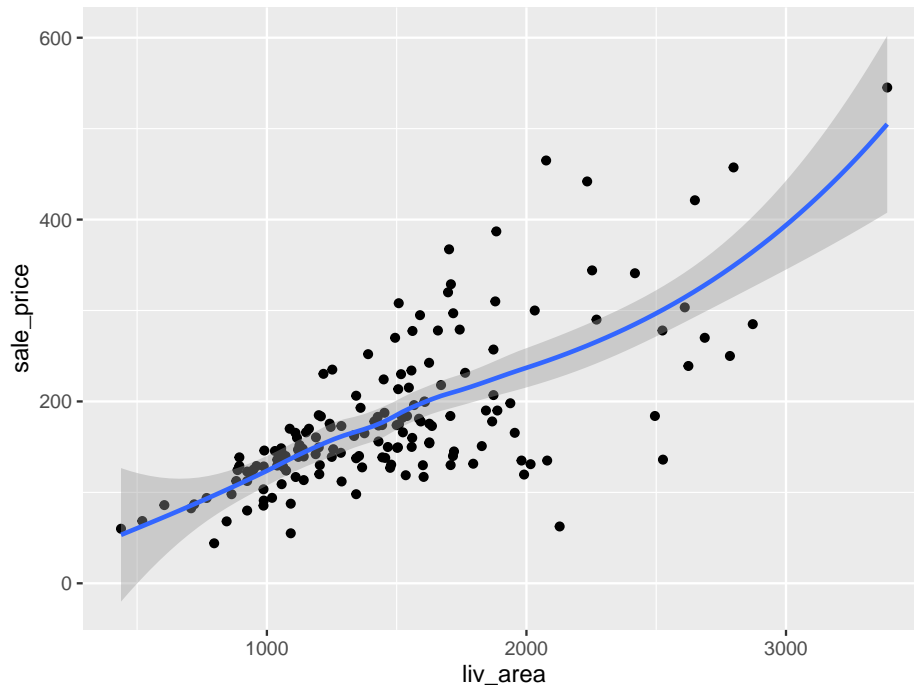
To combine multiple geoms, you can simply use + to add them.

```
ggplot(data = sahp) + geom_point(mapping = aes(x = liv_area, y = sale_price)) + geom_smooth()
```

```
#> `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```

```
#> Warning: Removed 1 rows containing non-finite values (stat_smooth).
```

```
#> Warning: Removed 1 rows containing missing values (geom_point).
```



As expected, you see all the points and the smoothline fit on the same plot, which contains very rich information.

As usual, you can add aesthetics to both geoms.

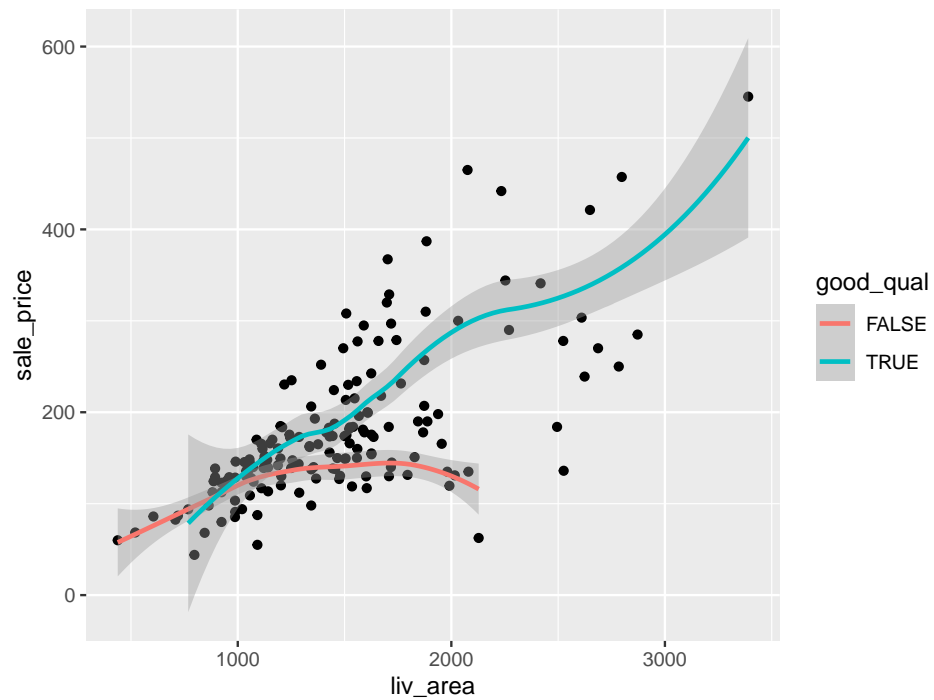
Let's first map `good_qual` (defined as `oa_qual > 5` in Section 3.4) to the color aesthetic for `geom_smooth()`.

```
ggplot(data = sahp) + geom_point(mapping = aes(x = liv_area, y = sale_price)) + geom_smooth(mapping = aes(x = liv_area, y = sale_price, color = good_qual))
```

```
#> `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```

```
#> Warning: Removed 1 rows containing non-finite values (stat_smooth).
```

```
#> Warning: Removed 1 rows containing missing values (geom_point).
```



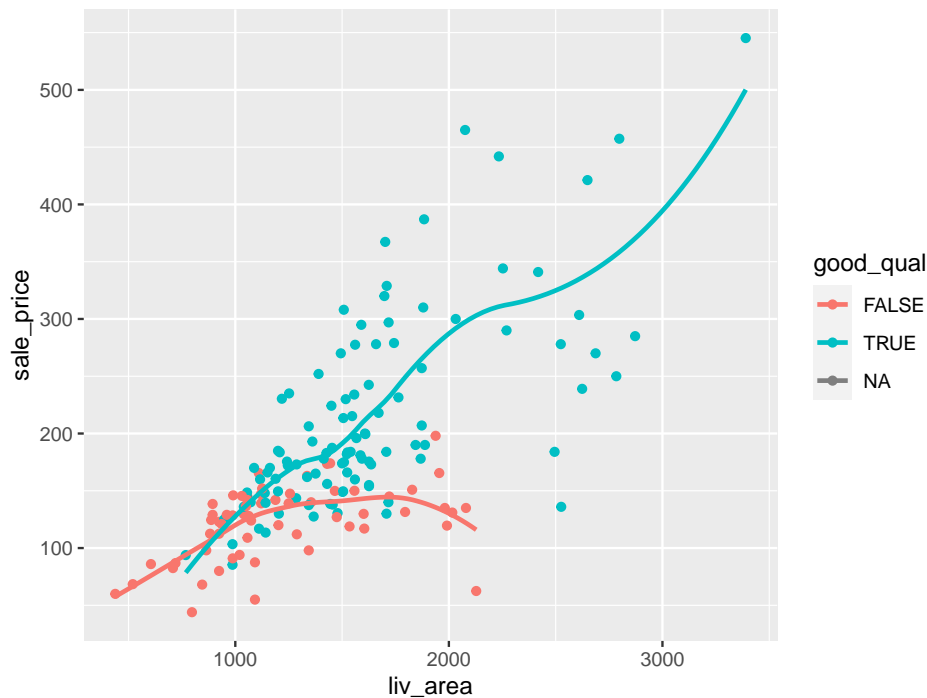
To verify the two smoothline fits are indeed fitted from the data points in the two groups, you can map `good_qual` to the color aesthetic for `geom_point()` as well.

```
ggplot(data = sahp) + geom_point(mapping = aes(x = liv_area, y = sale_price, color = good_qual))
```

```
#> `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```

```
#> Warning: Removed 1 rows containing non-finite values (stat_smooth).
```

```
#> Warning: Removed 1 rows containing missing values (geom_point).
```



The plot is reassuring that the two smoothline fits indeed correspond to the data points in the two groups defined by `good_qual`. Note that the legend also shows a NA category since there is a missing value in the variable `good_qual`.

In addition to mapping variables to aesthetics, you can also use global aesthetics.

```
ggplot(data = sahp) + geom_point(mapping = aes(x = liv_area, y = sale_price, color = good_qual, s
```

```
#> `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```

```
#> Warning: Removed 1 rows containing non-finite values (stat_smooth).
```

```
#> Warning: Removed 1 rows containing missing values (geom_point).
```



### 3.6 Global and Local Aesthetic Mappings

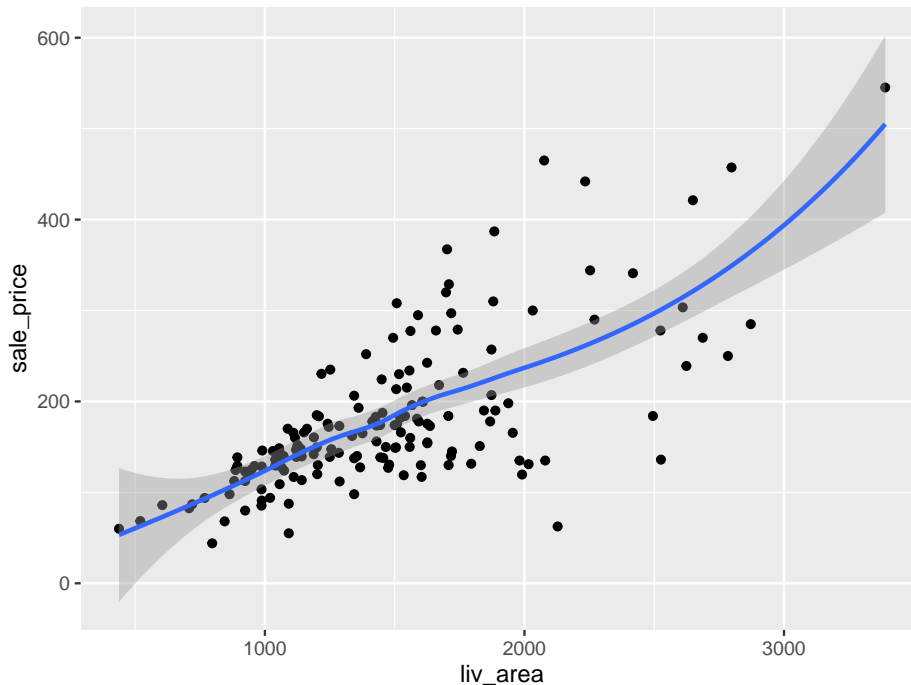
In Section 3.5, you learned how to combine the scatterplot and smoothline fit into a single plot by using two geoms.

Let's first review the code.

```
library(r02pro)
library(tidyverse)
ggplot(data = sahp) + geom_point(mapping = aes(x = liv_area, y = sale_price)) + geom_smooth(
  #> `geom_smooth()` using method = 'loess' and formula 'y ~ x'

  #> Warning: Removed 1 rows containing non-finite values (stat_smooth).

  #> Warning: Removed 1 rows containing missing values (geom_point).
```



You may notice that the arguments inside `geom_point()` and `geom_smooth()` are identical. Now, thinking about if we want to generate another plot by replacing the `liv_area` with `lot_area` (the size of lot area), both instances of `liv_area` need to be changed to `lot_area`, which is a bit cumbersome. It turns out we can use the so-called **global mapping** to simplify the code. Correspondingly, we call a mapping inside a specific geom **local mapping**.

To use the global mapping, we move the mapping argument from the geoms into the `ggplot()` function.

```
ggplot(data = sahp, mapping = aes(x = liv_area, y = sale_price)) + geom_point() + geom_smooth()
```

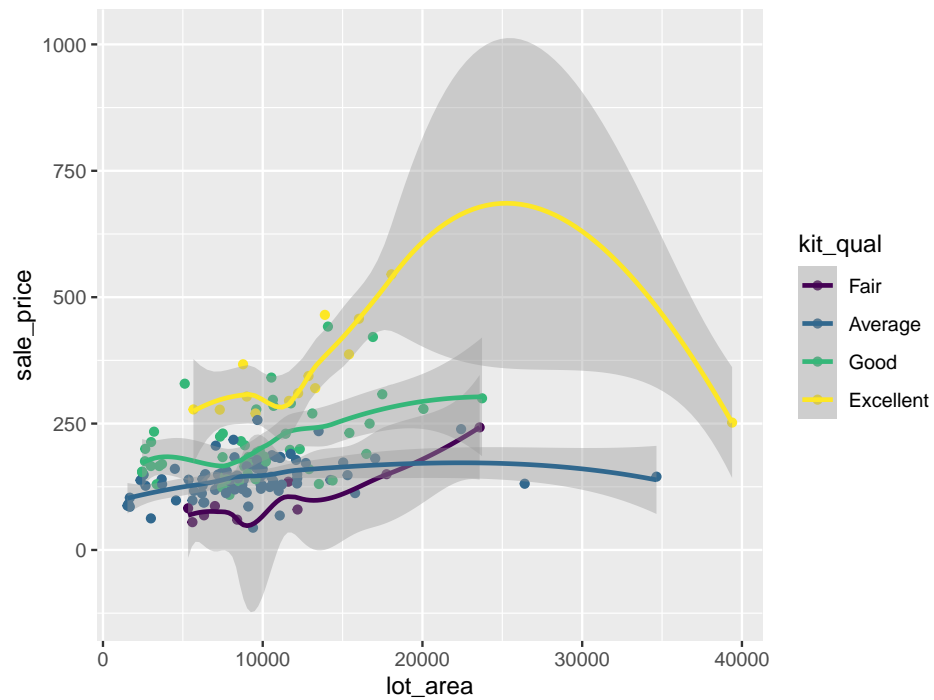
We can also map variables to aesthetics in the global mapping. The global aesthetic mapping will be inherited in all geoms used.

```
ggplot(data = sahp, mapping = aes(x = lot_area, y = sale_price, color = kit_qual)) + geom_point()
```

```
#> `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```

```
#> Warning: Removed 1 rows containing non-finite values (stat_smooth).
```

```
#> Warning: Removed 1 rows containing missing values (geom_point).
```



Clearly, the use of global mapping greatly simplify our code as we would need to repeat the same mapping argument in both geoms.

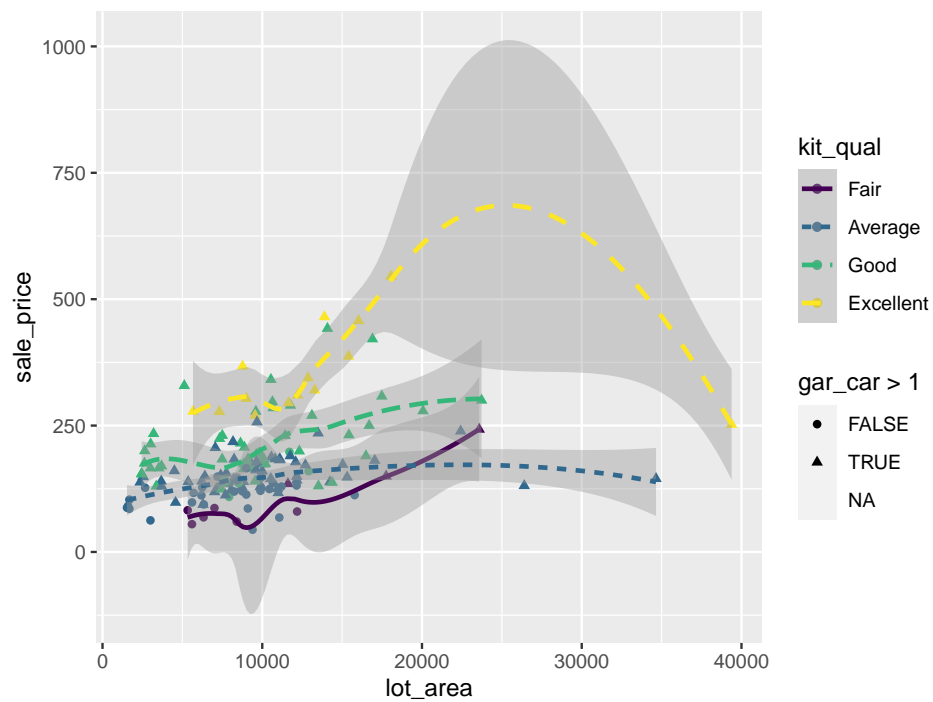
In addition to use the same mapping in each geom as the global mapping, we can extend or overwrite the global mapping in each geom.

### 1. Extend Global Aesthetic Mappings

```
ggplot(data = sahp, mapping = aes(x = lot_area, y = sale_price, color = kit_qual)) + geom_point() +
```

```
#> `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```

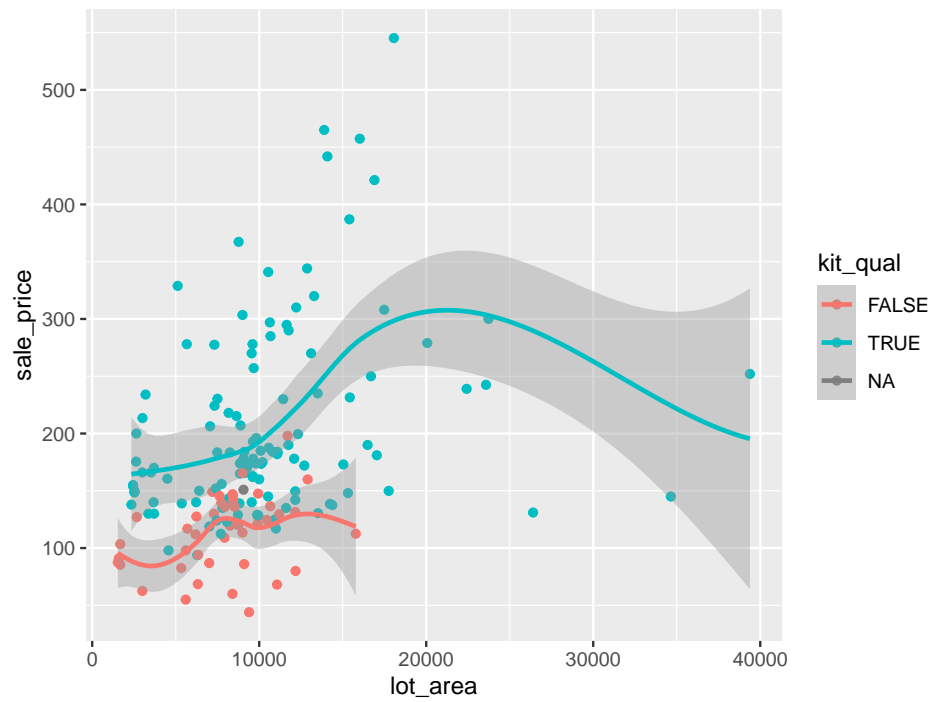




## 2. Overwrite Global Aesthetic Mappings

```
ggplot(data = sahp, mapping = aes(x = lot_area, y = sale_price, color = kit_qual)) + geom_point(m
```

```
#> `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



### 3. Mix

```
ggplot(data = sahp, mapping = aes(x = liv_area, y = sale_price, color = kit_qual)) + g
```

```
#> `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```





## Chapter 4

# Data Import and Export

So far in this book, you have been creating objects by yourself or working with existing data in R packages. When working on a project, you often need to **import** existing data into R, or **export** the created object into a file on the computer. In this chapter, you will learn how to import and export data of different file types.

### 4.1 Exporting Data to Delimited Files

We will start by introducing how to export data to a file in this chapter.

#### 4.1.1 Set the working directory

Firstly, we will introduce an important concept of **Working Directory**. To conduct the data export and import, you are recommended to set the *working directory* since we usually use a path relative to the working directory for interacting with files on the computer in R, . To set the working directory, you can click *Session* on the menu and click *Set Working Directory*. **ADD a screenshot with this menu here** There are three options under this menu.

- *To Source File Location*: this is the same directory as the current R script.
- *To Files Pane Location*: this is the same directory as shown in the Files Panel on the bottom right of RStudio.
- *Choose Directory...*: this will open up a window from which you can choose any desired directory.

After selecting any of the three options, we can see a line of code containing the function `setwd()` executed in the console. **ADD a screenshot with this line**

**of code here** Indeed, this menu operation is equivalent to using the `setwd()` function with the argument being the full path or relative path of the desired directory.

Another related function is `getwd()` which tells us the absolute path representing the current working directory.

```
getwd()
```

### 4.1.2 Delimited files

In most applications, you will use a specific file type called **delimited** file. In a delimited file, each row represents a single observation, and it has values separated by the **delimiter**. In principle, *any character (including letters, numbers, or symbols)* can be used as a delimiter, with the most commonly used ones being comma, tab, colon, and space.

### 4.1.3 Write an object into a .csv file

First, let's work with one popular kind of *delimited* files called *comma-separated value* file, usually with the file extension `.csv`. In a `.csv` file, the *delimiter* is *comma* (`,`).

Let's review the data frame you created in **Section???**.

```
dig_num <- 7:1
ani_char <- c("sheep", "pig", "monkey", "pig", "monkey", NA, "pig")
conditions <- c("Excellent", "Good", "N", "Fair", "Good", "Good", "Excellent")
my_animals <- data.frame(dig_num, ani_char, conditions)
my_animals
```

Now, let's write the data frame `my_animals` into a file called "my\_animals.csv" in the currently working directory. To write an object into a `.csv` file, you will use the `write_csv()` function in the **readr** package. Since **readr** is a subpackage of **tidyverse**, you don't need to install it separately, but you need to load the package in each new R session.

```
library(readr)
write_csv(my_animals, "my_animals.csv")
```

You can verify the `.csv` file has been indeed created and open the file with RStudio or any text editor to verify its contents. **Shall we use a screenshot**

**again?** We can see that all the information has been written in the .csv file, which has *commas* separating the values on each line. In particular, you may find out the first row of the file corresponds to the column names. If you don't want to include the column names, you can set the argument `col_names` to be `FALSE`.

```
write_csv(my_animals, "my_animals_no_colname.csv", col_names = FALSE)
```

By default, `write_csv()` writes the data into a file in which `NA` is used to represent all the missing values, just like in the tibble. If you want to use another string to represent the missing values in the file, you can set the argument `na` to be the string.

```
write_csv(my_animals, "my_animals_missing.csv", na = "This value is missing!")
```

#### 4.1.4 Write an object into a delimited file

As introduced at the beginning, there are different types of *delimited files* depending on the specific *delimiter*. The function `write_delim()` enables us to write an object into a delimited file with any chosen delimiter. The usage of `write_delim()` is almost identical to `write_csv()`, except that it has an additional argument `delim`, which specifies the delimiter to be used. Let's see the following example with `*` as the delimiter.

```
write_delim(my_animals, "my_animals_star.csv", delim = "*")
```

#### 4.1.5 Exercise

You can run the following code to do the exercise.

```
r02pro("4")
```

## 4.2 Importing Data from Delimited Files

Knowing how to export data into delimited files, let's see how to **import** data from the delimited files.

### 4.2.1 Import .csv Files using `read_csv()`

To import .csv files, we can use the function `read_csv()` in the **readr** package, which is a sub-package of **tidyverse**. If you have already installed **tidyverse**, you can directly load the **readr** package.

After loading the **readr** package, you can try to import the data from “my\_animals.csv”, which you want to make sure is in the current working directory.

```
library(readr)
my_animals <- read_csv("my_animals.csv")
```

```
#>
#> -- Column specification -----
#> cols(
#>   dig_num = col_double(),
#>   ani_char = col_character(),
#>   conditions = col_character()
#> )
```

We can see there is a message showing the *Column specification* during the import process. In particular, we see `dig_num` is of type *double* (or *numeric*), and both `ani_char` and `cond_fac` are of type *character*. We can also check the value of `my_animals` and its structure.

```
my_animals
str(my_animals)
```

We can see that the tibble `my_animals` is generated along with the correct column types. In order to introduce the various options associated with `read_csv()` function, let’s move on to the topic of **inline** .csv files next.

### 4.2.2 Read Inline .csv Files

The `read_csv()` function not only can read files into R, it also accept *inline input* as its argument. While the inline input may not be commonly used in practice, it is particularly useful for learning how to use the function. Let’s see an example.

```
read_csv("x,y,z
1,3,5
2,4,6")
```



You can see that a tibble is generated with 2 rows and 3 columns with the column names being `x`, `y` and `z`. From the argument, we can see that by default, the first row of the input data will be interpreted as the column names. If the input data doesn't correspond to the variable names, you need to set `col_names = FALSE` as an additional argument in `read_csv()`.

```
read_csv("x,y,z
1,3,5
2,4,6", col_names = FALSE)
```

Now, a tibble of 3 rows and 3 columns was generated, with the column names being `X1`, `X2`, and `X3`. Note that these are the naming convention in the function when you don't supply the column names in the file. Another thing worth mentioning is that all three variables are of *character* types, due to the fact that there are character values for all variables (`x`, `y`, and `z`).

Sometimes, the first few lines of your data file may be descriptions of the data, which you want to skip when import into R. We can set the `skip` argument in the `read_csv()` function to skip a certain number of lines.

```
read_csv("The first line
The second line
The third line
x,y,z
1,3,5", skip = 3)
```

It is clear from the result that the first 3 lines of the input data is skipped.

Another useful argument to when we have comments in the data file is the `comment` argument, which tells R to skip all text after the string specified in the `comment` argument.

```
read_csv("x,y,z #variable names
1,3,5 #the first observation
2,4,6 #the second observation", comment = "#")
```

### 4.2.3 Handling Missing Values

In many real data sets, we may have missing values. You may recall that R uses `NA` to represent the missing values. If the data set was prepared by an R user, it probably already uses `NA` to represent all the missing values. In this case, `read_csv()` will automatically interpret all `NAs` as missing values.

```
read_csv("x,y,z
          999,3,5
          NA,-999,6")
```

In a typical application, however, the person who prepared the data may use other strings to represent missing value. For example, if 999 and -999 are used as the indicators for missing values, you can set the argument `na` to be the vector for those values.

```
read_csv("x,y,z
          999,3,5
          999,-999,6", na = c("999", "-999", "NA"))
```

You can see from the output tibble that all the missing values are now denoted as NA.

#### 4.2.4 Importing data from a delimited file

You now know how to import data from a .csv file using `read_csv()`. More generally, `read_delim()` allows us to import data from a delimited file with any chosen *delimiter*. The usage of `read_delim()` is almost identical to `read_csv()`, except that it has an additional argument `delim`, which specifies the delimiter to be used. Let's see the following example with `*` as the delimiter.

```
my_animals <- read_delim("my_animals_star.csv", delim = "*")
```

```
#>
#> -- Column specification -----
#> cols(
#>   dig_num = col_double(),
#>   ani_char = col_character(),
#>   conditions = col_character()
#> )
```

#### 4.2.5 Import data using the menu

Besides writing codes involving `read_csv()` or `read_delim()` to import data, you can also take advantage of the interactive menu RStudio provides.

Table 4.1: Import Data from Menu

Choice	Name
From Text (readr)	Delimited Files (.csv, .txt, and others)
From Excel	Excel Files (.xls and .xlsx)
From SPSS	SPSS Files (.sav)
From SAS	SAS Files (.sas7bdat and .sas7bcat)
From Stata	Stata Files (.dta)

To do this, you can click on the **Import Dataset** button in the **Environment** panel on the top right of RStudio. Here, you can see quite a few options which are summarized in the following table.

We will focus on importing delimited files in this section. We will cover importing Excel files in Section 4.3. Working with SPSS, SAS, and Stata files will be covered in Section 4.4.

For importing a .csv file, .txt file, or any other file with a delimiter, you can choose the **From Text (readr)** option. Then, you can click **Browse...** and select the data file.

After a file is selected, you can see the **Data Preview** which showing the first several rows of the data. Note that the first row shows the column names and their associate types in parentheses. For each column, you can click the dropdown menu after the type to change its type.

When we select a .csv file, we may see the function `read_csv()` in the *Code Preview* window. Indeed, `read_csv()` is the backbone for reading .csv files into R.

### We need a screen shot?

The bottom area shows many **Import Options**. Let's look at a few commonly used options, their corresponding arguments in the `read_csv()` or `read_delim()` function, and meanings.

```
\begin{table}
```

```
\caption{Menu Options and its Corresponding Arguments in read_delim()
and Meanings}
```

Option	Argument	Meaning
Name	-	The object name you would like to assign to.
Skip	'skip'	The number of rows to skip at the beginning of the file.
First Row as Names	'col_names'	Whether you want to use the first row as column names. 'TRUE' or 'FALSE'
Delimiter	'delim'	The delimiter of the data file.
Comment	'comment'	The character indicating the starting of comment. The contents after the co
NA	'na'	The way NA is represented in the data file.
Code Preview	-	The R code to be executed for importing the data

```
\end{table}
```

Note that when you change these options, the code in the *Code Preview* window will change accordingly, which is a great way to learn on how they work.

### 4.2.6 Exercise

You can run the following code to do the exercise.

```
r02pro("4")
```

## 4.3 Exporting and Importing Data from Excel Files

Now, you know how to export and import data from *delimited* files. In this section, you will learn how to export and import data from Excel files with extensions `.xls` and `.xlsx`.

### 4.3.1 Export data into Excel files

To export data into an Excel file, you can use the **writexl** package. Let's first install the package.

```
install.packages("writexl")
```

Now, we can load the **writexl** package and use the `write.xlsx()` function to write data into an Excel file with extension `.xlsx`.

```
library(writexl)
```

```
#> Error in library(writexl): there is no package called 'writexl'
```

```
write.xlsx(my_animals, "my_animals.xlsx")
```

```
#> Error in write.xlsx(my_animals, "my_animals.xlsx"): could not find function "write_
```

By default, the column name of the data frame/tibble will be written to the first row of the Excel file.

In addition to writing one data frame to an Excel file, `write.xlsx()` can also write multiple data frames into a single Excel file, with each sheet containing each data frame. Let's take a look at the following example which write both `my_animals` and `sahp` (a tibble in the `r02pro` package) into an Excel file named "two\_data.xlsx".

```
two_data <- list(my_animals = my_animals, sahp = sahp)
write.xlsx(two_data, "two_data.xlsx")
```

```
#> Error in write.xlsx(two_data, "two_data.xlsx"): could not find function "write.xlsx"
```

### 4.3.2 Import Excel Files (.xls and .xlsx ) using `read_excel()`

Having learned how to export data into an Excel file, let's see how to read an existing Excel file into R. We can use the `read_excel()` function in the `readxl` package to import Excel files. Here, `readxl` is another subpackage in the `tidyverse` package. Thus we can directly load the package and use it.

Let's import the sheet `sahp` from the Excel file "two\_data.xlsx".

```
library(readxl)
shap_1 <- read_excel("two_data.xlsx", sheet = "sahp")
head(shap_1)
```

```
#> # A tibble: 6 x 13
#>   dt_sold      bedroom bathroom gar_car oa_qual liv_area lot_area
#>   <dtm>      <dbl>    <dbl>   <dbl>   <dbl>   <dbl>   <dbl>
#> 1 2010-03-25 00:00:00      3     2.5     2       6    1479    13517
#> 2 2009-04-10 00:00:00      4     3.5     2       7    2122    11492
#> 3 2010-01-15 00:00:00      3      2     1       5    1057     7922
#> 4 2010-04-19 00:00:00      3     2.5     2       5    1444     9802
#> 5 2010-03-22 00:00:00      3      2     2       6    1445    14235
#> 6 2010-06-06 00:00:00      2     2.5     2       6    1888    16492
#> # ... with 6 more variables: house_style <chr>, kit_qual <chr>,
#> #   heat_qual <chr>, central_air <lgl>, sale_price <dbl>, good_qual <lgl>
```

If we only want to import a portion of the data, let's say the first 5 rows and the first four columns, then we can set the argument `range = "A1:D5"`.

```
shap_2 <- read_excel("two_data.xlsx", sheet = "sahp", range = "A1:D5")
shap_2
```

```
#> # A tibble: 4 x 4
#>   dt_sold      bedroom bathroom gar_car
#>   <dtm>      <dbl>      <dbl>   <dbl>
#> 1 2010-03-25 00:00:00      3      2.5      2
#> 2 2009-04-10 00:00:00      4      3.5      2
#> 3 2010-01-15 00:00:00      3      2       1
#> 4 2010-04-19 00:00:00      3      2.5      2
```

### 4.3.3 Import Excel file using the menu

Besides using `read_excel()` to import Excel files, you can again use the interactive menu we introduced in Section 4.2 .

As introduced in Table 4.1, to import Excel files, you can select *From Excel* after choosing the **Import Dataset** option. As before, you can click **Browse...** and select the data file. Let's select the “two\_data.xlsx” file we just created.

Similar to importing the delimited files, we can see the first several rows in the **Data Preview** windows. The first row shows the column names and their associate types in parentheses. For each column, you can click the dropdown menu after the type to change its type. Now, let's discuss several options in the **Import Options** section and their corresponding arguments in the `read_excel()` function.

\begin{table}

\caption{Menu Options and its Corresponding Arguments in `read_excel()` and Meanings}

Option	Argument	Meaning
Name	-	The object name you would like to assign to.
Sheet	'sheet'	The Sheet you want to import from.
Range	'range'	The data range you want to import.
Max Rows	'n_max'	The maximum number of rows to import.
Skip	'skip'	The number of rows to skip at the beginning of the file.
NA	'na'	The way NA is represented in the data file.
First Row as Names	'col_names'	Whether you want to use the first row as column names. 'TRUE'
Code Preview	-	The R code to be executed for importing the data

\end{table}

Note that similar as importing delimited files, when you change these options, the code in the *Code Preview* window will change accordingly, which is a great way to learn on how they work.

### 4.3.4 Exercise

You can run the following code to do the exercise.

```
r02pro(4)
```

## 4.4 Working with Data from SPSS, SAS, and Stata Files

Now, you know how to export and import data from delimited files and Excel files. In the section, you will learn how to export and import data from other statistical software including SPSS, SAS, and Stata. We will use the package **haven**, another member of the **tidyverse** family.

### 4.4.1 Export and Import SPSS Files

Let's first defined a data frame for

```
dig_num <- 7:1
ani_char <- c("sheep", "pig", "monkey", "pig", "monkey", NA, "pig")
conditions <- c("Excellent", "Good", "N", "Fair", "Good", "Good", "Excellent")
my_animals <- tibble(dig_num, ani_char, conditions)
my_animals
```

The data frame `my_animals` will be used as in Section 4.1. You can use the function `write_sav()` to export a data frame into a SPSS `.sav` file.

```
library(haven)
write_sav(my_animals, "my_animals.sav")
```

To read a SPSS file ending in `.sav` or `.por`, you can use the function `read_spss()` which will automatically call `read_sav()` for `.sav` files and `read_por()` for `.por` files.

```
my_animals_spss <- read_spss("my_animals.sav")
head(my_animals_spss)
```

### 4.4.2 Export and Import SAS Files

You can use the function `write_sas()` to export a data frame into a SAS `.sas7bdat` file.

```
write_sas(my_animals, "my_animals.sas7bdat")
```

To import a SAS file, you can use the function `read_sas()`.

```
my_animals_sas <- read_sas("my_animals.sas7bdat")
head(my_animals_sas)
```

### 4.4.3 Export and Import Stata Files

Lastly, let's talk about Stata files. You can use the function `write_dta()` to export a data frame into a Stata `.dta` file.

```
write_dta(my_animals, "my_animals.dta")
```

To read a Stata file ending in `.dta`, you can use the function `read_dta()`.

```
my_animals_stata <- read_dta("my_animals.dta")
head(my_animals_stata)
```

### 4.4.4 Import using the menu

Similarly as Sections 4.2 and 4.3, you can also use the menu in Table 4.1 to import SPSS, SAS, and Stata Files.

### 4.4.5 Exercise

You can run the following code to do the exercise.



```
r02pro("8.1")
```

## 4.5 Save and Restore Objects and Workspace

Now, you know how to export and import data frames (or tibbles) to and from various types of file. In this section, you will learn how to save and restore one or more objects that can be **of any types**, and even the **whole workspace** that includes all the named objects.

To get started, let's first clear our workspace using `rm(list = ls())` and create a few objects with different types.

```
rm(list = ls())
dig_num <- 7:1
ani_char <- c("sheep", "pig", "monkey", "pig", "monkey", NA, "pig")
my_list<- list(dig_num = dig_num, ani_char = ani_char)
```

Recall that we can use `ls()` to get a vector of strings giving the names of the objects in the current environment.

```
ls()
```

```
#> [1] "ani_char" "dig_num" "my_list"
```

### 4.5.1 Save and Restore Objects using .RData

In R, you can use the function `save()` to save one or more objects into an .RData file. Note that you want to make sure to change the working directory as needed. Let's see the following example where we save the object `dig_num` into a file named "dig\_num.RData".

```
save(dig_num, file = "dig_num.RData")
```

Before introducing how to restore objects, let's first remove `dig_num` from our workspace using the `rm()` function.

```
rm(dig_num)
dig_num
```

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

You can see that `dig_num` has indeed been removed from the workspace. To restore it, you can use the function `load()` with the corresponding `.RData` in double quotes as its argument.

```
load("dig_num.RData")
dig_num
```

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

You can verify from the value of `dig_num` that we have successfully restored the object `dig_num` from the file “`dig_num.RData`”.

To save more than one objects into one file, you just need to enter them as additional arguments in the `save()` function.

```
save(dig_num, ani_char, file = "dig_num_and_ani_char.RData")
```

To save everything in the workspace, you can use the function `save.image()` with the desired file name in double quotes as the argument.

```
save.image("all.RData")
```

To verify that “`all.RData`” indeed contains all the named object, let’s do the following.

```
rm(list = ls())  #remove everything from the workspace.
ls()             #confirm the workspace is empty.
```

```
#> character(0)
```

```
load("all.RData") #restore from "all.RData".
ls()               #check what's in the workspace.
```

```
#> [1] "ani_char" "dig_num" "my_list"
```

### 4.5.2 Save and Restore a Single Object using `saveRDS()` and `loadRDS()`

Before introducing the new method, there is one drawback of `load()` worth noting: if the imported `.RData` file contains objects with the same names as in the current workspace, **all** these objects in the current workspace will be silently **overwritten** without any warning! Let's see the following example.

```
dig_num <- 724
dig_num
```

```
#> [1] 724
```

```
load("all.RData")
dig_num
```

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

We can see that the value of `dig_num` was indeed silently *overwritten* by the `load()` function, which could be sometimes dangerous.

To avoid this issue, another pair of functions to save and restore a single object is `saveRDS()` and `loadRDS()`. The usage of `saveRDS()` is almost identical to `save()` except we usually use a file with extension `“rds”` to store the object.

```
saveRDS(dig_num, file = "dig_num.rds")
```

To highlight the different behaviors of `readRDS()` and `load()`, let's change the `dig_num` again.

```
dig_num <- 826
dig_num
```

```
#> [1] 826
```

To restore the object in an `“rds”` file, we use the `readRDS()` in the following way.

```
dig_num_new <- readRDS("dig_num.rds")
dig_num_new
```

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

```
dig_num
```

```
#> [1] 826
```

As it is clearly from this example, you need to assign the value of the `readRDS()` function to a name, which helps to prevent any objects been overwritten silently. In fact, the `saveRDS()` only saves the value of the object without the object name.

For this reason, you are recommended to use the function pair `saveRDS()` and `readRDS()` if you want to save and load one R object. While `save()` and `load()` may be simpler to use when saving and loading multiple objects, you want to be extremely careful with the overwriting issues we discussed here.

### 4.5.3 Exercise

You can run the following code to do the exercise.

```
r02pro(4)
```

## Chapter 5

# Data Manipulation

For conducting data analysis, we often need to conduct various kinds of data manipulation. We will use the **ahp** dataset in the **r02pro** package throughout this chapter. Let's first review the dataset.

```
library(r02pro)
ahp
```

**ahp** is a dataset of 2048 houses in Ames, Iowa from 2006 to 2010, with 56 features including the sale date and price. To learn more about each variable, you can look at its documentation.

```
?ahp
```

To view the entire dataset, you can use the **View()** function, which will open the dataset in the new file window.

```
View(ahp)
```

To get the first 6 rows of **ahp**, you can use the **head()** function, which also has an optional argument if you want a different number of top rows.

```
head(ahp)
head(ahp, n = 10) #the first 10 rows of ahp
```

The following are some possible questions we may want to explore.

1. (pick observations by their values) Find the houses that are sold in Jan 2009.
2. (reorder the observations) Find the houses with the highest sale prices.
3. (pick variable by their names) We see there are 56 columns. For a particular data analysis question, perhaps we want to focus on a subset of the columns.
4. (create new variables as functions of existing ones) From the existing variables, perhaps we want to create new ones, for instance, the average price per living area.
5. (create various summary statistics) We may want to create certain summary statistics. For example, what is the average sale price for each type of houses?

## 5.1 Filter Observations and Objects Masking

Let's start with the first task outlined at the beginning of this chapter. Suppose we want to find the houses that are sold in Jan 2009. You can use the function `filter()` in the **dplyr** package, a member of the **tidyverse** package.

If you haven't installed the **tidyverse** package, you need to install it. Let's first load the **dplyr** package.

```
library(dplyr)
```

### 5.1.1 Objects Masking

After loading the package **dplyr**, you can see the following message

The following objects are masked from 'package:stats':

```
filter, lag
```

The message appears because **dplyr** contains the functions `filter()` and `lag()` which are already defined and preloaded in the R package **stats**. As a result, the original functions are masked by the new definition in **dplyr**.

In this scenario when the same function name is shared by multiple packages, we can add the package name as a prefix to the function name with *double colon* (`::`). For example, `stats::filter()` represents the `filter()` function in the **stats** package, while `dplyr::filter()` represents the `filter()` function in the **dplyr** package. You can also look at their documentations.

```
?stats::filter  
?dplyr::filter
```

It is helpful to verify which version of `filter()` you are using by typing the function name `filter`.

```
filter
```

Usually, R will use the function in the package that is loaded last. To verify the search path, you can use the `search()` function. R will

```
search()
```

`ahp` is a dataset of 2048 houses in Ames, Iowa from 2006 to 2010, with 56 features including the sale date and price. To learn more about each variable, you can look at its documentation.

```
?ahp
```

The following are some possible questions we may want to explore.

1. (pick observations by their values) Find the houses that are larger than 2K sq. ft.
2. (reorder the observations) Find the houses with the highest sale prices.
3. (pick variable by their names) We see there are 56 columns. For a particular data analysis question, perhaps we want to focus on a subset of the columns.
4. (create new variables as functions of existing ones) From the existing variables, perhaps we want to create new ones, for instance, the average price per living area.
5. (create various summary statistics) We may want to create certain summary statistics. For example, what is the average sale price for each type of houses?





## Chapter 6

# Tidy Data

In the part several chapters, you have learned a lot in data visualization, data import and export, and data manipulation. All the data you have seen so far share a very attractive property, namely, they are all *tidy*. So, what is the so called **tidy data**? Following the definition in Wickham and Grolemund (2016), tidy data has the following three interrelated properties.

1. Each variable must have its own column.
2. Each observation must have its own row.
3. Each value must have its own cell.

These properties of *tidy data* enable us to conduct efficient data manipulation and visualization. Note that in practical applications, many collected data is *untidy*. Although *untidy* data could also be very useful in terms of reporting and visually more intuitive, you are recommended to *tidy* it before applying the tools we learned in this course.

### 6.1 Convert Between Names and Values

First, let's create an artificial dataset which contains the weights of a sheep and a pig for years 2019, 2020, and 2021.

```
library(tibble)
animal <- rep(c("sheep", "pig"), c(3, 3))
year <- rep(2019:2021, 2)
weight <- c(110, 120, 140, NA, 300, 800)
animal_tidy <- tibble(animal, year, weight)
animal_tidy
```

```
#> # A tibble: 6 x 3
#>   animal year weight
#>   <chr>  <int>  <dbl>
#> 1 sheep  2019    110
#> 2 sheep  2020    120
#> 3 sheep  2021    140
#> 4 pig    2019     NA
#> 5 pig    2020    300
#> 6 pig    2021    800
```

By checking the definition of **tidy data**, it is clear `animal_tidy` is indeed tidy. Let's make it untidy.

### 6.1.1 Convert Values into Column Names

In `animal_tidy`, each row contains the year when the weight measurement was taken. Suppose we want to convert the year value into column names. You can use the `pivot_wider()` function in **tidyr** package, another member of the **tidyverse** package. In `pivot_wider()`, you need to specify two arguments: `names_from` denotes which column in the original tibble contains the values of the new column names, `values_from` denotes which column in the original tibble contains the values for each cell in the new tibble. The reason why the function is called `pivot_wider()` is due to the fact that it will create a **wider** dataset than the original one, containing more columns.

```
library(tidyr)
animal_wide <- animal_tidy %>% pivot_wider(names_from = year,
animal_wide   #untidy animal: wide
```

```
#> # A tibble: 2 x 4
#>   animal `2019` `2020` `2021`
#>   <chr>   <dbl>  <dbl>  <dbl>
#> 1 sheep    110    120    140
#> 2 pig      NA    300    800
```

In `animal_wide`, we have the columns names 2019, 2020, and 2021 coming from the `year` variable, and the values 110, 120, 140, NA, 300, and 800 from the `weight` variable, both of which are in the original tibble `animal_tidy`. The `animal_wide` is clearly *untidy*, since neither the `weight` nor the `year` information is contained in a single column. Note that this data format is commonly encountered in practice.

As it is clear from the resulting tibble, the name `weight` is lost during the pivoting process, which is not desirable. Fortunately, you can add a prefix

“weight” to the column names via an argument `names_prefix` in the `pivot_wider()` function.

```
animal_wide_weight <- animal_tidy %>% pivot_wider(names_from = year,
  names_prefix = "weight",
  values_from = weight)
animal_wide_weight
```

```
#> # A tibble: 2 x 4
#>   animal weight2019 weight2020 weight2021
#>   <chr>      <dbl>      <dbl>      <dbl>
#> 1 sheep         110         120         140
#> 2 pig           NA         300         800
```

### 6.1.2 Convert Column Names into Values

Now, you will learn how to *tidy* `animal_wide` into a tidy data. To do this, you can use the `pivot_longer()` function to convert the columns names 2019, 2020, and 2021 into values of a variable, for example, `year`.

```
animal_wide %>%
  pivot_longer(cols = -1,
    names_to = "year",
    values_to = "weight")
```

In `pivot_longer()`, `cols` specifies the column names that you want to convert from, which accept the same format as that in `dplyr::select()` introduced in **Section ?**. `names_to` specifies the variable name you want to use for the column names. Finally, `values_to` specifies the variable name for holding the values in the selected columns. You can see that we have recovered the `animal_tidy` through the tidy process. To tidy `animal_wide_weight`, we can use the same function `pivot_longer()` along with the argument `names_prefix` as below.

```
animal_wide_weight %>%
  pivot_longer(cols = -1,
    names_to = "year",
    names_prefix = "weight",
    values_to = "weight")
```

In this regards, `pivot_wider()` and `pivot_longer()` can be viewed as *opposite* functions.



## Chapter 7

# Strings



## Chapter 8

# Statistics

In this chapter, you will dive into the world of statistics. As a language initially designed for statistical computing, R undoubtedly provides a wide range of functions related to all aspects of probability and statistics. You will start with functions related to normal distribution in Section 8.1.

### 8.1 Normal Distribution

First, let's review the definition of **normal distribution**, which is also called **Gaussian distribution**. If  $X \sim N(\mu, \sigma^2)$ , we say  $X$  is a random variable following a normal distribution with mean  $\mu$  and variance  $\sigma^2$ .

In the following table, we list the four useful functions for normal distribution, and they will be introduced in the subsequent four parts, respectively.

Code	Name	Section
<code>'dnorm(x, mean, sd)'</code>	probability density function	\@ref(pdf)
<code>'pnorm(q, mean, sd)'</code>	cumulative distribution function	\@ref(cdf)
<code>'qnorm(p, mean, sd)'</code>	quantile function	\@ref(qf)
<code>'rnorm(n, mean, sd)'</code>	random number generator	\@ref(rng)

#### 8.1.1 Probability density function (pdf)

To characterize the distribution of a continuous random variable, you can use the **probability density function (pdf)**. When  $X \sim N(\mu, \sigma^2)$ , its pdf is

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp \left[ -\frac{(x - \mu)^2}{2\sigma^2} \right].$$

In R, you can use `dnorm(x, mean, sd)` to calculate the pdf of normal distribution.

- The argument `x` represent the location(s) at which to compute the pdf.
- The arguments `mean` and `sd` represent the mean and standard deviation of the normal distribution, respectively.

For example, `dnorm(0, mean = 1, sd = 2)` computes the pdf at location 0 of  $N(1, 4)$ , normal distribution with mean 1 and variance 4.



Note that the argument `sd` is the standard deviation, which is the square root of the variance.

In particular, `dnorm()` without specifying the `mean` and `sd` arguments will compute the pdf of  $N(0, 1)$ , which is the standard normal distribution. Let's see examples of computing the pdf at one location for three different normal distributions.

```
dnorm(0, mean = 1, sd = 2)
dnorm(1, mean = -1, sd = 0.5)
dnorm(0) #standard normal
```

In addition to computing the pdf at one location for a single normal distribution, `dnorm` also accepts vectors with more than one elements in all three arguments. For example, you can use the following code to compute the three pdf values in the previous code block.

```
dnorm(c(0,1,0), mean = c(1, -1, 0), sd= c(2, 0.5, 1))
```

If you want to compute the pdf at the same location 0 for distributions  $N(1, 4)$ ,  $N(-1, 0.25)$ , and  $N(0, 1)$ , you can use the following code.

```
dnorm(0, mean = c(1, -1, 0), sd= c(2, 0.5, 1))
```

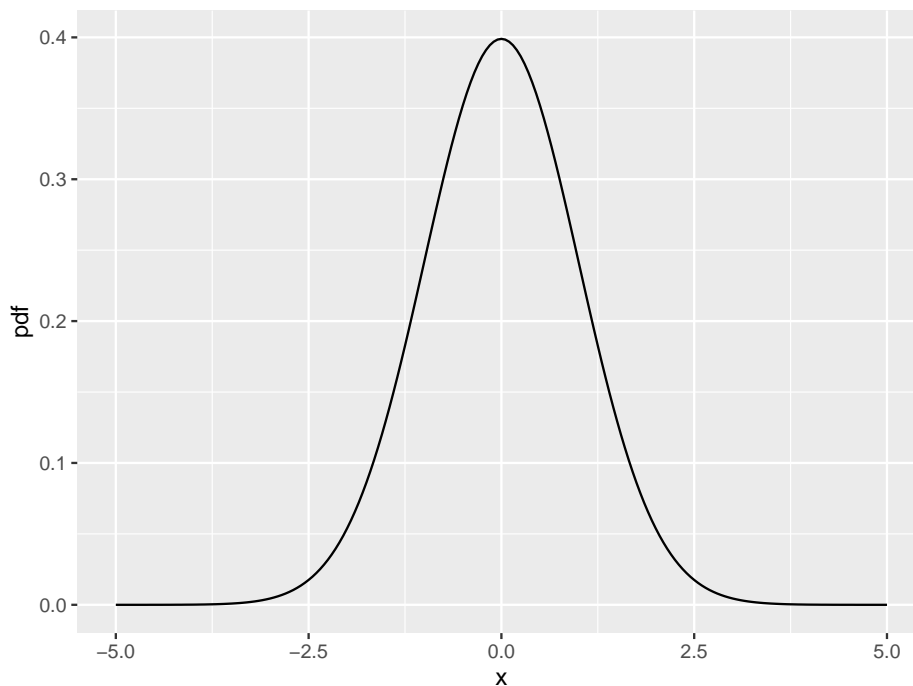
If you want to compute the pdf at three different locations (-3, 2, and 5) for distribution  $N(3, 4)$ , you can use the following code.



```
dnorm(c(-3, 2, 5), mean = 3, sd = 2)
```

To get a better understanding on the shape of the normal pdf, let's visualize the pdf of  $N(0, 1)$ . You first need to create a equal-spaced vector  $x$  from -5 to 5 with increment 0.1. Then, you can compute the pdf value for each element of  $x$  using `dnorm`. Finally, you can visualize the pdf using `geom_line`.

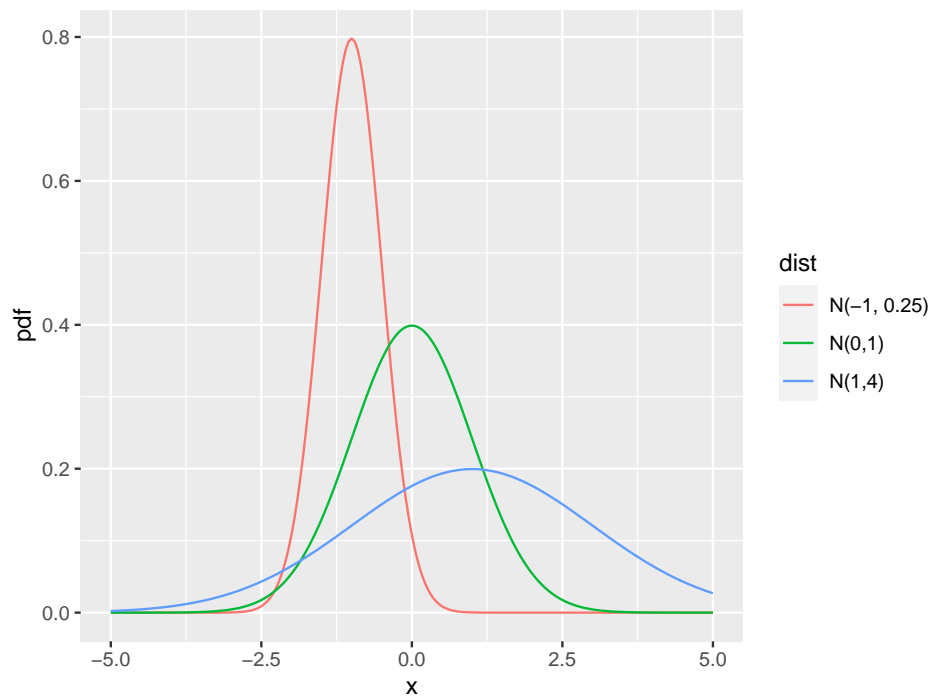
```
library(ggplot2)
x <- seq(from = -5, to = 5, by = 0.05)
norm_dat <- data.frame(x = x, pdf = dnorm(x))
ggplot(norm_dat) + geom_line(aes(x = x, y = pdf))
```



Next, you can take a step further to visualize three different normal distributions in the same plot,  $N(0, 1)$ ,  $N(1, 4)$ , and  $N(-1, 0.25)$ . You can use the same vector  $x$  and compute the three pdfs on each element of  $x$ . `geom_line` is still used with the variable `dist` mapped to the `color` aesthetic.

```
x <- seq(from = -5, to = 5, by = 0.05)
norm_dat_1 <- data.frame(dist = "N(0,1)", x = x, pdf = dnorm(x))
norm_dat_2 <- data.frame(dist = "N(1,4)", x = x, pdf = dnorm(x, mean = 1, sd = 2))
```

```
norm_dat_3 <- data.frame(dist = "N(-1, 0.25)", x = x, pdf = dnorm(x, mean = -1, sd = 0.25))
norm_dat <- rbind(norm_dat_1, norm_dat_2, norm_dat_3)
ggplot(norm_dat) + geom_line(aes(x = x, y = pdf, color = dist))
```



### 8.1.2 Cumulative distribution function (cdf)

In addition to pdf, you can compute the **cumulative distribution function (cdf)** of the normal distribution using the function `pnorm(q, mean, sd)`.

Generally speaking, the cdf of a random variable  $X$  is defined as

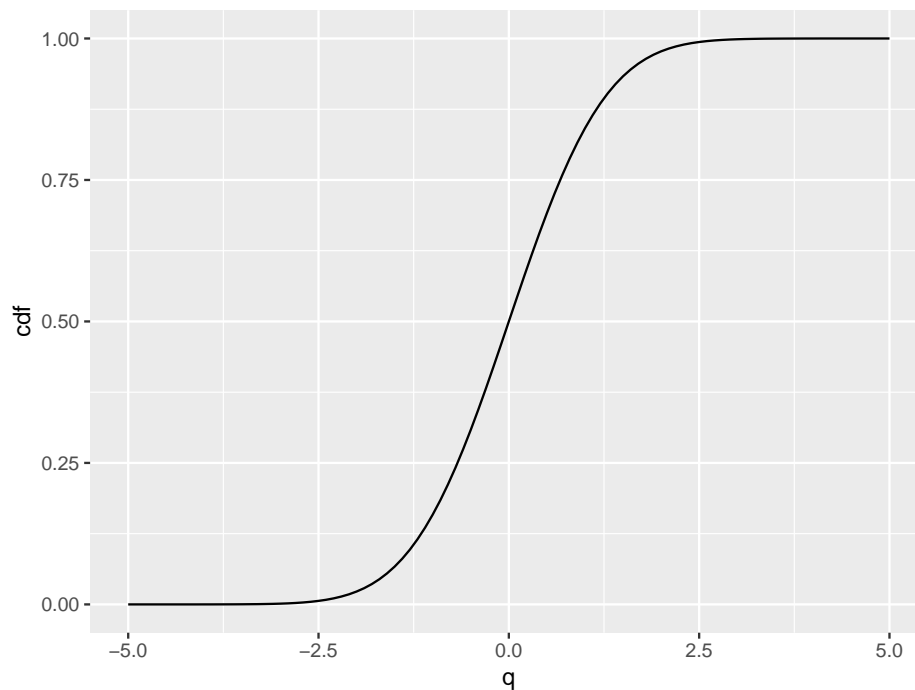
$$F(x) = P(X \leq x).$$

Similar to `dnorm()`, `pnorm()` also has two optional arguments, `mean` and `sd`, which represent the mean and standard deviation of the normal distribution, respectively. If you don't specify these two arguments, `pnorm()` will compute the cdf of  $N(0, 1)$ .

```
pnorm(0, mean = 1, sd = 2)
pnorm(0) # cdf at 0 of standard normal
```

You can also use `pnorm()` to visualize the cdf of the standard normal distribution.

```
q <- seq(from = -5, to = 5, by = 0.1)
norm_dat <- data.frame(q = q, cdf = pnorm(q))
ggplot(norm_dat) + geom_line(aes(x = q, y = cdf))
```



### 8.1.3 Quantile function

The third useful function related to distributions is the **quantile function**.

You can compute the quantile of the normal distribution using `qnorm(p, mean, sd)`. The quantile function is the inverse function of the cdf. In particular, the  $p$  quantile returns the value  $x$  such that

$$F(x) = P(X \leq x) = p$$

Let's verify `qnorm()` is indeed the inverse function of `pnorm()` using the following example.

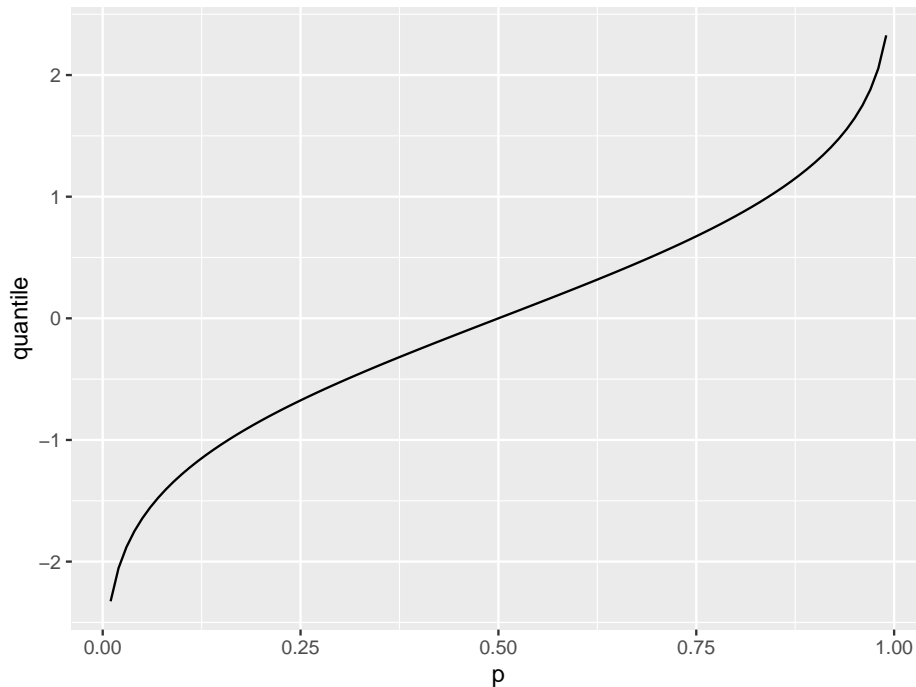
```
pnorm(qnorm(c(0.5,0.7)))
```

When  $p = 0.5$ , `qnorm()` gives us the median of the normal distribution. Let's see a few examples for computing the quantiles.

```
qnorm(0.5, mean = 1, sd = 2)  
qnorm(0.5)
```

You can also visualize the shape of the quantile function.

```
p <- seq(from = 0.01, to = 0.99, by = 0.01)  
norm_dat <- data.frame(p = p, quantile = qnorm(p))  
ggplot(norm_dat) + geom_line(aes(x = p, y = quantile))
```



#### 8.1.4 Random Number Generator

Lastly, to generate (pick up) random numbers from normal distributions, you can use the function `rnorm(n, mean, sd)`, with the argument `n` represents

the number of random numbers to generate, the arguments `mean` and `sd` are the mean and standard deviation of the normal distribution you would like to generate from, respectively. Again, if you only supply the argument `n`, you will be generating random numbers from  $N(0, 1)$ .

```
rmnorm(3, mean = 0, sd = 1) #generate 3 random numbers from N(0, 1)
rmnorm(3) #generate another 3 random numbers from N(0,1)
```

Since you are generating random numbers, the results may be different each time. In many applications, however, you may want to make the results reproducible. To do this, you can set random seed using the function `set.seed()` before generating the random numbers. Let's see the following example.

```
set.seed(724)
rmnorm(3)
```

Now, let's run it one more time.

```
set.seed(724)
rmnorm(3)
```

You can see that the exact 3 numbers are reproduced since you are using the same random seed 724. You can run these two lines of code on any machine and will get the exact same three random numbers.



Note that the code that involves randomness needs to be identical to reproduce the results. If you change the arguments in `rmnorm()`, you will get totally different results. See the following example.

```
set.seed(724)
rmnorm(1)
rmnorm(3)
```

By setting a different random seed, you will see different results as the following example.

```
set.seed(826)
rnorm(3)
```

Lastly, let's do a simple statistical exercise by checking the closeness of the *sample mean* and *sample standard deviation* to their population counterparts.

```
x <- rnorm(1e6, mean = 1, sd = 2)
mean(x) #sample mean
sd(x)   #sample standard deviation
```

### 8.1.5 Exercise

You can run the following code to do the exercise.

```
r02pro("8.1")
```

## 8.2 Other Distributions

In Section 8.1, we gave a detailed introduction to the four functions for a normal distribution, which is a popular *continuous* distribution. In particular, we now know that `dnorm()` produces the pdf of a normal distribution. In the case of *discrete* distributions, however, we would have **probability mass function (pmf)** instead of the pdf. Let's use the **binomial** distribution as a representative example of discrete distributions with the four functions as below.

Now, let's look at a few other commonly used distributions. For simplicity, let's just use the random number generator for each distribution in the following table.

As we can see from this table, all random number generator functions are formed by the letter **r** followed by the name of the distribution we would like to generate from. For the other three functions, we just need to change the initial letter **r**:

- to **d** for pdf (continuous distribution) or pmf (discrete distribution),
- to **p** for cdf,
- to **q** for quantile function.

Let's do some statistical exercise with those distributions.

### 8.2.1 Exercise

You can run the following code to do the exercise.

```
r02pro("8.2")
```

## 8.3 Random Permutation and Random Sampling

Now, you have covered how to work with distributions in R with the four useful functions for each distribution. In many applications, you may want to randomly permute or sample elements from a vector. Let's see how to do that.

The vector `x <- 6:10` will be used throughout this section.

### 8.3.1 Random Permutation

In statistics and machine learning, you usually need to do a random permutation of the data. For example, you can evaluate a model's performance by dividing the data randomly into two parts for training and validation, respectively.

For the vector `x <- 6:10`, you can use the function `sample()` to get a permutation for `x`.

```
x <- 6:10
set.seed(97)
sample(x) #a random permutation of x
```

To reproduce the random permutation, we can use the same seed.

```
set.seed(97)
sample(x) #reproduce the random permutation
```

### 8.3.2 Random Sampling without Replacement

Note that the vector `x` has 5 elements in total. To sample a few elements from `x`, you can again use the `sample()` function. For example, if you want to randomly sample two elements from `x`, you can use the following code

```
sample(x, size = 2)
```

Here, the `size` argument specifies the targeted number of elements. By default, the `sample` function takes a sample **without replacement**, i.e. the results sample has no duplicated elements. Because of this, if the `size` is larger than the length of the vector `x`, you will see an error message as follows.

```
sample(x, size = 6)
```

```
#> Error in sample.int(length(x), size, replace, prob): cannot take a sample larger than
```

In addition to using a vector in the first argument of `sample`, you can also use a positive integer (e.g., 10), which will be equivalent to `x = 1:10`. See the following code for an example.

```
sample(10, size = 4)  #sample 4 integers from 1 to 10.  
sample(1:10, size = 4) #sample 4 integers from 1 to 10.
```

### 8.3.3 Random Sampling with Replacement

Sometimes, you may want to get a sample with replacements. You will still be using the `sample` function, but setting the argument `replace = TRUE`. The following code samples 10 elements with replacement from `x`.

```
sample(x, size = 10, replace = TRUE)
```

As expected, you will see some duplicated elements in the output vector.

A very important application of random sample with replacement is **bootstrap**. A bootstrap sample is a sample of *the same size as the original data* with replacement. So, if you want to get a bootstrap sample from `x`, you will sample 5 elements with replacement from `x`.

```
sample(x, replace = TRUE) #a bootstrap sample
```



Note that, when the argument `size` is not provided, it will take the default value: the length of `x`.



### 8.3.4 Random Sampling with Unequal Probabilities

By default, the `sample()` function will draw each element with the same probability. In some cases, you may want to assign different probabilities for different elements.

To draw elements with different probabilities, the first method is to use the random number generator for Binomial distribution or Bernoulli distribution.

Let's say we want to randomly sample 100 elements from a Bernoulli distribution with success probability  $p = 0.2$ .

```
rbinom(100, size = 1, prob = 0.2)
```

In addition to using the `rbinom` function introduced in Section 8.2, you can use the `sample` function with the `prob` argument inside to achieve the same goal.

```
sample(c(0, 1), size = 100, replace = TRUE, prob = c(0.8, 0.2))
```

You will sample 100 elements with replacement from `c(0,1)` here, and the probability of drawing 0 is 0.8, the probability of drawing 1 is 0.2.

### 8.3.5 Exercise

You can run the following code to do the exercise.

```
r02pro("8.3")
```

## 8.4 Covariance and Correlation

In this section, we will dive further into statistics, this time talking about the relationship among variables. For two random variables  $X$  and  $Y$ , the covariance between them is defined as  $Cov(X, Y) = E[(X - E(X))(Y - E(Y))]$ .

```
set.seed(724)
n <- 1e5
x1 <- rnorm(n)
x2 <- x1 + rnorm(n)
x3 <- x2 + rnorm(n)
cor(x1, x2)
```

```
cor(x1, x3)
cor(x2, x3)
x <- data.frame(x1, x2, x3)
cor(x) #correlation matrix
cov(x) #covariance matrix
```

### 8.4.1 Exercise

You can run the following code to do the exercise.

```
r02pro("8.4")
```

## Chapter 9

# Writing Complicated Codes



## Chapter 10

### A Case Study: 24 Solver



# Bibliography

Wickham, H. (2010). A layered grammar of graphics. *Journal of Computational and Graphical Statistics*, 19(1):3–28.

Wickham, H. and Grolemund, G. (2016). *R for data science: import, tidy, transform, visualize, and model data.* " O'Reilly Media, Inc."